The ITK Software Guide
Second Edition
Updated for ITK version 2.4

Luis Ibáñez
Will Schroeder
Lydia Ng
Josh Cates
and the *Insight Software Consortium*

November 21, 2005

http://www.itk.org
Email: insight-users@itk.org

The purpose of computing is Insight, not numbers.

Richard Hamming

Abstract

The Insight Toolkit (ITK) is an open-source software toolkit for performing registration and segmentation. *Segmentation* is the process of identifying and classifying data found in a digitally sampled representation. Typically the sampled representation is an image acquired from such medical instrumentation as CT or MRI scanners. *Registration* is the task of aligning or developing correspondences between data. For example, in the medical environment, a CT scan may be aligned with a MRI scan in order to combine the information contained in both.

ITK is implemented in C++. It is cross-platform, using a build environment known as CMake to manage the compilation process in a platform-independent way. In addition, an automated wrapping process (Cable) generates interfaces between C++ and interpreted programming languages such as Tcl, Java, and Python. This enables developers to create software using a variety of programming languages. ITK's C++ implementation style is referred to as generic programming, which is to say that it uses templates so that the same code can be applied *generically* to any class or type that happens to support the operations used. Such C++ templating means that the code is highly efficient, and that many software problems are discovered at compile-time, rather than at run-time during program execution.

Because ITK is an open-source project, developers from around the world can use, debug, maintain, and extend the software. ITK uses a model of software development referred to as Extreme Programming. Extreme Programming collapses the usual software creation methodology into a simultaneous and iterative process of design-implement-test-release. The key features of Extreme Programming are communication and testing. Communication among the members of the ITK community is what helps manage the rapid evolution of the software. Testing is what keeps the software stable. In ITK, an extensive testing process (using a system known as Dart) is in place that measures the quality on a daily basis. The ITK Testing Dashboard is posted continuously, reflecting the quality of the software at any moment.

This book is a guide to using and developing with ITK. The sample code in the directory provides a companion to the material presented here. The most recent version of this document is available online at `http://www.itk.org/ItkSoftwareGuide.pdf`.

Contributors

The Insight Toolkit (ITK) has been created by the efforts of many talented individuals and prestigious organizations. It is also due in great part to the vision of the program established by Dr. Terry Yoo and Dr. Michael Ackerman at the National Library of Medicine.

This book lists a few of these contributors in the following paragraphs. Not all developers of ITK are credited here, so please visit the Web pages at http://www.itk.org/HTML/About.htm for the names of additional contributors, as well as checking the CVS source logs for code contributions.

The following is a brief description of the contributors to this software guide and their contributions.

Luis Ibáñez is principal author of this text. He assisted in the design and layout of the text, implemented the bulk of the LaTeX and CMake build process, and was responsible for the bulk of the content. He also developed most of the example code found in the `Insight/Examples` directory.

Will Schroeder helped design and establish the organization of this text and the `Insight/Examples` directory. He is principal content editor, and has authored several chapters.

Lydia Ng authored the description for the registration framework and its components, the section on the multiresolution framework, and the section on deformable registration methods. She also edited the section on the resampling image filter and the sections on various level set segmentation algorithms.

Joshua Cates authored the iterators chapter and the text and examples describing watershed segmentation. He also co-authored the level-set segmentation material.

Jisung Kim authored the chapter on the statistics framework.

Julien Jomier contributed the chapter on spatial objects and examples on model-based registration using spatial objects.

Karthik Krishnan reconfigured the process for automatically generating images from all the examples. Added a large number of new examples and updated the Filtering and Segmentation chapters for the second edition.

Stephen Aylward contributed material describing spatial objects and their application.

Tessa Sundaram contributed the section on deformable registration using the finite element method.

YinPeng Jin contributed the examples on hybrid segmentation methods.

Celina Imielinska authored the section describing the principles of hybrid segmentation methods.

Mark Foskey contributed the examples on the AutomaticTopologyMeshSource class.

Mathieu Malaterre contributed the entire section on the description and use of DICOM readers and writers based on the GDCM library. He also contributed an example on the use of the VTKImageIO class.

Gavin Baker contributed the section on how to write composite filters. Also known as minipipeline filters.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Part I

Introduction

CHAPTER

ONE

Welcome

Welcome to the *Insight Segmentation and Registration Toolkit (ITK) Software Guide*. This book has been updated for ITK 2.4 and later versions of the Insight Toolkit software.

ITK is an open-source, object-oriented software system for image processing, segmentation, and registration. Although it is large and complex, ITK is designed to be easy to use once you learn about its basic object-oriented and implementation methodology. The purpose of this Software Guide is to help you learn just this, plus to familiarize you with the important algorithms and data representations found throughout the toolkit. The material is taught using an extensive set of examples that we encourage you to compile and run while you read this guide.

ITK is a large system. As a result it is not possible to completely document all ITK objects and their methods in this text. Instead, this guide will introduce you to important system concepts and lead you up the learning curve as fast and efficiently as possible. Once you master the basics, we suggest that you take advantage of the many resources available including the Doxygen documentation pages (`http://www.itk.org/HTML/Documentation.htm`) and the community of ITK users (see Section 1.5 on page 10.)

The Insight Toolkit is an open-source software system. What this means is that the community of ITK users and developers has great impact on the evolution of the software. Users and developers can make significant contributions to ITK by providing bug reports, bug fixes, tests, new classes, and other feedback. Please feel free to contribute your ideas to the community (the ITK user mailing list is the preferred method; a developer's mailing list is also available).

1.1 Organization

This software guide is divided into three parts, each of which is further divided into several chapters. Part I is a general introduction to ITK, with—in the next chapter—a description of how to install the Insight Toolkit on your computer. This includes installing pre-compiled libraries and executables, and compiling the software from the source code. Part I also introduces basic system concepts such as an overview of the system architecture, and how to build applications in the C++, Tcl, and Python programming languages. Part II describes the system from the user point of view. Dozens of examples are used to illustrate important system features. Part III is for the ITK developer. Part III explains how to create your own classes, extend the system, and interface to various windowing and GUI systems.

1.2 How to Learn ITK

There are two broad categories of users of ITK. First are class developers, those who create classes in C++. The second, users, employ existing C++ classes to build applications. Class developers must be proficient in C++, and if they are extending or modifying ITK, they must also be familiar with ITK's internal structures and design (material covered in Part III). Users may or may not use C++, since the compiled C++ class library has been *wrapped* with the Tcl and Python interpreted languages. However, as a user you must understand the external interface to ITK classes and the relationships between them.

The key to learning how to use ITK is to become familiar with its palette of objects and the ways of combining them. If you are a new Insight Toolkit user, begin by installing the software. If you are a class developer, you'll want to install the source code and then compile it. Users may only need the precompiled binaries and executables. We recommend that you learn the system by studying the examples and then, if you are a class developer, study the source code. Start by reading Chapter 3, which provides an overview of some of the key concepts in the system, and then review the examples in Part II. You may also wish to compile and run the dozens of examples distributed with the source code found in the directory `Insight/Examples`. (Please see the file `Insight/Examples/README.txt` for a description of the examples contained in the various subdirectories.) There are also several hundred tests found in the source distribution in `Insight/Testing/Code`, most of which are minimally documented testing code. However, they may be useful to see how classes are used together in ITK, especially since they are designed to exercise as much of the functionality of each class as possible.

1.3 Software Organization

The following sections describe the directory contents, summarize the software functionality in each directory, and locate the documentation and data.

1.3.1 Obtaining the Software

There are three different ways to access the ITK source code (see Section 1.4 on page 5).

1. from periodic releases available on the ITK Web site,

2. from CD-ROM, and
3. from direct access to the CVS source code repository.

Official releases are available a few times a year and announced on the ITK Web pages and mailing lists. However, they may not provide the latest and greatest features of the toolkit. In general, the periodic releases and CD-ROM releases are the same, except that the CD release typically contains additional resources and data. CVS access provides immediate access to the latest toolkit additions, but on any given day the source code may not be stable as compared to the official releases—i.e., the code may not compile, it may crash, or it might even produce incorrect results.

This software guide assumes that you are working with the official ITK version 2.4 release (available on the ITK Web site). If you are a new user, we highly recommend that you use the released version of the software. It is stable, consistent, and better tested than the code available from the CVS repository. Later, as you gain experience with ITK, you may wish to work from the CVS repository. However, if you do so, please be aware of the ITK quality testing dashboard. The Insight Toolkit is heavily tested using the open-source DART regression testing system (`http://public.kitware.com/dashboard.php`). Before updating the CVS repository, make sure that the dashboard is *green* indicating stable code. If not green it is likely that your software update is unstable. (Learn more about the ITK quality dashboard in Section 14.2 on page 774.)

1.4 Downloading ITK

ITK can be downloaded without cost from the following web site:

`http://www.itk.org/HTML/Download.php`

In order to track the kind of applications for which ITK is being used, you will be asked to complete a form prior to downloading the software. The information you provide in this form will help developers to get a better idea of the interests and skills of the toolkit users. It also assists in future funding requests to sponsoring agencies.

Once you fill out this form you will have access to the download page where two options for obtaining the software will be found. (This page can be book marked to facilitate subsequent visits to the download site without having to complete any form again.) You can get the tarball of a stable release or you can get the development version through CVS. The release version is stable and dependable but may lack the latest features of the toolkit. The CVS version will have the latest additions but is inherently unstable and may contain components with work in progress. The following sections describe the details of each one of these two alternatives.

1.4.1 Downloading Packaged Releases

Please read the `GettingStarted.txt`[1] document first. It will give you an overview of the download and installation processes. Then choose the tarball that better fits your system. The options are `.zip` and `.tgz` files. The first type is better suited for MS-Windows while the second one is the preferred format for UNIX systems.

Once you unzip or untar the file a directory called `Insight` will be created in your disk and you will be ready for starting the configuration process described in Section 2.1.1 on page 14.

1.4.2 Downloading from CVS

The Concurrent Versions System (CVS) is a tool for software version control [27]. Generally only developers should be using CVS, so here we assume that you know what CVS is and how to use it. For more information about CVS please see Section 14.1 on page 773. (Note: please make sure that you access the software via CVS only when the ITK Quality Dashboard indicates that the code is stable. Learn more about the Quality Dashboard at 14.2 on page 774.)

Access ITK via CVS using the following commands (under UNIX and Cygwin):

```
cvs -d :pserver:anonymous@www.itk.org:/cvsroot/Insight login
(respond with password "insight")

cvs -d :pserver:anonymous@www.itk.org:/cvsroot/Insight co Insight
```

This will trigger the download of the software into a directory named `Insight`. Any time you want to update your version, it will be enough to change into this directory `Insight` and type:

```
cvs update -d -P
```

Once you obtain the software you are ready to configure and compile it (see Section 2.1.1 on page 14). First, however, we recommend that you join the mailing list and read the following sections describing the organization of the software.

1.4.3 Join the Mailing List

It is strongly recommended that you join the users mailing list. This is one of the primary resources for guidance and help regarding the use of the toolkit. You can subscribe to the users list online at

`http://www.itk.org/HTML/MailingLists.htm`

[1]http://www.itk.org/HTML/GettingStarted.txt

The insight-users mailing list is also the best mechanism for expressing your opinions about the toolkit and to let developers know about features that you find useful, desirable or even unnecessary. ITK developers are committed to creating a self-sustaining open-source ITK community. Feedback from users is fundamental to achieving this goal.

1.4.4 Directory Structure

To begin your ITK odyssey, you will first need to know something about ITK's software organization and directory structure. Even if you are installing pre-compiled binaries, it is helpful to know enough to navigate through the code base to find examples, code, and documentation.

ITK is organized into several different modules, or CVS checkouts. If you are using an official release or CD release, you will see three important modules: the `Insight`, `InsightDocuments` and `InsightApplications` modules. The source code, examples and applications are found in the `Insight` module; documentation, tutorials, and material related to the design and marketing of ITK are found in `InsightDocuments`; and fairly complex applications using ITK (and other systems such as VTK, Qt, and FLTK) are available from `InsightApplications`. Usually you will work with the `Insight` module unless you are a developer, are teaching a course, or are looking at the details of various design documents. The `InsightApplications` module should only be downloaded and compiled once the `Insight` module is functioning properly.

The `Insight` module contains the following subdirectories:

- `Insight/Auxiliary`—code that interfaces packages to ITK.
- `Insight/Code`—the heart of the software; the location of the majority of the source code.
- `Insight/Documentation`—a compact subset of documentation to get users started with ITK.
- `Insight/Examples`—a suite of simple, well-documented examples used by this guide and to illustrate important ITK concepts.
- `Insight/Testing`—a large number of small programs used to test ITK. These examples tend to be minimally documented but may be useful to demonstrate various system concepts. These tests are used by DART to produce the ITK Quality Dashboard (see Section 14.2 on page 774.)
- `Insight/Utilities`—supporting software for the ITK source code. For example, DART and Doxygen support, as well as libraries such as `png` and `zlib`.
- `Insight/Validation`—a series of validation case studies including the source code used to produce the results.
- `Insight/Wrapping`—support for the CABLE wrapping tool. CABLE is used by ITK to build interfaces between the C++ library and various interpreted languages (currently Tcl and Python are supported).

The source code directory structure—found in `Insight/Code`—is important to understand since other directory structures (such as the `Testing` and `Wrapping` directories) shadow the structure of the `Insight/Code` directory.

- `Insight/Code/Common`—core classes, macro definitions, typedefs, and other software constructs central to ITK.
- `Insight/Code/Numerics`—mathematical library and supporting classes. (Note: ITK's mathematical library is based on the VXL/VNL software package `http://vxl.sourceforge.net`.)
- `Insight/Code/BasicFilters`—basic image processing filters.
- `Insight/Code/IO`—classes that support the reading and writing of data.
- `Insight/Code/Algorithms`—the location of most segmentation and registration algorithms.
- `Insight/Code/SpatialObject`—classes that represent and organize data using spatial relationships (e.g., the leg bone is connected to the hip bone, etc.)
- `Insight/Code/Patented`—any patented algorithms are placed here. Using this code in commercial application requires a patent license.
- `Insight/Code/Local`—an empty directory used by developers and users to experiment with new code.

The `InsightDocuments` module contains the following subdirectories:

- `InsightDocuments/CourseWare`—material related to teaching ITK.
- `InsightDocuments/Developer`—historical documents covering the design and creation of ITK including progress reports and design documents.
- `InsightDocuments/Latex`—LaTeX styles to produce this work as well as other documents.
- `InsightDocuments/Marketing`—marketing flyers and literature used to succinctly describe ITK.
- `InsightDocuments/Papers`—papers related to the many algorithms, data representations, and software tools used in ITK.
- `InsightDocuments/SoftwareGuide`—LaTeX files used to create this guide. (Note that the code found in `Insight/Examples` is used in conjunction with these LaTeX files.)
- `InsightDocuments/Validation`—validation case studies using ITK.

- InsightDocuments/Web—the source HTML and other material used to produce the Web pages found at http://www.itk.org.

Similar to the Insight module, access to the InsightDocuments module is also available via CVS using the following commands (under UNIX and Cygwin):

```
cvs -d :pserver:anonymous@www.itk.org:/cvsroot/Insight co InsightDocuments
```

The InsightApplications module contains large, relatively complex examples of ITK usage. See the web pages at http://www.itk.org/HTML/Applications.htm for a description. Some of these applications require GUI toolkits such as Qt and FLTK or other packages such as VTK (*The Visualization Toolkit* http://www.vtk.org). Do not attempt to compile and build this module until you have successfully built the core Insight module.

Similar to the Insight and InsightDocuments module, access to the InsightApplications module is also available via CVS using the following commands (under UNIX and Cygwin):

```
cvs -d:pserver:anonymous@www.itk.org:/cvsroot/Insight \
  co InsightApplications
```

1.4.5 Documentation

Besides this text, there are other documentation resources that you should be aware of.

Doxygen Documentation. The Doxygen documentation is an essential resource when working with ITK. These extensive Web pages describe in detail every class and method in the system. The documentation also contains inheritance and collaboration diagrams, listing of event invocations, and data members. The documentation is heavily hyper-linked to other classes and to the source code. The Doxygen documentation is available on the companion CD, or on-line at http://www.itk.org. Make sure that you have the right documentation for your version of the source code.

Header Files. Each ITK class is implemented with a .h and .cxx/.txx file (.txx file for templated classes). All methods found in the .h header files are documented and provide a quick way to find documentation for a particular method. (Indeed, Doxygen uses the header documentation to produces its output.)

1.4.6 Data

The Insight Toolkit was designed to support the Visible Human Project and its associated data. This data is available from the National Library of Medicine at http://www.nlm.nih.gov/research/visible/visible_human.html.

Another source of data can be obtained from the ITK Web site at either of the following:

http://www.itk.org/HTML/Data.htm
ftp://public.kitware.com/pub/itk/Data/.

1.5 The Insight Community and Support

ITK was created from its inception as a collaborative, community effort. Research, teaching, and commercial uses of the toolkit are expected. If you would like to participate in the community, there are a number of possibilities.

- Users may actively report bugs, defects in the system API, and/or submit feature requests. Currently the best way to do this is through the ITK users mailing list.
- Developers may contribute classes or improve existing classes. If you are a developer, you may request permission to join the ITK developers mailing list. Please do so by sending email to will.schroeder "at" kitware.com. To become a developer you need to demonstrate both a level of competence as well as trustworthiness. You may wish to begin by submitting fixes to the ITK users mailing list.
- Research partnerships with members of the Insight Software Consortium are encouraged. Both NIH and NLM will likely provide limited funding over the next few years, and will encourage the use of ITK in proposed work.
- For those developing commercial applications with ITK, support and consulting are available from Kitware at http://www.kitware.com. Kitware also offers short ITK courses either at a site of your choice or periodically at Kitware.
- Educators may wish to use ITK in courses. Materials are being developed for this purpose, e.g., a one-day, conference course and semester-long graduate courses. Watch the ITK web pages or check in the `InsightDocuments/CourseWare` directory for more information.

1.6 A Brief History of ITK

In 1999 the US National Library of Medicine of the National Institutes of Health awarded six three-year contracts to develop an open-source registration and segmentation toolkit, that eventually came to be known as the Insight Toolkit (ITK) and formed the basis of the Insight Software Consortium. ITK's NIH/NLM Project Manager was Dr. Terry Yoo, who coordinated the six prime contractors composing the Insight consortium. These consortium members included three commercial partners—GE Corporate R&D, Kitware, Inc., and MathSoft (the company name is now Insightful)—and three academic partners—University of North Carolina (UNC), University of Tennessee (UT) (Ross Whitaker subsequently moved to University of Utah), and University of Pennsylvania (UPenn). The Principle Investigators for these partners were, respectively, Bill Lorensen at GE CRD, Will Schroeder at Kitware, Vikram Chalana at Insightful, Stephen Aylward with Luis Ibanez at UNC (Luis is now at Kitware), Ross Whitaker with Josh Cates at UT (both now at Utah), and Dimitri Metaxas at UPenn (now at Rutgers). In addition, several subcontractors rounded out the consortium including Peter Raitu at Brigham & Women's Hospital, Celina Imielinska and Pat Molholt at Columbia University, Jim Gee at UPenn's Grasp Lab, and George Stetten at the University of Pittsburgh.

In 2002 the first official public release of ITK was made available. In addition, the National Library of Medicine awarded thirteen contracts to several organizations to extend ITK's capabilities. NLM funding of Insight Toolkit development is continuing through 2003, with additional application and maintenance support anticipated beyond 2003. If you are interested in potential funding opportunities, we suggest that you contact Dr. Terry Yoo at the National Library of Medicine for more information.

CHAPTER

TWO

Installation

This section describes the process for installing ITK on your system. Keep in mind that ITK is a toolkit, and as such, once it is installed in your computer there will be no application to run. Rather, you will use ITK to build your own applications. What ITK does provide—besides the toolkit proper—is a large set of test files and examples that will introduce you to ITK concepts and will show you how to use ITK in your own projects.

Some of the examples distributed with ITK require third party libraries that you may have to download. For an initial installation of ITK you may want to ignore these extra libraries and just build the toolkit itself. In the past, a large fraction of the traffic on the insight-users mailing list has originated from difficulties in getting third party libraries compiled and installed rather than with actual problems building ITK.

ITK has been developed and tested across different combinations of operating systems, compilers, and hardware platforms including MS-Windows, Linux on Intel-compatible hardware, Solaris, IRIX, Mac OSX, and Cygwin. It is known to work with the following compilers:

- Visual Studio 6, .NET 2002, .NET 2003
- GCC 2.95.x, 2.96, 3.x
- SGI MIPSpro 7.3x
- Borland 5.5

Given the advanced usage of C++ features in the toolkit, some compilers may have difficulties processing the code. If you are currently using an outdated compiler this may be an excellent excuse for upgrading this old piece of software!

2.1 Configuring ITK

The challenge of supporting ITK across platforms has been solved through the use of CMake, a cross-platform, open-source build system. CMake is used to control the software compilation process using simple platform and compiler independent configuration files. CMake generates native makefiles and workspaces that can be used in the compiler environment of your choice. CMake is quite sophisticated—it supports complex environments requiring system configuration, compiler feature testing, and code generation.

CMake generates Makefiles under UNIX and Cygwin systems and generates Visual Studio workspaces under Windows (and appropriate build files for other compilers like Borland). The information used by CMake is provided by `CMakeLists.txt` files that are present in every directory of the ITK source tree. These files contain information that the user provides to CMake at configuration time. Typical information includes paths to utilities in the system and the selection of software options specified by the user.

2.1.1 Preparing CMake

CMake can be downloaded at no cost from

`http://www.cmake.org`

ITK requires at least CMake version 2.0. You can download binary versions for most of the popular platforms including Windows, Solaris, IRIX, HP, Mac and Linux. Alternatively you can download the source code and build CMake on your system. Follow the instructions in the CMake Web page for downloading and installing the software.

Running CMake initially requires that you provide two pieces of information: where the source code directory is located (ITK_SOURCE_DIR), and where the object code is to be produced (ITK_BINARY_DIR). These are referred to as the *source directory* and the *binary directory*. We recommend setting the binary directory to be different than the source directory (an *out-of-source* build), but ITK will still build if they are set to the same directory (an *in-source* build). On Unix, the binary directory is created by the user and CMake is invoked with the path to the source directory. For example:

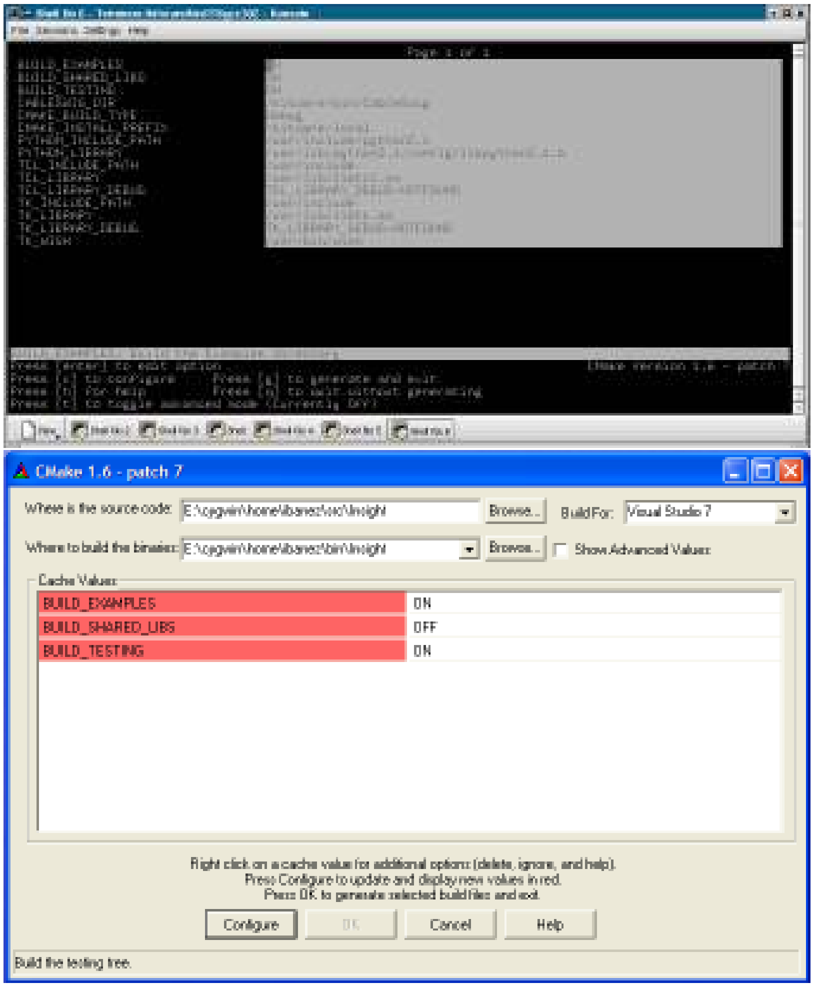
```
mkdir Insight-binary
cd Insight-binary
ccmake ../Insight
```

On Windows, the CMake GUI is used to specify the source and build directories (Figure 2.1).

CMake runs in an interactive mode in that you iteratively select options and configure according to these options. The iteration proceeds until no more options remain to be selected. At this point, a generation step produces the appropriate build files for your configuration.

This interactive configuration process can be better understood if you imagine that you are walking through a decision tree. Every option that you select introduces the possibility that new, dependent options may become relevant. These new options are presented by CMake at the top of the options list in its interface. Only when no new options appear after a configuration iteration can you be sure that the necessary decisions have all been made. At this point build files are generated for the current configuration.

2.1.2 Configuring ITK



```
                                        Page 1 of 1
 BUILD_EXAMPLES                         ON
 BUILD_SHARED_LIBS                      OFF
 BUILD_TESTING                          ON
 CABLESWIG_DIR                          [illegible]
 CMAKE_BUILD_TYPE                       Debug
 CMAKE_INSTALL_PREFIX                   [illegible]
 PYTHON_INCLUDE_PATH                    [illegible]
 PYTHON_LIBRARY                         [illegible]
 TCL_INCLUDE_PATH                       [illegible]
 TCL_LIBRARY                            [illegible]
 TCL_LIBRARY_DEBUG                      TCL_LIBRARY_DEBUG-NOTFOUND
 TK_INCLUDE_PATH                        [illegible]
 TK_LIBRARY                             [illegible]
 TK_LIBRARY_DEBUG                       TK_LIBRARY_DEBUG-NOTFOUND
 TK_WISH                                [illegible]

Press [enter] to edit option                                   CMake Version 1.6 - patch
Press [c] to configure     Press [g] to generate and exit
Press [h] for help         Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
```

CMake 1.6 - patch 7

Where is the source code: E:\cygwin\home\ibanez\src\Insight

Build For: Visual Studio 7

Where to build the binaries: E:\cygwin\home\ibanez\bin\Insight

Show Advanced Values

Cache Values

BUILD_EXAMPLES	ON
BUILD_SHARED_LIBS	OFF
BUILD_TESTING	ON

Right click on a cache value for additional options (delete, ignore, and help).
Press Configure to update and display new values in red.
Press OK to generate selected build files and exit.

Build the testing tree.

Figure 2.1: CMake interface. Top) `ccmake`, the UNIX version based on `curses`. Bottom) `CMakeSetup`, the MS-Windows version based on MFC.

Figure 2.1 shows the CMake interface for UNIX and MS-Windows. In order to speed up the build process you may want to disable the compilation of the testing and examples. This is done with the variables `BUILD_TESTING=OFF` and `BUILD_EXAMPLES=OFF`. The examples distributed with the toolkit are a helpful resource for learning how to use ITK components but are not essential for the use of the toolkit itself. The testing section includes a large number of small programs that exercise the capabilities of ITK classes. Due to the large number of tests, enabling the testing option will considerably increase the build time. It is not desirable to enable this option for a first build of the toolkit.

An additional resource is available in the `InsightApplications` module, which contains multiple applications incorporating GUIs and different levels of visualization. However, due to the large number of applications and the fact that some of them rely on third party libraries, building this module should be postponed until you are familiar with the basic structure of the toolkit and the building process.

Begin running CMake by using ccmake on Unix, and CMakeSetup on Windows. Remember to run ccmake from the binary directory on Unix. On Windows, specify the source and binary directories in the GUI, then begin to set the build variables in the GUI as necessary. Most variables should have default values that are sensible. Each time you change a set of variables in CMake, it is necessary to proceed to another configuration step. In the Windows version this is done by clicking on the "Configure" button. In the UNIX version this is done in an interface using the curses library, where you can configure by hitting the "c" key.

When no new options appear in CMake, you can proceed to generate Makefiles or Visual Studio projects (or appropriate build file(s) depending on your compiler). This is done in Windows by clicking on the "Ok" button. In the UNIX version this is done by hitting the "g" key. After the generation process CMake will quit silently. To initiate the build process on UNIX, simply type `make` in the binary directory. Under Windows, load the workspace named `ITK.dsw` (if using MSDEV) or `ITK.sln` (if using the .NET compiler) from the binary directory you specified in the CMake GUI.

The build process will typically take anywhere from 15 to 30 minutes depending on the performance of your system. If you decide to enable testing as part of the normal build process, about 600 small test programs will be compiled. This will verify that the basic components of ITK have been correctly built on your system.

2.2 Getting Started With ITK

The simplest way to create a new project with ITK is to create a new directory somewhere in your disk and create two files in it. The first one is a `CMakeLists.txt` file that will be used by CMake to generate a Makefile (if you are using UNIX) or a Visual Studio workspace (if you are using MS-Windows). The second file is an actual C++ program that will exercise some of the large number of classes available in ITK. The details of these files are described in the following section.

Once both files are in your directory you can run CMake in order to configure your project. Under UNIX, you can cd to your newly created directory and type "`ccmake .`". Note the "." in the command line for indicating that the `CMakeLists.txt` file is in the current directory.

The curses interface will require you to provide the directory where ITK was built. This is the same path that you indicated for the ITK_BINARY_DIR variable at the time of configuring ITK. Under Windows you can run CMakeSetup and provide your newly created directory as being both the source directory and the binary directory for your new project (i.e., an in-source build). Then CMake will require you to provide the path to the binary directory where ITK was built. The ITK binary directory will contain a file named ITKConfig.cmake generated during the configuration process at the time ITK was built. From this file, CMake will recover all the information required to configure your new ITK project.

2.2.1 Hello World !

Here is the content of the two files to write in your new project. These two files can be found in the Insight/Examples/Installation directory. The CMakeLists.txt file contains the following lines:

```
PROJECT(HelloWorld)

FIND_PACKAGE(ITK)
IF(ITK_FOUND)
  INCLUDE(${ITK_USE_FILE})
ELSE(ITK_FOUND)
  MESSAGE(FATAL_ERROR
          "ITK not found. Please set ITK_DIR.")
ENDIF(ITK_FOUND)

ADD_EXECUTABLE(HelloWorld HelloWorld.cxx )

TARGET_LINK_LIBRARIES(HelloWorld ITKCommon)
```

The first line defines the name of your project as it appears in Visual Studio (it will have no effect under UNIX). The second line loads a CMake file with a predefined strategy for finding ITK [1]. If the strategy for finding ITK fails, CMake will prompt you for the directory where ITK is installed in your system. In that case you will write this information in the ITK_BINARY_DIR variable. The line INCLUDE(${USE_ITK_FILE}) loads the UseITK.cmake file to set all the configuration information from ITK. The line ADD_EXECUTABLE defines as its first argument the name of the executable that will be produced as result of this project. The remaining arguments of ADD_EXECUTABLE are the names of the source files to be compiled and linked. Finally, the TARGET_LINK_LIBRARIES line specifies which ITK libraries will be linked against this project.

The source code for this section can be found in the file
Examples/Installation/HelloWorld.cxx.

The following code is an implementation of a small Insight program. It tests including header files and linking with ITK libraries.

[1] Similar files are provided in CMake for other commonly used libraries, all of them named Find*.cmake

```
#include "itkImage.h"
#include <iostream>

int main()
{
  typedef itk::Image< unsigned short, 3 > ImageType;

  ImageType::Pointer image = ImageType::New();

  std::cout << "ITK Hello World !" << std::endl;

  return 0;
}
```

This code instantiates a $3D$ image[2] whose pixels are represented with type `unsigned short`. The image is then constructed and assigned to a `itk::SmartPointer`. Although later in the text we will discuss `SmartPointer`'s in detail, for now think of it as a handle on an instance of an object (see section 3.2.4 for more information). The `itk::Image` class will be described in Section 4.1.

At this point you have successfully installed and compiled ITK, and created your first simple program. If you have difficulties, please join the insight-users mailing list (Section 1.4.3 on page 6) and pose questions there.

[2] Also known as a *volume*.

CHAPTER

THREE

System Overview

The purpose of this chapter is to provide you with an overview of the *Insight Toolkit* system. We recommend that you read this chapter to gain an appreciation for the breadth and area of application of ITK.

3.1 System Organization

The Insight Toolkit consists of several subsystems. A brief description of these subsystems follows. Later sections in this chapter—and in some cases additional chapters—cover these concepts in more detail. (Note: in the previous chapter two other modules—`InsightDocumentation` and `InsightApplications` were briefly described.)

Essential System Concepts. Like any software system, ITK is built around some core design concepts. Some of the more important concepts include generic programming, smart pointers for memory management, object factories for adaptable object instantiation, event management using the command/observer design paradigm, and multithreading support.

Numerics ITK uses VXL's VNL numerics libraries. These are easy-to-use C++ wrappers around the Netlib Fortran numerical analysis routines (`http://www.netlib.org`).

Data Representation and Access. Two principal classes are used to represent data: the `itk::Image` and `itk::Mesh` classes. In addition, various types of iterators and containers are used to hold and traverse the data. Other important but less popular classes are also used to represent data such as histograms and BLOX images.

Data Processing Pipeline. The data representation classes (known as *data objects*) are operated on by *filters* that in turn may be organized into data flow *pipelines*. These pipelines maintain state and therefore execute only when necessary. They also support multithreading, and are streaming capable (i.e., can operate on pieces of data to minimize the memory footprint).

IO Framework. Associated with the data processing pipeline are *sources*, filters that initiate the pipeline, and *mappers*, filters that terminate the pipeline. The standard examples of sources and mappers are *readers* and *writers* respectively. Readers input data (typically from a file), and writers output data from the pipeline.

Spatial Objects. Geometric shapes are represented in ITK using the spatial object hierarchy. These classes are intended to support modeling of anatomical structures. Using a common basic interface, the spatial objects are capable of representing regions of space in a variety of different ways. For example: mesh structures, image masks, and implicit equations may be used as the underlying representation scheme. Spatial objects are a natural data structure for communicating the results of segmentation methods and for introducing anatomical priors in both segmentation and registration methods.

Registration Framework. A flexible framework for registration supports four different types of registration: image registration, multiresolution registration, PDE-based registration, and FEM (finite element method) registration.

FEM Framework. ITK includes a subsystem for solving general FEM problems, in particular non-rigid registration. The FEM package includes mesh definition (nodes and elements), loads, and boundary conditions.

Level Set Framework. The level set framework is a set of classes for creating filters to solve partial differential equations on images using an iterative, finite difference update scheme. The level set framework consists of finite difference solvers including a sparse level set solver, a generic level set segmentation filter, and several specific subclasses including threshold, Canny, and Laplacian based methods.

Wrapping. ITK uses a unique, powerful system for producing interfaces (i.e., “wrappers”) to interpreted languages such as Tcl and Python. The GCC_XML tool is used to produce an XML description of arbitrarily complex C++ code; CSWIG is then used to transform the XML description into wrappers using the SWIG package.

Auxiliary / Utilities Several auxiliary subsystems are available to supplement other classes in the system. For example, calculators are classes that perform specialized operations in support of filters (e.g., MeanCalculator computes the mean of a sample). Other utilities include a partial DICOM parser, MetaIO file support, png, zlib, FLTK / Qt image viewers, and interfaces to the Visualization Toolkit (VTK) system.

3.2 Essential System Concepts

This section describes some of the core concepts and implementation features found in ITK.

3.2.1 Generic Programming

Generic programming is a method of organizing libraries consisting of generic—or reusable—software components [58]. The idea is to make software that is capable of "plugging together" in an efficient, adaptable manner. The essential ideas of generic programming are *containers* to hold data, *iterators* to access the data, and *generic algorithms* that use containers and iterators to create efficient, fundamental algorithms such as sorting. Generic programming is implemented in C++ with the *template* programming mechanism and the use of the STL Standard Template Library [6].

C++ templating is a programming technique allowing users to write software in terms of one or more unknown types `T`. To create executable code, the user of the software must specify all types `T` (known as *template instantiation*) and successfully process the code with the compiler. The `T` may be a native type such as `float` or `int`, or `T` may be a user-defined type (e.g., `class`). At compile-time, the compiler makes sure that the templated types are compatible with the instantiated code and that the types are supported by the necessary methods and operators.

ITK uses the techniques of generic programming in its implementation. The advantage of this approach is that an almost unlimited variety of data types are supported simply by defining the appropriate template types. For example, in ITK it is possible to create images consisting of almost any type of pixel. In addition, the type resolution is performed at compile-time, so the compiler can optimize the code to deliver maximal performance. The disadvantage of generic programming is that many compilers still do not support these advanced concepts and cannot compile ITK. And even if they do, they may produce completely undecipherable error messages due to even the simplest syntax errors. If you are not familiar with templated code and generic programming, we recommend the two books cited above.

3.2.2 Include Files and Class Definitions

In ITK classes are defined by a maximum of two files: a header `.h` file and an implementation file—`.cxx` if a non-templated class, and a `.txx` if a templated class. The header files contain class declarations and formatted comments that are used by the Doxygen documentation system to automatically produce HTML manual pages.

In addition to class headers, there are a few other important header files.

`itkMacro.h` is found in the `Code/Common` directory and defines standard system-wide macros (such as `Set/Get`, constants, and other parameters).

`itkNumericTraits.h` is found in the `Code/Common` directory and defines numeric characteristics for native types such as its maximum and minimum possible values.

`itkWin32Header.h` is found in the `Code/Common` and is used to define operating system parameters to control the compilation process.

3.2.3 Object Factories

Most classes in ITK are instantiated through an *object factory* mechanism. That is, rather than using the standard C++ class constructor and destructor, instances of an ITK class are created with the static class `New()` method. In fact, the constructor and destructor are `protected:` so it is generally not possible to construct an ITK instance on the heap. (Note: this behavior pertains to classes that are derived from `itk::LightObject`. In some cases the need for speed or reduced memory footprint dictates that a class not be derived from LightObject and in this case instances may be created on the heap. An example of such a class is `itk::EventObject`.)

The object factory enables users to control run-time instantiation of classes by registering one or more factories with `itk::ObjectFactoryBase`. These registered factories support the method `CreateInstance(classname)` which takes as input the name of a class to create. The factory can choose to create the class based on a number of factors including the computer system configuration and environment variables. For example, in a particular application an ITK user may wish to deploy their own class implemented using specialized image processing hardware (i.e., to realize a performance gain). By using the object factory mechanism, it is possible at run-time to replace the creation of a particular ITK filter with such a custom class. (Of course, the class must provide the exact same API as the one it is replacing.) To do this, the user compiles her class (using the same compiler, build options, etc.) and inserts the object code into a shared library or DLL. The library is then placed in a directory referred to by the `ITK_AUTOLOAD_PATH` environment variable. On instantiation, the object factory will locate the library, determine that it can create a class of a particular name with the factory, and use the factory to create the instance. (Note: if the `CreateInstance()` method cannot find a factory that can create the named class, then the instantiation of the class falls back to the usual constructor.)

In practice object factories are used mainly (and generally transparently) by the ITK input/output (IO) classes. For most users the greatest impact is on the use of the `New()` method to create a class. Generally the `New()` method is declared and implemented via the macro `itkNewMacro()` found in `Common/itkMacro.h`.

3.2.4 Smart Pointers and Memory Management

By their nature object-oriented systems represent and operate on data through a variety of object types, or classes. When a particular class is instantiated to produce an instance of that class, memory allocation occurs so that the instance can store data attribute values and method pointers (i.e., the vtable). This object may then be referenced by other classes or data structures during normal operation of the program. Typically during program execution all references to the instance may disappear at which point the instance must be deleted to recover memory resources. Knowing when to delete an instance, however, is difficult. Deleting the instance too soon results in program crashes; deleting it too late and memory leaks (or excessive memory consumption) will occur. This process of allocating and releasing memory is known as memory management.

In ITK, memory management is implemented through reference counting. This compares to another popular approach—garbage collection—used by many systems including Java. In reference counting, a count of the number of references to each instance is kept. When the reference goes to zero, the object destroys itself. In garbage collection, a background process sweeps the system identifying instances no longer referenced in the system and deletes them. The problem with garbage collection is that the actual point in time at which memory is deleted is variable. This is unacceptable when an object size may be gigantic (think of a large 3D volume gigabytes in size). Reference counting deletes memory immediately (once all references to an object disappear).

Reference counting is implemented through a `Register()`/`Delete()` member function interface. All instances of an ITK object have a `Register()` method invoked on them by any other object that references an them. The `Register()` method increments the instances' reference count. When the reference to the instance disappears, a `Delete()` method is invoked on the instance that decrements the reference count—this is equivalent to an `UnRegister()` method. When the reference count returns to zero, the instance is destroyed.

This protocol is greatly simplified by using a helper class called a `itk::SmartPointer`. The smart pointer acts like a regular pointer (e.g. supports operators `->` and `*`) but automagically performs a `Register()` when referring to an instance, and an `UnRegister()` when it no longer points to the instance. Unlike most other instances in ITK, SmartPointers can be allocated on the program stack, and are automatically deleted when the scope that the SmartPointer was created is closed. As a result, you should *rarely if ever call Register() or Delete()* in ITK. For example:

```
MyRegistrationFunction()
  { <----- Start of scope

  // here an interpolator is created and associated to the
  // SmartPointer "interp".
  InterpolatorType::Pointer interp = InterpolatorType::New();

  } <------ End of scope
```

In this example, reference counted objects are created (with the `New()` method) with a reference count of one. Assignment to the SmartPointer `interp` does not change the reference count. At the end of scope, `interp` is destroyed, the reference count of the actual interpolator object (referred to by `interp`) is decremented, and if it reaches zero, then the interpolator is also destroyed.

Note that in ITK SmartPointers are always used to refer to instances of classes derived from `itk::LightObject`. Method invocations and function calls often return "real" pointers to instances, but they are immediately assigned to a SmartPointer. Raw pointers are used for non-LightObject classes when the need for speed and/or memory demands a smaller, faster class.

3.2.5 Error Handling and Exceptions

In general, ITK uses exception handling to manage errors during program execution. Exception handling is a standard part of the C++ language and generally takes the form as illustrated below:

```
try
  {
  //...try executing some code here...
  }
catch ( itk::ExceptionObject exp )
  {
  //...if an exception is thrown catch it here
  }
```

where a particular class may throw an exceptions as demonstrated below (this code snippet is taken from `itk::ByteSwapper`:

```
switch ( sizeof(T) )
  {
  //non-error cases go here followed by error case
  default:
    ByteSwapperError e(__FILE__, __LINE__);
    e.SetLocation("SwapBE");
    e.SetDescription("Cannot swap number of bytes requested");
    throw e;
  }
```

Note that `itk::ByteSwapperError` is a subclass of `itk::ExceptionObject`. (In fact in ITK all exceptions should be derived from ExceptionObject.) In this example a special constructor and C++ preprocessor variables `__FILE__` and `__LINE__` are used to instantiate the exception object and provide additional information to the user. You can choose to catch a particular exception and hence a specific ITK error, or you can trap *any* ITK exception by catching ExceptionObject.

3.2.6 Event Handling

Event handling in ITK is implemented using the Subject/Observer design pattern [28] (sometimes referred to as the Command/Observer design pattern). In this approach, objects indicate that they are watching for a particular event—invoked by a particular instance–by registering with the instance that they are watching. For example, filters in ITK periodically invoke the `itk::ProgressEvent`. Objects that have registered their interest in this event are notified when the event occurs. The notification occurs via an invocation of a command (i.e., function callback, method invocation, etc.) that is specified during the registration process. (Note that events in ITK are subclasses of EventObject; look in `itkEventObject.h` to determine which events are available.)

To recap via example: various objects in ITK will invoke specific events as they execute (from ProcessObject):

```
this->InvokeEvent( ProgressEvent() );
```

To watch for such an event, registration is required that associates a command (e.g., callback function) with the event: `Object::AddObserver()` method:

```
unsigned long progressTag =
  filter->AddObserver(ProgressEvent(), itk::Command*);
```

When the event occurs, all registered observers are notified via invocation of the associated `Command::Execute()` method. Note that several subclasses of Command are available supporting const and non-const member functions as well as C-style functions. (Look in `Common/Command.h` to find pre-defined subclasses of Command. If nothing suitable is found, derivation is another possibility.)

3.2.7 Multi-Threading

Multithreading is handled in ITK through a high-level design abstraction. This approach provides portable multithreading and hides the complexity of differing thread implementations on the many systems supported by ITK. For example, the class `itk::MultiThreader` provides support for multithreaded execution using `sproc()` on an SGI, or `pthread_create` on any platform supporting POSIX threads.

Multithreading is typically employed by an algorithm during its execution phase. MultiThreader can be used to execute a single method on multiple threads, or to specify a method per thread. For example, in the class `itk::ImageSource` (a superclass for most image processing filters) the `GenerateData()` method uses the following methods:

```
multiThreader->SetNumberOfThreads(int);
multiThreader->SetSingleMethod(ThreadFunctionType, void* data);
multiThreader->SingleMethodExecute();
```

In this example each thread invokes the same method. The multithreaded filter takes care to divide the image into different regions that do not overlap for write operations.

The general philosophy in ITK regarding thread safety is that accessing different instances of a class (and its methods) is a thread-safe operation. Invoking methods on the same instance in different threads is to be avoided.

3.3 Numerics

ITK uses the VNL numerics library to provide resources for numerical programming combining the ease of use of packages like Mathematica and Matlab with the speed of C and the elegance of C++. It provides a C++ interface to the high-quality Fortran routines made available in the public domain by numerical analysis researchers. ITK extends the functionality of VNL by including interface classes between VNL and ITK proper.

The VNL numerics library includes classes for

Matrices and vectors. Standard matrix and vector support and operations on these types.

Specialized matrix and vector classes. Several special matrix and vector class with special numerical properties are available. Class `vnl_diagonal_matrix` provides a fast and convenient diagonal matrix, while fixed size matrices and vectors allow ”fast-as-C” computations (see `vnl_matrix_fixed<T,n,m>` and example subclasses `vnl_double_3x3` and `vnl_double_3`).

Matrix decompositions. Classes `vnl_svd<T>`, `vnl_symmetric_eigensystem<T>`, and `vnl_generalized_eigensystem`.

Real polynomials. Class `vnl_real_polynomial` stores the coefficients of a real polynomial, and provides methods of evaluation of the polynomial at any x, while class `vnl_rpoly_roots` provides a root finder.

Optimization. Classes `vnl_levenberg_marquardt`, `vnl_amoeba`, `vnl_conjugate_gradient`, `vnl_lbfgs` allow optimization of user-supplied functions either with or without user-supplied derivatives.

Standardized functions and constants. Class `vnl_math` defines constants (pi, e, eps...) and simple functions (sqr, abs, rnd...). Class `numeric_limits` is from the ISO standard document, and provides a way to access basic limits of a type. For example `numeric_limits<short>::max()` returns the maximum value of a short.

Most VNL routines are implemented as wrappers around the high-quality Fortran routines that have been developed by the numerical analysis community over the last forty years and placed in the public domain. The central repository for these programs is the ”netlib” server `http://www.netlib.org/`. The National Institute of Standards and Technology (NIST) provides an excellent search interface to this repository in its *Guide to Available Mathematical Software (GAMS)* at `http://gams.nist.gov`, both as a decision tree and a text search.

ITK also provides additional numerics functionality. A suite of optimizers, that use VNL under the hood and integrate with the registration framework are available. A large collection of statistics functions—not available from VNL—are also provided in the `Insight/Numerics/Statistics` directory. In addition, a complete finite element (FEM) package is available, primarily to support the deformable registration in ITK.

3.4 Data Representation

There are two principle types of data represented in ITK: images and meshes. This functionality is implemented in the classes Image and Mesh, both of which are subclasses of `itk::DataObject`. In ITK, data objects are classes that are meant to be passed around the system and may participate in data flow pipelines (see Section 3.5 on page 28 for more information).

`itk::Image` represents an *n*-dimensional, regular sampling of data. The sampling direction is parallel to each of the coordinate axes, and the origin of the sampling, inter-pixel spacing, and the number of samples in each direction (i.e., image dimension) can be specified. The sample, or pixel, type in ITK is arbitrary—a template parameter `TPixel` specifies the type upon template instantiation. (The dimensionality of the image must also be specified when the image class is instantiated.) The key is that the pixel type must support certain operations (for example, addition or difference) if the code is to compile in all cases (for example, to be processed by a particular filter that uses these operations). In practice the ITK user will use a C++ simple type (e.g., `int`, `float`) or a pre-defined pixel type and will rarely create a new type of pixel class.

One of the important ITK concepts regarding images is that rectangular, continuous pieces of the image are known as *regions*. Regions are used to specify which part of an image to process, for example in multithreading, or which part to hold in memory. In ITK there are three common types of regions:

1. `LargestPossibleRegion`—the image in its entirety.
2. `BufferedRegion`—the portion of the image retained in memory.
3. `RequestedRegion`—the portion of the region requested by a filter or other class when operating on the image.

The Mesh class represents an *n*-dimensional, unstructured grid. The topology of the mesh is represented by a set of *cells* defined by a type and connectivity list; the connectivity list in turn refers to points. The geometry of the mesh is defined by the *n*-dimensional points in combination with associated cell interpolation functions. `Mesh` is designed as an adaptive representational structure that changes depending on the operations performed on it. At a minimum, points and cells are required in order to represent a mesh; but it is possible to add additional topological information. For example, links from the points to the cells that use each point can be added; this provides implicit neighborhood information assuming the implied topology is the desired one. It is also possible to specify boundary cells explicitly, to indicate different connectivity from the implied neighborhood relationships, or to store information on the boundaries of cells.

The mesh is defined in terms of three template parameters: 1) a pixel type associated with the points, cells, and cell boundaries; 2) the dimension of the points (which in turn limits the maximum dimension of the cells); and 3) a "mesh traits" template parameter that specifies the types of the containers and identifiers used to access the points, cells, and/or boundaries. By using the mesh traits carefully, it is possible to create meshes better suited for editing, or those better suited for "read-only" operations, allowing a trade-off between representation flexibility, memory, and speed.

Mesh is a subclass of itk::PointSet. The PointSet class can be used to represent point clouds or randomly distributed landmarks, etc. The PointSet class has no associated topology.

3.5 Data Processing Pipeline

While data objects (e.g., images and meshes) are used to represent data, *process objects* are classes that operate on data objects and may produce new data objects. Process objects are classed as *sources*, *filter objects*, or *mappers*. Sources (such as readers) produce data, filter objects take in data and process it to produce new data, and mappers accept data for output either to a file or some other system. Sometimes the term *filter* is used broadly to refer to all three types.

The data processing pipeline ties together data objects (e.g., images and meshes) and process objects. The pipeline supports an automatic updating mechanism that causes a filter to execute if and only if its input or its internal state changes. Further, the data pipeline supports *streaming*, the ability to automatically break data into smaller pieces, process the pieces one by one, and reassemble the processed data into a final result.

Typically data objects and process objects are connected together using the `SetInput()` and `GetOutput()` methods as follows:

```
typedef itk::Image<float,2> FloatImage2DType;

itk::RandomImageSource<FloatImage2DType>::Pointer random;
random = itk::RandomImageSource<FloatImage2DType>::New();
random->SetMin(0.0);
random->SetMax(1.0);

itk::ShrinkImageFilter<FloatImage2DType,FloatImage2DType>::Pointer shrink;
shrink = itk::ShrinkImageFilter<FloatImage2DType,FloatImage2DType>::New();
shrink->SetInput(random->GetOutput());
shrink->SetShrinkFactors(2);

itk::ImageFileWriter::Pointer<FloatImage2DType> writer;
writer = itk::ImageFileWriter::Pointer<FloatImage2DType>::New();
writer->SetInput (shrink->GetOutput());
writer->SetFileName( ''test.raw'' );
writer->Update();
```

In this example the source object itk::RandomImageSource is connected to the itk::ShrinkImageFilter, and the shrink filter is connected to the mapper itk::ImageFileWriter. When the Update() method is invoked on the writer, the data processing pipeline causes each of these filters in order, culminating in writing the final data to a file on disk.

3.6 Spatial Objects

The ITK spatial object framework supports the philosophy that the task of image segmentation and registration is actually the task of object processing. The image is but one medium for representing objects of interest, and much processing and data analysis can and should occur at the object level and not based on the medium used to represent the object.

ITK spatial objects provide a common interface for accessing the physical location and geometric properties of and the relationship between objects in a scene that is independent of the form used to represent those objects. That is, the internal representation maintained by a spatial object may be a list of points internal to an object, the surface mesh of the object, a continuous or parametric representation of the object's internal points or surfaces, and so forth.

The capabilities provided by the spatial objects framework supports their use in object segmentation, registration, surface/volume rendering, and other display and analysis functions. The spatial object framework extends the concept of a "scene graph" that is common to computer rendering packages so as to support these new functions. With the spatial objects framework you can:

1. Specify a spatial object's parent and children objects. In this way, a liver may contain vessels and those vessels can be organized in a tree structure.
2. Query if a physical point is inside an object or (optionally) any of its children.
3. Request the value and derivatives, at a physical point, of an associated intensity function, as specified by an object or (optionally) its children.
4. Specify the coordinate transformation that maps a parent object's coordinate system into a child object's coordinate system.
5. Compute the bounding box of a spatial object and (optionally) its children.
6. Query the resolution at which the object was originally computed. For example, you can query the resolution (i.e., voxel spacing) of the image used to generate a particular instance of a itk::BlobSpatialObject.

Currently implemented types of spatial objects include: Blob, Ellipse, Group, Image, Line, Surface, and Tube. The itk::Scene object is used to hold a list of spatial objects that may in turn have children. Each spatial object can be assigned a color property. Each spatial object type has its own capabilities. For example, itk::TubeSpatialObjects indicate to what point on their parent tube they connect.

There are a limited number of spatial objects and their methods in ITK, but their number is growing and their potential is huge. Using the nominal spatial object capabilities, methods such as marching cubes or mutual information registration, can be applied to objects regardless of their internal representation. By having a common API, the same method can be used to register a parametric representation of a heart with an individual's CT data or to register two hand segmentations of a liver.

3.7 Wrapping

While the core of ITK is implemented in C++, Tcl and Python bindings can be automatically generated and ITK programs can be created using these programming languages. This capability is under active development and is for the advanced user only. However, this brief description will give you an idea of what is possible and where to look if you are interested in this facility.

The wrapping process in ITK is quite complex due to the use of generic programming (i.e., extensive use of C++ templates). Systems like VTK that use their own wrapping facility are non-templated and customized to the coding methodology found in the system. Even systems like SWIG that are designed for general wrapper generation have difficulty with ITK code because general C++ is difficult to parse. As a result, the ITK wrapper generator uses a combination of tools to produce language bindings.

1. gccxml is a modified version of the GNU compiler gcc that produces an XML description of an input C++ program.
2. CABLE processes XML information from gccxml and produces additional input to the next tool (i.e., CSWIG indicating what is to be wrapped).
3. CSWIG is a modified version of SWIG that has SWIG's usual parser replaced with an XML parser (XML produced from CABLE and gccxml.) CSWIG produces the appropriate language bindings (either Tcl or Python). (Note: since SWIG is capable of producing language bindings for eleven different interpreted languages including Java, and Perl, it is expected that support for some of these languages will be added in the future.)

To learn more about the wrapping process, please read the file found in `Wrapping/CSwig/README`. Also note that there are some simple test scripts found in `Wrapping/CSwig/Tests`. Additional tests and examples are found in the Testing/Code/*/ directories.

The result of the wrapping process is a set of shared libraries/dll's that can be used by the interpreted languages. There is almost a direct translation from C++, with the differences being the particular syntactical requirements of each language. For example, in the directory `Testing/Code/Algorithms`, the test `itkCurvatureFlowTestTcl2.tcl` has a code fragment that appears as follows:

```
set reader [itkImageFileReaderF2_New]
```

```
    $reader SetFileName "${ITK_TEST_INPUT}/cthead1.png"

  set cf [itkCurvatureFlowImageFilterF2F2_New]
    $cf SetInput [$reader GetOutput]
    $cf SetTimeStep 0.25
    $cf SetNumberOfIterations 10
```

The same code in C++ would appear as follows:

```
  itk::ImageFileReader<ImageType>::Pointer reader =
              itk::ImageFileReader<ImageType>::New();
  reader->SetFileName("cthead1.png");

  itk::CurvatureFlowImageFilter<ImageType,ImageType>::Pointer cf =
      itk::CurvatureFlowImageFilter<ImageType,ImageType>::New();
    cf->SetInput(reader->GetOutput());
    cf->SetTimeStep(0.25);
    cf->SetNumberOfIterations(10);
```

This example demonstrates an important difference between C++ and a wrapped language such as Tcl. Templated classes must be instantiated prior to wrapping. That is, the template parameters must be specified as part of the wrapping process. In the example above, the `CurvatureFlowImageFilterF2F2` indicates that this filter has been instantiated using an input and output image type of two-dimensional float values (e.g., `F2`). Typically just a few common types are selected for the wrapping process to avoid an explosion of types and hence, library size. To add a new type requires rerunning the wrapping process to produce new libraries.

The advantage of interpreted languages is that they do not require the lengthy compile/link cycle of a compiled language like C++. Moreover, they typically come with a suite of packages that provide useful functionality. For example, the Tk package (i.e., Tcl/Tk and Python/Tk) provides tools for creating sophisticated user interfaces. In the future it is likely that more applications and tests will be implemented in the various interpreted languages supported by ITK.

Part II

User’s Guide

CHAPTER

FOUR

DataRepresentation

This chapter introduces the basic classes responsible for representing data in ITK. The most common classes are the `itk::Image`, the `itk::Mesh` and the `itk::PointSet`.

4.1 Image

The `itk::Image` class follows the spirit of Generic Programming, where types are separated from the algorithmic behavior of the class. ITK supports images with any pixel type and any spatial dimension.

4.1.1 Creating an Image

The source code for this section can be found in the file
`Examples/DataRepresentation/Image/Image1.cxx`.

This example illustrates how to manually construct an `itk::Image` class. The following is the minimal code needed to instantiate, declare and create the image class.

First, the header file of the Image class must be included.

```
#include "itkImage.h"
```

Then we must decide with what type to represent the pixels and what the dimension of the image will be. With these two parameters we can instantiate the image class. Here we create a 3D image with `unsigned short` pixel data.

```
  typedef itk::Image< unsigned short, 3 > ImageType;
```

The image can then be created by invoking the `New()` operator from the corresponding image type and assigning the result to a `itk::SmartPointer`.

```
ImageType::Pointer image = ImageType::New();
```

In ITK, images exist in combination with one or more *regions*. A region is a subset of the image and indicates a portion of the image that may be processed by other classes in the system. One of the most common regions is the *LargestPossibleRegion*, which defines the image in its entirety. Other important regions found in ITK are the *BufferedRegion*, which is the portion of the image actually maintained in memory, and the *RequestedRegion*, which is the region requested by a filter or other class when operating on the image.

In ITK, manually creating an image requires that the image is instantiated as previously shown, and that regions describing the image are then associated with it.

A region is defined by two classes: the `itk::Index` and `itk::Size` classes. The origin of the region within the image with which it is associated is defined by Index. The extent, or size, of the region is defined by Size. Index is represented by a n-dimensional array where each component is an integer indicating—in topological image coordinates—the initial pixel of the image. When an image is created manually, the user is responsible for defining the image size and the index at which the image grid starts. These two parameters make it possible to process selected regions.

The starting point of the image is defined by an Index class that is an n-dimensional array where each component is an integer indicating the grid coordinates of the initial pixel of the image.

```
ImageType::IndexType start;

start[0] =   0;  // first index on X
start[1] =   0;  // first index on Y
start[2] =   0;  // first index on Z
```

The region size is represented by an array of the same dimension of the image (using the Size class). The components of the array are unsigned integers indicating the extent in pixels of the image along every dimension.

```
ImageType::SizeType  size;

size[0]  = 200;  // size along X
size[1]  = 200;  // size along Y
size[2]  = 200;  // size along Z
```

Having defined the starting index and the image size, these two parameters are used to create an ImageRegion object which basically encapsulates both concepts. The region is initialized with the starting index and size of the image.

```
ImageType::RegionType region;

region.SetSize( size );
region.SetIndex( start );
```

Finally, the region is passed to the `Image` object in order to define its extent and origin. The `SetRegions` method sets the LargestPossibleRegion, BufferedRegion, and RequestedRegion simultaneously. Note that none of the operations performed to this point have allocated memory for the image pixel data. It is necessary to invoke the `Allocate()` method to do this. Allocate does not require any arguments since all the information needed for memory allocation has already been provided by the region.

```
  image->SetRegions( region );
  image->Allocate();
```

In practice it is rare to allocate and initialize an image directly. Images are typically read from a source, such a file or data acquisition hardware. The following example illustrates how an image can be read from a file.

4.1.2 Reading an Image from a File

The source code for this section can be found in the file
`Examples/DataRepresentation/Image/Image2.cxx`.

The first thing required to read an image from a file is to include the header file of the `itk::ImageFileReader` class.

```
#include "itkImageFileReader.h"
```

Then, the image type should be defined by specifying the type used to represent pixels and the dimensions of the image.

```
  typedef unsigned char          PixelType;
  const unsigned int             Dimension = 3;

  typedef itk::Image< PixelType, Dimension >   ImageType;
```

Using the image type, it is now possible to instantiate the image reader class. The image type is used as a template parameter to define how the data will be represented once it is loaded into memory. This type does not have to correspond exactly to the type stored in the file. However, a conversion based on C-style type casting is used, so the type chosen to represent the data on disk must be sufficient to characterize it accurately. Readers do not apply any transformation to the pixel data other than casting from the pixel type of the file to the pixel type of the ImageFileReader. The following illustrates a typical instantiation of the ImageFileReader type.

```
  typedef itk::ImageFileReader< ImageType >  ReaderType;
```

The reader type can now be used to create one reader object. A `itk::SmartPointer` (defined by the `::Pointer` notation) is used to receive the reference to the newly created reader. The `New()` method is invoked to create an instance of the image reader.

```
ReaderType::Pointer reader = ReaderType::New();
```

The minimum information required by the reader is the filename of the image to be loaded in memory. This is provided through the `SetFileName()` method. The file format here is inferred from the filename extension. The user may also explicitly specify the data format explicitly using the `itk::ImageIO` (See Chapter 7.1 263 for more information

```
const char * filename = argv[1];
reader->SetFileName( filename );
```

Reader objects are referred to as pipeline source objects; they respond to pipeline update requests and initiate the data flow in the pipeline. The pipeline update mechanism ensures that the reader only executes when a data request is made to the reader and the reader has not read any data. In the current example we explicitly invoke the `Update()` method because the output of the reader is not connected to other filters. In normal application the reader's output is connected to the input of an image filter and the update invocation on the filter triggers an update of the reader. The following line illustrates how an explicit update is invoked on the reader.

```
reader->Update();
```

Access to the newly read image can be gained by calling the `GetOutput()` method on the reader. This method can also be called before the update request is sent to the reader. The reference to the image will be valid even though the image will be empty until the reader actually executes.

```
ImageType::Pointer image = reader->GetOutput();
```

Any attempt to access image data before the reader executes will yield an image with no pixel data. It is likely that a program crash will result since the image will not have been properly initialized.

4.1.3 Accessing Pixel Data

The source code for this section can be found in the file
`Examples/DataRepresentation/Image/Image3.cxx`.

This example illustrates the use of the `SetPixel()` and `GetPixel()` methods. These two methods provide direct access to the pixel data contained in the image. Note that these two methods are relatively slow and should not be used in situations where high-performance access is required. Image iterators are the appropriate mechanism to efficiently access image pixel data. (See Chapter 11 on page 701 for information about image iterators.)

The individual position of a pixel inside the image is identified by a unique index. An index is an array of integers that defines the position of the pixel along each coordinate dimension of the image. The IndexType is automatically defined by the image and can be accessed using the scope operator like `itk::Index`. The length of the array will match the dimensions of the associated image.

The following code illustrates the declaration of an index variable and the assignment of values to each of its components. Please note that `Index` does not use SmartPointers to access it. This is because `Index` is a light-weight object that is not intended to be shared between objects. It is more efficient to produce multiple copies of these small objects than to share them using the SmartPointer mechanism.

The following lines declare an instance of the index type and initialize its content in order to associate it with a pixel position in the image.

```
ImageType::IndexType pixelIndex;

pixelIndex[0] = 27;   // x position
pixelIndex[1] = 29;   // y position
pixelIndex[2] = 37;   // z position
```

Having defined a pixel position with an index, it is then possible to access the content of the pixel in the image. The `GetPixel()` method allows us to get the value of the pixels.

```
ImageType::PixelType   pixelValue = image->GetPixel( pixelIndex );
```

The `SetPixel()` method allows us to set the value of the pixel.

```
image->SetPixel(   pixelIndex,   pixelValue+1  );
```

Please note that `GetPixel()` returns the pixel value using copy and not reference semantics. Hence, the method cannot be used to modify image data values.

Remember that both `SetPixel()` and `GetPixel()` are inefficient and should only be used for debugging or for supporting interactions like querying pixel values by clicking with the mouse.

4.1.4 Defining Origin and Spacing

The source code for this section can be found in the file
`Examples/DataRepresentation/Image/Image4.cxx`.

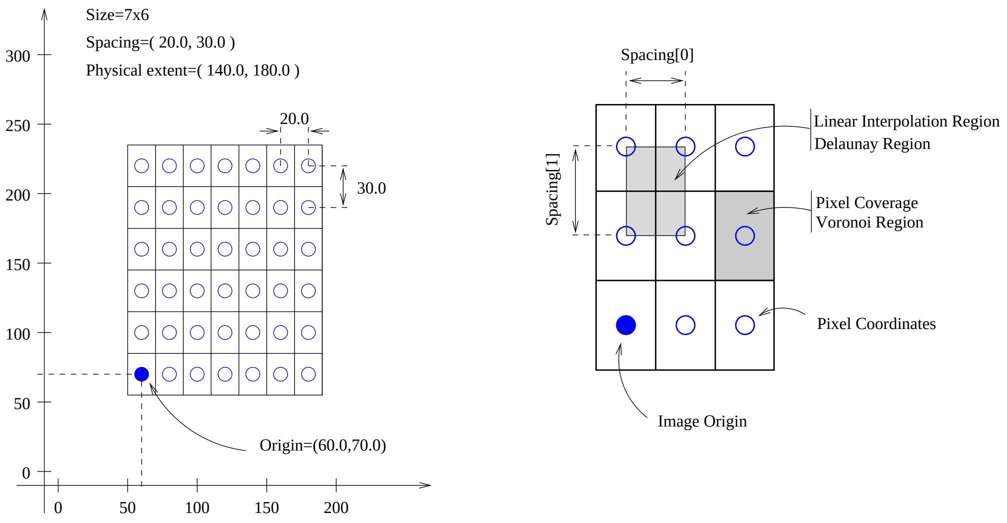


Figure 4.1: Geometrical concepts associated with the ITK image.

Even though ITK can be used to perform general image processing tasks, the primary purpose of the toolkit is the processing of medical image data. In that respect, additional information about the images is considered mandatory. In particular the information associated with the physical spacing between pixels and the position of the image in space with respect to some world coordinate system are extremely important.

Image origin and spacing are fundamental to many applications. Registration, for example, is performed in physical coordinates. Improperly defined spacing and origins will result in inconsistent results in such processes. Medical images with no spatial information should not be used for medical diagnosis, image analysis, feature extraction, assisted radiation therapy or image guided surgery. In other words, medical images lacking spatial information are not only useless but also hazardous.

Figure 4.1 illustrates the main geometrical concepts associated with the `itk::Image`. In this figure, circles are used to represent the center of pixels. The value of the pixel is assumed to exist as a Dirac Delta Function located at the pixel center. Pixel spacing is measured between the pixel centers and can be different along each dimension. The image origin is associated with the coordinates of the first pixel in the image. A *pixel* is considered to be the rectangular region surrounding the pixel center holding the data value. This can be viewed as the Voronoi region of the image grid, as illustrated in the right side of the figure. Linear interpolation of image values is performed inside the Delaunay region whose corners are pixel centers.

Image spacing is represented in a `FixedArray` whose size matches the dimension of the image. In order to manually set the spacing of the image, an array of the corresponding type must be created. The elements of the array should then be initialized with the spacing between the centers of adjacent pixels. The following code illustrates the methods available in the Image class for dealing with spacing and origin.

```
ImageType::SpacingType spacing;

// Note: measurement units (e.g., mm, inches, etc.) are defined by the application.
spacing[0] = 0.33; // spacing along X
spacing[1] = 0.33; // spacing along Y
spacing[2] = 1.20; // spacing along Z
```

The array can be assigned to the image using the `SetSpacing()` method.

```
image->SetSpacing( spacing );
```

The spacing information can be retrieved from an image by using the `GetSpacing()` method. This method returns a reference to a `FixedArray`. The returned object can then be used to read the contents of the array. Note the use of the `const` keyword to indicate that the array will not be modified.

```
const ImageType::SpacingType& sp = image->GetSpacing();

std::cout << "Spacing = ";
std::cout << sp[0] << ", " << sp[1] << ", " << sp[2] << std::endl;
```

The image origin is managed in a similar way to the spacing. A `Point` of the appropriate dimension must first be allocated. The coordinates of the origin can then be assigned to every component. These coordinates correspond to the position of the first pixel of the image with respect to an arbitrary reference system in physical space. It is the user's responsibility to make sure that multiple images used in the same application are using a consistent reference system. This is extremely important in image registration applications.

The following code illustrates the creation and assignment of a variable suitable for initializing the image origin.

```
ImageType::PointType origin;

origin[0] = 0.0;  // coordinates of the
origin[1] = 0.0;  // first pixel in N-D
origin[2] = 0.0;

image->SetOrigin( origin );
```

The origin can also be retrieved from an image by using the `GetOrigin()` method. This will return a reference to a `Point`. The reference can be used to read the contents of the array. Note again the use of the `const` keyword to indicate that the array contents will not be modified.

```
const ImageType::PointType& orgn = image->GetOrigin();

std::cout << "Origin = ";
std::cout << orgn[0] << ", " << orgn[1] << ", " << orgn[2] << std::endl;
```

Once the spacing and origin of the image have been initialized, the image will correctly map pixel indices to and from physical space coordinates. The following code illustrates how a point in physical space can be mapped into an image index for the purpose of reading the content of the closest pixel.

First, a `itk::Point` type must be declared. The point type is templated over the type used to represent coordinates and over the dimension of the space. In this particular case, the dimension of the point must match the dimension of the image.

```
typedef itk::Point< double, ImageType::ImageDimension > PointType;
```

The Point class, like an `itk::Index`, is a relatively small and simple object. For this reason, it is not reference-counted like the large data objects in ITK. Consequently, it is also not manipulated with `itk::SmartPointer`s. Point objects are simply declared as instances of any other C++ class. Once the point is declared, its components can be accessed using traditional array notation. In particular, the `[]` operator is available. For efficiency reasons, no bounds checking is performed on the index used to access a particular point component. It is the user's responsibility to make sure that the index is in the range $\{0, Dimension - 1\}$.

```
PointType point;

point[0] = 1.45;    // x coordinate
point[1] = 7.21;    // y coordinate
point[2] = 9.28;    // z coordinate
```

The image will map the point to an index using the values of the current spacing and origin. An index object must be provided to receive the results of the mapping. The index object can be instantiated by using the `IndexType` defined in the Image type.

```
ImageType::IndexType pixelIndex;
```

The `TransformPhysicalPointToIndex()` method of the image class will compute the pixel index closest to the point provided. The method checks for this index to be contained inside the current buffered pixel data. The method returns a boolean indicating whether the resulting index falls inside the buffered region or not. The output index should not be used when the returned value of the method is `false`.

The following lines illustrate the point to index mapping and the subsequent use of the pixel index for accessing pixel data from the image.

```
bool isInside = image->TransformPhysicalPointToIndex( point, pixelIndex );

if ( isInside )
  {
  ImageType::PixelType pixelValue = image->GetPixel( pixelIndex );

  pixelValue += 5;

  image->SetPixel( pixelIndex, pixelValue );
  }
```

Remember that `GetPixel()` and `SetPixel()` are very inefficient methods for accessing pixel data. Image iterators should be used when massive access to pixel data is required.

4.1.5 RGB Images

The term RGB (Red, Green, Blue) stands for a color representation commonly used in digital imaging. RGB is a representation of the human physiological capability to analyze visual light using three spectral-selective sensors [53, 94]. The human retina possess different types of light sensitive cells. Three of them, known as *cones*, are sensitive to color [31] and their regions of sensitivity loosely match regions of the spectrum that will be perceived as red, green and blue respectively. The *rods* on the other hand provide no color discrimination and favor high resolution and high sensitivity[1]. A fifth type of receptors, the *ganglion cells*, also known as circadian[2] receptors are sensitive to the lighting conditions that differentiate day from night. These receptors evolved as a mechanism for synchronizing the physiology with the time of the day. Cellular controls for circadian rythms are present in every cell of an organism and are known to be exquisitively precise [50].

The RGB space has been constructed as a representation of a physiological response to light by the three types of *cones* in the human eye. RGB is not a Vector space. For example, negative numbers are not appropriate in a color space because they will be the equivalent of “negative stimulation” on the human eye. In the context of colorimetry, negative color values are used as an artificial construct for color comparison in the sense that

$$ColorA = ColorB - ColorC \tag{4.1}$$

just as a way of saying that we can produce *ColorB* by combining *ColorA* and *ColorC*. However, we must be aware that (at least in emitted light) it is not possible to *substract light*. So when we mention Equation 4.1 we actually mean

$$ColorB = ColorA + ColorC \tag{4.2}$$

[1]The human eye is capable of perceiving a single isolated photon.
[2]The term *Circadian* refers to the cycle of day and night, that is, events that are repeated with 24 hours intervals.

On the other hand, when dealing with printed color and with paint, as opposed to emitted light like in computer screens, the physical behavior of color allows for subtraction. This is because strictly speaking the objects that we see as red are those that absorb all light frequencies except those in the red section of the spectrum [94].

The concept of addition and subtraction of colors has to be carefully interpreted. In fact, RGB has a different definition regarding whether we are talking about the channels associated to the three color sensors of the human eye, or to the three phosphors found in most computer monitors or to the color inks that are used for printing reproduction. Color spaces are usually non linear and do not even from a Group. For example, not all visible colors can be represented in RGB space [94].

ITK introduces the `itk::RGBPixel` type as a support for representing the values of an RGB color space. As such, the RGBPixel class embodies a different concept from the one of an `itk::Vector` in space. For this reason, the RGBPixel lack many of the operators that may be naively expected from it. In particular, there are no defined operations for subtraction or addition.

When you anticipate to perform the operation of “Mean” on a RGB type you are assuming that in the color space provides the action of finding a color in the middle of two colors, can be found by using a linear operation between their numerical representation. This is unfortunately not the case in color spaces due to the fact that they are based on a human physiological response [53].

If you decide to interpret RGB images as simply three independent channels then you should rather use the `itk::Vector` type as pixel type. In this way, you will have access to the set of operations that are defined in Vector spaces. The current implementation of the RGBPixel in ITK presumes that RGB color images are intended to be used in applications where a formal interpretation of color is desired, therefore only the operations that are valid in a color space are available in the RGBPixel class.

The following example illustrates how RGB images can be represented in ITK.

The source code for this section can be found in the file
`Examples/DataRepresentation/Image/RGBImage.cxx`.

Thanks to the flexibility offered by the Generic Programming style on which ITK is based, it is possible to instantiate images of arbitrary pixel type. The following example illustrates how a color image with RGB pixels can be defined.

A class intended to support the RGB pixel type is available in ITK. You could also define your own pixel class and use it to instantiate a custom image type. In order to use the `itk::RGBPixel` class, it is necessary to include its header file.

```
#include "itkRGBPixel.h"
```

The RGB pixel class is templated over a type used to represent each one of the red, green and blue pixel components. A typical instantiation of the templated class is as follows.

```
  typedef itk::RGBPixel< unsigned char >    PixelType;
```

The type is then used as the pixel template parameter of the image.

```
typedef itk::Image< PixelType, 3 >   ImageType;
```

The image type can be used to instantiate other filter, for example, an `itk::ImageFileReader` object that will read the image from a file.

```
typedef itk::ImageFileReader< ImageType >  ReaderType;
```

Access to the color components of the pixels can now be performed using the methods provided by the RGBPixel class.

```
PixelType onePixel = image->GetPixel( pixelIndex );

PixelType::ValueType red   = onePixel.GetRed();
PixelType::ValueType green = onePixel.GetGreen();
PixelType::ValueType blue  = onePixel.GetBlue();
```

The subindex notation can also be used since the `itk::RGBPixel` inherits the `[]` operator from the `itk::FixedArray` class.

```
red   = onePixel[0];  // extract Red   component
green = onePixel[1];  // extract Green component
blue  = onePixel[2];  // extract Blue  component

std::cout << "Pixel values:" << std::endl;
std::cout << "Red = "
          << itk::NumericTraits<PixelType::ValueType>::PrintType(red)
          << std::endl;
std::cout << "Green = "
          << itk::NumericTraits<PixelType::ValueType>::PrintType(green)
          << std::endl;
std::cout << "Blue = "
          << itk::NumericTraits<PixelType::ValueType>::PrintType(blue)
          << std::endl;
```

4.1.6 Vector Images

The source code for this section can be found in the file
`Examples/DataRepresentation/Image/VectorImage.cxx`.

Many image processing tasks require images of non-scalar pixel type. A typical example is an image of vectors. This is the image type required to represent the gradient of a scalar image.

The following code illustrates how to instantiate and use an image whose pixels are of vector type.

For convenience we use the itk::Vector class to define the pixel type. The Vector class is intended to represent a geometrical vector in space. It is not intended to be used as an array container like the std::vector in STL. If you are interested in containers, the itk::VectorContainer class may provide the functionality you want.

The first step is to include the header file of the Vector class.

```
#include "itkVector.h"
```

The Vector class is templated over the type used to represent the coordinate in space and over the dimension of the space. In this example, we want the vector dimension to match the image dimension, but this is by no means a requirement. We could have defined a four-dimensional image with three-dimensional vectors as pixels.

```
  typedef itk::Vector< float, 3 >       PixelType;
  typedef itk::Image< PixelType, 3 >    ImageType;
```

The Vector class inherits the operator [] from the itk::FixedArray class. This makes it possible to access the Vector's components using index notation.

```
  ImageType::PixelType   pixelValue;

  pixelValue[0] =  1.345;   // x component
  pixelValue[1] =  6.841;   // y component
  pixelValue[2] =  3.295;   // x component
```

We can now store this vector in one of the image pixels by defining an index and invoking the SetPixel() method.

```
  image->SetPixel(   pixelIndex,   pixelValue  );
```

4.1.7 Importing Image Data from a Buffer

The source code for this section can be found in the file
Examples/DataRepresentation/Image/Image5.cxx.

This example illustrates how to import data into the itk::Image class. This is particularly useful for interfacing with other software systems. Many systems use a contiguous block of memory as a buffer for image pixel data. The current example assumes this is the case and feeds the buffer into an itk::ImportImageFilter, thereby producing an Image as output.

For fun we create a synthetic image with a centered sphere in a locally allocated buffer and pass this block of memory to the ImportImageFilter. This example is set up so that on execution, the user must provide the name of an output file as a command-line argument.

First, the header file of the ImportImageFilter class must be included.

```
#include "itkImage.h"
#include "itkImportImageFilter.h"
```

Next, we select the data type to use to represent the image pixels. We assume that the external block of memory uses the same data type to represent the pixels.

```
  typedef unsigned char   PixelType;
  const unsigned int Dimension = 3;
  typedef itk::Image< PixelType, Dimension > ImageType;
```

The type of the ImportImageFilter is instantiated in the following line.

```
  typedef itk::ImportImageFilter< PixelType, Dimension >   ImportFilterType;
```

A filter object created using the `New()` method is then assigned to a `SmartPointer`.

```
  ImportFilterType::Pointer importFilter = ImportFilterType::New();
```

This filter requires the user to specify the size of the image to be produced as output. The `SetRegion()` method is used to this end. The image size should exactly match the number of pixels available in the locally allocated buffer.

```
  ImportFilterType::SizeType  size;

  size[0]  = 200;  // size along X
  size[1]  = 200;  // size along Y
  size[2]  = 200;  // size along Z

  ImportFilterType::IndexType start;
  start.Fill( 0 );

  ImportFilterType::RegionType region;
  region.SetIndex( start );
  region.SetSize(  size  );

  importFilter->SetRegion( region );
```

The origin of the output image is specified with the `SetOrigin()` method.

```
double origin[ Dimension ];
origin[0] = 0.0;    // X coordinate
origin[1] = 0.0;    // Y coordinate
origin[2] = 0.0;    // Z coordinate

importFilter->SetOrigin( origin );
```

The spacing of the image is passed with the `SetSpacing()` method.

```
double spacing[ Dimension ];
spacing[0] = 1.0;    // along X direction
spacing[1] = 1.0;    // along Y direction
spacing[2] = 1.0;    // along Z direction

importFilter->SetSpacing( spacing );
```

Next we allocate the memory block containing the pixel data to be passed to the ImportImage-Filter. Note that we use exactly the same size that was specified with the `SetRegion()` method. In a practical application, you may get this buffer from some other library using a different data structure to represent the images.

```
const unsigned int numberOfPixels =  size[0] * size[1] * size[2];
PixelType * localBuffer = new PixelType[ numberOfPixels ];
```

Here we fill up the buffer with a binary sphere. We use simple `for()` loops here similar to those found in the C or FORTRAN programming languages. Note that ITK does not use `for()` loops in its internal code to access pixels. All pixel access tasks are instead performed using `itk::ImageIterator`s that support the management of n-dimensional images.

```
const double radius2 = radius * radius;
PixelType * it = localBuffer;

for(unsigned int z=0; z < size[2]; z++)
  {
  const double dz = static_cast<double>( z ) - static_cast<double>(size[2])/2.0;
  for(unsigned int y=0; y < size[1]; y++)
    {
    const double dy = static_cast<double>( y ) - static_cast<double>(size[1])/2.0;
    for(unsigned int x=0; x < size[0]; x++)
      {
      const double dx = static_cast<double>( x ) - static_cast<double>(size[0])/2.0;
      const double d2 = dx*dx + dy*dy + dz*dz;
      *it++ = ( d2 < radius2 ) ? 255 : 0;
      }
    }
  }
```

The buffer is passed to the ImportImageFilter with the `SetImportPointer()`. Note that the last argument of this method specifies who will be responsible for deleting the memory block once it is no longer in use. A `false` value indicates that the ImportImageFilter will not try to delete the buffer when its destructor is called. A `true` value, on the other hand, will allow the filter to delete the memory block upon destruction of the import filter.

For the ImportImageFilter to appropriately delete the memory block, the memory must be allocated with the C++ `new()` operator. Memory allocated with other memory allocation mechanisms, such as C `malloc` or `calloc`, will not be deleted properly by the ImportImageFilter. In other words, it is the application programmer's responsibility to ensure that ImportImageFilter is only given permission to delete the C++ `new` operator-allocated memory.

```
  const bool importImageFilterWillOwnTheBuffer = true;
  importFilter->SetImportPointer( localBuffer, numberOfPixels,
                                  importImageFilterWillOwnTheBuffer );
```

Finally, we can connect the output of this filter to a pipeline. For simplicity we just use a writer here, but it could be any other filter.

```
  writer->SetInput(  importFilter->GetOutput()  );
```

Note that we do not call `delete` on the buffer since we pass `true` as the last argument of `SetImportPointer()`. Now the buffer is owned by the ImportImageFilter.

4.2 PointSet

4.2.1 Creating a PointSet

The source code for this section can be found in the file
`Examples/DataRepresentation/Mesh/PointSet1.cxx`.

The `itk::PointSet` is a basic class intended to represent geometry in the form of a set of points in n-dimensional space. It is the base class for the `itk::Mesh` providing the methods necessary to manipulate sets of point. Points can have values associated with them. The type of such values is defined by a template parameter of the `itk::PointSet` class (i.e., `TPixelType`. Two basic interaction styles of PointSets are available in ITK. These styles are referred to as *static* and *dynamic*. The first style is used when the number of points in the set is known in advance and is not expected to change as a consequence of the manipulations performed on the set. The dynamic style, on the other hand, is intended to support insertion and removal of points in an efficient manner. Distinguishing between the two styles is meant to facilitate the fine tuning of a `PointSet`'s behavior while optimizing performance and memory management.

In order to use the PointSet class, its header file should be included.

```
#include "itkPointSet.h"
```

Then we must decide what type of value to associate with the points. This is generally called the PixelType in order to make the terminology consistent with the itk::Image. The PointSet is also templated over the dimension of the space in which the points are represented. The following declaration illustrates a typical instantiation of the PointSet class.

```
typedef itk::PointSet< unsigned short, 3 > PointSetType;
```

A PointSet object is created by invoking the New() method on its type. The resulting object must be assigned to a SmartPointer. The PointSet is then reference-counted and can be shared by multiple objects. The memory allocated for the PointSet will be released when the number of references to the object is reduced to zero. This simply means that the user does not need to be concerned with invoking the Delete() method on this class. In fact, the Delete() method should **never** be called directly within any of the reference-counted ITK classes.

```
PointSetType::Pointer  pointsSet = PointSetType::New();
```

Following the principles of Generic Programming, the PointSet class has a set of associated defined types to ensure that interacting objects can be declared with compatible types. This set of type definitions is commonly known as a set of *traits*. Among them we can find the PointType type, for example. This is the type used by the point set to represent points in space. The following declaration takes the point type as defined in the PointSet traits and renames it to be conveniently used in the global namespace.

```
typedef PointSetType::PointType     PointType;
```

The PointType can now be used to declare point objects to be inserted in the PointSet. Points are fairly small objects, so it is inconvenient to manage them with reference counting and smart pointers. They are simply instantiated as typical C++ classes. The Point class inherits the [] operator from the itk::Array class. This makes it possible to access its components using index notation. For efficiency's sake no bounds checking is performed during index access. It is the user's responsibility to ensure that the index used is in the range $\{0, Dimension-1\}$. Each of the components in the point is associated with space coordinates. The following code illustrates how to instantiate a point and initialize its components.

```
PointType p0;
p0[0] = -1.0;     //  x coordinate
p0[1] = -1.0;     //  y coordinate
p0[2] =  0.0;     //  z coordinate
```

Points are inserted in the PointSet by using the SetPoint() method. This method requires the user to provide a unique identifier for the point. The identifier is typically an unsigned integer that will enumerate the points as they are being inserted. The following code shows how three points are inserted into the PointSet.

```
pointsSet->SetPoint( 0, p0 );
pointsSet->SetPoint( 1, p1 );
pointsSet->SetPoint( 2, p2 );
```

It is possible to query the PointSet in order to determine how many points have been inserted into it. This is done with the `GetNumberOfPoints()` method as illustrated below.

```
const unsigned int numberOfPoints = pointsSet->GetNumberOfPoints();
std::cout << numberOfPoints << std::endl;
```

Points can be read from the PointSet by using the `GetPoint()` method and the integer identifier. The point is stored in a pointer provided by the user. If the identifier provided does not match an existing point, the method will return `false` and the contents of the point will be invalid. The following code illustrates point access using defensive programming.

```
PointType pp;
bool pointExists =  pointsSet->GetPoint( 1, & pp );

if( pointExists )
  {
  std::cout << "Point is = " << pp << std::endl;
  }
```

`GetPoint()` and `SetPoint()` are not the most efficient methods to access points in the PointSet. It is preferable to get direct access to the internal point container defined by the *traits* and use iterators to walk sequentially over the list of points (as shown in the following example).

4.2.2 Getting Access to Points

The source code for this section can be found in the file
`Examples/DataRepresentation/Mesh/PointSet2.cxx`.

The `itk::PointSet` class uses an internal container to manage the storage of `itk::Point`s. It is more efficient, in general, to manage points by using the access methods provided directly on the points container. The following example illustrates how to interact with the point container and how to use point iterators.

The type is defined by the *traits* of the PointSet class. The following line conveniently takes the PointsContainer type from the PointSet traits and declare it in the global namespace.

```
typedef PointSetType::PointsContainer      PointsContainer;
```

The actual type of the PointsContainer depends on what style of PointSet is being used. The dynamic PointSet use the `itk::MapContainer` while the static PointSet uses the itk::VectorContainer. The vector and map containers are basically ITK wrappers around the STL classes std::map and std::vector. By default, the PointSet uses a static style, hence the default type of point container is an VectorContainer. Both the map and vector container are templated over the type of the elements they contain. In this case they are templated over PointType. Containers are reference counted object. They are then created with the New() method and assigned to a itk::SmartPointer after creation. The following line creates a point container compatible with the type of the PointSet from which the trait has been taken.

```
PointsContainer::Pointer points = PointsContainer::New();
```

Points can now be defined using the PointType trait from the PointSet.

```
typedef PointSetType::PointType   PointType;
PointType p0;
PointType p1;
p0[0] = -1.0; p0[1] = 0.0; p0[2] = 0.0; // Point 0 = {-1,0,0 }
p1[0] =  1.0; p1[1] = 0.0; p1[2] = 0.0; // Point 1 = { 1,0,0 }
```

The created points can be inserted in the PointsContainer using the generic method InsertElement() which requires an identifier to be provided for each point.

```
unsigned int pointId = 0;
points->InsertElement( pointId++ , p0 );
points->InsertElement( pointId++ , p1 );
```

Finally the PointsContainer can be assigned to the PointSet. This will substitute any previously existing PointsContainer on the PointSet. The assignment is done using the SetPoints() method.

```
pointSet->SetPoints( points );
```

The PointsContainer object can be obtained from the PointSet using the GetPoints() method. This method returns a pointer to the actual container owned by the PointSet which is then assigned to a SmartPointer.

```
PointsContainer::Pointer  points2 = pointSet->GetPoints();
```

The most efficient way to sequentially visit the points is to use the iterators provided by PointsContainer. The Iterator type belongs to the traits of the PointsContainer classes. It behaves pretty much like the STL iterators.[3] The Points iterator is not a reference counted class, so it is created directly from the traits without using SmartPointers.

[3]If you dig deep enough into the code, you will discover that these iterators are actually ITK wrappers around STL iterators.

```
  typedef PointsContainer::Iterator     PointsIterator;
```

The subsequent use of the iterator follows what you may expect from a STL iterator. The iterator to the first point is obtained from the container with the `Begin()` method and assigned to another iterator.

```
  PointsIterator  pointIterator = points->Begin();
```

The `++` operator on the iterator can be used to advance from one point to the next. The actual value of the Point to which the iterator is pointing can be obtained with the `Value()` method. The loop for walking through all the points can be controlled by comparing the current iterator with the iterator returned by the `End()` method of the PointsContainer. The following lines illustrate the typical loop for walking through the points.

```
  PointsIterator end = points->End();
  while( pointIterator != end )
    {
    PointType p = pointIterator.Value();   // access the point
    std::cout << p << std::endl;           // print the point
    ++pointIterator;                       // advance to next point
    }
```

Note that as in STL, the iterator returned by the `End()` method is not a valid iterator. This is called a past-end iterator in order to indicate that it is the value resulting from advancing one step after visiting the last element in the container.

The number of elements stored in a container can be queried with the `Size()` method. In the case of the PointSet, the following two lines of code are equivalent, both of them returning the number of points in the PointSet.

```
  std::cout << pointSet->GetNumberOfPoints() << std::endl;
  std::cout << pointSet->GetPoints()->Size() << std::endl;
```

4.2.3 Getting Access to Data in Points

The source code for this section can be found in the file
`Examples/DataRepresentation/Mesh/PointSet3.cxx`.

The `itk::PointSet` class was designed to interact with the Image class. For this reason it was found convenient to allow the points in the set to hold values that could be computed from images. The value associated with the point is referred as `PixelType` in order to make it consistent with image terminology. Users can define the type as they please thanks to the flexibility offered by the Generic Programming approach used in the toolkit. The `PixelType` is the first template parameter of the PointSet.

The following code defines a particular type for a pixel type and instantiates a PointSet class with it.

```
typedef unsigned short  PixelType;
typedef itk::PointSet< PixelType, 3 > PointSetType;
```

Data can be inserted into the PointSet using the `SetPointData()` method. This method requires the user to provide an identifier. The data in question will be associated to the point holding the same identifier. It is the user's responsibility to verify the appropriate matching between inserted data and inserted points. The following line illustrates the use of the `SetPointData()` method.

```
unsigned int dataId =  0;
PixelType value     = 79;
pointSet->SetPointData( dataId++, value );
```

Data associated with points can be read from the PointSet using the `GetPointData()` method. This method requires the user to provide the identifier to the point and a valid pointer to a location where the pixel data can be safely written. In case the identifier does not match any existing identifier on the PointSet the method will return `false` and the pixel value returned will be invalid. It is the user's responsibility to check the returned boolean value before attempting to use it.

```
const bool found = pointSet->GetPointData( dataId, & value );
if( found )
  {
  std::cout << "Pixel value = " << value << std::endl;
  }
```

The `SetPointData()` and `GetPointData()` methods are not the most efficient way to get access to point data. It is far more efficient to use the Iterators provided by the `PointDataContainer`.

Data associated with points is internally stored in `PointDataContainers`. In the same way as with points, the actual container type used depend on whether the style of the PointSet is static or dynamic. Static point sets will use an `itk::VectorContainer` while dynamic point sets will use an `itk::MapContainer`. The type of the data container is defined as one of the traits in the PointSet. The following declaration illustrates how the type can be taken from the traits and used to conveniently declare a similar type on the global namespace.

```
typedef PointSetType::PointDataContainer      PointDataContainer;
```

Using the type it is now possible to create an instance of the data container. This is a standard reference counted object, henceforth it uses the `New()` method for creation and assigns the newly created object to a SmartPointer.

```
PointDataContainer::Pointer pointData = PointDataContainer::New();
```

Pixel data can be inserted in the container with the method `InsertElement()`. This method requires an identified to be provided for each point data.

```
unsigned int pointId = 0;

PixelType value0 = 34;
PixelType value1 = 67;

pointData->InsertElement( pointId++ , value0 );
pointData->InsertElement( pointId++ , value1 );
```

Finally the PointDataContainer can be assigned to the PointSet. This will substitute any previously existing PointDataContainer on the PointSet. The assignment is done using the `SetPointData()` method.

```
pointSet->SetPointData( pointData );
```

The PointDataContainer can be obtained from the PointSet using the `GetPointData()` method. This method returns a pointer (assigned to a SmartPointer) to the actual container owned by the PointSet.

```
PointDataContainer::Pointer  pointData2 = pointSet->GetPointData();
```

The most efficient way to sequentially visit the data associated with points is to use the iterators provided by `PointDataContainer`. The `Iterator` type belongs to the traits of the PointsContainer classes. The iterator is not a reference counted class, so it is just created directly from the traits without using SmartPointers.

```
typedef PointDataContainer::Iterator     PointDataIterator;
```

The subsequent use of the iterator follows what you may expect from a STL iterator. The iterator to the first point is obtained from the container with the `Begin()` method and assigned to another iterator.

```
PointDataIterator  pointDataIterator = pointData2->Begin();
```

The `++` operator on the iterator can be used to advance from one data point to the next. The actual value of the PixelType to which the iterator is pointing can be obtained with the `Value()` method. The loop for walking through all the point data can be controlled by comparing the current iterator with the iterator returned by the `End()` method of the PointsContainer. The following lines illustrate the typical loop for walking through the point data.

```
  PointDataIterator end = pointData2->End();
  while( pointDataIterator != end )
    {
    PixelType p = pointDataIterator.Value();  // access the pixel data
    std::cout << p << std::endl;              // print the pixel data
    ++pointDataIterator;                      // advance to next pixel/point
    }
```

Note that as in STL, the iterator returned by the `End()` method is not a valid iterator. This is called a *past-end* iterator in order to indicate that it is the value resulting from advancing one step after visiting the last element in the container.

4.2.4 RGB as Pixel Type

The source code for this section can be found in the file
`Examples/DataRepresentation/Mesh/RGBPointSet.cxx`.

The following example illustrates how a point set can be parameterized to manage a particular pixel type. In this case, pixels of RGB type are used. The first step is then to include the header files of the `itk::RGBPixel` and `itk::PointSet` classes.

```
#include "itkRGBPixel.h"
#include "itkPointSet.h"
```

Then, the pixel type can be defined by selecting the type to be used to represent each one of the RGB components.

```
  typedef itk::RGBPixel< float >    PixelType;
```

The newly defined pixel type is now used to instantiate the PointSet type and subsequently create a point set object.

```
  typedef itk::PointSet< PixelType, 3 > PointSetType;
  PointSetType::Pointer  pointSet = PointSetType::New();
```

The following code is generating a sphere and assigning RGB values to the points. The components of the RGB values in this example are computed to represent the position of the points.

```
  PointSetType::PixelType   pixel;
  PointSetType::PointType   point;
  unsigned int pointId =  0;
  const double radius = 3.0;
```

```
for(unsigned int i=0; i<360; i++)
  {
  const double angle = i * atan(1.0) / 45.0;
  point[0] = radius * sin( angle );
  point[1] = radius * cos( angle );
  point[2] = 1.0;
  pixel.SetRed(    point[0] * 2.0 );
  pixel.SetGreen(  point[1] * 2.0 );
  pixel.SetBlue(   point[2] * 2.0 );
  pointSet->SetPoint( pointId, point );
  pointSet->SetPointData( pointId, pixel );
  pointId++;
  }
```

All the points on the PointSet are visited using the following code.

```
typedef  PointSetType::PointsContainer::ConstIterator     PointIterator;
PointIterator pointIterator = pointSet->GetPoints()->Begin();
PointIterator pointEnd      = pointSet->GetPoints()->End();
while( pointIterator != pointEnd )
  {
  PointSetType::PointType point = pointIterator.Value();
  std::cout << point << std::endl;
  ++pointIterator;
  }
```

Note that here the `ConstIterator` was used instead of the `Iterator` since the pixel values are not expected to be modified. ITK supports const-correctness at the API level.

All the pixel values on the PointSet are visited using the following code.

```
typedef  PointSetType::PointDataContainer::ConstIterator     PointDataIterator;
PointDataIterator pixelIterator = pointSet->GetPointData()->Begin();
PointDataIterator pixelEnd      = pointSet->GetPointData()->End();
while( pixelIterator != pixelEnd )
  {
  PointSetType::PixelType pixel = pixelIterator.Value();
  std::cout << pixel << std::endl;
  ++pixelIterator;
  }
```

Again, please note the use of the `ConstIterator` instead of the `Iterator`.

4.2.5 Vectors as Pixel Type

The source code for this section can be found in the file
`Examples/DataRepresentation/Mesh/PointSetWithVectors.cxx`.

This example illustrates how a point set can be parameterized to manage a particular pixel type. It is quite common to associate vector values with points for producing geometric representations. The following code shows how vector values can be used as pixel type on the PointSet class. The `itk::Vector` class is used here as the pixel type. This class is appropriate for representing the relative position between two points. It could then be used to manage displacements, for example.

In order to use the vector class it is necessary to include its header file along with the header of the point set.

```
#include "itkVector.h"
#include "itkPointSet.h"
```

The Vector class is templated over the type used to represent the spatial coordinates and over the space dimension. Since the PixelType is independent of the PointType, we are free to select any dimension for the vectors to be used as pixel type. However, for the sake of producing an interesting example, we will use vectors that represent displacements of the points in the PointSet. Those vectors are then selected to be of the same dimension as the PointSet.

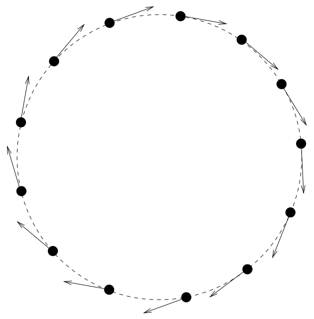


Figure 4.2: Vectors as PixelType.

```
  const unsigned int Dimension = 3;
  typedef itk::Vector< float, Dimension >    PixelType;
```

Then we use the PixelType (which are actually Vectors) to instantiate the PointSet type and subsequently create a PointSet object.

```
  typedef itk::PointSet< PixelType, Dimension > PointSetType;
  PointSetType::Pointer  pointSet = PointSetType::New();
```

The following code is generating a sphere and assigning vector values to the points. The components of the vectors in this example are computed to represent the tangents to the circle as shown in Figure 4.2.

```
  PointSetType::PixelType   tangent;
  PointSetType::PointType   point;

  unsigned int pointId =  0;
  const double radius = 300.0;

  for(unsigned int i=0; i<360; i++)
```

```
    {
    const double angle = i * atan(1.0) / 45.0;
    point[0] = radius * sin( angle );
    point[1] = radius * cos( angle );
    point[2] = 1.0;   // flat on the Z plane
    tangent[0] =  cos(angle);
    tangent[1] = -sin(angle);
    tangent[2] = 0.0;  // flat on the Z plane
    pointSet->SetPoint( pointId, point );
    pointSet->SetPointData( pointId, tangent );
    pointId++;
    }
```

We can now visit all the points and use the vector on the pixel values to apply a displacement on the points. This is along the spirit of what a deformable model could do at each one of its iterations.

```
  typedef  PointSetType::PointDataContainer::ConstIterator PointDataIterator;
  PointDataIterator pixelIterator = pointSet->GetPointData()->Begin();
  PointDataIterator pixelEnd      = pointSet->GetPointData()->End();

  typedef  PointSetType::PointsContainer::Iterator     PointIterator;
  PointIterator pointIterator = pointSet->GetPoints()->Begin();
  PointIterator pointEnd      = pointSet->GetPoints()->End();

  while( pixelIterator != pixelEnd  && pointIterator != pointEnd )
    {
    pointIterator.Value() = pointIterator.Value() + pixelIterator.Value();
    ++pixelIterator;
    ++pointIterator;
    }
```

Note that the `ConstIterator` was used here instead of the normal `Iterator` since the pixel values are only intended to be read and not modified. ITK supports const-correctness at the API level.

The `itk::Vector` class has overloaded the + operator with the `itk::Point`. In other words, vectors can be added to points in order to produce new points. This property is exploited in the center of the loop in order to update the points positions with a single statement.

We can finally visit all the points and print out the new values

```
  pointIterator = pointSet->GetPoints()->Begin();
  pointEnd      = pointSet->GetPoints()->End();
  while( pointIterator != pointEnd )
    {
    std::cout << pointIterator.Value() << std::endl;
    ++pointIterator;
    }
```

Note that `itk::Vector` is not the appropriate class for representing normals to surfaces and gradients of functions. This is due to the way in which vectors behave under affine transforms. ITK has a specific class for representing normals and function gradients. This is the `itk::CovariantVector` class.

4.2.6 Normals as Pixel Type

The source code for this section can be found in the file
`Examples/DataRepresentation/Mesh/PointSetWithCovariantVectors.cxx`.

It is common to represent geometric object by using points on their surfaces and normals associated with those points. This structure can be easily instantiated with the `itk::PointSet` class.

The natural class for representing normals to surfaces and gradients of functions is the `itk::CovariantVector`. A covariant vector differs from a vector in the way they behave under affine transforms, in particular under anisotropic scaling. If a covariant vector represents the gradient of a function, the transformed covariant vector will still be the valid gradient of the transformed function, a property which would not hold with a regular vector.

The following code shows how vector values can be used as pixel type on the PointSet class. The CovariantVector class is used here as the pixel type. The example illustrates how a deformable model could move under the influence of the gradient of potential function.

In order to use the CovariantVector class it is necessary to include its header file along with the header of the point set.

```
#include "itkCovariantVector.h"
#include "itkPointSet.h"
```

The CovariantVector class is templated over the type used to represent the spatial coordinates and over the space dimension. Since the PixelType is independent of the PointType, we are free to select any dimension for the covariant vectors to be used as pixel type. However, we want to illustrate here the spirit of a deformable model. It is then required for the vectors representing gradients to be of the same dimension as the points in space.

```
  const unsigned int Dimension = 3;
  typedef itk::CovariantVector< float, Dimension >    PixelType;
```

Then we use the PixelType (which are actually CovariantVectors) to instantiate the PointSet type and subsequently create a PointSet object.

```
typedef itk::PointSet< PixelType, Dimension > PointSetType;
PointSetType::Pointer  pointSet = PointSetType::New();
```

The following code generates a sphere and assigns gradient values to the points. The components of the CovariantVectors in this example are computed to represent the normals to the circle.

```
PointSetType::PixelType   gradient;
PointSetType::PointType   point;

unsigned int pointId =  0;
const double radius = 300.0;

for(unsigned int i=0; i<360; i++)
  {
  const double angle = i * atan(1.0) / 45.0;
  point[0] = radius * sin( angle );
  point[1] = radius * cos( angle );
  point[2] = 1.0;   // flat on the Z plane
  gradient[0] =  sin(angle);
  gradient[1] =  cos(angle);
  gradient[2] = 0.0;  // flat on the Z plane
  pointSet->SetPoint( pointId, point );
  pointSet->SetPointData( pointId, gradient );
  pointId++;
  }
```

We can now visit all the points and use the vector on the pixel values to apply a deformation on the points by following the gradient of the function. This is along the spirit of what a deformable model could do at each one of its iterations. To be more formal we should use the function gradients as forces and multiply them by local stress tensors in order to obtain local deformations. The resulting deformations would finally be used to apply displacements on the points. However, to shorten the example, we will ignore this complexity for the moment.

```
typedef  PointSetType::PointDataContainer::ConstIterator PointDataIterator;
PointDataIterator pixelIterator = pointSet->GetPointData()->Begin();
PointDataIterator pixelEnd      = pointSet->GetPointData()->End();

typedef  PointSetType::PointsContainer::Iterator     PointIterator;
PointIterator pointIterator = pointSet->GetPoints()->Begin();
PointIterator pointEnd      = pointSet->GetPoints()->End();

while( pixelIterator != pixelEnd  && pointIterator != pointEnd )
  {
  PointSetType::PointType point    = pointIterator.Value();
  PointSetType::PixelType gradient = pixelIterator.Value();
```

```
    for(unsigned int i=0; i<Dimension; i++)
      {
      point[i] += gradient[i];
      }
    pointIterator.Value() = point;
    ++pixelIterator;
    ++pointIterator;
    }
```

The CovariantVector class does not overload the + operator with the `itk::Point`. In other words, CovariantVectors can not be added to points in order to get new points. Further, since we are ignoring physics in the example, we are also forced to do the illegal addition manually between the components of the gradient and the coordinates of the points.

Note that the absence of some basic operators on the ITK geometry classes is completely intentional with the aim of preventing the incorrect use of the mathematical concepts they represent.

4.3 Mesh

4.3.1 Creating a Mesh

The source code for this section can be found in the file
`Examples/DataRepresentation/Mesh/Mesh1.cxx`.

The `itk::Mesh` class is intended to represent shapes in space. It derives from the `itk::PointSet` class and hence inherits all the functionality related to points and access to the pixel-data associated with the points. The mesh class is also n-dimensional which allows a great flexibility in its use.

In practice a Mesh class can be seen as a PointSet to which cells (also known as elements) of many different dimensions and shapes have been added. Cells in the mesh are defined in terms of the existing points using their point-identifiers.

In the same way as for the PointSet, two basic styles of Meshes are available in ITK. They are referred to as *static* and *dynamic*. The first one is used when the number of points in the set can be known in advance and it is not expected to change as a consequence of the manipulations performed on the set. The dynamic style, on the other hand, is intended to support insertion and removal of points in an efficient manner. The reason for making the distinction between the two styles is to facilitate fine tuning its behavior with the aim of optimizing performance and memory management. In the case of the Mesh, the dynamic/static aspect is extended to the management of cells.

In order to use the Mesh class, its header file should be included.

```
#include "itkMesh.h"
```

Then, the type associated with the points must be selected and used for instantiating the Mesh type.

```
typedef   float   PixelType;
```

The Mesh type extensively uses the capabilities provided by Generic Programming. In particular the Mesh class is parameterized over the PixelType and the dimension of the space. PixelType is the type of the value associated with every point just as is done with the PointSet. The following line illustrates a typical instantiation of the Mesh.

```
const unsigned int Dimension = 3;
typedef itk::Mesh< PixelType, Dimension >   MeshType;
```

Meshes are expected to take large amounts of memory. For this reason they are reference counted objects and are managed using SmartPointers. The following line illustrates how a mesh is created by invoking the `New()` method of the MeshType and the resulting object is assigned to a `itk::SmartPointer`.

```
MeshType::Pointer  mesh = MeshType::New();
```

The management of points in the Mesh is exactly the same as in the PointSet. The type point associated with the mesh can be obtained through the `PointType` trait. The following code shows the creation of points compatible with the mesh type defined above and the assignment of values to its coordinates.

```
MeshType::PointType p0;
MeshType::PointType p1;
MeshType::PointType p2;
MeshType::PointType p3;

p0[0]= -1.0; p0[1]= -1.0; p0[2]= 0.0; // first  point ( -1, -1, 0 )
p1[0]=  1.0; p1[1]= -1.0; p1[2]= 0.0; // second point (  1, -1, 0 )
p2[0]=  1.0; p2[1]=  1.0; p2[2]= 0.0; // third  point (  1,  1, 0 )
p3[0]= -1.0; p3[1]=  1.0; p3[2]= 0.0; // fourth point ( -1,  1, 0 )
```

The points can now be inserted in the Mesh using the `SetPoint()` method. Note that points are copied into the mesh structure. This means that the local instances of the points can now be modified without affecting the Mesh content.

```
mesh->SetPoint( 0, p0 );
mesh->SetPoint( 1, p1 );
mesh->SetPoint( 2, p2 );
mesh->SetPoint( 3, p3 );
```

The current number of points in the Mesh can be queried with the `GetNumberOfPoints()` method.

```
std::cout << "Points = " << mesh->GetNumberOfPoints() << std::endl;
```

The points can now be efficiently accessed using the Iterator to the PointsContainer as it was done in the previous section for the PointSet. First, the point iterator type is extracted through the mesh traits.

```
typedef MeshType::PointsContainer::Iterator     PointsIterator;
```

A point iterator is initialized to the first point with the `Begin()` method of the PointsContainer.

```
PointsIterator  pointIterator = mesh->GetPoints()->Begin();
```

The `++` operator on the iterator is now used to advance from one point to the next. The actual value of the Point to which the iterator is pointing can be obtained with the `Value()` method. The loop for walking through all the points is controlled by comparing the current iterator with the iterator returned by the `End()` method of the PointsContainer. The following lines illustrate the typical loop for walking through the points.

```
PointsIterator end = mesh->GetPoints()->End();
while( pointIterator != end )
  {
  MeshType::PointType p = pointIterator.Value();  // access the point
  std::cout << p << std::endl;                    // print the point
  ++pointIterator;                                // advance to next point
  }
```

4.3.2 Inserting Cells

The source code for this section can be found in the file
`Examples/DataRepresentation/Mesh/Mesh2.cxx`.

A `itk::Mesh` can contain a variety of cell types. Typical cells are the `itk::LineCell`, `itk::TriangleCell`, `itk::QuadrilateralCell` and `itk::TetrahedronCell`. Additional flexibility is provided for managing cells at the price of a bit more of complexity than in the case of point management.

The following code creates a polygonal line in order to illustrate the simplest case of cell management in a Mesh. The only cell type used here is the LineCell. The header file of this class has to be included.

```
#include "itkLineCell.h"
```

In order to be consistent with the Mesh, cell types have to be configured with a number of custom types taken from the mesh traits. The set of traits relevant to cells are packaged by the Mesh class into the `CellType` trait. This trait needs to be passed to the actual cell types at the moment of their instantiation. The following line shows how to extract the Cell traits from the Mesh type.

```
typedef MeshType::CellType                CellType;
```

The LineCell type can now be instantiated using the traits taken from the Mesh.

```
typedef itk::LineCell< CellType >         LineType;
```

The main difference in the way cells and points are managed by the Mesh is that points are stored by copy on the PointsContainer while cells are stored in the CellsContainer using pointers. The reason for using pointers is that cells use C++ polymorphism on the mesh. This means that the mesh is only aware of having pointers to a generic cell which is the base class of all the specific cell types. This architecture makes it possible to combine different cell types in the same mesh. Points, on the other hand, are of a single type and have a small memory footprint, which makes it efficient to copy them directly into the container.

Managing cells by pointers add another level of complexity to the Mesh since it is now necessary to establish a protocol to make clear who is responsible for allocating and releasing the cells' memory. This protocol is implemented in the form of a specific type of pointer called the `CellAutoPointer`. This pointer, based on the `itk::AutoPointer`, differs in many respects from the SmartPointer. The CellAutoPointer has an internal pointer to the actual object and a boolean flag that indicates if the CellAutoPointer is responsible for releasing the cell memory whenever the time comes for its own destruction. It is said that a `CellAutoPointer` *owns* the cell when it is responsible for its destruction. Many CellAutoPointer can point to the same cell but at any given time, only **one** CellAutoPointer can own the cell.

The `CellAutoPointer` trait is defined in the MeshType and can be extracted as illustrated in the following line.

```
typedef CellType::CellAutoPointer         CellAutoPointer;
```

Note that the CellAutoPointer is pointing to a generic cell type. It is not aware of the actual type of the cell, which can be for example LineCell, TriangleCell or TetrahedronCell. This fact will influence the way in which we access cells later on.

At this point we can actually create a mesh and insert some points on it.

```
MeshType::Pointer  mesh = MeshType::New();

MeshType::PointType p0;
MeshType::PointType p1;
```

```
  MeshType::PointType p2;

  p0[0] = -1.0; p0[1] = 0.0; p0[2] = 0.0;
  p1[0] =  1.0; p1[1] = 0.0; p1[2] = 0.0;
  p2[0] =  1.0; p2[1] = 1.0; p2[2] = 0.0;

  mesh->SetPoint( 0, p0 );
  mesh->SetPoint( 1, p1 );
  mesh->SetPoint( 2, p2 );
```

The following code creates two CellAutoPointers and initializes them with newly created cell objects. The actual cell type created in this case is LineCell. Note that cells are created with the normal `new` C++ operator. The CellAutoPointer takes ownership of the received pointer by using the method `TakeOwnership()`. Even though this may seem verbose, it is necessary in order to make it explicit from the code that the responsibility of memory release is assumed by the AutoPointer.

```
  CellAutoPointer line0;
  CellAutoPointer line1;

  line0.TakeOwnership(  new LineType  );
  line1.TakeOwnership(  new LineType  );
```

The LineCells should now be associated with points in the mesh. This is done using the identifiers assigned to points when they were inserted in the mesh. Every cell type has a specific number of points that must be associated with it.[4] For example a LineCell requires two points, a TriangleCell requires three and a TetrahedronCell requires four. Cells use an internal numbering system for points. It is simply an index in the range $\{0, NumberOfPoints - 1\}$. The association of points and cells is done by the `SetPointId()` method which requires the user to provide the internal index of the point in the cell and the corresponding PointIdentifier in the Mesh. The internal cell index is the first parameter of `SetPointId()` while the mesh point-identifier is the second.

```
  line0->SetPointId( 0, 0 ); // line between points 0 and 1
  line0->SetPointId( 1, 1 );

  line1->SetPointId( 0, 1 ); // line between points 1 and 2
  line1->SetPointId( 1, 2 );
```

Cells are inserted in the mesh using the `SetCell()` method. It requires an identifier and the AutoPointer to the cell. The Mesh will take ownership of the cell to which the AutoPointer is pointing. This is done internally by the `SetCell()` method. In this way, the destruction of the CellAutoPointer will not induce the destruction of the associated cell.

[4]Some cell types like polygons have a variable number of points associated with them.

```
mesh->SetCell( 0, line0 );
mesh->SetCell( 1, line1 );
```

After serving as an argument of the `SetCell()` method, a CellAutoPointer no longer holds ownership of the cell. It is important not to use this same CellAutoPointer again as argument to `SetCell()` without first securing ownership of another cell.

The number of Cells currently inserted in the mesh can be queried with the `GetNumberOfCells()` method.

```
std::cout << "Cells  = " << mesh->GetNumberOfCells()  << std::endl;
```

In a way analogous to points, cells can be accessed using Iterators to the CellsContainer in the mesh. The trait for the cell iterator can be extracted from the mesh and used to define a local type.

```
typedef MeshType::CellsContainer::Iterator  CellIterator;
```

Then the iterators to the first and past-end cell in the mesh can be obtained respectively with the `Begin()` and `End()` methods of the CellsContainer. The CellsContainer of the mesh is returned by the `GetCells()` method.

```
CellIterator  cellIterator = mesh->GetCells()->Begin();
CellIterator  end          = mesh->GetCells()->End();
```

Finally a standard loop is used to iterate over all the cells. Note the use of the `Value()` method used to get the actual pointer to the cell from the CellIterator. Note also that the values returned are pointers to the generic CellType. These pointers have to be down-casted in order to be used as actual LineCell types. Safe down-casting is performed with the `dynamic_cast` operator which will throw an exception if the conversion cannot be safely performed.

```
while( cellIterator != end )
  {
  MeshType::CellType * cellptr = cellIterator.Value();
  LineType * line = dynamic_cast<LineType *>( cellptr );
  std::cout << line->GetNumberOfPoints() << std::endl;
  ++cellIterator;
  }
```

4.3.3 Managing Data in Cells

The source code for this section can be found in the file
`Examples/DataRepresentation/Mesh/Mesh3.cxx`.

In the same way that custom data can be associated with points in the mesh, it is also possible to associate custom data with cells. The type of the data associated with the cells can be different from the data type associated with points. By default, however, these two types are the same. The following example illustrates how to access data associated with cells. The approach is analogous to the one used to access point data.

Consider the example of a mesh containing lines on which values are associated with each line. The mesh and cell header files should be included first.

```
#include "itkMesh.h"
#include "itkLineCell.h"
```

Then the PixelType is defined and the mesh type is instantiated with it.

```
  typedef float                            PixelType;
  typedef itk::Mesh< PixelType, 2 >        MeshType;
```

The `itk::LineCell` type can now be instantiated using the traits taken from the Mesh.

```
  typedef MeshType::CellType               CellType;
  typedef itk::LineCell< CellType >        LineType;
```

Let's now create a Mesh and insert some points into it. Note that the dimension of the points matches the dimension of the Mesh. Here we insert a sequence of points that look like a plot of the log() function.

```
  MeshType::Pointer  mesh = MeshType::New();

  typedef MeshType::PointType PointType;
  PointType point;

  const unsigned int numberOfPoints = 10;
  for(unsigned int id=0; id<numberOfPoints; id++)
    {
    point[0] = static_cast<PointType::ValueType>( id ); // x
    point[1] = log( static_cast<double>( id ) );        // y
    mesh->SetPoint( id, point );
    }
```

A set of line cells is created and associated with the existing points by using point identifiers. In this simple case, the point identifiers can be deduced from cell identifiers since the line cells are ordered in the same way.

```
  CellType::CellAutoPointer line;
```

```
const unsigned int numberOfCells = numberOfPoints-1;
for(unsigned int cellId=0; cellId<numberOfCells; cellId++)
  {
  line.TakeOwnership(  new LineType  );
  line->SetPointId( 0, cellId   ); // first point
  line->SetPointId( 1, cellId+1 ); // second point
  mesh->SetCell( cellId, line );   // insert the cell
  }
```

Data associated with cells is inserted in the itk::Mesh by using the SetCellData() method. It requires the user to provide an identifier and the value to be inserted. The identifier should match one of the inserted cells. In this simple example, the square of the cell identifier is used as cell data. Note the use of static_cast to PixelType in the assignment.

```
for(unsigned int cellId=0; cellId<numberOfCells; cellId++)
  {
  mesh->SetCellData( cellId, static_cast<PixelType>( cellId * cellId ) );
  }
```

Cell data can be read from the Mesh with the GetCellData() method. It requires the user to provide the identifier of the cell for which the data is to be retrieved. The user should provide also a valid pointer to a location where the data can be copied.

```
for(unsigned int cellId=0; cellId<numberOfCells; cellId++)
  {
  PixelType value;
  mesh->GetCellData( cellId, &value );
  std::cout << "Cell " << cellId << " = " << value << std::endl;
  }
```

Neither SetCellData() or GetCellData() are efficient ways to access cell data. More efficient access to cell data can be achieved by using the Iterators built into the CellDataContainer.

```
typedef MeshType::CellDataContainer::ConstIterator CellDataIterator;
```

Note that the ConstIterator is used here because the data is only going to be read. This approach is exactly the same already illustrated for getting access to point data. The iterator to the first cell data item can be obtained with the Begin() method of the CellDataContainer. The past-end iterator is returned by the End() method. The cell data container itself can be obtained from the mesh with the method GetCellData().

```
CellDataIterator cellDataIterator  = mesh->GetCellData()->Begin();
CellDataIterator end               = mesh->GetCellData()->End();
```

Finally a standard loop is used to iterate over all the cell data entries. Note the use of the `Value()` method used to get the actual value of the data entry. `PixelType` elements are copied into the local variable `cellValue`.

```
  while( cellDataIterator != end )
    {
    PixelType cellValue = cellDataIterator.Value();
    std::cout << cellValue << std::endl;
    ++cellDataIterator;
    }
```

4.3.4 Customizing the Mesh

The source code for this section can be found in the file
`Examples/DataRepresentation/Mesh/MeshTraits.cxx`.

This section illustrates the full power of Generic Programming. This is sometimes perceived as *too much of a good thing*!

The toolkit has been designed to offer flexibility while keeping the complexity of the code to a moderate level. This is achieved in the Mesh by hiding most of its parameters and defining reasonable defaults for them.

The generic concept of a mesh integrates many different elements. It is possible in principle to use independent types for every one of such elements. The mechanism used in generic programming for specifying the many different types involved in a concept is called *traits*. They are basically the list of all types that interact with the current class.

The `itk::Mesh` is templated over three parameters. So far only two of them have been discussed, namely the `PixelType` and the `Dimension`. The third parameter is a class providing the set of traits required by the mesh. When the third parameter is omitted a default class is used. This default class is the `itk::DefaultStaticMeshTraits`. If you want to customize the types used by the mesh, the way to proceed is to modify the default traits and provide them as the third parameter of the Mesh class instantiation.

There are two ways of achieving this. The first is to use the existing DefaultStaticMeshTraits class. This class is itself templated over six parameters. Customizing those parameters could provide enough flexibility to define a very specific kind of mesh. The second way is to write a traits class from scratch, in which case the easiest way to proceed is to copy the DefaultStaticMeshTraits into another file and edit its content. Only the first approach is illustrated here. The second is discouraged unless you are familiar with Generic Programming, feel comfortable with C++ templates and have access to an abundant supply of (Columbian) coffee.

The first step in customizing the mesh is to include the header file of the Mesh and its static traits.

```
#include "itkMesh.h"
```

```
#include "itkDefaultStaticMeshTraits.h"
```

Then the MeshTraits class is instantiated by selecting the types of each one of its six template arguments. They are in order

PixelType. The type associated with every point.

PointDimension. The dimension of the space in which the mesh is embedded.

MaxTopologicalDimension. The highest dimension of the mesh cells.

CoordRepType. The type used to represent space coordinates.

InterpolationWeightType. The type used to represent interpolation weights.

CellPixelType. The type associated with every cell.

Let's define types and values for each one of those elements. For example the following code will use points in 3D space as nodes of the Mesh. The maximum dimension of the cells will be two which means that this is a 2D manifold better know as a *surface*. The data type associated with points is defined to be a four-dimensional vector. This type could represent values of membership for a four-classes segmentation method. The value selected for the cells are 4×3 matrices which could have for example the derivative of the membership values with respect to coordinates in space. Finally a `double` type is selected for representing space coordinates on the mesh points and also for the weight used for interpolating values.

```
  const unsigned int PointDimension = 3;
  const unsigned int MaxTopologicalDimension = 2;

  typedef itk::Vector<double,4>                   PixelType;
  typedef itk::Matrix<double,4,3>                 CellDataType;

  typedef double CoordinateType;
  typedef double InterpolationWeightType;

  typedef itk::DefaultStaticMeshTraits<
            PixelType, PointDimension, MaxTopologicalDimension,
            CoordinateType, InterpolationWeightType, CellDataType > MeshTraits;

  typedef itk::Mesh< PixelType, PointDimension, MeshTraits > MeshType;
```

The `itk::LineCell` type can now be instantiated using the traits taken from the Mesh.

```
  typedef MeshType::CellType                 CellType;
  typedef itk::LineCell< CellType >          LineType;
```

Let's now create an Mesh and insert some points on it. Note that the dimension of the points matches the dimension of the Mesh. Here we insert a sequence of points that look like a plot of the $log()$ function.

```
MeshType::Pointer  mesh = MeshType::New();

typedef MeshType::PointType PointType;
PointType point;

const unsigned int numberOfPoints = 10;
for(unsigned int id=0; id<numberOfPoints; id++)
  {
  point[0] = 1.565;   // Initialize points here
  point[1] = 3.647;   // with arbitrary values
  point[2] = 4.129;
  mesh->SetPoint( id, point );
  }
```

A set of line cells is created and associated with the existing points by using point identifiers. In this simple case, the point identifiers can be deduced from cell identifiers since the line cells are ordered in the same way. Note that in the code above, the values assigned to point components are arbitrary. In a more realistic example, those values would be computed from another source.

```
CellType::CellAutoPointer line;
const unsigned int numberOfCells = numberOfPoints-1;
for(unsigned int cellId=0; cellId<numberOfCells; cellId++)
  {
  line.TakeOwnership(  new LineType  );
  line->SetPointId( 0, cellId   ); // first point
  line->SetPointId( 1, cellId+1 ); // second point
  mesh->SetCell( cellId, line );   // insert the cell
  }
```

Data associated with cells is inserted in the Mesh by using the `SetCellData()` method. It requires the user to provide an identifier and the value to be inserted. The identifier should match one of the inserted cells. In this simple example, the square of the cell identifier is used as cell data. Note the use of `static_cast` to `PixelType` in the assignment.

```
for(unsigned int cellId=0; cellId<numberOfCells; cellId++)
  {
  CellDataType value;
  mesh->SetCellData( cellId, value );
  }
```

Cell data can be read from the Mesh with the `GetCellData()` method. It requires the user to provide the identifier of the cell for which the data is to be retrieved. The user should provide also a valid pointer to a location where the data can be copied.

```
for(unsigned int cellId=0; cellId<numberOfCells; cellId++)
  {
  CellDataType value;
  mesh->GetCellData( cellId, &value );
  std::cout << "Cell " << cellId << " = " << value << std::endl;
  }
```

Neither `SetCellData()` or `GetCellData()` are efficient ways to access cell data. Efficient access to cell data can be achieved by using the Iterators built into the CellDataContainer.

```
typedef MeshType::CellDataContainer::ConstIterator CellDataIterator;
```

Note that the `ConstIterator` is used here because the data is only going to be read. This approach is exactly the same already illustrated for getting access to point data. The iterator to the first cell data item can be obtained with the `Begin()` method of the CellDataContainer. The past-end iterator is returned by the `End()` method. The cell data container itself can be obtained from the mesh with the method `GetCellData()`.

```
CellDataIterator cellDataIterator  = mesh->GetCellData()->Begin();
CellDataIterator end               = mesh->GetCellData()->End();
```

Finally a standard loop is used to iterate over all the cell data entries. Note the use of the `Value()` method used to get the actual value of the data entry. `PixelType` elements are returned by copy.

```
while( cellDataIterator != end )
  {
  CellDataType cellValue = cellDataIterator.Value();
  std::cout << cellValue << std::endl;
  ++cellDataIterator;
  }
```

4.3.5 Topology and the K-Complex

The source code for this section can be found in the file
`Examples/DataRepresentation/Mesh/MeshKComplex.cxx`.

The `itk::Mesh` class supports the representation of formal topologies. In particular the concept of *K-Complex* can be correctly represented in the Mesh. An informal definition of K-Complex may be as follows: a K-Complex is a topological structure in which for every cell of dimension N, its boundary faces which are cells of dimension $N-1$ also belong to the structure.

This section illustrates how to instantiate a K-Complex structure using the mesh. The example structure is composed of one tetrahedron, its four triangle faces, its six line edges and its four vertices.

The header files of all the cell types involved should be loaded along with the header file of the mesh class.

```
#include "itkMesh.h"
#include "itkVertexCell.h"
#include "itkLineCell.h"
#include "itkTriangleCell.h"
#include "itkTetrahedronCell.h"
```

Then the PixelType is defined and the mesh type is instantiated with it. Note that the dimension of the space is three in this case.

```
  typedef float                            PixelType;
  typedef itk::Mesh< PixelType, 3 >        MeshType;
```

The cell type can now be instantiated using the traits taken from the Mesh.

```
  typedef MeshType::CellType               CellType;
  typedef itk::VertexCell< CellType >      VertexType;
  typedef itk::LineCell< CellType >        LineType;
  typedef itk::TriangleCell< CellType >    TriangleType;
  typedef itk::TetrahedronCell< CellType > TetrahedronType;
```

The mesh is created and the points associated with the vertices are inserted. Note that there is an important distinction between the points in the mesh and the `itk::VertexCell` concept. A VertexCell is a cell of dimension zero. Its main difference as compared to a point is that the cell can be aware of neighborhood relationships with other cells. Points are not aware of the existence of cells. In fact, from the pure topological point of view, the coordinates of points in the mesh are completely irrelevant. They may as well be absent from the mesh structure altogether. VertexCells on the other hand are necessary to represent the full set of neighborhood relationships on the K-Complex.

The geometrical coordinates of the nodes of a regular tetrahedron can be obtained by taking every other node from a regular cube.

```
  MeshType::Pointer  mesh = MeshType::New();

  MeshType::PointType   point0;
```

```
MeshType::PointType   point1;
MeshType::PointType   point2;
MeshType::PointType   point3;

point0[0] = -1; point0[1] = -1; point0[2] = -1;
point1[0] =  1; point1[1] =  1; point1[2] = -1;
point2[0] =  1; point2[1] = -1; point2[2] =  1;
point3[0] = -1; point3[1] =  1; point3[2] =  1;

mesh->SetPoint( 0, point0 );
mesh->SetPoint( 1, point1 );
mesh->SetPoint( 2, point2 );
mesh->SetPoint( 3, point3 );
```

We proceed now to create the cells, associate them with the points and insert them on the mesh. Starting with the tetrahedron we write the following code.

```
CellType::CellAutoPointer cellpointer;

cellpointer.TakeOwnership( new TetrahedronType );
cellpointer->SetPointId( 0, 0 );
cellpointer->SetPointId( 1, 1 );
cellpointer->SetPointId( 2, 2 );
cellpointer->SetPointId( 3, 3 );
mesh->SetCell( 0, cellpointer );
```

Four triangular faces are created and associated with the mesh now. The first triangle connects points 0,1,2.

```
cellpointer.TakeOwnership( new TriangleType );
cellpointer->SetPointId( 0, 0 );
cellpointer->SetPointId( 1, 1 );
cellpointer->SetPointId( 2, 2 );
mesh->SetCell( 1, cellpointer );
```

The second triangle connects points 0, 2, 3

```
cellpointer.TakeOwnership( new TriangleType );
cellpointer->SetPointId( 0, 0 );
cellpointer->SetPointId( 1, 2 );
cellpointer->SetPointId( 2, 3 );
mesh->SetCell( 2, cellpointer );
```

The third triangle connects points 0, 3, 1

```
  cellpointer.TakeOwnership( new TriangleType );
  cellpointer->SetPointId( 0, 0 );
  cellpointer->SetPointId( 1, 3 );
  cellpointer->SetPointId( 2, 1 );
  mesh->SetCell( 3, cellpointer );
```

The fourth triangle connects points 3, 2, 1

```
  cellpointer.TakeOwnership( new TriangleType );
  cellpointer->SetPointId( 0, 3 );
  cellpointer->SetPointId( 1, 2 );
  cellpointer->SetPointId( 2, 1 );
  mesh->SetCell( 4, cellpointer );
```

Note how the `CellAutoPointer` is reused every time. Reminder: the `itk::AutoPointer` loses ownership of the cell when it is passed as an argument of the `SetCell()` method. The AutoPointer is attached to a new cell by using the `TakeOwnership()` method.

The construction of the K-Complex continues now with the creation of the six lines on the tetrahedron edges.

```
  cellpointer.TakeOwnership( new LineType );
  cellpointer->SetPointId( 0, 0 );
  cellpointer->SetPointId( 1, 1 );
  mesh->SetCell( 5, cellpointer );

  cellpointer.TakeOwnership( new LineType );
  cellpointer->SetPointId( 0, 1 );
  cellpointer->SetPointId( 1, 2 );
  mesh->SetCell( 6, cellpointer );

  cellpointer.TakeOwnership( new LineType );
  cellpointer->SetPointId( 0, 2 );
  cellpointer->SetPointId( 1, 0 );
  mesh->SetCell( 7, cellpointer );

  cellpointer.TakeOwnership( new LineType );
  cellpointer->SetPointId( 0, 1 );
  cellpointer->SetPointId( 1, 3 );
  mesh->SetCell( 8, cellpointer );

  cellpointer.TakeOwnership( new LineType );
  cellpointer->SetPointId( 0, 3 );
  cellpointer->SetPointId( 1, 2 );
  mesh->SetCell( 9, cellpointer );

  cellpointer.TakeOwnership( new LineType );
```

```
cellpointer->SetPointId( 0, 3 );
cellpointer->SetPointId( 1, 0 );
mesh->SetCell( 10, cellpointer );
```

Finally the zero dimensional cells represented by the itk::VertexCell are created and inserted in the mesh.

```
cellpointer.TakeOwnership( new VertexType );
cellpointer->SetPointId( 0, 0 );
mesh->SetCell( 11, cellpointer );

cellpointer.TakeOwnership( new VertexType );
cellpointer->SetPointId( 0, 1 );
mesh->SetCell( 12, cellpointer );

cellpointer.TakeOwnership( new VertexType );
cellpointer->SetPointId( 0, 2 );
mesh->SetCell( 13, cellpointer );

cellpointer.TakeOwnership( new VertexType );
cellpointer->SetPointId( 0, 3 );
mesh->SetCell( 14, cellpointer );
```

At this point the Mesh contains four points and fifteen cells enumerated from 0 to 14. The points can be visited using PointContainer iterators

```
typedef MeshType::PointsContainer::ConstIterator  PointIterator;
PointIterator pointIterator = mesh->GetPoints()->Begin();
PointIterator pointEnd      = mesh->GetPoints()->End();

while( pointIterator != pointEnd )
  {
  std::cout << pointIterator.Value() << std::endl;
  ++pointIterator;
  }
```

The cells can be visited using CellsContainer iterators

```
typedef MeshType::CellsContainer::ConstIterator  CellIterator;

CellIterator cellIterator = mesh->GetCells()->Begin();
CellIterator cellEnd      = mesh->GetCells()->End();

while( cellIterator != cellEnd )
  {
  CellType * cell = cellIterator.Value();
```

```
    std::cout << cell->GetNumberOfPoints() << std::endl;
    ++cellIterator;
    }
```

Note that cells are stored as pointer to a generic cell type that is the base class of all the specific cell classes. This means that at this level we can only have access to the virtual methods defined in the `CellType`.

The point identifiers to which the cells have been associated can be visited using iterators defined in the `CellType` trait. The following code illustrates the use of the PointIdIterators. The `PointIdsBegin()` method returns the iterator to the first point-identifier in the cell. The `PointIdsEnd()` method returns the iterator to the past-end point-identifier in the cell.

```
typedef CellType::PointIdIterator     PointIdIterator;

PointIdIterator pointIditer = cell->PointIdsBegin();
PointIdIterator pointIdend  = cell->PointIdsEnd();

while( pointIditer != pointIdend )
  {
  std::cout << *pointIditer << std::endl;
  ++pointIditer;
  }
```

Note that the point-identifier is obtained from the iterator using the more traditional `*iterator` notation instead the `Value()` notation used by cell-iterators.

Up to here, the topology of the K-Complex is not completely defined since we have only introduced the cells. ITK allows the user to define explicitly the neighborhood relationships between cells. It is clear that a clever exploration of the point identifiers could have allowed a user to figure out the neighborhood relationships. For example, two triangle cells sharing the same two point identifiers will probably be neighbor cells. Some of the drawbacks on this implicit discovery of neighborhood relationships is that it takes computing time and that some applications may not accept the same assumptions. A specific case is surgery simulation. This application typically simulates bistoury cuts in a mesh representing an organ. A small cut in the surface may be made by specifying that two triangles are not considered to be neighbors any more.

Neighborhood relationships are represented in the mesh by the notion of *BoundaryFeature*. Every cell has an internal list of cell-identifiers pointing to other cells that are considered to be its neighbors. Boundary features are classified by dimension. For example, a line will have two boundary features of dimension zero corresponding to its two vertices. A tetrahedron will have boundary features of dimension zero, one and two, corresponding to its four vertices, six edges and four triangular faces. It is up to the user to specify the connections between the cells.

Let's take in our current example the tetrahedron cell that was associated with the cell-identifier `0` and assign to it the four vertices as boundaries of dimension zero. This is done by invoking the `SetBoundaryAssignment()` method on the Mesh class.

```
MeshType::CellIdentifier cellId = 0;  // the tetrahedron

int dimension = 0;                    // vertices

MeshType::CellFeatureIdentifier featureId = 0;

mesh->SetBoundaryAssignment( dimension, cellId, featureId++, 11 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++, 12 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++, 13 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++, 14 );
```

The `featureId` is simply a number associated with the sequence of the boundary cells of the same dimension in a specific cell. For example, the zero-dimensional features of a tetrahedron are its four vertices. Then the zero-dimensional feature-Ids for this cell will range from zero to three. The one-dimensional features of the tetrahedron are its six edges, hence its one-dimensional feature-Ids will range from zero to five. The two-dimensional features of the tetrahedron are its four triangular faces. The two-dimensional feature ids will then range from zero to three. The following table summarizes the use on indices for boundary assignments.

Dimension	CellType	FeatureId range	Cell Ids
0	VertexCell	[0:3]	{11,12,13,14}
1	LineCell	[0:5]	{5,6,7,8,9,10}
2	TriangleCell	[0:3]	{1,2,3,4}

In the code example above, the values of featureId range from zero to three. The cell identifiers of the triangle cells in this example are the numbers {1,2,3,4}, while the cell identifiers of the vertex cells are the numbers {11,12,13,14}.

Let's now assign one-dimensional boundary features of the tetrahedron. Those are the line cells with identifiers {5,6,7,8,9,10}. Note that the feature identifier is reinitialized to zero since the count is independent for each dimension.

```
cellId    = 0;  // still the tetrahedron
dimension = 1;  // one-dimensional features = edges
featureId = 0;  // reinitialize the count

mesh->SetBoundaryAssignment( dimension, cellId, featureId++,  5 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++,  6 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++,  7 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++,  8 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++,  9 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++, 10 );
```

Finally we assign the two-dimensional boundary features of the tetrahedron. These are the four triangular cells with identifiers {1,2,3,4}. The featureId is reset to zero since feature-Ids are independent on each dimension.

```
cellId    = 0;  // still the tetrahedron
dimension = 2;  // two-dimensional features = triangles
featureId = 0;  // reinitialize the count

mesh->SetBoundaryAssignment( dimension, cellId, featureId++,  1 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++,  2 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++,  3 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++,  4 );
```

At this point we can query the tetrahedron cell for information about its boundary features. For example, the number of boundary features of each dimension can be obtained with the method `GetNumberOfBoundaryFeatures()`.

```
cellId = 0; // still the tetrahedron

MeshType::CellFeatureCount n0;  // number of zero-dimensional features
MeshType::CellFeatureCount n1;  // number of  one-dimensional features
MeshType::CellFeatureCount n2;  // number of  two-dimensional features

n0 = mesh->GetNumberOfCellBoundaryFeatures( 0, cellId );
n1 = mesh->GetNumberOfCellBoundaryFeatures( 1, cellId );
n2 = mesh->GetNumberOfCellBoundaryFeatures( 2, cellId );
```

The boundary assignments can be recovered with the method `GetBoundaryAssigment()`. For example, the zero-dimensional features of the tetrahedron can be obtained with the following code.

```
dimension = 0;
for(unsigned int b0=0; b0 < n0; b0++)
  {
  MeshType::CellIdentifier id;
  bool found = mesh->GetBoundaryAssignment( dimension, cellId, b0, &id );
  if( found ) std::cout << id << std::endl;
  }
```

The following code illustrates how to set the edge boundaries for one of the triangular faces.

```
cellId     =  2;    // one of the triangles
dimension  =  1;    // boundary edges
featureId  =  0;    // start the count of features

mesh->SetBoundaryAssignment( dimension, cellId, featureId++,  7 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++,  9 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++, 10 );
```

4.3.6 Representing a PolyLine

The source code for this section can be found in the file
`Examples/DataRepresentation/Mesh/MeshPolyLine.cxx`.

This section illustrates how to represent a classical *PolyLine* structure using the `itk::Mesh`

A PolyLine only involves zero and one dimensional cells, which are represented by the `itk::VertexCell` and the `itk::LineCell`.

```
#include "itkMesh.h"
#include "itkVertexCell.h"
#include "itkLineCell.h"
```

Then the PixelType is defined and the mesh type is instantiated with it. Note that the dimension of the space is two in this case.

```
  typedef float                              PixelType;
  typedef itk::Mesh< PixelType, 2 >          MeshType;
```

The cell type can now be instantiated using the traits taken from the Mesh.

```
  typedef MeshType::CellType                 CellType;
  typedef itk::VertexCell< CellType >        VertexType;
  typedef itk::LineCell< CellType >          LineType;
```

The mesh is created and the points associated with the vertices are inserted. Note that there is an important distinction between the points in the mesh and the `itk::VertexCell` concept. A VertexCell is a cell of dimension zero. Its main difference as compared to a point is that the cell can be aware of neighborhood relationships with other cells. Points are not aware of the existence of cells. In fact, from the pure topological point of view, the coordinates of points in the mesh are completely irrelevant. They may as well be absent from the mesh structure altogether. VertexCells on the other hand are necessary to represent the full set of neighborhood relationships on the Polyline.

In this example we create a polyline connecting the four vertices of a square by using three of the square sides.

```
  MeshType::Pointer  mesh = MeshType::New();

  MeshType::PointType   point0;
  MeshType::PointType   point1;
  MeshType::PointType   point2;
  MeshType::PointType   point3;
  point0[0] = -1; point0[1] = -1;
  point1[0] =  1; point1[1] = -1;
  point2[0] =  1; point2[1] =  1;
  point3[0] = -1; point3[1] =  1;

  mesh->SetPoint( 0, point0 );
  mesh->SetPoint( 1, point1 );
  mesh->SetPoint( 2, point2 );
  mesh->SetPoint( 3, point3 );
```

We proceed now to create the cells, associate them with the points and insert them on the mesh.

```
  CellType::CellAutoPointer cellpointer;

  cellpointer.TakeOwnership( new LineType );
  cellpointer->SetPointId( 0, 0 );
  cellpointer->SetPointId( 1, 1 );
  mesh->SetCell( 0, cellpointer );

  cellpointer.TakeOwnership( new LineType );
  cellpointer->SetPointId( 0, 1 );
  cellpointer->SetPointId( 1, 2 );
  mesh->SetCell( 1, cellpointer );

  cellpointer.TakeOwnership( new LineType );
  cellpointer->SetPointId( 0, 2 );
  cellpointer->SetPointId( 1, 0 );
  mesh->SetCell( 2, cellpointer );
```

Finally the zero dimensional cells represented by the `itk::VertexCell` are created and inserted in the mesh.

```
  cellpointer.TakeOwnership( new VertexType );
  cellpointer->SetPointId( 0, 0 );
  mesh->SetCell( 3, cellpointer );

  cellpointer.TakeOwnership( new VertexType );
  cellpointer->SetPointId( 0, 1 );
  mesh->SetCell( 4, cellpointer );

  cellpointer.TakeOwnership( new VertexType );
  cellpointer->SetPointId( 0, 2 );
  mesh->SetCell( 5, cellpointer );

  cellpointer.TakeOwnership( new VertexType );
  cellpointer->SetPointId( 0, 3 );
  mesh->SetCell( 6, cellpointer );
```

At this point the Mesh contains four points and three cells. The points can be visited using PointContainer iterators

```
typedef MeshType::PointsContainer::ConstIterator  PointIterator;
PointIterator pointIterator = mesh->GetPoints()->Begin();
PointIterator pointEnd      = mesh->GetPoints()->End();

while( pointIterator != pointEnd )
  {
  std::cout << pointIterator.Value() << std::endl;
  ++pointIterator;
  }
```

The cells can be visited using CellsContainer iterators

```
typedef MeshType::CellsContainer::ConstIterator  CellIterator;

CellIterator cellIterator = mesh->GetCells()->Begin();
CellIterator cellEnd      = mesh->GetCells()->End();

while( cellIterator != cellEnd )
  {
  CellType * cell = cellIterator.Value();
  std::cout << cell->GetNumberOfPoints() << std::endl;
  ++cellIterator;
  }
```

Note that cells are stored as pointer to a generic cell type that is the base class of all the specific cell classes. This means that at this level we can only have access to the virtual methods defined in the `CellType`.

The point identifiers to which the cells have been associated can be visited using iterators defined in the `CellType` trait. The following code illustrates the use of the PointIdIterator. The `PointIdsBegin()` method returns the iterator to the first point-identifier in the cell. The `PointIdsEnd()` method returns the iterator to the past-end point-identifier in the cell.

```
  typedef CellType::PointIdIterator     PointIdIterator;

  PointIdIterator pointIditer = cell->PointIdsBegin();
  PointIdIterator pointIdend  = cell->PointIdsEnd();

  while( pointIditer != pointIdend )
    {
    std::cout << *pointIditer << std::endl;
    ++pointIditer;
    }
```

Note that the point-identifier is obtained from the iterator using the more traditional *iterator notation instead the Value() notation used by cell-iterators.

4.3.7 Simplifying Mesh Creation

The source code for this section can be found in the file
Examples/DataRepresentation/Mesh/AutomaticMesh.cxx.

The itk::Mesh class is extremely general and flexible, but there is some cost to convenience. If convenience is exactly what you need, then it is possible to get it, in exchange for some of that flexibility, by means of the itk::AutomaticTopologyMeshSource class. This class automatically generates an explicit K-Complex, based on the cells you add. It explicitly includes all boundary information, so that the resulting mesh can be easily traversed. It merges all shared edges, vertices, and faces, so no geometric feature appears more than once.

This section shows how you can use the AutomaticTopologyMeshSource to instantiate a mesh representing a K-Complex. We will first generate the same tetrahedron from Section 4.3.5, after which we will add a hollow one to illustrate some additional features of the mesh source.

The header files of all the cell types involved should be loaded along with the header file of the mesh class.

```
#include "itkMesh.h"
#include "itkVertexCell.h"
#include "itkLineCell.h"
#include "itkTriangleCell.h"
#include "itkTetrahedronCell.h"
#include "itkAutomaticTopologyMeshSource.h"
```

We then define the necessary types and instantiate the mesh source. Two new types are IdentifierType and IdentifierArrayType. Every cell in a mesh has an identifier, whose type is determined by the mesh traits. AutomaticTopologyMeshSource requires that the identifier type of all vertices and cells be unsigned long, which is already the default. However, if you created a new mesh traits class to use string tags as identifiers, the resulting mesh would not be compatible with itk::AutomaticTopologyMeshSource. An IdentifierArrayType is simply an itk::Array of IdentifierType objects.

```
  typedef float                                  PixelType;
  typedef itk::Mesh< PixelType, 3 >              MeshType;

  typedef MeshType::PointType                    PointType;
  typedef MeshType::CellType                     CellType;

  typedef itk::AutomaticTopologyMeshSource< MeshType >   MeshSourceType;
  typedef MeshSourceType::IdentifierType                 IdentifierType;
  typedef MeshSourceType::IdentifierArrayType            IdentifierArrayType;
  MeshSourceType::Pointer meshSource;

  meshSource = MeshSourceType::New();
```

Now let us generate the tetrahedron. The following line of code generates all the vertices, edges, and faces, along with the tetrahedral solid, and adds them to the mesh along with the connectivity information.

```
  meshSource->AddTetrahedron(
    meshSource->AddPoint( -1, -1, -1 ),
    meshSource->AddPoint(  1,  1, -1 ),
    meshSource->AddPoint(  1, -1,  1 ),
    meshSource->AddPoint( -1,  1,  1 )
    );
```

The function `AutomaticTopologyMeshSource::AddTetrahedron()` takes point identifiers as parameters; the identifiers must correspond to points that have already been added. `AutomaticTopologyMeshSource::AddPoint()` returns the appropriate identifier type for the point being added. It first checks to see if the point is already in the mesh. If so, it returns the ID of the point in the mesh, and if not, it generates a new unique ID, adds the point with that ID, and returns the ID.

Actually, `AddTetrahedron()` behaves in the same way. If the tetrahedron has already been added, it leaves the mesh unchanged and returns the ID that the tetrahedron already has. If not, it adds the tetrahedron (and all its faces, edges, and vertices), and generates a new ID, which it returns.

It is also possible to add all the points first, and then add a number of cells using the point IDs directly. This approach corresponds with the way the data is stored in many file formats for 3D polygonal models.

First we add the points (in this case the vertices of a larger tetrahedron). This example also illustrates that `AddPoint()` can take a single `PointType` as a parameter if desired, rather than a sequence of floats. Another possibility (not illustrated) is to pass in a C-style array.

```
  PointType p;
  IdentifierArrayType idArray( 4 );

  p[ 0 ] = -2;
  p[ 1 ] = -2;
  p[ 2 ] = -2;
  idArray[ 0 ] = meshSource->AddPoint( p );

  p[ 0 ] =  2;
  p[ 1 ] =  2;
  p[ 2 ] = -2;
```

```
idArray[ 1 ] = meshSource->AddPoint( p );

p[ 0 ] =  2;
p[ 1 ] = -2;
p[ 2 ] =  2;
idArray[ 2 ] = meshSource->AddPoint( p );

p[ 0 ] = -2;
p[ 1 ] =  2;
p[ 2 ] =  2;
idArray[ 3 ] = meshSource->AddPoint( p );
```

Now we add the cells. This time we are just going to create the boundary of a tetrahedron, so we must add each face separately.

```
meshSource->AddTriangle( idArray[0], idArray[1], idArray[2] );
meshSource->AddTriangle( idArray[1], idArray[2], idArray[3] );
meshSource->AddTriangle( idArray[2], idArray[3], idArray[0] );
meshSource->AddTriangle( idArray[3], idArray[0], idArray[1] );
```

Actually, we could have called, e.g., `AddTriangle( 4, 5, 6 )`, since IDs are assigned sequentially starting at zero, and `idArray[0]` contains the ID for the fifth point added. But you should only do this if you are confident that you know what the IDs are. If you add the same point twice and don't realize it, your count will differ from that of the mesh source.

You may be wondering what happens if you call, say, `AddEdge(0, 1)` followed by `AddEdge(1, 0)`. The answer is that they do count as the same edge, and so only one edge is added. The order of the vertices determines an orientation, and the first orientation specified is the one that is kept.

Once you have built the mesh you want, you can access it by calling `GetOutput()`. Here we send it to `cout`, which prints some summary data for the mesh.

In contrast to the case with typical filters, `GetOutput()` does not trigger an update process. The mesh is always maintained in a valid state as cells are added, and can be accessed at any time. It would, however, be a mistake to modify the mesh by some other means until AutomaticTopologyMeshSource is done with it, since the mesh source would then have an inaccurate record of which points and cells are currently in the mesh.

4.3.8 Iterating Through Cells

The source code for this section can be found in the file
`Examples/DataRepresentation/Mesh/MeshCellsIteration.cxx`.

Cells are stored in the `itk::Mesh` as pointers to a generic cell `itk::CellInterface`. This implies that only the virtual methods defined on this base cell class can be invoked. In order to use methods that are specific to each cell type it is necessary to down-cast the pointer to the actual type of the cell. This can be done safely by taking advantage of the `GetType()` method that allows to identify the actual type of a cell.

Let's start by assuming a mesh defined with one tetrahedron and all its boundary faces. That is, four triangles, six edges and four vertices.

The cells can be visited using CellsContainer iterators . The iterator `Value()` corresponds to a raw pointer to the `CellType` base class.

```
typedef MeshType::CellsContainer::ConstIterator  CellIterator;

CellIterator cellIterator = mesh->GetCells()->Begin();
CellIterator cellEnd      = mesh->GetCells()->End();

while( cellIterator != cellEnd )
  {
  CellType * cell = cellIterator.Value();
  std::cout << cell->GetNumberOfPoints() << std::endl;
  ++cellIterator;
  }
```

In order to perform down-casting in a safe manner, the cell type can be queried first using the `GetType()` method. Codes for the cell types have been defined with an `enum` type on the `itkCellInterface.h` header file. These codes are :

- VERTEX_CELL
- LINE_CELL
- TRIANGLE_CELL
- QUADRILATERAL_CELL
- POLYGON_CELL
- TETRAHEDRON_CELL
- HEXAHEDRON_CELL
- QUADRATIC_EDGE_CELL
- QUADRATIC_TRIANGLE_CELL

The method `GetType()` returns one of these codes. It is then possible to test the type of the cell before down-casting its pointer to the actual type. For example, the following code visits all the cells in the mesh and tests which ones are actually of type `LINE_CELL`. Only those cells are down-casted to `LineType` cells and a method specific for the `LineType` is invoked.

```
  cellIterator = mesh->GetCells()->Begin();
  cellEnd      = mesh->GetCells()->End();

  while( cellIterator != cellEnd )
    {
    CellType * cell = cellIterator.Value();
    if( cell->GetType() == CellType::LINE_CELL )
      {
      LineType * line = static_cast<LineType *>( cell );
      std::cout << "dimension = " << line->GetDimension();
      std::cout << " # points = " << line->GetNumberOfPoints();
      std::cout << std::endl;
      }
    ++cellIterator;
    }
```

In order to perform different actions on different cell types a `switch` statement can be used with cases for every cell type. The following code illustrates an iteration over the cells and the invocation of different methods on each cell type.

```
  cellIterator = mesh->GetCells()->Begin();
  cellEnd      = mesh->GetCells()->End();

  while( cellIterator != cellEnd )
    {
    CellType * cell = cellIterator.Value();
    switch( cell->GetType() )
      {
      case CellType::VERTEX_CELL:
        {
        std::cout << "VertexCell : " << std::endl;
        VertexType * line = dynamic_cast<VertexType *>( cell );
        std::cout << "dimension = " << line->GetDimension()      << std::endl;
        std::cout << "# points  = " << line->GetNumberOfPoints() << std::endl;
        break;
        }
      case CellType::LINE_CELL:
        {
        std::cout << "LineCell : " << std::endl;
        LineType * line = dynamic_cast<LineType *>( cell );
        std::cout << "dimension = " << line->GetDimension()      << std::endl;
        std::cout << "# points  = " << line->GetNumberOfPoints() << std::endl;
        break;
        }
      case CellType::TRIANGLE_CELL:
        {
        std::cout << "TriangleCell : " << std::endl;
        TriangleType * line = dynamic_cast<TriangleType *>( cell );
        std::cout << "dimension = " << line->GetDimension()      << std::endl;
        std::cout << "# points  = " << line->GetNumberOfPoints() << std::endl;
        break;
        }
      default:
        {
        std::cout << "Cell with more than three points" << std::endl;
        std::cout << "dimension = " << cell->GetDimension()      << std::endl;
        std::cout << "# points  = " << cell->GetNumberOfPoints() << std::endl;
        break;
        }
     }
    ++cellIterator;
    }
```

4.3.9 Visiting Cells

The source code for this section can be found in the file
`Examples/DataRepresentation/Mesh/MeshCellVisitor.cxx`.

In order to facilitate access to particular cell types, a convenience mechanism has been built-in on the `itk::Mesh`. This mechanism is based on the *Visitor Pattern* presented in [28]. The visitor pattern is designed to facilitate the process of walking through an heterogeneous list of objects sharing a common base class.

The first requirement for using the `CellVisitor` mechanism it to include the `CellInterfaceVisitor` header file.

```
#include "itkCellInterfaceVisitor.h"
```

The typical mesh types are now declared

```
  typedef float                                  PixelType;
  typedef itk::Mesh< PixelType, 3 >         MeshType;

  typedef MeshType::CellType                CellType;

  typedef itk::VertexCell< CellType >       VertexType;
  typedef itk::LineCell< CellType >         LineType;
  typedef itk::TriangleCell< CellType >     TriangleType;
  typedef itk::TetrahedronCell< CellType >  TetrahedronType;
```

Then, a custom CellVisitor class should be declared. In this particular example, the visitor class is intended to act only on `TriangleType` cells. The only requirement on the declaration of the visitor class is that it must provide a method named `Visit()`. This method expects as arguments a cell identifier and a pointer to the *specific* cell type for which this visitor is intended. Nothing prevents a visitor class from providing `Visit()` methods for several different cell types. The multiple methods will be differentiated by the natural C++ mechanism of function overload. The following code illustrates a minimal cell visitor class.

```
class CustomTriangleVisitor
  {
  public:
    typedef itk::TriangleCell<CellType>      TriangleType;

  public:
    void Visit(unsigned long cellId, TriangleType * t )
      {
      std::cout << "Cell # " << cellId << " is a TriangleType ";
      std::cout << t->GetNumberOfPoints() << std::endl;
      }
  };
```

This newly defined class will now be used to instantiate a cell visitor. In this particular example we create a class `CustomTriangleVisitor` which will be invoked each time a triangle cell is found while the mesh iterates over the cells.

```
typedef itk::CellInterfaceVisitorImplementation<
                            PixelType,
                            MeshType::CellTraits,
                            TriangleType,
                            CustomTriangleVisitor
                                     > TriangleVisitorInterfaceType;
```

Note that the actual `CellInterfaceVisitorImplementation` is templated over the PixelType, the CellTraits, the CellType to be visited and the Visitor class that defines with will be done with the cell.

A visitor implementation class can now be created using the normal invocation to its `New()` method and assigning the result to a `itk::SmartPointer`.

```
TriangleVisitorInterfaceType::Pointer  triangleVisitor =
                                 TriangleVisitorInterfaceType::New();
```

Many different visitors can be configured in this way. The set of all visitors can be registered with the MultiVisitor class provided for the mesh. An instance of the MultiVisitor class will walk through the cells and delegate action to every registered visitor when the appropriate cell type is encountered.

```
typedef CellType::MultiVisitor CellMultiVisitorType;
CellMultiVisitorType::Pointer multiVisitor = CellMultiVisitorType::New();
```

The visitor is registered with the Mesh using the `AddVisitor()` method.

```
  multiVisitor->AddVisitor( triangleVisitor );
```

Finally, the iteration over the cells is triggered by calling the method `Accept()` on the `itk::Mesh`.

```
  mesh->Accept( multiVisitor );
```

The `Accept()` method will iterate over all the cells and for each one will invite the MultiVisitor to attempt an action on the cell. If no visitor is interested on the current cell type the cell is just ignored and skipped.

MultiVisitors make it possible to add behavior to the cells without having to create new methods on the cell types or creating a complex visitor class that knows about every CellType.

4.3.10 More on Visiting Cells

The source code for this section can be found in the file
`Examples/DataRepresentation/Mesh/MeshCellVisitor2.cxx`.

The following section illustrates a realistic example of the use of Cell visitors on the `itk::Mesh`. A set of different visitors is defined here, each visitor associated with a particular type of cell. All the visitors are registered with a MultiVisitor class which is passed to the mesh.

The first step is to include the `CellInterfaceVisitor` header file.

```
#include "itkCellInterfaceVisitor.h"
```

The typical mesh types are now declared

```
  typedef float                             PixelType;
  typedef itk::Mesh< PixelType, 3 >         MeshType;

  typedef MeshType::CellType                CellType;

  typedef itk::VertexCell< CellType >       VertexType;
  typedef itk::LineCell< CellType >         LineType;
  typedef itk::TriangleCell< CellType >     TriangleType;
  typedef itk::TetrahedronCell< CellType >  TetrahedronType;
```

Then, custom CellVisitor classes should be declared. The only requirement on the declaration of each visitor class is to provide a method named `Visit()`. This method expects as arguments a cell identifier and a pointer to the *specific* cell type for which this visitor is intended.

The following Vertex visitor simply prints out the identifier of the point with which the cell is associated. Note that the cell uses the method `GetPointId()` without any arguments. This method is only defined on the VertexCell.

```
class CustomVertexVisitor
  {
  public:
    void Visit(unsigned long cellId, VertexType * t )
      {
      std::cout << "cell " << cellId << " is a Vertex " << std::endl;
      std::cout << "    associated with point id = ";
      std::cout << t->GetPointId() << std::endl;
      }
  };
```

The following Line visitor computes the length of the line. Note that this visitor is slightly more complicated since it needs to get access to the actual mesh in order to get point coordinates from the point identifiers returned by the line cell. This is done by holding a pointer to the mesh and querying the mesh each time point coordinates are required. The mesh pointer is set up in this case with the `SetMesh()` method.

```
class CustomLineVisitor
  {
  public:
    CustomLineVisitor():m_Mesh( 0 ) {}

    void SetMesh( MeshType * mesh ) { m_Mesh = mesh; }

    void Visit(unsigned long cellId, LineType * t )
      {
      std::cout << "cell " << cellId << " is a Line " << std::endl;
      LineType::PointIdIterator pit = t->PointIdsBegin();
      MeshType::PointType p0;
      MeshType::PointType p1;
      m_Mesh->GetPoint( *pit++, &p0 );
      m_Mesh->GetPoint( *pit++, &p1 );
      const double length = p0.EuclideanDistanceTo( p1 );
      std::cout << " length = " << length << std::endl;
      }

  private:
    MeshType::Pointer m_Mesh;
  };
```

The Triangle visitor below prints out the identifiers of its points. Note the use of the `PointIdIterator` and the `PointIdsBegin()` and `PointIdsEnd()` methods.

```
class CustomTriangleVisitor
  {
  public:
    void Visit(unsigned long cellId, TriangleType * t )
      {
      std::cout << "cell " << cellId << " is a Triangle " << std::endl;
      LineType::PointIdIterator pit = t->PointIdsBegin();
      LineType::PointIdIterator end = t->PointIdsEnd();
      while( pit != end )
        {
        std::cout << "  point id = " << *pit << std::endl;
        ++pit;
        }
      }
  };
```

The TetrahedronVisitor below simply returns the number of faces on this figure. Note that `GetNumberOfFaces()` is a method exclusive of 3D cells.

```
class CustomTetrahedronVisitor
  {
  public:
    void Visit(unsigned long cellId, TetrahedronType * t )
      {
      std::cout << "cell " << cellId << " is a Tetrahedron " << std::endl;
      std::cout << "  number of faces = ";
      std::cout << t->GetNumberOfFaces() << std::endl;
      }
  };
```

With the cell visitors we proceed now to instantiate CellVisitor implementations. The visitor classes defined above are used as template arguments of the cell visitor implementation.

```
typedef itk::CellInterfaceVisitorImplementation<
    PixelType, MeshType::CellTraits, VertexType, CustomVertexVisitor
                                            > VertexVisitorInterfaceType;

typedef itk::CellInterfaceVisitorImplementation<
    PixelType, MeshType::CellTraits, LineType, CustomLineVisitor
                                            > LineVisitorInterfaceType;

typedef itk::CellInterfaceVisitorImplementation<
    PixelType, MeshType::CellTraits, TriangleType, CustomTriangleVisitor
                                            > TriangleVisitorInterfaceType;

typedef itk::CellInterfaceVisitorImplementation<
    PixelType, MeshType::CellTraits, TetrahedronType, CustomTetrahedronVisitor
```

```
                                   > TetrahedronVisitorInterfaceType;
```

Note that the actual `CellInterfaceVisitorImplementation` is templated over the PixelType, the CellTraits, the CellType to be visited and the Visitor class defining what to do with the cell.

A visitor implementation class can now be created using the normal invocation to its `New()` method and assigning the result to a `itk::SmartPointer`.

```
  VertexVisitorInterfaceType::Pointer  vertexVisitor =
                                  VertexVisitorInterfaceType::New();

  LineVisitorInterfaceType::Pointer  lineVisitor =
                                  LineVisitorInterfaceType::New();

  TriangleVisitorInterfaceType::Pointer  triangleVisitor =
                                  TriangleVisitorInterfaceType::New();

  TetrahedronVisitorInterfaceType::Pointer  tetrahedronVisitor =
                                  TetrahedronVisitorInterfaceType::New();
```

Remember that the LineVisitor requires the pointer to the mesh object since it needs to get access to actual point coordinates. This is done by invoking the `SetMesh()` method defined above.

```
  lineVisitor->SetMesh( mesh );
```

Looking carefully you will notice that the `SetMesh()` method is declared in `CustomLineVisitor` but we are invoking it on `LineVisitorInterfaceType`. This is possible thanks to the way in which the VisitorInterfaceImplementation is defined. This class derives from the visitor type provided by the user as the fourth template parameter. `LineVisitorInterfaceType` is then a derived class of `CustomLineVisitor`.

The set of visitors should now be registered with the MultiVisitor class that will walk through the cells and delegate action to every registered visitor when the appropriate cell type is encountered. The following lines create a MultiVisitor object.

```
  typedef CellType::MultiVisitor CellMultiVisitorType;
  CellMultiVisitorType::Pointer multiVisitor = CellMultiVisitorType::New();
```

Every visitor implementation is registered with the Mesh using the `AddVisitor()` method.

```
  multiVisitor->AddVisitor( vertexVisitor      );
  multiVisitor->AddVisitor( lineVisitor        );
  multiVisitor->AddVisitor( triangleVisitor    );
  multiVisitor->AddVisitor( tetrahedronVisitor );
```

Finally, the iteration over the cells is triggered by calling the method `Accept()` on the Mesh class.

```
  mesh->Accept( multiVisitor );
```

The `Accept()` method will iterate over all the cells and for each one will invite the MultiVisitor to attempt an action on the cell. If no visitor is interested on the current cell type, the cell is just ignored and skipped.

4.4 Path

4.4.1 Creating a PolyLineParametricPath

The source code for this section can be found in the file
`Examples/DataRepresentation/Path/PolyLineParametricPath1.cxx`.

This example illustrates how to use the `itk::PolyLineParametricPath`. This class will typically be used for representing in a concise way the output of an image segmentation algorithm in 2D. The `PolyLineParametricPath` however could also be used for representing any open or close curve in N-Dimensions as a linear piece-wise approximation.

First, the header file of the `PolyLineParametricPath` class must be included.

```
#include "itkPolyLineParametricPath.h"
```

The path is instantiated over the dimension of the image. In this case 2D. //

```
  const unsigned int Dimension = 2;

  typedef itk::Image< unsigned char, Dimension > ImageType;

  typedef itk::PolyLineParametricPath< Dimension > PathType;


  ImageType::ConstPointer image = reader->GetOutput();


  PathType::Pointer path = PathType::New();


  path->Initialize();


  typedef PathType::ContinuousIndexType    ContinuousIndexType;
  ContinuousIndexType cindex;

  typedef ImageType::PointType             ImagePointType;

  ImagePointType origin = image->GetOrigin();


  ImageType::SpacingType spacing = image->GetSpacing();
  ImageType::SizeType    size    = image->GetBufferedRegion().GetSize();

  ImagePointType point;

  point[0] = origin[0] + spacing[0] * size[0];
  point[1] = origin[1] + spacing[1] * size[1];

  image->TransformPhysicalPointToContinuousIndex( origin, cindex );

  path->AddVertex( cindex );

  image->TransformPhysicalPointToContinuousIndex( point, cindex );

  path->AddVertex( cindex );
```

4.5 Containers

The source code for this section can be found in the file
`Examples/DataRepresentation/Containers/TreeContainer.cxx`.

This example shows how to use the `itk::TreeContainer` and the associated TreeIterators. The `itk::TreeContainer` implements the notion of tree and is templated over the type of node so it can virtually handle any objects. Each node is supposed to have only one parent so no cycle is present in the tree. No checking is done to ensure a cycle-free tree.

Let's begin by including the appropriate header file.

```
#include <itkTreeContainer.h>
#include "itkTreeContainer.h"
#include "itkChildTreeIterator.h"
#include "itkLeafTreeIterator.h"
#include "itkLevelOrderTreeIterator.h"
#include "itkInOrderTreeIterator.h"
#include "itkPostOrderTreeIterator.h"
```

```
#include "itkPreOrderTreeIterator.h"
#include "itkRootTreeIterator.h"
#include "itkTreeIteratorClone.h"
```

First, we create a tree of integers. The TreeContainer is templated over the type of nodes.

```
  typedef int NodeType;
  typedef itk::TreeContainer<NodeType> TreeType;
  TreeType::Pointer tree = TreeType::New();
```

Next we set the value of the root node using `SetRoot()`.

```
  tree->SetRoot(0);
```

Then we use the `Add()` function to add nodes to the tree The first argument is the value of the new node and the second argument is the value of the parent node. If two nodes have the same values then the first one is picked. In this particular case it is better to use an iterator to fill the tree.

```
  tree->Add(1,0);
  tree->Add(2,0);
  tree->Add(3,0);
  tree->Add(4,2);
  tree->Add(5,2);
  tree->Add(6,5);
  tree->Add(7,1);
```

We define an `itk::LevelOrderTreeIterator` to parse the tree in level order. This particular iterator takes three arguments. The first one is the actual tree to be parsed, the second one is the maximum depth level and the third one is the starting node. The `GetNode()` function return a node given its value. Once again the first node that corresponds to the value is returned.

```
  itk::LevelOrderTreeIterator<TreeType> levelIt(tree,10,tree->GetNode(2));
  levelIt.GoToBegin();
  while(!levelIt.IsAtEnd())
    {
    std::cout << levelIt.Get() << " ("<< levelIt.GetLevel() << ")" << std::endl;;
    ++levelIt;
    }
  std::cout << std::endl;
```

The TreeIterators have useful functions to test the property of the current pointed node. Among these functions: `IsLeaf` returns true if the current node is a leaf, `IsRoot` returns true if the node is a root, `HasParent` returns true if the node has a parent and `CountChildren` returns the number of children for this particular node.

```
levelIt.IsLeaf();
levelIt.IsRoot();
levelIt.HasParent();
levelIt.CountChildren();
```

The `itk::ChildTreeIterator` provides another way to iterate through a tree by listing all the children of a node.

```
itk::ChildTreeIterator<TreeType> childIt(tree);
childIt.GoToBegin();
while(!childIt.IsAtEnd())
  {
  std::cout << childIt.Get() << std::endl;;
  ++childIt;
  }
std::cout << std::endl;
```

The `GetType()` function returns the type of iterator used. The list of enumerated types is as follow: PREORDER, INORDER, POSTORDER, LEVELORDER, CHILD, ROOT and LEAF.

```
if(childIt.GetType() != itk::TreeIteratorBase<TreeType>::CHILD)
  {
  std::cout << "[FAILURE]" << std::endl;
  return EXIT_FAILURE;
  }
```

Every TreeIterator has a `Clone()` function which returns a copy of the current iterator. Note that the user should delete the created iterator by hand.

```
childIt.GoToParent();
itk::TreeIteratorBase<TreeType>* childItClone = childIt.Clone();
delete childItClone;
```

The `itk::LeafTreeIterator` iterates through the leaves of the tree.

```
itk::LeafTreeIterator<TreeType> leafIt(tree);
leafIt.GoToBegin();
while(!leafIt.IsAtEnd())
  {
  std::cout << leafIt.Get() << std::endl;;
  ++leafIt;
  }
std::cout << std::endl;
```

The `itk::InOrderTreeIterator` iterates through the tree in the order from left to right.

```
itk::InOrderTreeIterator<TreeType> InOrderIt(tree);
```

```
InOrderIt.GoToBegin();
while(!InOrderIt.IsAtEnd())
  {
  std::cout << InOrderIt.Get() << std::endl;;
  ++InOrderIt;
  }
std::cout << std::endl;
```

The `itk::PreOrderTreeIterator` iterates through the tree from left to right but do a depth first search.

```
itk::PreOrderTreeIterator<TreeType> PreOrderIt(tree);
PreOrderIt.GoToBegin();
while(!PreOrderIt.IsAtEnd())
  {
  std::cout << PreOrderIt.Get() << std::endl;;
  ++PreOrderIt;
  }
std::cout << std::endl;
```

The `itk::PostOrderTreeIterator` iterates through the tree from left to right but goes from the leaves to the root in the search.

```
itk::PostOrderTreeIterator<TreeType> PostOrderIt(tree);
PostOrderIt.GoToBegin();
while(!PostOrderIt.IsAtEnd())
  {
  std::cout << PostOrderIt.Get() << std::endl;;
  ++PostOrderIt;
  }
std::cout << std::endl;
```

The `itk::RootTreeIterator` goes from one node to the root. The second arguments is the starting node. Here we go from the leaf node (value = 6) up to the root.

```
itk::RootTreeIterator<TreeType> RootIt(tree,tree->GetNode(6));
RootIt.GoToBegin();
while(!RootIt.IsAtEnd())
  {
  std::cout << RootIt.Get() << std::endl;;
  ++RootIt;
  }
std::cout << std::endl;
```

All the nodes of the tree can be removed by using the `Clear()` function.

```
tree->Clear();
```

We show how to use a TreeIterator to form a tree by creating nodes. The `Add()` function is used to add a node and put a value on it. The `GoToChild()` is used to jump to a node.

```
itk::PreOrderTreeIterator<TreeType> PreOrderIt2(tree);
PreOrderIt2.Add(0);
PreOrderIt2.Add(1);
PreOrderIt2.Add(2);
PreOrderIt2.Add(3);
PreOrderIt2.GoToChild(2);
PreOrderIt2.Add(4);
PreOrderIt2.Add(5);
```

The `itk::TreeIteratorClone` can be used to have a generic copy of an iterator.

```
typedef itk::TreeIteratorBase<TreeType> IteratorType;
typedef itk::TreeIteratorClone<IteratorType> IteratorCloneType;
itk::PreOrderTreeIterator<TreeType> anIterator(tree);
IteratorCloneType aClone = anIterator;
```

CHAPTER

FIVE

Spatial Objects

This chapter introduces the basic classes that describe `itk::SpatialObject`s.

5.1 Introduction

We promote the philosophy that many of the goals of medical image processing are more effectively addressed if we consider them in the broader context of object processing. ITK's Spatial Object class hierarchy provides a consistent API for querying, manipulating, and interconnecting objects in physical space. Via this API, methods can be coded to be invariant to the data structure used to store the objects being processed. By abstracting the representations of objects to support their representation by data structures other than images, a broad range of medical image analysis research is supported; key examples are described in the following.

Model-to-image registration. A mathematical instance of an object can be registered with an image to localize the instance of that object in the image. Using SpatialObjects, mutual information, cross-correlation, and boundary-to-image metrics can be applied without modification to perform spatial object-to-image registration.

Model-to-model registration. Iterative closest point, landmark, and surface distance minimization methods can be used with any ITK transform, to rigidly and non-rigidly register image, FEM, and Fourier descriptor-based representations of objects as SpatialObjects.

Atlas formation. Collections of images or SpatialObjects can be integrated to represent expected object characteristics and their common modes of variation. Labels can be associated with the objects of an atlas.

Storing segmentation results from one or multiple scans. Results of segmentations are best stored in physical/world coordinates so that they can be combined and compared with other segmentations from other images taken at other resolutions. Segmentation results from hand drawn contours, pixel labelings, or model-to-image registrations are treated consistently.

Capturing functional and logical relationships between objects. SpatialObjects can have parent and children objects. Queries made of an object (such as to determine if a point is inside of the object) can be made to integrate the responses from the children object. Transformations applied to a parent can also be propagated to the children. Thus, for example, when a liver model is moved, its vessels move with it.

Conversion to and from images. Basic functions are provided to render any SpatialObject (or collection of SpatialObjects) into an image.

IO. SpatialObject reading and writing to disk is independent of the SpatialObject class hierarchy. Meta object IO (through `itk::MetaImageIO`) methods are provided, and others are easily defined.

Tubes, blobs, images, surfaces. Are a few of the many SpatialObject data containers and types provided. New types can be added, generally by only defining one or two member functions in a derived class.

In the remainder of this chapter several examples are used to demonstrate the many spatial objects found in ITK and how they can be organized into hierarchies using `itk::SceneSpatialObject`. Further the examples illustrate how to use SpatialObject transformations to control and calculate the position of objects in space.

5.2 Hierarchy

Spatial objects can be combined to form a hierarchy as a tree. By design, a SpatialObject can have one parent and only one. Moreover, each transform is stored within each object, therefore the hierarchy cannot be described as a Directed Acyclic Graph (DAG) but effectively as a tree. The user is responsible for maintaining the tree structure, no checking is done to ensure a cycle-free tree.

The source code for this section can be found in the file
`Examples/SpatialObjects/SpatialObjectHierarchy.cxx`.

This example describes how `itk::SpatialObject` can form a hierarchy. This first example also shows how to create and manipulate spatial objects.

```
#include "itkSpatialObject.h"
```

First, we create two spatial objects and give them the names `First Object` and `Second Object`, respectively.

```
  typedef itk::SpatialObject<3> SpatialObjectType;

  SpatialObjectType::Pointer object1 = SpatialObjectType ::New();
```

```
object1->GetProperty()->SetName("First Object");

SpatialObjectType::Pointer object2 = SpatialObjectType ::New();
object2->GetProperty()->SetName("Second Object");
```

We then add the second object to the first one by using the `AddSpatialObject()` method. As a result `object2` becomes a child of object1.

```
object1->AddSpatialObject(object2);
```

We can query if an object has a parent by using the HasParent() method. If it has one, the `GetParent()` method returns a constant pointer to the parent. In our case, if we ask the parent's name of the object2 we should obtain: `First Object`.

```
if(object2->HasParent())
  {
  std::cout << "Name of the parent of the object2: ";
  std::cout << object2->GetParent()->GetProperty()->GetName() << std::endl;
  }
```

To access the list of children of the object, the `GetChildren()` method returns a pointer to the (STL) list of children.

```
SpatialObjectType::ChildrenListType * childrenList = object1->GetChildren();
std::cout << "object1 has " << childrenList->size() << " child" << std::endl;

SpatialObjectType::ChildrenListType::const_iterator it = childrenList->begin();
while(it != childrenList->end())
  {
  std::cout << "Name of the child of the object 1: ";
  std::cout << (*it)->GetProperty()->GetName() << std::endl;
  it++;
  }
```

Do NOT forget to delete the list of children since the `GetChildren()` function creates an internal list.

```
delete childrenList;
```

An object can also be removed by using the `RemoveSpatialObject()` method.

```
object1->RemoveSpatialObject(object2);
```

We can query the number of children an object has with the `GetNumberOfChildren()` method.

```
  std::cout << "Number of children for object1: ";
  std::cout << object1->GetNumberOfChildren() << std::endl;
```

The `Clear()` method erases all the information regarding the object as well as the data. This method is usually overloaded by derived classes.

```
  object1->Clear();
```

The output of this first example looks like the following:

```
   Name of the parent of the object2: First Object
   object1 has 1 child
   Name of the child of the object 1: Second Object
   Number of children for object1: 0
```

5.3 SpatialObject Tree Container

The source code for this section can be found in the file
`Examples/SpatialObjects/SpatialObjectTreeContainer.cxx`.

This example describes how to use the `itk::SpatialObjectTreeContainer` to form a hierarchy of SpatialObjects. First we include the appropriate header file.

```
#include "itkSpatialObjectTreeContainer.h"
```

Next we define the type of node and the type of tree we plan to use. Both are templated over the dimensionality of the space. Let's create a 2-dimensional tree.

```
  typedef itk::GroupSpatialObject<2> NodeType;
  typedef itk::SpatialObjectTreeContainer<2> TreeType;
```

Then, we can create three nodes and set their corresponding identification numbers (using `SetId`).

```
  NodeType::Pointer object0 = NodeType::New();
  object0->SetId(0);
  NodeType::Pointer object1 = NodeType::New();
  object1->SetId(1);
  NodeType::Pointer object2 = NodeType::New();
  object2->SetId(2);
```

The hierarchy is formed using the `AddSpatialObject()` function.

```
object0->AddSpatialObject(object1);
object1->AddSpatialObject(object2);
```

After instantiation of the tree we set its root using the `SetRoot()` function.

```
TreeType::Pointer tree = TreeType::New();
tree->SetRoot(object0.GetPointer());
```

The tree iterators described in a previous section of this guide can be used to parse the hierarchy. For example, via an `itk::LevelOrderTreeIterator` templated over the type of tree, we can parse the hierarchy of SpatialObjects. We set the maximum level to 10 which is enough in this case since our hierarchy is only 2 deep.

```
itk::LevelOrderTreeIterator<TreeType> levelIt(tree,10);
levelIt.GoToBegin();
while(!levelIt.IsAtEnd())
  {
  std::cout << levelIt.Get()->GetId() << " ("<< levelIt.GetLevel()
    << ")" << std::endl;;
  ++levelIt;
  }
```

Tree iterators can also be used to add spatial objects to the hierarchy. Here we show how to use the `itk::PreOrderTreeIterator` to add a fourth object to the tree.

```
NodeType::Pointer object4 = NodeType::New();
itk::PreOrderTreeIterator<TreeType> preIt( tree );
preIt.Add(object4.GetPointer());
```

5.4 Transformations

The source code for this section can be found in the file
`Examples/SpatialObjects/SpatialObjectTransforms.cxx`.

This example describes the different transformations associated with a spatial object.

Figure 5.1 shows our set of transformations.

Like the first example, we create two spatial objects and give them the names `First Object` and `Second Object`, respectively.

```
typedef itk::SpatialObject<2>             SpatialObjectType;
typedef SpatialObjectType::TransformType  TransformType;

SpatialObjectType::Pointer object1 = SpatialObjectType ::New();
object1->GetProperty()->SetName("First Object");
```

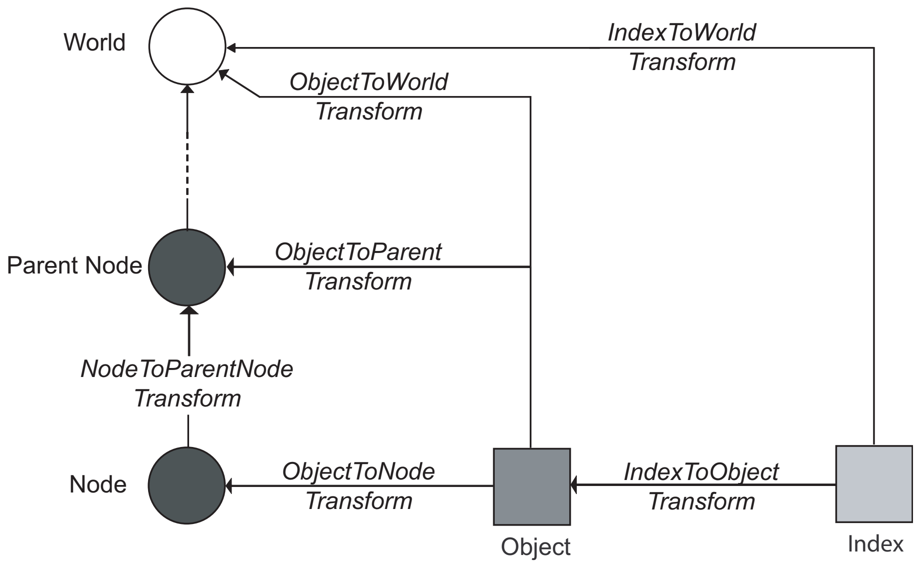


Figure 5.1: Set of transformations associated with a Spatial Object

```
SpatialObjectType::Pointer object2 = SpatialObjectType ::New();
object2->GetProperty()->SetName("Second Object");
object1->AddSpatialObject(object2);
```

Instances of `itk::SpatialObject` maintain three transformations internally that can be used to compute the position and orientation of data and objects. These transformations are: an IndexToObjectTransform, an ObjectToParentTransform, and an ObjectToWorldTransform. As a convenience to the user, the global transformation IndexToWorldTransform and its inverse, WorldToIndexTransform, are also maintained by the class. Methods are provided by SpatialObject to access and manipulate these transforms.

The two main transformations, IndexToObjectTransform and ObjectToParentTransform, are applied successively. ObjectToParentTransform is applied to children.

The IndexToObjectTransform transforms points from the internal data coordinate system of the object (typically the indices of the image from which the object was defined) to "physical" space (which accounts for the spacing, orientation, and offset of the indices).

The ObjectToParentTransform transforms points from the object-specific "physical" space to the "physical" space of its parent object. As one can see from the figure 5.1, the ObjectToParentTransform is composed of two transforms: ObjectToNodeTransform and NodeToParentNodeTransform. The ObjectToNodeTransform is not applied to the children, but the ObjectToNodeTransform is. Therefore, if one sets the ObjectToParentTransform, the ObjectToNodeTransform is modified.

The ObjectToWorldTransform maps points from the reference system of the SpatialObject into the global coordinate system. This is useful when the position of the object is known only in the global coordinate frame. Note that by setting this transform, the ObjectToParent transform is recomputed.

These transformations use the `itk::FixedCenterOfRotationAffineTransform`. They are created in the constructor of the spatial `itk::SpatialObject`.

First we define an index scaling factor of 2 for the object2. This is done by setting the Scale of the IndexToObjectTransform.

```
double scale[2];
scale[0]=2;
scale[1]=2;
object2->GetIndexToObjectTransform()->SetScale(scale);
```

Next, we apply an offset on the ObjectToParentTransform of the child object Therefore, object2 is now translated by a vector [4,3] regarding to its parent.

```
TransformType::OffsetType Object2ToObject1Offset;
Object2ToObject1Offset[0] = 4;
Object2ToObject1Offset[1] = 3;
object2->GetObjectToParentTransform()->SetOffset(Object2ToObject1Offset);
```

To realize the previous operations on the transformations, we should invoke the `ComputeObjectToWorldTransform()` that recomputes all dependent transformations.

```
object2->ComputeObjectToWorldTransform();
```

We can now display the ObjectToWorldTransform for both objects. One should notice that the FixedCenterOfRotationAffineTransform derives from `itk::AffineTransform` and therefore the only valid members of the transformation are a Matrix and an Offset. For instance, when we invoke the `Scale()` method the internal Matrix is recomputed to reflect this change.

The FixedCenterOfRotationAffineTransform performs the following computation

$$X' = R \cdot (S \cdot X - C) + C + V \tag{5.1}$$

Where R is the rotation matrix, S is a scaling factor, C is the center of rotation and V is a translation vector or offset. Therefore the affine matrix M and the affine offset T are defined as:

$$M = R \cdot S \tag{5.2}$$

$$T = C + V - R \cdot C \tag{5.3}$$

This means that `GetScale()` and `GetOffset()` as well as the `GetMatrix()` might not be set to the expected value, especially if the transformation results from a composition with another transformation since the composition is done using the Matrix and the Offset of the affine transformation.

Next, we show the two affine transformations corresponding to the two objects.

```
std::cout << "object2 IndexToObject Matrix: " << std::endl;
std::cout << object2->GetIndexToObjectTransform()->GetMatrix() << std::endl;
std::cout << "object2 IndexToObject Offset: ";
std::cout << object2->GetIndexToObjectTransform()->GetOffset() << std::endl;
std::cout << "object2 IndexToWorld Matrix: " << std::endl;
std::cout << object2->GetIndexToWorldTransform()->GetMatrix() << std::endl;
std::cout << "object2 IndexToWorld Offset: ";
std::cout << object2->GetIndexToWorldTransform()->GetOffset() << std::endl;
```

Then, we decide to translate the first object which is the parent of the second by a vector [3,3]. This is still done by setting the offset of the ObjectToParentTransform. This can also be done by setting the ObjectToWorldTransform because the first object does not have any parent and therefore is attached to the world coordinate frame.

```
TransformType::OffsetType Object1ToWorldOffset;
Object1ToWorldOffset[0] = 3;
Object1ToWorldOffset[1] = 3;
object1->GetObjectToParentTransform()->SetOffset(Object1ToWorldOffset);
```

Next we invoke `ComputeObjectToWorldTransform()` on the modified object. This will propagate the transformation through all its children.

```
object1->ComputeObjectToWorldTransform();
```

Figure 5.2 shows our set of transformations.

Finally, we display the resulting affine transformations.

```
std::cout << "object1 IndexToWorld Matrix: " << std::endl;
std::cout << object1->GetIndexToWorldTransform()->GetMatrix() << std::endl;
std::cout << "object1 IndexToWorld Offset: ";
std::cout << object1->GetIndexToWorldTransform()->GetOffset() << std::endl;
std::cout << "object2 IndexToWorld Matrix: " << std::endl;
std::cout << object2->GetIndexToWorldTransform()->GetMatrix() << std::endl;
std::cout << "object2 IndexToWorld Offset: ";
std::cout << object2->GetIndexToWorldTransform()->GetOffset() << std::endl;
```

The output of this second example looks like the following:

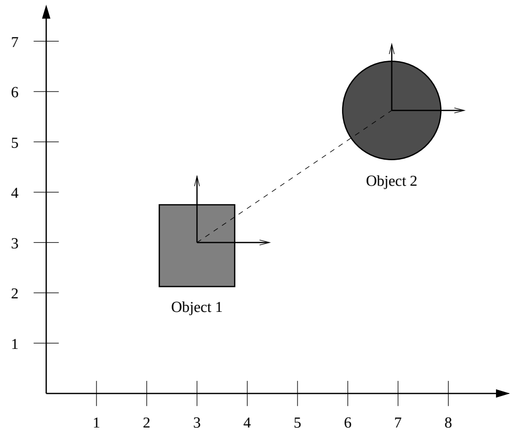


Figure 5.2: Physical positions of the two objects in the world frame (shapes are merely for illustration purposes).

```
object2 IndexToObject Matrix:
2 0
0 2
object2 IndexToObject Offset: 0  0
object2 IndexToWorld Matrix:
2 0
0 2
object2 IndexToWorld Offset: 4  3
object1 IndexToWorld Matrix:
1 0
0 1
object1 IndexToWorld Offset: 3  3
object2 IndexToWorld Matrix:
2 0
0 2
object2 IndexToWorld Offset: 7  6
```

5.5 Types of Spatial Objects

This section describes in detail the variety of spatial objects implemented in ITK.

5.5.1 ArrowSpatialObject

The source code for this section can be found in the file
`Examples/SpatialObjects/ArrowSpatialObject.cxx`.

This example shows how to create a `itk::ArrowSpatialObject`. Let's begin by including the appropriate header file.

```
#include <itkArrowSpatialObject.h>
```

The `itk::ArrowSpatialObject`, like many SpatialObjects, is templated over the dimensionality of the object.

```
  typedef itk::ArrowSpatialObject<3>   ArrowType;
  ArrowType::Pointer myArrow = ArrowType::New();
```

The length of the arrow in the local coordinate frame is done using the SetLength() function. By default the length is set to 1.

```
  myArrow->SetLength(2);
```

The direction of the arrow can be set using the SetDirection() function. The SetDirection() function modifies the ObjectToParentTransform (not the IndexToObjectTransform). By default the direction is set along the X axis (first direction).

```
  ArrowType::VectorType direction;
  direction.Fill(0);
  direction[1] = 1.0;
  myArrow->SetDirection(direction);
```

5.5.2 BlobSpatialObject

The source code for this section can be found in the file
`Examples/SpatialObjects/BlobSpatialObject.cxx`.

`itk::BlobSpatialObject` defines an N-dimensional blob. Like other SpatialObjects this class derives from `itk::itkSpatialObject`. A blob is defined as a list of points which compose the object.

Let's start by including the appropriate header file.

```
#include "itkBlobSpatialObject.h"
```

BlobSpatialObject is templated over the dimension of the space. A BlobSpatialObject contains a list of SpatialObjectPoints. Basically, a SpatialObjectPoint has a position and a color.

```
#include "itkSpatialObjectPoint.h"
```

First we declare some type definitions.

```
  typedef itk::BlobSpatialObject<3>    BlobType;
  typedef BlobType::Pointer            BlobPointer;
  typedef itk::SpatialObjectPoint<3>   BlobPointType;
```

Then, we create a list of points and we set the position of each point in the local coordinate system using the `SetPosition()` method. We also set the color of each point to be red.

```
 BlobType::PointListType list;

  for( unsigned int i=0; i<4; i++)
    {
    BlobPointType p;
    p.SetPosition(i,i+1,i+2);
    p.SetRed(1);
    p.SetGreen(0);
    p.SetBlue(0);
    p.SetAlpha(1.0);
    list.push_back(p);
    }
```

Next, we create the blob and set its name using the `SetName()` function. We also set its Identification number with `SetId()` and we add the list of points previously created.

```
  BlobPointer blob = BlobType::New();
  blob->GetProperty()->SetName("My Blob");
  blob->SetId(1);
  blob->SetPoints(list);
```

The `GetPoints()` method returns a reference to the internal list of points of the object.

```
   BlobType::PointListType pointList = blob->GetPoints();
   std::cout << "The blob contains " << pointList.size();
   std::cout << " points" << std::endl;
```

Then we can access the points using standard STL iterators and `GetPosition()` and `GetColor()` functions return respectively the position and the color of the point.

```
  BlobType::PointListType::const_iterator it = blob->GetPoints().begin();
  while(it != blob->GetPoints().end())
    {
    std::cout << "Position = " << (*it).GetPosition() << std::endl;
    std::cout << "Color = " << (*it).GetColor() << std::endl;
    it++;
    }
```

5.5.3 CylinderSpatialObject

The source code for this section can be found in the file
`Examples/SpatialObjects/CylinderSpatialObject.cxx`.

This example shows how to create a `itk::CylinderSpatialObject`. Let's begin by including the appropriate header file.

```
#include "itkCylinderSpatialObject.h"
```

An `itk::CylinderSpatialObject` exists only in 3D, therefore, it is not templated.

```
  typedef itk::CylinderSpatialObject   CylinderType;
```

We create a cylinder using the standard smart pointers.

```
  CylinderType::Pointer myCylinder = CylinderType::New();
```

The radius of the cylinder is set using the `SetRadius()` function. By default the radius is set to 1.

```
  double radius = 3.0;
  myCylinder->SetRadius(radius);
```

The height of the cylinder is set using the `SetHeight()` function. By default the cylinder is defined along the X axis (first dimension).

```
  double height = 12.0;
  myCylinder->SetHeight(height);
```

Like any other `itk::SpatialObject`s, the `IsInside()` function can be used to query if a point is inside or outside the cylinder.

```
  itk::Point<double,3> insidePoint;
  insidePoint[0]=1;
  insidePoint[1]=2;
  insidePoint[2]=0;
  std::cout << "Is my point "<< insidePoint << " inside the cylinder? : "
    << myCylinder->IsInside(insidePoint) << std::endl;
```

We can print the cylinder information using the `Print()` function.

```
  myCylinder->Print(std::cout);
```

5.5.4 EllipseSpatialObject

The source code for this section can be found in the file
`Examples/SpatialObjects/EllipseSpatialObject.cxx`.

`itk::EllipseSpatialObject` defines an n-Dimensional ellipse. Like other spatial objects this class derives from `itk::SpatialObject`. Let's start by including the appropriate header file.

```
#include "itkEllipseSpatialObject.h"
```

Like most of the SpatialObjects, the `itk::EllipseSpatialObject` is templated over the dimension of the space. In this example we create a 3-dimensional ellipse.

```
  typedef itk::EllipseSpatialObject<3>   EllipseType;
  EllipseType::Pointer myEllipse = EllipseType::New();
```

Then we set a radius for each dimension. By default the radius is set to 1.

```
  EllipseType::ArrayType radius;
  for(unsigned int i = 0; i<3; i++)
    {
    radius[i] = i;
    }

  myEllipse->SetRadius(radius);
```

Or if we have the same radius in each dimension we can do

```
  myEllipse->SetRadius(2.0);
```

We can then display the current radius by using the `GetRadius()` function:

```
  EllipseType::ArrayType myCurrentRadius = myEllipse->GetRadius();
  std::cout << "Current radius is " << myCurrentRadius << std::endl;
```

Like other SpatialObjects, we can query the object if a point is inside the object by using the IsInside(itk::Point) function. This function expects the point to be in world coordinates.

```
  itk::Point<double,3> insidePoint;
  insidePoint.Fill(1.0);
  if(myEllipse->IsInside(insidePoint))
    {
```

```
    std::cout << "The point " << insidePoint;
    std::cout << " is really inside the ellipse" << std::endl;
    }

  itk::Point<double,3> outsidePoint;
  outsidePoint.Fill(3.0);
  if(!myEllipse->IsInside(outsidePoint))
    {
    std::cout << "The point " << outsidePoint;
    std::cout << " is really outside the ellipse" << std::endl;
    }
```

All spatial objects can be queried for a value at a point. The `IsEvaluableAt()` function returns a boolean to know if the object is evaluable at a particular point.

```
   if(myEllipse->IsEvaluableAt(insidePoint))
    {
    std::cout << "The point " << insidePoint;
    std::cout << " is evaluable at the point " << insidePoint << std::endl;
    }
```

If the object is evaluable at that point, the `ValueAt()` function returns the current value at that position. Most of the objects returns a boolean value which is set to true when the point is inside the object and false when it is outside. However, for some objects, it is more interesting to return a value representing, for instance, the distance from the center of the object or the distance from from the boundary.

```
  double value;
  myEllipse->ValueAt(insidePoint,value);
  std::cout << "The value inside the ellipse is: " << value << std::endl;
```

Like other spatial objects, we can also query the bounding box of the object by using `GetBoundingBox()`. The resulting bounding box is expressed in the local frame.

```
  myEllipse->ComputeBoundingBox();
  EllipseType::BoundingBoxType * boundingBox = myEllipse->GetBoundingBox();
  std::cout << "Bounding Box: " << boundingBox->GetBounds() << std::endl;
```

5.5.5 GaussianSpatialObject

The source code for this section can be found in the file
`Examples/SpatialObjects/GaussianSpatialObject.cxx`.

This example shows how to create a `itk::GaussianSpatialObject` which defines a Gaussian in a N-dimensional space. This object is particularly useful to query the value at a point in physical space. Let's begin by including the appropriate header file.

```
#include "itkGaussianSpatialObject.h"
```

The itk::GaussianSpatialObject is templated over the dimensionality of the object.

```
  typedef itk::GaussianSpatialObject<3>   GaussianType;
  GaussianType::Pointer myGaussian = GaussianType::New();
```

The SetMaximum() function is used to set the maximum value of the Gaussian.

```
  myGaussian->SetMaximum(2);
```

The radius of the Gaussian is defined by the SetRadius() method. By default the radius is set to 1.0.

```
  myGaussian->SetRadius(3);
```

The standard ValueAt() function is used to determine the value of the Gaussian at a particular point in physical space.

```
  itk::Point<double,3> pt;
  pt[0]=1;
  pt[1]=2;
  pt[2]=1;
  double value;
  myGaussian->ValueAt(pt, value);
  std::cout << "ValueAt(" << pt << ") = " << value << std::endl;
```

5.5.6 GroupSpatialObject

The source code for this section can be found in the file
`Examples/SpatialObjects/GroupSpatialObject.cxx`.

A itk::GroupSpatialObject does not have any data associated with it. It can be used to group objects or to add transforms to a current object. In this example we show how to use a GroupSpatialObject.

Let's begin by including the appropriate header file.

```
#include <itkGroupSpatialObject.h>
```

The itk::GroupSpatialObject is templated over the dimensionality of the object.

```
  typedef itk::GroupSpatialObject<3>   GroupType;
  GroupType::Pointer myGroup = GroupType::New();
```

Next, we create an itk::EllipseSpatialObject and add it to the group.

```
typedef itk::EllipseSpatialObject<3>   EllipseType;
EllipseType::Pointer myEllipse = EllipseType::New();
myEllipse->SetRadius(2);

myGroup->AddSpatialObject(myEllipse);
```

We then translate the group by 10mm in each direction. Therefore the ellipse is translated in physical space at the same time.

```
GroupType::VectorType offset;
offset.Fill(10);
myGroup->GetObjectToParentTransform()->SetOffset(offset);
myGroup->ComputeObjectToWorldTransform();
```

We can then query if a point is inside the group using the `IsInside()` function. We need to specify in this case that we want to consider all the hierarchy, therefore we set the depth to 2.

```
GroupType::PointType point;
point.Fill(10);
std::cout << "Is my point " << point << " inside?: "
  <<  myGroup->IsInside(point,2) << std::endl;
```

Like any other SpatialObjects we can remove the ellipse from the group using the `RemoveSpatialObject()` method.

```
myGroup->RemoveSpatialObject(myEllipse);
```

5.5.7 ImageSpatialObject

The source code for this section can be found in the file
`Examples/SpatialObjects/ImageSpatialObject.cxx`.

An itk::ImageSpatialObject contains an itk::Image but adds the notion of spatial transformations and parent-child hierarchy. Let's begin the next example by including the appropriate header file.

```
#include "itkImageSpatialObject.h"
```

We first create a simple 2D image of size 10 by 10 pixels.

```
typedef itk::Image<short,2> Image;
Image::Pointer image = Image::New();
```

```
Image::SizeType size = {{ 10, 10 }};
Image::RegionType region;
region.SetSize(size);
image->SetRegions(region);
image->Allocate();
```

Next we fill the image with increasing values.

```
typedef itk::ImageRegionIterator<Image> Iterator;
Iterator it(image,region);
short pixelValue =0;
it.GoToBegin();
for(; !it.IsAtEnd(); ++it, ++pixelValue)
  {
  it.Set(pixelValue);
  }
```

We can now define the ImageSpatialObject which is templated over the dimension and the pixel type of the image.

```
typedef itk::ImageSpatialObject<2,short> ImageSpatialObject;
ImageSpatialObject::Pointer imageSO = ImageSpatialObject::New();
```

Then we set the itkImage to the ImageSpatialObject by using the `SetImage()` function.

```
imageSO->SetImage(image);
```

At this point we can use `IsInside()`, `ValueAt()` and `DerivativeAt()` functions inherent in SpatialObjects. The `IsInside()` value can be useful when dealing with registration.

```
typedef itk::Point<double,2> Point;
Point insidePoint;
insidePoint.Fill(9);

if( imageSO->IsInside(insidePoint) )
  {
  std::cout << insidePoint << " is inside the image." << std::endl;
  }
```

The `ValueAt()` returns the value of the closest pixel, i.e no interpolation, to a given physical point.

```
double returnedValue;
imageSO->ValueAt(insidePoint,returnedValue);

std::cout << "ValueAt(" << insidePoint << ") = " << returnedValue << std::endl;
```

The derivative at a specified position in space can be computed using the DerivativeAt() function. The first argument is the point in physical coordinates where we are evaluating the derivatives. The second argument is the order of the derivation, and the third argument is the result expressed as a itk::Vector. Derivatives are computed iteratively using finite differences and, like the ValueAt(), no interpolator is used.

```
ImageSpatialObject::OutputVectorType returnedDerivative;
imageSO->DerivativeAt(insidePoint,1,returnedDerivative);
std::cout << "First derivative at " << insidePoint;
std::cout << " = " << returnedDerivative << std::endl;
```

5.5.8 ImageMaskSpatialObject

The source code for this section can be found in the file
Examples/SpatialObjects/ImageMaskSpatialObject.cxx.

An itk::ImageMaskSpatialObject is similar to the itk::ImageSpatialObject and derived from it. However, the main difference is that the IsInside() returns true if the pixel intensity in the image is not zero.

The supported pixel types does not include itk::RGBPixel, itk::RGBAPixel, etc... So far it only allows to manage images of simple types like unsigned short, unsigned int, or itk::Vector. Let's begin by including the appropriate header file.

```
#include "itkImageMaskSpatialObject.h"
```

The ImageMaskSpatialObject is templated over the dimensionality.

```
typedef itk::ImageMaskSpatialObject<3> ImageMaskSpatialObject;
```

Next we create an itk::Image of size 50x50x50 filled with zeros except a bright square in the middle which defines the mask.

```
typedef ImageMaskSpatialObject::PixelType  PixelType;
typedef ImageMaskSpatialObject::ImageType  ImageType;
typedef itk::ImageRegionIterator<ImageType> Iterator;


ImageType::Pointer image = ImageType::New();
ImageType::SizeType size = {{ 50, 50, 50 }};
ImageType::IndexType index = {{ 0, 0, 0 }};
ImageType::RegionType region;

region.SetSize(size);
region.SetIndex(index);
```

```
image->SetRegions( region );
image->Allocate();

PixelType p = itk::NumericTraits< PixelType >::Zero;

image->FillBuffer( p );

ImageType::RegionType insideRegion;
ImageType::SizeType insideSize   = {{ 30, 30, 30 }};
ImageType::IndexType insideIndex = {{ 10, 10, 10 }};
insideRegion.SetSize( insideSize );
insideRegion.SetIndex( insideIndex );

Iterator it( image, insideRegion );
it.GoToBegin();

while( !it.IsAtEnd() )
  {
  it.Set( itk::NumericTraits< PixelType >::max() );
  ++it;
  }
```

Then, we create an ImageMaskSpatialObject.

```
ImageMaskSpatialObject::Pointer maskSO = ImageMaskSpatialObject::New();
```

and we pass the corresponding pointer to the image.

```
maskSO->SetImage(image);
```

We can then test if a physical itk::Point is inside or outside the mask image. This is particularly useful during the registration process when only a part of the image should be used to compute the metric.

```
ImageMaskSpatialObject::PointType  inside;
inside.Fill(20);
std::cout << "Is my point " << inside << " inside my mask? "
  << maskSO->IsInside(inside) << std::endl;
ImageMaskSpatialObject::PointType  outside;
outside.Fill(45);
std::cout << "Is my point " << outside << " outside my mask? "
  << !maskSO->IsInside(outside) << std::endl;
```

5.5.9 LandmarkSpatialObject

The source code for this section can be found in the file
`Examples/SpatialObjects/LandmarkSpatialObject.cxx`.

`itk::LandmarkSpatialObject` contains a list of `itk::SpatialObjectPoint`s which have a position and a color. Let's begin this example by including the appropriate header file.

```
#include "itkLandmarkSpatialObject.h"
```

LandmarkSpatialObject is templated over the dimension of the space.

Here we create a 3-dimensional landmark.

```
  typedef itk::LandmarkSpatialObject<3>  LandmarkType;
  typedef LandmarkType::Pointer          LandmarkPointer;
  typedef itk::SpatialObjectPoint<3>     LandmarkPointType;

  LandmarkPointer landmark = LandmarkType::New();
```

Next, we set some properties of the object like its name and its identification number.

```
  landmark->GetProperty()->SetName("Landmark1");
  landmark->SetId(1);
```

We are now ready to add points into the landmark. We first create a list of SpatialObjectPoint and for each point we set the position and the color.

```
  LandmarkType::PointListType list;

  for( unsigned int i=0; i<5; i++)
    {
    LandmarkPointType p;
    p.SetPosition(i,i+1,i+2);
    p.SetColor(1,0,0,1);
    list.push_back(p);
    }
```

Then we add the list to the object using the `SetPoints()` method.

```
  landmark->SetPoints(list);
```

The current point list can be accessed using the `GetPoints()` method. The method returns a reference to the (STL) list.

```
  unsigned int nPoints = landmark->GetPoints().size();
  std::cout << "Number of Points in the landmark: " << nPoints << std::endl;

  LandmarkType::PointListType::const_iterator it = landmark->GetPoints().begin();
  while(it != landmark->GetPoints().end())
    {
    std::cout << "Position: " << (*it).GetPosition() << std::endl;
    std::cout << "Color: " << (*it).GetColor() << std::endl;
    it++;
    }
```

5.5.10 LineSpatialObject

The source code for this section can be found in the file
`Examples/SpatialObjects/LineSpatialObject.cxx`.

`itk::LineSpatialObject` defines a line in an n-dimensional space. A line is defined as a list of points which compose the line, i.e a polyline. We begin the example by including the appropriate header files.

```
#include "itkLineSpatialObject.h"
#include "itkLineSpatialObjectPoint.h"
```

LineSpatialObject is templated over the dimension of the space. A LineSpatialObject contains a list of LineSpatialObjectPoints. A LineSpatialObjectPoint has a position, $n-1$ normals and a color. Each normal is expressed as a `itk::CovariantVector` of size N.

First, we define some type definitions and we create our line.

```
  typedef itk::LineSpatialObject<3>        LineType;
  typedef LineType::Pointer                LinePointer;
  typedef itk::LineSpatialObjectPoint<3>   LinePointType;
  typedef itk::CovariantVector<double,3>   VectorType;

  LinePointer Line = LineType::New();
```

We create a point list and we set the position of each point in the local coordinate system using the `SetPosition()` method. We also set the color of each point to red.

The two normals are set using the `SetNormal()` function; the first argument is the normal itself and the second argument is the index of the normal.

```
  LineType::PointListType list;

  for(unsigned int i=0; i<3; i++)
    {
```

```
    LinePointType p;
    p.SetPosition(i,i+1,i+2);
    p.SetColor(1,0,0,1);

    VectorType normal1;
    VectorType normal2;
    for(unsigned int j=0;j<3;j++)
      {
      normal1[j]=j;
      normal2[j]=j*2;
      }

    p.SetNormal(normal1,0);
    p.SetNormal(normal2,1);
    list.push_back(p);
    }
```

Next, we set the name of the object using `SetName()`. We also set its identification number with `SetId()` and we set the list of points previously created.

```
  Line->GetProperty()->SetName("Line1");
  Line->SetId(1);
  Line->SetPoints(list);
```

The `GetPoints()` method returns a reference to the internal list of points of the object.

```
   LineType::PointListType pointList = Line->GetPoints();
   std::cout << "Number of points representing the line: ";
   std::cout << pointList.size() << std::endl;
```

Then we can access the points using standard STL iterators. The `GetPosition()` and `GetColor()` functions return respectively the position and the color of the point. Using the GetNormal(unsigned int) function we can access each normal.

```
   LineType::PointListType::const_iterator it = Line->GetPoints().begin();
   while(it != Line->GetPoints().end())
     {
     std::cout << "Position = " << (*it).GetPosition() << std::endl;
     std::cout << "Color = " << (*it).GetColor() << std::endl;
     std::cout << "First normal = " << (*it).GetNormal(0) << std::endl;
     std::cout << "Second normal = " << (*it).GetNormal(1) << std::endl;
     std::cout << std::endl;
     it++;
     }
```

5.5.11 MeshSpatialObject

The source code for this section can be found in the file
`Examples/SpatialObjects/MeshSpatialObject.cxx`.

A `itk::MeshSpatialObject` contains a pointer to an `itk::Mesh` but adds the notion of spatial transformations and parent-child hierarchy. This example shows how to create an `itk::MeshSpatialObject`, use it to form a binary image and how to write the mesh on disk.

Let's begin by including the appropriate header file.

```
#include <itkMeshSpatialObject.h>
#include <itkSpatialObjectReader.h>
#include <itkSpatialObjectWriter.h>
#include <itkSpatialObjectToImageFilter.h>
```

The MeshSpatialObject wraps an `itk::Mesh`, therefore we first create a mesh.

```
  typedef itk::DefaultDynamicMeshTraits< float , 3, 3 > MeshTrait;
  typedef itk::Mesh<float,3,MeshTrait>                  MeshType;
  typedef MeshType::CellTraits                          CellTraits;
  typedef itk::CellInterface< float, CellTraits >       CellInterfaceType;
  typedef itk::TetrahedronCell<CellInterfaceType>       TetraCellType;
  typedef MeshType::PointType                           PointType;
  typedef MeshType::CellType                            CellType;
  typedef CellType::CellAutoPointer                     CellAutoPointer;


  MeshType::Pointer myMesh = MeshType::New();

  MeshType::CoordRepType testPointCoords[4][3]
    = { {0,0,0}, {9,0,0}, {9,9,0}, {0,0,9} };

  unsigned long tetraPoints[4] = {0,1,2,4};
  int i;
  for(i=0; i < 4 ; ++i)
    {
    myMesh->SetPoint(i, PointType(testPointCoords[i]));
    }

  myMesh->SetCellsAllocationMethod(
      MeshType::CellsAllocatedDynamicallyCellByCell );
  CellAutoPointer testCell1;
  testCell1.TakeOwnership(  new TetraCellType );
  testCell1->SetPointIds(tetraPoints);


  myMesh->SetCell(0, testCell1 );
```

We then create a MeshSpatialObject which is templated over the type of mesh previously defined...

```
typedef itk::MeshSpatialObject<MeshType>     MeshSpatialObjectType;
MeshSpatialObjectType::Pointer myMeshSpatialObject =
                                      MeshSpatialObjectType::New();
```

... and pass the Mesh pointer to the MeshSpatialObject

```
myMeshSpatialObject->SetMesh(myMesh);
```

The actual pointer to the passed mesh can be retrieved using the `GetMesh()` function.

```
myMeshSpatialObject->GetMesh();
```

Like any other SpatialObjects. The `GetBoundingBox()`, `ValueAt()`, `IsInside()` functions can be used to access important information.

```
std::cout << "Mesh bounds : " <<
  myMeshSpatialObject->GetBoundingBox()->GetBounds() << std::endl;
MeshSpatialObjectType::PointType myPhysicalPoint;
myPhysicalPoint.Fill(1);
std::cout << "Is my physical point inside? : " <<
  myMeshSpatialObject->IsInside(myPhysicalPoint) << std::endl;
```

Now that we have defined the MeshSpatialObject, we can save the actual mesh using the `itk::SpatialObjectWriter`. To be able to do so, we need to specify the type of Mesh we are writing.

```
typedef itk::SpatialObjectWriter<3,float,MeshTrait> WriterType;
WriterType::Pointer writer = WriterType::New();
```

Then we set the mesh spatial object and the name of the file and call the the `Update()` function.

```
writer->SetInput(myMeshSpatialObject);
writer->SetFileName("myMesh.meta");
writer->Update();
```

Reading the saved mesh is done using the `itk::SpatialObjectReader`. Once again we need to specify the type of mesh we intend to read.

```
typedef itk::SpatialObjectReader<3,float,MeshTrait> ReaderType;
ReaderType::Pointer reader = ReaderType::New();
```

We set the name of the file we want to read and call update

```
  reader->SetFileName("myMesh.meta");
  reader->Update();
```

Next, we show how to create a binary image of a MeshSpatialObject using the itk::SpatialObjectToImageFilter. The resulting image will have ones inside and zeros outside the mesh. First we define and instantiate the SpatialObjectToImageFilter.

```
  typedef itk::Image<unsigned char,3> ImageType;
  typedef itk::GroupSpatialObject<3> GroupType;
  typedef itk::SpatialObjectToImageFilter< GroupType, ImageType >
    SpatialObjectToImageFilterType;
  SpatialObjectToImageFilterType::Pointer imageFilter =
    SpatialObjectToImageFilterType::New();
```

Then we pass the output of the reader, i.e the MeshSpatialObject, to the filter.

```
  imageFilter->SetInput(  reader->GetGroup()  );
```

Finally we trigger the execution of the filter by calling the `Update()` method. Note that depending on the size of the mesh, the computation time can increase significantly.

```
  imageFilter->Update();
```

Then we can get the resulting binary image using the `GetOutput()` function.

```
  ImageType::Pointer myBinaryMeshImage = imageFilter->GetOutput();
```

5.5.12 SurfaceSpatialObject

The source code for this section can be found in the file
`Examples/SpatialObjects/SurfaceSpatialObject.cxx`.

itk::SurfaceSpatialObject defines a surface in n-dimensional space. A SurfaceSpatialObject is defined by a list of points which lie on the surface. Each point has a position and a unique normal. The example begins by including the appropriate header file.

```
#include "itkSurfaceSpatialObject.h"
#include "itkSurfaceSpatialObjectPoint.h"
```

SurfaceSpatialObject is templated over the dimension of the space. A SurfaceSpatialObject contains a list of SurfaceSpatialObjectPoints. A SurfaceSpatialObjectPoint has a position, a normal and a color.

First we define some type definitions

```
  typedef itk::SurfaceSpatialObject<3>        SurfaceType;
  typedef SurfaceType::Pointer                SurfacePointer;
  typedef itk::SurfaceSpatialObjectPoint<3>   SurfacePointType;
  typedef itk::CovariantVector<double,3>      VectorType;

  SurfacePointer Surface = SurfaceType::New();
```

We create a point list and we set the position of each point in the local coordinate system using the `SetPosition()` method. We also set the color of each point to red.

```
  SurfaceType::PointListType list;

  for( unsigned int i=0; i<3; i++)
    {
    SurfacePointType p;
    p.SetPosition(i,i+1,i+2);
    p.SetColor(1,0,0,1);
    VectorType normal;
    for(unsigned int j=0;j<3;j++)
      {
      normal[j]=j;
      }
    p.SetNormal(normal);
    list.push_back(p);
    }
```

Next, we create the surface and set his name using `SetName()`. We also set its Identification number with `SetId()` and we add the list of points previously created.

```
  Surface->GetProperty()->SetName("Surface1");
  Surface->SetId(1);
  Surface->SetPoints(list);
```

The `GetPoints()` method returns a reference to the internal list of points of the object.

```
   SurfaceType::PointListType pointList = Surface->GetPoints();
   std::cout << "Number of points representing the surface: ";
   std::cout << pointList.size() << std::endl;
```

Then we can access the points using standard STL iterators. `GetPosition()` and `GetColor()` functions return respectively the position and the color of the point. `GetNormal()` returns the normal as a `itk::CovariantVector`.

```
   SurfaceType::PointListType::const_iterator it = Surface->GetPoints().begin();
   while(it != Surface->GetPoints().end())
```

```
    {
    std::cout << "Position = " << (*it).GetPosition() << std::endl;
    std::cout << "Normal = " << (*it).GetNormal() << std::endl;
    std::cout << "Color = " << (*it).GetColor() << std::endl;
    std::cout << std::endl;
    it++;
    }
```

5.5.13 TubeSpatialObject

`itk::TubeSpatialObject` represents a base class for the representation of tubular structures using SpatialObjects. The classes `itk::VesselTubeSpatialObject` and `itk::DTITubeSpatialObject` derive from this base class. VesselTubeSpatialObject represents blood vessels extracted for an image and DTITubeSpatialObject is used to represent fiber tracts from diffusion tensor images.

The source code for this section can be found in the file
`Examples/SpatialObjects/TubeSpatialObject.cxx`.

`itk::TubeSpatialObject` defines an n-dimensional tube. A tube is defined as a list of centerline points which have a position, a radius, some normals and other properties. Let's start by including the appropriate header file.

```
#include "itkTubeSpatialObject.h"
#include "itkTubeSpatialObjectPoint.h"
```

TubeSpatialObject is templated over the dimension of the space. A TubeSpatialObject contains a list of TubeSpatialObjectPoints.

First we define some type definitions and we create the tube.

```
  typedef itk::TubeSpatialObject<3>            TubeType;
  typedef TubeType::Pointer                    TubePointer;
  typedef itk::TubeSpatialObjectPoint<3>       TubePointType;
  typedef TubePointType::CovariantVectorType   VectorType;

  TubePointer tube = TubeType::New();
```

We create a point list and we set:

1. The position of each point in the local coordinate system using the `SetPosition()` method.
2. The radius of the tube at this position using `SetRadius()`.
3. The two normals at the tube is set using `SetNormal1()` and `SetNormal2()`.

4. The color of the point is set to red in our case.

```
TubeType::PointListType list;
for( i=0; i<5; i++)
  {
  TubePointType p;
  p.SetPosition(i,i+1,i+2);
  p.SetRadius(1);
  VectorType normal1;
  VectorType normal2;
  for(unsigned int j=0;j<3;j++)
    {
    normal1[j]=j;
    normal2[j]=j*2;
    }

  p.SetNormal1(normal1);
  p.SetNormal2(normal2);
  p.SetColor(1,0,0,1);

  list.push_back(p);
  }
```

Next, we create the tube and set its name using `SetName()`. We also set its identification number with `SetId()` and, at the end, we add the list of points previously created.

```
tube->GetProperty()->SetName("Tube1");
tube->SetId(1);
tube->SetPoints(list);
```

The `GetPoints()` method return a reference to the internal list of points of the object.

```
TubeType::PointListType pointList = tube->GetPoints();
std::cout << "Number of points representing the tube: ";
std::cout << pointList.size() << std::endl;
```

The `ComputeTangentAndNormals()` function computes the normals and the tangent for each point using finite differences.

```
tube->ComputeTangentAndNormals();
```

Then we can access the points using STL iterators. `GetPosition()` and `GetColor()` functions return respectively the position and the color of the point. `GetRadius()` returns the radius at that point. `GetNormal1()` and `GetNormal1()` functions return a `itk::CovariantVector` and `GetTangent()` returns a `itk::Vector`.

```
  TubeType::PointListType::const_iterator it = tube->GetPoints().begin();
  i=0;
  while(it != tube->GetPoints().end())
    {
    std::cout << std::endl;
    std::cout << "Point #" << i << std::endl;
    std::cout << "Position: " << (*it).GetPosition() << std::endl;
    std::cout << "Radius: " << (*it).GetRadius() << std::endl;
    std::cout << "Tangent: " << (*it).GetTangent() << std::endl;
    std::cout << "First Normal: " << (*it).GetNormal1() << std::endl;
    std::cout << "Second Normal: " << (*it).GetNormal2() << std::endl;
    std::cout << "Color = " << (*it).GetColor() << std::endl;
    it++;
    i++;
    }
```

VesselTubeSpatialObject

The source code for this section can be found in the file
`Examples/SpatialObjects/VesselTubeSpatialObject.cxx`.

`itk::VesselTubeSpatialObject` derives from `itk::TubeSpatialObject`. It represents a blood vessel segmented from an image. A VesselTubeSpatialObject is described as a list of centerline points which have a position, a radius, normals,

Let's start by including the appropriate header file.

```
#include "itkVesselTubeSpatialObject.h"
#include "itkVesselTubeSpatialObjectPoint.h"
```

VesselTubeSpatialObject is templated over the dimension of the space. A VesselTubeSpatialObject contains a list of VesselTubeSpatialObjectPoints.

First we define some type definitions and we create the tube.

```
  typedef itk::VesselTubeSpatialObject<3>          VesselTubeType;
  typedef itk::VesselTubeSpatialObjectPoint<3>     VesselTubePointType;

  VesselTubeType::Pointer VesselTube = VesselTubeType::New();
```

We create a point list and we set:

1. The position of each point in the local coordinate system using the `SetPosition()` method.

2. The radius of the tube at this position using `SetRadius()`.

3. The medialness value describing how the point lies in the middle of the vessel using `SetMedialness()`.
4. The ridgeness value describing how the point lies on the ridge using `SetRidgeness()`.
5. The branchness value describing if the point is a branch point using `SetBranchness()`.
6. The three alpha values corresponding to the eigenvalues of the Hessian using `SetAlpha1()`,`SetAlpha2()` and `SetAlpha3()`.
7. The mark value using `SetMark()`.
8. The color of the point is set to red in this example with an opacity of 1.

```
VesselTubeType::PointListType list;
for( i=0; i<5; i++)
  {
  VesselTubePointType p;
  p.SetPosition(i,i+1,i+2);
  p.SetRadius(1);
  p.SetAlpha1(i);
  p.SetAlpha2(i+1);
  p.SetAlpha3(i+2);
  p.SetMedialness(i);
  p.SetRidgeness(i);
  p.SetBranchness(i);
  p.SetMark(true);
  p.SetColor(1,0,0,1);
  list.push_back(p);
  }
```

Next, we create the tube and set its name using `SetName()`. We also set its identification number with `SetId()` and, at the end, we add the list of points previously created.

```
VesselTube->GetProperty()->SetName("VesselTube");
VesselTube->SetId(1);
VesselTube->SetPoints(list);
```

The `GetPoints()` method return a reference to the internal list of points of the object.

```
VesselTubeType::PointListType pointList = VesselTube->GetPoints();
std::cout << "Number of points representing the blood vessel: ";
std::cout << pointList.size() << std::endl;
```

Then we can access the points using STL iterators. `GetPosition()` and `GetColor()` functions return respectively the position and the color of the point.

```
  VesselTubeType::PointListType::const_iterator
            it = VesselTube->GetPoints().begin();
  i=0;
  while(it != VesselTube->GetPoints().end())
    {
    std::cout << std::endl;
    std::cout << "Point #" << i << std::endl;
    std::cout << "Position: " << (*it).GetPosition() << std::endl;
    std::cout << "Radius: " << (*it).GetRadius() << std::endl;
    std::cout << "Medialness: " << (*it).GetMedialness() << std::endl;
    std::cout << "Ridgeness: " << (*it).GetRidgeness() << std::endl;
    std::cout << "Branchness: " << (*it).GetBranchness() << std::endl;
    std::cout << "Mark: " << (*it).GetMark() << std::endl;
    std::cout << "Alpha1: " << (*it).GetAlpha1() << std::endl;
    std::cout << "Alpha2: " << (*it).GetAlpha2() << std::endl;
    std::cout << "Alpha3: " << (*it).GetAlpha3() << std::endl;
    std::cout << "Color = " << (*it).GetColor() << std::endl;
    it++;
    i++;
    }
```

DTITubeSpatialObject

The source code for this section can be found in the file
`Examples/SpatialObjects/DTITubeSpatialObject.cxx`.

`itk::DTITubeSpatialObject` derives from `itk::TubeSpatialObject`. It represents a fiber tracts from Diffusion Tensor Imaging. A DTITubeSpatialObject is described as a list of centerline points which have a position, a radius, normals, the fractional anisotropy (FA) value, the ADC value, the geodesic anisotropy (GA) value, the eigenvalues and vectors as well as the full tensor matrix.

Let's start by including the appropriate header file.

```
#include "itkDTITubeSpatialObject.h"
#include "itkDTITubeSpatialObjectPoint.h"
```

DTITubeSpatialObject is templated over the dimension of the space. A DTITubeSpatialObject contains a list of DTITubeSpatialObjectPoints.

First we define some type definitions and we create the tube.

```
  typedef itk::DTITubeSpatialObject<3>            DTITubeType;
  typedef itk::DTITubeSpatialObjectPoint<3>       DTITubePointType;

  DTITubeType::Pointer dtiTube = DTITubeType::New();
```

We create a point list and we set:

1. The position of each point in the local coordinate system using the `SetPosition()` method.
2. The radius of the tube at this position using `SetRadius()`.
3. The FA value using `AddField(DTITubePointType::FA)`.
4. The ADC value using `AddField(DTITubePointType::ADC)`.
5. The GA value using `AddField(DTITubePointType::GA)`.
6. The full tensor matrix supposed to be symmetric definite positive value using `SetTensorMatrix()`.
7. The color of the point is set to red in our case.

```
DTITubeType::PointListType list;
for( i=0; i<5; i++)
  {
  DTITubePointType p;
  p.SetPosition(i,i+1,i+2);
  p.SetRadius(1);
  p.AddField(DTITubePointType::FA,i);
  p.AddField(DTITubePointType::ADC,2*i);
  p.AddField(DTITubePointType::GA,3*i);
  p.AddField("Lambda1",4*i);
  p.AddField("Lambda2",5*i);
  p.AddField("Lambda3",6*i);
  float* v = new float[6];
  for(unsigned int k=0;k<6;k++)
    {
    v[k] = k;
    }
  p.SetTensorMatrix(v);
  delete v;
  p.SetColor(1,0,0,1);
  list.push_back(p);
  }
```

Next, we create the tube and set its name using `SetName()`. We also set its identification number with `SetId()` and, at the end, we add the list of points previously created.

```
dtiTube->GetProperty()->SetName("DTITube");
dtiTube->SetId(1);
dtiTube->SetPoints(list);
```

The `GetPoints()` method return a reference to the internal list of points of the object.

```
  DTITubeType::PointListType pointList = dtiTube->GetPoints();
  std::cout << "Number of points representing the fiber tract: ";
  std::cout << pointList.size() << std::endl;
```

Then we can access the points using STL iterators. `GetPosition()` and `GetColor()` functions return respectively the position and the color of the point.

```
  DTITubeType::PointListType::const_iterator it = dtiTube->GetPoints().begin();
  i=0;
  while(it != dtiTube->GetPoints().end())
    {
    std::cout << std::endl;
    std::cout << "Point #" << i << std::endl;
    std::cout << "Position: " << (*it).GetPosition() << std::endl;
    std::cout << "Radius: " << (*it).GetRadius() << std::endl;
    std::cout << "FA: " << (*it).GetField(DTITubePointType::FA) << std::endl;
    std::cout << "ADC: " << (*it).GetField(DTITubePointType::ADC) << std::endl;
    std::cout << "GA: " << (*it).GetField(DTITubePointType::GA) << std::endl;
    std::cout << "Lambda1: " << (*it).GetField("Lambda1") << std::endl;
    std::cout << "Lambda2: " << (*it).GetField("Lambda2") << std::endl;
    std::cout << "Lambda3: " << (*it).GetField("Lambda3") << std::endl;
    std::cout << "TensorMatrix: " << (*it).GetTensorMatrix()[0] << " : ";
    std::cout << (*it).GetTensorMatrix()[1] << " : ";
    std::cout << (*it).GetTensorMatrix()[2] << " : ";
    std::cout << (*it).GetTensorMatrix()[3] << " : ";
    std::cout << (*it).GetTensorMatrix()[4] << " : ";
    std::cout << (*it).GetTensorMatrix()[5] << std::endl;
    std::cout << "Color = " << (*it).GetColor() << std::endl;
    it++;
    i++;
    }
```

5.6 SceneSpatialObject

The source code for this section can be found in the file
`Examples/SpatialObjects/SceneSpatialObject.cxx`.

This example describes how to use the `itk::SceneSpatialObject`. A SceneSpatialObject contains a collection of SpatialObjects. This example begins by including the appropriate header file.

```
#include "itkSceneSpatialObject.h"
```

An SceneSpatialObject is templated over the dimension of the space which requires all the objects referenced by the SceneSpatialObject to have the same dimension.

First we define some type definitions and we create the SceneSpatialObject.

```
typedef itk::SceneSpatialObject<3> SceneSpatialObjectType;
SceneSpatialObjectType::Pointer scene = SceneSpatialObjectType::New();
```

Then we create two `itk::EllipseSpatialObject`s.

```
typedef itk::EllipseSpatialObject<3> EllipseType;
EllipseType::Pointer ellipse1 = EllipseType::New();
ellipse1->SetRadius(1);
ellipse1->SetId(1);
EllipseType::Pointer ellipse2 = EllipseType::New();
ellipse2->SetId(2);
ellipse2->SetRadius(2);
```

Then we add the two ellipses into the SceneSpatialObject.

```
scene->AddSpatialObject(ellipse1);
scene->AddSpatialObject(ellipse2);
```

We can query the number of object in the SceneSpatialObject with the `GetNumberOfObjects()` function. This function takes two optional arguments: the depth at which we should count the number of objects (default is set to infinity) and the name of the object to count (default is set to NULL). This allows the user to count, for example, only ellipses.

```
std::cout << "Number of objects in the SceneSpatialObject = ";
std::cout << scene->GetNumberOfObjects() << std::endl;
```

The `GetObjectById()` returns the first object in the SceneSpatialObject that has the specified identification number.

```
std::cout << "Object in the SceneSpatialObject with an ID == 2: " << std::endl;
scene->GetObjectById(2)->Print(std::cout);
```

Objects can also be removed from the SceneSpatialObject using the `RemoveSpatialObject()` function.

```
scene->RemoveSpatialObject(ellipse1);
```

The list of current objects in the SceneSpatialObject can be retrieved using the `GetObjects()` method. Like the `GetNumberOfObjects()` method, `GetObjects()` can take two arguments: a search depth and a matching name.

```
  SceneSpatialObjectType::ObjectListType * myObjectList =  scene->GetObjects();
  std::cout << "Number of objects in the SceneSpatialObject = ";
  std::cout << myObjectList->size() << std::endl;
```

In some cases, it is useful to define the hierarchy by using `ParentId()` and the current identification number. This results in having a flat list of SpatialObjects in the SceneSpatialObject. Therefore, the SceneSpatialObject provides the `FixHierarchy()` method which reorganizes the Parent-Child hierarchy based on identification numbers.

```
  scene->FixHierarchy();
```

The scene can also be cleared by using the `Clear()` function.

```
  scene->Clear();
```

5.7 Read/Write SpatialObjects

The source code for this section can be found in the file
`Examples/SpatialObjects/ReadWriteSpatialObject.cxx`.

Reading and writing SpatialObjects is a fairly simple task. The classes `itk::SpatialObjectReader` and `itk::SpatialObjectWriter` are used to read and write these objects, respectively. (Note these classes make use of the MetaIO auxiliary I/O routines and therefore have a `.meta` file suffix.)

We begin this example by including the appropriate header files.

```
#include "itkSpatialObjectWriter.h"
#include "itkSpatialObjectReader.h"
```

Next, we create a SpatialObjectWriter that is templated over the dimension of the object(s) we want to write.

```
  typedef itk::SpatialObjectWriter<3> WriterType;
  WriterType::Pointer writer = WriterType::New();
```

For this example, we create an `itk::EllipseSpatialObject`.

```
  typedef itk::EllipseSpatialObject<3> EllipseType;
  EllipseType::Pointer ellipse = EllipseType::New();
  ellipse->SetRadius(3);
```

Finally, we set to the writer the object to write using the SetInput() method and we set the name of the file with SetFileName() and call the Update() method to actually write the information.

```
writer->SetInput(ellipse);
writer->SetFileName("ellipse.meta");
writer->Update();
```

Now we are ready to open the freshly created object. We first create a SpatialObjectReader which is also templated over the dimension of the object in the file. This means that the file should contain only objects with the same dimension.

```
typedef itk::SpatialObjectReader<3> ReaderType;
ReaderType::Pointer reader = ReaderType::New();
```

Next we set the name of the file to read using SetFileName() and we call the Update() method to read the file.

```
reader->SetFileName("ellipse.meta");
reader->Update();
```

To get the objects in the file you can call the GetScene() method or the GetGroup() method. GetScene() returns an pointer to a itk::SceneSpatialObject.

```
ReaderType::SceneType * scene = reader->GetScene();
std::cout << "Number of objects in the scene: ";
std::cout << scene->GetNumberOfObjects() << std::endl;
ReaderType::GroupType * group = reader->GetGroup();
std::cout << "Number of objects in the group: ";
std::cout << group->GetNumberOfChildren() << std::endl;
```

5.8 Statistics Computation via SpatialObjects

The source code for this section can be found in the file
Examples/SpatialObjects/SpatialObjectToImageStatisticsCalculator.cxx.

This example describes how to use the itk::SpatialObjectToImageStatisticsCalculator to compute statistics of an itk::Image only in a region defined inside a given itk::SpatialObject.

```
#include "itkSpatialObjectToImageStatisticsCalculator.h"
```

We first create a test image using the itk::RandomImageSource

```
typedef itk::Image<unsigned char,2> ImageType;
typedef itk::RandomImageSource<ImageType> RandomImageSourceType;
RandomImageSourceType::Pointer randomImageSource = RandomImageSourceType::New();
unsigned long size[2];
size[0] = 10;
size[1] = 10;
randomImageSource->SetSize(size);
randomImageSource->Update();
ImageType::Pointer image = randomImageSource->GetOutput();
```

Next we create an itk::EllipseSpatialObject with a radius of 2. We also move the ellipse to the center of the image by increasing the offset of the IndexToObjectTransform.

```
typedef itk::EllipseSpatialObject<2> EllipseType;
EllipseType::Pointer ellipse = EllipseType::New();
ellipse->SetRadius(2);
EllipseType::VectorType offset;
offset.Fill(5);
ellipse->GetIndexToObjectTransform()->SetOffset(offset);
ellipse->ComputeObjectToParentTransform();
```

Then we can create the itk::SpatialObjectToImageStatisticsCalculator

```
typedef itk::SpatialObjectToImageStatisticsCalculator<
  ImageType, EllipseType > CalculatorType;
CalculatorType::Pointer calculator = CalculatorType::New();
```

We pass a pointer to the image to the calculator.

```
calculator->SetImage(image);
```

And we also pass the SpatialObject. The statistics will be computed inside the SpatialObject (Internally the calculator is using the `IsInside()` function).

```
calculator->SetSpatialObject(ellipse);
```

At the end we trigger the computation via the `Update()` function and we can retrieve the mean and the covariance matrix using `GetMean()` and `GetCovarianceMatrix()` respectively.

```
calculator->Update();
std::cout << "Sample mean = " << calculator->GetMean() << std::endl ;
std::cout << "Sample covariance = " << calculator->GetCovarianceMatrix();
```

CHAPTER

SIX

Filtering

This chapter introduces the most commonly used filters found in the toolkit. Most of these filters are intended to process images. They will accept one or more images as input and will produce one or more images as output. ITK is based on a data pipeline architecture in which the output of one filter is passed as input to another filter. (See Section 3.5 on page 28 for more information.)

6.1 Thresholding

The thresholding operation is used to change or identify pixel values based on specifying one or more values (called the *threshold* value). The following sections describe how to perform thresholding operations using ITK.

6.1.1 Binary Thresholding

The source code for this section can be found in the file
`Examples/Filtering/BinaryThresholdImageFilter.cxx`.

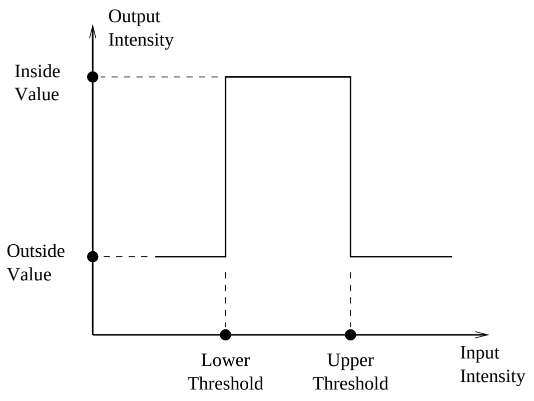


Figure 6.1: Transfer function of the BinaryThresholdImage-Filter.

This example illustrates the use of the binary threshold image filter. This filter is used to transform an image into a binary image by changing the pixel values according to the rule illustrated in Figure 6.1. The user defines two thresholds—Upper and Lower—and two intensity values—Inside and Outside. For each pixel in the input image, the value of the pixel is compared with the lower and upper thresholds. If the pixel value is inside the range defined by $[Lower, Upper]$ the output pixel is assigned the InsideValue. Otherwise the output pixels are assigned to the OutsideValue. Thresholding is commonly applied as the last operation of a segmentation pipeline.

The first step required to use the `itk::BinaryThresholdImageFilter` is to include its header file.

```
#include "itkBinaryThresholdImageFilter.h"
```

The next step is to decide which pixel types to use for the input and output images.

```
  typedef  unsigned char  InputPixelType;
  typedef  unsigned char  OutputPixelType;
```

The input and output image types are now defined using their respective pixel types and dimensions.

```
  typedef itk::Image< InputPixelType,  2 >   InputImageType;
  typedef itk::Image< OutputPixelType, 2 >   OutputImageType;
```

The filter type can be instantiated using the input and output image types defined above.

```
  typedef itk::BinaryThresholdImageFilter<
               InputImageType, OutputImageType >  FilterType;
```

An `itk::ImageFileReader` class is also instantiated in order to read image data from a file. (See Section 7 on page 263 for more information about reading and writing data.)

```
  typedef itk::ImageFileReader< InputImageType >  ReaderType;
```

An itk::ImageFileWriter is instantiated in order to write the output image to a file.

```
typedef itk::ImageFileWriter< InputImageType >  WriterType;
```

Both the filter and the reader are created by invoking their New() methods and assigning the result to itk::SmartPointers.

```
ReaderType::Pointer reader = ReaderType::New();
FilterType::Pointer filter = FilterType::New();
```

The image obtained with the reader is passed as input to the BinaryThresholdImageFilter.

```
filter->SetInput( reader->GetOutput() );
```

The method SetOutsideValue() defines the intensity value to be assigned to those pixels whose intensities are outside the range defined by the lower and upper thresholds. The method SetInsideValue() defines the intensity value to be assigned to pixels with intensities falling inside the threshold range.

```
filter->SetOutsideValue( outsideValue );
filter->SetInsideValue(  insideValue  );
```

The methods SetLowerThreshold() and SetUpperThreshold() define the range of the input image intensities that will be transformed into the InsideValue. Note that the lower and upper thresholds are values of the type of the input image pixels, while the inside and outside values are of the type of the output image pixels.

```
filter->SetLowerThreshold( lowerThreshold );
filter->SetUpperThreshold( upperThreshold );
```

The execution of the filter is triggered by invoking the Update() method. If the filter's output has been passed as input to subsequent filters, the Update() call on any posterior filters in the pipeline will indirectly trigger the update of this filter.

```
filter->Update();
```

Figure 6.2 illustrates the effect of this filter on a MRI proton density image of the brain. This figure shows the limitations of this filter for performing segmentation by itself. These limitations are particularly noticeable in noisy images and in images lacking spatial uniformity as is the case with MRI due to field bias.

The following classes provide similar functionality:

- itk::ThresholdImageFilter

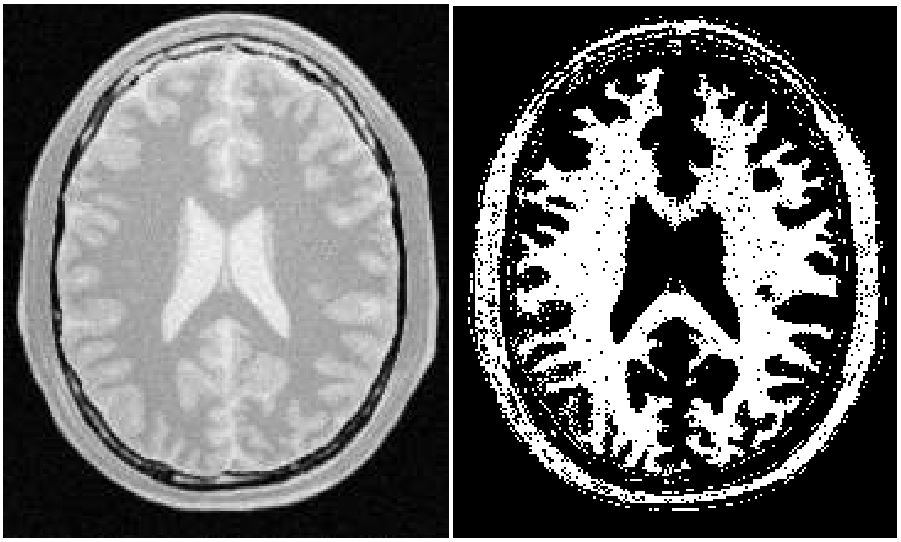


Figure 6.2: Effect of the BinaryThresholdImageFilter on a slice from a MRI proton density image of the brain.

6.1.2 General Thresholding

The source code for this section can be found in the file
`Examples/Filtering/ThresholdImageFilter.cxx`.

This example illustrates the use of the `itk::ThresholdImageFilter`. This filter can be used to transform the intensity levels of an image in three different ways.

- First, the user can define a single threshold. Any pixels with values below this threshold will be replaced by a user defined value, called here the `OutsideValue`. Pixels with values above the threshold remain unchanged. This type of thresholding is illustrated in Figure 6.3.
- Second, the user can define a particular threshold such that all the pixels with values above the threshold will be replaced by the `OutsideValue`. Pixels with values below the threshold remain unchanged. This is illustrated in Figure 6.4.
- Third, the user can provide two thresholds. All the pixels with intensity values inside the range defined by the two thresholds will remain unchanged. Pixels with values outside this range will be assigned to the `OutsideValue`. This is illustrated in Figure 6.5.

The following methods choose among the three operating modes of the filter.

- `ThresholdBelow()`

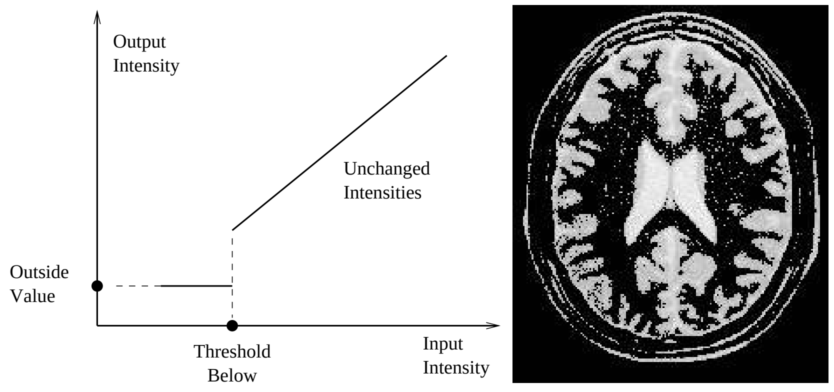


Figure 6.3: ThresholdImageFilter using the threshold-below mode.

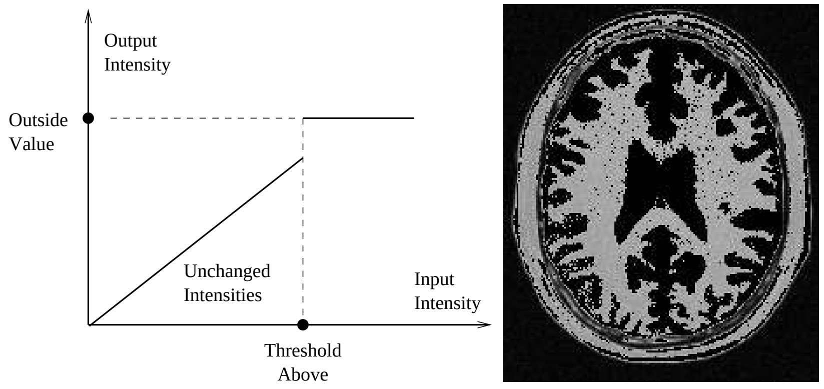


Figure 6.4: ThresholdImageFilter using the threshold-above mode.

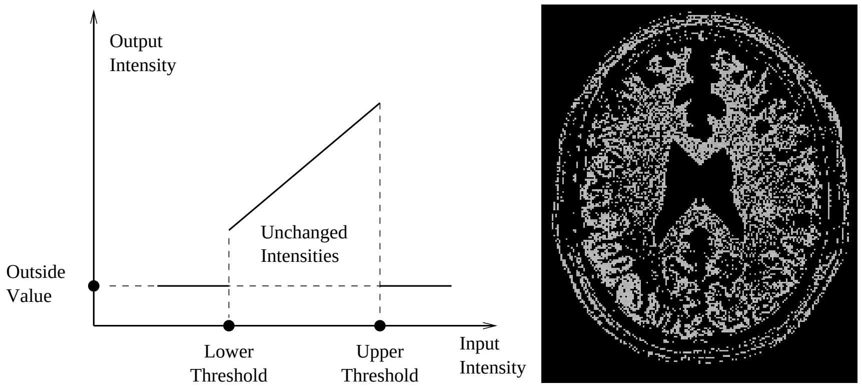


Figure 6.5: ThresholdImageFilter using the threshold-outside mode.

- `ThresholdAbove()`
- `ThresholdOutside()`

The first step required to use this filter is to include its header file.

```
#include "itkThresholdImageFilter.h"
```

Then we must decide what pixel type to use for the image. This filter is templated over a single image type because the algorithm only modifies pixel values outside the specified range, passing the rest through unchanged.

```
  typedef  unsigned char  PixelType;
```

The image is defined using the pixel type and the dimension.

```
  typedef itk::Image< PixelType,  2 >   ImageType;
```

The filter can be instantiated using the image type defined above.

```
  typedef itk::ThresholdImageFilter< ImageType >  FilterType;
```

An `itk::ImageFileReader` class is also instantiated in order to read image data from a file.

```
  typedef itk::ImageFileReader< ImageType >  ReaderType;
```

An `itk::ImageFileWriter` is instantiated in order to write the output image to a file.

```
  typedef itk::ImageFileWriter< ImageType >  WriterType;
```

Both the filter and the reader are created by invoking their `New()` methods and assigning the result to SmartPointers.

```
  ReaderType::Pointer reader = ReaderType::New();
  FilterType::Pointer filter = FilterType::New();
```

The image obtained with the reader is passed as input to the `itk::ThresholdImageFilter`.

```
  filter->SetInput( reader->GetOutput() );
```

The method `SetOutsideValue()` defines the intensity value to be assigned to those pixels whose intensities are outside the range defined by the lower and upper thresholds.

```
  filter->SetOutsideValue( 0 );
```

The method `ThresholdBelow()` defines the intensity value below which pixels of the input image will be changed to the `OutsideValue`.

```
  filter->ThresholdBelow( 180 );
```

The filter is executed by invoking the `Update()` method. If the filter is part of a larger image processing pipeline, calling `Update()` on a downstream filter will also trigger update of this filter.

```
  filter->Update();
```

The output of this example is shown in Figure 6.3. The second operating mode of the filter is now enabled by calling the method `ThresholdAbove()`.

```
  filter->ThresholdAbove( 180 );
  filter->Update();
```

Updating the filter with this new setting produces the output shown in Figure 6.4. The third operating mode of the filter is enabled by calling `ThresholdOutside()`.

```
  filter->ThresholdOutside( 170,190 );
  filter->Update();
```

The output of this third, “band-pass” thresholding mode is shown in Figure 6.5.

The examples in this section also illustrate the limitations of the thresholding filter for performing segmentation by itself. These limitations are particularly noticeable in noisy images and in images lacking spatial uniformity, as is the case with MRI due to field bias.

The following classes provide similar functionality:

- `itk::BinaryThresholdImageFilter`

6.2 Edge Detection

6.2.1 Canny Edge Detection

The source code for this section can be found in the file
`Examples/Filtering/CannyEdgeDetectionImageFilter.cxx`.

This example introduces the use of the `itk::CannyEdgeDetectionImageFilter`. This filter is widely used for edge detection since it is the optimal solution satisfying the constraints of good sensitivity, localization and noise robustness.

The first step required for using this filter is to include its header file

```
#include "itkCannyEdgeDetectionImageFilter.h"
```

This filter operates on image of pixel type float. It is then necessary to cast the type of the input images that are usually of integer type. The `itk::CastImageFilter` is used here for that purpose. Its image template parameters are defined for casting from the input type to the float type using for processing.

```
  typedef itk::CastImageFilter< CharImageType, RealImageType> CastToRealFilterType;
```

The `itk::CannyEdgeDetectionImageFilter` is instantiated using the float image type.

6.3 Casting and Intensity Mapping

The filters discussed in this section perform pixel-wise intensity mappings. Casting is used to convert one pixel type to another, while intensity mappings also take into account the different intensity ranges of the pixel types.

6.3.1 Linear Mappings

The source code for this section can be found in the file
`Examples/Filtering/CastingImageFilters.cxx`.

Due to the use of Generic Programming in the toolkit, most types are resolved at compile-time. Few decisions regarding type conversion are left to run-time. It is up to the user to anticipate the pixel type-conversions required in the data pipeline. In medical imaging applications it is usually not desirable to use a general pixel type since this may result in the loss of valuable information.

This section introduces the mechanisms for explicit casting of images that flow through the pipeline. The following four filters are treated in this section: `itk::CastImageFilter`, `itk::RescaleIntensityImageFilter`, `itk::ShiftScaleImageFilter` and `itk::NormalizeImageFilter`. These filters are not directly related to each other except that they all modify pixel values. They are presented together here with the purpose of comparing their individual features.

The CastImageFilter is a very simple filter that acts pixel-wise on an input image, casting every pixel to the type of the output image. Note that this filter does not perform any arithmetic operation on the intensities. Applying CastImageFilter is equivalent to performing a `C-Style` cast on every pixel.

```
outputPixel = static_cast<OutputPixelType>( inputPixel )
```

The RescaleIntensityImageFilter linearly scales the pixel values in such a way that the minimum and maximum values of the input are mapped to minimum and maximum values provided by the user. This is a typical process for forcing the dynamic range of the image to fit within a particular scale and is common for image display. The linear transformation applied by this filter can be expressed as

$$outputPixel = (inputPixel - inpMin) \times \frac{(outMax - outMin)}{(inpMax - inpMin)} + outMin$$

The ShiftScaleImageFilter also applies a linear transformation to the intensities of the input image, but the transformation is specified by the user in the form of a multiplying factor and a value to be added. This can be expressed as

$$outputPixel = (inputPixel + Shift) \times Scale$$

.

The parameters of the linear transformation applied by the NormalizeImageFilter are computed internally such that the statistical distribution of gray levels in the output image have zero mean and a variance of one. This intensity correction is particularly useful in registration applications as a preprocessing step to the evaluation of mutual information metrics. The linear transformation of NormalizeImageFilter is given as

$$outputPixel = \frac{(inputPixel - mean)}{\sqrt{variance}}$$

As usual, the first step required to use these filters is to include their header files.

```
#include "itkCastImageFilter.h"
#include "itkRescaleIntensityImageFilter.h"
#include "itkShiftScaleImageFilter.h"
#include "itkNormalizeImageFilter.h"
```

Let’s define pixel types for the input and output images.

```
  typedef   unsigned char    InputPixelType;
  typedef   float            OutputPixelType;
```

Then, the input and output image types are defined.

```
  typedef itk::Image< InputPixelType,  3 >   InputImageType;
  typedef itk::Image< OutputPixelType, 3 >   OutputImageType;
```

The filters are instantiated using the defined image types.

```
  typedef itk::CastImageFilter<
               InputImageType, OutputImageType >  CastFilterType;

  typedef itk::RescaleIntensityImageFilter<
               InputImageType, OutputImageType >  RescaleFilterType;

  typedef itk::ShiftScaleImageFilter<
               InputImageType, OutputImageType >  ShiftScaleFilterType;

  typedef itk::NormalizeImageFilter<
               InputImageType, OutputImageType >  NormalizeFilterType;
```

Object filters are created by invoking the `New()` operator and assigning the result to `itk::SmartPointer`s.

```
  CastFilterType::Pointer       castFilter       = CastFilterType::New();
  RescaleFilterType::Pointer    rescaleFilter    = RescaleFilterType::New();
  ShiftScaleFilterType::Pointer shiftFilter      = ShiftScaleFilterType::New();
  NormalizeFilterType::Pointer  normalizeFilter = NormalizeFilterType::New();
```

The output of a reader filter (whose creation is not shown here) is now connected as input to the various casting filters.

```
  castFilter->SetInput(       reader->GetOutput() );
  shiftFilter->SetInput(      reader->GetOutput() );
  rescaleFilter->SetInput(    reader->GetOutput() );
  normalizeFilter->SetInput( reader->GetOutput() );
```

Next we proceed to setup the parameters required by each filter. The CastImageFilter and the NormalizeImageFilter do not require any parameters. The RescaleIntensityImageFilter, on the other hand, requires the user to provide the desired minimum and maximum pixel values of the output image. This is done by using the `SetOutputMinimum()` and `SetOutputMaximum()` methods as illustrated below.

```
  rescaleFilter->SetOutputMinimum(  10 );
  rescaleFilter->SetOutputMaximum( 250 );
```

The ShiftScaleImageFilter requires a multiplication factor (scale) and a post-scaling additive value (shift). The methods `SetScale()` and `SetShift()` are used, respectively, to set these values.

```
  shiftFilter->SetScale( 1.2 );
  shiftFilter->SetShift( 25 );
```

Finally, the filters are executed by invoking the `Update()` method.

```
  castFilter->Update();
  shiftFilter->Update();
  rescaleFilter->Update();
  normalizeFilter->Update();
```

6.3.2 Non Linear Mappings

The following filter can be seen as a variant of the casting filters. Its main difference is the use of a smooth and continuous transition function of non-linear form.

The source code for this section can be found in the file
`Examples/Filtering/SigmoidImageFilter.cxx`.

The `itk::SigmoidImageFilter` is commonly used as an intensity transform. It maps a specific range of intensity values into a new intensity range by making a very smooth and continuous transition in the borders of the range. Sigmoids are widely used as a mechanism for focusing attention on a particular set of values and progressively attenuating the values outside that range. In order to extend the flexibility of the Sigmoid filter, its implementation in ITK includes four parameters that can be tuned to select its input and output intensity ranges. The following equation represents the Sigmoid intensity transformation, applied pixel-wise.

$$I' = (Max - Min) \cdot \frac{1}{\left(1 + e^{-\left(\frac{I-\beta}{\alpha}\right)}\right)} + Min \tag{6.1}$$

In the equation above, I is the intensity of the input pixel, I' the intensity of the output pixel, Min, Max are the minimum and maximum values of the output image, α defines the width of the input intensity range, and β defines the intensity around which the range is centered. Figure 6.6 illustrates the significance of each parameter.

This filter will work on images of any dimension and will take advantage of multiple processors when available.

The header file corresponding to this filter should be included first.

```
#include "itkSigmoidImageFilter.h"
```

Then pixel and image types for the filter input and output must be defined.

```
  typedef   unsigned char   InputPixelType;
```

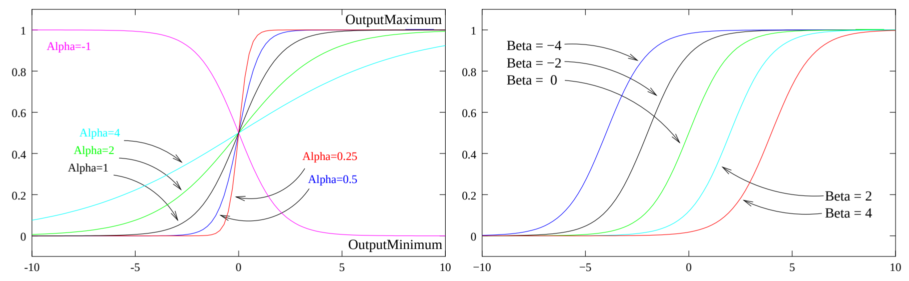


Figure 6.6: Effects of the various parameters in the SigmoidImageFilter. The alpha parameter defines the width of the intensity window. The beta parameter defines the center of the intensity window.

```
typedef   unsigned char  OutputPixelType;

typedef itk::Image< InputPixelType,  2 >   InputImageType;
typedef itk::Image< OutputPixelType, 2 >   OutputImageType;
```

Using the image types, we instantiate the filter type and create the filter object.

```
typedef itk::SigmoidImageFilter<
             InputImageType, OutputImageType >  SigmoidFilterType;
SigmoidFilterType::Pointer sigmoidFilter = SigmoidFilterType::New();
```

The minimum and maximum values desired in the output are defined using the methods `SetOutputMinimum()` and `SetOutputMaximum()`.

```
sigmoidFilter->SetOutputMinimum(   outputMinimum  );
sigmoidFilter->SetOutputMaximum(   outputMaximum  );
```

The coefficients α and β are set with the methods `SetAlpha()` and `SetBeta()`. Note that α is proportional to the width of the input intensity window. As rule of thumb, we may say that the window is the interval $[-3\alpha, 3\alpha]$. The boundaries of the intensity window are not sharp. The α curve approaches its extrema smoothly, as shown in Figure 6.6. You may want to think about this in the same terms as when taking a range in a population of measures by defining an interval of $[-3\sigma, +3\sigma]$ around the population mean.

```
sigmoidFilter->SetAlpha(  alpha  );
sigmoidFilter->SetBeta(   beta   );
```

The input to the SigmoidImageFilter can be taken from any other filter, such as an image file reader, for example. The output can be passed down the pipeline to other filters, like an image file writer. An update call on any downstream filter will trigger the execution of the Sigmoid filter.

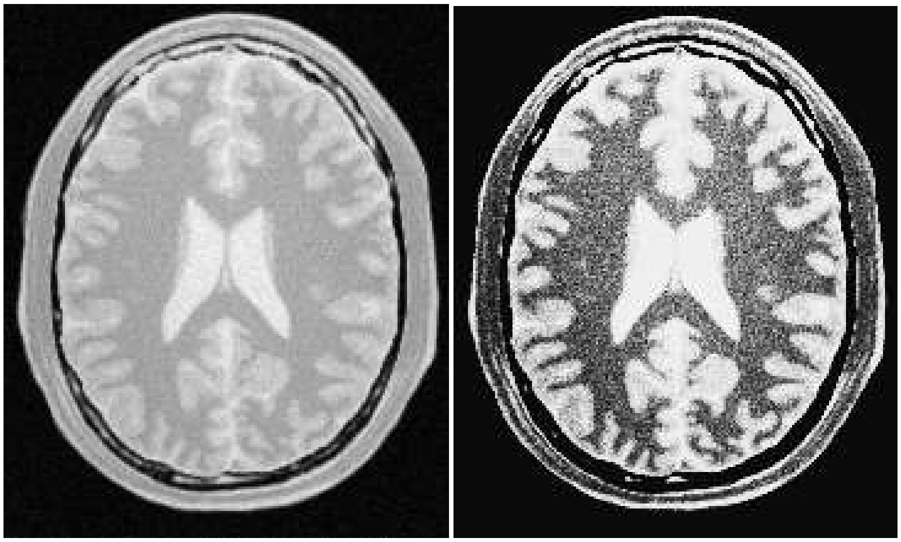


Figure 6.7: Effect of the Sigmoid filter on a slice from a MRI proton density brain image.

```
sigmoidFilter->SetInput( reader->GetOutput() );
writer->SetInput( sigmoidFilter->GetOutput() );
writer->Update();
```

Figure 6.7 illustrates the effect of this filter on a slice of MRI brain image using the following parameters.

- Minimum = 10
- Maximum = 240
- $\alpha = 10$
- $\beta = 170$

As can be seen from the figure, the intensities of the white matter were expanded in their dynamic range, while intensity values lower than $\beta - 3\alpha$ and higher than $\beta + 3\alpha$ became progressively mapped to the minimum and maximum output values. This is the way in which a Sigmoid can be used for performing smooth intensity windowing.

Note that both α and β can be positive and negative. A negative α will have the effect of *negating* the image. This is illustrated on the left side of Figure 6.6. An application of the Sigmoid filter as preprocessing for segmentation is presented in Section 9.3.1.

Sigmoid curves are common in the natural world. They represent the plot of sensitivity to a stimulus. They are also the integral curve of the Gaussian and, therefore, appear naturally as the response to signals whose distribution is Gaussian.

6.4 Gradients

Computation of gradients is a fairly common operation in image processing. The term "gradient" may refer in some contexts to the gradient vectors and in others to the magnitude of the gradient vectors. ITK filters attempt to reduce this ambiguity by including the *magnitude* term when appropriate. ITK provides filters for computing both the image of gradient vectors and the image of magnitudes.

6.4.1 Gradient Magnitude

The source code for this section can be found in the file
`Examples/Filtering/GradientMagnitudeImageFilter.cxx`.

The magnitude of the image gradient is extensively used in image analysis, mainly to help in the determination of object contours and the separation of homogeneous regions. The `itk::GradientMagnitudeImageFilter` computes the magnitude of the image gradient at each pixel location using a simple finite differences approach. For example, in the case of $2D$ the computation is equivalent to convolving the image with masks of type

$$\begin{array}{|c|c|c|}\hline -1 & 0 & 1 \\ \hline \end{array} \quad \begin{array}{|c|}\hline -1 \\ \hline 0 \\ \hline 1 \\ \hline \end{array}$$

then adding the sum of their squares and computing the square root of the sum.

This filter will work on images of any dimension thanks to the internal use of `itk::NeighborhoodIterator` and `itk::NeighborhoodOperator`.

The first step required to use this filter is to include its header file.

```
#include "itkGradientMagnitudeImageFilter.h"
```

Types should be chosen for the pixels of the input and output images.

```
  typedef    float    InputPixelType;
  typedef    float    OutputPixelType;
```

The input and output image types can be defined using the pixel types.

```
  typedef itk::Image< InputPixelType,  2 >   InputImageType;
  typedef itk::Image< OutputPixelType, 2 >   OutputImageType;
```

The type of the gradient magnitude filter is defined by the input image and the output image types.

```
  typedef itk::GradientMagnitudeImageFilter<
                 InputImageType, OutputImageType >  FilterType;
```

A filter object is created by invoking the `New()` method and assigning the result to a `itk::SmartPointer`.

```
  FilterType::Pointer filter = FilterType::New();
```

The input image can be obtained from the output of another filter. Here, the source is an image reader.

```
  filter->SetInput( reader->GetOutput() );
```

Finally, the filter is executed by invoking the `Update()` method.

```
  filter->Update();
```

If the output of this filter has been connected to other filters in a pipeline, updating any of the downstream filters will also trigger an update of this filter. For example, the gradient magnitude filter may be connected to an image writer.

```
  rescaler->SetInput( filter->GetOutput() );
  writer->SetInput( rescaler->GetOutput() );
  writer->Update();
```

Figure 6.8 illustrates the effect of the gradient magnitude filter on a MRI proton density image of the brain. The figure shows the sensitivity of this filter to noisy data.

Attention should be paid to the image type chosen to represent the output image since the dynamic range of the gradient magnitude image is usually smaller than the dynamic range of the input image. As always, there are exceptions to this rule, for example, synthetic images that contain high contrast objects.

This filter does not apply any smoothing to the image before computing the gradients. The results can therefore be very sensitive to noise and may not be best choice for scale space analysis.

6.4.2 Gradient Magnitude With Smoothing

The source code for this section can be found in the file
`Examples/Filtering/GradientMagnitudeRecursiveGaussianImageFilter.cxx`.

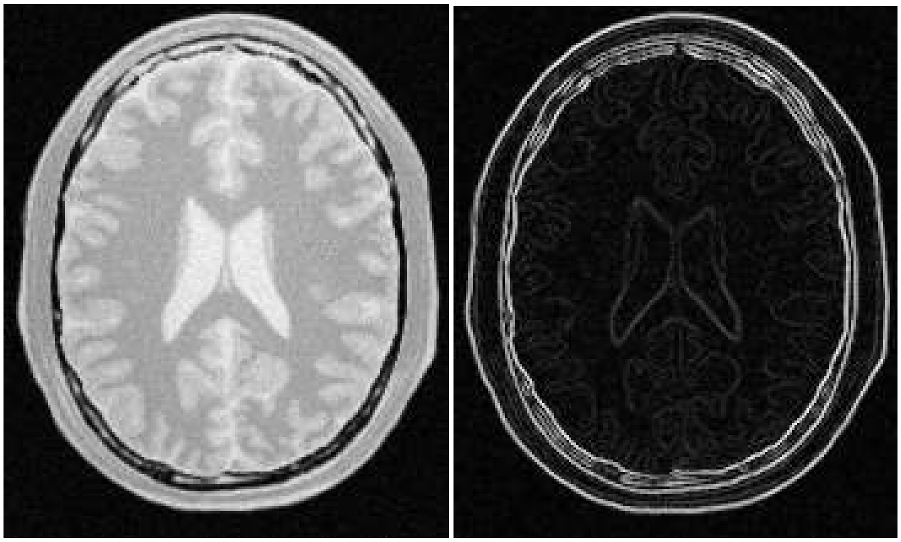


Figure 6.8: Effect of the GradientMagnitudeImageFilter on a slice from a MRI proton density image of the brain.

Differentiation is an ill-defined operation over digital data. In practice it is convenient to define a scale in which the differentiation should be performed. This is usually done by preprocessing the data with a smoothing filter. It has been shown that a Gaussian kernel is the most convenient choice for performing such smoothing. By choosing a particular value for the standard deviation (σ) of the Gaussian, an associated scale is selected that ignores high frequency content, commonly considered image noise.

The `itk::GradientMagnitudeRecursiveGaussianImageFilter` computes the magnitude of the image gradient at each pixel location. The computational process is equivalent to first smoothing the image by convolving it with a Gaussian kernel and then applying a differential operator. The user selects the value of σ.

Internally this is done by applying an IIR [1] filter that approximates a convolution with the derivative of the Gaussian kernel. Traditional convolution will produce a more accurate result, but the IIR approach is much faster, especially using large σs [21, 22].

GradientMagnitudeRecursiveGaussianImageFilter will work on images of any dimension by taking advantage of the natural separability of the Gaussian kernel and its derivatives.

The first step required to use this filter is to include its header file.

```
#include "itkGradientMagnitudeRecursiveGaussianImageFilter.h"
```

Types should be instantiated based on the pixels of the input and output images.

[1]Infinite Impulse Response

```
typedef    float    InputPixelType;
typedef    float    OutputPixelType;
```

With them, the input and output image types can be instantiated.

```
typedef itk::Image< InputPixelType,  2 >   InputImageType;
typedef itk::Image< OutputPixelType, 2 >   OutputImageType;
```

The filter type is now instantiated using both the input image and the output image types.

```
typedef itk::GradientMagnitudeRecursiveGaussianImageFilter<
                      InputImageType, OutputImageType >  FilterType;
```

A filter object is created by invoking the `New()` method and assigning the result to a `itk::SmartPointer`.

```
FilterType::Pointer filter = FilterType::New();
```

The input image can be obtained from the output of another filter. Here, an image reader is used as source.

```
filter->SetInput( reader->GetOutput() );
```

The standard deviation of the Gaussian smoothing kernel is now set.

```
filter->SetSigma( sigma );
```

Finally the filter is executed by invoking the `Update()` method.

```
filter->Update();
```

If connected to other filters in a pipeline, this filter will automatically update when any downstream filters are updated. For example, we may connect this gradient magnitude filter to an image file writer and then update the writer.

```
rescaler->SetInput( filter->GetOutput() );
writer->SetInput( rescaler->GetOutput() );
writer->Update();
```

Figure 6.9 illustrates the effect of this filter on a MRI proton density image of the brain using σ values of 3 (left) and 5 (right). The figure shows how the sensitivity to noise can be regulated by selecting an appropriate σ. This type of scale-tunable filter is suitable for performing scale-space analysis.

/ Attention should be paid to the image type chosen to represent the output image since the dynamic range of the gradient magnitude image is usually smaller than the dynamic range of the input image.

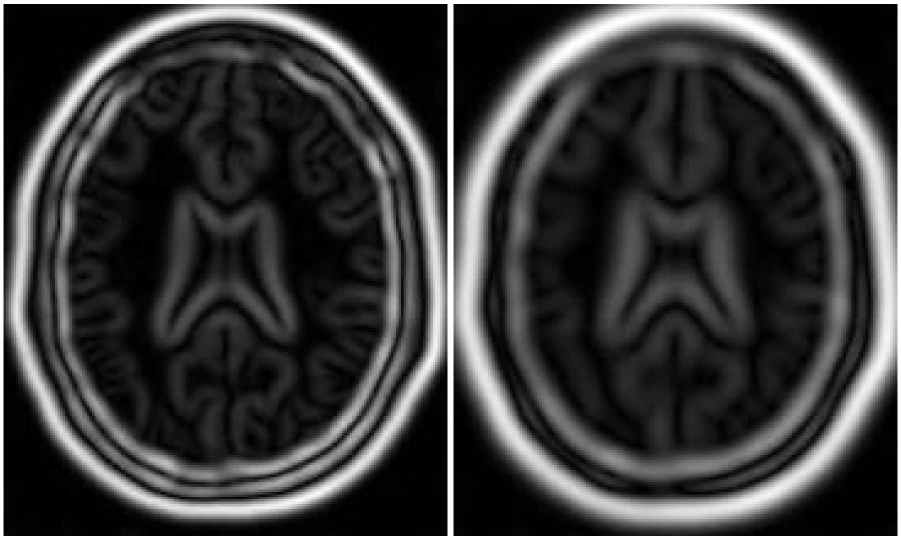


Figure 6.9: Effect of the GradientMagnitudeRecursiveGaussianImageFilter on a slice from a MRI proton density image of the brain.

6.4.3 Derivative Without Smoothing

The source code for this section can be found in the file
`Examples/Filtering/DerivativeImageFilter.cxx`.

The `itk::DerivativeImageFilter` is used for computing the partial derivative of an image, the derivative of an image along a particular axial direction.

The header file corresponding to this filter should be included first.

```
#include "itkDerivativeImageFilter.h"
```

Next, the pixel types for the input and output images must be defined and, with them, the image types can be instantiated. Note that it is important to select a signed type for the image, since the values of the derivatives will be positive as well as negative.

```
  typedef   float  InputPixelType;
  typedef   float  OutputPixelType;

  const unsigned int Dimension = 2;

  typedef itk::Image< InputPixelType,  Dimension >   InputImageType;
  typedef itk::Image< OutputPixelType, Dimension >   OutputImageType;
```

Using the image types, it is now possible to define the filter type and create the filter object.

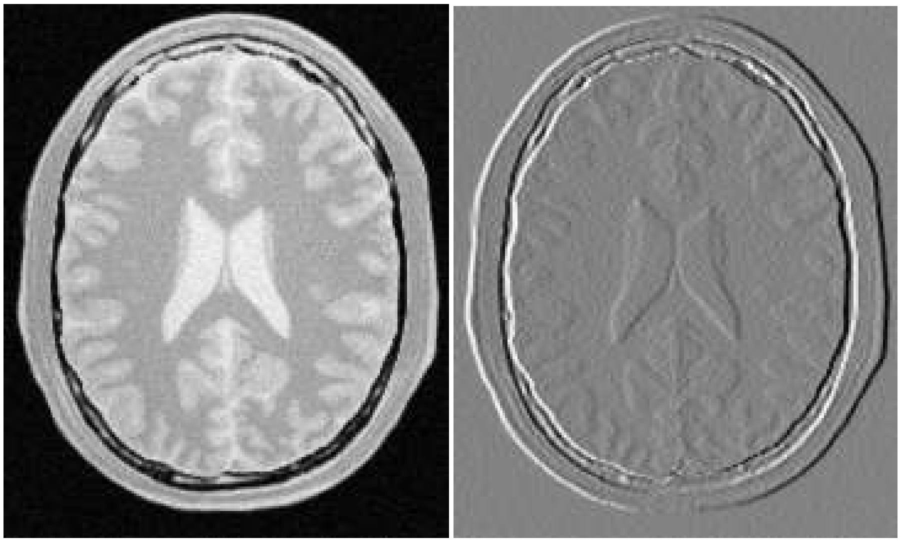


Figure 6.10: Effect of the Derivative filter on a slice from a MRI proton density brain image.

```
typedef itk::DerivativeImageFilter<
             InputImageType, OutputImageType >  FilterType;

FilterType::Pointer filter = FilterType::New();
```

The order of the derivative is selected with the `SetOrder()` method. The direction along which the derivative will be computed is selected with the `SetDirection()` method.

```
filter->SetOrder(     atoi( argv[4] ) );
filter->SetDirection( atoi( argv[5] ) );
```

The input to the filter can be taken from any other filter, for example a reader. The output can be passed down the pipeline to other filters, for example, a writer. An update call on any downstream filter will trigger the execution of the derivative filter.

```
filter->SetInput( reader->GetOutput() );
writer->SetInput( filter->GetOutput() );
writer->Update();
```

Figure 6.10 illustrates the effect of the DerivativeImageFilter on a slice of MRI brain image. The derivative is taken along the *x* direction. The sensitivity to noise in the image is evident from this result.

6.5 Second Order Derivatives

6.5.1 Second Order Recursive Gaussian

The source code for this section can be found in the file
`Examples/Filtering/SecondDerivativeRecursiveGaussianImageFilter.cxx`.

This example illustrates how to compute second derivatives of a 3D image using the `itk::RecursiveGaussianImageFilter`.

In this example, all the second derivatives are computed independently in the same way as if they were intended to be used for building the Hessian matrix of the image.

```
#include "itkRecursiveGaussianImageFilter.h"
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
#include "itkImageDuplicator.h"
#include "itkImage.h"
#include <string>


int main(int argc, char * argv [] )
{

  if( argc < 3 )
    {
    std::cerr << "Usage: " << std::endl;
    std::cerr << "SecondDerivativeRecursiveGaussianImageFilter inputImage outputPrefix  [sigma] " << st
    return EXIT_FAILURE;
    }

  typedef float            PixelType;
  typedef float            OutputPixelType;

  const unsigned int  Dimension = 3;

  typedef itk::Image< PixelType,       Dimension >  ImageType;
  typedef itk::Image< OutputPixelType, Dimension >  OutputImageType;

  typedef itk::ImageFileReader< ImageType       >   ReaderType;
  typedef itk::ImageFileWriter< OutputImageType >   WriterType;

  typedef itk::ImageDuplicator< OutputImageType >   DuplicatorType;

  typedef itk::RecursiveGaussianImageFilter<
                                    ImageType,
                                    ImageType >  FilterType;
```

```
ReaderType::Pointer  reader  = ReaderType::New();
WriterType::Pointer  writer  = WriterType::New();

DuplicatorType::Pointer duplicator  = DuplicatorType::New();

reader->SetFileName( argv[1] );

std::string outputPrefix = argv[2];
std::string outputFileName;

try
  {
  reader->Update();
  }
catch( itk::ExceptionObject & excp )
  {
  std::cerr << "Problem reading the input file" << std::endl;
  std::cerr << excp << std::endl;
  return EXIT_FAILURE;
  }

FilterType::Pointer ga = FilterType::New();
FilterType::Pointer gb = FilterType::New();
FilterType::Pointer gc = FilterType::New();

ga->SetDirection( 0 );
gb->SetDirection( 1 );
gc->SetDirection( 2 );

if( argc > 3 )
  {
  const float sigma = atof( argv[3] );
  ga->SetSigma( sigma );
  gb->SetSigma( sigma );
  gc->SetSigma( sigma );
  }

ga->SetZeroOrder();
gb->SetZeroOrder();
gc->SetSecondOrder();

ImageType::Pointer inputImage = reader->GetOutput();

ga->SetInput( inputImage );
gb->SetInput( ga->GetOutput() );
gc->SetInput( gb->GetOutput() );

duplicator->SetInputImage( gc->GetOutput() );
```

```
gc->Update();
duplicator->Update();

ImageType::Pointer Izz = duplicator->GetOutput();

writer->SetInput( Izz );
outputFileName = outputPrefix + "-Izz.mhd";
writer->SetFileName( outputFileName.c_str() );
writer->Update();

gc->SetDirection( 1 );  // gc now works along Y
gb->SetDirection( 2 );  // gb now works along Z

gc->Update();
duplicator->Update();

ImageType::Pointer Iyy = duplicator->GetOutput();

writer->SetInput( Iyy );
outputFileName = outputPrefix + "-Iyy.mhd";
writer->SetFileName( outputFileName.c_str() );
writer->Update();


gc->SetDirection( 0 );  // gc now works along X
ga->SetDirection( 1 );  // ga now works along Y

gc->Update();
duplicator->Update();

ImageType::Pointer Ixx = duplicator->GetOutput();

writer->SetInput( Ixx );
outputFileName = outputPrefix + "-Ixx.mhd";
writer->SetFileName( outputFileName.c_str() );
writer->Update();


ga->SetDirection( 0 );
gb->SetDirection( 1 );
gc->SetDirection( 2 );

ga->SetZeroOrder();
gb->SetFirstOrder();
gc->SetFirstOrder();
```

```
gc->Update();
duplicator->Update();

ImageType::Pointer Iyz = duplicator->GetOutput();

writer->SetInput( Iyz );
outputFileName = outputPrefix + "-Iyz.mhd";
writer->SetFileName( outputFileName.c_str() );
writer->Update();


ga->SetDirection( 1 );
gb->SetDirection( 0 );
gc->SetDirection( 2 );

ga->SetZeroOrder();
gb->SetFirstOrder();
gc->SetFirstOrder();

gc->Update();
duplicator->Update();

ImageType::Pointer Ixz = duplicator->GetOutput();

writer->SetInput( Ixz );
outputFileName = outputPrefix + "-Ixz.mhd";
writer->SetFileName( outputFileName.c_str() );
writer->Update();

ga->SetDirection( 2 );
gb->SetDirection( 0 );
gc->SetDirection( 1 );

ga->SetZeroOrder();
gb->SetFirstOrder();
gc->SetFirstOrder();

gc->Update();
duplicator->Update();

ImageType::Pointer Ixy = duplicator->GetOutput();

writer->SetInput( Ixy );
outputFileName = outputPrefix + "-Ixy.mhd";
writer->SetFileName( outputFileName.c_str() );
writer->Update();
```

6.5.2 Laplacian Filters

Laplacian Filter Finite Difference

Laplacian Filter Recursive Gaussian

The source code for this section can be found in the file
`Examples/Filtering/LaplacianRecursiveGaussianImageFilter1.cxx`.

This example illustrates how to use the `itk::RecursiveGaussianImageFilter` for computing the Laplacian of a 2D image.

The first step required to use this filter is to include its header file.

```
#include "itkRecursiveGaussianImageFilter.h"
```

Types should be selected on the desired input and output pixel types.

```
  typedef    float    InputPixelType;
  typedef    float    OutputPixelType;
```

The input and output image types are instantiated using the pixel types.

```
  typedef itk::Image< InputPixelType,  2 >   InputImageType;
  typedef itk::Image< OutputPixelType, 2 >   OutputImageType;
```

The filter type is now instantiated using both the input image and the output image types.

```
  typedef itk::RecursiveGaussianImageFilter<
                        InputImageType, OutputImageType >  FilterType;
```

This filter applies the approximation of the convolution along a single dimension. It is therefore necessary to concatenate several of these filters to produce smoothing in all directions. In this example, we create a pair of filters since we are processing a $2D$ image. The filters are created by invoking the `New()` method and assigning the result to a `itk::SmartPointer`.

We need two filters for computing the X component of the Laplacian and two other filters for computing the Y component.

```
  FilterType::Pointer filterX1 = FilterType::New();
  FilterType::Pointer filterY1 = FilterType::New();

  FilterType::Pointer filterX2 = FilterType::New();
  FilterType::Pointer filterY2 = FilterType::New();
```

Since each one of the newly created filters has the potential to perform filtering along any dimension, we have to restrict each one to a particular direction. This is done with the `SetDirection()` method.

```
filterX1->SetDirection( 0 );   // 0 --> X direction
filterY1->SetDirection( 1 );   // 1 --> Y direction

filterX2->SetDirection( 0 );   // 0 --> X direction
filterY2->SetDirection( 1 );   // 1 --> Y direction
```

The `itk::RecursiveGaussianImageFilter` can approximate the convolution with the Gaussian or with its first and second derivatives. We select one of these options by using the `SetOrder()` method. Note that the argument is an `enum` whose values can be `ZeroOrder`, `FirstOrder` and `SecondOrder`. For example, to compute the x partial derivative we should select `FirstOrder` for x and `ZeroOrder` for y. Here we want only to smooth in x and y, so we select `ZeroOrder` in both directions.

```
filterX1->SetOrder( FilterType::ZeroOrder );
filterY1->SetOrder( FilterType::SecondOrder );

filterX2->SetOrder( FilterType::SecondOrder );
filterY2->SetOrder( FilterType::ZeroOrder );
```

There are two typical ways of normalizing Gaussians depending on their application. For scale-space analysis it is desirable to use a normalization that will preserve the maximum value of the input. This normalization is represented by the following equation.

$$\frac{1}{\sigma\sqrt{2\pi}} \tag{6.2}$$

In applications that use the Gaussian as a solution of the diffusion equation it is desirable to use a normalization that preserve the integral of the signal. This last approach can be seen as a conservation of mass principle. This is represented by the following equation.

$$\frac{1}{\sigma^2\sqrt{2\pi}} \tag{6.3}$$

The `itk::RecursiveGaussianImageFilter` has a boolean flag that allows users to select between these two normalization options. Selection is done with the method `SetNormalizeAcrossScale()`. Enable this flag to analyzing an image across scale-space. In the current example, this setting has no impact because we are actually renormalizing the output to the dynamic range of the reader, so we simply disable the flag.

```
const bool normalizeAcrossScale = false;
```

```
  filterX1->SetNormalizeAcrossScale( normalizeAcrossScale );
  filterY1->SetNormalizeAcrossScale( normalizeAcrossScale );
  filterX2->SetNormalizeAcrossScale( normalizeAcrossScale );
  filterY2->SetNormalizeAcrossScale( normalizeAcrossScale );
```

The input image can be obtained from the output of another filter. Here, an image reader is used as the source. The image is passed to the *x* filter and then to the *y* filter. The reason for keeping these two filters separate is that it is usual in scale-space applications to compute not only the smoothing but also combinations of derivatives at different orders and smoothing. Some factorization is possible when separate filters are used to generate the intermediate results. Here this capability is less interesting, though, since we only want to smooth the image in all directions.

```
  filterX1->SetInput( reader->GetOutput() );
  filterY1->SetInput( filterX1->GetOutput() );

  filterY2->SetInput( reader->GetOutput() );
  filterX2->SetInput( filterY2->GetOutput() );
```

It is now time to select the σ of the Gaussian used to smooth the data. Note that σ must be passed to both filters and that sigma is considered to be in millimeters. That is, at the moment of applying the smoothing process, the filter will take into account the spacing values defined in the image.

```
  filterX1->SetSigma( sigma );
  filterY1->SetSigma( sigma );
  filterX2->SetSigma( sigma );
  filterY2->SetSigma( sigma );
```

Finally the two components of the Laplacian should be added together. The `itk::AddImageFilter` is used for this purpose.

```
  typedef itk::AddImageFilter<
                OutputImageType,
                OutputImageType,
                OutputImageType > AddFilterType;

  AddFilterType::Pointer addFilter = AddFilterType::New();

  addFilter->SetInput1( filterY1->GetOutput() );
  addFilter->SetInput2( filterX2->GetOutput() );
```

The filters are triggered by invoking `Update()` on the Add filter at the end of the pipeline.

```
try
  {
  addFilter->Update();
  }
catch( itk::ExceptionObject & err )
  {
  std::cout << "ExceptionObject caught !" << std::endl;
  std::cout << err << std::endl;
  return EXIT_FAILURE;
  }
```

The resulting image could be saved to a file using the `itk::ImageFileWriter` class.

```
typedef  float WritePixelType;

typedef itk::Image< WritePixelType, 2 >    WriteImageType;

typedef itk::ImageFileWriter< WriteImageType >  WriterType;

WriterType::Pointer writer = WriterType::New();

writer->SetInput( addFilter->GetOutput() );

writer->SetFileName( argv[2] );

writer->Update();
```

Figure 6.11 illustrates the effect of this filter on a MRI proton density image of the brain using σ values of 3 (left) and 5 (right). The figure shows how the attenuation of noise can be regulated by selecting the appropriate standard deviation. This type of scale-tunable filter is suitable for performing scale-space analysis.

The source code for this section can be found in the file
`Examples/Filtering/LaplacianRecursiveGaussianImageFilter2.cxx`.

The previous exampled showed how to use the `itk::RecursiveGaussianImageFilter` for computing the equivalent of a Laplacian of an image after smoothing with a Gaussian. The elements used in this previous example have been packaged together in the `itk::LaplacianRecursiveGaussianImageFilter` in order to simplify its usage. This current example shows how to use this convenience filter for achieving the same results as the previous example.

The first step required to use this filter is to include its header file.

```
#include "itkLaplacianRecursiveGaussianImageFilter.h"
```

Types should be selected on the desired input and output pixel types.

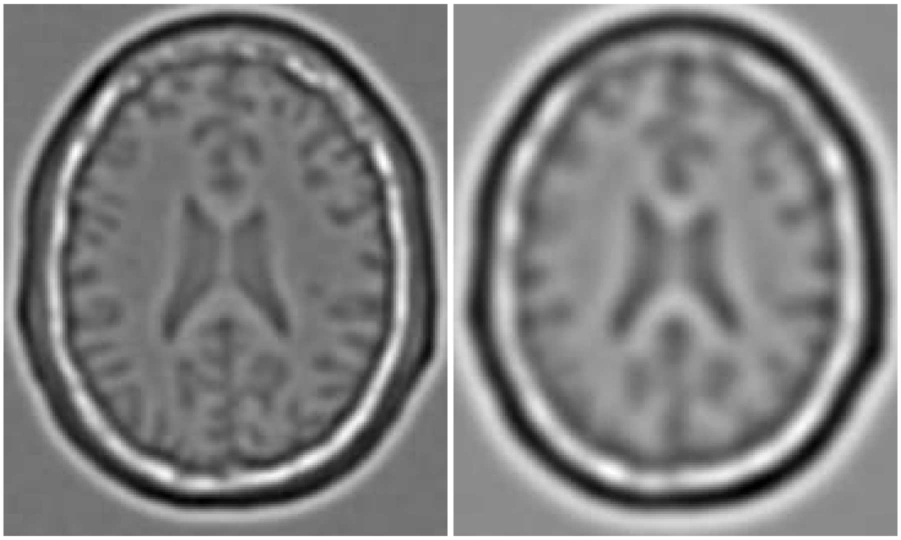


Figure 6.11: Effect of the LaplacianRecursiveGaussianImageFilter on a slice from a MRI proton density image of the brain.

```
  typedef    float    InputPixelType;
  typedef    float    OutputPixelType;
```

The input and output image types are instantiated using the pixel types.

```
  typedef itk::Image< InputPixelType,  2 >   InputImageType;
  typedef itk::Image< OutputPixelType, 2 >   OutputImageType;
```

The filter type is now instantiated using both the input image and the output image types.

```
  typedef itk::LaplacianRecursiveGaussianImageFilter<
                            InputImageType, OutputImageType >  FilterType;
```

This filter packages all the components illustrated in the previous example. The filter is created by invoking the `New()` method and assigning the result to a `itk::SmartPointer`.

```
  FilterType::Pointer laplacian = FilterType::New();
```

The option for normalizing across scale space can also be selected in this filter.

```
  laplacian->SetNormalizeAcrossScale( false );
```

The input image can be obtained from the output of another filter. Here, an image reader is used as the source.

```
laplacian->SetInput( reader->GetOutput() );
```

It is now time to select the σ of the Gaussian used to smooth the data. Note that σ must be passed to both filters and that sigma is considered to be in millimeters. That is, at the moment of applying the smoothing process, the filter will take into account the spacing values defined in the image.

```
laplacian->SetSigma( sigma );
```

Finally the pipeline is executed by invoking the `Update()` method.

```
try
  {
  laplacian->Update();
  }
catch( itk::ExceptionObject & err )
  {
  std::cout << "ExceptionObject caught !" << std::endl;
  std::cout << err << std::endl;
  return EXIT_FAILURE;
  }
```

Figure 6.12 illustrates the effect of this filter on a MRI proton density image of the brain using σ values of 3 (left) and 5 (right). The figure shows how the attenuation of noise can be regulated by selecting the appropriate standard deviation. This type of scale-tunable filter is suitable for performing scale-space analysis.

6.6 Neighborhood Filters

The concept of locality is frequently encountered in image processing in the form of filters that compute every output pixel using information from a small region in the neighborhood of the input pixel. The classical form of these filters are the 3×3 filters in 2D images. Convolution masks based on these neighborhoods can perform diverse tasks ranging from noise reduction, to differential operations, to mathematical morphology.

The Insight toolkit implements an elegant approach to neighborhood-based image filtering. The input image is processed using a special iterator called the `itk::NeighborhoodIterator`. This iterator is capable of moving over all the pixels in an image and, for each position, it can address the pixels in a local neighborhood. Operators are defined that apply an algorithmic operation in the neighborhood of the input pixel to produce a value for the output pixel. The following section describes some of the more commonly used filters that take advantage of this construction. (See Chapter 11 on page 701 for more information about iterators.)

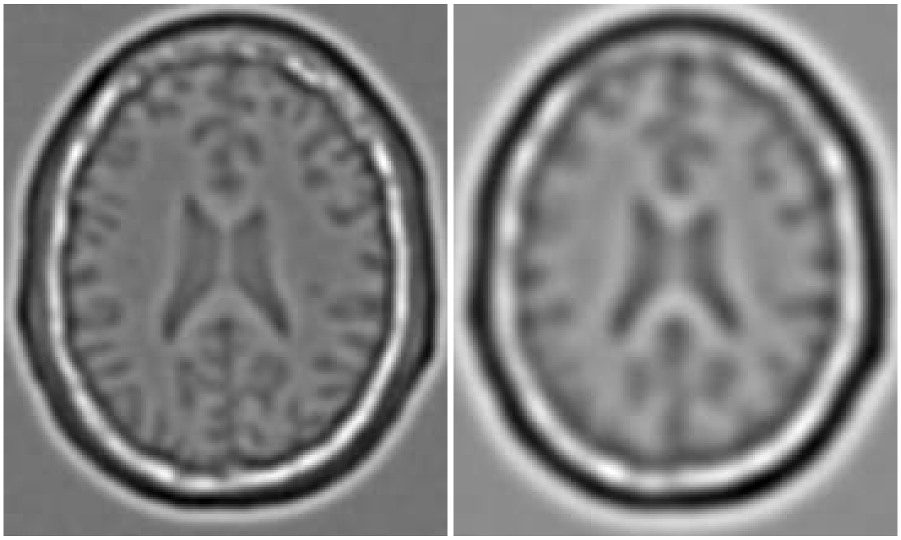
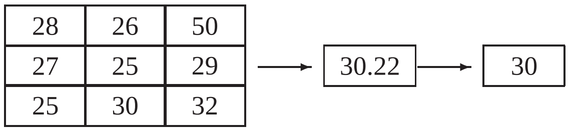


Figure 6.12: Effect of the LaplacianRecursiveGaussianImageFilter on a slice from a MRI proton density image of the brain.

6.6.1 Mean Filter

The source code for this section can be found in the file
`Examples/Filtering/MeanImageFilter.cxx`.

The `itk::MeanImageFilter` is commonly used for noise reduction. The filter computes the value of each output pixel by finding the statistical mean of the neighborhood of the corresponding input pixel. The following figure illustrates the local effect of the MeanImageFilter in a 2*D* case. The statistical mean of the neighborhood on the left is passed as the output value associated with the pixel at the center of the neighborhood.



Note that this algorithm is sensitive to the presence of outliers in the neighborhood. This filter will work on images of any dimension thanks to the internal use of `itk::SmartNeighborhoodIterator` and `itk::NeighborhoodOperator`. The size of the neighborhood over which the mean is computed can be set by the user.

The header file corresponding to this filter should be included first.

```
#include "itkMeanImageFilter.h"
```

Then the pixel types for input and output image must be defined and, with them, the image types can be instantiated.

```
typedef   unsigned char  InputPixelType;
typedef   unsigned char  OutputPixelType;

typedef itk::Image< InputPixelType,  2 >   InputImageType;
typedef itk::Image< OutputPixelType, 2 >   OutputImageType;
```

Using the image types it is now possible to instantiate the filter type and create the filter object.

```
typedef itk::MeanImageFilter<
             InputImageType, OutputImageType >  FilterType;

FilterType::Pointer filter = FilterType::New();
```

The size of the neighborhood is defined along every dimension by passing a `SizeType` object with the corresponding values. The value on each dimension is used as the semi-size of a rectangular box. For example, in $2D$ a size of $1,2$ will result in a 3×5 neighborhood.

```
InputImageType::SizeType indexRadius;

indexRadius[0] = 1; // radius along x
indexRadius[1] = 1; // radius along y

filter->SetRadius( indexRadius );
```

The input to the filter can be taken from any other filter, for example a reader. The output can be passed down the pipeline to other filters, for example, a writer. An update call on any downstream filter will trigger the execution of the mean filter.

```
filter->SetInput( reader->GetOutput() );
writer->SetInput( filter->GetOutput() );
writer->Update();
```

Figure 6.13 illustrates the effect of this filter on a slice of MRI brain image using neighborhood radii of $1,1$ which corresponds to a 3×3 classical neighborhood. It can be seen from this picture that edges are rapidly degraded by the diffusion of intensity values among neighbors.

6.6.2 Median Filter

The source code for this section can be found in the file
`Examples/Filtering/MedianImageFilter.cxx`.

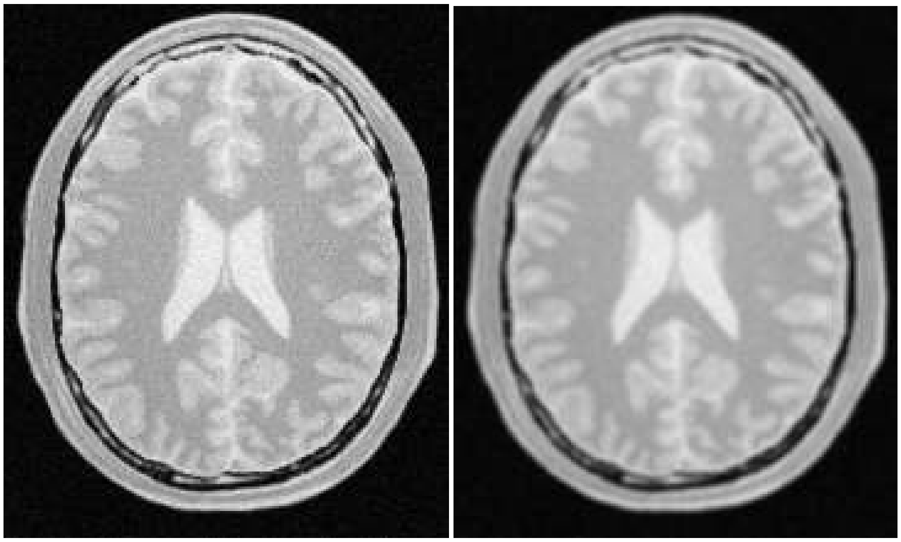


Figure 6.13: Effect of the MeanImageFilter on a slice from a MRI proton density brain image.

The itk::MedianImageFilter is commonly used as a robust approach for noise reduction. This filter is particularly efficient against *salt-and-pepper* noise. In other words, it is robust to the presence of gray-level outliers. MedianImageFilter computes the value of each output pixel as the statistical median of the neighborhood of values around the corresponding input pixel. The following figure illustrates the local effect of this filter in a 2*D* case. The statistical median of the neighborhood on the left is passed as the output value associated with the pixel at the center of the neighborhood.

$$\begin{array}{|c|c|c|} \hline 28 & 26 & 50 \\ \hline 27 & 25 & 29 \\ \hline 25 & 30 & 32 \\ \hline \end{array} \longrightarrow \boxed{28}$$

This filter will work on images of any dimension thanks to the internal use of itk::NeighborhoodIterator and itk::NeighborhoodOperator. The size of the neighborhood over which the median is computed can be set by the user.

The header file corresponding to this filter should be included first.

```
#include "itkMedianImageFilter.h"
```

Then the pixel and image types of the input and output must be defined.

```
  typedef   unsigned char  InputPixelType;
```

```
typedef   unsigned char  OutputPixelType;

typedef itk::Image< InputPixelType,  2 >   InputImageType;
typedef itk::Image< OutputPixelType, 2 >   OutputImageType;
```

Using the image types, it is now possible to define the filter type and create the filter object.

```
typedef itk::MedianImageFilter<
             InputImageType, OutputImageType >  FilterType;

FilterType::Pointer filter = FilterType::New();
```

The size of the neighborhood is defined along every dimension by passing a `SizeType` object with the corresponding values. The value on each dimension is used as the semi-size of a rectangular box. For example, in $2D$ a size of $1,2$ will result in a 3×5 neighborhood.

```
InputImageType::SizeType indexRadius;

indexRadius[0] = 1; // radius along x
indexRadius[1] = 1; // radius along y

filter->SetRadius( indexRadius );
```

The input to the filter can be taken from any other filter, for example a reader. The output can be passed down the pipeline to other filters, for example, a writer. An update call on any downstream filter will trigger the execution of the median filter.

```
filter->SetInput( reader->GetOutput() );
writer->SetInput( filter->GetOutput() );
writer->Update();
```

Figure 6.14 illustrates the effect of the MedianImageFilter filter on a slice of MRI brain image using a neighborhood radius of $1,1$, which corresponds to a 3×3 classical neighborhood. The filtered image demonstrates the moderate tendency of the median filter to preserve edges.

6.6.3 Mathematical Morphology

Mathematical morphology has proved to be a powerful resource for image processing and analysis [73]. ITK implements mathematical morphology filters using NeighborhoodIterators and `itk::NeighborhoodOperator`s. The toolkit contains two types of image morphology algorithms, filters that operate on binary images and filters that operate on grayscale images.

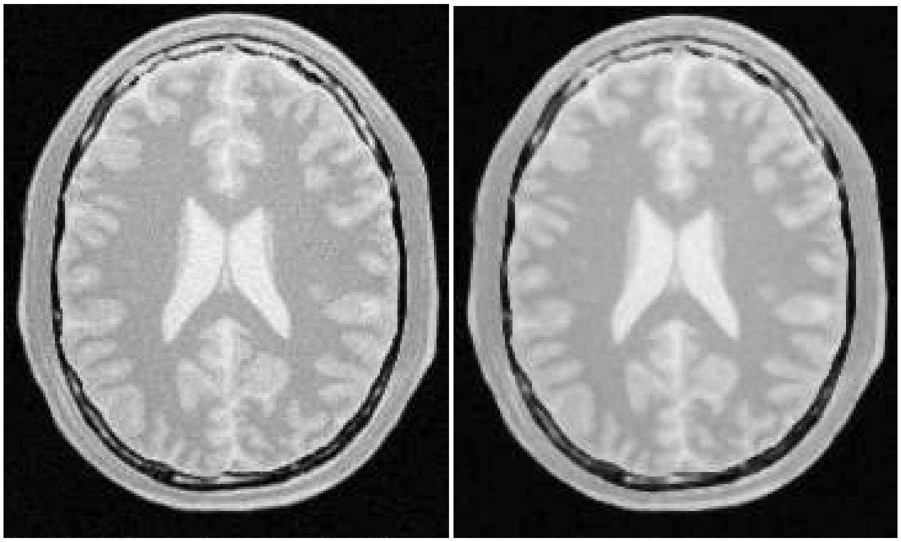


Figure 6.14: Effect of the MedianImageFilter on a slice from a MRI proton density brain image.

Binary Filters

The source code for this section can be found in the file
`Examples/Filtering/MathematicalMorphologyBinaryFilters.cxx`.

The following section illustrates the use of filters that perform basic mathematical morphology operations on binary images. The `itk::BinaryErodeImageFilter` and `itk::BinaryDilateImageFilter` are described here. The filter names clearly specify the type of image on which they operate. The header files required to construct a simple example of the use of the mathematical morphology filters are included below.

```
#include "itkBinaryErodeImageFilter.h"
#include "itkBinaryDilateImageFilter.h"
#include "itkBinaryBallStructuringElement.h"
```

The following code defines the input and output pixel types and their associated image types.

```
  const unsigned int Dimension = 2;

  typedef unsigned char   InputPixelType;
  typedef unsigned char   OutputPixelType;

  typedef itk::Image< InputPixelType,  Dimension >   InputImageType;
  typedef itk::Image< OutputPixelType, Dimension >   OutputImageType;
```

Mathematical morphology operations are implemented by applying an operator over the neighborhood of each input pixel. The combination of the rule and the neighborhood is known as *structuring element*. Although some rules have become de facto standards for image processing, there is a good deal of freedom as to what kind of algorithmic rule should be applied to the neighborhood. The implementation in ITK follows the typical rule of minimum for erosion and maximum for dilation.

The structuring element is implemented as a NeighborhoodOperator. In particular, the default structuring element is the `itk::BinaryBallStructuringElement` class. This class is instantiated using the pixel type and dimension of the input image.

```
typedef itk::BinaryBallStructuringElement<
                    InputPixelType,
                    Dimension  >            StructuringElementType;
```

The structuring element type is then used along with the input and output image types for instantiating the type of the filters.

```
typedef itk::BinaryErodeImageFilter<
                          InputImageType,
                          OutputImageType,
                          StructuringElementType >  ErodeFilterType;

typedef itk::BinaryDilateImageFilter<
                          InputImageType,
                          OutputImageType,
                          StructuringElementType >  DilateFilterType;
```

The filters can now be created by invoking the `New()` method and assigning the result to `itk::SmartPointer`s.

```
ErodeFilterType::Pointer  binaryErode  = ErodeFilterType::New();
DilateFilterType::Pointer binaryDilate = DilateFilterType::New();
```

The structuring element is not a reference counted class. Thus it is created as a C++ stack object instead of using `New()` and SmartPointers. The radius of the neighborhood associated with the structuring element is defined with the `SetRadius()` method and the `CreateStructuringElement()` method is invoked in order to initialize the operator. The resulting structuring element is passed to the mathematical morphology filter through the `SetKernel()` method, as illustrated below.

```
StructuringElementType  structuringElement;

structuringElement.SetRadius( 1 );  // 3x3 structuring element
```

```
  structuringElement.CreateStructuringElement();

  binaryErode->SetKernel(  structuringElement );
  binaryDilate->SetKernel( structuringElement );
```

A binary image is provided as input to the filters. This image might be, for example, the output of a binary threshold image filter.

```
  thresholder->SetInput( reader->GetOutput() );

  InputPixelType background =   0;
  InputPixelType foreground = 255;

  thresholder->SetOutsideValue( background );
  thresholder->SetInsideValue(  foreground );

  thresholder->SetLowerThreshold( lowerThreshold );
  thresholder->SetUpperThreshold( upperThreshold );


  binaryErode->SetInput( thresholder->GetOutput() );
  binaryDilate->SetInput( thresholder->GetOutput() );
```

The values that correspond to “objects” in the binary image are specified with the methods `SetErodeValue()` and `SetDilateValue()`. The value passed to these methods will be considered the value over which the dilation and erosion rules will apply.

```
  binaryErode->SetErodeValue( foreground );
  binaryDilate->SetDilateValue( foreground );
```

The filter is executed by invoking its `Update()` method, or by updating any downstream filter, like, for example, an image writer.

```
  writerDilation->SetInput( binaryDilate->GetOutput() );
  writerDilation->Update();
```

Figure 6.15 illustrates the effect of the erosion and dilation filters on a binary image from a MRI brain slice. The figure shows how these operations can be used to remove spurious details from segmented images.

Grayscale Filters

The source code for this section can be found in the file
`Examples/Filtering/MathematicalMorphologyGrayscaleFilters.cxx`.

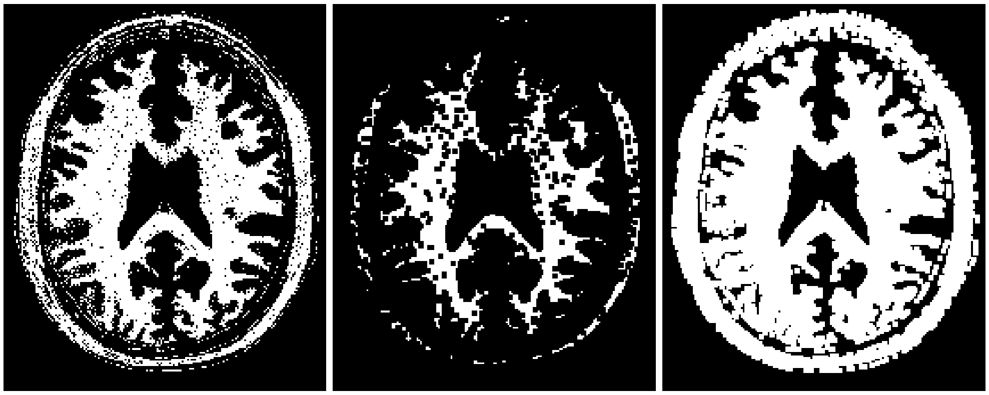


Figure 6.15: Effect of erosion and dilation in a binary image.

The following section illustrates the use of filters for performing basic mathematical morphology operations on grayscale images. The `itk::GrayscaleErodeImageFilter` and `itk::GrayscaleDilateImageFilter` are covered in this example. The filter names clearly specify the type of image on which they operate. The header files required for a simple example of the use of grayscale mathematical morphology filters are presented below.

```
#include "itkGrayscaleErodeImageFilter.h"
#include "itkGrayscaleDilateImageFilter.h"
#include "itkBinaryBallStructuringElement.h"
```

The following code defines the input and output pixel types and their associated image types.

```
  const unsigned int Dimension = 2;

  typedef unsigned char   InputPixelType;
  typedef unsigned char   OutputPixelType;

  typedef itk::Image< InputPixelType,  Dimension >   InputImageType;
  typedef itk::Image< OutputPixelType, Dimension >   OutputImageType;
```

Mathematical morphology operations are based on the application of an operator over a neighborhood of each input pixel. The combination of the rule and the neighborhood is known as *structuring element*. Although some rules have become the de facto standard in image processing there is a good deal of freedom as to what kind of algorithmic rule should be applied on the neighborhood. The implementation in ITK follows the typical rule of minimum for erosion and maximum for dilation.

The structuring element is implemented as a `itk::NeighborhoodOperator`. In particular, the default structuring element is the `itk::BinaryBallStructuringElement` class. This class is instantiated using the pixel type and dimension of the input image.

```
typedef itk::BinaryBallStructuringElement<
                    InputPixelType,
                    Dimension  >             StructuringElementType;
```

The structuring element type is then used along with the input and output image types for instantiating the type of the filters.

```
typedef itk::GrayscaleErodeImageFilter<
                          InputImageType,
                          OutputImageType,
                          StructuringElementType >  ErodeFilterType;

typedef itk::GrayscaleDilateImageFilter<
                          InputImageType,
                          OutputImageType,
                          StructuringElementType >  DilateFilterType;
```

The filters can now be created by invoking the `New()` method and assigning the result to SmartPointers.

```
ErodeFilterType::Pointer  grayscaleErode  = ErodeFilterType::New();
DilateFilterType::Pointer grayscaleDilate = DilateFilterType::New();
```

The structuring element is not a reference counted class. Thus it is created as a C++ stack object instead of using `New()` and SmartPointers. The radius of the neighborhood associated with the structuring element is defined with the `SetRadius()` method and the `CreateStructuringElement()` method is invoked in order to initialize the operator. The resulting structuring element is passed to the mathematical morphology filter through the `SetKernel()` method, as illustrated below.

```
StructuringElementType  structuringElement;

structuringElement.SetRadius( 1 );  // 3x3 structuring element

structuringElement.CreateStructuringElement();

grayscaleErode->SetKernel(  structuringElement );
grayscaleDilate->SetKernel( structuringElement );
```

A grayscale image is provided as input to the filters. This image might be, for example, the output of a reader.

```
grayscaleErode->SetInput(  reader->GetOutput() );
grayscaleDilate->SetInput( reader->GetOutput() );
```

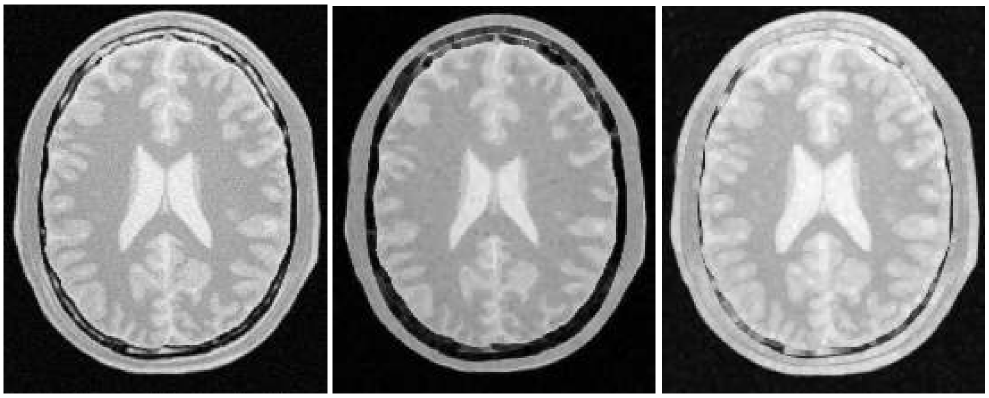


Figure 6.16: Effect of erosion and dilation in a grayscale image.

The filter is executed by invoking its `Update()` method, or by updating any downstream filter, like, for example, an image writer.

```
writerDilation->SetInput( grayscaleDilate->GetOutput() );
writerDilation->Update();
```

Figure 6.16 illustrates the effect of the erosion and dilation filters on a binary image from a MRI brain slice. The figure shows how these operations can be used to remove spurious details from segmented images.

6.6.4 Voting Filters

Voting filters are quite a generic family of filters. In fact, both the Dilate and Erode filters from Mathematical Morphology are very particular cases of the broader family of voting filters. In a voting filter, the outcome of a pixel is decided by counting the number of pixels in its neighborhood and applying a rule to the result of that counting.For example, the typical implementation of Erosion in terms of a voting filter will be to say that a foreground pixel will become background if the numbers of background neighbors is greater or equal than 1. In this context, you could imagine variations of Erosion in which the count could be changed to require at least 3 foreground.

Binary Median Filter

One of the particular cases of Voting filters is the BinaryMedianImageFilter. This filter is equivalent to applying a Median filter over a binary image. The fact of having a binary image as input makes possible to optimize the execution of the filter since there is no real need for sorting the pixels according to their frequency in the neighborhood.

The source code for this section can be found in the file
`Examples/Filtering/BinaryMedianImageFilter.cxx`.

The `itk::BinaryMedianImageFilter` is commonly used as a robust approach for noise reduction. BinaryMedianImageFilter computes the value of each output pixel as the statistical median of the neighborhood of values around the corresponding input pixel. When the input images are binary, the implementation can be optimized by simply counting the number of pixels ON/OFF around the current pixel.

This filter will work on images of any dimension thanks to the internal use of `itk::NeighborhoodIterator` and `itk::NeighborhoodOperator`. The size of the neighborhood over which the median is computed can be set by the user.

The header file corresponding to this filter should be included first.

```
#include "itkBinaryMedianImageFilter.h"
```

Then the pixel and image types of the input and output must be defined.

```
  typedef   unsigned char  InputPixelType;
  typedef   unsigned char  OutputPixelType;

  typedef itk::Image< InputPixelType,  2 >   InputImageType;
  typedef itk::Image< OutputPixelType, 2 >   OutputImageType;
```

Using the image types, it is now possible to define the filter type and create the filter object.

```
  typedef itk::BinaryMedianImageFilter<
               InputImageType, OutputImageType >  FilterType;

  FilterType::Pointer filter = FilterType::New();
```

The size of the neighborhood is defined along every dimension by passing a `SizeType` object with the corresponding values. The value on each dimension is used as the semi-size of a rectangular box. For example, in $2D$ a size of $1,2$ will result in a 3×5 neighborhood.

```
  InputImageType::SizeType indexRadius;

  indexRadius[0] = radiusX; // radius along x
  indexRadius[1] = radiusY; // radius along y

  filter->SetRadius( indexRadius );
```

The input to the filter can be taken from any other filter, for example a reader. The output can be passed down the pipeline to other filters, for example, a writer. An update call on any downstream filter will trigger the execution of the median filter.

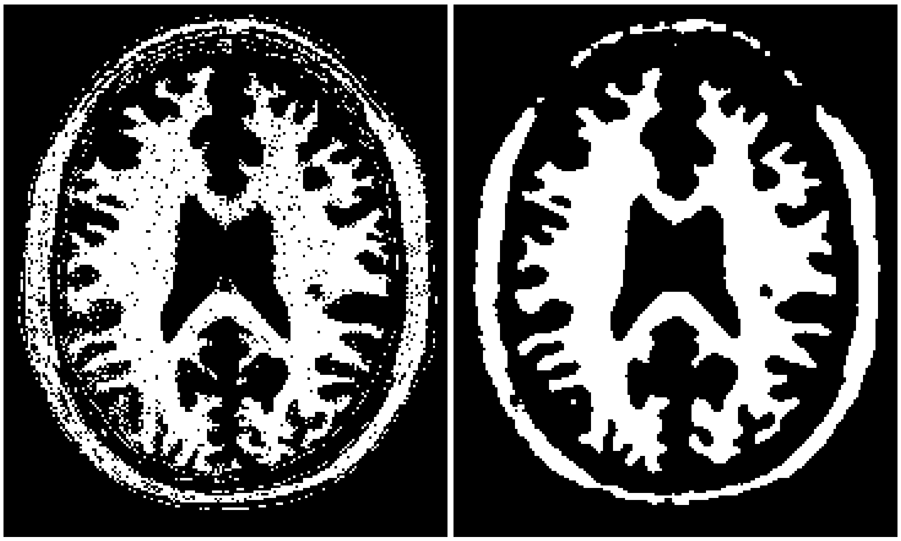


Figure 6.17: Effect of the BinaryMedianImageFilter on a slice from a MRI proton density brain image that has been thresholded in order to produce a binary image.

```
  filter->SetInput( reader->GetOutput() );
  writer->SetInput( filter->GetOutput() );
  writer->Update();
```

Figure 6.17 illustrates the effect of the BinaryMedianImageFilter filter on a slice of MRI brain image using a neighborhood radius of $2,2$, which corresponds to a 5×5 classical neighborhood. The filtered image demonstrates the capability of this filter for reducing noise both in the background and foreground of the image, as well as smoothing the contours of the regions.

The typical effect of median filtration on a noisy digital image is a dramatic reduction in impulse noise spikes. The filter also tends to preserve brightness differences across signal steps, resulting in reduced blurring of regional boundaries. The filter also tends to preserve the positions of boundaries in an image.

Figure 6.18 below shows the effect of running the median filter with a 3x3 classical window size 1, 10 and 50 times. There is a tradeoff in noise reduction and the sharpness of the image when the window size is increased.

Hole Filling Filter

Another variation of Voting filters is the Hole Filling filter. This filter converts background pixels into foreground only when the number of foreground pixels is a majority of the neighbors. By selecting the size of the majority, this filter can be tuned to fill-in holes of different size. To

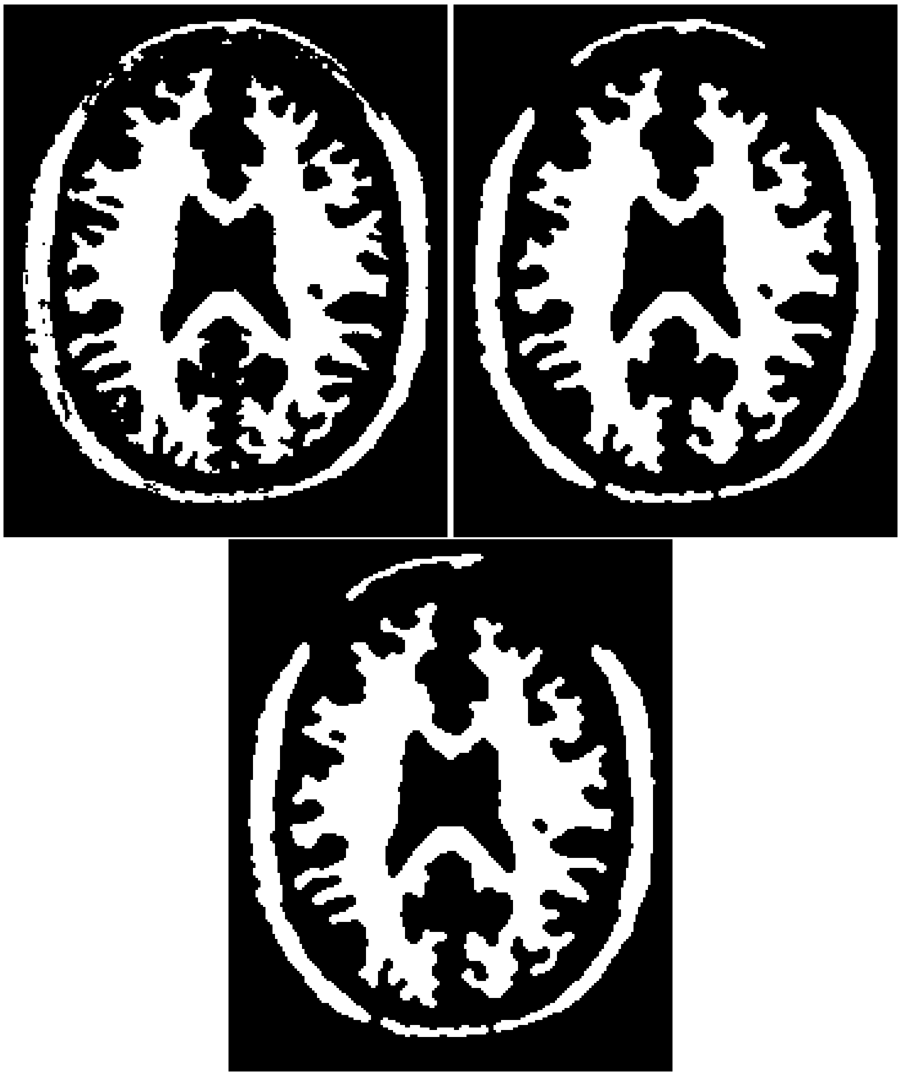


Figure 6.18: Effect of 1, 10 and 50 iterations of the BinaryMedianImageFilter using a 3x3 window.

be more precise, the effect of the filter is actually related to the curvature of the edge in which the pixel is located.

The source code for this section can be found in the file
`Examples/Filtering/VotingBinaryHoleFillingImageFilter.cxx`.

The `itk::VotingBinaryHoleFillingImageFilter` applies a voting operation in order to fill-in cavities. This can be used for smoothing contours and for filling holes in binary images.

The header file corresponding to this filter should be included first.

```
#include "itkVotingBinaryHoleFillingImageFilter.h"
```

Then the pixel and image types of the input and output must be defined.

```
  typedef   unsigned char  InputPixelType;
  typedef   unsigned char  OutputPixelType;

  typedef itk::Image< InputPixelType,  2 >   InputImageType;
  typedef itk::Image< OutputPixelType, 2 >   OutputImageType;
```

Using the image types, it is now possible to define the filter type and create the filter object.

```
  typedef itk::VotingBinaryHoleFillingImageFilter<
               InputImageType, OutputImageType >  FilterType;

  FilterType::Pointer filter = FilterType::New();
```

The size of the neighborhood is defined along every dimension by passing a `SizeType` object with the corresponding values. The value on each dimension is used as the semi-size of a rectangular box. For example, in $2D$ a size of $1,2$ will result in a 3×5 neighborhood.

```
  InputImageType::SizeType indexRadius;

  indexRadius[0] = radiusX; // radius along x
  indexRadius[1] = radiusY; // radius along y

  filter->SetRadius( indexRadius );
```

Since the filter is expecting a binary image as input, we must specify the levels that are going to be considered background and foreground. This is done with the `SetForegroundValue()` and `SetBackgroundValue()` methods.

```
  filter->SetBackgroundValue(   0 );
  filter->SetForegroundValue( 255 );
```

We must also specify the majority threshold that is going to be used as the decision criterion for converting a background pixel into a foreground pixel. The rule of conversion is that a background pixel will be converted into a foreground pixel if the number of foreground neighbors surpass the number of background neighbors by the majority value. For example, in a 2D image, with neighborhood or radius 1, the neighborhood will have size 3×3. If we set the majority value to 2, then we are requiring that the number of foreground neighbors should be at least (3x3 -1)/2 + majority. This is done with the `SetMajorityThreshold()` method.

```
  filter->SetMajorityThreshold( 2 );
```

The input to the filter can be taken from any other filter, for example a reader. The output can be passed down the pipeline to other filters, for example, a writer. An update call on any downstream filter will trigger the execution of the median filter.

```
  filter->SetInput( reader->GetOutput() );
  writer->SetInput( filter->GetOutput() );
  writer->Update();
```

Figure 6.19 illustrates the effect of the VotingBinaryHoleFillingImageFilter filter on a thresholded slice of MRI brain image using neighborhood radii of $1,1$, $2,2$ and $3,3$ that correspond respectively to neighborhoods of size 3×3, 5×5, 7×7. The filtered image demonstrates the capability of this filter for reducing noise both in the background and foreground of the image, as well as smoothing the contours of the regions.

Iterative Hole Filling Filter

The Hole Filling filter can be used in an iterative way, by applying it repeatedly until no pixel changes. In this context, the filter can be seen as a binary variation of a Level Set filter.

The source code for this section can be found in the file
`Examples/Filtering/VotingBinaryIterativeHoleFillingImageFilter.cxx`.

The `itk::VotingBinaryIterativeHoleFillingImageFilter` applies a voting operation in order to fill-in cavities. This can be used for smoothing contours and for filling holes in binary images. This filter runs internally a `itk::VotingBinaryHoleFillingImageFilter` until no pixels change or the maximum number of iterations has been reached.

The header file corresponding to this filter should be included first.

```
#include "itkVotingBinaryIterativeHoleFillingImageFilter.h"
```

Then the pixel and image types must be defined. Note that this filter requires the input and output images to be of the same type, therefore a single image type is required for the template instantiation.

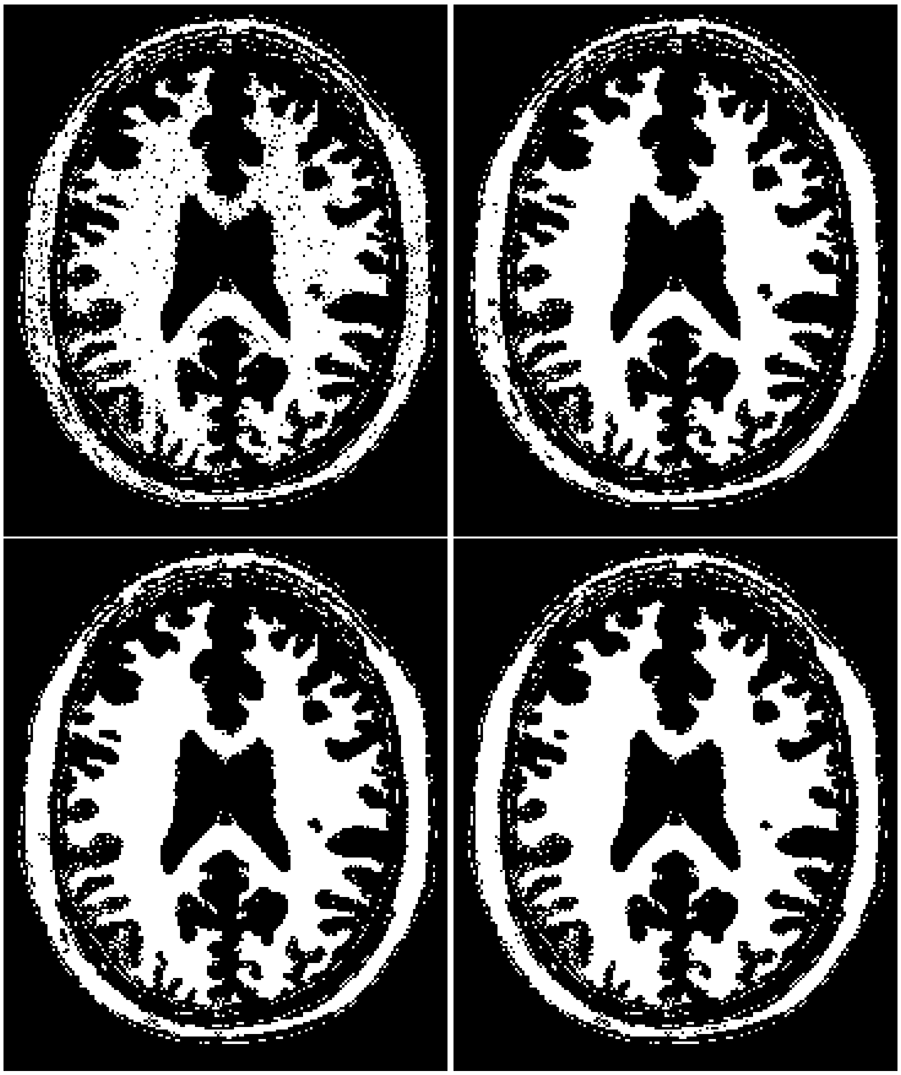


Figure 6.19: Effect of the VotingBinaryHoleFillingImageFilter on a slice from a MRI proton density brain image that has been thresholded in order to produce a binary image. The output images have used radius 1,2 and 3 respectively.

```
  typedef   unsigned char  PixelType;

  typedef itk::Image< PixelType, 2 >   ImageType;
```

Using the image types, it is now possible to define the filter type and create the filter object.

```
  typedef itk::VotingBinaryIterativeHoleFillingImageFilter<
                                        ImageType >  FilterType;

  FilterType::Pointer filter = FilterType::New();
```

The size of the neighborhood is defined along every dimension by passing a `SizeType` object with the corresponding values. The value on each dimension is used as the semi-size of a rectangular box. For example, in $2D$ a size of $1,2$ will result in a 3×5 neighborhood.

```
  ImageType::SizeType indexRadius;

  indexRadius[0] = radiusX; // radius along x
  indexRadius[1] = radiusY; // radius along y

  filter->SetRadius( indexRadius );
```

Since the filter is expecting a binary image as input, we must specify the levels that are going to be considered background and foreground. This is done with the `SetForegroundValue()` and `SetBackgroundValue()` methods.

```
  filter->SetBackgroundValue(   0 );
  filter->SetForegroundValue( 255 );
```

We must also specify the majority threshold that is going to be used as the decision criterion for converting a background pixel into a foreground pixel. The rule of conversion is that a background pixel will be converted into a foreground pixel if the number of foreground neighbors surpass the number of background neighbors by the majority value. For example, in a 2D image, with neighborhood or radius 1, the neighborhood will have size 3×3. If we set the majority value to 2, then we are requiring that the number of foreground neighbors should be at least (3x3 -1)/2 + majority. This is done with the `SetMajorityThreshold()` method.

```
  filter->SetMajorityThreshold( 2 );
```

Finally we specify the maximum number of iterations that this filter should be run. The number of iteration will determine the maximum size of holes and cavities that this filter will be able to fill-in. The more iterations you ran, the larger the cavities that will be filled in.

```
  filter->SetMaximumNumberOfIterations( numberOfIterations );
```

The input to the filter can be taken from any other filter, for example a reader. The output can be passed down the pipeline to other filters, for example, a writer. An update call on any downstream filter will trigger the execution of the median filter.

```
filter->SetInput( reader->GetOutput() );
writer->SetInput( filter->GetOutput() );
writer->Update();
```

Figure 6.20 illustrates the effect of the VotingBinaryIterativeHoleFillingImageFilter filter on a thresholded slice of MRI brain image using neighborhood radii of $1,1$, $2,2$ and $3,3$ that correspond respectively to neighborhoods of size 3×3, 5×5, 7×7. The filtered image demonstrates the capability of this filter for reducing noise both in the background and foreground of the image, as well as smoothing the contours of the regions.

6.7 Smoothing Filters

Real image data has a level of uncertainty that is manifested in the variability of measures assigned to pixels. This uncertainty is usually interpreted as noise and considered an undesirable component of the image data. This section describes several methods that can be applied to reduce noise on images.

6.7.1 Blurring

Blurring is the traditional approach for removing noise from images. It is usually implemented in the form of a convolution with a kernel. The effect of blurring on the image spectrum is to attenuate high spatial frequencies. Different kernels attenuate frequencies in different ways. One of the most commonly used kernels is the Gaussian. Two implementations of Gaussian smoothing are available in the toolkit. The first one is based on a traditional convolution while the other is based on the application of IIR filters that approximate the convolution with a Gaussian [21, 22].

Discrete Gaussian

The source code for this section can be found in the file
`Examples/Filtering/DiscreteGaussianImageFilter.cxx`.

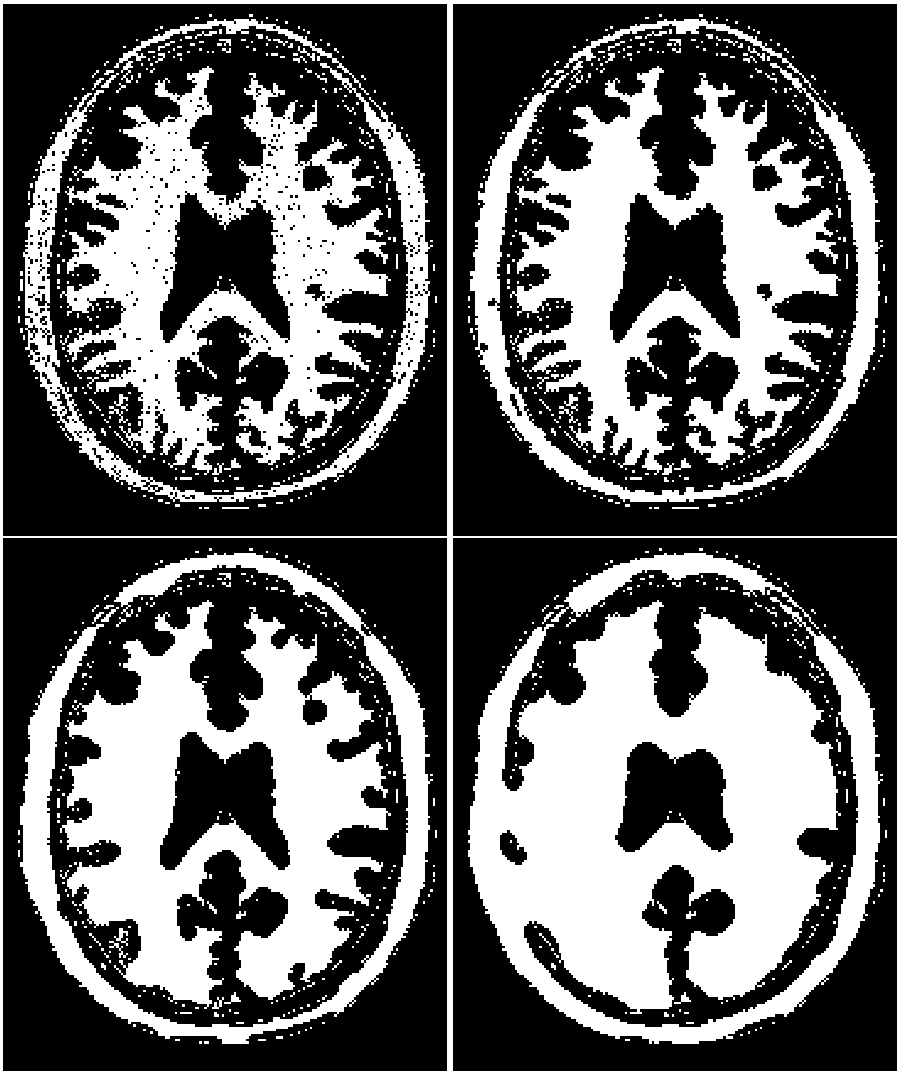


Figure 6.20: Effect of the VotingBinaryIterativeHoleFillingImageFilter on a slice from a MRI proton density brain image that has been thresholded in order to produce a binary image. The output images have used radius 1,2 and 3 respectively.

The itk::DiscreteGaussianImageFilter computes the convolution of the input image with a Gaussian kernel. This is done in *ND* by taking advantage of the separability of the Gaussian kernel. A one-dimensional Gaussian function is discretized on a convolution kernel. The size of the kernel is extended until there are enough discrete points in the Gaussian to ensure that a user-provided maximum error is not exceeded. Since the size of the kernel is unknown a priori, it is necessary to impose a limit to its growth. The user can thus provide a value to be the maximum admissible size of the kernel. Discretization error is defined as the difference between the area under the discrete Gaussian curve (which has finite support) and the area under the continuous Gaussian.

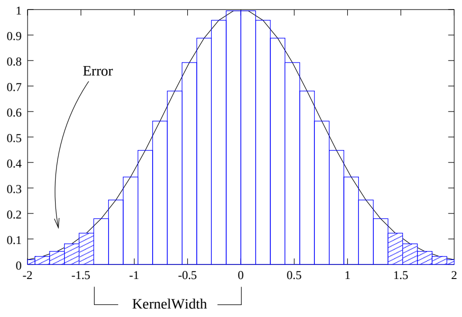


Figure 6.21: Discretized Gaussian.

Gaussian kernels in ITK are constructed according to the theory of Tony Lindeberg [49] so that smoothing and derivative operations commute before and after discretization. In other words, finite difference derivatives on an image I that has been smoothed by convolution with the Gaussian are equivalent to finite differences computed on I by convolving with a derivative of the Gaussian.

The first step required to use this filter is to include its header file.

```
#include "itkDiscreteGaussianImageFilter.h"
```

Types should be chosen for the pixels of the input and output images. Image types can be instantiated using the pixel type and dimension.

```
  typedef    float    InputPixelType;
  typedef    float    OutputPixelType;

  typedef itk::Image< InputPixelType,  2 >   InputImageType;
  typedef itk::Image< OutputPixelType, 2 >   OutputImageType;
```

The discrete Gaussian filter type is instantiated using the input and output image types. A corresponding filter object is created.

```
  typedef itk::DiscreteGaussianImageFilter<
                 InputImageType, OutputImageType >  FilterType;

  FilterType::Pointer filter = FilterType::New();
```

The input image can be obtained from the output of another filter. Here, an image reader is used as its input.

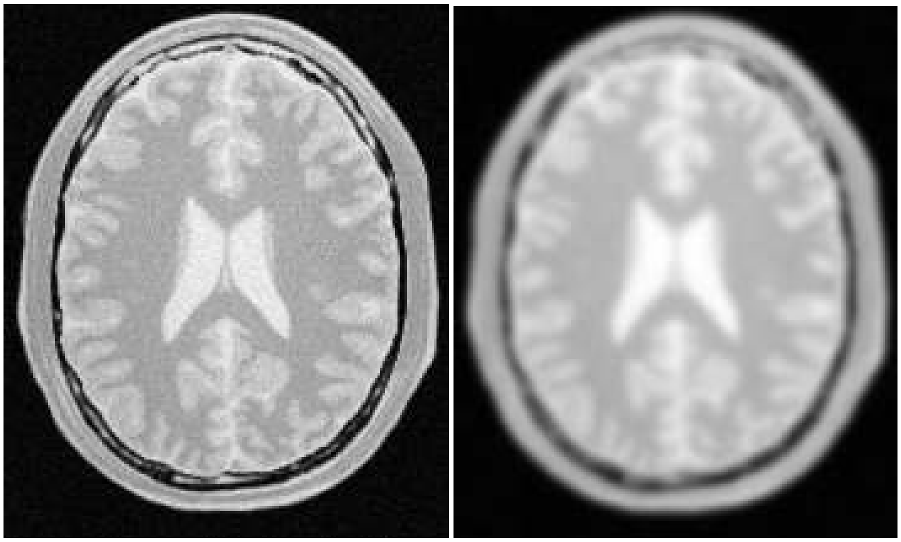


Figure 6.22: Effect of the DiscreteGaussianImageFilter on a slice from a MRI proton density image of the brain.

```
filter->SetInput( reader->GetOutput() );
```

The filter requires the user to provide a value for the variance associated with the Gaussian kernel. The method `SetVariance()` is used for this purpose. The discrete Gaussian is constructed as a convolution kernel. The maximum kernel size can be set by the user. Note that the combination of variance and kernel-size values may result in a truncated Gaussian kernel.

```
filter->SetVariance( gaussianVariance );
filter->SetMaximumKernelWidth( maxKernelWidth );
```

Finally, the filter is executed by invoking the `Update()` method.

```
filter->Update();
```

If the output of this filter has been connected to other filters down the pipeline, updating any of the downstream filters would have triggered the execution of this one. For example, a writer could have been used after the filter.

```
rescaler->SetInput( filter->GetOutput() );
writer->SetInput( rescaler->GetOutput() );
writer->Update();
```

Figure 6.22 illustrates the effect of this filter on a MRI proton density image of the brain.

Note that large Gaussian variances will produce large convolution kernels and correspondingly slower computation times. Unless a high degree of accuracy is required, it may be more desirable to use the approximating `itk::RecursiveGaussianImageFilter` with large variances.

Binomial Blurring

The source code for this section can be found in the file
`Examples/Filtering/BinomialBlurImageFilter.cxx`.

The `itk::BinomialBlurImageFilter` computes a nearest neighbor average along each dimension. The process is repeated a number of times, as specified by the user. In principle, after a large number of iterations the result will approach the convolution with a Gaussian.

The first step required to use this filter is to include its header file.

```
#include "itkBinomialBlurImageFilter.h"
```

Types should be chosen for the pixels of the input and output images. Image types can be instantiated using the pixel type and dimension.

```
  typedef    float    InputPixelType;
  typedef    float    OutputPixelType;

  typedef itk::Image< InputPixelType,  2 >   InputImageType;
  typedef itk::Image< OutputPixelType, 2 >   OutputImageType;
```

The filter type is now instantiated using both the input image and the output image types. Then a filter object is created.

```
  typedef itk::BinomialBlurImageFilter<
                 InputImageType, OutputImageType >  FilterType;
  FilterType::Pointer filter = FilterType::New();
```

The input image can be obtained from the output of another filter. Here, an image reader is used as the source. The number of repetitions is set with the `SetRepetitions()` method. Computation time will increase linearly with the number of repetitions selected. Finally, the filter can be executed by calling the `Update()` method.

```
  filter->SetInput( reader->GetOutput() );
  filter->SetRepetitions( repetitions );
  filter->Update();
```

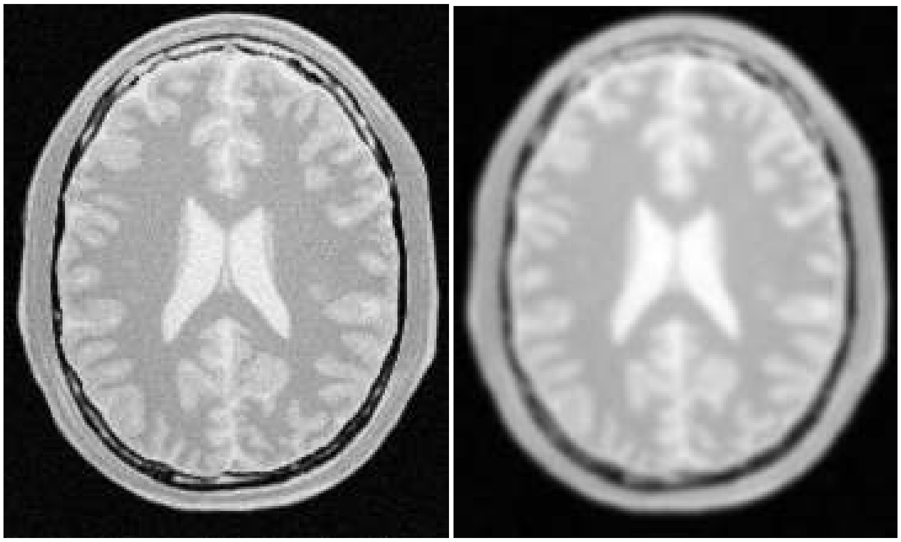


Figure 6.23: Effect of the BinomialBlurImageFilter on a slice from a MRI proton density image of the brain.

Figure 6.23 illustrates the effect of this filter on a MRI proton density image of the brain.

Note that the standard deviation σ of the equivalent Gaussian is fixed. In the spatial spectrum, the effect of every iteration of this filter is like a multiplication with a sinus cardinal function.

Recursive Gaussian IIR

The source code for this section can be found in the file
`Examples/Filtering/SmoothingRecursiveGaussianImageFilter.cxx`.

The classical method of smoothing an image by convolution with a Gaussian kernel has the drawback that it is slow when the standard deviation σ of the Gaussian is large. This is due to the larger size of the kernel, which results in a higher number of computations per pixel.

The `itk::RecursiveGaussianImageFilter` implements an approximation of convolution with the Gaussian and its derivatives by using IIR[2] filters. In practice this filter requires a constant number of operations for approximating the convolution, regardless of the σ value [21, 22].

The first step required to use this filter is to include its header file.

```
#include "itkRecursiveGaussianImageFilter.h"
```

Types should be selected on the desired input and output pixel types.

[2]Infinite Impulse Response

```
typedef    float    InputPixelType;
typedef    float    OutputPixelType;
```

The input and output image types are instantiated using the pixel types.

```
typedef itk::Image< InputPixelType,  2 >   InputImageType;
typedef itk::Image< OutputPixelType, 2 >   OutputImageType;
```

The filter type is now instantiated using both the input image and the output image types.

```
typedef itk::RecursiveGaussianImageFilter<
                      InputImageType, OutputImageType >  FilterType;
```

This filter applies the approximation of the convolution along a single dimension. It is therefore necessary to concatenate several of these filters to produce smoothing in all directions. In this example, we create a pair of filters since we are processing a $2D$ image. The filters are created by invoking the `New()` method and assigning the result to a `itk::SmartPointer`.

```
FilterType::Pointer filterX = FilterType::New();
FilterType::Pointer filterY = FilterType::New();
```

Since each one of the newly created filters has the potential to perform filtering along any dimension, we have to restrict each one to a particular direction. This is done with the `SetDirection()` method.

```
filterX->SetDirection( 0 );   // 0 --> X direction
filterY->SetDirection( 1 );   // 1 --> Y direction
```

The `itk::RecursiveGaussianImageFilter` can approximate the convolution with the Gaussian or with its first and second derivatives. We select one of these options by using the `SetOrder()` method. Note that the argument is an `enum` whose values can be `ZeroOrder`, `FirstOrder` and `SecondOrder`. For example, to compute the x partial derivative we should select `FirstOrder` for x and `ZeroOrder` for y. Here we want only to smooth in x and y, so we select `ZeroOrder` in both directions.

```
filterX->SetOrder( FilterType::ZeroOrder );
filterY->SetOrder( FilterType::ZeroOrder );
```

There are two typical ways of normalizing Gaussians depending on their application. For scale-space analysis it is desirable to use a normalization that will preserve the maximum value of the input. This normalization is represented by the following equation.

$$\frac{1}{\sigma\sqrt{2\pi}} \tag{6.4}$$

In applications that use the Gaussian as a solution of the diffusion equation it is desirable to use a normalization that preserve the integral of the signal. This last approach can be seen as a conservation of mass principle. This is represented by the following equation.

$$\frac{1}{\sigma^2\sqrt{2\pi}} \tag{6.5}$$

The itk::RecursiveGaussianImageFilter has a boolean flag that allows users to select between these two normalization options. Selection is done with the method `SetNormalizeAcrossScale()`. Enable this flag to analyzing an image across scale-space. In the current example, this setting has no impact because we are actually renormalizing the output to the dynamic range of the reader, so we simply disable the flag.

```
filterX->SetNormalizeAcrossScale( false );
filterY->SetNormalizeAcrossScale( false );
```

The input image can be obtained from the output of another filter. Here, an image reader is used as the source. The image is passed to the x filter and then to the y filter. The reason for keeping these two filters separate is that it is usual in scale-space applications to compute not only the smoothing but also combinations of derivatives at different orders and smoothing. Some factorization is possible when separate filters are used to generate the intermediate results. Here this capability is less interesting, though, since we only want to smooth the image in all directions.

```
filterX->SetInput( reader->GetOutput() );
filterY->SetInput( filterX->GetOutput() );
```

It is now time to select the σ of the Gaussian used to smooth the data. Note that σ must be passed to both filters and that sigma is considered to be in millimeters. That is, at the moment of applying the smoothing process, the filter will take into account the spacing values defined in the image.

```
filterX->SetSigma( sigma );
filterY->SetSigma( sigma );
```

Finally the pipeline is executed by invoking the `Update()` method.

```
filterY->Update();
```

Figure 6.24 illustrates the effect of this filter on a MRI proton density image of the brain using σ values of 3 (left) and 5 (right). The figure shows how the attenuation of noise can be regulated by selecting the appropriate standard deviation. This type of scale-tunable filter is suitable for performing scale-space analysis.

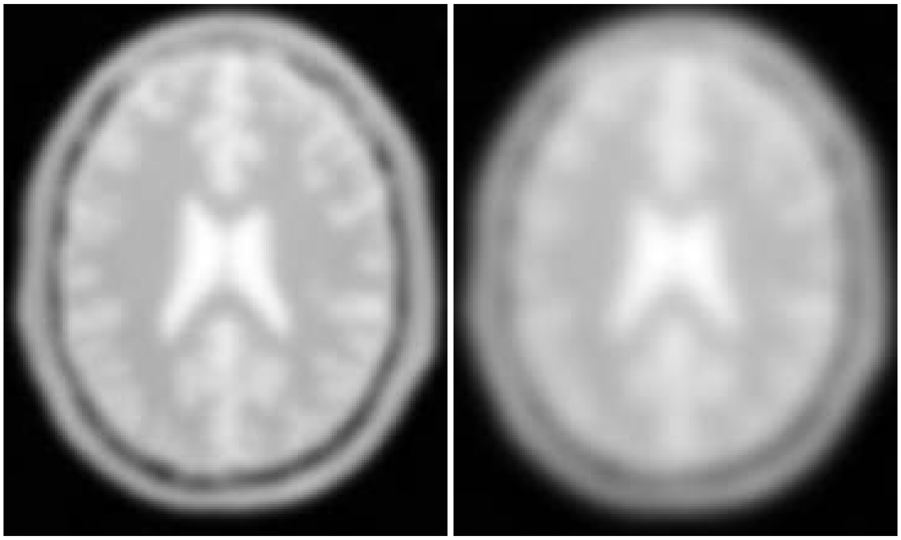


Figure 6.24: Effect of the SmoothingRecursiveGaussianImageFilter on a slice from a MRI proton density image of the brain.

The RecursiveGaussianFilters can also be applied on multi-component images. For instance, the above filter could have applied with RGBPixel as the pixel type. Each component is then independently filtered. However the RescaleIntensityImageFilter will not work on RGBPixels since it does not mathematically make sense to rescale the output of multi-component images.

6.7.2 Local Blurring

In some cases it is desirable to compute smoothing in restricted regions of the image, or to do it using different parameters that are computed locally. The following sections describe options for applying local smoothing in images.

Gaussian Blur Image Function

The source code for this section can be found in the file
`Examples/Filtering/GaussianBlurImageFunction.cxx`.

6.7.3 Edge Preserving Smoothing

Introduction to Anisotropic Diffusion

The drawback of image denoising (smoothing) is that it tends to blur away the sharp boundaries in the image that help to distinguish between the larger-scale anatomical structures that one is trying to characterize (which also limits the size of the smoothing kernels in most applications). Even in cases where smoothing does not obliterate boundaries, it tends to distort the fine structure of the image and thereby changes subtle aspects of the anatomical shapes in question.

Perona and Malik [63] introduced an alternative to linear-filtering that they called *anisotropic diffusion*. Anisotropic diffusion is closely related to the earlier work of Grossberg [32], who used similar nonlinear diffusion processes to model human vision. The motivation for anisotropic diffusion (also called *nonuniform* or *variable conductance* diffusion) is that a Gaussian smoothed image is a single time slice of the solution to the heat equation, that has the original image as its initial conditions. Thus, the solution to

$$\frac{\partial g(x,y,t)}{\partial t} = \nabla \cdot \nabla g(x,y,t), \tag{6.6}$$

where $g(x,y,0) = f(x,y)$ is the input image, is $g(x,y,t) = G(\sqrt{2t}) \otimes f(x,y)$, where $G(\sigma)$ is a Gaussian with standard deviation σ.

Anisotropic diffusion includes a variable conductance term that, in turn, depends on the differential structure of the image. Thus, the variable conductance can be formulated to limit the smoothing at "edges" in images, as measured by high gradient magnitude, for example.

$$g_t = \nabla \cdot c(|\nabla g|)\nabla g, \tag{6.7}$$

where, for notational convenience, we leave off the independent parameters of g and use the subscripts with respect to those parameters to indicate partial derivatives. The function $c(|\nabla g|)$ is a fuzzy cutoff that reduces the conductance at areas of large $|\nabla g|$, and can be any one of a number of functions. The literature has shown

$$c(|\nabla g|) = e^{-\frac{|\nabla g|^2}{2k^2}} \tag{6.8}$$

to be quite effective. Notice that conductance term introduces a free parameter k, the *conductance parameter*, that controls the sensitivity of the process to edge contrast. Thus, anisotropic diffusion entails two free parameters: the conductance parameter, k, and the time parameter, t, that is analogous to σ, the effective width of the filter when using Gaussian kernels.

Equation 6.7 is a nonlinear partial differential equation that can be solved on a discrete grid using finite forward differences. Thus, the smoothed image is obtained only by an iterative process, not a convolution or non-stationary, linear filter. Typically, the number of iterations required for practical results are small, and large 2D images can be processed in several tens of seconds using carefully written code running on modern, general purpose, single-processor computers. The technique applies readily and effectively to 3D images, but requires more processing time.

In the early 1990's several research groups [29, 89] demonstrated the effectiveness of anisotropic diffusion on medical images. In a series of papers on the subject [93, 91, 92, 89, 90, 87], Whitaker described a detailed analytical and empirical analysis, introduced a smoothing term in the conductance that made the process more robust, invented a numerical scheme that virtually eliminated directional artifacts in the original algorithm, and generalized anisotropic diffusion to vector-valued images, an image processing technique that can be used on vector-valued medical data (such as the color cryosection data of the Visible Human Project).

For a vector-valued input $\vec{F} : U \mapsto \Re^m$ the process takes the form

$$\vec{F}_t = \nabla \cdot c(\mathcal{D}\vec{F})\vec{F}, \tag{6.9}$$

where $\mathcal{D}\vec{F}$ is a *dissimilarity* measure of $\vec{F}$, a generalization of the gradient magnitude to vector-valued images, that can incorporate linear and nonlinear coordinate transformations on the range of $\vec{F}$. In this way, the smoothing of the multiple images associated with vector-valued data is coupled through the conductance term, that fuses the information in the different images. Thus vector-valued, nonlinear diffusion can combine low-level image features (e.g. edges) across all "channels" of a vector-valued image in order to preserve or enhance those features in all of image "channels".

Vector-valued anisotropic diffusion is useful for denoising data from devices that produce multiple values such as MRI or color photography. When performing nonlinear diffusion on a color image, the color channels are diffused separately, but linked through the conductance term. Vector-valued diffusion it is also useful for processing registered data from different devices or for denoising higher-order geometric or statistical features from scalar-valued images [87, 95].

The output of anisotropic diffusion is an image or set of images that demonstrates reduced noise and texture but preserves, and can also enhance, edges. Such images are useful for a variety of processes including statistical classification, visualization, and geometric feature extraction. Previous work has shown [90] that anisotropic diffusion, over a wide range of conductance parameters, offers quantifiable advantages over linear filtering for edge detection in medical images.

Since the effectiveness of nonlinear diffusion was first demonstrated, numerous variations of this approach have surfaced in the literature [79]. These include alternatives for constructing dissimilarity measures [71], directional (i.e., tensor-valued) conductance terms [86, 3] and level set interpretations [88].

Gradient Anisotropic Diffusion

The source code for this section can be found in the file
`Examples/Filtering/GradientAnisotropicDiffusionImageFilter.cxx`.

The `itk::GradientAnisotropicDiffusionImageFilter` implements an N-dimensional version of the classic Perona-Malik anisotropic diffusion equation for scalar-valued images [63].

The conductance term for this implementation is chosen as a function of the gradient magnitude of the image at each point, reducing the strength of diffusion at edge pixels.

$$C(\mathbf{x}) = e^{-\left(\frac{\|\nabla U(\mathbf{x})\|}{K}\right)^2} \tag{6.10}$$

The numerical implementation of this equation is similar to that described in the Perona-Malik paper [63], but uses a more robust technique for gradient magnitude estimation and has been generalized to N-dimensions.

The first step required to use this filter is to include its header file.

```
#include "itkGradientAnisotropicDiffusionImageFilter.h"
```

Types should be selected based on the pixel types required for the input and output images. The image types are defined using the pixel type and the dimension.

```
  typedef    float    InputPixelType;
  typedef    float    OutputPixelType;

  typedef itk::Image< InputPixelType,  2 >   InputImageType;
  typedef itk::Image< OutputPixelType, 2 >   OutputImageType;
```

The filter type is now instantiated using both the input image and the output image types. The filter object is created by the `New()` method.

```
  typedef itk::GradientAnisotropicDiffusionImageFilter<
               InputImageType, OutputImageType >  FilterType;
  FilterType::Pointer filter = FilterType::New();
```

The input image can be obtained from the output of another filter. Here, an image reader is used as source.

```
  filter->SetInput( reader->GetOutput() );
```

This filter requires three parameters, the number of iterations to be performed, the time step and the conductance parameter used in the computation of the level set evolution. These parameters are set using the methods `SetNumberOfIterations()`, `SetTimeStep()` and `SetConductanceParameter()` respectively. The filter can be executed by invoking Update().

```
  filter->SetNumberOfIterations( numberOfIterations );
  filter->SetTimeStep( timeStep );
  filter->SetConductanceParameter( conductance );

  filter->Update();
```

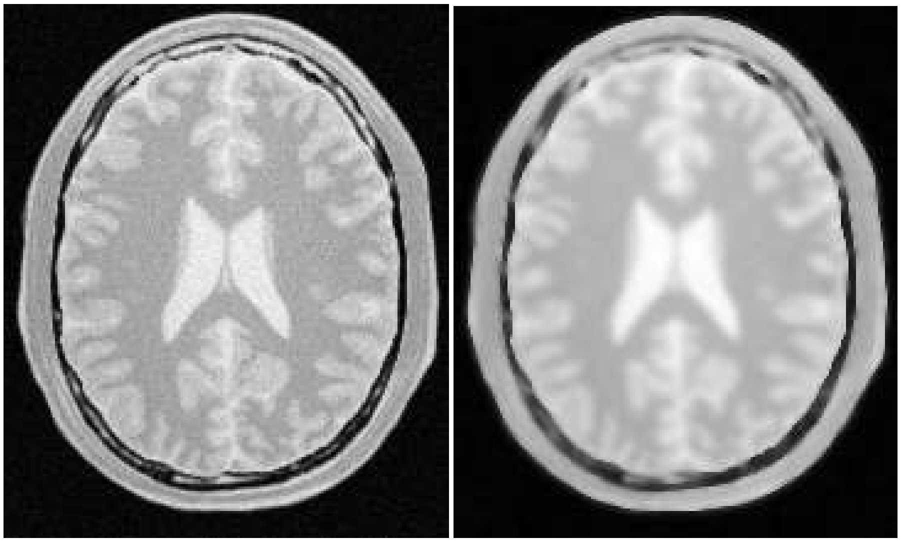


Figure 6.25: Effect of the GradientAnisotropicDiffusionImageFilter on a slice from a MRI Proton Density image of the brain.

Typical values for the time step are 0.25 in 2*D* images and 0.125 in 3*D* images. The number of iterations is typically set to 5; more iterations result in further smoothing and will increase the computing time linearly.

Figure 6.25 illustrates the effect of this filter on a MRI proton density image of the brain. In this example the filter was run with a time step of 0.25, and 5 iterations. The figure shows how homogeneous regions are smoothed and edges are preserved.

The following classes provide similar functionality:

- `itk::BilateralImageFilter`
- `itk::CurvatureAnisotropicDiffusionImageFilter`
- `itk::CurvatureFlowImageFilter`

Curvature Anisotropic Diffusion

The source code for this section can be found in the file
`Examples/Filtering/CurvatureAnisotropicDiffusionImageFilter.cxx`.

The `itk::CurvatureAnisotropicDiffusionImageFilter` performs anisotropic diffusion on an image using a modified curvature diffusion equation (MCDE).

MCDE does not exhibit the edge enhancing properties of classic anisotropic diffusion, which can under certain conditions undergo a “negative” diffusion, which enhances the contrast of edges. Equations of the form of MCDE always undergo positive diffusion, with the conductance term only varying the strength of that diffusion.

Qualitatively, MCDE compares well with other non-linear diffusion techniques. It is less sensitive to contrast than classic Perona-Malik style diffusion, and preserves finer detailed structures in images. There is a potential speed trade-off for using this function in place of itkGradientNDAnisotropicDiffusionFunction. Each iteration of the solution takes roughly twice as long. Fewer iterations, however, may be required to reach an acceptable solution.

The MCDE equation is given as:

$$f_t = \mid \nabla f \mid \nabla \cdot c(\mid \nabla f \mid) \frac{\nabla f}{\mid \nabla f \mid} \tag{6.11}$$

where the conductance modified curvature term is

$$\nabla \cdot \frac{\nabla f}{\mid \nabla f \mid} \tag{6.12}$$

The first step required for using this filter is to include its header file

```
#include "itkCurvatureAnisotropicDiffusionImageFilter.h"
```

Types should be selected based on the pixel types required for the input and output images. The image types are defined using the pixel type and the dimension.

```
  typedef    float    InputPixelType;
  typedef    float    OutputPixelType;

  typedef itk::Image< InputPixelType,  2 >   InputImageType;
  typedef itk::Image< OutputPixelType, 2 >   OutputImageType;
```

The filter type is now instantiated using both the input image and the output image types. The filter object is created by the `New()` method.

```
  typedef itk::CurvatureAnisotropicDiffusionImageFilter<
               InputImageType, OutputImageType >  FilterType;

  FilterType::Pointer filter = FilterType::New();
```

The input image can be obtained from the output of another filter. Here, an image reader is used as source.

```
  filter->SetInput( reader->GetOutput() );
```

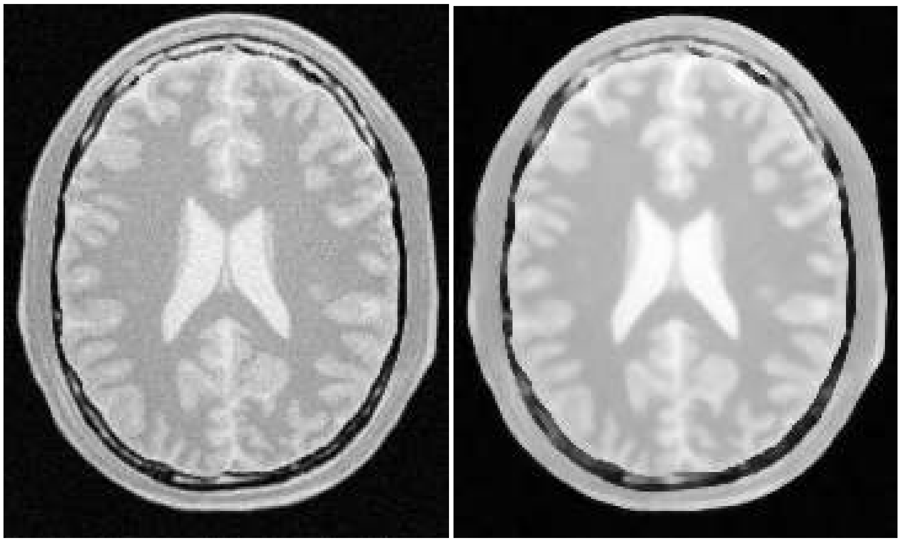


Figure 6.26: Effect of the CurvatureAnisotropicDiffusionImageFilter on a slice from a MRI Proton Density image of the brain.

This filter requires three parameters, the number of iterations to be performed, the time step used in the computation of the level set evolution and the value of conductance. These parameters are set using the methods `SetNumberOfIterations()`, `SetTimeStep()` and `SetConductance()` respectively. The filter can be executed by invoking `Update()`.

```
filter->SetNumberOfIterations( numberOfIterations );
filter->SetTimeStep( timeStep );
filter->SetConductanceParameter( conductance );
if (useImageSpacing)
  {
  filter->UseImageSpacingOn();
  }
filter->Update();
```

Typical values for the time step are 0.125 in 2*D* images and 0.0625 in 3*D* images. The number of iterations can be usually around 5, more iterations will result in further smoothing and will increase linearly the computing time. The conductance parameter is usually around 3.0.

Figure 6.26 illustrates the effect of this filter on a MRI proton density image of the brain. In this example the filter was run with a time step of 0.125, 5 iterations and a conductance value of 3.0. The figure shows how homogeneous regions are smoothed and edges are preserved.

The following classes provide similar functionality:

- `itk::BilateralImageFilter`

- itk::CurvatureFlowImageFilter
- itk::GradientAnisotropicDiffusionImageFilter

Curvature Flow

The source code for this section can be found in the file
`Examples/Filtering/CurvatureFlowImageFilter.cxx`.

The itk::CurvatureFlowImageFilter performs edge-preserving smoothing in a similar fashion to the classical anisotropic diffusion. The filter uses a level set formulation where the iso-intensity contours in a image are viewed as level sets, where pixels of a particular intensity form one level set. The level set function is then evolved under the control of a diffusion equation where the speed is proportional to the curvature of the contour:

$$I_t = \kappa |\nabla I| \tag{6.13}$$

where κ is the curvature.

Areas of high curvature will diffuse faster than areas of low curvature. Hence, small jagged noise artifacts will disappear quickly, while large scale interfaces will be slow to evolve, thereby preserving sharp boundaries between objects. However, it should be noted that although the evolution at the boundary is slow, some diffusion still occur. Thus, continual application of this curvature flow scheme will eventually result is the removal of information as each contour shrinks to a point and disappears.

The first step required to use this filter is to include its header file.

```
#include "itkCurvatureFlowImageFilter.h"
```

Types should be selected based on the pixel types required for the input and output images.

```
  typedef    float    InputPixelType;
  typedef    float    OutputPixelType;
```

With them, the input and output image types can be instantiated.

```
  typedef itk::Image< InputPixelType,  2 >   InputImageType;
  typedef itk::Image< OutputPixelType, 2 >   OutputImageType;
```

The CurvatureFlow filter type is now instantiated using both the input image and the output image types.

```
  typedef itk::CurvatureFlowImageFilter<
               InputImageType, OutputImageType >  FilterType;
```

A filter object is created by invoking the `New()` method and assigning the result to a `itk::SmartPointer`.

```
  FilterType::Pointer filter = FilterType::New();
```

The input image can be obtained from the output of another filter. Here, an image reader is used as source.

```
  filter->SetInput( reader->GetOutput() );
```

The CurvatureFlow filter requires two parameters, the number of iterations to be performed and the time step used in the computation of the level set evolution. These two parameters are set using the methods `SetNumberOfIterations()` and `SetTimeStep()` respectively. Then the filter can be executed by invoking `Update()`.

```
  filter->SetNumberOfIterations( numberOfIterations );
  filter->SetTimeStep( timeStep );
  filter->Update();
```

Typical values for the time step are 0.125 in $2D$ images and 0.0625 in $3D$ images. The number of iterations can be usually around 10, more iterations will result in further smoothing and will increase linearly the computing time. Edge-preserving behavior is not guaranteed by this filter, some degradation will occur on the edges and will increase as the number of iterations is increased.

If the output of this filter has been connected to other filters down the pipeline, updating any of the downstream filters will triggered the execution of this one. For example, a writer filter could have been used after the curvature flow filter.

```
  rescaler->SetInput( filter->GetOutput() );
  writer->SetInput( rescaler->GetOutput() );
  writer->Update();
```

Figure 6.27 illustrates the effect of this filter on a MRI proton density image of the brain. In this example the filter was run with a time step of 0.25 and 10 iterations. The figure shows how homogeneous regions are smoothed and edges are preserved.

The following classes provide similar functionality:

- `itk::GradientAnisotropicDiffusionImageFilter`
- `itk::CurvatureAnisotropicDiffusionImageFilter`
- `itk::BilateralImageFilter`

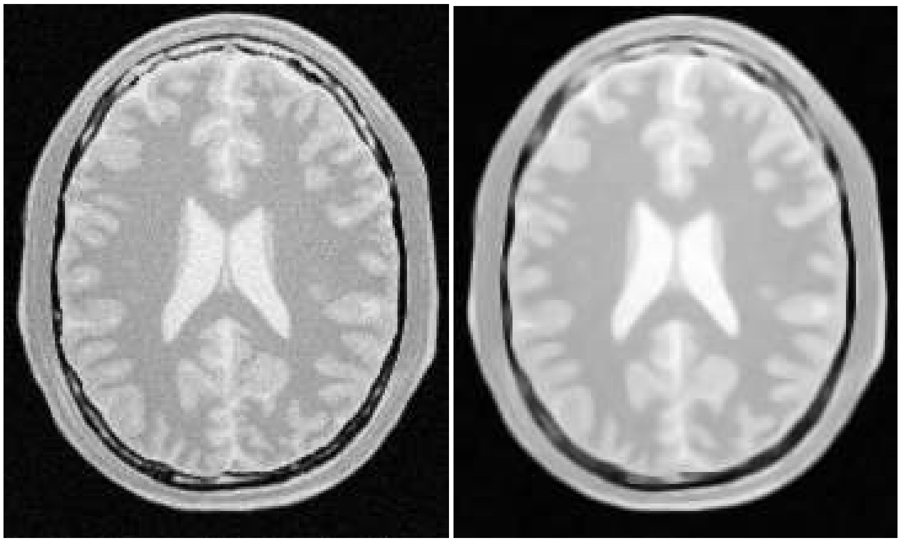


Figure 6.27: Effect of the CurvatureFlowImageFilter on a slice from a MRI proton density image of the brain.

MinMaxCurvature Flow

The source code for this section can be found in the file
`Examples/Filtering/MinMaxCurvatureFlowImageFilter.cxx`.

The MinMax curvature flow filter applies a variant of the curvature flow algorithm where diffusion is turned on or off depending of the scale of the noise that one wants to remove. The evolution speed is switched between $\min(\kappa,0)$ and $\max(\kappa,0)$ such that:

$$I_t = F|\nabla I| \tag{6.14}$$

where F is defined as

$$F = \begin{cases} \max(\kappa,0) & : \quad \text{Average} < Threshold \\ \min(\kappa,0) & : \quad \text{Average} \geq Threshold \end{cases} \tag{6.15}$$

The *Average* is the average intensity computed over a neighborhood of a user specified radius of the pixel. The choice of the radius governs the scale of the noise to be removed. The $Threshold$ is calculated as the average of pixel intensities along the direction perpendicular to the gradient at the *extrema* of the local neighborhood.

A speed of $F = max(\kappa,0)$ will cause small dark regions in a predominantly light region to shrink. Conversely, a speed of $F = min(\kappa,0)$, will cause light regions in a predominantly dark region to shrink. Comparison between the neighborhood average and the threshold is used to

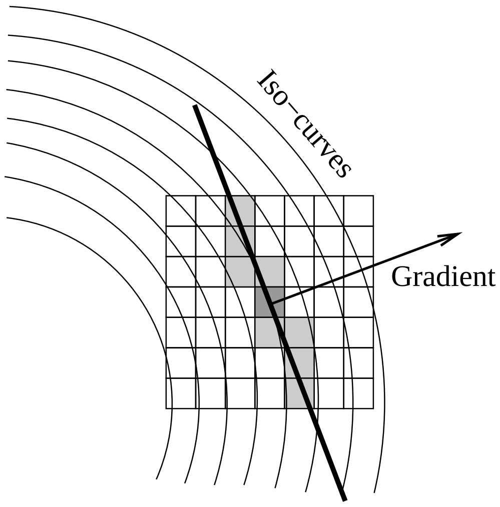


Figure 6.28: Elements involved in the computation of min-max curvature flow.

select the the right speed function to use. This switching prevents the unwanted diffusion of the simple curvature flow method.

Figure 6.28 shows the main elements involved in the computation. The set of square pixels represent the neighborhood over which the average intensity is being computed. The gray pixels are those lying close to the direction perpendicular to the gradient. The pixels which intersect the neighborhood bounds are used to compute the threshold value in the equation above. The integer radius of the neighborhood is selected by the user.

The first step required to use the `itk::MinMaxCurvatureFlowImageFilter` is to include its header file.

```
#include "itkMinMaxCurvatureFlowImageFilter.h"
```

Types should be selected based on the pixel types required for the input and output images. The input and output image types are instantiated.

```
  typedef    float    InputPixelType;
  typedef    float    OutputPixelType;

  typedef itk::Image< InputPixelType,  2 >   InputImageType;
  typedef itk::Image< OutputPixelType, 2 >   OutputImageType;
```

The `itk::MinMaxCurvatureFlowImageFilter` type is now instantiated using both the input image and the output image types. The filter is then created using the `New()` method.

```
typedef itk::MinMaxCurvatureFlowImageFilter<
             InputImageType, OutputImageType >  FilterType;
FilterType::Pointer filter = FilterType::New();
```

The input image can be obtained from the output of another filter. Here, an image reader is used as source.

```
filter->SetInput( reader->GetOutput() );
```

The `itk::MinMaxCurvatureFlowImageFilter` requires the two normal parameters of the CurvatureFlow image, the number of iterations to be performed and the time step used in the computation of the level set evolution. In addition to them, the radius of the neighborhood is also required. This last parameter is passed using the `SetStencilRadius()` method. Note that the radius is provided as an integer number since it is referring to a number of pixels from the center to the border of the neighborhood. Then the filter can be executed by invoking `Update()`.

```
filter->SetTimeStep( timeStep );
filter->SetNumberOfIterations( numberOfIterations );
filter->SetStencilRadius( radius );
filter->Update();
```

Typical values for the time step are 0.125 in $2D$ images and 0.0625 in $3D$ images. The number of iterations can be usually around 10, more iterations will result in further smoothing and will increase the computing time linearly. The radius of the stencil can be typically 1. The *edge-preserving* characteristic is not perfect on this filter, some degradation will occur on the edges and will increase as the number of iterations is increased.

If the output of this filter has been connected to other filters down the pipeline, updating any of the downstream filters would have triggered the execution of this one. For example, a writer filter could have been used after the curvature flow filter.

```
rescaler->SetInput( filter->GetOutput() );
writer->SetInput( rescaler->GetOutput() );
writer->Update();
```

Figure 6.29 illustrates the effect of this filter on a MRI proton density image of the brain. In this example the filter was run with a time step of 0.125, 10 iterations and a radius of 1. The figure shows how homogeneous regions are smoothed and edges are preserved. Notice also, that the results in the figure has sharper edges than the same example using simple curvature flow in Figure 6.27.

The following classes provide similar functionality:

- `itk::CurvatureFlowImageFilter`

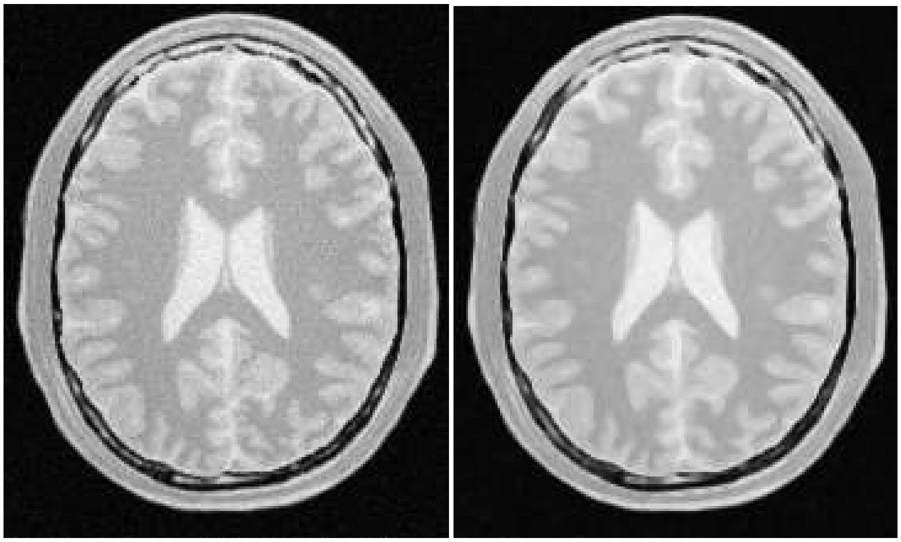


Figure 6.29: Effect of the MinMaxCurvatureFlowImageFilter on a slice from a MRI proton density image of the brain.

Bilateral Filter

The source code for this section can be found in the file
`Examples/Filtering/BilateralImageFilter.cxx`.

The `itk::BilateralImageFilter` performs smoothing by using both domain and range neighborhoods. Pixels that are close to a pixel in the image domain and similar to a pixel in the image range are used to calculate the filtered value. Two Gaussian kernels (one in the image domain and one in the image range) are used to smooth the image. The result is an image that is smoothed in homogeneous regions yet has edges preserved. The result is similar to anisotropic diffusion but the implementation in non-iterative. Another benefit to bilateral filtering is that any distance metric can be used for kernel smoothing the image range. Bilateral filtering is capable of reducing the noise in an image by an order of magnitude while maintaining edges. The bilateral operator used here was described by Tomasi and Manduchi (*Bilateral Filtering for Gray and Color Images*. IEEE ICCV. 1998.)

The filtering operation can be described by the following equation

$$h(\mathbf{x}) = k(\mathbf{x})^{-1} \int_{\omega} f(\mathbf{w}) c(\mathbf{x}, \mathbf{w}) s(f(\mathbf{x}), f(\mathbf{w})) d\mathbf{w} \tag{6.16}$$

where $\mathbf{x}$ holds the coordinates of a ND point, $f(\mathbf{x})$ is the input image and $h(\mathbf{x})$ is the output image. The convolution kernels $c()$ and $s()$ are associated with the spatial and intensity domain respectively. The ND integral is computed over ω which is a neighborhood of the pixel located at $\mathbf{x}$. The normalization factor $k(\mathbf{x})$ is computed as

$$k(\mathbf{x}) = \int_{\omega} c(\mathbf{x},\mathbf{w}) s(f(\mathbf{x}), f(\mathbf{w})) d\mathbf{w} \tag{6.17}$$

The default implementation of this filter uses Gaussian kernels for both $c()$ and $s()$. The c kernel can be described as

$$c(\mathbf{x},\mathbf{w}) = e^{\left(\frac{||\mathbf{x}-\mathbf{w}||^2}{\sigma_c^2}\right)} \tag{6.18}$$

where σ_c is provided by the user and defines how close pixel neighbors should be in order to be considered for the computation of the output value. The s kernel is given by

$$s(f(\mathbf{x}), f(\mathbf{w})) = e^{\left(\frac{(f(\mathbf{x})-f(\mathbf{w})^2}{\sigma_s^2}\right)} \tag{6.19}$$

where σ_s is provided by the user and defines how close should the neighbors intensity be in order to be considered for the computation of the output value.

The first step required to use this filter is to include its header file.

```
#include "itkBilateralImageFilter.h"
```

The image types are instantiated using pixel type and dimension.

```
  typedef    unsigned char    InputPixelType;
  typedef    unsigned char    OutputPixelType;

  typedef itk::Image< InputPixelType,  2 >   InputImageType;
  typedef itk::Image< OutputPixelType, 2 >   OutputImageType;
```

The bilateral filter type is now instantiated using both the input image and the output image types and the filter object is created.

```
  typedef itk::BilateralImageFilter<
               InputImageType, OutputImageType >  FilterType;
  FilterType::Pointer filter = FilterType::New();
```

The input image can be obtained from the output of another filter. Here, an image reader is used as a source.

```
  filter->SetInput( reader->GetOutput() );
```

The Bilateral filter requires two parameters. First, the σ to be used for the Gaussian kernel on image intensities. Second, the set of σs to be used along each dimension in the space domain. This second parameter is supplied as an array of `float` or `double` values. The array dimension matches the image dimension. This mechanism makes possible to enforce more coherence along some directions. For example, more smoothing can be done along the X direction than along the Y direction.

In the following code example, the σ values are taken from the command line. Note the use of `ImageType::ImageDimension` to get access to the image dimension at compile time.

```
const unsigned int Dimension = InputImageType::ImageDimension;
double domainSigmas[ Dimension ];
for(unsigned int i=0; i<Dimension; i++)
  {
  domainSigmas[i] = atof( argv[3] );
  }
const double rangeSigma = atof( argv[4] );
```

The filter parameters are set with the methods SetRangeSigma() and SetDomainSigma().

```
filter->SetDomainSigma( domainSigmas );
filter->SetRangeSigma(  rangeSigma   );
```

The output of the filter is connected here to a intensity rescaler filter and then to a writer. Invoking `Update()` on the writer triggers the execution of both filters.

```
rescaler->SetInput( filter->GetOutput() );
writer->SetInput( rescaler->GetOutput() );
writer->Update();
```

Figure 6.30 illustrates the effect of this filter on a MRI proton density image of the brain. In this example the filter was run with a range sigma of 5.0 and a domain σ of 6.0. The figure shows how homogeneous regions are smoothed and edges are preserved.

The following classes provide similar functionality:

- `itk::GradientAnisotropicDiffusionImageFilter`
- `itk::CurvatureAnisotropicDiffusionImageFilter`
- `itk::CurvatureFlowImageFilter`

6.7.4 Edge Preserving Smoothing in Vector Images

Anisotropic diffusion can also be applied to images whose pixels are vectors. In this case the diffusion is computed independently for each vector component. The following classes implement versions of anisotropic diffusion on vector images.

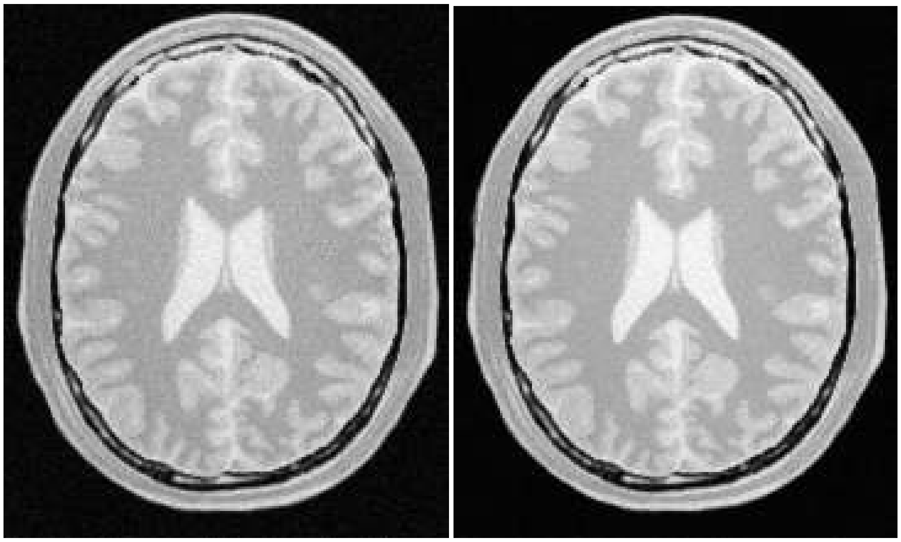


Figure 6.30: Effect of the BilateralImageFilter on a slice from a MRI proton density image of the brain.

Vector Gradient Anisotropic Diffusion

The source code for this section can be found in the file
`Examples/Filtering/VectorGradientAnisotropicDiffusionImageFilter.cxx`.

The `itk::VectorGradientAnisotropicDiffusionImageFilter` implements an *N*-dimensional version of the classic Perona-Malik anisotropic diffusion equation for vector-valued images. Typically in vector-valued diffusion, vector components are diffused independently of one another using a conductance term that is linked across the components. The diffusion equation was illustrated in 6.7.3

This filter is designed to process images of `itk::Vector` type. The code relies on various typedefs and overloaded operators defined in Vector. It is perfectly reasonable, however, to apply this filter to images of other, user-defined types as long as the appropriate typedefs and operator overloads are in place. As a general rule, follow the example of Vector in defining your data types.

The first step required to use this filter is to include its header file.

```
#include "itkVectorGradientAnisotropicDiffusionImageFilter.h"
```

Types should be selected based on required pixel type for the input and output images. The image types are defined using the pixel type and the dimension.

```
  typedef    float    InputPixelType;
```

```
typedef itk::CovariantVector<float,2>    VectorPixelType;
typedef itk::Image< InputPixelType,  2 >   InputImageType;
typedef itk::Image< VectorPixelType, 2 >   VectorImageType;
```

The filter type is now instantiated using both the input image and the output image types. The filter object is created by the `New()` method.

```
typedef itk::VectorGradientAnisotropicDiffusionImageFilter<
                     VectorImageType, VectorImageType >  FilterType;
FilterType::Pointer filter = FilterType::New();
```

The input image can be obtained from the output of another filter. Here, an image reader is used as source and its data is passed through a gradient filter in order to generate an image of vectors.

```
gradient->SetInput( reader->GetOutput() );
filter->SetInput( gradient->GetOutput() );
```

This filter requires two parameters, the number of iterations to be performed and the time step used in the computation of the level set evolution. These parameters are set using the methods `SetNumberOfIterations()` and `SetTimeStep()` respectively. The filter can be executed by invoking `Update()`.

```
filter->SetNumberOfIterations( numberOfIterations );
filter->SetTimeStep( timeStep );
filter->SetConductanceParameter(1.0);
filter->Update();
```

Typical values for the time step are 0.125 in $2D$ images and 0.0625 in $3D$ images. The number of iterations can be usually around 5, more iterations will result in further smoothing and will linearly increase the computing time.

Figure 6.31 illustrates the effect of this filter on a MRI proton density image of the brain. The images show the X component of the gradient before (left) and after (right) the application of the filter. In this example the filter was run with a time step of 0.25, and 5 iterations.

Vector Curvature Anisotropic Diffusion

The source code for this section can be found in the file
`Examples/Filtering/VectorCurvatureAnisotropicDiffusionImageFilter.cxx`.

The `itk::VectorCurvatureAnisotropicDiffusionImageFilter` performs anisotropic diffusion on a vector image using a modified curvature diffusion equation (MCDE). The MCDE is the same described in 6.7.3.

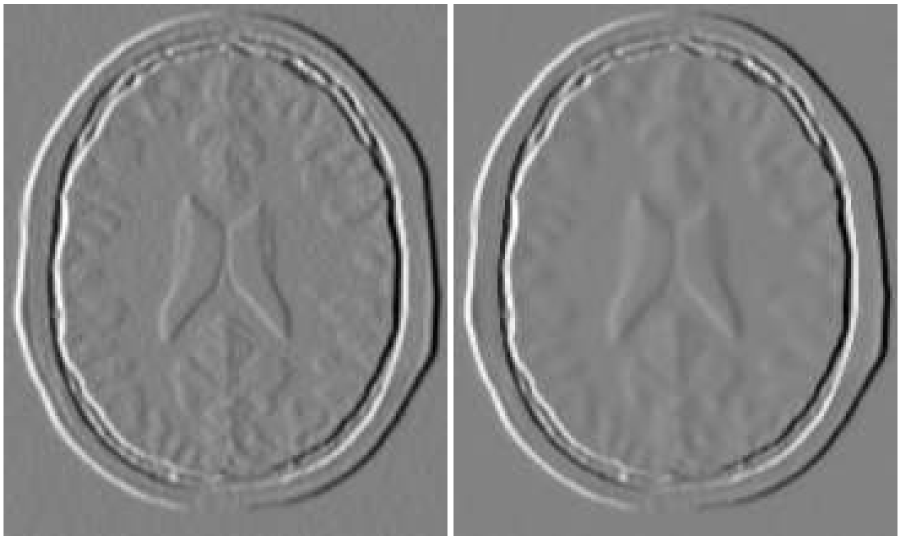


Figure 6.31: Effect of the VectorGradientAnisotropicDiffusionImageFilter on the X component of the gradient from a MRI proton density brain image.

Typically in vector-valued diffusion, vector components are diffused independently of one another using a conductance term that is linked across the components.

This filter is designed to process images of `itk::Vector` type. The code relies on various typedefs and overloaded operators defined in Vector. It is perfectly reasonable, however, to apply this filter to images of other, user-defined types as long as the appropriate typedefs and operator overloads are in place. As a general rule, follow the example of the Vector class in defining your data types.

The first step required to use this filter is to include its header file.

```
#include "itkVectorCurvatureAnisotropicDiffusionImageFilter.h"
```

Types should be selected based on required pixel type for the input and output images. The image types are defined using the pixel type and the dimension.

```
  typedef    float    InputPixelType;
  typedef itk::CovariantVector<float,2>    VectorPixelType;
  typedef itk::Image< InputPixelType,  2 >   InputImageType;
  typedef itk::Image< VectorPixelType, 2 >   VectorImageType;
```

The filter type is now instantiated using both the input image and the output image types. The filter object is created by the `New()` method.

```
  typedef itk::VectorCurvatureAnisotropicDiffusionImageFilter<
                        VectorImageType, VectorImageType >  FilterType;
  FilterType::Pointer filter = FilterType::New();
```

The input image can be obtained from the output of another filter. Here, an image reader is used as source and its data is passed through a gradient filter in order to generate an image of vectors.

```
  gradient->SetInput( reader->GetOutput() );
  filter->SetInput( gradient->GetOutput() );
```

This filter requires two parameters, the number of iterations to be performed and the time step used in the computation of the level set evolution. These parameters are set using the methods `SetNumberOfIterations()` and `SetTimeStep()` respectively. The filter can be executed by invoking `Update()`.

```
  filter->SetNumberOfIterations( numberOfIterations );
  filter->SetTimeStep( timeStep );
  filter->SetConductanceParameter(1.0);
  filter->Update();
```

Typical values for the time step are 0.125 in $2D$ images and 0.0625 in $3D$ images. The number of iterations can be usually around 5, more iterations will result in further smoothing and will increase linearly the computing time.

Figure 6.32 illustrates the effect of this filter on a MRI proton density image of the brain. The images show the X component of the gradient before (left) and after (right) the application of the filter. In this example the filter was run with a time step of 0.25, and 5 iterations.

6.7.5 Edge Preserving Smoothing in Color Images

Gradient Anisotropic Diffusion

The source code for this section can be found in the file
`Examples/Filtering/RGBGradientAnisotropicDiffusionImageFilter.cxx`.

The vector anisotropic diffusion approach can equally well be applied to color images. As in the vector case, each RGB component is diffused independently. The following example illustrates the use of the Vector curvature anisotropic diffusion filter on an image with `itk::RGBPixel` type.

The first step required to use this filter is to include its header file.

```
#include "itkVectorGradientAnisotropicDiffusionImageFilter.h"
```

Also the headers for `Image` and `RGBPixel` type are required.

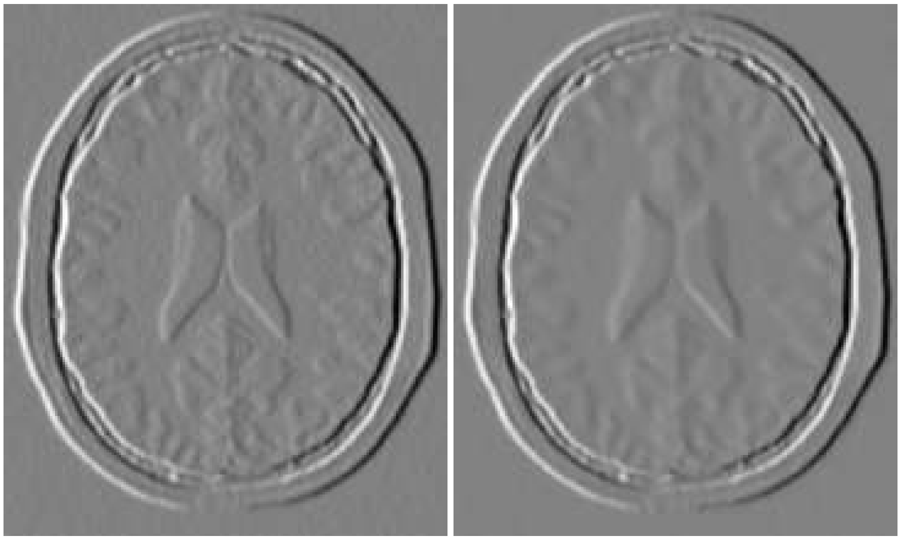


Figure 6.32: Effect of the VectorCurvatureAnisotropicDiffusionImageFilter on the X component of the gradient from a MRIproton density brain image.

```
#include "itkRGBPixel.h"
#include "itkImage.h"
```

It is desirable to perform the computation on the RGB image using `float` representation. However for input and output purposes `unsigned char` RGB components are commonly used. It is necessary to cast the type of color components along the pipeline before writing them to a file. The `itk::VectorCastImageFilter` is used to achieve this goal.

```
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
#include "itkVectorCastImageFilter.h"
```

The image type is defined using the pixel type and the dimension.

```
  typedef   itk::RGBPixel< float >     InputPixelType;
  typedef itk::Image< InputPixelType,  2 >   InputImageType;
```

The filter type is now instantiated and a filter object is created by the `New()` method.

```
  typedef itk::VectorGradientAnisotropicDiffusionImageFilter<
                       InputImageType, InputImageType >  FilterType;
  FilterType::Pointer filter = FilterType::New();
```

The input image can be obtained from the output of another filter. Here, an image reader is used as source.

```
typedef itk::ImageFileReader< InputImageType >  ReaderType;
ReaderType::Pointer reader = ReaderType::New();
reader->SetFileName( argv[1] );
filter->SetInput( reader->GetOutput() );
```

This filter requires two parameters, the number of iterations to be performed and the time step used in the computation of the level set evolution. These parameters are set using the methods `SetNumberOfIterations()` and `SetTimeStep()` respectively. The filter can be executed by invoking `Update()`.

```
filter->SetNumberOfIterations( numberOfIterations );
filter->SetTimeStep( timeStep );
filter->SetConductanceParameter(1.0);
filter->Update();
```

The filter output is now cast to `unsigned char` RGB components by using the `itk::VectorCastImageFilter`.

```
typedef itk::RGBPixel< unsigned char >   WritePixelType;
typedef itk::Image< WritePixelType, 2 >  WriteImageType;
typedef itk::VectorCastImageFilter<
              InputImageType, WriteImageType >  CasterType;
CasterType::Pointer caster = CasterType::New();
```

Finally, the writer type can be instantiated. One writer is created and connected to the output of the cast filter.

```
typedef itk::ImageFileWriter< WriteImageType >  WriterType;
WriterType::Pointer writer = WriterType::New();
caster->SetInput( filter->GetOutput() );
writer->SetInput( caster->GetOutput() );
writer->SetFileName( argv[2] );
writer->Update();
```

Figure 6.33 illustrates the effect of this filter on a RGB image from a cryogenic section of the Visible Woman data set. In this example the filter was run with a time step of 0.125, and 20 iterations. The input image has 570×670 pixels and the processing took 4 minutes on a Pentium 4 2Ghz.

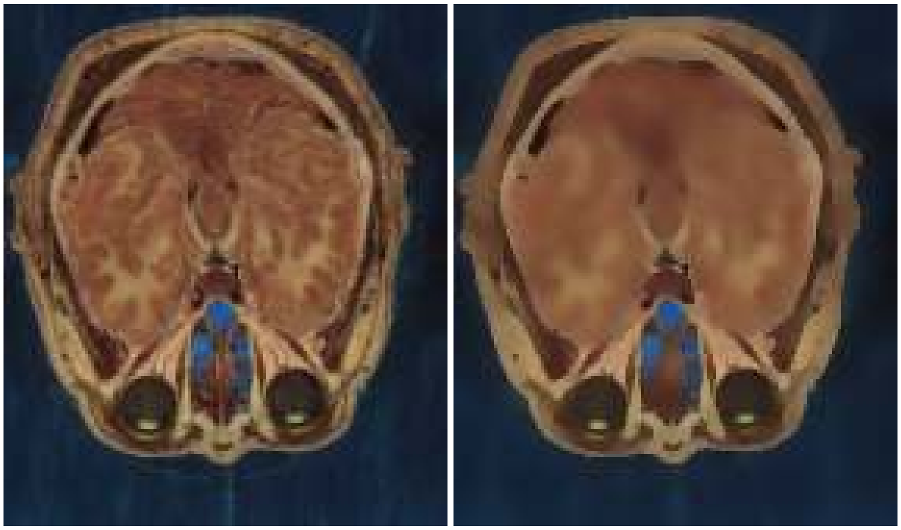


Figure 6.33: Effect of the VectorGradientAnisotropicDiffusionImageFilter on a RGB image from a cryogenic section of the Visible Woman data set.

Curvature Anisotropic Diffusion

The source code for this section can be found in the file
`Examples/Filtering/RGBCurvatureAnisotropicDiffusionImageFilter.cxx`.

The vector anisotropic diffusion approach can equally well be applied to color images. As in the vector case, each RGB component is diffused independently. The following example illustrates the use of the `itk::VectorCurvatureAnisotropicDiffusionImageFilter` on an image with `itk::RGBPixel` type.

The first step required to use this filter is to include its header file.

```
#include "itkVectorCurvatureAnisotropicDiffusionImageFilter.h"
```

Also the headers for `Image` and `RGBPixel` type are required.

```
#include "itkRGBPixel.h"
#include "itkImage.h"
```

It is desirable to perform the computation on the RGB image using `float` representation. However for input and output purposes `unsigned char` RGB components are commonly used. It is necessary to cast the type of color components in the pipeline before writing them to a file. The `itk::VectorCastImageFilter` is used to achieve this goal.

```
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
#include "itkVectorCastImageFilter.h"
```

The image type is defined using the pixel type and the dimension.

```
  typedef   itk::RGBPixel< float >     InputPixelType;
  typedef itk::Image< InputPixelType,  2 >   InputImageType;
```

The filter type is now instantiated and a filter object is created by the `New()` method.

```
  typedef itk::VectorCurvatureAnisotropicDiffusionImageFilter<
                       InputImageType, InputImageType >  FilterType;
  FilterType::Pointer filter = FilterType::New();
```

The input image can be obtained from the output of another filter. Here, an image reader is used as source.

```
  typedef itk::ImageFileReader< InputImageType >  ReaderType;
  ReaderType::Pointer reader = ReaderType::New();
  reader->SetFileName( argv[1] );
  filter->SetInput( reader->GetOutput() );
```

This filter requires two parameters, the number of iterations to be performed and the time step used in the computation of the level set evolution. These parameters are set using the methods `SetNumberOfIterations()` and `SetTimeStep()` respectively. The filter can be executed by invoking `Update()`.

```
  filter->SetNumberOfIterations( numberOfIterations );
  filter->SetTimeStep( timeStep );
  filter->SetConductanceParameter(1.0);
  filter->Update();
```

The filter output is now cast to `unsigned char` RGB components by using the VectorCastImageFilter

```
  typedef itk::RGBPixel< unsigned char >   WritePixelType;
  typedef itk::Image< WritePixelType, 2 >  WriteImageType;
  typedef itk::VectorCastImageFilter<
                InputImageType, WriteImageType >  CasterType;
  CasterType::Pointer caster = CasterType::New();
```

Finally, the writer type can be instantiated. One writer is created and connected to the output of the cast filter.

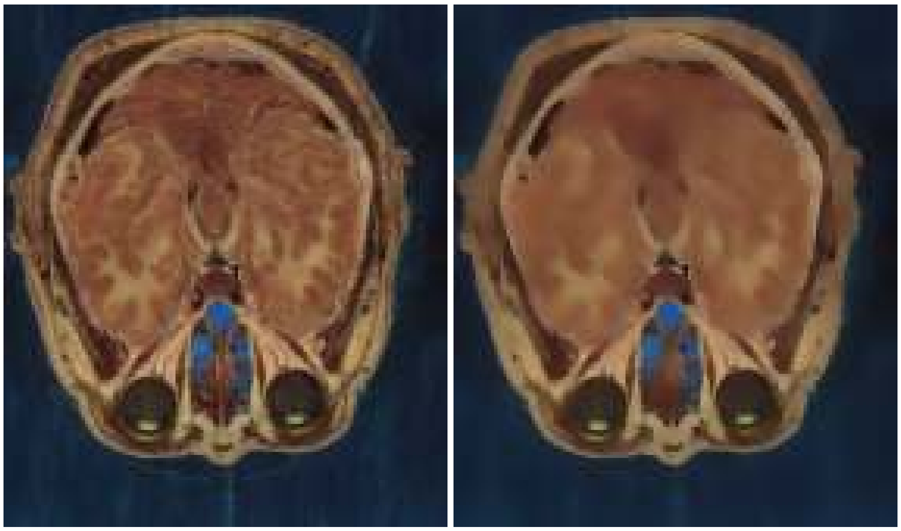


Figure 6.34: Effect of the VectorCurvatureAnisotropicDiffusionImageFilter on a RGB image from a cryogenic section of the Visible Woman data set.

```
typedef itk::ImageFileWriter< WriteImageType >  WriterType;
WriterType::Pointer writer = WriterType::New();
caster->SetInput( filter->GetOutput() );
writer->SetInput( caster->GetOutput() );
writer->SetFileName( argv[2] );
writer->Update();
```

Figure 6.34 illustrates the effect of this filter on a RGB image from a cryogenic section of the Visible Woman data set. In this example the filter was run with a time step of 0.125, and 20 iterations. The input image has 570×670 pixels and the processing took 4 minutes on a Pentium 4 at 2Ghz.

Figure 6.35 compares the effect of the gradient and curvature anisotropic diffusion filters on a small region of the same cryogenic slice used in Figure 6.34. The region used in this figure is only 127×162 pixels and took 14 seconds to compute on the same platform.

6.8 Distance Map

The source code for this section can be found in the file
`Examples/Filtering/DanielssonDistanceMapImageFilter.cxx`.

This example illustrates the use of the `itk::DanielssonDistanceMapImageFilter`. This filter generates a distance map from the input image using the algorithm developed by Danielsson

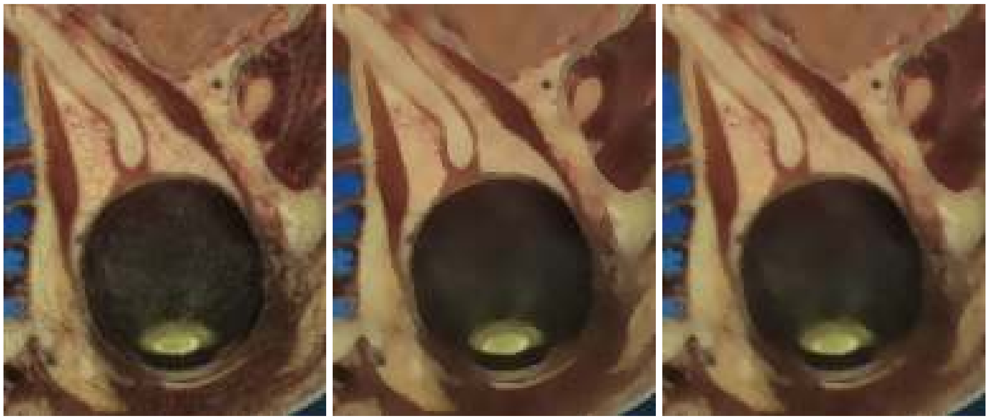


Figure 6.35: Comparison between the gradient (center) and curvature (right) Anisotropic Diffusion filters. Original image at left.

[18]. As secondary outputs, a Voronoi partition of the input elements is produced, as well as a vector image with the components of the distance vector to the closest point. The input to the map is assumed to be a set of points on the input image. Each point/pixel is considered to be a separate entity even if they share the same gray level value.

The first step required to use this filter is to include its header file.

```
#include "itkDanielssonDistanceMapImageFilter.h"
```

Then we must decide what pixel types to use for the input and output images. Since the output will contain distances measured in pixels, the pixel type should be able to represent at least the width of the image, or said in $N-D$ terms, the maximum extension along all the dimensions. The input and output image types are now defined using their respective pixel type and dimension.

```
  typedef  unsigned char   InputPixelType;
  typedef  unsigned short  OutputPixelType;
  typedef itk::Image< InputPixelType,  2 >   InputImageType;
  typedef itk::Image< OutputPixelType, 2 >   OutputImageType;
```

The filter type can be instantiated using the input and output image types defined above. A filter object is created with the `New()` method.

```
  typedef itk::DanielssonDistanceMapImageFilter<
               InputImageType, OutputImageType >  FilterType;
  FilterType::Pointer filter = FilterType::New();
```

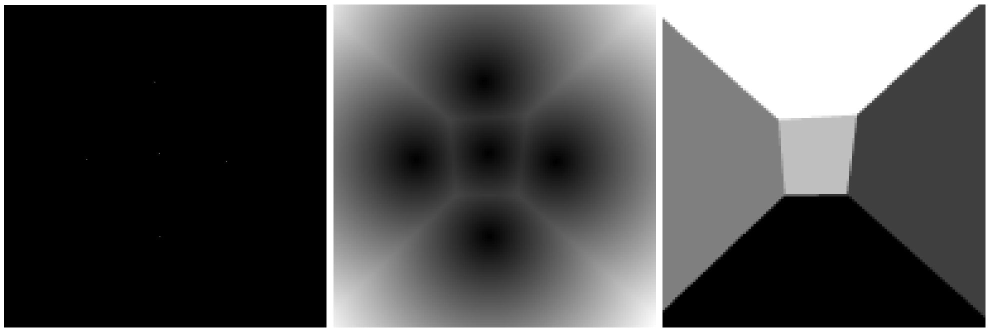


Figure 6.36: DanielssonDistanceMapImageFilter output. Set of pixels, distance map and Voronoi partition.

The input to the filter is taken from a reader and its output is passed to a itk::RescaleIntensityImageFilter and then to a writer.

```
filter->SetInput( reader->GetOutput() );
scaler->SetInput( filter->GetOutput() );
writer->SetInput( scaler->GetOutput() );
```

The type of input image has to be specified. In this case, a binary image is selected.

```
filter->InputIsBinaryOn();
```

Figure 6.36 illustrates the effect of this filter on a binary image with a set of points. The input image is shown at left, the distance map at the center and the Voronoi partition at right. This filter computes distance maps in N-dimensions and is therefore capable of producing $N-D$ Voronoi partitions.

The Voronoi map is obtained with the `GetVoronoiMap()` method. In the lines below we connect this output to the intensity rescaler and save the result in a file.

```
scaler->SetInput( filter->GetVoronoiMap() );
writer->SetFileName( voronoiMapFileName );
writer->Update();
```

The distance filter also produces an image of itk::Offset pixels representing the vectorial distance to the closest object in the scene. The type of this output image is defined by the VectorImageType trait of the filter type.

```
typedef FilterType::VectorImageType   OffsetImageType;
```

We can use this type for instantiating an `itk::ImageFileWriter` type and creating an object of this class in the following lines.

```
  typedef itk::ImageFileWriter< OffsetImageType >  WriterOffsetType;
  WriterOffsetType::Pointer offsetWriter = WriterOffsetType::New();
```

The output of the distance filter can be connected as input to the writer.

```
  offsetWriter->SetInput(  filter->GetVectorDistanceMap()  );
```

Execution of the writer is triggered by the invocation of the `Update()` method. Since this method can potentially throw exceptions it must be placed in a `try/catch` block.

```
  try
    {
    offsetWriter->Update();
    }
  catch( itk::ExceptionObject exp )
    {
    std::cerr << "Exception caught !" << std::endl;
    std::cerr <<     exp    << std::endl;
    }
```

Note that only the `itk::MetaImageIO` class supports reading and writing images of pixel type `itk::Offset`.

The source code for this section can be found in the file
`Examples/Filtering/SignedDanielssonDistanceMapImageFilter.cxx`.

This example illustrates the use of the `itk::SignedDanielssonDistanceMapImageFilter`. This filter generates a distance map by running Danielsson distance map twice, once on the input image and once on the flipped image.

The first step required to use this filter is to include its header file.

```
#include "itkSignedDanielssonDistanceMapImageFilter.h"
```

Then we must decide what pixel types to use for the input and output images. Since the output will contain distances measured in pixels, the pixel type should be able to represent at least the width of the image, or said in $N-D$ terms, the maximum extension along all the dimensions. The input and output image types are now defined using their respective pixel type and dimension.

```
  typedef  unsigned char   InputPixelType;
  typedef  float  OutputPixelType;
```

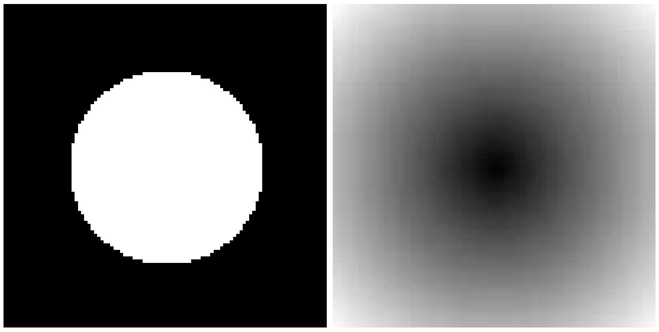


Figure 6.37: SignedDanielssonDistanceMapImageFilter applied on a binary circle image. The intensity has been rescaled for purposes of display.

```
const unsigned int Dimension = 2;

typedef itk::Image< InputPixelType,  Dimension >   InputImageType;
typedef itk::Image< OutputPixelType, Dimension >   OutputImageType;
```

The only change with respect to the previous example is to replace the DanielssonDistanceMapImageFilter with the SignedDanielssonDistanceMapImageFilter

```
typedef itk::SignedDanielssonDistanceMapImageFilter<
                                       InputImageType,
                                       OutputImageType >  FilterType;

FilterType::Pointer filter = FilterType::New();
```

The inside is considered as having negative distances. Outside is treated as having positive distances. To change the convention, use the InsideIsPositive(bool) function.

Figure 6.37 illustrates the effect of this filter. The input image and the distance map are shown.

6.9 Geometric Transformations

6.9.1 Filters You Should be Afraid to Use

6.9.2 Change Information Image Filter

This one is the scariest and more dangerous filter in the entire toolkit. You should not use this filter unless you are entirely certain that you know what you are doing. In fact if you decide to use this filter, you should write your code, then go for a long walk, get more coffee and ask yourself if you really needed to use this filter. If the answer is yes, then you should discuss this issue with someone you trust and get his/her opinion in writing. In general, if you need to use this filter, it means that you have a poor image provider that is putting your career at risk along with the life of any potential patient whose images you may end up processing.

6.9.3 Flip Image Filter

The source code for this section can be found in the file
`Examples/Filtering/FlipImageFilter.cxx`.

The `itk::FlipImageFilter` is used for flipping the image content in any of the coordinate axis. This filter must be used with **EXTREME** caution. You probably don't want to appear in the newspapers as the responsible of a surgery mistake in which a doctor extirpates the left kidney when it should have extracted the right one[3] . If that prospect doesn't scares you, maybe it is time for you to reconsider your career in medical image processing. Flipping effects that may seem innocuous at first view may still have dangerous consequences. For example flipping the cranio-caudal axis of a CT scans forces an observer to flip the left-right axis in order to make sense of the image.

The header file corresponding to this filter should be included first.

```
#include "itkFlipImageFilter.h"
```

Then the pixel types for input and output image must be defined and, with them, the image types can be instantiated.

```
  typedef   unsigned char  PixelType;

  typedef itk::Image< PixelType,  2 >   ImageType;
```

Using the image types it is now possible to instantiate the filter type and create the filter object.

```
  typedef itk::FlipImageFilter< ImageType >  FilterType;

  FilterType::Pointer filter = FilterType::New();
```

The axis to flip are specified in the form of an Array. In this case we take them from the command line arguments.

```
  typedef FilterType::FlipAxesArrayType     FlipAxesArrayType;
```

[3] *Wrong side* surgery accounts for 2% of the reported medical errors in the United States. Trivial... but equally dangerous.

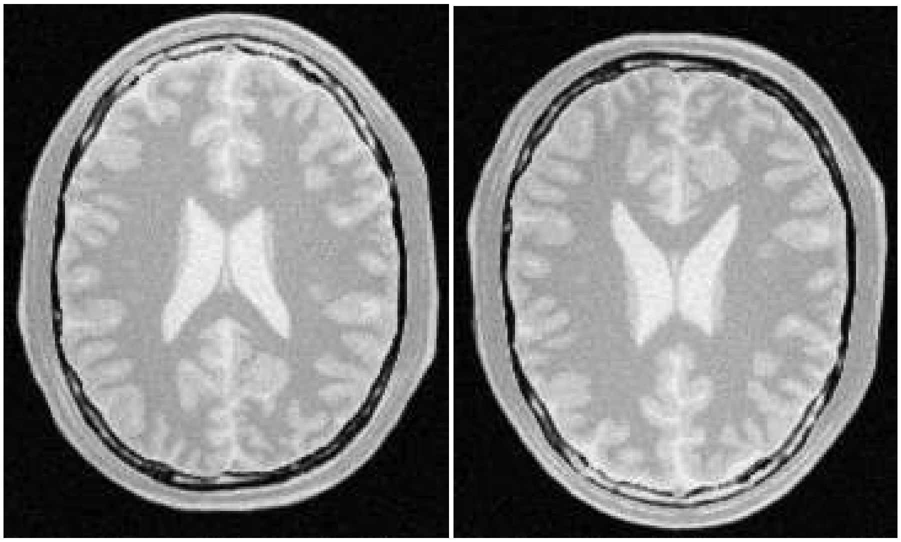


Figure 6.38: Effect of the FlipImageFilter on a slice from a MRI proton density brain image.

```
  FlipAxesArrayType flipArray;

  flipArray[0] = atoi( argv[3] );
  flipArray[1] = atoi( argv[4] );

  filter->SetFlipAxes( flipArray );
```

The input to the filter can be taken from any other filter, for example a reader. The output can be passed down the pipeline to other filters, for example, a writer. An update call on any downstream filter will trigger the execution of the mean filter.

```
  filter->SetInput( reader->GetOutput() );
  writer->SetInput( filter->GetOutput() );
  writer->Update();
```

Figure 6.38 illustrates the effect of this filter on a slice of MRI brain image using a flip array $[0,1]$ which means that the Y axis was flipped while the X axis was conserved.

6.9.4 Resample Image Filter

Introduction

The source code for this section can be found in the file
`Examples/Filtering/ResampleImageFilter.cxx`.

Resampling an image is a very important task in image analysis. It is especially important in the frame of image registration. The `itk::ResampleImageFilter` implements image resampling through the use of `itk::Transform`s. The inputs expected by this filter are an image, a transform and an interpolator. The space coordinates of the image are mapped through the transform in order to generate a new image. The extent and spacing of the resulting image are selected by the user. Resampling is performed in space coordinates, not pixel/grid coordinates. It is quite important to ensure that image spacing is properly set on the images involved. The interpolator is required since the mapping from one space to the other will often require evaluation of the intensity of the image at non-grid positions.

The header file corresponding to this filter should be included first.

```
#include "itkResampleImageFilter.h"
```

The header files corresponding to the transform and interpolator must also be included.

```
#include "itkAffineTransform.h"
#include "itkNearestNeighborInterpolateImageFunction.h"
```

The dimension and pixel types for input and output image must be defined and with them the image types can be instantiated.

```
  const     unsigned int   Dimension = 2;
  typedef   unsigned char  InputPixelType;
  typedef   unsigned char  OutputPixelType;
  typedef itk::Image< InputPixelType,  Dimension >   InputImageType;
  typedef itk::Image< OutputPixelType, Dimension >   OutputImageType;
```

Using the image and transform types it is now possible to instantiate the filter type and create the filter object.

```
  typedef itk::ResampleImageFilter<InputImageType,OutputImageType> FilterType;
  FilterType::Pointer filter = FilterType::New();
```

The transform type is typically defined using the image dimension and the type used for representing space coordinates.

```
  typedef itk::AffineTransform< double, Dimension >  TransformType;
```

An instance of the transform object is instantiated and passed to the resample filter. By default, the parameters of transform is set to represent the identity transform.

```
  TransformType::Pointer transform = TransformType::New();
  filter->SetTransform( transform );
```

The interpolator type is defined using the full image type and the type used for representing space coordinates.

```
  typedef itk::NearestNeighborInterpolateImageFunction<
                       InputImageType, double >  InterpolatorType;
```

An instance of the interpolator object is instantiated and passed to the resample filter.

```
  InterpolatorType::Pointer interpolator = InterpolatorType::New();
  filter->SetInterpolator( interpolator );
```

Given that some pixels of the output image may end up being mapped outside the extent of the input image it is necessary to decide what values to assign to them. This is done by invoking the `SetDefaultPixelValue()` method.

```
  filter->SetDefaultPixelValue( 0 );
```

The sampling grid of the output space is specified with the spacing along each dimension and the origin.

```
  double spacing[ Dimension ];
  spacing[0] = 1.0; // pixel spacing in millimeters along X
  spacing[1] = 1.0; // pixel spacing in millimeters along Y

  filter->SetOutputSpacing( spacing );

  double origin[ Dimension ];
  origin[0] = 0.0;  // X space coordinate of origin
  origin[1] = 0.0;  // Y space coordinate of origin

  filter->SetOutputOrigin( origin );
```

The extent of the sampling grid on the output image is defined by a `SizeType` and is set using the `SetSize()` method.

```
  InputImageType::SizeType   size;

  size[0] = 300;  // number of pixels along X
  size[1] = 300;  // number of pixels along Y

  filter->SetSize( size );
```

The input to the filter can be taken from any other filter, for example a reader. The output can be passed down the pipeline to other filters, for example a writer. An update call on any downstream filter will trigger the execution of the resampling filter.

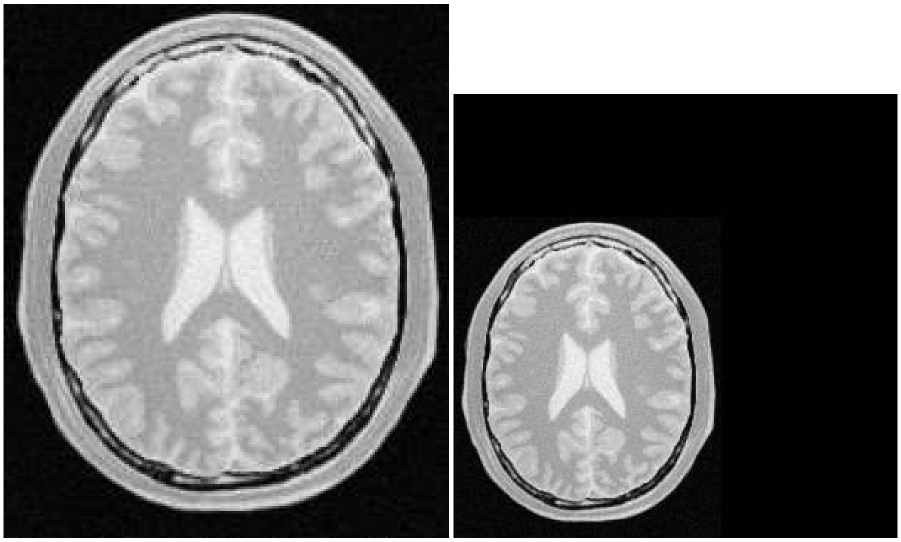


Figure 6.39: Effect of the resample filter.

```
filter->SetInput( reader->GetOutput() );
writer->SetInput( filter->GetOutput() );
writer->Update();
```

Figure 6.39 illustrates the effect of this filter on a slice of MRI brain image using an affine transform containing an identity transform. Note that any analysis of the behavior of this filter must be done on the space coordinate system in millimeters, not with respect to the sampling grid in pixels. The figure shows the resulting image in the lower left quarter of the extent. This may seem odd if analyzed in terms of the image grid but is quite clear when seen with respect to space coordinates. Figure 6.39 is particularly misleading because the images are rescaled to fit nicely on the text of this book. Figure 6.40 clarifies the situation. It shows the two same images placed on a equally scaled coordinate system. It becomes clear here that an identity transform is being used to map the image data, and that simply, we have requested to resample additional empty space around the image. The input image is 181×217 pixels in size and we have requested an output of 300×300 pixels. In this case, the input and output images both have spacing of $1mm \times 1mm$ and origin of $(0.0, 0.0)$.

Let's now set values on the transform. Note that the supplied transform represents the mapping of points from the output space to the input space. The following code sets up a translation.

```
TransformType::OutputVectorType translation;
translation[0] = -30;  // X translation in millimeters
translation[1] = -50;  // Y translation in millimeters
transform->Translate( translation );
```

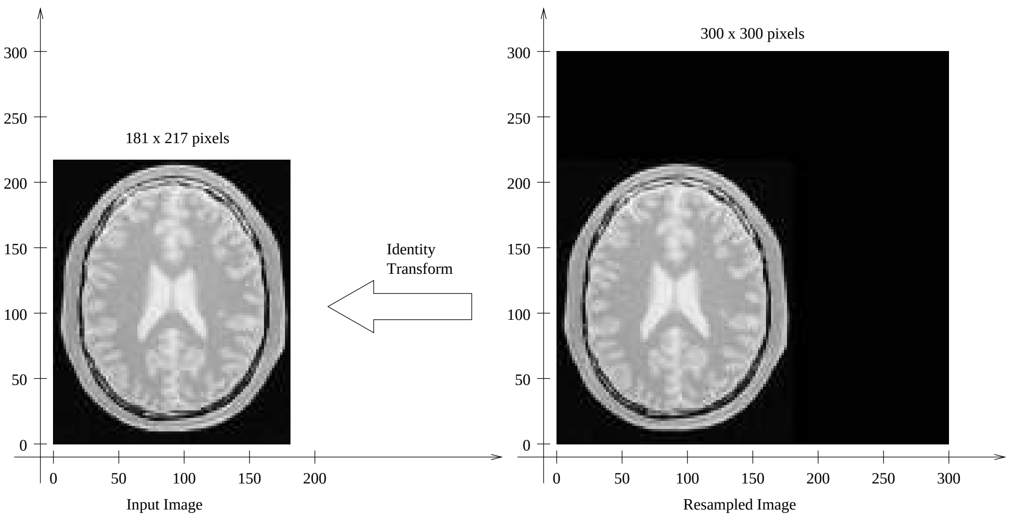


Figure 6.40: Analysis of the resample image done in a common coordinate system.

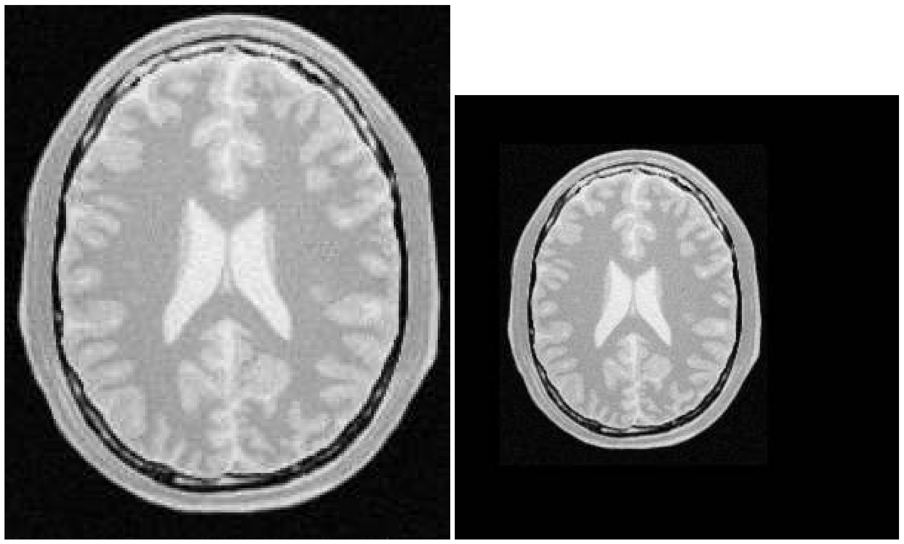


Figure 6.41: ResampleImageFilter with a translation by $(-30,-50)$.

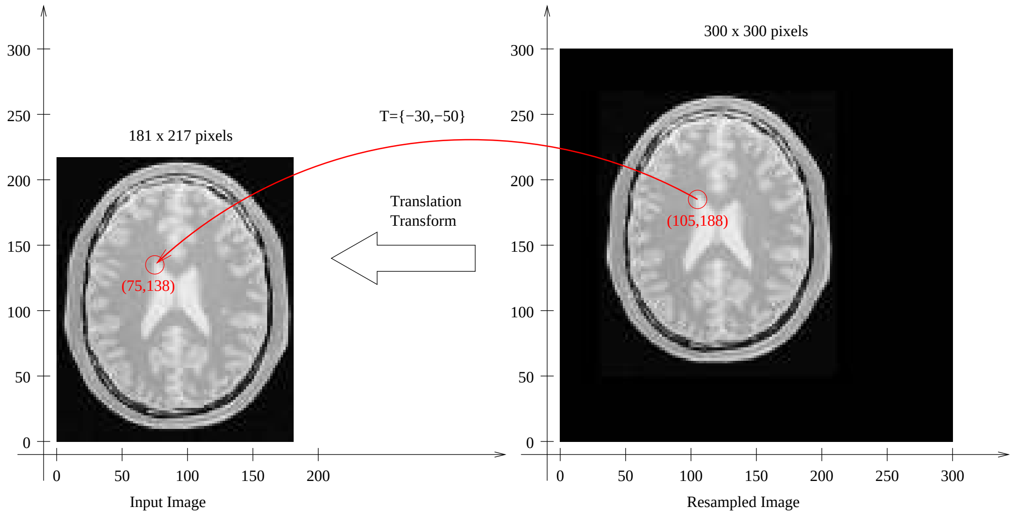


Figure 6.42: ResampleImageFilter. Analysis of a translation by $(-30,-50)$.

The output image resulting from the translation can be seen in Figure 6.41. Again, it is better to interpret the result in a common coordinate system as illustrated in Figure 6.42.

Probably the most important thing to keep in mind when resampling images is that the transform is used to map points from the **output** image space into the **input** image space. In this case, Figure 6.42 shows that the translation is applied to every point of the output image and the resulting position is used to read the intensity from the input image. In this way, the gray level of the point P in the output image is taken from the point $T(P)$ in the input image. Where T is the transformation. In the specific case of the Figure 6.42, the value of point $(105,188)$ in the output image is taken from the point $(75,138)$ of the input image because the transformation applied was a translation of $(-30,-50)$.

It is sometimes useful to intentionally set the default output value to a distinct gray value in order to highlight the mapping of the image borders. For example, the following code sets the default external value of 100. The result is shown in the right side of Figure 6.43

```
filter->SetDefaultPixelValue( 100 );
```

With this change we can better appreciate the effect of the previous translation transform on the image resampling. Figure 6.43 illustrates how the point $(30,50)$ of the output image gets its gray value from the point $(0,0)$ of the input image.

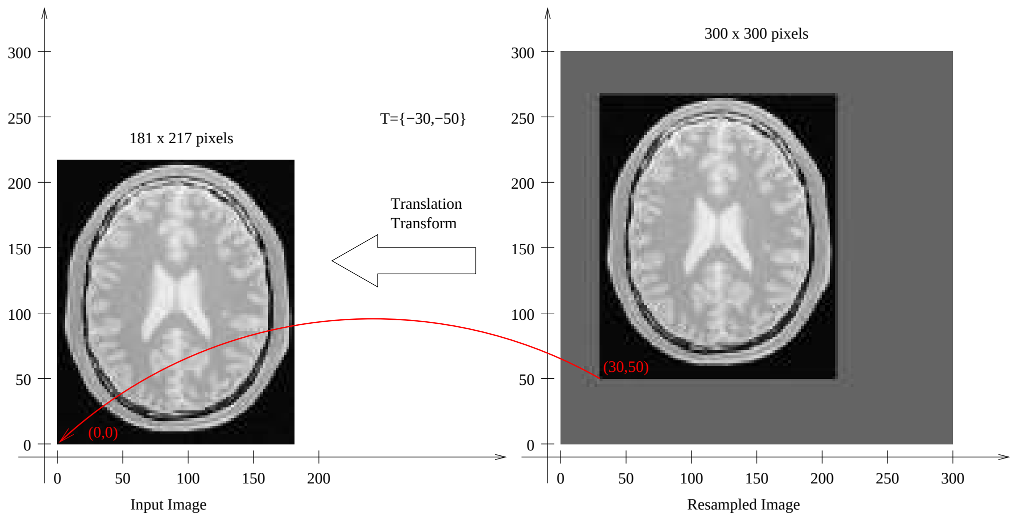


Figure 6.43: ResampleImageFilter highlighting image borders with SetDefaultPixelValue().

Importance of Spacing and Origin

The source code for this section can be found in the file
`Examples/Filtering/ResampleImageFilter2.cxx`.

During the computation of the resampled image all the pixels in the output region are visited. This visit is performed using `ImageIterators` which walk in the integer grid-space of the image. For each pixel, we need to convert grid position to space coordinates using the image spacing and origin.

For example, the pixel of index $I = (20,50)$ in an image of origin $O = (19.0,29.0)$ and pixel spacing $S = (1.3,1.5)$ corresponds to the spatial position

$$P[i] = I[i] \times S[i] + O[i] \tag{6.20}$$

which in this case leads to $P = (20 \times 1.3 + 19.0, 50 \times 1.5 + 29.0)$ and finally $P = (45.0,104.0)$

The space coordinates of P are mapped using the transform T supplied to the `itk::ResampleImageFilter` in order to map the point P to the input image space point $Q = T(P)$.

The whole process is illustrated in Figure 6.44. In order to correctly interpret the process of the ResampleImageFilter you should be aware of the origin and spacing settings of both the input and output images.

In order to facilitate the interpretation of the transform we set the default pixel value to a distinct from the image background.

```
filter->SetDefaultPixelValue( 50 );
```

Let's set up a uniform spacing for the output image.

```
double spacing[ Dimension ];
spacing[0] = 1.0; // pixel spacing in millimeters along X
spacing[1] = 1.0; // pixel spacing in millimeters along Y

filter->SetOutputSpacing( spacing );
```

Additionally, we will specify a non-zero origin. Note that the values provided here will be those of the space coordinates for the pixel of index $(0,0)$.

```
double origin[ Dimension ];
origin[0] = 30.0;  // X space coordinate of origin
origin[1] = 40.0;  // Y space coordinate of origin
filter->SetOutputOrigin( origin );
```

We set the transform to identity in order to better appreciate the effect of the origin selection.

```
transform->SetIdentity();
filter->SetTransform( transform );
```

The output resulting from these filter settings is analyzed in Figure 6.44

In the figure, the output image point with index $I = (0,0)$ has space coordinates $P = (30,40)$. The identity transform maps this point to $Q = (30,40)$ in the input image space. Because the input image in this case happens to have spacing $(1.0,1.0)$ and origin $(0.0,0.0)$, the physical point $Q = (30,40)$ maps to the pixel with index $I = (30,40)$.

The code for a different selection of origin and image size is illustrated below. The resulting output is presented in Figure 6.45

```
size[0] = 150;  // number of pixels along X
size[1] = 200;  // number of pixels along Y
filter->SetSize( size );


origin[0] = 60.0;  // X space coordinate of origin
origin[1] = 30.0;  // Y space coordinate of origin
filter->SetOutputOrigin( origin );
```

The output image point with index $I = (0,0)$ now has space coordinates $P = (60,30)$. The identity transform maps this point to $Q = (60,30)$ in the input image space. Because the input

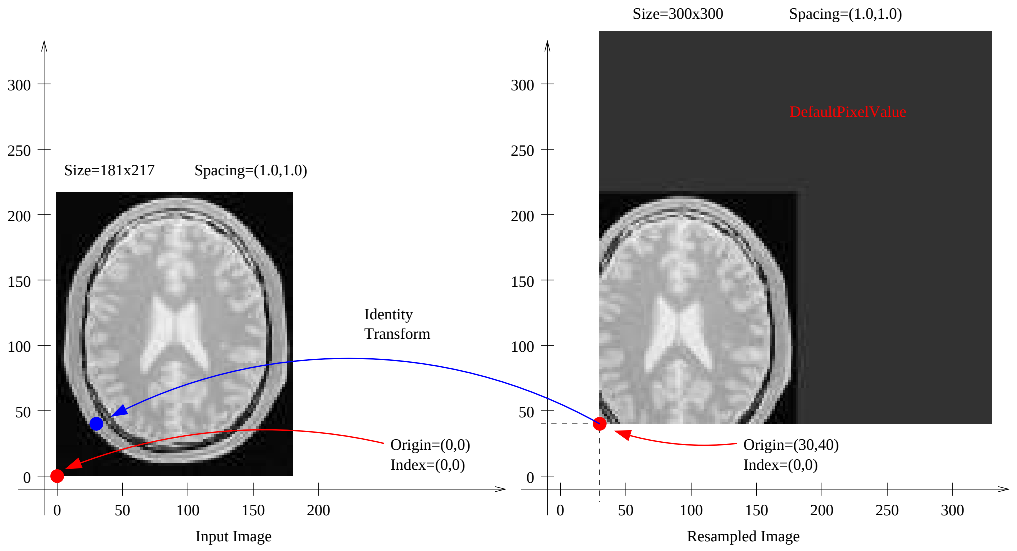


Figure 6.44: ResampleImageFilter selecting the origin of the output image.

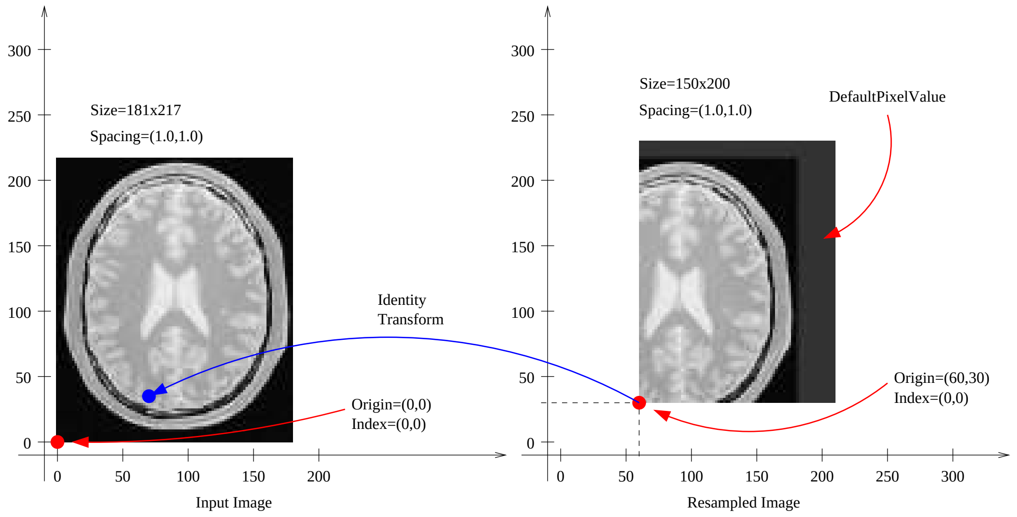


Figure 6.45: ResampleImageFilter selecting the origin of the output image.

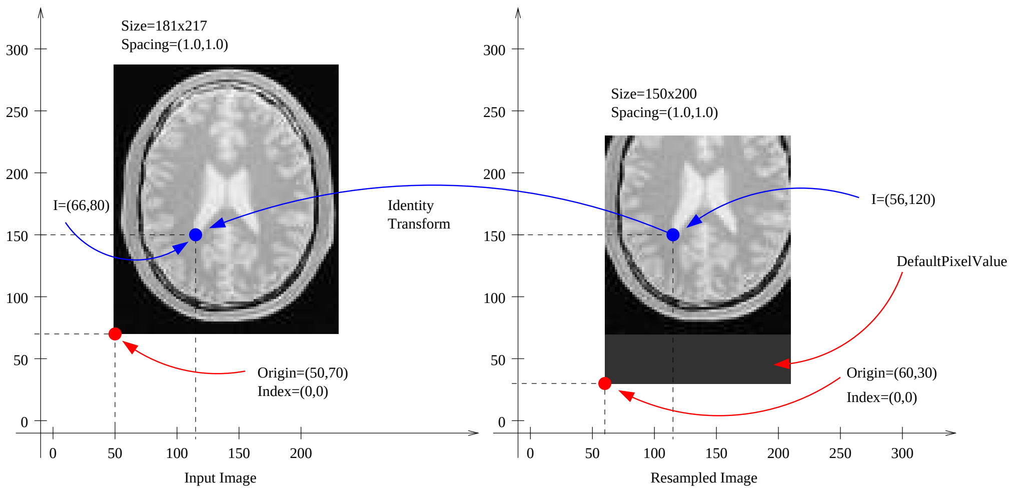


Figure 6.46: Effect of selecting the origin of the input image with ResampleImageFilter.

image in this case happens to have spacing $(1.0, 1.0)$ and origin $(0.0, 0.0)$, the physical point $Q = (60, 30)$ maps to the pixel with index $I = (60, 30)$.

Let's now analyze the effect of a non-zero origin in the input image. Keeping the output image settings of the previous example, we modify only the origin values on the file header of the input image. The new origin assigned to the input image is $O = (50, 70)$. An identity transform is still used as input for the ResampleImageFilter. The result of executing the filter with these parameters is presented in Figure 6.46

The pixel with index $I = (56, 120)$ on the output image has coordinates $P = (116, 150)$ in physical space. The identity transform maps P to the point $Q = (116, 150)$ on the input image space. The coordinates of Q are associated with the pixel of index $I = (66, 80)$ on the input image.

Now consider the effect of the output spacing on the process of image resampling. In order to simplify the analysis, let's put the origin back to zero in both the input and output images.

```
origin[0] = 0.0;  // X space coordinate of origin
origin[1] = 0.0;  // Y space coordinate of origin
filter->SetOutputOrigin( origin );
```

We then specify a non-unit spacing for the output image.

```
spacing[0] = 2.0; // pixel spacing in millimeters along X
spacing[1] = 3.0; // pixel spacing in millimeters along Y
filter->SetOutputSpacing( spacing );
```

Additionally, we reduce the output image extent, since the new pixels are now covering a larger area of $2.0\text{mm} \times 3.0\text{mm}$.

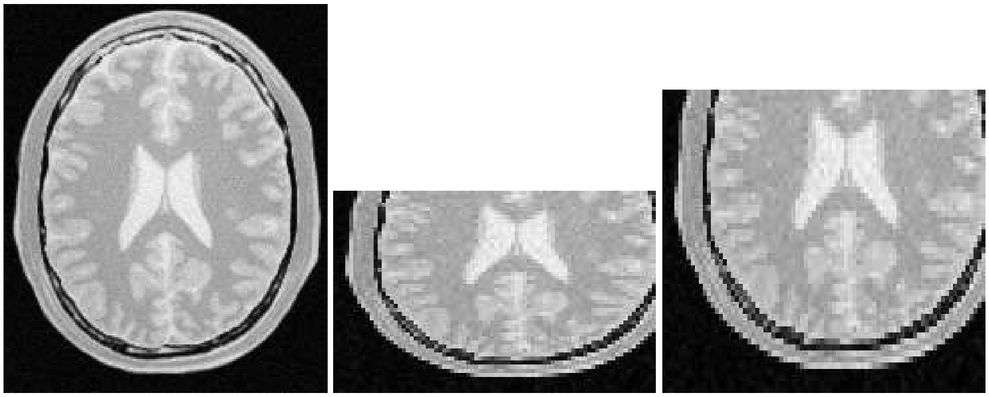


Figure 6.47: Resampling with different spacing seen by a naive viewer (center) and a correct viewer (right), input image (left).

```
size[0] = 80;  // number of pixels along X
size[1] = 50;  // number of pixels along Y
filter->SetSize( size );
```

With these new parameters the physical extent of the output image is 160 millimeters by 150 millimeters.

Before attempting to analyze the effect of the resampling image filter it is important to make sure that the image viewer used to display the input and output images take the spacing into account and use it to appropriately scale the images on the screen. Please note that images in formats like PNG are not capable of representing origin and spacing. The toolkit assume trivial default values for them. Figure 6.47 (center) illustrates the effect of using a naive viewer that does not take pixel spacing into account. A correct display is presented at the right in the same figure[4].

The filter output is analyzed in a common coordinate system with the input from Figure 6.48. In this figure, pixel $I = (33,27)$ of the output image is located at coordinates $P = (66.0,81.0)$ of the physical space. The identity transform maps this point to $Q = (66.0,81.0)$ in the input image physical space. The point Q is then associated to the pixel of index $I = (66,81)$ on the input image, because this image has zero origin and unit spacing.

The input image spacing is also an important factor in the process of resampling an image. The following example illustrates the effect of non-unit pixel spacing on the input image. An input image similar to the those used in Figures 6.44 to 6.48 has been resampled to have pixel spacing of 2mm $\times$ 3mm. The input image is presented in Figure 6.49 as viewed with a naive image viewer (left) and with a correct image viewer (right).

The following code is used to transform this non-unit spacing input image into another non-unit

[4]A viewer is provided with ITK under the name of MetaImageViewer. This viewer takes into account pixel spacing.

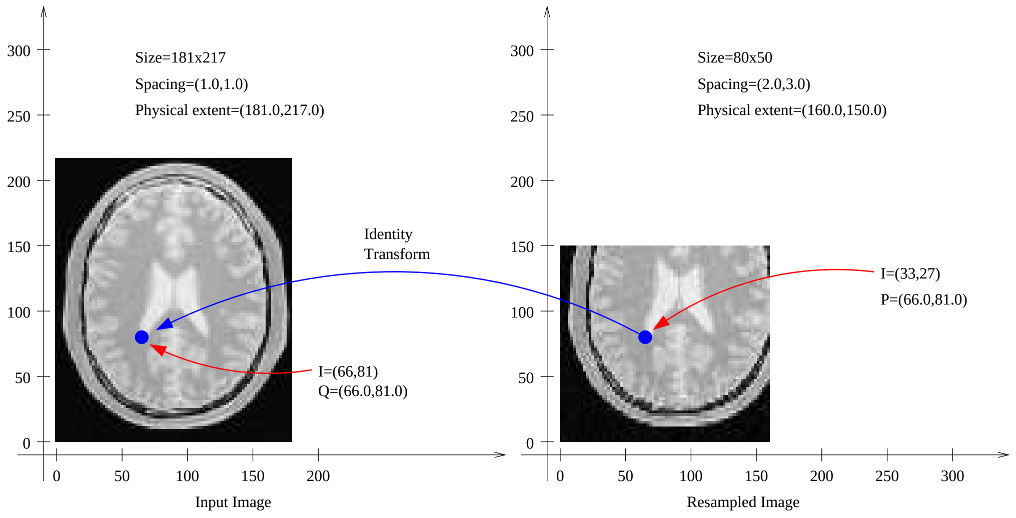


Figure 6.48: Effect of selecting the spacing on the output image.

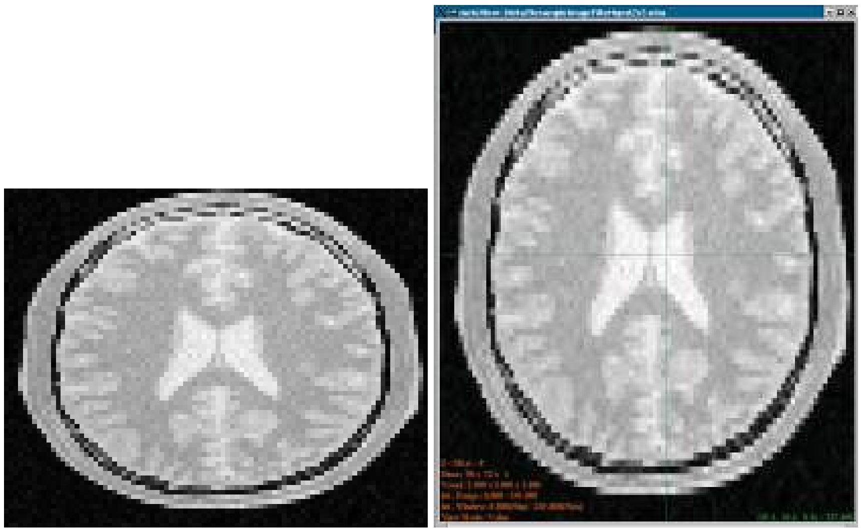


Figure 6.49: Input image with 2×3mm spacing as seen with a naive viewer (left) and a correct viewer (right).

spacing image located at a non-zero origin. The comparison between input and output in a common reference system is presented in figure 6.50.

Here we start by selecting the origin of the output image.

```
origin[0] = 25.0;  // X space coordinate of origin
origin[1] = 35.0;  // Y space coordinate of origin
filter->SetOutputOrigin( origin );
```

We then select the number of pixels along each dimension.

```
size[0] = 40;  // number of pixels along X
size[1] = 45;  // number of pixels along Y
filter->SetSize( size );
```

Finally, we set the output pixel spacing.

```
spacing[0] = 4.0; // pixel spacing in millimeters along X
spacing[1] = 4.5; // pixel spacing in millimeters along Y
filter->SetOutputSpacing( spacing );
```

Figure 6.50 shows the analysis of the filter output under these conditions. First, notice that the origin of the output image corresponds to the settings $O = (25.0, 35.0)$ millimeters, spacing $(4.0, 4.5)$ millimeters and size $(40, 45)$ pixels. With these parameters the pixel of index $I = (10, 10)$ in the output image is associated with the spatial point of coordinates $P = (10 \times 4.0 + 25.0, 10 \times 4.5 + 35.0)) = (65.0, 80.0)$. This point is mapped by the transform—identity in this particular case—to the point $Q = (65.0, 80.0)$ in the input image space. The point Q is then associated with the pixel of index $I = ((65.0 - 0.0)/2.0 - (80.0 - 0.0)/3.0) = (32.5, 26.6)$. Note that the index does not fall on grid position, for this reason the value to be assigned to the output pixel is computed by interpolating values on the input image around the non-integer index $I = (32.5, 26.6)$.

Note also that the discretization of the image is more visible on the output presented on the right side of Figure 6.50 due to the choice of a low resolution—just 40×45 pixels.

A Complete Example

The source code for this section can be found in the file
`Examples/Filtering/ResampleImageFilter3.cxx`.

Previous examples have described the basic principles behind the `itk::ResampleImageFilter`. Now it's time to have some fun with it.

Figure 6.52 illustrates the general case of the resampling process. The origin and spacing of the output image has been selected to be different from those of the input image. The circles

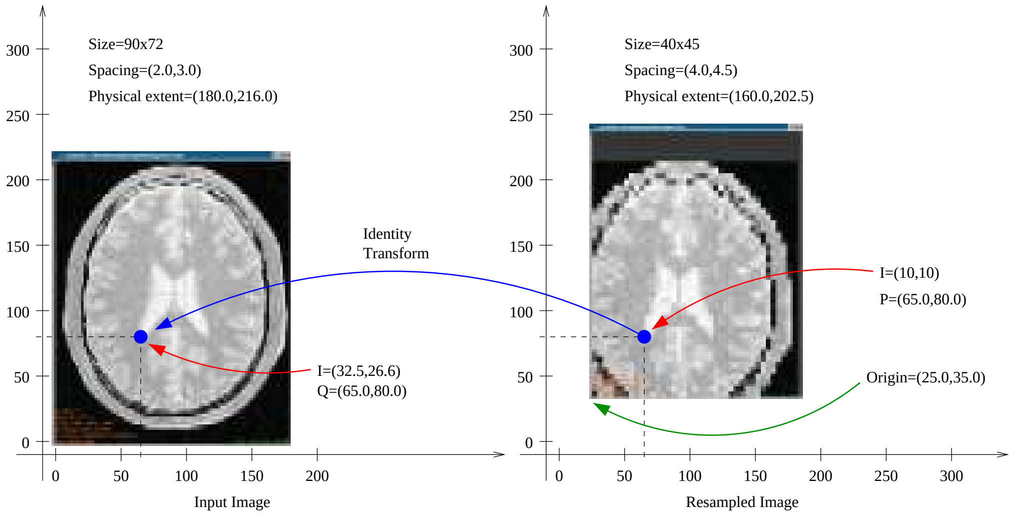


Figure 6.50: Effect of non-unit spacing on the input and output images.

represent the *center* of pixels. They are inscribed in a rectangle representing the *coverage* of this pixel. The spacing specifies the distance between pixel centers along every dimension.

The transform applied is a rotation of 30 degrees. It is important to note here that the transform supplied to the `itk::ResampleImageFilter` is a *clockwise* rotation. This transform rotates the *coordinate system* of the output image 30 degrees clockwise. When the two images are relocated in a common coordinate system—as in Figure 6.52—the result is that the frame of the output image appears rotated 30 degrees *clockwise*. If the output image is seen with its coordinate system vertically aligned—as in Figure 6.51—the image content appears rotated 30 degrees *counter-clockwise*. Before continuing to read this section, you may want to meditate a bit on this fact while enjoying a cup of (Columbian) coffee.

The following code implements the conditions illustrated in Figure 6.52 with the only difference of the output spacing being 40 times smaller and a number of pixels 40 times larger in both dimensions. Without these changes, few detail will be recognizable on the images. Note that the spacing and origin of the input image should be prepared in advance by using other means since this filter cannot alter in any way the actual content of the input image.

In order to facilitate the interpretation of the transform we set the default pixel value to value distinct from the image background.

```
filter->SetDefaultPixelValue( 100 );
```

The spacing is selected here to be 40 times smaller than the one illustrated in Figure 6.52.

```
double spacing[ Dimension ];
```

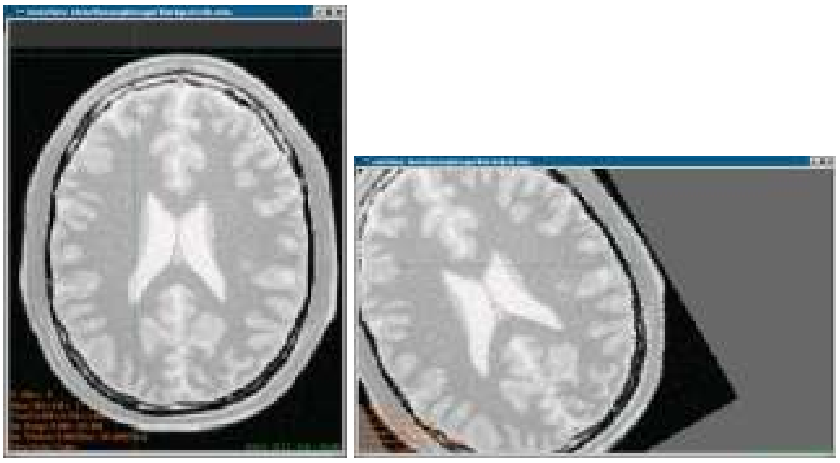


Figure 6.51: Effect of a rotation on the resampling filter. Input image at left, output image at right.

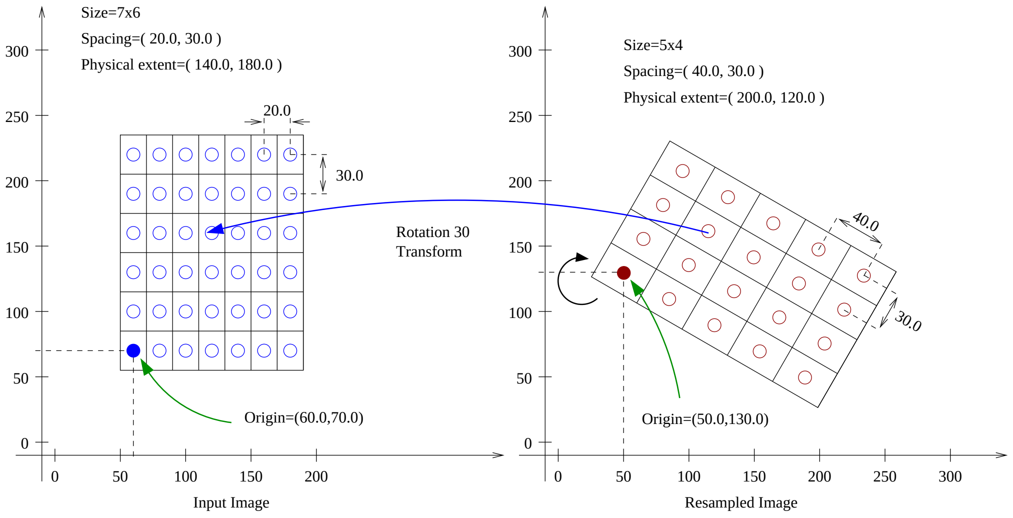


Figure 6.52: Input and output image placed in a common reference system.

```
spacing[0] = 40.0 / 40.0; // pixel spacing in millimeters along X
spacing[1] = 30.0 / 40.0; // pixel spacing in millimeters along Y
filter->SetOutputSpacing( spacing );
```

Let us now set up the origin of the output image. Note that the values provided here will be those of the space coordinates for the output image pixel of index $(0,0)$.

```
double origin[ Dimension ];
origin[0] =  50.0;  // X space coordinate of origin
origin[1] = 130.0;  // Y space coordinate of origin
filter->SetOutputOrigin( origin );
```

The output image size is defined to be 40 times the one illustrated on the Figure 6.52.

```
InputImageType::SizeType   size;
size[0] = 5 * 40;  // number of pixels along X
size[1] = 4 * 40;  // number of pixels along Y
filter->SetSize( size );
```

Rotations are performed around the origin of physical coordinates—not the image origin nor the image center. Hence, the process of positioning the output image frame as it is shown in Figure 6.52 requires three steps. First, the image origin must be moved to the origin of the coordinate system, this is done by applying a translation equal to the negative values of the image origin.

```
TransformType::OutputVectorType translation1;
translation1[0] =   -origin[0];
translation1[1] =   -origin[1];
transform->Translate( translation1 );
```

In a second step, a rotation of 30 degrees is performed. In the `itk::AffineTransform`, angles are specified in *radians*. Also, a second boolean argument is used to specify if the current modification of the transform should be pre-composed or post-composed with the current transform content. In this case the argument is set to `false` to indicate that the rotation should be applied *after* the current transform content.

```
const double degreesToRadians = atan(1.0) / 45.0;
transform->Rotate2D( -30.0 * degreesToRadians, false );
```

The third and final step implies translating the image origin back to its previous location. This is be done by applying a translation equal to the origin values.

```
TransformType::OutputVectorType translation2;
translation2[0] =   origin[0];
translation2[1] =   origin[1];
transform->Translate( translation2, false );
filter->SetTransform( transform );
```

Figure 6.51 presents the actual input and output images of this example as shown by a correct viewer which takes spacing into account. Note the *clockwise* versus *counter-clockwise* effect discussed previously between the representation in Figure 6.52 and Figure 6.51.

As a final exercise, let's track the mapping of an individual pixel. Keep in mind that the transformation is initiated by walking through the pixels of the *output* image. This is the only way to ensure that the image will be generated without holes or redundant values. When you think about transformation it is always useful to analyze things from the output image towards the input image.

Let's take the pixel with index $I = (1,2)$ from the output image. The physical coordinates of this point in the output image reference system are $P = (1 \times 40.0 + 50.0, 2 \times 30.0 + 130.0) = (90.0, 190.0)$ millimeters.

This point P is now mapped through the `itk::AffineTransform` into the input image space. The operation requires to subtract the origin, apply a 30 degrees rotation and add the origin back. Let's follow those steps. Subtracting the origin from P leads to $P1 = (40.0, 60.0)$, the rotation maps $P1$ to $P2 = (40.0 \times cos(30.0) + 60.0 \times sin(30.0), 40.0 \times sin(30.0) - 60.0 \times cos(30.0)) = (64.64, 31.96)$. Finally this point is translated back by the amount of the image origin. This moves $P2$ to $P3 = (114.64, 161.96)$.

The point $P3$ is now in the coordinate system of the input image. The pixel of the input image associated with this physical position is computed using the origin and spacing of the input image. $I = ((114.64 - 60.0)/20.0, (161 - 70.0)/30.0)$ which results in $I = (2.7, 3.0)$. Note that this is a non-grid position since the values are non-integers. This means that the gray value to be assigned to the output image pixel $I = (1,2)$ must be computed by interpolation of the input image values.

In this particular code the interpolator used is simply a `itk::NearestNeighborInterpolateImageFunction` which will assign the value of the closest pixel. This ends up being the pixel of index $I = (3,3)$ and can be seen from Figure 6.52.

Rotating an Image

The source code for this section can be found in the file
`Examples/Filtering/ResampleImageFilter4.cxx`.

The following example illustrates how to rotate an image around its center. In this particular case an `itk::AffineTransform` is used to map the input space into the output space.

The header of the affine transform is included below.

```
#include "itkAffineTransform.h"
```

The transform type is instantiated using the coordinate representation type and the space dimension. Then a transform object is constructed with the New() method and passed to a

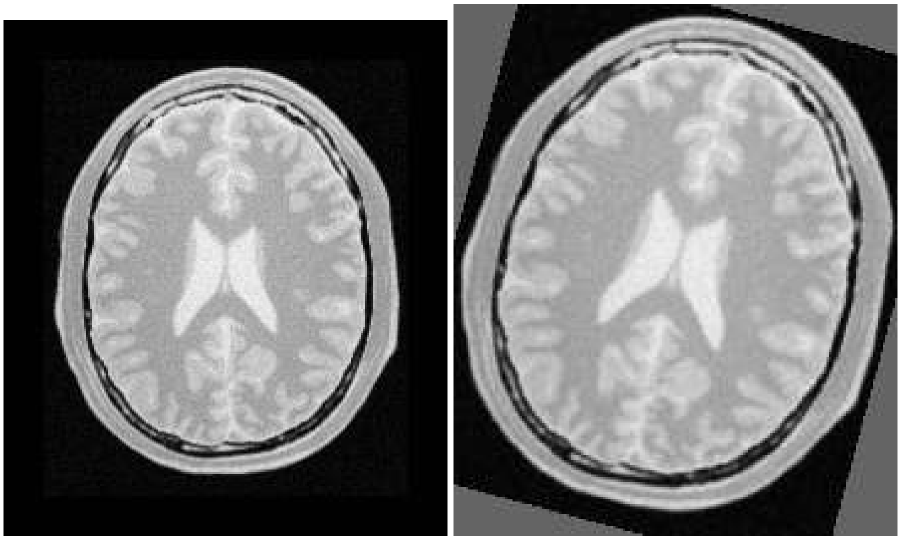


Figure 6.53: Effect of the resample filter rotating an image.

itk::SmartPointer.

```
  typedef itk::AffineTransform< double, Dimension >  TransformType;
  TransformType::Pointer transform = TransformType::New();
```

The parameters of the output image are taken from the input image.

```
  reader->Update();
  const InputImageType::SpacingType&
    spacing = reader->GetOutput()->GetSpacing();
  const InputImageType::PointType&
    origin  = reader->GetOutput()->GetOrigin();
  InputImageType::SizeType size =
      reader->GetOutput()->GetLargestPossibleRegion().GetSize();
  filter->SetOutputOrigin( origin );
  filter->SetOutputSpacing( spacing );
  filter->SetSize( size );
```

Rotations are performed around the origin of physical coordinates—not the image origin nor the image center. Hence, the process of positioning the output image frame as it is shown in Figure 6.53 requires three steps. First, the image origin must be moved to the origin of the coordinate system, this is done by applying a translation equal to the negative values of the image origin.

```
  TransformType::OutputVectorType translation1;
```

```
const double imageCenterX = origin[0] + spacing[0] * size[0] / 2.0;
const double imageCenterY = origin[1] + spacing[1] * size[1] / 2.0;

translation1[0] =   -imageCenterX;
translation1[1] =   -imageCenterY;

transform->Translate( translation1 );
```

In a second step, the rotation is specified using the method `Rotate2D()`.

```
const double degreesToRadians = atan(1.0) / 45.0;
const double angle = angleInDegrees * degreesToRadians;
transform->Rotate2D( -angle, false );
```

The third and final step requires translating the image origin back to its previous location. This is be done by applying a translation equal to the origin values.

```
TransformType::OutputVectorType translation2;
translation2[0] =   imageCenterX;
translation2[1] =   imageCenterY;
transform->Translate( translation2, false );
filter->SetTransform( transform );
```

The output of the resampling filter is connected to a writer and the execution of the pipeline is triggered by a writer update.

```
try
  {
  writer->Update();
  }
catch( itk::ExceptionObject & excep )
  {
  std::cerr << "Exception caught !" << std::endl;
  std::cerr << excep << std::endl;
  }
```

Rotating and Scaling an Image

The source code for this section can be found in the file
`Examples/Filtering/ResampleImageFilter5.cxx`.

This example illustrates the use of the `itk::Similarity2DTransform`. A similarity transform involves rotation, translation and scaling. Since the parameterization of rotations is difficult to get in a generic *ND* case, a particular implementation is available for 2*D*.

The header file of the transform is included below.

```
#include "itkSimilarity2DTransform.h"
```

The transform type is instantiated using the coordinate representation type as the single template parameter.

```
  typedef itk::Similarity2DTransform< double >  TransformType;
```

A transform object is constructed by calling `New()` and passing the result to a `itk::SmartPointer`.

```
  TransformType::Pointer transform = TransformType::New();
```

The parameters of the output image are taken from the input image.

The Similarity2DTransform allows the user to select the center of rotation. This center is used for both rotation and scaling operations.

```
  TransformType::InputPointType rotationCenter;
  rotationCenter[0] = origin[0] + spacing[0] * size[0] / 2.0;
  rotationCenter[1] = origin[1] + spacing[1] * size[1] / 2.0;
  transform->SetCenter( rotationCenter );
```

The rotation is specified with the method `SetAngle()`.

```
  const double degreesToRadians = atan(1.0) / 45.0;
  const double angle = angleInDegrees * degreesToRadians;
  transform->SetAngle( angle );
```

The scale change is defined using the method `SetScale()`.

```
  transform->SetScale( scale );
```

A translation to be applied after the rotation and scaling can be specified with the method `SetTranslation()`.

```
  TransformType::OutputVectorType translation;

  translation[0] =   13.0;
  translation[1] =   17.0;

  transform->SetTranslation( translation );

  filter->SetTransform( transform );
```

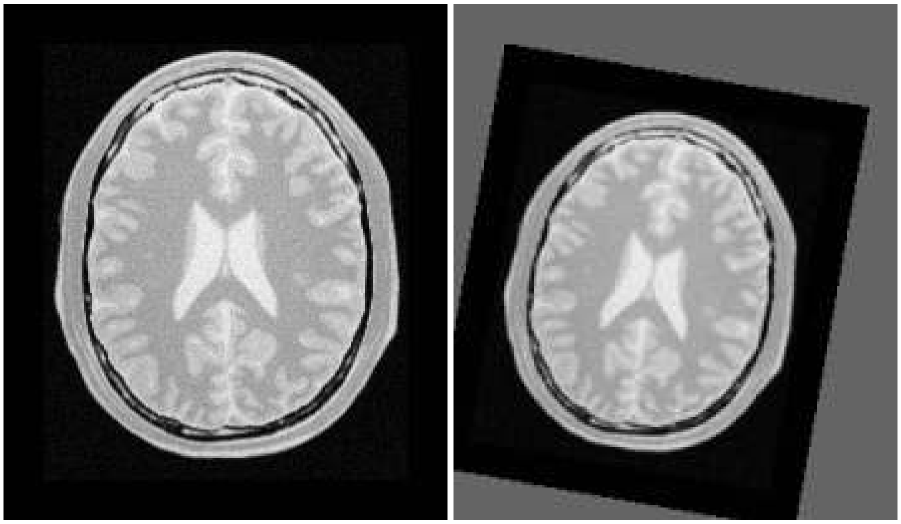


Figure 6.54: Effect of the resample filter rotating and scaling an image.

Note that the order in which rotation, scaling and translation are defined is irrelevant in this transform. This is not the case in the Affine transform which is very generic and allow different combinations for initialization. In the Similarity2DTransform class the rotation and scaling will always be applied before the translation.

Figure 6.54 shows the effect of this rotation, translation and scaling on a slice of a brain MRI. The scale applied for producing this figure was 1.2 and the rotation angle was 10°.

Resampling using a deformation field

The source code for this section can be found in the file
`Examples/Filtering/WarpImageFilter1.cxx`.

This example illustrates how to use the WarpImageFilter and a deformation field for resampling an image. This is typically done as the last step of a deformable registration algorithm.

```
#include "itkWarpImageFilter.h"
#include "itkLinearInterpolateImageFunction.h"
```

The deformation field is represented as an image of vector pixel types. The dimension of the vectors is the same as the dimension of the input image. Each vector in the deformation field represents the distance between a geometric point in the input space and a point in the output space such that:

$$p_{in} = p_{out} + distance \tag{6.21}$$

```
typedef   float VectorComponentType;
typedef   itk::Vector< VectorComponentType, Dimension > VectorPixelType;
typedef   itk::Image< VectorPixelType,  Dimension >   DeformationFieldType;

typedef   unsigned char  PixelType;
typedef   itk::Image< PixelType,  Dimension >   ImageType;
```

The field is read from a file, through a reader instantiated over the vector pixel types.

```
typedef   itk::ImageFileReader< DeformationFieldType >  FieldReaderType;


FieldReaderType::Pointer fieldReader = FieldReaderType::New();
fieldReader->SetFileName( argv[2] );
fieldReader->Update();

DeformationFieldType::ConstPointer deformationField = fieldReader->GetOutput();
```

The itk::WarpImageFilter is templated over the input image type, output image type and the deformation field type.

```
typedef itk::WarpImageFilter< ImageType,
                              ImageType,
                              DeformationFieldType  >  FilterType;

FilterType::Pointer filter = FilterType::New();
```

Typically the mapped position does not correspond to an integer pixel position in the input image. Interpolation via an image function is used to compute values at non-integer positions. This is done via the `SetInterpolator()` method.

```
typedef itk::LinearInterpolateImageFunction<
                     ImageType, double >  InterpolatorType;

InterpolatorType::Pointer interpolator = InterpolatorType::New();

filter->SetInterpolator( interpolator );
```

The output image spacing and origin may be set via SetOutputSpacing(), SetOutputOrigin(). This is taken from the deformation field.

```
filter->SetOutputSpacing( deformationField->GetSpacing() );
filter->SetOutputOrigin(  deformationField->GetOrigin() );

filter->SetDeformationField( deformationField );
```

Subsampling and image in the same space

The source code for this section can be found in the file
`Examples/Filtering/SubsampleVolume.cxx`.

This example illustrates how to perform subsampling of a volume using ITK classes. In order to avoid aliasing artifacts, the volume must be processed by a low-pass filter before resampling. Here we use the `itk::RecursiveGaussianImageFilter` as low-pass filter. The image is then resampled by using three different factors, one per dimension of the image.

The most important headers to include here are the ones corresponding to the resampling image filter, the transform, the interpolator and the smoothing filter.

```
#include "itkResampleImageFilter.h"
#include "itkIdentityTransform.h"
#include "itkLinearInterpolateImageFunction.h"
#include "itkRecursiveGaussianImageFilter.h"
```

We explicitly instantiate the pixel type and dimension of the input image, and the images that will be used internally for computing the resampling.

```
  const     unsigned int    Dimension = 3;

  typedef   unsigned char   InputPixelType;

  typedef   float           InternalPixelType;
  typedef   unsigned char   OutputPixelType;

  typedef itk::Image< InputPixelType,    Dimension >   InputImageType;
  typedef itk::Image< InternalPixelType, Dimension >   InternalImageType;
  typedef itk::Image< OutputPixelType,   Dimension >   OutputImageType;
```

In this particular case we take the factors for resampling directly from the command line arguments.

```
  const double factorX = atof( argv[3] );
  const double factorY = atof( argv[4] );
  const double factorZ = atof( argv[5] );
```

A casting filter is instantiated in order to convert the pixel type of the input image into the pixel type desired for computing the resampling.

```
  typedef itk::CastImageFilter< InputImageType,
                                InternalImageType >   CastFilterType;
```

```
CastFilterType::Pointer caster = CastFilterType::New();

caster->SetInput( inputImage );
```

The smoothing filter of choice is the `RecursiveGaussianImageFilter`. We create three of them in order to have the freedom of performing smoothing with different Sigma values along each dimension.

```
typedef itk::RecursiveGaussianImageFilter<
                                InternalImageType,
                                InternalImageType > GaussianFilterType;

GaussianFilterType::Pointer smootherX = GaussianFilterType::New();
GaussianFilterType::Pointer smootherY = GaussianFilterType::New();
GaussianFilterType::Pointer smootherZ = GaussianFilterType::New();
```

The smoothing filters are connected in a cascade in the pipeline.

```
smootherX->SetInput( caster->GetOutput() );
smootherY->SetInput( smootherX->GetOutput() );
smootherZ->SetInput( smootherY->GetOutput() );
```

The Sigma values to use in the smoothing filters is computed based on the pixel spacings of the input image and the factors provided as arguments.

```
const InputImageType::SpacingType& inputSpacing = inputImage->GetSpacing();

const double sigmaX = inputSpacing[0] * factorX;
const double sigmaY = inputSpacing[1] * factorY;
const double sigmaZ = inputSpacing[2] * factorZ;

smootherX->SetSigma( sigmaX );
smootherY->SetSigma( sigmaY );
smootherZ->SetSigma( sigmaZ );
```

We instruct each one of the smoothing filters to act along a particular direction of the image, and set them to use normalization across scale space in order to prevent for the reduction of intensity that accompanies the diffusion process associated with the Gaussian smoothing.

```
smootherX->SetDirection( 0 );
smootherY->SetDirection( 1 );
smootherZ->SetDirection( 2 );

smootherX->SetNormalizeAcrossScale( false );
smootherY->SetNormalizeAcrossScale( false );
smootherZ->SetNormalizeAcrossScale( false );
```

The type of the resampling filter is instantiated using the internal image type and the output image type.

```
typedef itk::ResampleImageFilter<
                InternalImageType, OutputImageType >  ResampleFilterType;

ResampleFilterType::Pointer resampler = ResampleFilterType::New();
```

Since the resampling is performed in the same physical extent of the input image, we select the IdentityTransform as the one to be used by the resampling filter.

```
typedef itk::IdentityTransform< double, Dimension >  TransformType;

TransformType::Pointer transform = TransformType::New();
transform->SetIdentity();
resampler->SetTransform( transform );
```

The Linear interpolator is selected given that it provides a good run-time performance. For applications that require better precision you may want to replace this interpolator with the itk::BSplineInterpolateImageFunction interpolator or with the itk::WindowedSincInterpolateImageFunction interpolator.

```
typedef itk::LinearInterpolateImageFunction<
                                 InternalImageType, double >  InterpolatorType;

InterpolatorType::Pointer interpolator = InterpolatorType::New();

resampler->SetInterpolator( interpolator );
```

The spacing to be used in the grid of the resampled image is computed using the input image spacing and the factors provided in the command line arguments.

```
OutputImageType::SpacingType spacing;

spacing[0] = inputSpacing[0] * factorX;
spacing[1] = inputSpacing[1] * factorY;
spacing[2] = inputSpacing[2] * factorZ;

resampler->SetOutputSpacing( spacing );
```

The origin of the input image is preserved and passed to the output image.

```
resampler->SetOutputOrigin( inputImage->GetOrigin() );
```

The number of pixels to use along each direction on the grid of the resampled image is computed using the number of pixels in the input image and the sampling factors.

```
InputImageType::SizeType   inputSize =
            inputImage->GetLargestPossibleRegion().GetSize();

typedef InputImageType::SizeType::SizeValueType SizeValueType;

InputImageType::SizeType   size;

size[0] = static_cast< SizeValueType >( inputSize[0] / factorX );
size[1] = static_cast< SizeValueType >( inputSize[1] / factorY );
size[2] = static_cast< SizeValueType >( inputSize[2] / factorZ );

resampler->SetSize( size );
```

Finally, the input to the resampler is taken from the output of the smoothing filter.

```
resampler->SetInput( smootherZ->GetOutput() );
```

At this point we can trigger the execution of the resampling by calling the `Update()` method, or we can chose to pass the output of the resampling filter to another section of pipeline, for example, an image writer.

Resampling an Anisotropic image to make it Isotropic

The source code for this section can be found in the file
`Examples/Filtering/ResampleVolumesToBeIsotropic.cxx`.

It is unfortunate that it is still very common to find medical image datasets that have been acquired with large inter-sclice spacings that result in voxels with anisotropic shapes. In many cases these voxels have ratios of $[1:5]$ or even $[1:10]$ between the resolution in the plane (x,y) and the resolution along the z axis. Such dataset are close to **useless** for the purpose of computer assisted image analysis. The persistent tendency for acquiring dataset in such formats just reveals how small is the understanding of the third dimension that have been gained in the clinical settings and in many radiology reading rooms. Datasets that are acquired with such large anisotropies bring with them the retrograde message: *"I do not think 3D is informative"*. They repeat stubbornly that: *"all that you need to know, can be known by looking at individual slices, one by one"*. However, the fallacy of such statement is made evident with the simple act of looking at the slices when reconstructed in any of the ortogonal planes. The ugliness of the extreme rectangular pixel shapes becomes obvious, along with the clear technical realization that no decent signal processing or algorithms can be performed in such images.

Image analysts have a long educational battle to fight in the radiological setting in order to bring the message that 3D datasets acquired with anisotropies larger than $[1:2]$ are simply dismissive of the most fundamental concept of digital signal processing: The Shannon Sampling Theorem [75, 76].

Facing the inertia of many clinical imaging departments and their insistence that these images should be good enough for image processing, some image analysts have stoically tried to deal with these poor datasets. These image analysts usually proceed to subsample the high in-plane resolution and to super-sample the inter-slice resolution with the purpose of faking the type of dataset that they should have received in the first place: an **isotropic** dataset. This example is an illustration of how such operation can be performed using the filter available in the Insight Toolkit.

Note that this example is not presented here as a *solution* to the problem of anisotropic datasets. On the contrary, this is simply a *dangerous palliative* that will help to perpetuate the mistake of the image acquisition departments. This code is just an analgesic that will make you believe that you don't have pain, while a real and lethal disease is growing inside you. The real solution to the problem of the atrophic anisotropic dataset is to educate radiologist on the fundamental principles of image processing. If you really care about the technical decency of the medical image processing field, and you really care about providing your best effort to the patients who will receive health care directly or indirectly affected by your processed images, then it is your duty to reject anisotropic datasets and to patiently explain radiologist why a barbarity such as a $[1:5]$ anisotropy ratio makes a data set to be just "a collection of slices" instead of an authentic 3D datasets.

Please, before employing the techniques covered in this section, do kindly invite your fellow radiologist to see the dataset in an orthogonal slice. Zoom in that image in a viewer without any linear interpolation until you see the daunting reality of the rectangular pixels. Let her/him know how absurd is to process digital data that have been sampled at ratios of $[1:5]$ or $[1:10]$. Then, let them know that the first thing that you are going to do is to throw away all that high in-plane resolution and to *make up* data in-between the slices in order to compensate for their low resolution. Only then, you will have gained the right to use this code.

Let's now move into the code.... and, yes, bring with you that guilt[5], because the fact that you are going to use the code below, is the evidence that we have lost one more battle on the quest for real 3D dataset processing.

This example performs subsampling on the in-plane resolution and performs super-sampling along the inter-slices resolution. The subsampling process requires that we preprocess the data with a smoothing filter in order to avoid the occurrence of aliasing effects due to overlap of the spectrum in the frequency domain [75, 76]. The smoothing is performed here using the `RecursiveGaussian` filter, given that it provides a convenient run-time performance.

The first thing that you will need to do in order to resample this ugly anisotropic dataset is to include the header files for the `itk::ResampleImageFilter`, and the Gaussian smoothing filter.

[5]A feeling of regret or remorse for having committed some improper act; a recognition of one's own responsibility for doing something wrong.

```
#include "itkResampleImageFilter.h"
#include "itkRecursiveGaussianImageFilter.h"
```

The resampling filter will need a Transform in order to map point coordinates and will need an interpolator in order to compute intensity values for the new resampled image. In this particular case we use the itk::IdentityTransform because the image is going to be resampled by preserving the physical extent of the sampled region. The Linear interpolator is used as a common trade-off, although arguably we should use one type of interpolator for the in-plane subsampling process and another one for the inter-slice supersampling, but again, one should wonder why to enter into technical sophistication here, when what we are doing is to cover-up for an improper acquisition of medical data, and we are just trying to make it look as if it was correctly acquired.

```
#include "itkIdentityTransform.h"
#include "itkLinearInterpolateImageFunction.h"
```

Note that, as part of the preprocessing of the image, in this example we are also rescaling the range of intensities. This operation has already been described as Intensity Windowing. In a real clinical application, this step requires careful consideration of the range of intensities that contain information about the anatomical structures that are of interest for the current clinical application. It practice you may want to remove this step of intensity rescaling.

```
#include "itkIntensityWindowingImageFilter.h"
```

We made explicit now our choices for the pixel type and dimension of the input image to be processed, as well as the pixel type that we intend to use for the internal computation during the smoothing and resampling.

```
  const     unsigned int    Dimension = 3;

  typedef   unsigned short  InputPixelType;
  typedef   float           InternalPixelType;

  typedef itk::Image< InputPixelType,    Dimension >   InputImageType;
  typedef itk::Image< InternalPixelType, Dimension >   InternalImageType;
```

We instantiate the smoothing filter that will be used on the preprocessing for subsampling the in-plane resolution of the dataset.

```
  typedef itk::RecursiveGaussianImageFilter<
                                  InternalImageType,
                                  InternalImageType > GaussianFilterType;
```

We create two instances of the smoothing filter, one will smooth along the *X* direction while the other will smooth along the *Y* direction. They are connected in a cascade in the pipeline, while taking their input from the intensity windowing filter. Note that you may want to skip the intensity windowing scale and simply take the input directly from the reader.

```
  GaussianFilterType::Pointer smootherX = GaussianFilterType::New();
  GaussianFilterType::Pointer smootherY = GaussianFilterType::New();

  smootherX->SetInput( intensityWindowing->GetOutput() );
  smootherY->SetInput( smootherX->GetOutput() );
```

We must now provide the settings for the resampling itself. This is done by searching for a value of isotropic resolution that will provide a trade-off between the evil of subsampling and the evil of supersampling. We advance here the conjecture that the geometrical mean between the in-plane and the inter-slice resolutions should be a convenient isotropic resolution to use. This conjecture is supported on nothing else than intuition and common sense. You can rightfully argue that this choice deserves a more technical consideration, but then, if you are so inclined to the technical correctness of the image sampling process, you should not be using this code, and should rather we talking about such technical correctness to the radiologist who acquired this ugly anisotropic dataset.

We take the image from the input and then request its array of pixel spacing values.

```
  InputImageType::ConstPointer inputImage = reader->GetOutput();

  const InputImageType::SpacingType& inputSpacing = inputImage->GetSpacing();
```

and apply our ad-hoc conjecture that the correct anisotropic resolution to use is the geometrical mean of the in-plane and inter-slice resolutions. Then set this spacing as the Sigma value to be used for the Gaussian smoothing at the preprocessing stage.

```
  const double isoSpacing = sqrt( inputSpacing[2] * inputSpacing[0] );

  smootherX->SetSigma( isoSpacing );
  smootherY->SetSigma( isoSpacing );
```

We instruct the smoothing filters to act along the *X* and *Y* direction respectively. And define the settings for avoiding the loss of intensity as a result of the diffusion process that is inherited from the use of a Gaussian filter.

```
  smootherX->SetDirection( 0 );
  smootherY->SetDirection( 1 );

  smootherX->SetNormalizeAcrossScale( true );
  smootherY->SetNormalizeAcrossScale( true );
```

Now that we have taken care of the smoothing in-plane, we proceed to instantiate the resampling filter that will reconstruct an isotropic image. We start by declaring the pixel type to be use at the output of such filter, then instantiate the image type and the type for the resampling filter. Finally we construct an instantiation of such a filter.

```
typedef   unsigned char   OutputPixelType;

typedef itk::Image< OutputPixelType,   Dimension >   OutputImageType;

typedef itk::ResampleImageFilter<
              InternalImageType, OutputImageType >  ResampleFilterType;

ResampleFilterType::Pointer resampler = ResampleFilterType::New();
```

The resampling filter requires that we provide a Transform, that in this particular case can simply be an identity transform.

```
typedef itk::IdentityTransform< double, Dimension >  TransformType;

TransformType::Pointer transform = TransformType::New();
transform->SetIdentity();

resampler->SetTransform( transform );
```

The filter also requires an interpolator to be passed to it. In this case we chose to use a linear interpolator.

```
typedef itk::LinearInterpolateImageFunction<
                        InternalImageType, double >  InterpolatorType;

InterpolatorType::Pointer interpolator = InterpolatorType::New();

resampler->SetInterpolator( interpolator );
```

The pixel spacing of the resampled dataset is loaded in a `SpacingType` and passed to the resampling filter.

```
OutputImageType::SpacingType spacing;

spacing[0] = isoSpacing;
spacing[1] = isoSpacing;
spacing[2] = isoSpacing;

resampler->SetOutputSpacing( spacing );
```

The origin of the output image is maintained, since we decided to resample the image in the same physical extent of the input anisotropic image.

```
resampler->SetOutputOrigin( inputImage->GetOrigin() );
```

The number of pixels to use along each dimension in the grid of the resampled image is computed using the ratio between the pixel spacings of the input image and those of the output image. Note that the computation of the number of pixels along the *Z* direction is slightly different with the purpose of making sure that we don't attempt to compute pixels that are outside of the original anisotropic dataset.

```
InputImageType::SizeType   inputSize =
                  inputImage->GetLargestPossibleRegion().GetSize();

typedef InputImageType::SizeType::SizeValueType SizeValueType;

const double dx = inputSize[0] * inputSpacing[0] / isoSpacing;
const double dy = inputSize[1] * inputSpacing[1] / isoSpacing;

const double dz = (inputSize[2] - 1 ) * inputSpacing[2] / isoSpacing;
```

Finally the values are stored in a `SizeType` and passed to the resampling filter. Note that this process requires a casting since the computation are performed in `double`, while the elements of the `SizeType` are integers.

```
InputImageType::SizeType   size;

size[0] = static_cast<SizeValueType>( dx );
size[1] = static_cast<SizeValueType>( dy );
size[2] = static_cast<SizeValueType>( dz );

resampler->SetSize( size );
```

Our last action is to take the input for the resampling image filter from the output of the cascade of smoothing filters, and then to trigger the execution of the pipeline by invoking the `Update()` method on the resampling filter.

```
resampler->SetInput( smootherY->GetOutput() );

resampler->Update();
```

At this point we should take some minutes in silence to reflect on the circumstances that have lead us to accept to cover-up for the improper acquisition of medical data.

6.10 Frequency Domain

6.10.1 Computing a Fast Fourier Transform (FFT)

The source code for this section can be found in the file
`Examples/Filtering/FFTImageFilter.cxx`.

In this section we assume that you are familiar with Spectral Analysis, in particular with the concepts of the Fourier Transform and the numerical implementation of the Fast Fourier transform. If you are not familiar with these concepts you may want to consult first any of the many available introductory books to spectral analysis [10, 11].

This example illustrates how to use the Fast Fourier Transform filter (FFT) for processing an image in the spectral domain. Given that FFT computation can be CPU intensive, there are multiple hardware specific implementations of FFT. IT is convenient in many cases to delegate the actual computation of the transform to local available libraries. Particular examples of those libraries are fftw[6] and the VXL implementation of FFT. For this reason ITK provides a base abstract class that factorizes the interface to multiple specific implementations of FFT. This base class is the `itk::FFTRealToComplexConjugateImageFilter`, and two of its derived classes are `itk::VnlFFTRealToComplexConjugateImageFilter` and `itk::FFTWRealToComplexConjugateImageFilter`.

A typical application that uses FFT will need to include the following header files.

```
#include "itkImage.h"
#include "itkVnlFFTRealToComplexConjugateImageFilter.h"
#include "itkComplexToRealImageFilter.h"
#include "itkComplexToImaginaryImageFilter.h"
```

The first decision to make is related to the pixel type and dimension of the images on which we want to compute the Fourier transform.

```
  typedef float  PixelType;
  const unsigned int Dimension = 2;

  typedef itk::Image< PixelType, Dimension > ImageType;
```

We use the same image type in order to instantiate the FFT filter. In this case the `itk::VnlFFTRealToComplexConjugateImageFilter`. Note that contrary to most ITK filters, the FFT filter is instantiated using the Pixel type and the image dimension explicitly. Once the filter type is instantiated, we can use it for creating one object by invoking the `New()` method and assigning the result to a SmartPointer.

```
  typedef itk::VnlFFTRealToComplexConjugateImageFilter<
```

[6]http://www.fftw.org

```
                                          PixelType, Dimension >  FFTFilterType;

  FFTFilterType::Pointer fftFilter = FFTFilterType::New();
```

The input to this filter can be taken from a reader, for example.

```
  typedef itk::ImageFileReader< ImageType >  ReaderType;
  ReaderType::Pointer reader = ReaderType::New();
  reader->SetFileName( argv[1] );

  fftFilter->SetInput( reader->GetOutput() );
```

The execution of the filter can be triggered by invoking the `Update()` method. Since this invocation can eventually throw and exception, the call must be placed inside a try/catch block.

```
  try
    {
    fftFilter->Update();
    }
  catch( itk::ExceptionObject & excp )
    {
    std::cerr << "Error: " << std::endl;
    std::cerr << excp << std::endl;
    return EXIT_FAILURE;
    }
```

In general the output of the FFT filter will be a complex image. We can proceed to save this image in a file for further analysis. This can be done by simply instantiating an `itk::ImageFileWriter` using the trait of the output image from the FFT filter. We construct one instance of the writer and pass the output of the FFT filter as the input of the writer.

```
  typedef FFTFilterType::OutputImageType    ComplexImageType;

  typedef itk::ImageFileWriter< ComplexImageType > ComplexWriterType;

  ComplexWriterType::Pointer complexWriter = ComplexWriterType::New();
  complexWriter->SetFileName("complexImage.mhd");

  complexWriter->SetInput( fftFilter->GetOutput() );
```

Finally we invoke the `Update()` method placing inside a try/catch block.

```
  try
    {
    complexWriter->Update();
    }
  catch( itk::ExceptionObject & excp )
    {
    std::cerr << "Error: " << std::endl;
    std::cerr << excp << std::endl;
    return EXIT_FAILURE;
    }
```

In addition to saving the complex image into a file, we could also extract its real and imaginary parts for further analysis. This can be done with the `itk::ComplexToRealImageFilter` and the `itk::ComplexToImaginaryImageFilter`.

We instantiate first the ImageFilter that will help us to extract the real part from the complex image. The `ComplexToRealImageFilter` takes as first template parameter the type of the complex image and as second template parameter it takes the type of the output image pixel. We create one instance of this filter and connect as its input the output of the FFT filter.

```
  typedef itk::ComplexToRealImageFilter<
                 ComplexImageType, ImageType > RealFilterType;

  RealFilterType::Pointer realFilter = RealFilterType::New();

  realFilter->SetInput( fftFilter->GetOutput() );
```

Since the range of intensities in the Fourier domain can be quite concentrated, it result convenient to rescale the image in order to visualize it. For this purpose we instantiate here a `itk::RescaleIntensityImageFilter` that will rescale the intensities of the `real` image into a range suitable for writing in a file. We also set the minimum and maximum values of the output to the range of the pixel type used for writing.

```
  typedef itk::RescaleIntensityImageFilter<
                                ImageType,
                                WriteImageType > RescaleFilterType;

  RescaleFilterType::Pointer intensityRescaler = RescaleFilterType::New();

  intensityRescaler->SetInput( realFilter->GetOutput() );

  intensityRescaler->SetOutputMinimum(  0  );
  intensityRescaler->SetOutputMaximum( 255 );
```

We can now instantiate the ImageFilter that will help us to extract the imaginary part from the complex image. The filter that we use here is the `itk::ComplexToImaginaryImageFilter`. It takes as first template parameter the type of the complex image and as second template parameter it takes the type of the output image pixel. An instance of the filter is created, and its input is connected to the output of the FFT filter.

```
  typedef FFTFilterType::OutputImageType    ComplexImageType;

  typedef itk::ComplexToImaginaryImageFilter<
                        ComplexImageType, ImageType > ImaginaryFilterType;

  ImaginaryFilterType::Pointer imaginaryFilter = ImaginaryFilterType::New();

  imaginaryFilter->SetInput( fftFilter->GetOutput() );
```

The Imaginary image can then be rescaled and saved into a file, just as we did with the Real part.

For the sake of illustrating the use of a `itk::ImageFileReader` on Complex images, here we instantiate a reader that will load the Complex image that we just saved. Note that nothing special is required in this case. The instantiation is done just the same as for any other type of image. Which once again illustrates the power of Generic Programming.

```
  typedef itk::ImageFileReader< ComplexImageType > ComplexReaderType;

  ComplexReaderType::Pointer complexReader = ComplexReaderType::New();

  complexReader->SetFileName("complexImage.mhd");
  complexReader->Update();
```

6.10.2 Filtering on the Frequency Domain

The source code for this section can be found in the file
`Examples/Filtering/FFTImageFilterFourierDomainFiltering.cxx`.

One of the most common image processing operations performed in the Fourier Domain is the masking of the spectrum in order to eliminate a range of spatial frequencies from the input image. This operation is typically performed by taking the input image, computing its Fourier transform using a FFT filter, masking the resulting image in the Fourier domain with a mask, and finally taking the result of the masking and computing its inverse Fourier transform.

This typical processing is what it is illustrated in the example below.

We start by including the headers of the FFT filters and the Mask image filter. Note that we use two different types of FFT filters here. The first one expects as input an image of real pixel type (real in the sense of complex numbers) and produces as output a complex image. The second FFT filter expects as in put a complex image and produces a real image as output.

```
#include "itkVnlFFTRealToComplexConjugateImageFilter.h"
#include "itkVnlFFTComplexConjugateToRealImageFilter.h"
#include "itkMaskImageFilter.h"
```

The first decision to make is related to the pixel type and dimension of the images on which we want to compute the Fourier transform.

```
typedef float  InputPixelType;
const unsigned int Dimension = 2;

typedef itk::Image< InputPixelType, Dimension > InputImageType;
```

Then we select the pixel type to use for the mask image and instantiate the image type of the mask.

```
typedef unsigned char  MaskPixelType;

typedef itk::Image< MaskPixelType, Dimension > MaskImageType;
```

Both the input image and the mask image can be read from files or could be obtained as the output of a preprocessing pipeline. We omit here the details of reading the image since the process is quite standard.

Now the `itk::VnlFFTRealToComplexConjugateImageFilter` can be instantiated. Note that contrary to most ITK filters, the FFT filter is instantiated using the Pixel type and the image dimension explicitly. Using the type we construct one instance of the filter.

```
typedef itk::VnlFFTRealToComplexConjugateImageFilter<
                                InputPixelType, Dimension >  FFTFilterType;

FFTFilterType::Pointer fftFilter = FFTFilterType::New();

fftFilter->SetInput( inputReader->GetOutput() );
```

Since our purpose is to perform filtering in the frequency domain by altering the weights of the image spectrum, we need here a filter that will mask the Fourier transform of the input image with a binary image. Note that the type of the spectral image is taken here from the traits of the FFT filter.

```
typedef FFTFilterType::OutputImageType    SpectralImageType;

typedef itk::MaskImageFilter< SpectralImageType,
                                  MaskImageType,
                                  SpectralImageType >  MaskFilterType;

MaskFilterType::Pointer maskFilter = MaskFilterType::New();
```

We connect the inputs to the mask filter by taking the outputs from the first FFT filter and from the reader of the Mask image.

```
maskFilter->SetInput1( fftFilter->GetOutput() );
maskFilter->SetInput2( maskReader->GetOutput() );
```

For the purpose of verifying the aspect of the spectrum after being filtered with the mask, we can write out the output of the Mask filter to a file.

```
typedef itk::ImageFileWriter< SpectralImageType > SpectralWriterType;
SpectralWriterType::Pointer spectralWriter = SpectralWriterType::New();
spectralWriter->SetFileName("filteredSpectrum.mhd");
spectralWriter->SetInput( maskFilter->GetOutput() );
spectralWriter->Update();
```

The output of the mask filter will contain the *filtered* spectrum of the input image. We must then apply an inverse Fourier transform on it in order to obtain the filtered version of the input image. For that purpose we create another instance of the FFT filter.

```
  typedef itk::VnlFFTComplexConjugateToRealImageFilter<
                                InputPixelType, Dimension >  IFFTFilterType;

  IFFTFilterType::Pointer fftInverseFilter = IFFTFilterType::New();

  fftInverseFilter->SetInput( maskFilter->GetOutput() );
```

The execution of the pipeline can be triggered by invoking the `Update()` method in this last filter. Since this invocation can eventually throw and exception, the call must be placed inside a try/catch block.

```
  try
    {
    fftInverseFilter->Update();
    }
  catch( itk::ExceptionObject & excp )
    {
    std::cerr << "Error: " << std::endl;
    std::cerr << excp << std::endl;
    return EXIT_FAILURE;
    }
```

The result of the filtering can now be saved into an image file, or be passed to a subsequent processing pipeline. Here we simply write it out to an image file.

```
  typedef itk::ImageFileWriter< InputImageType > WriterType;
  WriterType::Pointer writer = WriterType::New();
  writer->SetFileName( argv[3] );
  writer->SetInput( fftInverseFilter->GetOutput() );
```

Note that this example is just a minimal illustration of the multiple types of processing that are possible in the Fourier domain.

6.11 Extracting Surfaces

6.11.1 Surface extraction

The source code for this section can be found in the file
`Examples/Filtering/SurfaceExtraction.cxx`.

Surface extraction has attracted continuous interest since the early days of image analysis, in particular on the context of medical applications. Although it is commonly associated with image segmentation, surface extraction is not in itself a segmentation technique, instead it is a transformation that changes the way a segmentation is represented. In its most common form, isosurface extraction is the equivalent of image thresholding followed by surface extraction.

Probably the most widely known method of surface extraction is the *Marching Cubes* algorithm [51]. Although it has been followed by a number of variants [72], Marching Cubes has become an icon on medical image processing. The following example illustrates how to perform surface extraction in ITK using an algorithm similar to Marching Cubes [7].

The representation of unstructured data in ITK is done with the `itk::Mesh`. This class allows to represent N-Dimensional grids of varied topology. It is natural for the filter that extracts surfaces from an Image to produce a Mesh as its output.

We initiate our example by including the header files of the surface extraction filter, the image and the Mesh.

```
#include "itkBinaryMask3DMeshSource.h"
#include "itkImage.h"
#include "itkMesh.h"
```

We define then the pixel type and dimension of the image from which we are going to extract the surface.

```
  const unsigned int Dimension = 3;
  typedef unsigned char  PixelType;

  typedef itk::Image< PixelType, Dimension >   ImageType;
```

With the same image type we instantiate the type of an ImageFileReader and construct one with the purpose of reading in the input image.

[7]Note that the Marching Cubes algorithm is covered by a patent that expired on June 5th 2005.

```
typedef itk::ImageFileReader< ImageType >    ReaderType;
ReaderType::Pointer reader = ReaderType::New();
reader->SetFileName( argv[1] );
```

The type of the `itk::Mesh` is instantiated by specifying the type to be associated with the pixel value of the Mesh nodes. This particular pixel type happens to be irrelevant for the purpose of extracting the surface.

```
typedef itk::Mesh<double>                         MeshType;
```

Having declared the Image and Mesh types we can now instantiate the surface extraction filter, and construct one by invoking its `New()` method.

```
typedef itk::BinaryMask3DMeshSource< ImageType, MeshType >   MeshSourceType;

MeshSourceType::Pointer meshSource = MeshSourceType::New();
```

In this particular example, the pixel value to be associated to the object to be extracted is read from the command line arguments and it is passed to the filter by using the `SetObjectValue()` method. Note that this is different from the traditional isovalue used in the Marching Cubes algorithm. In the case of the `BinaryMask3DMeshSource` filter, the object values defines the membership of pixels to the object from which the surface will be extracted. In other words, the surface will be surrounding all pixels with value equal to the ObjectValue parameter.

```
const PixelType objectValue = static_cast<PixelType>( atof( argv[2] ) );

meshSource->SetObjectValue( objectValue );
```

The input to the surface extraction filter is taken from the output of the image reader.

```
meshSource->SetInput( reader->GetOutput() );
```

Finally we trigger the execution of the pipeline by invoking the `Update()` method. Given that the pipeline may throw an exception this call must be place inside a `try/catch` block.

```
try
  {
  meshSource->Update();
  }
catch( itk::ExceptionObject & exp )
  {
  std::cerr << "Exception thrown during Update() " << std::endl;
  std::cerr << exp << std::endl;
  return EXIT_FAILURE;
  }
```

As a way of taking a look at the output Mesh we print out here its number of Nodes and Cells.

```
std::cout << "Nodes = " << meshSource->GetNumberOfNodes() << std::endl;
std::cout << "Cells = " << meshSource->GetNumberOfCells() << std::endl;
```

This resulting Mesh could be used as input for a deformable model segmentation algorithm, or it could be converted to a format suitable for visualization in an interactive application.

CHAPTER

SEVEN

Reading and Writing Images

This chapter describes the toolkit architecture supporting reading and writing of images to files. ITK does not enforce any particular file format, instead, it provides a structure supporting a variety of formats that can be easily extended by the user as new formats become available.

We begin the chapter with some simple examples of file I/O.

7.1 Basic Example

The source code for this section can be found in the file
`Examples/IO/ImageReadWrite.cxx`.

The classes responsible for reading and writing images are located at the beginning and end of the data processing pipeline. These classes are known as data sources (readers) and data sinks (writers). Generally speaking they are referred to as filters, although readers have no pipeline input and writers have no pipeline output.

The reading of images is managed by the class `itk::ImageFileReader` while writing is performed by the class `itk::ImageFileWriter`. These two classes are independent of any particular file format. The actual low level task of reading and writing specific file formats is done behind the scenes by a family of classes of type `itk::ImageIO`.

The first step for performing reading and writing is to include the following headers.

```
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
```

Then, as usual, a decision must be made about the type of pixel used to represent the image processed by the pipeline. Note that when reading and writing images, the pixel type of the image **is not necessarily** the same as the pixel type stored in the file. Your choice of the pixel type (and hence template parameter) should be driven mainly by two considerations:

- It should be possible to cast the file pixel type in the file to the pixel type you select. This casting will be performed using the standard C-language rules, so you will have to make sure that the conversion does not result in information being lost.
- The pixel type in memory should be appropriate to the type of processing you intended to apply on the images.

A typical selection for medical images is illustrated in the following lines.

```
typedef unsigned short      PixelType;
const   unsigned int        Dimension = 2;
typedef itk::Image< PixelType, Dimension >    ImageType;
```

Note that the dimension of the image in memory should match the one of the image in file. There are a couple of special cases in which this condition may be relaxed, but in general it is better to ensure that both dimensions match.

We can now instantiate the types of the reader and writer. These two classes are parameterized over the image type.

```
typedef itk::ImageFileReader< ImageType >  ReaderType;
typedef itk::ImageFileWriter< ImageType >  WriterType;
```

Then, we create one object of each type using the New() method and assigning the result to a `itk::SmartPointer`.

```
ReaderType::Pointer reader = ReaderType::New();
WriterType::Pointer writer = WriterType::New();
```

The name of the file to be read or written is passed with the SetFileName() method.

```
reader->SetFileName( inputFilename  );
writer->SetFileName( outputFilename );
```

We can now connect these readers and writers to filters to create a pipeline. For example, we can create a short pipeline by passing the output of the reader directly to the input of the writer.

```
writer->SetInput( reader->GetOutput() );
```

At first view, this may seem as a quite useless program, but it is actually implementing a powerful file format conversion tool! The execution of the pipeline is triggered by the invocation of the `Update()` methods in one of the final objects. In this case, the final data pipeline object is the writer. It is a wise practice of defensive programming to insert any `Update()` call inside a `try/catch` block in case exceptions are thrown during the execution of the pipeline.

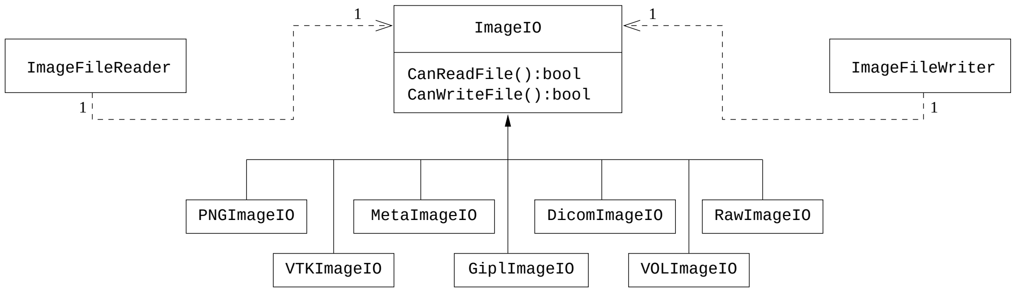


Figure 7.1: Collaboration diagram of the ImageIO classes.

```
try
  {
  writer->Update();
  }
catch( itk::ExceptionObject & err )
  {
  std::cerr << "ExceptionObject caught !" << std::endl;
  std::cerr << err << std::endl;
  return EXIT_FAILURE;
  }
```

Note that exceptions should only be caught by pieces of code that know what to do with them. In a typical application this `catch` block should probably reside on the GUI code. The action on the `catch` block could inform the user about the failure of the IO operation.

The IO architecture of the toolkit makes it possible to avoid explicit specification of the file format used to read or write images.[1] The object factory mechanism enables the ImageFileReader and ImageFileWriter to determine (at run-time) with which file format it is working with. Typically, file formats are chosen based on the filename extension, but the architecture supports arbitrarily complex processes to determine whether a file can be read or written. Alternatively, the user can specify the data file format by explicit instantiation and assignment the appropriate `itk::ImageIO` subclass.

For historical reasons and as a convenience to the user, the `itk::ImageFileWriter` also has a Write() method that is aliased to the `Update()` method. You can in principle use either of them but `Update()` is recommended since Write() may be deprecated in the future.

To better understand the IO architecture, please refer to Figures 7.1, 7.2, and 7.3.

The following section describes the internals of the IO architecture provided in the toolkit.

[1] In this example no file format is specified; this program can be used as a general file conversion utility.

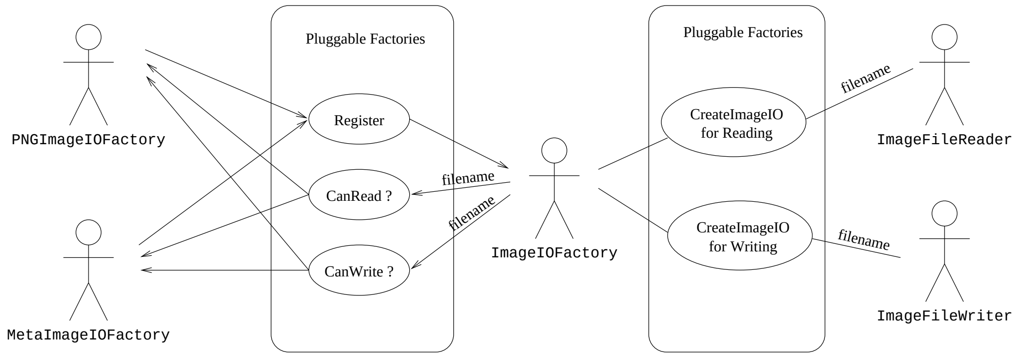


Figure 7.2: Use cases of ImageIO factories.

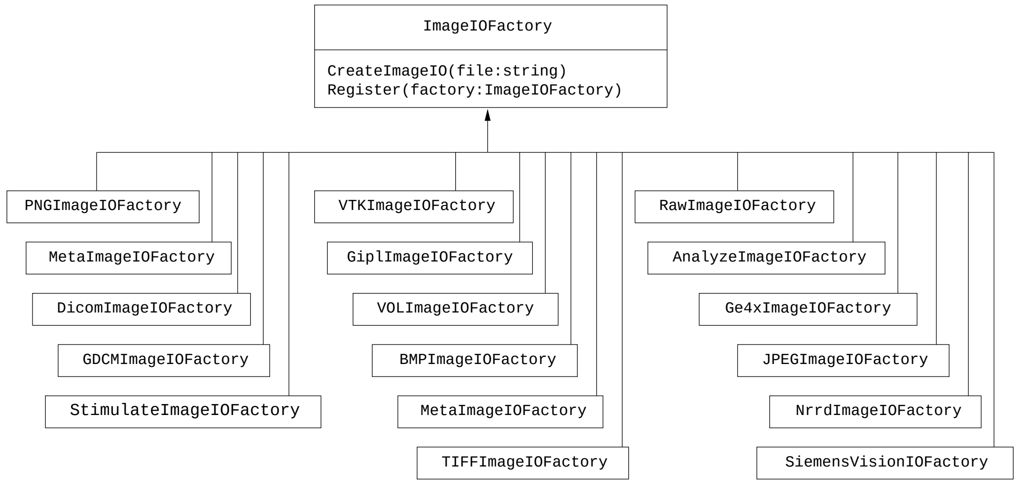


Figure 7.3: Class diagram of the ImageIO factories.

7.2 Pluggable Factories

The principle behind the input/output mechanism used in ITK is known as *pluggable-factories* [28]. This concept is illustrated in the UML diagram in Figure 7.1. From the user's point of view the objects responsible for reading and writing files are the `itk::ImageFileReader` and `itk::ImageFileWriter` classes. These two classes, however, are not aware of the details involved in reading or writing particular file formats like PNG or DICOM. What they do is to dispatch the user's requests to a set of specific classes that are aware of the details of image file formats. These classes are the `itk::ImageIO` classes. The ITK delegation mechanism enables users to extend the number of supported file formats by just adding new classes to the ImageIO hierarchy.

Each instance of ImageFileReader and ImageFileWriter has a pointer to an ImageIO object. If this pointer is empty, it will be impossible to read or write an image and the image file reader/writer must determine which ImageIO class to use to perform IO operations. This is done basically by passing the filename to a centralized class, the `itk::ImageIOFactory` and asking it to identify any subclass of ImageIO capable of reading or writing the user-specified file. This is illustrated by the use cases on the right side of Figure 7.2.

Each class derived from ImageIO must provide an associated factory class capable of producing an instance of the ImageIO class. For example, for PNG files, there is a `itk::PNGImageIO` object that knows how to read this image files and there is a `itk::PNGImageIOFactory` class capable of constructing a PNGImageIO object and returning a pointer to it. Each time a new file format is added (i.e., a new ImageIO subclass is created), a factory must be implemented as a derived class of the ImageIOFactory class as illustrated in Figure 7.3.

For example, in order to read PNG files, a PNGImageIOFactory is created and registered with the central ImageIOFactory singleton[2] class as illustrated in the left side of Figure 7.2. When the ImageFileReader asks the ImageIOFactory for an ImageIO capable of reading the file identified with *filename* the ImageIOFactory will iterate over the list of registered factories and will ask each one of them is they know how to read the file. The factory that responds affirmatively will be used to create the specific ImageIO instance that will be returned to the ImageFileReader and used to perform the read operations.

In most cases the mechanism is transparent to the user who only interacts with the Image-FileReader and ImageFileWriter. It is possible, however, to explicitly select the type of ImageIO object to use. This is illustrated by the following example.

7.3 Using ImageIO Classes Explicitly

The source code for this section can be found in the file
`Examples/IO/ImageReadExportVTK.cxx`.

[2] *Singleton* means that there is only one instance of this class in a particular application

In cases where the user knows what file format to use and wants to indicate this explicitly, a specific `itk::ImageIO` class can be instantiated and assigned to the image file reader or writer. This circumvents the `itk::ImageIOFactory` mechanism which tries to find the appropriate ImageIO class for performing the IO operations. Explicit selection of the ImageIO also allows the user to invoke specialized features of a particular class which may not be available from the general API provide by ImageIO.

The following example illustrates explicit instantiating of an IO class (in this case a VTK file format), setting its parameters and then connecting it to the `itk::ImageFileWriter`.

The example begins by including the appropriate headers.

```
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
#include "itkVTKImageIO.h"
```

Then, as usual, we select the pixel types and the image dimension. Remember, if the file format represents pixels with a particular type, C-style casting will be performed to convert the data.

```
  typedef unsigned short      PixelType;
  const   unsigned int        Dimension = 2;
  typedef itk::Image< PixelType, Dimension >    ImageType;
```

We can now instantiate the reader and writer. These two classes are parameterized over the image type. We instantiate the `itk::VTKImageIO` class as well. Note that the ImageIO objects are not templated.

```
  typedef itk::ImageFileReader< ImageType >  ReaderType;
  typedef itk::ImageFileWriter< ImageType >  WriterType;
  typedef itk::VTKImageIO                    ImageIOType;
```

Then, we create one object of each type using the New() method and assigning the result to a `itk::SmartPointer`.

```
  ReaderType::Pointer reader = ReaderType::New();
  WriterType::Pointer writer = WriterType::New();
  ImageIOType::Pointer vtkIO = ImageIOType::New();
```

The name of the file to be read or written is passed with the SetFileName() method.

```
  reader->SetFileName( inputFilename  );
  writer->SetFileName( outputFilename );
```

We can now connect these readers and writers to filters in a pipeline. For example, we can create a short pipeline by passing the output of the reader directly to the input of the writer.

```
writer->SetInput( reader->GetOutput() );
```

Explicitly declaring the specific VTKImageIO allow users to invoke methods specific to a particular IO class. For example, the following line specifies to the writer to use ASCII format when writing the pixel data.

```
vtkIO->SetFileTypeToASCII();
```

The VTKImageIO object is then connected to the ImageFileWriter. This will short-circuit the action of the ImageIOFactory mechanism. The ImageFileWriter will not attempt to look for other ImageIO objects capable of performing the writing tasks. It will simply invoke the one provided by the user.

```
writer->SetImageIO( vtkIO );
```

Finally we invoke Update() on the ImageFileWriter and place this call inside a try/catch block in case any errors occur during the writing process.

```
try
  {
  writer->Update();
  }
catch( itk::ExceptionObject & err )
  {
  std::cerr << "ExceptionObject caught !" << std::endl;
  std::cerr << err << std::endl;
  return EXIT_FAILURE;
  }
```

Although this example only illustrates how to use an explicit ImageIO class with the ImageFileWriter, the same can be done with the ImageFileReader. The typical case in which this is done is when reading raw image files with the `itk::RawImageIO` object. The drawback of this approach is that the parameters of the image have to be explicitly written in the code. The direct use of raw file is **strongly discouraged** in medical imaging. It is always better to create a header for a raw file by using any of the file formats that combine a text header file and a raw binary file, like `itk::MetaImageIO`, `itk::GiplImageIO` and `itk::VTKImageIO`.

7.4 Reading and Writing RGB Images

The source code for this section can be found in the file
`Examples/IO/RGBImageReadWrite.cxx`.

RGB images are commonly used for representing data acquired from cryogenic sections, optical microscopy and endoscopy. This example illustrates how to read and write RGB color images to and from a file. This requires the following headers as shown.

```
#include "itkRGBPixel.h"
#include "itkImage.h"
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
```

The itk::RGBPixel class is templated over the type used to represent each one of the red, green and blue components. A typical instantiation of the RGB image class might be as follows.

```
  typedef itk::RGBPixel< unsigned char >   PixelType;
  typedef itk::Image< PixelType, 2 >       ImageType;
```

The image type is used as a template parameter to instantiate the reader and writer.

```
  typedef itk::ImageFileReader< ImageType >  ReaderType;
  typedef itk::ImageFileWriter< ImageType >  WriterType;

  ReaderType::Pointer reader = ReaderType::New();
  WriterType::Pointer writer = WriterType::New();
```

The filenames of the input and output files must be provided to the reader and writer respectively.

```
  reader->SetFileName( inputFilename  );
  writer->SetFileName( outputFilename );
```

Finally, execution of the pipeline can be triggered by invoking the Update() method in the writer.

```
  writer->Update();
```

You may have noticed that apart from the declaration of the `PixelType` there is nothing in this code that is specific for RGB images. All the actions required to support color images are implemented internally in the itk::ImageIO objects.

7.5 Reading, Casting and Writing Images

The source code for this section can be found in the file
`Examples/IO/ImageReadCastWrite.cxx`.

Given that ITK is based on the Generic Programming paradigm, most of the types are defined at compilation time. It is sometimes important to anticipate conversion between different types of images. The following example illustrates the common case of reading an image of one pixel type and writing it on a different pixel type. This process not only involves casting but also rescaling the image intensity since the dynamic range of the input and output pixel types can be quite different. The `itk::RescaleIntensityImageFilter` is used here to linearly rescale the image values.

The first step in this example is to include the appropriate headers.

```
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
#include "itkRescaleIntensityImageFilter.h"
```

Then, as usual, a decision should be made about the pixel type that should be used to represent the images. Note that when reading an image, this pixel type **is not necessarily** the pixel type of the image stored in the file. Instead, it is the type that will be used to store the image as soon as it is read into memory.

```
  typedef float              InputPixelType;
  typedef unsigned char      OutputPixelType;
  const   unsigned int       Dimension = 2;

  typedef itk::Image< InputPixelType,  Dimension >    InputImageType;
  typedef itk::Image< OutputPixelType, Dimension >    OutputImageType;
```

Note that the dimension of the image in memory should match the one of the image in file. There are a couple of special cases in which this condition may be relaxed, but in general it is better to ensure that both dimensions match.

We can now instantiate the types of the reader and writer. These two classes are parameterized over the image type.

```
  typedef itk::ImageFileReader< InputImageType  >  ReaderType;
  typedef itk::ImageFileWriter< OutputImageType >  WriterType;
```

Below we instantiate the RescaleIntensityImageFilter class that will linearly scale the image intensities.

```
  typedef itk::RescaleIntensityImageFilter<
                                InputImageType,
                                OutputImageType >    FilterType;
```

A filter object is constructed and the minimum and maximum values of the output are selected using the SetOutputMinimum() and SetOutputMaximum() methods.

```
  FilterType::Pointer filter = FilterType::New();
  filter->SetOutputMinimum(   0 );
  filter->SetOutputMaximum( 255 );
```

Then, we create the reader and writer and connect the pipeline.

```
  ReaderType::Pointer reader = ReaderType::New();
  WriterType::Pointer writer = WriterType::New();

  filter->SetInput( reader->GetOutput() );
  writer->SetInput( filter->GetOutput() );
```

The name of the files to be read and written are passed with the SetFileName() method.

```
  reader->SetFileName( inputFilename  );
  writer->SetFileName( outputFilename );
```

Finally we trigger the execution of the pipeline with the Update() method on the writer. The output image will then be the scaled and cast version of the input image.

```
  try
    {
    writer->Update();
    }
  catch( itk::ExceptionObject & err )
    {
    std::cerr << "ExceptionObject caught !" << std::endl;
    std::cerr << err << std::endl;
    return EXIT_FAILURE;
    }
```

7.6 Extracting Regions

The source code for this section can be found in the file
`Examples/IO/ImageReadRegionOfInterestWrite.cxx`.

This example should arguably be placed in the previous filtering chapter. However its usefulness for typical IO operations makes it interesting to mention here. The purpose of this example is to read and image, extract a subregion and write this subregion to a file. This is a common task when we want to apply a computationally intensive method to the region of interest of an image.

As usual with ITK IO, we begin by including the appropriate header files.

```
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
```

The itk::RegionOfInterestImageFilter is the filter used to extract a region from an image. Its header is included below.

```
#include "itkRegionOfInterestImageFilter.h"
```

Image types are defined below.

```
  typedef signed short        InputPixelType;
  typedef signed short        OutputPixelType;
  const   unsigned int        Dimension = 2;

  typedef itk::Image< InputPixelType,  Dimension >    InputImageType;
  typedef itk::Image< OutputPixelType, Dimension >    OutputImageType;
```

The types for the itk::ImageFileReader and itk::ImageFileWriter are instantiated using the image types.

```
  typedef itk::ImageFileReader< InputImageType  >  ReaderType;
  typedef itk::ImageFileWriter< OutputImageType >  WriterType;
```

The RegionOfInterestImageFilter type is instantiated using the input and output image types. A filter object is created with the New() method and assigned to a itk::SmartPointer.

```
  typedef itk::RegionOfInterestImageFilter< InputImageType,
                                            OutputImageType > FilterType;

  FilterType::Pointer filter = FilterType::New();
```

The RegionOfInterestImageFilter requires a region to be defined by the user. The region is specified by an itk::Index indicating the pixel where the region starts and an itk::Size indicating how many pixels the region has along each dimension. In this example, the specification of the region is taken from the command line arguments (this example assumes that a 2D image is being processed).

```
  OutputImageType::IndexType start;
  start[0] = atoi( argv[3] );
  start[1] = atoi( argv[4] );

  OutputImageType::SizeType size;
  size[0] = atoi( argv[5] );
  size[1] = atoi( argv[6] );
```

An itk::ImageRegion object is created and initialized with start and size obtained from the command line.

```
  OutputImageType::RegionType desiredRegion;
  desiredRegion.SetSize(  size  );
  desiredRegion.SetIndex( start );
```

Then the region is passed to the filter using the SetRegionOfInterest() method.

```
  filter->SetRegionOfInterest( desiredRegion );
```

Below, we create the reader and writer using the New() method and assigning the result to a SmartPointer.

```
  ReaderType::Pointer reader = ReaderType::New();
  WriterType::Pointer writer = WriterType::New();
```

The name of the file to be read or written is passed with the SetFileName() method.

```
  reader->SetFileName( inputFilename  );
  writer->SetFileName( outputFilename );
```

Below we connect the reader, filter and writer to form the data processing pipeline.

```
  filter->SetInput( reader->GetOutput() );
  writer->SetInput( filter->GetOutput() );
```

Finally we execute the pipeline by invoking Update() on the writer. The call is placed in a `try/catch` block in case exceptions are thrown.

```
  try
    {
    writer->Update();
    }
  catch( itk::ExceptionObject & err )
    {
    std::cerr << "ExceptionObject caught !" << std::endl;
    std::cerr << err << std::endl;
    return EXIT_FAILURE;
    }
```

7.7 Extracting Slices

The source code for this section can be found in the file
`Examples/IO/ImageReadExtractWrite.cxx`.

This example illustrates the common task of extracting a 2D slice from a 3D volume. This is typically used for display purposes and for expediting user feedback in interactive programs. Here we simply read a 3D volume, extract one of its slices and save it as a 2D image. Note that caution should be used when working with 2D slices from a 3D dataset, since for most image processing operations, the application of a filter on a extracted slice is not equivalent to first applying the filter in the volume and then extracting the slice.

In this example we start by including the appropriate header files.

```
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
```

The filter used to extract a region from an image is the `itk::ExtractImageFilter`. Its header is included below. This filter is capable of extracting $(N-1)$-dimensional images from N-dimensional ones.

```
#include "itkExtractImageFilter.h"
```

Image types are defined below. Note that the input image type is $3D$ and the output image type is $2D$.

```
  typedef signed short        InputPixelType;
  typedef signed short        OutputPixelType;

  typedef itk::Image< InputPixelType,  3 >    InputImageType;
  typedef itk::Image< OutputPixelType, 2 >    OutputImageType;
```

The types for the `itk::ImageFileReader` and `itk::ImageFileWriter` are instantiated using the image types.

```
  typedef itk::ImageFileReader< InputImageType  >  ReaderType;
  typedef itk::ImageFileWriter< OutputImageType >  WriterType;
```

Below, we create the reader and writer using the New() method and assigning the result to a `itk::SmartPointer`.

```
  ReaderType::Pointer reader = ReaderType::New();
  WriterType::Pointer writer = WriterType::New();
```

The name of the file to be read or written is passed with the SetFileName() method.

```
  reader->SetFileName( inputFilename  );
  writer->SetFileName( outputFilename );
```

The ExtractImageFilter type is instantiated using the input and output image types. A filter object is created with the New() method and assigned to a SmartPointer.

```
typedef itk::ExtractImageFilter< InputImageType, OutputImageType > FilterType;
FilterType::Pointer filter = FilterType::New();
```

The ExtractImageFilter requires a region to be defined by the user. The region is specified by an `itk::Index` indicating the pixel where the region starts and an `itk::Size` indication how many pixels the region has along each dimension. In order to extract a $2D$ image from a $3D$ data set, it is enough to set the size of the region to 0 in one dimension. This will indicate to ExtractImageFilter that a dimensional reduction has been specified. Here we take the region from the largest possible region of the input image. Note that Update() is being called first on the reader, since otherwise the output would have invalid data.

```
reader->Update();
InputImageType::RegionType inputRegion =
         reader->GetOutput()->GetLargestPossibleRegion();
```

We take the size from the region and collapse the size in the Z component by setting its value to 0. This will indicate to the ExtractImageFilter that the output image should have a dimension less than the input image.

```
InputImageType::SizeType size = inputRegion.GetSize();
size[2] = 0;
```

Note that in this case we are extracting a Z slice, and for that reason, the dimension to be collapsed in the one with index 2. You may keep in mind the association of index components $\{X = 0, Y = 1, Z = 2\}$. If we were interested in extracting a slice perpendicular to the Y axis we would have set `size[1]=0;`.

Then, we take the index from the region and set its Z value to the slice number we want to extract. In this example we obtain the slice number from the command line arguments.

```
InputImageType::IndexType start = inputRegion.GetIndex();
const unsigned int sliceNumber = atoi( argv[3] );
start[2] = sliceNumber;
```

Finally, an `itk::ImageRegion` object is created and initialized with the start and size we just prepared using the slice information.

```
InputImageType::RegionType desiredRegion;
desiredRegion.SetSize(  size  );
desiredRegion.SetIndex( start );
```

Then the region is passed to the filter using the SetExtractionRegion() method.

```
  filter->SetExtractionRegion( desiredRegion );
```

Below we connect the reader, filter and writer to form the data processing pipeline.

```
  filter->SetInput( reader->GetOutput() );
  writer->SetInput( filter->GetOutput() );
```

Finally we execute the pipeline by invoking Update() on the writer. The call is placed in a `try/catch` block in case exceptions are thrown.

```
  try
    {
    writer->Update();
    }
  catch( itk::ExceptionObject & err )
    {
    std::cerr << "ExceptionObject caught !" << std::endl;
    std::cerr << err << std::endl;
    return EXIT_FAILURE;
    }
```

7.8 Reading and Writing Vector Images

Images whose pixel type is a Vector, a CovariantVector, an Array, or a Complex are quite common in image processing. It is convenient then to describe rapidly how those images can be saved into files and how they can be read from those files later on.

7.8.1 The Minimal Example

The source code for this section can be found in the file
`Examples/IO/VectorImageReadWrite.cxx`.

This example illustrates how to read and write an image of pixel type `itk::Vector`.

We should include the header files for the Image, the ImageFileReader and the ImageFileWriter.

```
#include "itkImage.h"
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
```

Then we define the specific type of vector to be used as pixel type.

```
const unsigned int VectorDimension = 3;

typedef itk::Vector< float, VectorDimension >    PixelType;
```

We define the image dimension, and along with the pixel type we use it for fully instantiating the image type.

```
const unsigned int ImageDimension = 2;

typedef itk::Image< PixelType, ImageDimension > ImageType;
```

Having the image type at hand, we can instantiate the reader and writer types, and use them for creating one object of each type.

```
typedef itk::ImageFileReader< ImageType > ReaderType;
typedef itk::ImageFileWriter< ImageType > WriterType;

ReaderType::Pointer reader = ReaderType::New();
WriterType::Pointer writer = WriterType::New();
```

Filename must be provided to both the reader and the writer. In this particular case we take those filenames from the command line arguments.

```
reader->SetFileName( argv[1] );
writer->SetFileName( argv[2] );
```

Being this a minimal example, we create a short pipeline where we simply connect the output of the reader to the input of the writer.

```
writer->SetInput( reader->GetOutput() );
```

The execution of this short pipeline is triggered by invoking the writer's Update() method. This invocation must be placed inside a try/catch block since its execution may result in exceptions being thrown.

```
try
  {
  writer->Update();
  }
catch( itk::ExceptionObject & err )
  {
  std::cerr << "ExceptionObject caught !" << std::endl;
  std::cerr << err << std::endl;
  return EXIT_FAILURE;
  }
```

Of course, you could envision the addition of filters in between the reader and the writer. Those filters could perform operations on the vector image.

7.8.2 Producing and Writing Covariant Images

The source code for this section can be found in the file
`Examples/IO/CovariantVectorImageWrite.cxx`.

This example illustrates how to write an image whose pixel type is `CovariantVector`. For practical purposes all the content in this example is applicable to images of pixel type `itk::Vector`, `itk::Point` and `itk::FixedArray`. These pixel types are similar in that they are all arrays of fixed size in which the components have the same representational type.

In order to make this example a bit more interesting we setup a pipeline to read an image, compute its gradient and write the gradient to a file. Gradients are represented with `itk::CovariantVector`s as opposed to Vectors. In this way, gradients are transformed correctly under `itk::AffineTransform`s or in general, any transform having anisotropic scaling.

Let's start by including the relevant header files.

```
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
```

We use the `itk::GradientRecursiveGaussianImageFilter` in order to compute the image gradient. The output of this filter is an image whose pixels are CovariantVectors.

```
#include "itkGradientRecursiveGaussianImageFilter.h"
```

We select to read an image of `signed short` pixels and compute the gradient to produce an image of CovariantVector where each component is of type `float`.

```
  typedef signed short          InputPixelType;
  typedef float                 ComponentType;
  const   unsigned int          Dimension = 2;

  typedef itk::CovariantVector< ComponentType,
                                    Dimension  >      OutputPixelType;

  typedef itk::Image< InputPixelType,  Dimension >    InputImageType;
  typedef itk::Image< OutputPixelType, Dimension >    OutputImageType;
```

The `itk::ImageFileReader` and `itk::ImageFileWriter` are instantiated using the image types.

```
  typedef itk::ImageFileReader< InputImageType  >  ReaderType;
  typedef itk::ImageFileWriter< OutputImageType >  WriterType;
```

The GradientRecursiveGaussianImageFilter class is instantiated using the input and output image types. A filter object is created with the New() method and assigned to a itk::SmartPointer.

```
  typedef itk::GradientRecursiveGaussianImageFilter<
                                        InputImageType,
                                        OutputImageType    > FilterType;

  FilterType::Pointer filter = FilterType::New();
```

We select a value for the σ parameter of the GradientRecursiveGaussianImageFilter. Note that this σ is specified in millimeters.

```
  filter->SetSigma( 1.5 );      // Sigma in millimeters
```

Below, we create the reader and writer using the New() method and assigning the result to a SmartPointer.

```
  ReaderType::Pointer reader = ReaderType::New();
  WriterType::Pointer writer = WriterType::New();
```

The name of the file to be read or written is passed with the SetFileName() method.

```
  reader->SetFileName( inputFilename  );
  writer->SetFileName( outputFilename );
```

Below we connect the reader, filter and writer to form the data processing pipeline.

```
  filter->SetInput( reader->GetOutput() );
  writer->SetInput( filter->GetOutput() );
```

Finally we execute the pipeline by invoking Update() on the writer. The call is placed in a `try/catch` block in case exceptions are thrown.

```
  try
    {
    writer->Update();
    }
  catch( itk::ExceptionObject & err )
    {
    std::cerr << "ExceptionObject caught !" << std::endl;
    std::cerr << err << std::endl;
    return EXIT_FAILURE;
    }
```

7.8.3 Reading Covariant Images

Let's now take the image that we just created and read it into another program.

The source code for this section can be found in the file
`Examples/IO/CovariantVectorImageRead.cxx`.

This example illustrates how to read an image whose pixel type is `CovariantVector`. For practical purposes this example is applicable to images of pixel type `itk::Vector`, `itk::Point` and `itk::FixedArray`. These pixel types are similar in that they are all arrays of fixed size in which the components have the same representation type.

In this example we are reading an gradient image from a file (written in the previous example) and computing its magnitude using the `itk::GradientToMagnitudeImageFilter`. Note that this filter is different from the `itk::GradientMagnitudeImageFilter` which actually takes a scalar image as input and compute the magnitude of its gradient. The GradientToMagnitudeImageFilter class takes an image of vector pixel type as input and computes pixel-wise the magnitude of each vector.

Let's start by including the relevant header files.

```
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
#include "itkGradientToMagnitudeImageFilter.h"
#include "itkRescaleIntensityImageFilter.h"
```

We read an image of `itk::CovariantVector` pixels and compute pixel magnitude to produce an image where each pixel is of type `unsigned short`. The components of the CovariantVector are selected to be `float` here. Notice that a renormalization is required in order to map the dynamic range of the magnitude values into the range of the output pixel type. The `itk::RescaleIntensityImageFilter` is used to achieve this.

```
  typedef float                  ComponentType;
  const   unsigned int           Dimension = 2;

  typedef itk::CovariantVector< ComponentType,
                                    Dimension  >      InputPixelType;

  typedef float                                       MagnitudePixelType;
  typedef unsigned short                              OutputPixelType;

  typedef itk::Image< InputPixelType,      Dimension >    InputImageType;
  typedef itk::Image< MagnitudePixelType,  Dimension >    MagnitudeImageType;
  typedef itk::Image< OutputPixelType,     Dimension >    OutputImageType;
```

The `itk::ImageFileReader` and `itk::ImageFileWriter` are instantiated using the image types.

```
typedef itk::ImageFileReader< InputImageType  >  ReaderType;
typedef itk::ImageFileWriter< OutputImageType >  WriterType;
```

The GradientToMagnitudeImageFilter is instantiated using the input and output image types. A filter object is created with the New() method and assigned to a `itk::SmartPointer`.

```
typedef itk::GradientToMagnitudeImageFilter<
                                        InputImageType,
                                        MagnitudeImageType    > FilterType;

FilterType::Pointer filter = FilterType::New();
```

The RescaleIntensityImageFilter class is instantiated next.

```
typedef itk::RescaleIntensityImageFilter<
                                MagnitudeImageType,
                                OutputImageType >      RescaleFilterType;

RescaleFilterType::Pointer  rescaler = RescaleFilterType::New();
```

In the following the minimum and maximum values for the output image are specified. Note the use of the `itk::NumericTraits` class which allows to define a number of type-related constant in a generic way. The use of traits is a fundamental characteristic of generic programming [6, 1].

```
rescaler->SetOutputMinimum( itk::NumericTraits< OutputPixelType >::min() );
rescaler->SetOutputMaximum( itk::NumericTraits< OutputPixelType >::max() );
```

Below, we create the reader and writer using the New() method and assign the result to a SmartPointer.

```
ReaderType::Pointer reader = ReaderType::New();
WriterType::Pointer writer = WriterType::New();
```

The name of the file to be read or written is passed with the SetFileName() method.

```
reader->SetFileName( inputFilename  );
writer->SetFileName( outputFilename );
```

Below we connect the reader, filter and writer to form the data processing pipeline.

```
filter->SetInput( reader->GetOutput() );
rescaler->SetInput( filter->GetOutput() );
writer->SetInput( rescaler->GetOutput() );
```

Finally we execute the pipeline by invoking Update() on the writer. The call is placed in a `try/catch` block in case exceptions are thrown.

```
  try
    {
    writer->Update();
    }
  catch( itk::ExceptionObject & err )
    {
    std::cerr << "ExceptionObject caught !" << std::endl;
    std::cerr << err << std::endl;
    return EXIT_FAILURE;
    }
```

7.9 Reading and Writing Complex Images

The source code for this section can be found in the file
`Examples/IO/ComplexImageReadWrite.cxx`.

This example illustrates how to read and write an image of pixel type `std::complex`. The complex type is defined as an integral part of the C++ language. The characteristics of the type are specified in the C++ standard document in Chapter 26 ”Numerics Library”, page 565, in particular in section 26.2 [5].

We start by including the headers of the complex class, the image, and the reader and writer classes.

```
#include <complex>
#include "itkImage.h"
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
```

The image dimension and pixel type must be declared. In this case we use the `std::complex<>` as the pixel type. Using the dimension and pixel type we proceed to instantiate the image type.

```
  const unsigned int Dimension = 2;

  typedef std::complex< float >    PixelType;
  typedef itk::Image< PixelType, Dimension > ImageType;
```

The image file reader and writer types are instantiated using the image type. We can then create objects for both of them.

```
  typedef itk::ImageFileReader< ImageType > ReaderType;
```

```
typedef itk::ImageFileWriter< ImageType > WriterType;

ReaderType::Pointer reader = ReaderType::New();
WriterType::Pointer writer = WriterType::New();
```

Filenames should be provided for both the reader and the writer. In this particular example we take those filenames from the command line arguments.

```
reader->SetFileName( argv[1] );
writer->SetFileName( argv[2] );
```

Here we simply connect the output of the reader as input to the writer. This simple program could be used for converting complex images from one fileformat to another.

```
writer->SetInput( reader->GetOutput() );
```

The execution of this short pipeline is triggered by invoking the Update() method of the writer. This invocation must be placed inside a try/catch block since its execution may result in exceptions being thrown.

```
try
  {
  writer->Update();
  }
catch( itk::ExceptionObject & err )
  {
  std::cerr << "ExceptionObject caught !" << std::endl;
  std::cerr << err << std::endl;
  return EXIT_FAILURE;
  }
```

For a more interesting use of this code, you may want to add a filter in between the reader and the writer and perform any complex image to complex image operation. A practical application of this code is presented in section 6.10 in the context of Fourier analysis.

7.10 Extracting Components from Vector Images

The source code for this section can be found in the file
`Examples/IO/CovariantVectorImageExtractComponent.cxx`.

This example illustrates how to read an image whose pixel type is `CovariantVector`, extract one of its components to form a scalar image and finally save this image into a file.

The `itk::VectorIndexSelectionCastImageFilter` is used to extract a scalar from the vector image. It is also possible to cast the component type when using this filter. It is the user's responsibility to make sure that the cast will not result in any information loss.

Let's start by including the relevant header files.

```
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
#include "itkVectorIndexSelectionCastImageFilter.h"
#include "itkRescaleIntensityImageFilter.h"
```

We read an image of `itk::CovariantVector` pixels and extract on of its components to generate a scalar image of a consistent pixel type. Then, we rescale the intensities of this scalar image and write it as a image of `unsigned short` pixels.

```
  typedef float                    ComponentType;
  const   unsigned int             Dimension = 2;

  typedef itk::CovariantVector< ComponentType,
                                    Dimension  >      InputPixelType;

  typedef unsigned short                              OutputPixelType;

  typedef itk::Image< InputPixelType,      Dimension >    InputImageType;
  typedef itk::Image< ComponentType,       Dimension >    ComponentImageType;
  typedef itk::Image< OutputPixelType,     Dimension >    OutputImageType;
```

The `itk::ImageFileReader` and `itk::ImageFileWriter` are instantiated using the image types.

```
  typedef itk::ImageFileReader< InputImageType  >  ReaderType;
  typedef itk::ImageFileWriter< OutputImageType >  WriterType;
```

The VectorIndexSelectionCastImageFilter is instantiated using the input and output image types. A filter object is created with the New() method and assigned to a `itk::SmartPointer`.

```
  typedef itk::VectorIndexSelectionCastImageFilter<
                                          InputImageType,
                                          ComponentImageType    > FilterType;

  FilterType::Pointer componentExtractor = FilterType::New();
```

The VectorIndexSelectionCastImageFilter class require us to specify which of the vector components is to be extracted from the vector image. This is done with the SetIndex() method. In this example we obtain this value from the command line arguments.

```
  componentExtractor->SetIndex( indexOfComponentToExtract );
```

The `itk::RescaleIntensityImageFilter` filter is instantiated here.

```
  typedef itk::RescaleIntensityImageFilter<
                                ComponentImageType,
                                OutputImageType >      RescaleFilterType;

  RescaleFilterType::Pointer  rescaler = RescaleFilterType::New();
```

The minimum and maximum values for the output image are specified in the following. Note the use of the `itk::NumericTraits` class which allows to define a number of type-related constant in a generic way. The use of traits is a fundamental characteristic of generic programming [6, 1].

```
  rescaler->SetOutputMinimum( itk::NumericTraits< OutputPixelType >::min() );
  rescaler->SetOutputMaximum( itk::NumericTraits< OutputPixelType >::max() );
```

Below, we create the reader and writer using the New() method and assign the result to a SmartPointer.

```
  ReaderType::Pointer reader = ReaderType::New();
  WriterType::Pointer writer = WriterType::New();
```

The name of the file to be read or written is passed with the SetFileName() method.

```
  reader->SetFileName( inputFilename  );
  writer->SetFileName( outputFilename );
```

Below we connect the reader, filter and writer to form the data processing pipeline.

```
  componentExtractor->SetInput( reader->GetOutput() );
  rescaler->SetInput( componentExtractor->GetOutput() );
  writer->SetInput( rescaler->GetOutput() );
```

Finally we execute the pipeline by invoking Update() on the writer. The call is placed in a `try/catch` block in case exceptions are thrown.

```
  try
    {
    writer->Update();
    }
  catch( itk::ExceptionObject & err )
    {
    std::cerr << "ExceptionObject caught !" << std::endl;
    std::cerr << err << std::endl;
    return EXIT_FAILURE;
    }
```

7.11 Reading and Writing Image Series

It is still quite common to store 3D medical images in sets of files each one containing a single slice of a volume dataset. Those 2D files can be read as individual 2D images, or can be grouped together in order to reconstruct a 3D dataset. The same practice can be extended to higher dimensions, for example, for managing 4D datasets by using sets of files each one containing a 3D image. This practice is common in the domain of cardiac imaging, perfusion, functional MRI and PET. This section illustrates the functionalities available in ITK for dealing with reading and writing series of images.

7.11.1 Reading Image Series

The source code for this section can be found in the file
`Examples/IO/ImageSeriesReadWrite.cxx`.

This example illustrates how to read a series of 2D slices from independent files in order to compose a volume. The class `itk::ImageSeriesReader` is used for this purpose. This class works in combination with a generator of filenames that will provide a list of files to be read. In this particular example we use the `itk::NumericSeriesFileNames` class as filename generator. This generator uses a `printf` style of string format with a "`%d`" field that will be successively replaced by a number specified by the user. Here we will use a format like "`file%03d.png`" for reading PNG files named file001.png, file002.png, file003.png... and so on.

This requires the following headers as shown.

```
#include "itkImage.h"
#include "itkImageSeriesReader.h"
#include "itkImageFileWriter.h"
#include "itkNumericSeriesFileNames.h"
#include "itkPNGImageIO.h"
```

We start by defining the `PixelType` and `ImageType`.

```
  typedef unsigned char                       PixelType;
  const unsigned int Dimension = 3;

  typedef itk::Image< PixelType, Dimension >  ImageType;
```

The image type is used as a template parameter to instantiate the reader and writer.

```
  typedef itk::ImageSeriesReader< ImageType >  ReaderType;
  typedef itk::ImageFileWriter<   ImageType >  WriterType;

  ReaderType::Pointer reader = ReaderType::New();
  WriterType::Pointer writer = WriterType::New();
```

Then, we declare the filenames generator type and create one instance of it.

```
typedef itk::NumericSeriesFileNames    NameGeneratorType;

NameGeneratorType::Pointer nameGenerator = NameGeneratorType::New();
```

The filenames generator requires us to provide a pattern of text for the filenames, and numbers for the initial value, last value and increment to be used for generating the names of the files.

```
nameGenerator->SetSeriesFormat( "vwe%03d.png" );

nameGenerator->SetStartIndex( first );
nameGenerator->SetEndIndex( last );
nameGenerator->SetIncrementIndex( 1 );
```

The ImageIO object that actually performs the read process is now connected to the ImageSeriesReader. This is the safest way of making sure that we use an ImageIO object that is appropriate for the type of files that we want to read.

```
reader->SetImageIO( itk::PNGImageIO::New() );
```

The filenames of the input files must be provided to the reader. While the writer is instructed to write the same volume dataset in a single file.

```
reader->SetFileNames( nameGenerator->GetFileNames()  );

writer->SetFileName( outputFilename );
```

We connect the output of the reader to the input of the writer.

```
writer->SetInput( reader->GetOutput() );
```

Finally, execution of the pipeline can be triggered by invoking the Update() method in the writer. This call must be placed in a try/catch block since exceptions be potentially be thrown in the process of reading or writing the images.

```
try
  {
  writer->Update();
  }
catch( itk::ExceptionObject & err )
  {
  std::cerr << "ExceptionObject caught !" << std::endl;
  std::cerr << err << std::endl;
  return EXIT_FAILURE;
  }
```

7.11.2 Writing Image Series

The source code for this section can be found in the file
`Examples/IO/ImageReadImageSeriesWrite.cxx`.

This example illustrates how to save an image using the `itk::ImageSeriesWriter`. This class enables the saving of a 3D volume as a set of files containing one 2D slice per file.

The type of the input image is declared here and it is used for declaring the type of the reader. This will be a conventional 3D image reader.

```
typedef itk::Image< unsigned char, 3 >      ImageType;
typedef itk::ImageFileReader< ImageType >   ReaderType;
```

The reader object is constructed using the `New()` operator and assigning the result to a `SmartPointer`. The filename of the 3D volume to be read is taken from the command line arguments and passed to the reader using the `SetFileName()` method.

```
ReaderType::Pointer reader = ReaderType::New();
reader->SetFileName( argv[1] );
```

The type of the series writer must be instantiated taking into account that the input file is a 3D volume and the output files are 2D images. Additionally, the output of the reader is connected as input to the writer.

```
typedef itk::Image< unsigned char, 2 >     Image2DType;

typedef itk::ImageSeriesWriter< ImageType, Image2DType > WriterType;

WriterType::Pointer writer = WriterType::New();

writer->SetInput( reader->GetOutput() );
```

The writer requires a list of filenames to be generated. This list can be produced with the help of the `itk::NumericSeriesFileNames` class.

```
typedef itk::NumericSeriesFileNames    NameGeneratorType;

NameGeneratorType::Pointer nameGenerator = NameGeneratorType::New();
```

The `NumericSeriesFileNames` class requires an input string in order to have a template for generating the filenames of all the output slices. Here we compose this string using a prefix taken from the command line arguments and adding the extension for PNG files.

```
  std::string format = argv[2];
  format += "%03d.";
  format += argv[3];   // filename extension

  nameGenerator->SetSeriesFormat( format.c_str() );
```

The input string is going to be used for generating filenames by setting the values of the first and last slice. This can be done by collecting information from the input image. Note that before attempting to take any image information from the reader, its execution must be triggered with the invocation of the `Update()` method, and since this invocation can potentially throw exceptions, it must be put inside a `try/catch` block.

```
  try
    {
    reader->Update();
    }
  catch( itk::ExceptionObject & excp )
    {
    std::cerr << "Exception thrown while reading the image" << std::endl;
    std::cerr << excp << std::endl;
    }
```

Now that the image has been read we can query its largest possible region and recover information about the number of pixels along every dimension.

```
  ImageType::ConstPointer inputImage = reader->GetOutput();
  ImageType::RegionType   region     = inputImage->GetLargestPossibleRegion();
  ImageType::IndexType    start      = region.GetIndex();
  ImageType::SizeType     size       = region.GetSize();
```

With this information we can find the number that will identify the first and last slices of the 3D data set. This numerical values are then passed to the filenames generator object that will compose the names of the files where the slices are going to be stored.

```
  const unsigned int firstSlice = start[2];
  const unsigned int lastSlice  = start[2] + size[2] - 1;

  nameGenerator->SetStartIndex( firstSlice );
  nameGenerator->SetEndIndex( lastSlice );
  nameGenerator->SetIncrementIndex( 1 );
```

The list of filenames is taken from the names generator and it is passed to the series writer.

```
  writer->SetFileNames( nameGenerator->GetFileNames() );
```

Finally we trigger the execution of the pipeline with the Update() method on the writer. At this point the slices of the image will be saved in individual files containing a single slice per file. The filenames used for these slices are those produced by the filenames generator.

```
  try
    {
    writer->Update();
    }
  catch( itk::ExceptionObject & excp )
    {
    std::cerr << "Exception thrown while reading the image" << std::endl;
    std::cerr << excp << std::endl;
    }
```

Note that by saving data into isolated slices we are losing information that may be significant for medical applications, such as the interslice spacing in millimeters.

7.11.3 Reading and Writing Series of RGB Images

The source code for this section can be found in the file
`Examples/IO/RGBImageSeriesReadWrite.cxx`.

RGB images are commonly used for representing data acquired from cryogenic sections, optical microscopy and endoscopy. This example illustrates how to read RGB color images from a set of files containing individual 2D slices in order to compose a 3D color dataset. Then save it into a single 3D file, and finally save it again as a set of 2D slices with other names.

This requires the following headers as shown.

```
#include "itkRGBPixel.h"
#include "itkImage.h"
#include "itkImageFileWriter.h"
#include "itkImageSeriesReader.h"
#include "itkImageSeriesWriter.h"
#include "itkNumericSeriesFileNames.h"
#include "itkPNGImageIO.h"
```

The `itk::RGBPixel` class is templated over the type used to represent each one of the Red, Green and Blue components. A typical instantiation of the RGB image class might be as follows.

```
  typedef itk::RGBPixel< unsigned char >        PixelType;
  const unsigned int Dimension = 3;

  typedef itk::Image< PixelType, Dimension >    ImageType;
```

The image type is used as a template parameter to instantiate the series reader and the volumetric writer.

```
typedef itk::ImageSeriesReader< ImageType >  SeriesReaderType;
typedef itk::ImageFileWriter<   ImageType >  WriterType;

SeriesReaderType::Pointer seriesReader = SeriesReaderType::New();
WriterType::Pointer       writer       = WriterType::New();
```

We use a NumericSeriesFileNames class in order to generate the filenames of the slices to be read. Later on in this example we will reuse this object in order to generate the filenames of the slices to be written.

```
typedef itk::NumericSeriesFileNames    NameGeneratorType;

NameGeneratorType::Pointer nameGenerator = NameGeneratorType::New();

nameGenerator->SetStartIndex( first );
nameGenerator->SetEndIndex( last );
nameGenerator->SetIncrementIndex( 1 );

nameGenerator->SetSeriesFormat( "vwe%03d.png" );
```

The ImageIO object that actually performs the read process is now connected to the ImageSeriesReader.

```
seriesReader->SetImageIO( itk::PNGImageIO::New() );
```

The filenames of the input slices are taken from the names generator and passed to the series reader.

```
seriesReader->SetFileNames( nameGenerator->GetFileNames()  );
```

The name of the volumetric output image is passed to the image writer, and we connect the output of the series reader to the input of the volumetric writer.

```
writer->SetFileName( outputFilename );

writer->SetInput( seriesReader->GetOutput() );
```

Finally, execution of the pipeline can be triggered by invoking the Update() method in the volumetric writer. This, of course, is done from inside a try/catch block.

```
try
  {
  writer->Update();
  }
catch( itk::ExceptionObject & excp )
  {
  std::cerr << "Error reading the series " << std::endl;
  std::cerr << excp << std::endl;
  }
```

We now proceed to save the same volumetric dataset as a set of slices. This is done only to illustrate the process for saving a volume as a series of 2D individual datasets. The type of the series writer must be instantiated taking into account that the input file is a 3D volume and the output files are 2D images. Additionally, the output of the series reader is connected as input to the series writer.

```
typedef itk::Image< PixelType, 2 >     Image2DType;

typedef itk::ImageSeriesWriter< ImageType, Image2DType > SeriesWriterType;

SeriesWriterType::Pointer seriesWriter = SeriesWriterType::New();

seriesWriter->SetInput( seriesReader->GetOutput() );
```

We now reuse the filenames generator in order to produce the list of filenames for the output series. In this case we just need to modify the format of the filenames generator. Then, we pass the list of output filenames to the series writer.

```
nameGenerator->SetSeriesFormat( "output%03d.png" );

seriesWriter->SetFileNames( nameGenerator->GetFileNames() );
```

Finally we trigger the execution of the series writer from inside a try/catch block.

```
try
  {
  seriesWriter->Update();
  }
catch( itk::ExceptionObject & excp )
  {
  std::cerr << "Error reading the series " << std::endl;
  std::cerr << excp << std::endl;
  }
```

You may have noticed that apart from the declaration of the `PixelType` there is nothing in this code that is specific for RGB images. All the actions required to support color images are implemented internally in the `itk::ImageIO` objects.

7.12 Reading and Writing DICOM Images

7.12.1 Foreword

With the introduction of computed tomography (CT) followed by other digital diagnostic imaging modalities such as MRI in the 1970's, and the increasing use of computers in clinical applications, the American College of Radiology (ACR)[3] and the National Electrical Manufacturers Association (NEMA)[4] recognized the need for a standard method for transferring images as well as associated information between devices manufactured from various vendors.

ACR and NEMA formed a joint committee to develop a standard for Digital Imaging and Communications in Medicine (DICOM). This standard was developed in liaison with other Standardization Organizations such as CEN TC251, JIRA including IEEE, HL7 and ANSI USA as reviewers.

DICOM is a comprehensive set of standards for handling, storing and transmitting information in medical imaging. The DICOM standard was developed based on the previous NEMA specification. The standard specifies a file format definition as well as a network communication protocol. DICOM was developed to enable integration of scanners, servers, workstations and network hardware from multiple vendors into an image archiving and communication system.

DICOM files consist of a header and a body of image data. The header contains standardized as well as free-form fields. The set of standardized fields is called the public DICOM dictionary, an instance of this dictionary is available in ITK in the file `Insight/Utilities/gdcm/Dict/dicomV3.dic`. The list of free-form fields is also called the *shadow dictionary*.

A single DICOM file can contain multiples frames, allowing storage of volumes or animations. Image data can be compressed using a large variety of standards, including JPEG (both lossy and lossless), LZW (Lempel Ziv Welch), and RLE (Run-length encoding).

The DICOM Standard is an evolving standard and it is maintained in accordance with the Procedures of the DICOM Standards Committee. Proposals for enhancements are forthcoming from the DICOM Committee member organizations based on input from users of the Standard. These proposals are considered for inclusion in future editions of the Standard. A requirement in updating the Standard is to maintain effective compatibility with previous editions.

For a more detailed description of the DICOM standard see [60].

The following sections illustrate how to use the functionalities that ITK provides for reading and writing DICOM files. This is extremely important in the domain of medical imaging since most of the images that are acquired a clinical setting are stored and transported using the DICOM standard.

DICOM functionalities in ITK are provided by the GDCM library. This open source library

[3] `http://www.acr.org`
[4] `http://www.nema.org`

was developed by the CREATIS Team [5] at INSA-Lyon [26]. Although originally this library was distributed under a LGPL License[6], the CREATIS Team was lucid enough to understand the limitations of that license and agreed to adopt the more open BSD-like License[7] that is used by ITK. This change in their licensing made possible to distribute GDCM along with ITK.

GDCM is still being maintained and improved at the original CREATIS site and the version distributed with ITK gets updated with major releases of the GDCM library.

7.12.2 Reading and Writing a 2D Image

The source code for this section can be found in the file
`Examples/IO/DicomImageReadWrite.cxx`.

This example illustrates how to read a single DICOM slice and write it back as another DICOM slice. In the process an intensity rescaling is also applied.

In order to read and write the slice we use here the `itk::GDCMImageIO` class that encapsulates a connection to the underlying GDCM library. In this way we gain access from ITK to the DICOM functionalities offered by GDCM. The GDCMImageIO object is connected as the ImageIO object to be used by the `itk::ImageFileWriter`.

We should first include the following header files.

```
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
#include "itkRescaleIntensityImageFilter.h"
#include "itkGDCMImageIO.h"
```

Then we declare the pixel type and image dimension, and use them for instantiating the image type to be read.

```
  typedef signed short InputPixelType;
  const unsigned int   InputDimension = 2;

  typedef itk::Image< InputPixelType, InputDimension > InputImageType;
```

With the image type we can instantiate the type of the reader, create one, and set the filename of the image to be read.

```
  typedef itk::ImageFileReader< InputImageType > ReaderType;

  ReaderType::Pointer reader = ReaderType::New();
  reader->SetFileName( argv[1] );
```

[5]http://www.creatis.insa-lyon.fr
[6]http://www.gnu.org/copyleft/lesser.html
[7]http://www.opensource.org/licenses/bsd-license.php

GDCMImageIO is an ImageIO class for reading and writing DICOM v3 and ACR/NEMA images. The GDCMImageIO object is constructed here and connected to the ImageFileReader.

```
typedef itk::GDCMImageIO           ImageIOType;

ImageIOType::Pointer gdcmImageIO = ImageIOType::New();

reader->SetImageIO( gdcmImageIO );
```

At this point we can trigger the reading process by invoking the Update() method. Since this reading process may eventually throw an exception, we place the invocation inside a try/catch block.

```
try
  {
  reader->Update();
  }
catch (itk::ExceptionObject & e)
  {
  std::cerr << "exception in file reader " << std::endl;
  std::cerr << e << std::endl;
  return EXIT_FAILURE;
  }
```

We have now the image in memory and can get access to it by using the GetOutput() method of the reader. In the remaining of this current example, we focus on showing how we can save this image again in DICOM format in a new file.

First, we must instantiate an ImageFileWriter type. Then, we construct one, set the filename to be used for writing and connect the input image to be written. Given that in this example we write the image in different ways, and in each case we use a different writer, we enumerated here the variable names of the writer objects as well as their types.

```
typedef itk::ImageFileWriter< InputImageType >  Writer1Type;

Writer1Type::Pointer writer1 = Writer1Type::New();

writer1->SetFileName( argv[2] );
writer1->SetInput( reader->GetOutput() );
```

We need to explicitly set the proper image IO (GDCMImageIO) to the writer filter since the input DICOM dictionary is being passed along the writing process. The dictionary contains all necessary information that a valid DICOM file should contain, like Patient Name, Patient ID, Institution Name, etc.

```
writer1->SetImageIO( gdcmImageIO );
```

The writing process is triggered by invoking the Update() method. Since this execution may result in exceptions being thrown we place the Update() call inside a try/catch block.

```
try
  {
  writer1->Update();
  }
catch (itk::ExceptionObject & e)
  {
  std::cerr << "exception in file writer " << std::endl;
  std::cerr << e << std::endl;
  return EXIT_FAILURE;
  }
```

We will now rescale the image into a rescaled image one using the rescale intensity image filter. For this purpose we use a better suited pixel type: `unsigned char` instead of `signed short`. The minimum and maximum values of the output image are explicitly defined in the rescaling filter.

```
typedef unsigned char WritePixelType;

typedef itk::Image< WritePixelType, 2 > WriteImageType;

typedef itk::RescaleIntensityImageFilter<
             InputImageType, WriteImageType > RescaleFilterType;

RescaleFilterType::Pointer rescaler = RescaleFilterType::New();

rescaler->SetOutputMinimum(   0 );
rescaler->SetOutputMaximum( 255 );
```

We create a second writer object that will save the rescaled image into a file. This time not in DICOM format. This is done only for the sake of verifying the image against the one that will be saved in DICOM format later on this example.

```
typedef itk::ImageFileWriter< WriteImageType >  Writer2Type;

Writer2Type::Pointer writer2 = Writer2Type::New();

writer2->SetFileName( argv[3] );

rescaler->SetInput( reader->GetOutput() );
writer2->SetInput( rescaler->GetOutput() );
```

The writer can be executed by invoking the Update() method from inside a try/catch block.

We proceed now to save the same rescaled image into a file in DICOM format. For this purpose we just need to set up a `itk::ImageFileWriter` and pass to it the rescaled image as input.

```
typedef itk::ImageFileWriter< WriteImageType >  Writer3Type;

Writer3Type::Pointer writer3 = Writer3Type::New();

writer3->SetFileName( argv[4] );
writer3->SetInput( rescaler->GetOutput() );
```

We now need to explicitly set the proper image IO (GDCMImageIO), but also we must tell the ImageFileWriter to not use the MetaDataDictionary from the input but from the GDCMImageIO since this is the one that contains the DICOM specific information

The GDCMImageIO object will automatically detect the pixel type, in this case `unsigned char` and it will update the DICOM header information accordingly.

```
writer3->UseInputMetaDataDictionaryOff ();
writer3->SetImageIO( gdcmImageIO );
```

Finally we trigger the execution of the DICOM writer by invoking the Update() method from inside a try/catch block.

```
try
  {
  writer3->Update();
  }
catch (itk::ExceptionObject & e)
  {
  std::cerr << "Exception in file writer " << std::endl;
  std::cerr << e << std::endl;
  return EXIT_FAILURE;
  }
```

7.12.3 Reading a 2D DICOM Series and Writing a Volume

The source code for this section can be found in the file
`Examples/IO/DicomSeriesReadImageWrite2.cxx`.

Probably the most common representation of datasets in clinical applications is the one that uses sets of DICOM slices in order to compose tridimensional images. This is the case for CT, MRI and PET scanners. It is very common therefore for image analysts to have to process volumetric images that are stored in the form of a set of DICOM files belonging to a common DICOM series.

The following example illustrates how to use ITK functionalities in order to read a DICOM series into a volume and then save this volume in another file format.

The example begins by including the appropriate headers. In particular we will need the `itk::GDCMImageIO` object in order to have access to the capabilities of the GDCM library for reading DICOM files, and the `itk::GDCMSeriesFileNames` object for generating the lists of filenames identifying the slices of a common volumetric dataset.

```
#include "itkGDCMImageIO.h"
#include "itkGDCMSeriesFileNames.h"
#include "itkImageSeriesReader.h"
#include "itkImageFileWriter.h"
```

We define the pixel type and dimension of the image to be read. In this particular case, the dimensionality of the image is 3, and we assume a `signed short` pixel type that is commonly used for X-Rays CT scanners.

```
  typedef signed short    PixelType;
  const unsigned int      Dimension = 3;

  typedef itk::Image< PixelType, Dimension >         ImageType;
```

We use the image type for instantiating the type of the series reader and for constructing one object of its type.

```
  typedef itk::ImageSeriesReader< ImageType >        ReaderType;
  ReaderType::Pointer reader = ReaderType::New();
```

A GDCMImageIO object is created and connected to the reader. This object is the one that is aware of the internal intricacies of the DICOM format.

```
  typedef itk::GDCMImageIO       ImageIOType;
  ImageIOType::Pointer dicomIO = ImageIOType::New();

  reader->SetImageIO( dicomIO );
```

Now we face one of the main challenges of the process of reading a DICOM series. That is, to identify from a given directory the set of filenames that belong together to the same volumetric image. Fortunately for us, GDCM offers functionalities for solving this problem and we just need to invoke those functionalities through an ITK class that encapsulates a communication with GDCM classes. This ITK object is the GDCMSeriesFileNames. Conveniently for us, we only need to pass to this class the name of the directory where the DICOM slices are stored. This is done with the `SetDirectory()` method. The GDCMSeriesFileNames object will explore the directory and will generate a sequence of filenames for DICOM files for one study/series. In this example, we also call the `SetUseSeriesDetails(true)` function that tells the GDCM-SereiesFileNames object to use additional DICOM information to distinguish unique volumes within the directory. This is useful, for example, if a DICOM device assigns the same SeriesID to a scout scan and its 3D volume; by using additional DICOM information the scout scan will not be included as part of the 3D volume. Note that `SetUseSeriesDetails(true)` must be called prior to calling `SetDirectory()`.

```
typedef itk::GDCMSeriesFileNames NamesGeneratorType;
NamesGeneratorType::Pointer nameGenerator = NamesGeneratorType::New();

nameGenerator->SetUseSeriesDetails( true );

nameGenerator->SetDirectory( argv[1] );
```

The GDCMSeriesFileNames object first identifies the list of DICOM series that are present in the given directory. We receive that list in a reference to a container of strings and then we can do things like printing out all the series identifiers that the generator had found. Since the process of finding the series identifiers can potentially throw exceptions, it is wise to put this code inside a try/catch block.

```
  typedef std::vector< std::string >    SeriesIdContainer;

  const SeriesIdContainer & seriesUID = nameGenerator->GetSeriesUIDs();

  SeriesIdContainer::const_iterator seriesItr = seriesUID.begin();
  SeriesIdContainer::const_iterator seriesEnd = seriesUID.end();
  while( seriesItr != seriesEnd )
    {
    std::cout << seriesItr->c_str() << std::endl;
    seriesItr++;
    }
```

Given that it is common to find multiple DICOM series in the same directory, we must tell the GDCM classes what specific series do we want to read. In this example we do this by checking first if the user has provided a series identifier in the command line arguments. If no series identifier has been passed, then we simply use the first series found during the exploration of the directory.

```
  std::string seriesIdentifier;

  if( argc > 3 ) // If no optional series identifier
    {
    seriesIdentifier = argv[3];
    }
  else
```

```
    {
    seriesIdentifier = seriesUID.begin()->c_str();
    }
```

We pass the series identifier to the name generator and ask for all the filenames associated to that series. This list is returned in a container of strings by the `GetFileNames()` method.

```
typedef std::vector< std::string >   FileNamesContainer;
FileNamesContainer fileNames;

fileNames = nameGenerator->GetFileNames( seriesIdentifier );
```

The list of filenames can now be passed to the `itk::ImageSeriesReader` using the `SetFileNames()` method.

```
reader->SetFileNames( fileNames );
```

Finally we can trigger the reading process by invoking the `Update()` method in the reader. This call as usual is placed inside a `try/catch` block.

```
try
  {
  reader->Update();
  }
catch (itk::ExceptionObject &ex)
  {
  std::cout << ex << std::endl;
  return EXIT_FAILURE;
  }
```

At this point, we have a volumetric image in memory that we can access by invoking the `GetOutput()` method of the reader.

We proceed now to save the volumetric image in another file, as specified by the user in the command line arguments of this program. Thanks to the ImageIO factory mechanism, only the filename extension is needed to identify the file format in this case.

```
typedef itk::ImageFileWriter< ImageType > WriterType;
WriterType::Pointer writer = WriterType::New();

writer->SetFileName( argv[2] );

writer->SetInput( reader->GetOutput() );
```

The process of writing the image is initiated by invoking the `Update()` method of the writer.

```
    writer->Update();
```

Note that in addition to writing the volumetric image to a file we could have used it as the input for any 3D processing pipeline. Keep in mind that DICOM is simply a file format and a network protocol. Once the image data has been loaded into memory, it behaves as any other volumetric dataset that you could have loaded from any other file format.

7.12.4 Reading a 2D DICOM Series and Writing a 2D DICOM Series

The source code for this section can be found in the file
`Examples/IO/DicomSeriesReadSeriesWrite.cxx`.

This example illustrates how to read a DICOM series into a volume and then save this volume into another DICOM series using the exact same header information. It makes use of the GDCM library.

The main purpose of this example is to show how to properly propagate the DICOM specific information along the pipeline to be able to correctly write back the image using the information from the input DICOM files.

Please note that writing DICOM files is quite a delicate operation since we are dealing with a significant amount of patient specific data. It is your responsibility to verify that the DICOM headers generated from this code are not introducing risks in the diagnosis or treatment of patients. It is as well your responsibility to make sure that the privacy of the patient is respected when you process data sets that contain personal information. Privacy issues are regulated in the United States by the HIPAA norms[8]. You would probably find similar legislation in every country.

When saving datasets in DICOM format it must be made clear whether this datasets have been processed in any way, and if so, you should inform the recipients of the data about the purpose and potential consequences of the processing. This is fundamental if the datasets are intended to be used for diagnosis, treatment or follow-up of patients. For example, the simple reduction of a dataset form a 16-bits/pixel to a 8-bits/pixel representation may make impossible to detect certain pathologies and as a result will expose the patient to the risk or remaining untreated for a long period of time while her/his pathology progresses.

You are strongly encouraged to get familiar with the report on medical errors "To Err is Human", produced by the U.S. Institute of Medicine [46]. Raising awareness about the high frequency of medical errors is a first step in reducing their occurrence.

After all these warnings, let us now go back to the code and get familiar with the use of ITK and GDCM for writing DICOM Series. The first step that we must take is to include the header files of the relevant classes. We include the GDCM image IO class, the GDCM filenames generator, the series reader and writer.

[8]The Health Insurance Portability and Accountability Act of 1996. `http://www.cms.hhs.gov/hipaa/`

```
#include "itkGDCMImageIO.h"
#include "itkGDCMSeriesFileNames.h"
#include "itkImageSeriesReader.h"
#include "itkImageSeriesWriter.h"
```

As a second step, we define the image type to be used in this example. This is done by explicitly selecting a pixel type and a dimension. Using the image type we can define the type of the series reader.

```
  typedef signed short    PixelType;
  const unsigned int      Dimension = 3;

  typedef itk::Image< PixelType, Dimension >      ImageType;
  typedef itk::ImageSeriesReader< ImageType >     ReaderType;
```

We also declare types for the itk::GDCMImageIO object that will actually read and write the DICOM images, and the itk::GDCMSeriesFileNames object that will generate and order all the filenames for the slices composing the volume dataset. Once we have the types, we proceed to create instances of both objects.

```
  typedef itk::GDCMImageIO                        ImageIOType;
  typedef itk::GDCMSeriesFileNames                NamesGeneratorType;

  ImageIOType::Pointer gdcmIO = ImageIOType::New();
  NamesGeneratorType::Pointer namesGenerator = NamesGeneratorType::New();
```

Just as the previous example, we get the DICOM filenames from the directory. Note however, that in this case we use the `SetInputDirectory()` method instead of the `SetDirectory()`. This is done because in the present case we will use the filenames generator for producing both the filenames for reading and the filenames for writing. Then, we invoke the `GetInputFileNames()` method in order to get the list of filenames to read.

```
  namesGenerator->SetInputDirectory( argv[1] );

  const ReaderType::FileNamesContainer & filenames =
                            namesGenerator->GetInputFileNames();
```

We construct one instance of the series reader object. Set the DICOM image IO object to be use with it, and set the list of filenames to read.

```
  ReaderType::Pointer reader = ReaderType::New();

  reader->SetImageIO( gdcmIO );
  reader->SetFileNames( filenames );
```

We can trigger the reading process by calling the `Update()` method on the series reader. It is wise to put this invocation inside a `try/catch` block since the process may eventually throw exceptions.

```
    reader->Update();
```

At this point we would have the volumetric data loaded in memory and we can get access to it by invoking the `GetOutput()` method in the reader.

Now we can prepare the process for writing the dataset. First, we take the name of the output directory from the command line arguments.

```
  const char * outputDirectory = argv[2];
```

Second, we make sure the output directory exist, using the cross platform tools: itksys::SystemTools. In this case we select to create the directory if it does not exist yet.

```
  itksys::SystemTools::MakeDirectory( outputDirectory );
```

We instantiate explicitly the image type to be used for writing, and use the image type for instantiating the type of the series writer.

```
  typedef signed short    OutputPixelType;
  const unsigned int      OutputDimension = 2;

  typedef itk::Image< OutputPixelType, OutputDimension >    Image2DType;

  typedef itk::ImageSeriesWriter<
                             ImageType, Image2DType >  SeriesWriterType;
```

We construct a series writer and connect to its input the output from the reader. Then we pass the GDCM image IO object in order to be able to write the images in DICOM format.

```
  SeriesWriterType::Pointer seriesWriter = SeriesWriterType::New();

  seriesWriter->SetInput( reader->GetOutput() );
  seriesWriter->SetImageIO( gdcmIO );
```

It is time now to setup the GDCMSeriesFileNames to generate new filenames using another output directory. Then simply pass those newly generated files to the series writer.

```
  namesGenerator->SetOutputDirectory( outputDirectory );

  seriesWriter->SetFileNames( namesGenerator->GetOutputFileNames() );
```

The following line of code is extremely important for this process to work correctly. The line is taking the MetaDataDictionary from the input reader and passing it to the output writer. The reason why this step is so important is that the MetaDataDictionary contains all the entries of the input DICOM header.

```
  seriesWriter->SetMetaDataDictionaryArray(
                        reader->GetMetaDataDictionaryArray() );
```

Finally we trigger the writing process by invoking the `Update()` method in the series writer. We place this call inside a try/catch block, in case any exception is thrown during the writing process.

```
  try
    {
    seriesWriter->Update();
    }
  catch( itk::ExceptionObject & excp )
    {
    std::cerr << "Exception thrown while writing the series " << std::endl;
    std::cerr << excp << std::endl;
    return EXIT_FAILURE;
    }
```

Please keep in mind that you should avoid to generate DICOM files that have the appearance of being produced by a scanner. It should be clear from the directory or filenames that this data was the result of the execution of some sort of algorithm. This will help to prevent your dataset from being used as scanner data by accident.

7.12.5 Printing DICOM Tags From One Slice

The source code for this section can be found in the file
`Examples/IO/DicomImageReadPrintTags.cxx`.

It is often valuable to be able to query the entries from the header of a DICOM file. This can be used for checking for consistency, or simply for verifying that we have the correct dataset in our hands. This example illustrates how to read a DICOM file and then print out most of the DICOM header information. The binary fields of the DICOM header are skipped.

The headers of the main classes involved in this example are specified below. They include the image file reader, the GDCM image IO object, the Meta data dictionary and its entry element the Meta data object.

```
#include "itkImageFileReader.h"
#include "itkGDCMImageIO.h"
#include "itkMetaDataDictionary.h"
#include "itkMetaDataObject.h"
```

We instantiate the type to be used for storing the image once it is read into memory.

```
typedef signed short       PixelType;
const unsigned int         Dimension = 2;

typedef itk::Image< PixelType, Dimension >      ImageType;
```

Using the image type as template parameter we instantiate the type of the image file reader and construct one instance of it.

```
typedef itk::ImageFileReader< ImageType >     ReaderType;

ReaderType::Pointer reader = ReaderType::New();
```

The GDCM image IO type is declared and used for constructing one image IO object.

```
typedef itk::GDCMImageIO       ImageIOType;
ImageIOType::Pointer dicomIO = ImageIOType::New();
```

We pass to the reader the filename of the image to be read and connect the ImageIO object to it too.

```
reader->SetFileName( argv[1] );
reader->SetImageIO( dicomIO );
```

The reading process is triggered with a call to the `Update()` method. This call should be placed inside a `try/catch` block because its execution may result in exceptions being thrown.

```
  reader->Update();
```

Now that the image has been read, we obtain the Meta data dictionary from the ImageIO object using the `GetMetaDataDictionary()` method.

```
typedef itk::MetaDataDictionary   DictionaryType;

const  DictionaryType & dictionary = dicomIO->GetMetaDataDictionary();
```

Since we are interested only in the DICOM tags that can be expressed in strings, we declare a MetaDataObject suitable for managing strings.

```
typedef itk::MetaDataObject< std::string > MetaDataStringType;
```

We instantiate the iterators that will make possible to walk through all the entries of the MetaDataDictionary.

```
  DictionaryType::ConstIterator itr = dictionary.Begin();
  DictionaryType::ConstIterator end = dictionary.End();
```

For each one of the entries in the dictionary, we check first if its element can be converted to a string, a `dynamic_cast` is used for this purpose.

```
  while( itr != end )
    {
    itk::MetaDataObjectBase::Pointer  entry = itr->second;

    MetaDataStringType::Pointer entryvalue =
      dynamic_cast<MetaDataStringType *>( entry.GetPointer() ) ;
```

For those entries that can be converted, we take their DICOM tag and pass it to the `GetLabelFromTag()` method of the GDCMImageIO class. This method checks the DICOM dictionary and returns the string label associated to the tag that we are providing in the `tagkey` variable. If the label is found, it is returned in `labelId` variable. The method itself return false if the tagkey is not found in the dictionary. For example ”0010—0010” in `tagkey` becomes ”Patient’s Name” in `labelId`.

```
    if( entryvalue )
      {
      std::string tagkey   = itr->first;
      std::string labelId;
      bool found =  itk::GDCMImageIO::GetLabelFromTag( tagkey, labelId );
```

The actual value of the dictionary entry is obtained as a string with the `GetMetaDataObjectValue()` method.

```
      std::string tagvalue = entryvalue->GetMetaDataObjectValue();
```

At this point we can print out an entry by concatenating the DICOM Name or label, the numeric tag and its actual value.

```
      if( found )
        {
        std::cout << "(" << tagkey << ") " << labelId;
        std::cout << " = " << tagvalue.c_str() << std::endl;
        }
```

Finally we just close the loop that will walk through all the Dictionary entries.

```
    ++itr;
    }
```

It is also possible to read a specific tag. In that case the string of the entry can be used for querying the MetaDataDictionary.

```
    std::string entryId = "0010|0010";
    DictionaryType::ConstIterator tagItr = dictionary.Find( entryId );
```

If the entry is actually found in the Dictionary, then we can attempt to convert it to a string entry by using a `dynamic_cast`.

```
    if( tagItr != end )
      {
      MetaDataStringType::ConstPointer entryvalue =
        dynamic_cast<const MetaDataStringType *>(
                                   tagItr->second.GetPointer() );
```

If the dynamic cast succeed, then we can print out the values of the label, the tag and the actual value.

```
      if( entryvalue )
        {
        std::string tagvalue = entryvalue->GetMetaDataObjectValue();
        std::cout << "Patient's Name (" << entryId <<  ") ";
        std::cout << " is: " << tagvalue << std::endl;
        }
```

For a full description of the DICOM dictionary please look at the file.

`Insight/Utilities/gdcm/Dicts/dicomV3.dic`

7.12.6 Printing DICOM Tags From a Series

The source code for this section can be found in the file
`Examples/IO/DicomSeriesReadPrintTags.cxx`.

This example illustrates how to read a DICOM series into a volume and then print most of the DICOM header information. The binary fields are skipped.

The header files for the series reader and the GDCM classes for image IO and name generation should be included first.

```
#include "itkImageSeriesReader.h"
#include "itkGDCMImageIO.h"
#include "itkGDCMSeriesFileNames.h"
```

We instantiate then the type to be used for storing the image once it is read into memory.

```
typedef signed short       PixelType;
const unsigned int         Dimension = 3;

typedef itk::Image< PixelType, Dimension >      ImageType;
```

We use the image type for instantiating the series reader type and then we construct one object of this class.

```
typedef itk::ImageSeriesReader< ImageType >     ReaderType;

ReaderType::Pointer reader = ReaderType::New();
```

A GDCMImageIO object is created and assigned to the reader.

```
typedef itk::GDCMImageIO       ImageIOType;

ImageIOType::Pointer dicomIO = ImageIOType::New();

reader->SetImageIO( dicomIO );
```

A GDCMSeriesFileNames is declared in order to generate the names of DICOM slices. We specify the directory with the `SetInputDirectory()` method and, in this case, take the directory name from the command line arguments. You could have obtained the directory name from a file dialog in a GUI.

```
typedef itk::GDCMSeriesFileNames     NamesGeneratorType;

NamesGeneratorType::Pointer nameGenerator = NamesGeneratorType::New();

nameGenerator->SetInputDirectory( argv[1] );
```

The list of files to read is obtained from the name generator by invoking the `GetInputFileNames()` method and receiving the results in a container of strings. The list of filenames is passed to the reader using the `SetFileNames()` method.

```
typedef std::vector<std::string>    FileNamesContainer;
FileNamesContainer fileNames = nameGenerator->GetInputFileNames();

reader->SetFileNames( fileNames );
```

We trigger the reader by invoking the `Update()` method. This invocation should normally be done inside a `try/catch` block given that it may eventually throw exceptions.

```
    reader->Update();
```

ITK internally queries GDCM and obtain all the DICOM tags from the file headers. The tag values are stored in the itk::MetaDataDictionary that is a general purpose container for {key,value} pairs. The Meta data dictionary can be recovered from any ImageIO class by invoking the GetMetaDataDictionary() method.

```
  typedef itk::MetaDataDictionary   DictionaryType;

  const  DictionaryType & dictionary = dicomIO->GetMetaDataDictionary();
```

In this example, we are only interested in the DICOM tags that can be represented as strings. We declare therefore a itk::MetaDataObject of string type in order to receive those particular values.

```
  typedef itk::MetaDataObject< std::string > MetaDataStringType;
```

The Meta data dictionary is organized as a container with its corresponding iterators. We can therefore visit all its entries by first getting access to its Begin() and End() methods.

```
  DictionaryType::ConstIterator itr = dictionary.Begin();
  DictionaryType::ConstIterator end = dictionary.End();
```

We are now ready for walking through the list of DICOM tags. For this purpose we use the iterators that we just declared. At every entry we attempt to convert it in to a string entry by using the dynamic_cast based on RTTI information[9]. The dictionary is organized like a std::map structure, we should use therefore the first and second members of every entry in order to get access to the {key,value} pairs.

```
  while( itr != end )
    {
    itk::MetaDataObjectBase::Pointer  entry = itr->second;

    MetaDataStringType::Pointer entryvalue =
      dynamic_cast<MetaDataStringType *>( entry.GetPointer() ) ;

    if( entryvalue )
      {
      std::string tagkey   = itr->first;
      std::string tagvalue = entryvalue->GetMetaDataObjectValue();
      std::cout << tagkey <<  " = " << tagvalue << std::endl;
      }

    ++itr;
    }
```

[9]Run Time Type Information

It is also possible to query for specific entries instead of reading all of them as we did above. In this case, the user must provide the tag identifier using the standard DICOM encoding. The identifier is stored in a string and used as key on the dictionary.

```
std::string entryId = "0010|0010";

DictionaryType::ConstIterator tagItr = dictionary.Find( entryId );

if( tagItr == end )
  {
  std::cerr << "Tag " << entryId;
  std::cerr << " not found in the DICOM header" << std::endl;
  }
```

Since the entry may or may not be of string type we must again use a `dynamic_cast` in order to attempt to convert it to a string dictionary entry. If the conversion is successful, then we can print out its content.

```
MetaDataStringType::ConstPointer entryvalue =
  dynamic_cast<const MetaDataStringType *>( tagItr->second.GetPointer() );

if( entryvalue )
  {
  std::string tagvalue = entryvalue->GetMetaDataObjectValue();
  std::cout << "Patient's Name (" << entryId <<  ") ";
  std::cout << " is: " << tagvalue << std::endl;
  }
```

This type of functionality will probably be more useful when provided through a graphical user interface. For a full description of the DICOM dictionary please look at the file

`Insight/Utilities/gdcm/Dicts/dicomV3.dic`

7.12.7 Changing a DICOM Header

The source code for this section can be found in the file
`Examples/IO/DicomImageReadChangeHeaderWrite.cxx`.

This example illustrates how to read a single DICOM slice and write it back with some changed header information as another DICOM slice. Header Key/Value pairs can be specified on the command line. The keys are defined in the file

`Insight/Utilities/gdcm/Dicts/dicomV3.dic`

Please note that modifying the content of a DICOM header is a very risky operation. The Header contains fundamental information about the patient and therefore its consistency must be protected from any data corruption. Before attempting to modify the DICOM headers of your files, you must make sure that you have a very good reason for doing so, and that you can ensure that this information change will not result in a lower quality of health care to be delivered to the patient.

We must start by including the relevant header files. Here we include the image reader, image writer, the image, the Meta data dictionary and its entries the Meta data objects and the GDCMImageIO. The Meta data dictionary is the data container that stores all the entries from the DICOM header once the DICOM image file is read into an ITK image.

```
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
#include "itkImage.h"
#include "itkMetaDataDictionary.h"
#include "itkMetaDataObject.h"
#include "itkGDCMImageIO.h"
```

We declare the image type by selecting a particular pixel type and image dimension.

```
  typedef signed short InputPixelType;
  const unsigned int   Dimension = 2;
  typedef itk::Image< InputPixelType, Dimension > InputImageType;
```

We instantiate the reader type by using the image type as template parameter. An instance of the reader is created and the file name to be read is taken from the command line arguments.

```
  typedef itk::ImageFileReader< InputImageType > ReaderType;
  ReaderType::Pointer reader = ReaderType::New();
  reader->SetFileName( argv[1] );
```

The GDCMImageIO object is created in order to provide the services for reading and writing DICOM files. The newly created image IO class is connected to the reader.

```
  typedef itk::GDCMImageIO           ImageIOType;
  ImageIOType::Pointer gdcmImageIO = ImageIOType::New();
  reader->SetImageIO( gdcmImageIO );
```

The reading of the image is triggered by invoking `Update()` in the reader.

```
    reader->Update();
```

We take the Meta data dictionary from the image that the reader had loaded in memory.

```
InputImageType::Pointer inputImage = reader->GetOutput();
typedef itk::MetaDataDictionary   DictionaryType;
DictionaryType & dictionary = inputImage->GetMetaDataDictionary();
```

Now we access the entries in the Meta data dictionary, and for particular key values we assign a new content to the entry. This is done here by taking {key,value} pairs from the command line arguments. The relevant method is the EncapsulateMetaData that takes the dictionary and for a given key provided by `entryId`, replaces the current value with the content of the `value` variable. This is repeated for every potential pair present in the command line arguments.

```
for (int i = 3; i < argc; i+=2)
  {
  std::string entryId( argv[i] );
  std::string value( argv[i+1] );
  itk::EncapsulateMetaData<std::string>( dictionary, entryId, value );
  }
```

Now that the Dictionary has been updated, we proceed to save the image. This output image will have the modified data associated to its DICOM header.

Using the image type, we instantiate a writer type and construct a writer. A short pipeline between the reader and the writer is connected. The filename to write is taken from the command line arguments. The image IO object is connected to the writer.

```
typedef itk::ImageFileWriter< InputImageType >  Writer1Type;

Writer1Type::Pointer writer1 = Writer1Type::New();

writer1->SetInput( reader->GetOutput() );
writer1->SetFileName( argv[2] );
writer1->SetImageIO( gdcmImageIO );
```

Execution of the writer is triggered by invoking the `Update()` method.

```
  writer1->Update();
```

Remember again, that modifying the header entries of a DICOM file involves very serious risks for patients and therefore must be done with extreme caution.

CHAPTER

EIGHT

Registration

This chapter introduces ITK's capabilities for performing image registration. Image registration is the process of determining the spatial transform that maps points from one image to homologous points on a object in the second image. This concept is schematically represented in Figure 8.1. In ITK, registration is performed within a framework of pluggable components that can easily be interchanged. This flexibility means that a combinatorial variety of registration methods can be created, allowing users to pick and choose the right tools for their specific application.

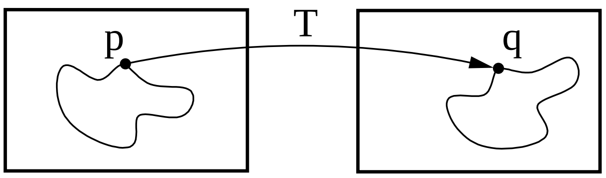


Figure 8.1: Image registration is the task of finding a spatial transform mapping on image into another.

8.1 Registration Framework

The components of the registration framework and their interconnections are shown in Figure 8.2. The basic input data to the registration process are two images: one is defined as the *fixed* image $f(\mathbf{X})$ and the other as the *moving* image $m(\mathbf{X})$. Where $\mathbf{X}$ represents a position in N-dimensional space. Registration is treated as an optimization problem with the goal of finding the spatial mapping that will bring the moving image into alignment with the fixed image.

The *transform* component $T(\mathbf{X})$ represents the spatial mapping of points from the fixed image space to points in the moving image space. The *interpolator* is used to evaluate moving image intensities at non-grid positions. The *metric* component $S(f, m \circ T)$ provides a measure of how well the fixed image is matched by the transformed moving image. This measure forms the quantitative criterion to be optimized by the *optimizer* over the search space defined by the parameters of the *transform*.

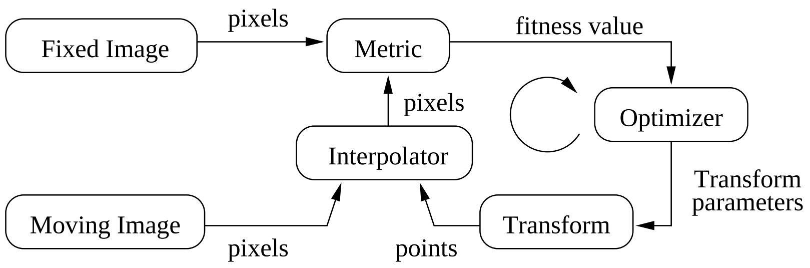


Figure 8.2: The basic components of the registration framework are two input images, a transform, a metric, an interpolator and an optimizer.

These various ITK registration components will be described in later sections. First, we begin with some simple registration examples.

8.2 ”Hello World” Registration

The source code for this section can be found in the file
`Examples/Registration/ImageRegistration1.cxx`.

This example illustrates the use of the image registration framework in Insight. It should be read as a ”Hello World” for ITK registration. Which means that for now, you don’t ask “why?”. Instead, use the example as an introduction to the elements that are typically involved in solving an image registration problem.

A registration method requires the following set of components: two input images, a transform, a metric, an interpolator and an optimizer. Some of these components are parameterized by the image type for which the registration is intended. The following header files provide declarations of common types used for these components.

```
#include "itkImageRegistrationMethod.h"
#include "itkTranslationTransform.h"
#include "itkMeanSquaresImageToImageMetric.h"
#include "itkLinearInterpolateImageFunction.h"
#include "itkRegularStepGradientDescentOptimizer.h"
#include "itkImage.h"
```

The types of each one of the components in the registration methods should be instantiated first. With that purpose, we start by selecting the image dimension and the type used for representing image pixels.

```
  const    unsigned int    Dimension = 2;
  typedef  float           PixelType;
```

The types of the input images are instantiated by the following lines.

```
typedef itk::Image< PixelType, Dimension >  FixedImageType;
typedef itk::Image< PixelType, Dimension >  MovingImageType;
```

The transform that will map the fixed image space into the moving image space is defined below.

```
typedef itk::TranslationTransform< double, Dimension > TransformType;
```

An optimizer is required to explore the parameter space of the transform in search of optimal values of the metric.

```
typedef itk::RegularStepGradientDescentOptimizer       OptimizerType;
```

The metric will compare how well the two images match each other. Metric types are usually parameterized by the image types as it can be seen in the following type declaration.

```
typedef itk::MeanSquaresImageToImageMetric<
                                  FixedImageType,
                                  MovingImageType >    MetricType;
```

Finally, the type of the interpolator is declared. The interpolator will evaluate the intensities of the moving image at non-grid positions.

```
typedef itk:: LinearInterpolateImageFunction<
                                  MovingImageType,
                                  double          >    InterpolatorType;
```

The registration method type is instantiated using the types of the fixed and moving images. This class is responsible for interconnecting all the components that we have described so far.

```
typedef itk::ImageRegistrationMethod<
                                  FixedImageType,
                                  MovingImageType >    RegistrationType;
```

Each one of the registration components is created using its `New()` method and is assigned to its respective `itk::SmartPointer`.

```
MetricType::Pointer         metric        = MetricType::New();
TransformType::Pointer      transform     = TransformType::New();
OptimizerType::Pointer      optimizer     = OptimizerType::New();
InterpolatorType::Pointer   interpolator  = InterpolatorType::New();
RegistrationType::Pointer   registration  = RegistrationType::New();
```

Each component is now connected to the instance of the registration method.

```
registration->SetMetric(        metric        );
registration->SetOptimizer(     optimizer     );
registration->SetTransform(     transform     );
registration->SetInterpolator(  interpolator  );
```

In this example, the fixed and moving images are read from files. This requires the `itk::ImageRegistrationMethod` to acquire its inputs from the output of the readers.

```
registration->SetFixedImage(    fixedImageReader->GetOutput()    );
registration->SetMovingImage(   movingImageReader->GetOutput()   );
```

The registration can be restricted to consider only a particular region of the fixed image as input to the metric computation. This region is defined with the `SetFixedImageRegion()` method. You could use this feature to reduce the computational time of the registration or to avoid unwanted objects present in the image from affecting the registration outcome. In this example we use the full available content of the image. This region is identified by the `BufferedRegion` of the fixed image. Note that for this region to be valid the reader must first invoke its `Update()` method.

```
fixedImageReader->Update();
registration->SetFixedImageRegion(
                  fixedImageReader->GetOutput()->GetBufferedRegion() );
```

The parameters of the transform are initialized by passing them in an array. This can be used to setup an initial known correction of the misalignment. In this particular case, a translation transform is being used for the registration. The array of parameters for this transform is simply composed of the translation values along each dimension. Setting the values of the parameters to zero initializes the transform to an *Identity* transform. Note that the array constructor requires the number of elements to be passed as an argument.

```
typedef RegistrationType::ParametersType ParametersType;
ParametersType initialParameters( transform->GetNumberOfParameters() );

initialParameters[0] = 0.0;  // Initial offset in mm along X
initialParameters[1] = 0.0;  // Initial offset in mm along Y

registration->SetInitialTransformParameters( initialParameters );
```

At this point the registration method is ready for execution. The optimizer is the component that drives the execution of the registration. However, the ImageRegistrationMethod class orchestrates the ensemble to make sure that everything is in place before control is passed to the optimizer.

It is usually desirable to fine tune the parameters of the optimizer. Each optimizer has particular parameters that must be interpreted in the context of the optimization strategy it implements. The optimizer used in this example is a variant of gradient descent that attempts to prevent it from taking steps that are too large. At each iteration, this optimizer will take a step along the direction of the `itk::ImageToImageMetric` derivative. The initial length of the step is defined by the user. Each time the direction of the derivative abruptly changes, the optimizer assumes that a local extrema has been passed and reacts by reducing the step length by a half. After several reductions of the step length, the optimizer may be moving in a very restricted area of the transform parameter space. The user can define how small the step length should be to consider convergence to have been reached. This is equivalent to defining the precision with which the final transform should be known.

The initial step length is defined with the method `SetMaximumStepLength()`, while the tolerance for convergence is defined with the method `SetMinimumStepLength()`.

```
optimizer->SetMaximumStepLength( 4.00 );
optimizer->SetMinimumStepLength( 0.01 );
```

In case the optimizer never succeeds reaching the desired precision tolerance, it is prudent to establish a limit on the number of iterations to be performed. This maximum number is defined with the method `SetNumberOfIterations()`.

```
optimizer->SetNumberOfIterations( 200 );
```

The registration process is triggered by an invocation to the `Update()` method. If something goes wrong during the initialization or execution of the registration an exception will be thrown. We should therefore place the `Update()` method inside a `try/catch` block as illustrated in the following lines.

```
try
  {
  registration->Update();
  }
catch( itk::ExceptionObject & err )
  {
  std::cerr << "ExceptionObject caught !" << std::endl;
  std::cerr << err << std::endl;
  return -1;
  }
```

In a real life application, you may attempt to recover from the error by taking more effective actions in the catch block. Here we are simply printing out a message and then terminating the execution of the program.

The result of the registration process is an array of parameters that defines the spatial transformation in an unique way. This final result is obtained using the `GetLastTransformParameters()` method.

```
ParametersType finalParameters = registration->GetLastTransformParameters();
```

In the case of the itk::TranslationTransform, there is a straightforward interpretation of the parameters. Each element of the array corresponds to a translation along one spatial dimension.

```
const double TranslationAlongX = finalParameters[0];
const double TranslationAlongY = finalParameters[1];
```

The optimizer can be queried for the actual number of iterations performed to reach convergence. The `GetCurrentIteration()` method returns this value. A large number of iterations may be an indication that the maximum step length has been set too small, which is undesirable since it results in long computational times.

```
const unsigned int numberOfIterations = optimizer->GetCurrentIteration();
```

The value of the image metric corresponding to the last set of parameters can be obtained with the `GetValue()` method of the optimizer.

```
const double bestValue = optimizer->GetValue();
```

Let's execute this example over two of the images provided in `Examples/Data`:

- `BrainProtonDensitySliceBorder20.png`
- `BrainProtonDensitySliceShifted13x17y.png`

The second image is the result of intentionally translating the first image by $(13,17)$ millimeters. Both images have unit-spacing and are shown in Figure 8.3. The registration takes 18 iterations and the resulting transform parameters are:

```
Translation X = 12.9959
Translation Y = 17.0001
```

As expected, these values match quite well the misalignment that we intentionally introduced in the moving image.

It is common, as the last step of a registration task, to use the resulting transform to map the moving image into the fixed image space. This is easily done with the itk::ResampleImageFilter. Please refer to Section 6.9.4 for details on the use of this filter. First, a ResampleImageFilter type is instantiated using the image types. It is convenient to use the fixed image type as the output type since it is likely that the transformed moving image will be compared with the fixed image.

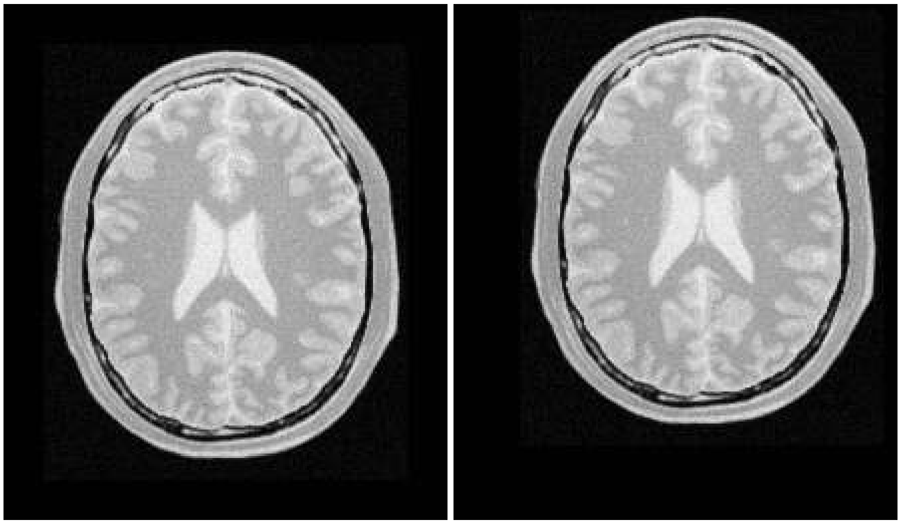


Figure 8.3: Fixed and Moving image provided as input to the registration method.

```
typedef itk::ResampleImageFilter<
                            MovingImageType,
                            FixedImageType >    ResampleFilterType;
```

A resampling filter is created and the moving image is connected as its input.

```
ResampleFilterType::Pointer resampler = ResampleFilterType::New();
resampler->SetInput( movingImageReader->GetOutput() );
```

The Transform that is produced as output of the Registration method is also passed as input to the resampling filter. Note the use of the methods `GetOutput()` and `Get()`. This combination is needed here because the registration method acts as a filter whose output is a transform decorated in the form of a `itk::DataObject`. For details in this construction you may want to read the documentation of the `itk::DataObjectDecorator`.

```
resampler->SetTransform( registration->GetOutput()->Get() );
```

As described in Section 6.9.4, the ResampleImageFilter requires additional parameters to be specified, in particular, the spacing, origin and size of the output image. The default pixel value is also set to a distinct gray level in order to highlight the regions that are mapped outside of the moving image.

```
FixedImageType::Pointer fixedImage = fixedImageReader->GetOutput();
resampler->SetSize( fixedImage->GetLargestPossibleRegion().GetSize() );
```

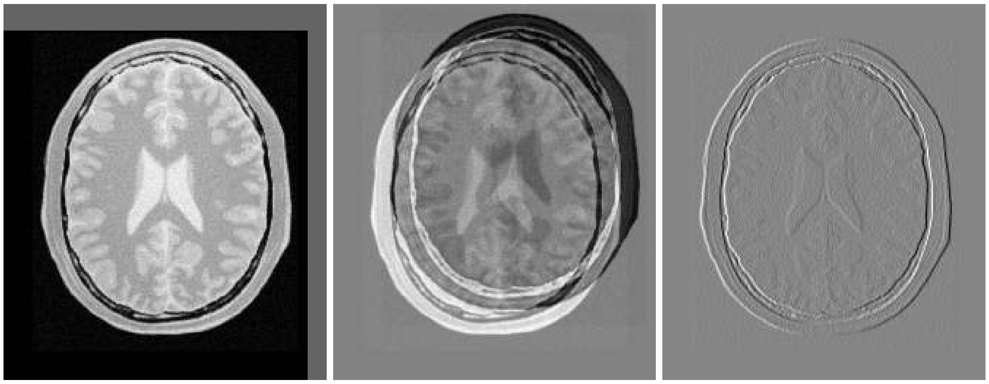


Figure 8.4: Mapped moving image and its difference with the fixed image before and after registration

```
  resampler->SetOutputOrigin(  fixedImage->GetOrigin() );
  resampler->SetOutputSpacing( fixedImage->GetSpacing() );
  resampler->SetDefaultPixelValue( 100 );
```

The output of the filter is passed to a writer that will store the image in a file. An itk::CastImageFilter is used to convert the pixel type of the resampled image to the final type used by the writer. The cast and writer filters are instantiated below.

```
  typedef unsigned char OutputPixelType;
  typedef itk::Image< OutputPixelType, Dimension > OutputImageType;
  typedef itk::CastImageFilter<
                        FixedImageType,
                        OutputImageType > CastFilterType;
  typedef itk::ImageFileWriter< OutputImageType >  WriterType;
```

The filters are created by invoking their `New()` method.

```
  WriterType::Pointer      writer =  WriterType::New();
  CastFilterType::Pointer  caster =  CastFilterType::New();
```

The filters are connected together and the `Update()` method of the writer is invoked in order to trigger the execution of the pipeline.

```
  caster->SetInput( resampler->GetOutput() );
  writer->SetInput( caster->GetOutput()   );
  writer->Update();
```

The fixed image and the transformed moving image can easily be compared using the itk::SubtractImageFilter. This pixel-wise filter computes the difference between homologous pixels of its two input images.

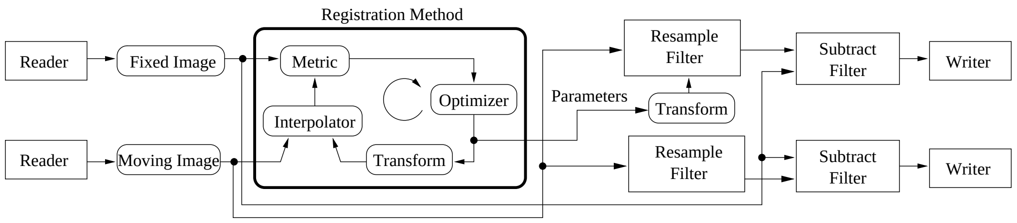


Figure 8.5: Pipeline structure of the registration example.

```
typedef itk::SubtractImageFilter<
                                FixedImageType,
                                FixedImageType,
                                FixedImageType > DifferenceFilterType;

DifferenceFilterType::Pointer difference = DifferenceFilterType::New();

difference->SetInput1( fixedImageReader->GetOutput() );
difference->SetInput2( resampler->GetOutput() );
```

Note that the use of subtraction as a method for comparing the images is appropriate here because we chose to represent the images using a pixel type `float`. A different filter would have been used if the pixel type of the images were any of the `unsigned` integer type.

Since the differences between the two images may correspond to very low values of intensity, we rescale those intensities with a `itk::RescaleIntensityImageFilter` in order to make them more visible. This rescaling will also make possible to visualize the negative values even if we save the difference image in a file format that only support unsigned pixel values[1]. We also reduce the `DefaultPixelValue` to “1” in order to prevent that value from absorbing the dynamic range of the differences between the two images.

```
typedef itk::RescaleIntensityImageFilter<
                                FixedImageType,
                                OutputImageType >   RescalerType;

RescalerType::Pointer intensityRescaler = RescalerType::New();

intensityRescaler->SetInput( difference->GetOutput() );
intensityRescaler->SetOutputMinimum(   0 );
intensityRescaler->SetOutputMaximum( 255 );

resampler->SetDefaultPixelValue( 1 );
```

Its output can be passed to another writer.

[1]This is the case of PNG, BMP, JPEG and TIFF among other common file formats.

```
  WriterType::Pointer writer2 = WriterType::New();
  writer2->SetInput( intensityRescaler->GetOutput() );
```

For the purpose of comparison, the difference between the fixed image and the moving image before registration can also be computed by simply setting the transform to an identity transform. Note that the resampling is still necessary because the moving image does not necessarily have the same spacing, origin and number of pixels as the fixed image. Therefore a pixel-by-pixel operation cannot in general be performed. The resampling process with an identity transform will ensure that we have a representation of the moving image in the grid of the fixed image.

```
  TransformType::Pointer identityTransform = TransformType::New();
  identityTransform->SetIdentity();
  resampler->SetTransform( identityTransform );
```

The complete pipeline structure of the current example is presented in Figure 8.5. The components of the registration method are depicted as well. Figure 8.4 (left) shows the result of resampling the moving image in order to map it onto the fixed image space. The top and right borders of the image appear in the gray level selected with the `SetDefaultPixelValue()` in the ResampleImageFilter. The center image shows the difference between the fixed image and the original moving image. That is, the difference before the registration is performed. The right image shows the difference between the fixed image and the transformed moving image. That is, after the registration has been performed. Both difference images have been rescaled in intensity in order to highlight those pixels where differences exist. Note that the final registration is still off by a fraction of a pixel, which results in bands around edges of anatomical structures to appear in the difference image. A perfect registration would have produced a null difference image.

It is always useful to keep in mind that registration is essentially an optimization problem. Figure 8.6 helps to reinforce this notion by showing the trace of translations and values of the image metric at each iteration of the optimizer. It can be seen from the top figure that the step length is reduced progressively as the optimizer gets closer to the metric extrema. The bottom plot clearly shows how the metric value decreases as the optimization advances. The log plot helps to highlight the normal oscillations of the optimizer around the extrema value.

8.3 Features of the Registration Framework

This section presents a discussion on the two most common difficulties that users encounter when they start using the ITK registration framework. They are, in order of difficulty

- The direction of the Transform mapping
- The fact that registration is done in physical coordinates

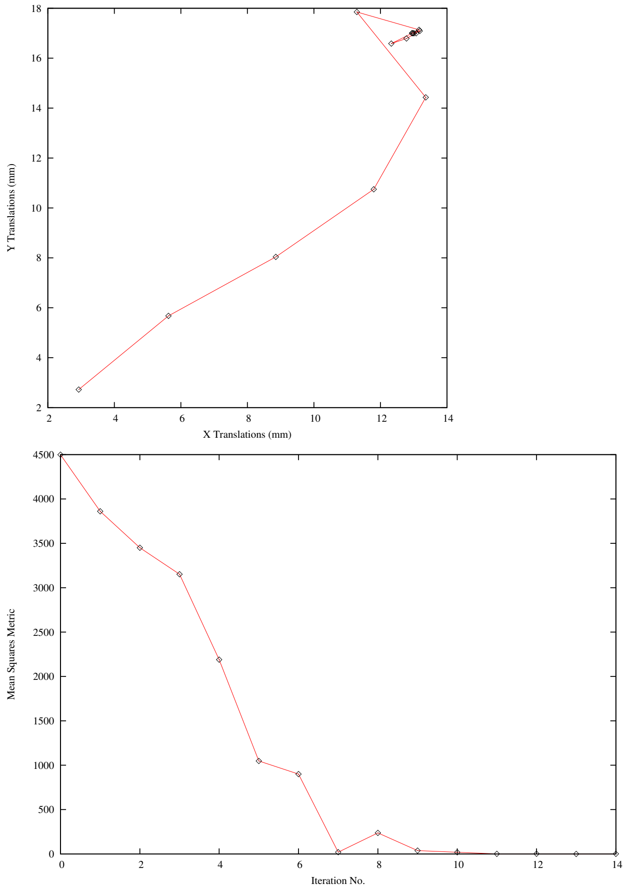


Figure 8.6: The sequence of translations and metric values at each iteration of the optimizer.

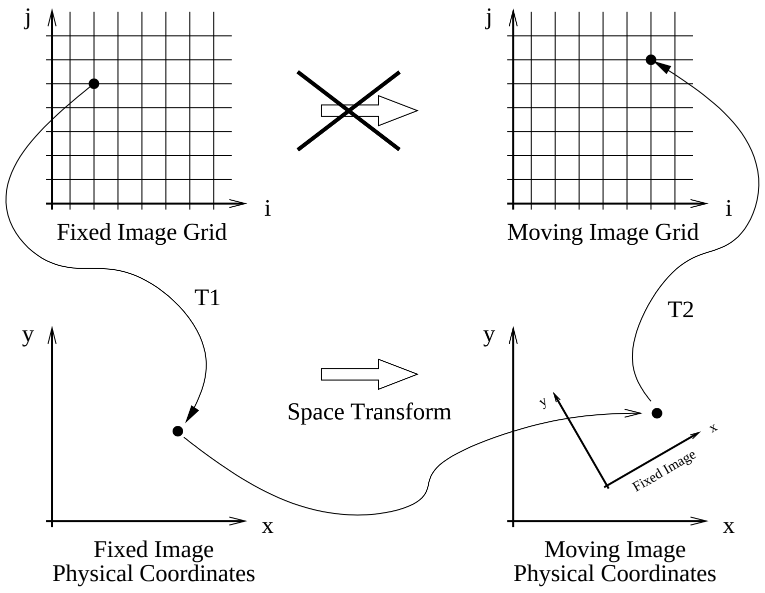


Figure 8.7: Different coordinate systems involved in the image registration process. Note that the transform being optimized is the one mapping from the physical space of the fixed image into the physical space of the moving image.

Probably the reason why these two topics tend to create confusion is that they are implemented in different ways in other systems and therefore users tend to have different expectations regarding how things should work in ITK. The situation is further complicated by the fact that most people describe image operations as if they were manually performed in a picture in paper.

8.3.1 Direction of the Transform Mapping

The Transform that is optimized in the ITK registration framework is the one that maps points from the physical space of the fixed image into the physical space of the moving image. This is illustrated in Figure 8.7. This implies that the Transform will accept as input points from the fixed image and it will compute the coordinates of the analogous points in the moving image. What tends to create confusion is the fact that when the Transform shifts a point on the **positive** X direction, the visual effect of this mapping, once the moving image is resampled, is equivalent to *manually shifting* the moving image along the **negative** X direction. In the same way, when the Transform applies a **clock-wise** rotation to the fixed image points, the visual effect of this mapping once the moving image has been resampled is equivalent to *manually rotating* the moving image **counter-clock-wise**.

The reason why this direction of mapping has been chosen for the ITK implementation of the registration framework is that this is the direction that better fits the fact that the moving image is expected to be resampled using the grid of the fixed image. The nature of the resampling process is such that an algorithm must go through every pixel of the *fixed* image and compute the intensity that should be assigned to this pixel from the mapping of the *moving* image. This computation involves taking the integral coordinates of the pixel in the image grid, usually called the "(i,j)" coordinates, mapping them into the physical space of the fixed image (transform **T1** in Figure 8.7), mapping those physical coordinates into the physical space of the moving image (Transform to be optimized), then mapping the physical coordinates of the moving image in to the integral coordinates of the discrete grid of the moving image (transform **T2** in the figure), where the value of the pixel intensity will be computed by interpolation.

If we have used the Transform that maps coordinates from the moving image physical space into the fixed image physical space, then the resampling process could not guarantee that every pixel in the grid of the fixed image was going to receive one and only one value. In other words, the resampling will have resulted in an image with holes and with redundant or overlapped pixel values.

As you have seen in the previous examples, and you will corroborate in the remaining examples in this chapter, the Transform computed by the registration framework is the Transform that can be used directly in the resampling filter in order to map the moving image into the discrete grid of the fixed image.

There are exceptional cases in which the transform that you want is actually the inverse transform of the one computed by the ITK registration framework. Only in those cases you may have to recur to invoking the `GetInverse()` method that most transform offer. Make sure that before you consider following that dark path, you interact with the examples of resampling illustrated in section 6.9 in order to get familiar with the correct interpretation of the transforms.

8.3.2 Registration is done in physical space

The second common difficulty that users encounter with the ITK registration framework is related to the fact that ITK performs registration in the context of physical space and not in the discrete space of the image grid. Figure 8.7 show this concept by crossing the transform that goes between the two image grids. One important consequence of this fact is that having the correct image origin and image pixel size is fundamental for the success of the registration process in ITK. Users must make sure that they provide correct values for the origin and spacing of both the fixed and moving images.

A typical case that helps to understand this issue, is to consider the registration of two images where one has a pixel size different from the other. For example, a PET[2] image and a CT[3] image. Typically a CT image will have a pixel size in the order of 1 millimeter, while a PET image will have a pixel size in the order of 5 millimeters to 1 centimeter. Therefore, the CT will need about 500 pixels in order to cover the extent across a human brain, while the PET image will only have about 50 pixels for covering the same physical extent of a human brain.

A user performing registration between a PET image and a CT image may be naively expecting that because the PET image has less pixels, a *scaling* factor is required in the Transform in order to map this image into the CT image. At that point, this person is attempting to interpret the registration process directly between the two image grids, or in *pixel space*. What ITK will do in this case is to take into account the pixel size that the user has provided and it will use that pixel size in order to compute a scaling factor for Transforms *T1* and *T2* in Figure 8.7. Since these two transforms take care of the required scaling factor, the spatial Transform to be computed during the registration process does not need to be concerned about such scaling. The transform that ITK is computing is the one that will physically map the brain from the moving image into the brain of the fixed image.

In order to better understand this concepts, it is very useful to draw sketches of the fixed and moving image *at scale* in the same physical coordinate system. That is the geometrical configuration that the ITK registration framework uses as context. Keeping this in mind helps a lot for interpreting correctly the results of a registration process performed with ITK.

8.4 Monitoring Registration

The source code for this section can be found in the file
`Examples/Registration/ImageRegistration3.cxx`.

Given the numerous parameters involved in tuning a registration method for a particular application, it is not uncommon for a registration process to run for several minutes and still produce

[2] Positron Emission Tomography
[3] Computer Tomography in X-rays

a useless result. To avoid this situation it is quite helpful to track the evolution of the registration as it progresses. The following section illustrates the mechanisms provided in ITK for monitoring the activity of the ImageRegistrationMethod class.

Insight implements the *Observer/Command* design pattern [28]. (See Section 3.2.6 for an overview.) The classes involved in this implementation are the itk::Object, itk::Command and itk::EventObject classes. The Object is the base class of most ITK objects. This class maintains a linked list of pointers to event observers. The role of observers is played by the Command class. Observers register themselves with an Object, declaring that they are interested in receiving notification when a particular event happens. A set of events is represented by the hierarchy of the Event class. Typical events are Start, End, Progress and Iteration.

Registration is controlled by an itk::Optimizer, which generally executes an iterative process. Most Optimizer classes invoke an itk::IterationEvent at the end of each iteration. When an event is invoked by an object, this object goes through its list of registered observers (Commands) and checks whether any one of them has expressed interest in the current event type. Whenever such an observer is found, its corresponding Execute() method is invoked. In this context, Execute() methods should be considered *callbacks*. As such, some of the common sense rules of callbacks should be respected. For example, Execute() methods should not perform heavy computational tasks. They are expected to execute rapidly, for example, printing out a message or updating a value in a GUI.

The following code illustrates a simple way of creating a Observer/Command to monitor a registration process. This new class derives from the Command class and provides a specific implementation of the Execute() method. First, the header file of the Command class must be included.

```
#include "itkCommand.h"
```

Our custom command class is called CommandIterationUpdate. It derives from the Command class and declares for convenience the types Self and Superclass. This facilitate the use of standard macros later in the class implementation.

```
class CommandIterationUpdate : public itk::Command
{
public:
  typedef  CommandIterationUpdate   Self;
  typedef  itk::Command             Superclass;
```

The following typedef declares the type of the SmartPointer capable of holding a reference to this object.

```
  typedef itk::SmartPointer<Self>  Pointer;
```

The itkNewMacro takes care of defining all the necessary code for the New() method. Those with curious minds are invited to see the details of the macro in the file itkMacro.h in the Insight/Code/Common directory.

```
  itkNewMacro( Self );
```

In order to ensure that the New() method is used to instantiate the class (and not the C++ new operator), the constructor is declared protected.

```
protected:
  CommandIterationUpdate() {};
```

Since this Command object will be observing the optimizer, the following typedefs are useful for converting pointers when the Execute() method is invoked. Note the use of const on the declaration of OptimizerPointer. This is relevant since, in this case, the observer is not intending to modify the optimizer in any way. A const interface ensures that all operations invoked on the optimizer are read-only.

```
  typedef itk::RegularStepGradientDescentOptimizer     OptimizerType;
  typedef const OptimizerType                         *OptimizerPointer;
```

ITK enforces const-correctness. There is hence a distinction between the Execute() method that can be invoked from a const object and the one that can be invoked from a non-const object. In this particular example the non-const version simply invoke the const version. In a more elaborate situation the implementation of both Execute() methods could be quite different. For example, you could imagine a non-const interaction in which the observer decides to stop the optimizer in response to a divergent behavior. A similar case could happen when a user is controlling the registration process from a GUI.

```
  void Execute(itk::Object *caller, const itk::EventObject & event)
  {
    Execute( (const itk::Object *)caller, event);
  }
```

Finally we get to the heart of the observer, the Execute() method. Two arguments are passed to this method. The first argument is the pointer to the object that invoked the event. The second argument is the event that was invoked.

```
  void Execute(const itk::Object * object, const itk::EventObject & event)
  {
```

Note that the first argument is a pointer to an Object even though the actual object invoking the event is probably a subclass of Object. In our case we know that the actual object is an optimizer. Thus we can perform a dynamic_cast to the real type of the object.

```
    OptimizerPointer optimizer =
                         dynamic_cast< OptimizerPointer >( object );
```

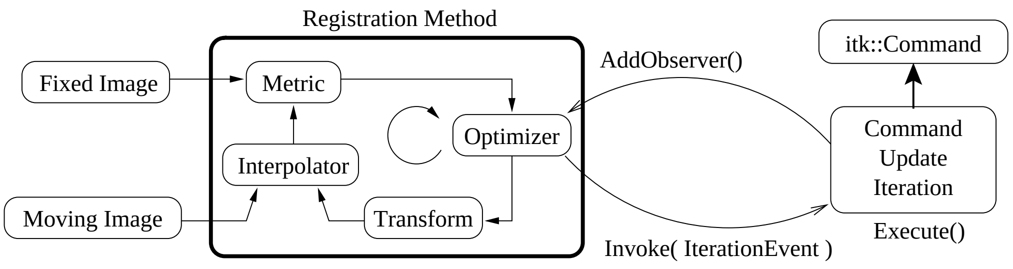


Figure 8.8: Interaction between the Command/Observer and the Registration Method.

The next step is to verify that the event invoked is actually the one in which we are interested. This is checked using the RTTI[4] support. The `CheckEvent()` method allows us to compare the actual type of two events. In this case we compare the type of the received event with an IterationEvent. The comparison will return true if `event` is of type `IterationEvent` or derives from `IterationEvent`. If we find that the event is not of the expected type then the `Execute()` method of this command observer should return without any further action.

```
    if( ! itk::IterationEvent().CheckEvent( &event ) )
      {
      return;
      }
```

If the event matches the type we are looking for, we are ready to query data from the optimizer. Here, for example, we get the current number of iterations, the current value of the cost function and the current position on the parameter space. All of these values are printed to the standard output. You could imagine more elaborate actions like updating a GUI or refreshing a visualization pipeline.

```
      std::cout << optimizer->GetCurrentIteration() << " = ";
      std::cout << optimizer->GetValue() << " : ";
      std::cout << optimizer->GetCurrentPosition() << std::endl;
```

This concludes our implementation of a minimal Command class capable of observing our registration method. We can now move on to configuring the registration process.

Once all the registration components are in place we can create one instance of our observer. This is done with the standard `New()` method and assigned to a SmartPointer.

```
  CommandIterationUpdate::Pointer observer = CommandIterationUpdate::New();
```

[4]RTTI stands for: Run-Time Type Information

The newly created command is registered as observer on the optimizer, using the `AddObserver()` method. Note that the event type is provided as the first argument to this method. In order for the RTTI mechanism to work correctly, a newly created event of the desired type must be passed as the first argument. The second argument is simply the smart pointer to the optimizer. Figure 8.8 illustrates the interaction between the Command/Observer class and the registration method.

```
  optimizer->AddObserver( itk::IterationEvent(), observer );
```

At this point, we are ready to execute the registration. The typical call to `StartRegistration()` will do it. Note again the use of the `try/catch` block around the `StartRegistration()` method in case an exception is thrown.

```
  try
    {
    registration->StartRegistration();
    }
  catch( itk::ExceptionObject & err )
    {
    std::cout << "ExceptionObject caught !" << std::endl;
    std::cout << err << std::endl;
    return -1;
    }
```

The registration process is applied to the following images in `Examples/Data`:

- `BrainProtonDensitySliceBorder20.png`
- `BrainProtonDensitySliceShifted13x17y.png`

It produces the following output.

```
     0 = 4499.45 : [2.9287, 2.72447]
     1 = 3860.84 : [5.62751, 5.67683]
     2 = 3450.68 : [8.85516, 8.03952]
     3 = 3152.07 : [11.7997, 10.7469]
     4 = 2189.97 : [13.3628, 14.4288]
     5 = 1047.21 : [11.292, 17.851]
     6 = 900.189 : [13.1602, 17.1372]
     7 = 19.6301 : [12.3268, 16.5846]
     8 = 237.317 : [12.7824, 16.7906]
     9 = 38.1331 : [13.1833, 17.0894]
     10 = 18.9201 : [12.949, 17.002]
     11 = 1.15456 : [13.074, 16.9979]
```

```
12 = 2.42488 : [13.0115, 16.9994]
13 = 0.0590549 : [12.949, 17.002]
14 = 1.15451 : [12.9803, 17.001]
15 = 0.173731 : [13.0115, 16.9997]
16 = 0.0586584 : [12.9959, 17.0001]
```

You can verify from the code in the `Execute()` method that the first column is the iteration number, the second column is the metric value and the third and fourth columns are the parameters of the transform, which is a $2D$ translation transform in this case. By tracking these values as the registration progresses, you will be able to determine whether the optimizer is advancing in the right direction and whether the step-length is reasonable or not. That will allow you to interrupt the registration process and fine-tune parameters without having to wait until the optimizer stops by itself.

8.5 Multi-Modality Registration

Some of the most challenging cases of image registration arise when images of different modalities are involved. In such cases, metrics based on direct comparison of gray levels are not applicable. It has been extensively shown that metrics based on the evaluation of mutual information are well suited for overcoming the difficulties of multi-modality registration.

The concept of Mutual Information is derived from Information Theory and its application to image registration has been proposed in different forms by different groups [17, 52, 85], a more detailed review can be found in [33, 64]. The Insight Toolkit currently provides five different implementations of Mutual Information metrics (see section 8.10 for details). The following examples illustrate the practical use of some of these metrics.

8.5.1 Viola-Wells Mutual Information

The source code for this section can be found in the file
`Examples/Registration/ImageRegistration2.cxx`.

The following simple example illustrates how multiple imaging modalities can be registered using the ITK registration framework. The first difference between this and previous examples is the use of the `itk::MutualInformationImageToImageMetric` as the cost-function to be optimized. The second difference is the use of the `itk::GradientDescentOptimizer`. Due to the stochastic nature of the metric computation, the values are too noisy to work successfully with the `itk::RegularStepGradientDescentOptimizer`. Therefore, we will use the simpler GradientDescentOptimizer with a user defined learning rate. The following headers declare the basic components of this registration method.

```
#include "itkImageRegistrationMethod.h"
```

```
#include "itkTranslationTransform.h"
#include "itkMutualInformationImageToImageMetric.h"
#include "itkLinearInterpolateImageFunction.h"
#include "itkGradientDescentOptimizer.h"
#include "itkImage.h"
```

One way to simplify the computation of the mutual information is to normalize the statistical distribution of the two input images. The `itk::NormalizeImageFilter` is the perfect tool for this task. It rescales the intensities of the input images in order to produce an output image with zero mean and unit variance. This filter has been discussed in Section 6.3.

```
#include "itkNormalizeImageFilter.h"
```

Additionally, low-pass filtering of the images to be registered will also increase robustness against noise. In this example, we will use the `itk::DiscreteGaussianImageFilter` for that purpose. The characteristics of this filter have been discussed in Section 6.7.1.

```
#include "itkDiscreteGaussianImageFilter.h"
```

The moving and fixed images types should be instantiated first.

```
  const    unsigned int    Dimension = 2;
  typedef  unsigned short  PixelType;

  typedef itk::Image< PixelType, Dimension >  FixedImageType;
  typedef itk::Image< PixelType, Dimension >  MovingImageType;
```

It is convenient to work with an internal image type because mutual information will perform better on images with a normalized statistical distribution. The fixed and moving images will be normalized and converted to this internal type.

```
  typedef   float     InternalPixelType;
  typedef itk::Image< InternalPixelType, Dimension > InternalImageType;
```

The rest of the image registration components are instantiated as illustrated in Section 8.2 with the use of the `InternalImageType`.

```
  typedef itk::TranslationTransform< double, Dimension > TransformType;
  typedef itk::GradientDescentOptimizer                  OptimizerType;
  typedef itk::LinearInterpolateImageFunction<
                                    InternalImageType,
                                    double             > InterpolatorType;
  typedef itk::ImageRegistrationMethod<
                                    InternalImageType,
                                    InternalImageType >  RegistrationType;
```

The mutual information metric type is instantiated using the image types.

```
typedef itk::MutualInformationImageToImageMetric<
                                        InternalImageType,
                                        InternalImageType >    MetricType;
```

The metric is created using the `New()` method and then connected to the registration object.

```
MetricType::Pointer         metric        = MetricType::New();
registration->SetMetric( metric  );
```

The metric requires a number of parameters to be selected, including the standard deviation of the Gaussian kernel for the fixed image density estimate, the standard deviation of the kernel for the moving image density and the number of samples use to compute the densities and entropy values. Details on the concepts behind the computation of the metric can be found in Section 8.10.4. Experience has shown that a kernel standard deviation of 0.4 works well for images which have been normalized to a mean of zero and unit variance. We will follow this empirical rule in this example.

```
metric->SetFixedImageStandardDeviation(  0.4 );
metric->SetMovingImageStandardDeviation( 0.4 );
```

The normalization filters are instantiated using the fixed and moving image types as input and the internal image type as output.

```
typedef itk::NormalizeImageFilter<
                              FixedImageType,
                              InternalImageType
                                      > FixedNormalizeFilterType;

typedef itk::NormalizeImageFilter<
                              MovingImageType,
                              InternalImageType
                                            > MovingNormalizeFilterType;

FixedNormalizeFilterType::Pointer fixedNormalizer =
                                          FixedNormalizeFilterType::New();

MovingNormalizeFilterType::Pointer movingNormalizer =
                                          MovingNormalizeFilterType::New();
```

The blurring filters are declared using the internal image type as both the input and output types. In this example, we will set the variance for both blurring filters to 2.0.

```
typedef itk::DiscreteGaussianImageFilter<
                                    InternalImageType,
                                    InternalImageType
                                                  > GaussianFilterType;

GaussianFilterType::Pointer fixedSmoother  = GaussianFilterType::New();
GaussianFilterType::Pointer movingSmoother = GaussianFilterType::New();

fixedSmoother->SetVariance( 2.0 );
movingSmoother->SetVariance( 2.0 );
```

The output of the readers becomes the input to the normalization filters. The output of the normalization filters is connected as input to the blurring filters. The input to the registration method is taken from the blurring filters.

```
fixedNormalizer->SetInput(  fixedImageReader->GetOutput() );
movingNormalizer->SetInput( movingImageReader->GetOutput() );

fixedSmoother->SetInput( fixedNormalizer->GetOutput() );
movingSmoother->SetInput( movingNormalizer->GetOutput() );

registration->SetFixedImage(    fixedSmoother->GetOutput()    );
registration->SetMovingImage(   movingSmoother->GetOutput()   );
```

We should now define the number of spatial samples to be considered in the metric computation. Note that we were forced to postpone this setting until we had done the preprocessing of the images because the number of samples is usually defined as a fraction of the total number of pixels in the fixed image.

The number of spatial samples can usually be as low as 1% of the total number of pixels in the fixed image. Increasing the number of samples improves the smoothness of the metric from one iteration to another and therefore helps when this metric is used in conjunction with optimizers that rely of the continuity of the metric values. The trade-off, of course, is that a larger number of samples result in longer computation times per every evaluation of the metric.

It has been demonstrated empirically that the number of samples is not a critical parameter for the registration process. When you start fine tuning your own registration process, you should start using high values of number of samples, for example in the range of 20% to 50% of the number of pixels in the fixed image. Once you have succeeded to register your images you can then reduce the number of samples progressively until you find a good compromise on the time it takes to compute one evaluation of the Metric. Note that it is not useful to have very fast evaluations of the Metric if the noise in their values results in more iterations being required by the optimizer to converge. You must then study the behavior of the metric values as the iterations progress, just as illustrated in section 8.4.

```
const unsigned int numberOfPixels = fixedImageRegion.GetNumberOfPixels();
const unsigned int numberOfSamples =
                      static_cast< unsigned int >( numberOfPixels * 0.01 );

metric->SetNumberOfSpatialSamples( numberOfSamples );
```

Since larger values of mutual information indicate better matches than smaller values, we need to maximize the cost function in this example. By default the GradientDescentOptimizer class is set to minimize the value of the cost-function. It is therefore necessary to modify its default behavior by invoking the `MaximizeOn()` method. Additionally, we need to define the optimizer's step size using the `SetLearningRate()` method.

```
optimizer->SetLearningRate( 15.0 );
optimizer->SetNumberOfIterations( 200 );
optimizer->MaximizeOn();
```

Note that large values of the learning rate will make the optimizer unstable. Small values, on the other hand, may result in the optimizer needing too many iterations in order to walk to the extrema of the cost function. The easy way of fine tuning this parameter is to start with small values, probably in the range of $\{5.0, 10.0\}$. Once the other registration parameters have been tuned for producing convergence, you may want to revisit the learning rate and start increasing its value until you observe that the optimization becomes unstable. The ideal value for this parameter is the one that results in a minimum number of iterations while still keeping a stable path on the parametric space of the optimization. Keep in mind that this parameter is a multiplicative factor applied on the gradient of the Metric. Therefore, its effect on the optimizer step length is proportional to the Metric values themselves. Metrics with large values will require you to use smaller values for the learning rate in order to maintain a similar optimizer behavior.

Let's execute this example over two of the images provided in `Examples/Data`:

- `BrainT1SliceBorder20.png`
- `BrainProtonDensitySliceShifted13x17y.png`

The second image is the result of intentionally translating the image `BrainProtonDensity-SliceBorder20.png` by $(13, 17)$ millimeters. Both images have unit-spacing and are shown in Figure 8.9. The registration is stopped at 200 iterations and produces as result the parameters:

```
Translation X = 12.9147
Translation Y = 17.0871
```

These values are approximately within one tenth of a pixel from the true misalignment introduced in the moving image.

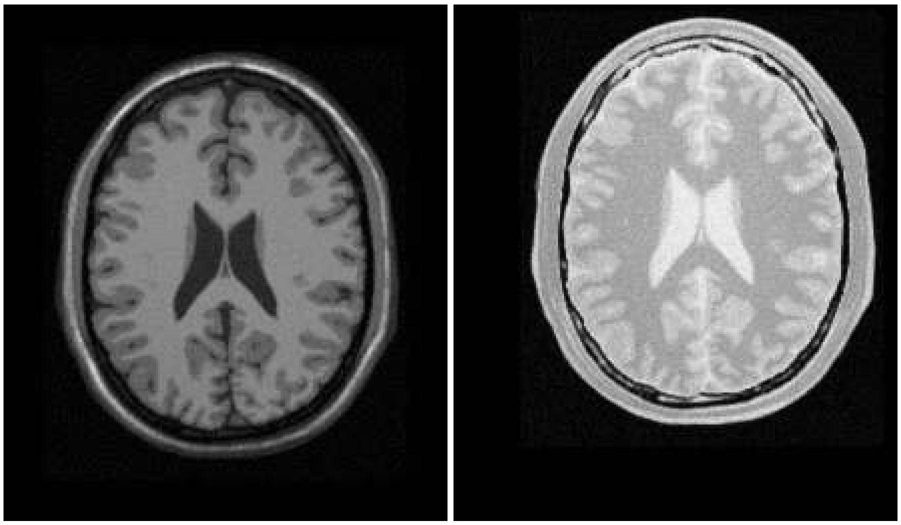


Figure 8.9: A T1 MRI (fixed image) and a proton density MRI (moving image) are provided as input to the registration method.

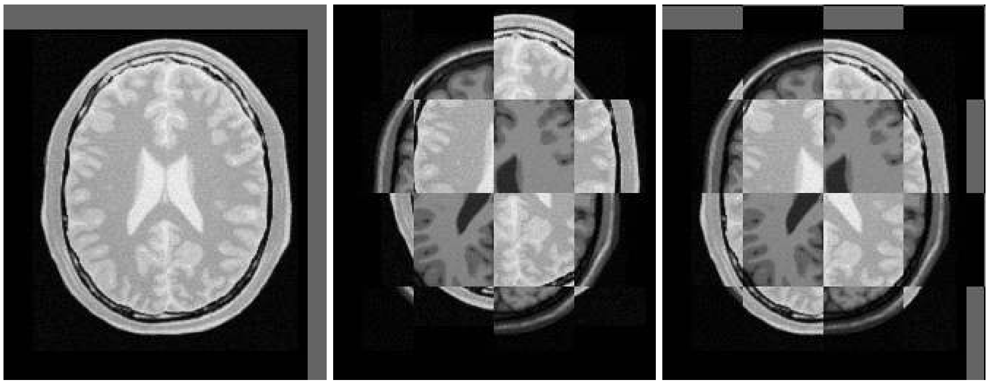


Figure 8.10: Mapped moving image (left) and composition of fixed and moving images before (center) and after (right) registration.

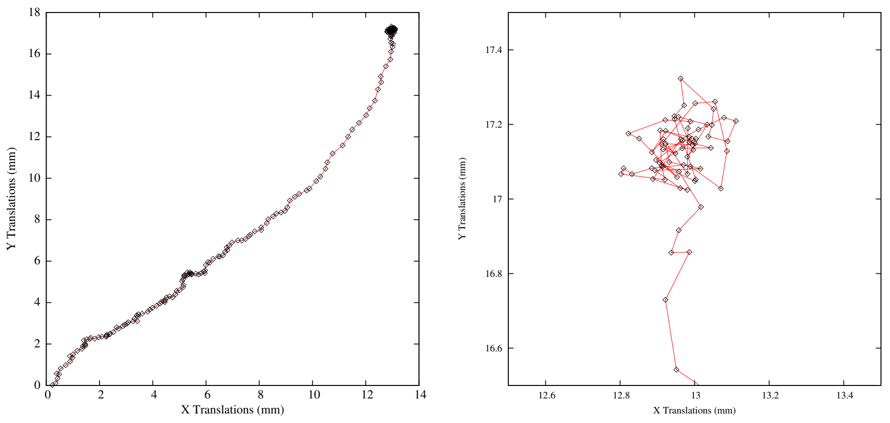


Figure 8.11: Sequence of translations during the registration process. On the left are iterations 0 to 200. On the right are iterations 150 to 200.

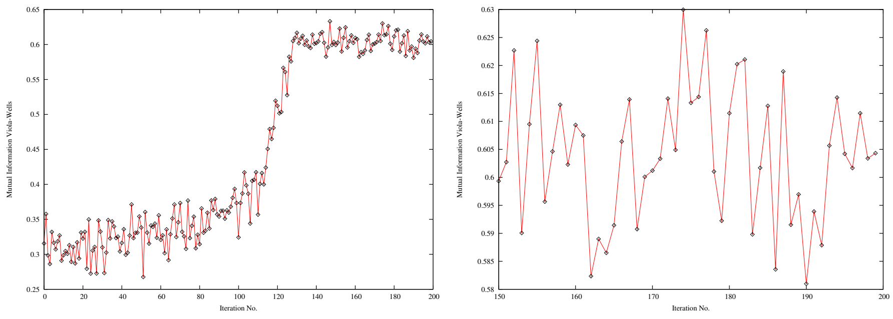


Figure 8.12: The sequence of metric values produced during the registration process. On the left are iterations 0 to 200. On the right are iterations 150 to 200.

The moving image after resampling is presented on the left side of Figure 8.10. The center and right figures present a checkerboard composite of the fixed and moving images before and after registration.

Figure 8.11 shows the sequence of translations followed by the optimizer as it searched the parameter space. The left plot shows iterations 0 to 200 while the right figure zooms into iterations 150 to 200. The area covered by the right figure has been highlighted by a rectangle in the left image. It can be seen that after a certain number of iterations the optimizer oscillates within one or two pixels of the true solution. At this point it is clear that more iterations will not help. Instead it is time to modify some of the parameters of the registration process, for example, reducing the learning rate of the optimizer and continuing the registration so that smaller steps are taken.

Figure 8.12 shows the sequence of metric values computed as the optimizer searched the parameter space. The left plot shows values when iterations are extended from 0 to 200 while the right figure zooms into iterations 150 to 200. The fluctuations in the metric value are due to the stochastic nature in which the measure is computed. At each call of `GetValue()`, two new sets of intensity samples are randomly taken from the image to compute the density and entropy estimates. Even with the fluctuations, the measure initially increases overall with the number of iterations. After about 150 iterations, the metric value merely oscillates without further noticeable convergence. The trace plots in Figure 8.12 highlight one of the difficulties associated with this particular metric: the stochastic oscillations make it difficult to determine convergence and limit the use of more sophisticated optimization methods. As explained above, the reduction of the learning rate as the registration progresses is very important in order to get precise results.

This example shows the importance of tracking the evolution of the registration method in order to obtain insight into the characteristics of the particular problem at hand and the components being used. The behavior revealed by these plots usually helps to identify possible improvements in the setup of the registration parameters.

The plots in Figures 8.11 and 8.12 were generated using Gnuplot[5]. The scripts used for this purpose are available in the `InsightDocuments` CVS module under the directory

`InsightDocuments/SoftwareGuide/Art`

Data for the plots was taken directly from the output that the Command/Observer in this example prints out to the console. The output was processed with the UNIX editor `sed`[6] in order to remove commas and brackets that were confusing for Gnuplot's parser. Both the shell script for running `sed` and for running Gnuplot are available in the directory indicated above. You may find useful to run them in order to verify the results presented here, and to eventually modify them for profiling your own registrations.

Open Science is not just an abstract concept. Open Science is something to be practiced every day with the simple gesture of sharing information with your peers, and by providing all the tools that they need for replicating the results that you are reporting. In Open Science, the only bad results are those that can not be replicated[7]. Science is dead when people blindly trust authorities [8] instead of verifying their statements by performing their own experiments [65, 66].

8.5.2 Mattes Mutual Information

The source code for this section can be found in the file
`Examples/Registration/ImageRegistration4.cxx`.

In this example, we will solve a simple multi-modality problem using another implementation of mutual information. This implementation was published by Mattes *et. al* [56]. One of the main differences between `itk::MattesMutualInformationImageToImageMetric` and `itk::MutualInformationImageToImageMetric` is that only one spatial sample set is used

[5] `http://www.gnuplot.info/`
[6] `http://www.gnu.org/software/sed/sed.html`
[7] `http://science.creativecommons.org/`
[8] For example: Reviewers of Scientific Journals.

for the whole registration process instead of using new samples every iteration. The use of a single sample set results in a much smoother cost function and hence allows the use of more intelligent optimizers. In this example, we will use the RegularStepGradientDescentOptimizer. Another noticeable difference is that pre-normalization of the images is not necessary as the metric rescales internally when building up the discrete density functions. Other differences between the two mutual information implementations are described in detail in Section 8.10.4.

First, we include the header files of the components used in this example.

```
#include "itkImageRegistrationMethod.h"
#include "itkTranslationTransform.h"
#include "itkMattesMutualInformationImageToImageMetric.h"
#include "itkLinearInterpolateImageFunction.h"
#include "itkRegularStepGradientDescentOptimizer.h"
#include "itkImage.h"
```

In this example the image types and all registration components, except the metric, are declared as in Section 8.2. The Mattes mutual information metric type is instantiated using the image types.

```
  typedef itk::MattesMutualInformationImageToImageMetric<
                                          FixedImageType,
                                          MovingImageType >    MetricType;
```

The metric is created using the `New()` method and then connected to the registration object.

```
  MetricType::Pointer metric = MetricType::New();
  registration->SetMetric( metric  );
```

The metric requires two parameters to be selected: the number of bins used to compute the entropy and the number of spatial samples used to compute the density estimates. In typical application 50 histogram bins are sufficient. Note however, that the number of bins may have dramatic effects on the optimizer's behavior. The number of spatial samples to be used depends on the content of the image. If the images are smooth and do not contain much detail, then using approximately 1 percent of the pixels will do. On the other hand, if the images are detailed, it may be necessary to use a much higher proportion, such as 20 percent.

```
  unsigned int numberOfBins = 24;
  unsigned int numberOfSamples = 10000;

  metric->SetNumberOfHistogramBins( numberOfBins );
  metric->SetNumberOfSpatialSamples( numberOfSamples );
```

One mechanism for bringing the Metric to its limit is to disable the sampling and use all the pixels present in the FixedImageRegion. This can be done with the `UseAllPixelsOn()` method.

You may want to try this option only while you are fine tuning all other parameters of your registration. We don't use this method in this current example though.

Another significant difference in the metric is that it computes the negative mutual information and hence we need to minimize the cost function in this case. In this example we will use the same optimization parameters as in Section 8.2.

```
optimizer->MinimizeOn();
optimizer->SetMaximumStepLength( 2.00 );
optimizer->SetMinimumStepLength( 0.001 );
optimizer->SetNumberOfIterations( 200 );
```

Whenever the regular step gradient descent optimizer encounters that the direction of movement has changed in the parametric space, it reduces the size of the step length. The rate at which the step length is reduced is controlled by a relaxation factor. The default value of the factor is 0.5. This value, however may prove to be inadequate for noisy metrics since they tend to induce very erratic movements on the optimizers and therefore result in many directional changes. In those conditions, the optimizer will rapidly shrink the step length while it is still too far from the location of the extrema in the cost function. In this example we set the relaxation factor to a number higher than the default in order to prevent the premature shrinkage of the step length.

```
optimizer->SetRelaxationFactor( 0.8 );
```

This example is executed using the same multi-modality images as the one in section 8.5.1 The registration converges after 59 iterations and produces the following results:

```
Translation X = 13.0283
Translation Y = 17.007
```

These values are a very close match to the true misalignment introduced in the moving image.

The result of resampling the moving image is presented on the left of Figure 8.13. The center and right parts of the figure present a checkerboard composite of the fixed and moving images before and after registration respectively.

Figure 8.14 (upper-left) shows the sequence of translations followed by the optimizer as it searched the parameter space. The upper-right figure presents a closer look at the convergence basin for the last iterations of the optimizer. The bottom of the same figure shows the sequence of metric values computed as the optimizer searched the parameter space. Comparing these trace plots with Figures 8.11 and 8.12, we can see that the measures produced by MattesMutualInformationImageToImageMetric are smoother than those of the MutualInformationImageToImageMetric. This smoothness allows the use of more sophisticated optimizers such as the `itk::RegularStepGradientDescentOptimizer` which efficiently locks onto the optimal value.

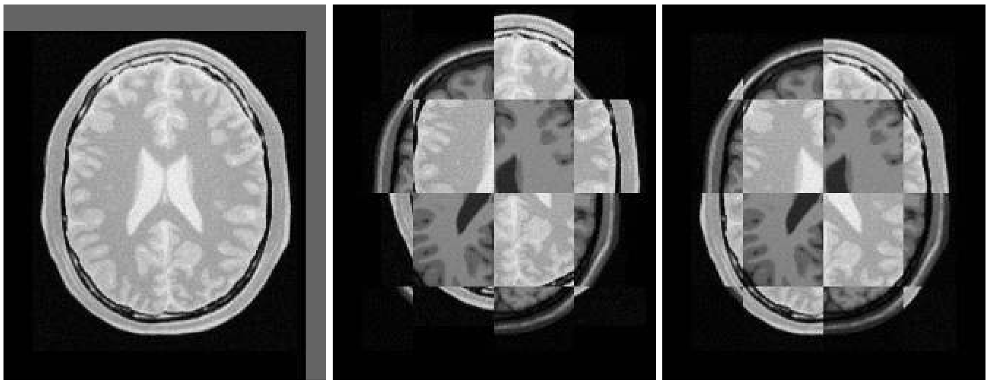


Figure 8.13: The mapped moving image (left) and the composition of fixed and moving images before (center) and after (right) registration with Mattes mutual information.

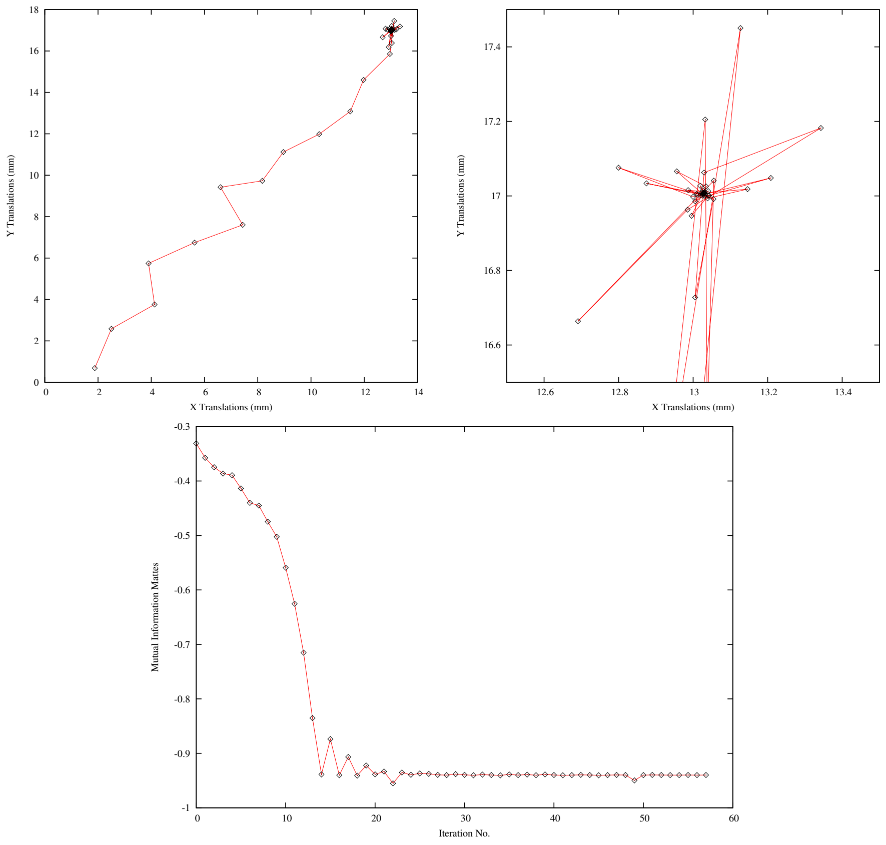


Figure 8.14: Sequence of translations and metric values at each iteration of the optimizer.

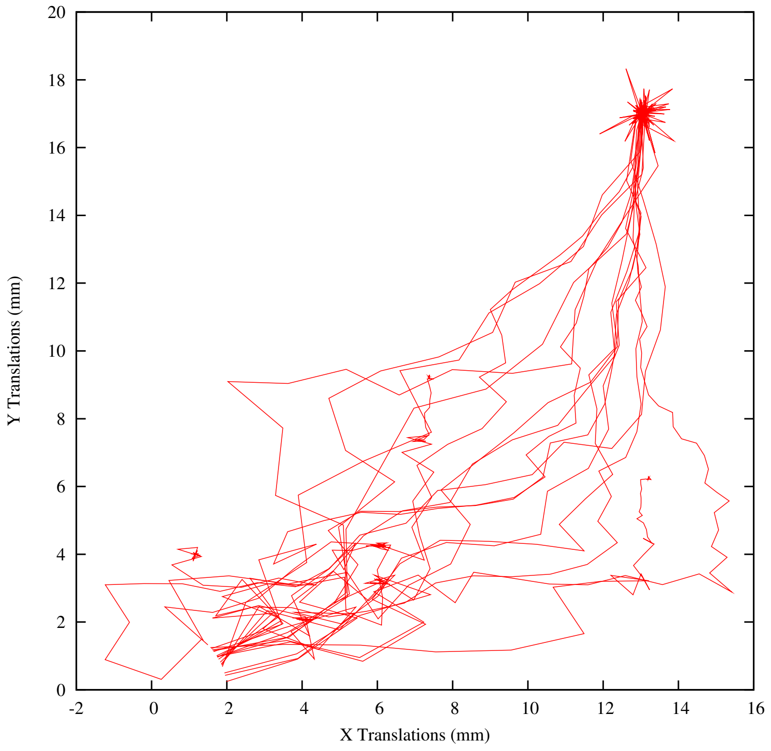


Figure 8.15: Sensitivity of the optimization path to the number of Bins used for estimating the value of Mutual Information with Mattes et al. approach.

You must note however that there are a number of non-trivial issues involved in the fine tuning of parameters for the optimization. For example, the number of bins used in the estimation of Mutual Information has a dramatic effect on the performance of the optimizer. In order to illustrate this effect, this same example has been executed using a range of different values for the number of bins, from 10 to 30. If you repeat this experiment, you will notice that depending on the number of bins used, the optimizer's path may get trapped early on in local minima. Figure 8.15 shows the multiple paths that the optimizer took in the parametric space of the transform as a result of different selections on the number of bins used by the Mattes Mutual Information metric. Note that many of the paths die in local minima instead of reaching the extrema value on the upper right corner.

Effects such as the one illustrated here highlight how useless is to compare different algorithms based on a non-exhaustive search of their parameter setting. It is quite difficult to be able to claim that a particular selection of parameters represent the best combination for running a particular algorithm. Therefore, when comparing the performance of two or more different algorithms, we are faced with the challenge of proving that none of the algorithms involved in the comparison is being run with a sub-optimal set of parameters.

The plots in Figures 8.14 and 8.15 were generated using Gnuplot. The scripts used for this purpose are available in the `InsightDocuments` CVS module under the directory

`InsightDocuments/SoftwareGuide/Art`

The use of these scripts was similar to what was described at the end of section 8.5.1.

8.5.3 Plotting joint histograms

The source code for this section can be found in the file
`Examples/Registration/ImageRegistrationHistogramPlotter.cxx`.

When fine tuning the parameters of an image registration process it is not always clear what factor are having a larger impact on the behavior of the registration. Even plotting the values of the metric and the transform parameters may not provide a clear indication on the best way to modify the optimizer and metric parameters in order to improve the convergence rate and stability. In such circumstances it is useful to take a closer look at the internals of the components involved in computing the registration. One of the critical components is, of course, the image metric. This section illustrates a mechanism that can be used for monitoring the behavior of the Mutual Information metric by continuously looking at the joint histogram at regular intervals during the iterations of the optimizer.

This particular example shows how to use the `itk::HistogramToEntropyImageFilter` class in order to get access to the joint histogram that is internally computed by the metric. This class represents the joint histogram as a $2D$ image and therefore can take advantage of the IO functionalities described in chapter 7. The example registers two images using the gradient descent optimizer. The transform used here is a simple translation transform. The metric is a `itk::MutualInformationHistogramImageToImageMetric`.

In the code below we create a helper class called the `HistogramWriter`. Its purpose is to save the joint histogram into a file using any of the file formats supported by ITK. This object is invoked after every iteration of the optimizer. The writer here saves the joint histogram into files with names: `JointHistogramXXX.mhd` where `XXX` is replaced with the iteration number. The output image contains the joint entropy histogram given by

$$f_{ij} = -p_{ij} \log_2(p_{ij}) \tag{8.1}$$

where the indices i and j identify the location of a bin in the Joint Histogram of the two images and are in the ranges $i \in [0 : N-1]$ and $j \in [0 : M-1]$. The image f representing the joint histogram has NxM pixels because the intensities of the Fixed image are quantized into N histogram bins and the intensities of the Moving image are quantized into M histogram bins. The probability value p_{ij} is computed from the frequency count of the histogram bins.

$$p_{ij} = \frac{q_{ij}}{\sum_{i=0}^{N-1} \sum_{j=0}^{M-1} q_{ij}} \tag{8.2}$$

The value q_{ij} is the frequency of a bin in the histogram and it is computed as the number of pixels where the Fixed image has intensities in the range of bin i and the Moving image has intensities on the range of bin j. The value p_{ij} is therefore the probability of the occurrence of the measurement vector centered in the bin ij. The filter produces an output image of pixel type `double`. For details on the use of Histograms in ITK please refer to section 10.1.3.

Depending on whether you want to see the joint histogram frequencies directly, or the joint probabilities, or log of joint probabilities, you may want to instantiate respectively any of the following classes

- `itk::HistogramToIntensityImageFilter`
- `itk::HistogramToProbabilityImageFilter`
- `itk::HistogramToLogProbabilityImageFilter`

The use of all of these classes is very similar. Note that the log of the probability is equivalent to units of information, also known as **bits**, more details on this concept can be found in section 10.3.2

The header files of the classes featured in this example are included as a first step.

```
#include "itkHistogramToEntropyImageFilter.h"
#include "itkMutualInformationHistogramImageToImageMetric.h"
```

Here we will create a simple class to write the joint histograms. This class, that we arbitrarily name as `HistogramWriter`, uses internally the `itk::HistogramToEntropyImageFilter` class among others.

```
class HistogramWriter
{
public:
  typedef float InternalPixelType;
  itkStaticConstMacro( Dimension, unsigned int, 2);

  typedef itk::Image< InternalPixelType, Dimension > InternalImageType;

  typedef itk::MutualInformationHistogramImageToImageMetric<
                                          InternalImageType,
                                          InternalImageType >    MetricType;


  typedef MetricType::HistogramType   HistogramType;

  typedef itk::HistogramToEntropyImageFilter< HistogramType >
                                  HistogramToEntropyImageFilterType;
  typedef HistogramToEntropyImageFilterType::Pointer
                              HistogramToImageFilterPointer;

  typedef HistogramToEntropyImageFilterType::OutputImageType OutputImageType;

  typedef itk::ImageFileWriter< OutputImageType > HistogramFileWriterType;
  typedef HistogramFileWriterType::Pointer        HistogramFileWriterPointer;
```

The `HistogramWriter` has a member variable `m_Filter` of type HistogramToEntropyImageFilter.

```
    this->m_Filter = HistogramToEntropyImageFilterType::New();
```

It also has an ImageFileWriter that has been instantiated using the image type that is produced as output from the histogram to image filter. We connect the output of the filter as input to the writer.

```
    this->m_HistogramFileWriter = HistogramFileWriterType::New();
    this->m_HistogramFileWriter->SetInput( this->m_Filter->GetOutput() );
```

The method of this class that is most relevant to our discussion is the one that writes the image into a file. In this method we assign the output histogram of the metric to the input of the histogram to image filter. In this way we construct an ITK 2*D* image where every pixel corresponds to one of the Bins of the joint histogram computed by the Metric.

```
  void WriteHistogramFile( const char * outputFilename  )
    {

    this->m_Filter->SetInput( m_Metric->GetHistogram() );
```

The output of the filter is connected to a filter that will rescale the intensities in order to improve the visualization of the values. This is done because it is common to find histograms of medical images that have a minority of bins that are largely dominant. Visualizing such histogram in direct values is challenging because only the dominant bins tend to become visible.

The following are the member variables of our `HistogramWriter` class.

```
private:
  MetricPointer                   m_Metric;
  HistogramToImageFilterPointer   m_Filter;
  HistogramFileWriterPointer      m_HistogramFileWriter;
```

We invoke the histogram writer within the Command/Observer of the optimizer to write joint histograms after every iteration.

```
      m_JointHistogramWriter.WriteHistogramFile( m_InitialHistogramFile.c_str() );
```

We instantiate an optimizer, interpolator and the registration method as shown in previous examples.

The number of bins in the metric is set with the `SetHistogramSize()` method. This will determine the number of pixels along each dimension of the joint histogram. Note that in this case we arbitrarily decided to use the same number of bins for the intensities of the Fixed image and those of the Moving image. However, this does not have to be the case, we could have selected different numbers of bins for each image.

```
  unsigned int numberOfHistogramBins = atoi( argv[7] );
  MetricType::HistogramType::SizeType histogramSize;
  histogramSize[0] = numberOfHistogramBins;
  histogramSize[1] = numberOfHistogramBins;
  metric->SetHistogramSize( histogramSize );
```

Mutual information attempts to re-group the joint entropy histograms into a more “meaningful” formation. An optimizer that minimizes the joint entropy seeks a transform that produces a small number of high value bins and a large majority of almost zero bins. Multi-modality registration seeks such a transform while also attempting to maximize the information contribution by the fixed and the moving images in the overall region of the metric.

A T1 MRI (fixed image) and a proton density MRI (moving image) as shown in Figure 8.9 are provided as input to this example.

Figure 8.16 shows the joint histograms before and after registration.

8.6 Centered Transforms

The ITK image coordinate origin is typically located in one of the image corners (see section 4.1.4 for details). This results in counter-intuitive transform behavior when rotations and scaling are involved. Users tend to assume that rotations and scaling are performed around a fixed point at the center of the image. In order to compensate for this difference in natural interpretation, the concept of *centered* transforms have been introduced into the toolkit. The following sections describe the main characteristics of such transforms.

The introduction of the centered transforms in the Insight Toolkit reflects the dynamic nature of a software library when it evolves in harmony with the requests of the community that it serves. This dynamism has, as everything else in real life, some advantages and some disadvantages. The main advantage is that when a need is identified by the users, it gets implemented in a matter of days or weeks. This capability for rapidly responding to the needs of a community is one of the major strengths of Open Source software. It has the additional safety that if the rest of the community does not wish to adopt a particular change, an isolated user can always implement that change in her local copy of the toolkit, since all the source code of ITK is available in a

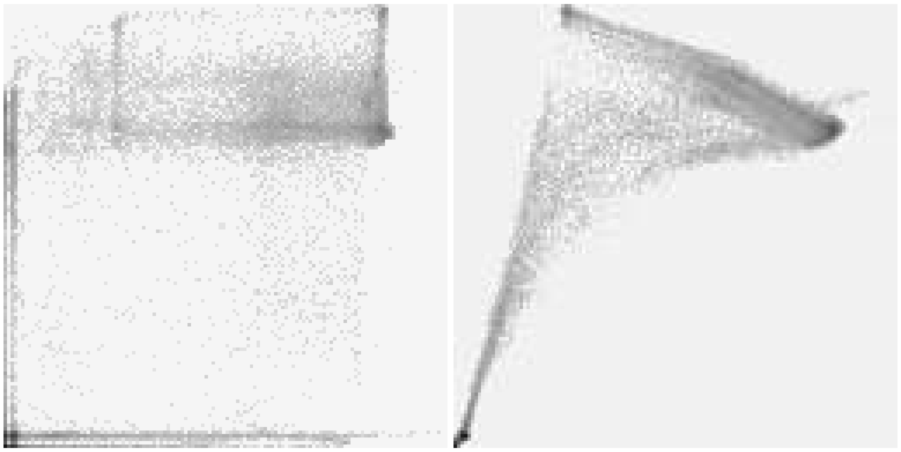


Figure 8.16: Joint entropy histograms before and after registration. The final transform was within half a pixel of true misalignment.

BSD-style license[9] that does not restrict modification nor distribution of the code, and that does not impose the assimilation demands of viral licenses such as GPL[10].

The main disadvantage of dynamism, is of course, the fact that there is continuous change and a need for perpetual adaptation. The evolution of software occurs at different scales, some changes happen to evolve in localized regions of the code, while from time to time accommodations of a larger scale are needed. The need for continuous changes is addressed in Extreme Programming with the methodology of *Refactoring*. At any given point, the structure of the code may not project the organized and neatly distributed architecture that may have resulted from a monolithic and static design. There is, after all, good reasons why living beings can not have straight angles. What you are about to witness in this section is a clear example of the diversity of species that flourishes when Evolution is in action [19].

8.6.1 Rigid Registration in 2D

The source code for this section can be found in the file
`Examples/Registration/ImageRegistration5.cxx`.

This example illustrates the use of the `itk::CenteredRigid2DTransform` for performing rigid registration in $2D$. The example code is for the most part identical to that presented in Section 8.2. The main difference is the use of the CenteredRigid2DTransform here instead of the `itk::TranslationTransform`.

In addition to the headers included in previous examples, the following header must also be

[9] http://www.opensource.org/licenses/bsd-license.php
[10] http://www.gnu.org/copyleft/gpl.html

included.

```
#include "itkCenteredRigid2DTransform.h"
```

The transform type is instantiated using the code below. The only template parameter for this class is the representation type of the space coordinates.

```
  typedef itk::CenteredRigid2DTransform< double > TransformType;
```

The transform object is constructed below and passed to the registration method.

```
  TransformType::Pointer  transform = TransformType::New();
  registration->SetTransform( transform );
```

In this example, the input images are taken from readers. The code below updates the readers in order to ensure that the image parameters (size, origin and spacing) are valid when used to initialize the transform. We intend to use the center of the fixed image as the rotation center and then use the vector between the fixed image center and the moving image center as the initial translation to be applied after the rotation.

```
  fixedImageReader->Update();
  movingImageReader->Update();
```

The center of rotation is computed using the origin, size and spacing of the fixed image.

```
  FixedImageType::Pointer fixedImage = fixedImageReader->GetOutput();

  const SpacingType fixedSpacing = fixedImage->GetSpacing();
  const OriginType  fixedOrigin  = fixedImage->GetOrigin();
  const RegionType  fixedRegion  = fixedImage->GetLargestPossibleRegion();
  const SizeType    fixedSize    = fixedRegion.GetSize();

  TransformType::InputPointType centerFixed;

  centerFixed[0] = fixedOrigin[0] + fixedSpacing[0] * fixedSize[0] / 2.0;
  centerFixed[1] = fixedOrigin[1] + fixedSpacing[1] * fixedSize[1] / 2.0;
```

The center of the moving image is computed in a similar way.

```
  MovingImageType::Pointer movingImage = movingImageReader->GetOutput();

  const SpacingType movingSpacing = movingImage->GetSpacing();
  const OriginType  movingOrigin  = movingImage->GetOrigin();
```

```
const RegionType  movingRegion  = movingImage->GetLargestPossibleRegion();
const SizeType    movingSize    = movingRegion.GetSize();

TransformType::InputPointType centerMoving;

centerMoving[0] = movingOrigin[0] + movingSpacing[0] * movingSize[0] / 2.0;
centerMoving[1] = movingOrigin[1] + movingSpacing[1] * movingSize[1] / 2.0;
```

The most straightforward method of initializing the transform parameters is to configure the transform and then get its parameters with the method `GetParameters()`. Here we initialize the transform by passing the center of the fixed image as the rotation center with the `SetCenter()` method. Then the translation is set as the vector relating the center of the moving image to the center of the fixed image. This last vector is passed with the method `SetTranslation()`.

```
transform->SetCenter( centerFixed );
transform->SetTranslation( centerMoving - centerFixed );
```

Let's finally initialize the rotation with a zero angle.

```
transform->SetAngle( 0.0 );
```

Now we pass the current transform's parameters as the initial parameters to be used when the registration process starts.

```
registration->SetInitialTransformParameters( transform->GetParameters() );
```

Keeping in mind that the scale of units in rotation and translation is quite different, we take advantage of the scaling functionality provided by the optimizers. We know that the first element of the parameters array corresponds to the angle that is measured in radians, while the other parameters correspond to translations that are measured in millimeters. For this reason we use small factors in the scales associated with translations and the coordinates of the rotation center .

```
typedef OptimizerType::ScalesType       OptimizerScalesType;
OptimizerScalesType optimizerScales( transform->GetNumberOfParameters() );
const double translationScale = 1.0 / 1000.0;

optimizerScales[0] = 1.0;
optimizerScales[1] = translationScale;
optimizerScales[2] = translationScale;
optimizerScales[3] = translationScale;
optimizerScales[4] = translationScale;

optimizer->SetScales( optimizerScales );
```

Next we set the normal parameters of the optimization method. In this case we are using an `itk::RegularStepGradientDescentOptimizer`. Below, we define the optimization parameters like the relaxation factor, initial step length, minimal step length and number of iterations. These last two act as stopping criteria for the optimization.

```
double initialStepLength = 0.1;

optimizer->SetRelaxationFactor( 0.6 );
optimizer->SetMaximumStepLength( initialStepLength );
optimizer->SetMinimumStepLength( 0.001 );
optimizer->SetNumberOfIterations( 200 );
```

Let's execute this example over two of the images provided in `Examples/Data`:

- `BrainProtonDensitySliceBorder20.png`
- `BrainProtonDensitySliceRotated10.png`

The second image is the result of intentionally rotating the first image by 10 degrees around the geometrical center of the image. Both images have unit-spacing and are shown in Figure 8.17. The registration takes 20 iterations and produces the results:

```
[0.177458, 110.489, 128.488, 0.0106296, 0.00194103]
```

These results are interpreted as

- Angle = 0.177458 radians
- Center = $(110.489, 128.488)$ millimeters
- Translation = $(0.0106296, 0.00194103)$ millimeters

As expected, these values match the misalignment intentionally introduced into the moving image quite well, since 10 degrees is about 0.174532 radians.

Figure 8.18 shows from left to right the resampled moving image after registration, the difference between fixed and moving images before registration, and the difference between fixed and resampled moving image after registration. It can be seen from the last difference image that the rotational component has been solved but that a small centering misalignment persists.

Figure 8.19 shows plots of the main output parameters produced from the registration process. This includes, the metric values at every iteration, the angle values at every iteration, and the translation components of the transform as the registration progress.

Let's now consider the case in which rotations and translations are present in the initial registration, as in the following pair of images:

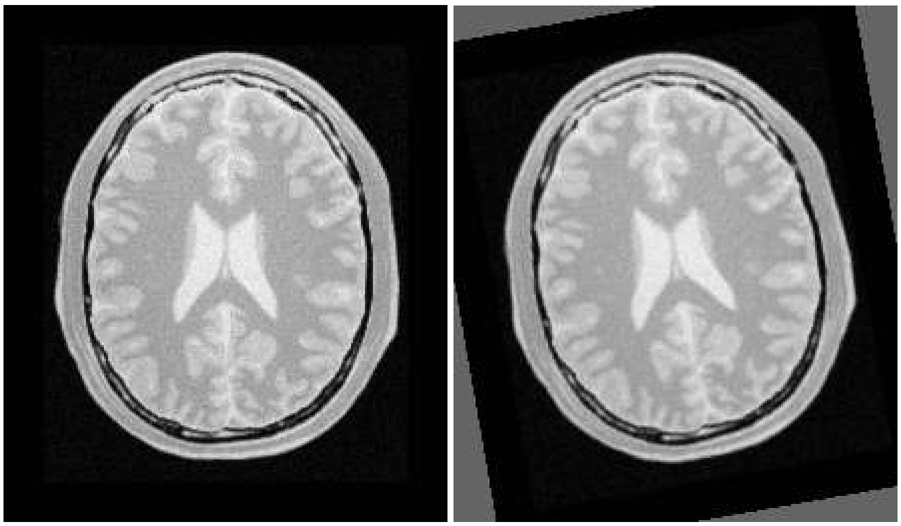


Figure 8.17: Fixed and moving images are provided as input to the registration method using the CenteredRigid2D transform.

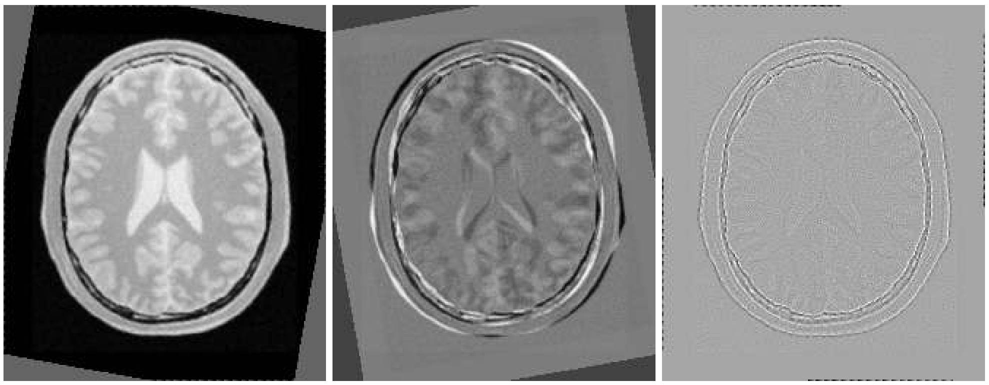


Figure 8.18: Resampled moving image (left). Differences between the fixed and moving images, before (center) and after (right) registration using the CenteredRigid2D transform.

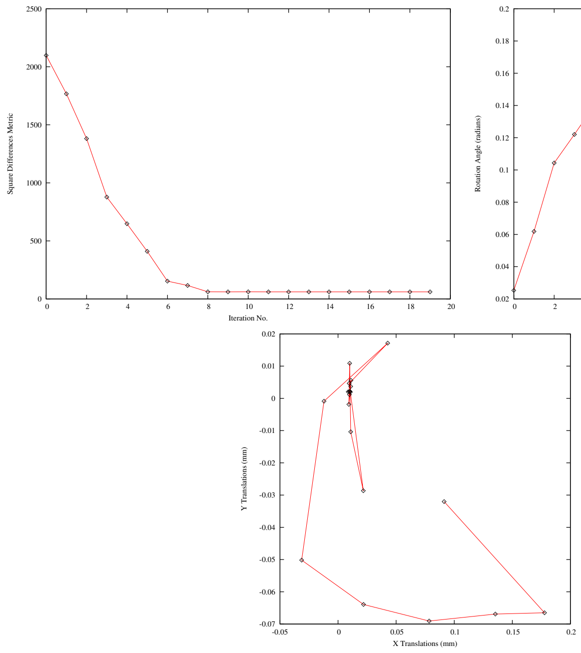


Figure 8.19: Metric values, rotation angle and translations during registration with the CenteredRigid2D transform.

- `BrainProtonDensitySliceBorder20.png`
- `BrainProtonDensitySliceR10X13Y17.png`

The second image is the result of intentionally rotating the first image by 10 degrees and then translating it $13mm$ in X and $17mm$ in Y. Both images have unit-spacing and are shown in Figure 8.20. In order to accelerate convergence it is convenient to use a larger step length as shown here.

```
optimizer->SetMaximumStepLength( 1.0 );
```

The registration now takes 46 iterations and produces the following results:

```
    [0.174454, 110.361, 128.647, 12.977, 15.9761]
```

These parameters are interpreted as

- Angle = 0.174454 radians
- Center = $(110.361, 128.647)$ millimeters
- Translation = $(12.977, 15.9761)$ millimeters

These values approximately match the initial misalignment intentionally introduced into the moving image, since 10 degrees is about 0.174532 radians. The horizontal translation is well resolved while the vertical translation ends up being off by about one millimeter.

Figure 8.21 shows the output of the registration. The rightmost image of this figure shows the difference between the fixed image and the resampled moving image after registration.

Figure 8.22 shows plots of the main output registration parameters when the rotation and translations are combined. These results include, the metric values at every iteration, the angle values at every iteration, and the translation components of the registration as the registration converges. It can be seen from the smoothness of these plots that a larger step length could have been supported easily by the optimizer. You may want to modify this value in order to get a better idea of how to tune the parameters.

8.6.2 Initializing with Image Moments

The source code for this section can be found in the file
`Examples/Registration/ImageRegistration6.cxx`.

This example illustrates the use of the `itk::CenteredRigid2DTransform` for performing registration. The example code is for the most part identical to the one presented in Section 8.6.1. Even though this current example is done in $2D$, the class

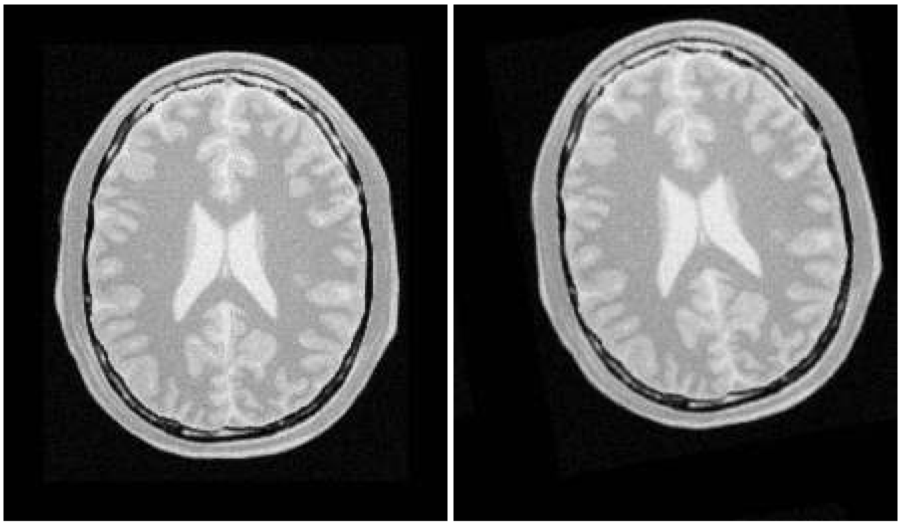


Figure 8.20: Fixed and moving images provided as input to the registration method using the Centered-Rigid2D transform.

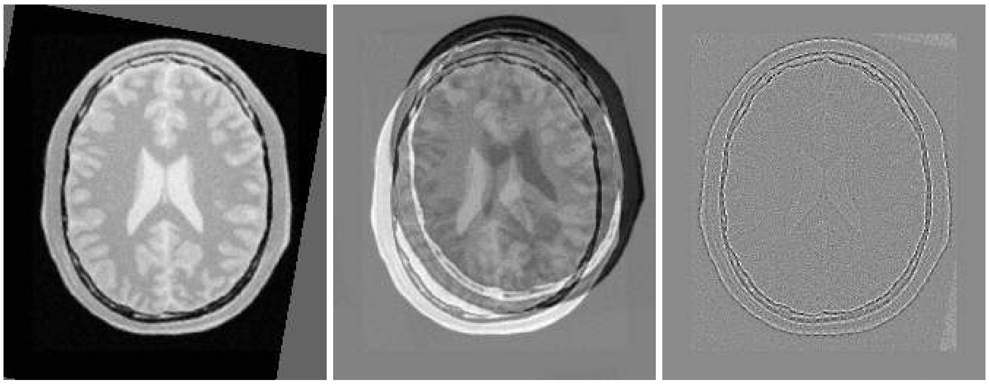


Figure 8.21: Resampled moving image (left). Differences between the fixed and moving images, before (center) and after (right) registration with the CenteredRigid2D transform.

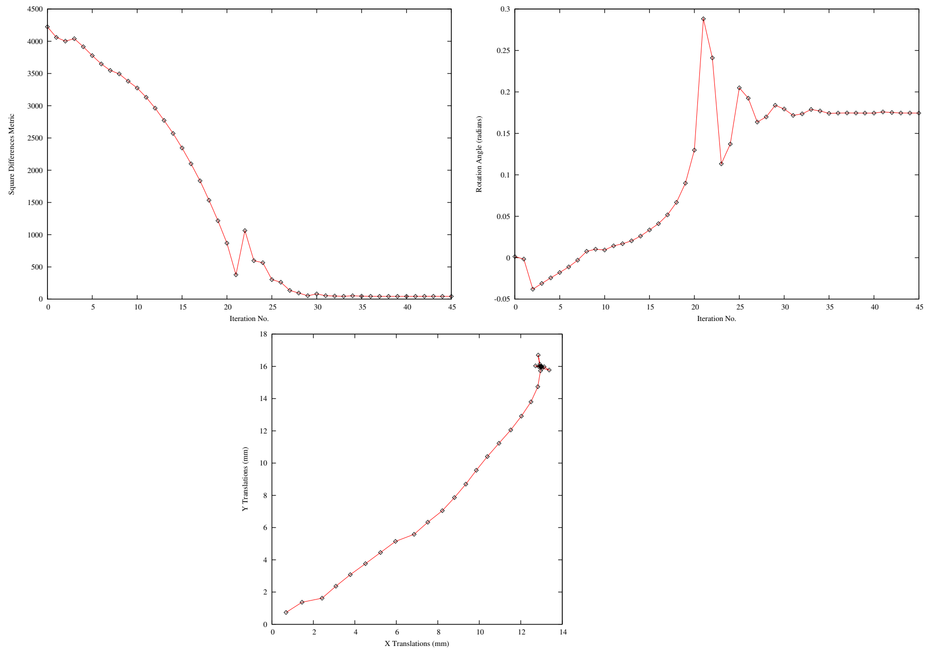


Figure 8.22: Metric values, rotation angle and translations during the registration using the Centered-Rigid2D transform on an image with rotation and translation mis-registration.

itk::CenteredTransformInitializer is quite generic and could be used in other dimensions. The objective of the initializer class is to simplify the computation of the center of rotation and the translation required to initialize certain transforms such as the CenteredRigid2DTransform. The initializer accepts two images and a transform as inputs. The images are considered to be the fixed and moving images of the registration problem, while the transform is the one used to register the images.

The CenteredRigid2DTransform supports two modes of operation. In the first mode, the centers of the images are computed as space coordinates using the image origin, size and spacing. The center of the fixed image is assigned as the rotational center of the transform while the vector going from the fixed image center to the moving image center is passed as the initial translation of the transform. In the second mode, the image centers are not computed geometrically but by using the moments of the intensity gray levels. The center of mass of each image is computed using the helper class itk::ImageMomentsCalculator. The center of mass of the fixed image is passed as the rotational center of the transform while the vector going from the fixed image center of mass to the moving image center of mass is passed as the initial translation of the transform. This second mode of operation is quite convenient when the anatomical structures of interest are not centered in the image. In such cases the alignment of the centers of mass provides a better rough initial registration than the simple use of the geometrical centers. The validity of the initial registration should be questioned when the two images are acquired in different imaging modalities. In those cases, the center of mass of intensities in one modality does not necessarily matches the center of mass of intensities in the other imaging modality.

The following are the most relevant headers in this example.

```
#include "itkCenteredRigid2DTransform.h"
#include "itkCenteredTransformInitializer.h"
```

The transform type is instantiated using the code below. The only template parameter of this class is the representation type of the space coordinates.

```
  typedef itk::CenteredRigid2DTransform< double > TransformType;
```

The transform object is constructed below and passed to the registration method.

```
  TransformType::Pointer  transform = TransformType::New();
  registration->SetTransform( transform );
```

The input images are taken from readers. It is not necessary to explicitly call `Update()` on the readers since the CenteredTransformInitializer class will do it as part of its initialization. The following code instantiates the initializer. This class is templated over the fixed and moving image type as well as the transform type. An initializer is then constructed by calling the `New()` method and assigning the result to a itk::SmartPointer.

```
typedef itk::CenteredTransformInitializer<
                                  TransformType,
                                  FixedImageType,
                                  MovingImageType >  TransformInitializerType;

TransformInitializerType::Pointer initializer = TransformInitializerType::New();
```

The initializer is now connected to the transform and to the fixed and moving images.

```
initializer->SetTransform(   transform );
initializer->SetFixedImage(  fixedImageReader->GetOutput() );
initializer->SetMovingImage( movingImageReader->GetOutput() );
```

The use of the geometrical centers is selected by calling `GeometryOn()` while the use of center of mass is selected by calling `MomentsOn()`. Below we select the center of mass mode.

```
initializer->MomentsOn();
```

Finally, the computation of the center and translation is triggered by the `InitializeTransform()` method. The resulting values will be passed directly to the transform.

```
initializer->InitializeTransform();
```

The remaining parameters of the transform are initialized as before.

```
transform->SetAngle( 0.0 );
```

Now the parameters of the current transform are passed as the initial parameters to be used when the registration process starts.

```
registration->SetInitialTransformParameters( transform->GetParameters() );
```

Let's execute this example over some of the images provided in `Examples/Data`, for example:

- `BrainProtonDensitySliceBorder20.png`
- `BrainProtonDensitySliceR10X13Y17.png`

The second image is the result of intentionally rotating the first image by 10 degrees and shifting it 13mm in X and 17mm in Y. Both images have unit-spacing and are shown in Figure 8.17. The registration takes 22 iterations and produces:

```
    [0.174475, 111.177, 131.572, 12.4566, 16.0729]
```

These parameters are interpreted as

- Angle = 0.174475 radians
- Center = $(111.177, 131.572)$ millimeters
- Translation = $(12.4566, 16.0729)$ millimeters

Note that the reported translation is not the translation of $(13, 17)$ that might be expected. The reason is that the five parameters of the CenteredRigid2DTransform are redundant. The actual movement in space is described by only 3 parameters. This means that there are infinite combinations of rotation center and translations that will represent the same actual movement in space. It is more illustrative in this case to take a look at the actual rotation matrix and offset resulting form the five parameters.

```
  transform->SetParameters( finalParameters );

  TransformType::MatrixType matrix = transform->GetRotationMatrix();
  TransformType::OffsetType offset = transform->GetOffset();

  std::cout << "Matrix = " << std::endl << matrix << std::endl;
  std::cout << "Offset = " << std::endl << offset << std::endl;
```

Which produces the following output.

```
    Matrix =
       0.984818 -0.173591
       0.173591 0.984818

    Offset =
       [36.9843, -1.22896]
```

This output illustrates how counter-intuitive the mix of center of rotation and translations can be. Figure 8.23 will clarify this situation. The figure shows the original image on the left. A rotation of 10° around the center of the image is shown in the middle. The same rotation performed around the origin of coordinates is shown on the right. It can be seen here that changing the center of rotation introduces additional translations.

Let's analyze what happens to the center of the image that we just registered. Under the point of view of rotating 10° around the center and then applying a translation of $(13mm, 17mm)$.

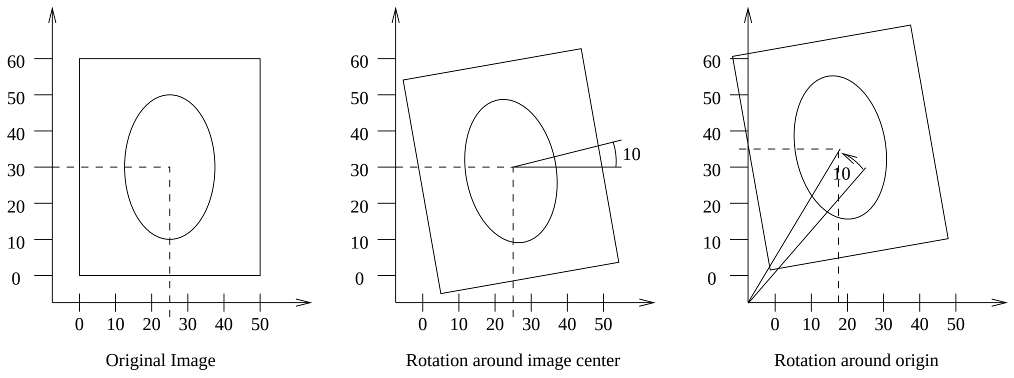


Figure 8.23: Effect of changing the center of rotation.

The image has a size of (221×257) pixels and unit spacing. Hence its center has coordinates $(110.5, 128.5)$. Since the rotation is done around this point, the center behaves as the fixed point of the transformation and remains unchanged. Then with the $(13mm, 17mm)$ translation it is mapped to $(123.5, 145.5)$ which becomes its final position.

The matrix and offset that we obtained at the end of the registration indicate that this should be equivalent to a rotation of $10°$ around the origin, followed by a translations of $(36.98, -1.22)$. Let's compute this in detail. First the rotation of the image center by $10°$ around the origin will move the point to $(86.52, 147.97)$. Now, applying a translation of $(36.98, -1.22)$ maps this point to $(123.5, 146.75)$. Which is close to the result of our previous computation.

It is unlikely that we could have chosen such translations as the initial guess, since we tend to think about image in a coordinate system whose origin is in the center of the image.

You may be wondering why the actual movement is represented by three parameters when we take the trouble of using five. In particular, why use a 5-dimensional optimizer space instead of a 3-dimensional one. The answer is that by using five parameters we have a much simpler way of initializing the transform with the rotation matrix and offset. Using the minimum three parameters it is not obvious how to determine what the initial rotation and translations should be.

Figure 8.25 shows the output of the registration. The image on the right of this figure shows the differences between the fixed image and the resampled moving image after registration.

Figure 8.26 plots the output parameters of the registration process. It includes, the metric values at every iteration, the angle values at every iteration, and the values of the translation components as the registration progress. Note that this is the complementary translation as used in the transform, not the actual total translation that is used in the transform offset. We could modify the observer to print the total offset instead of printing the array of parameters. Let's call that an exercise for the reader!

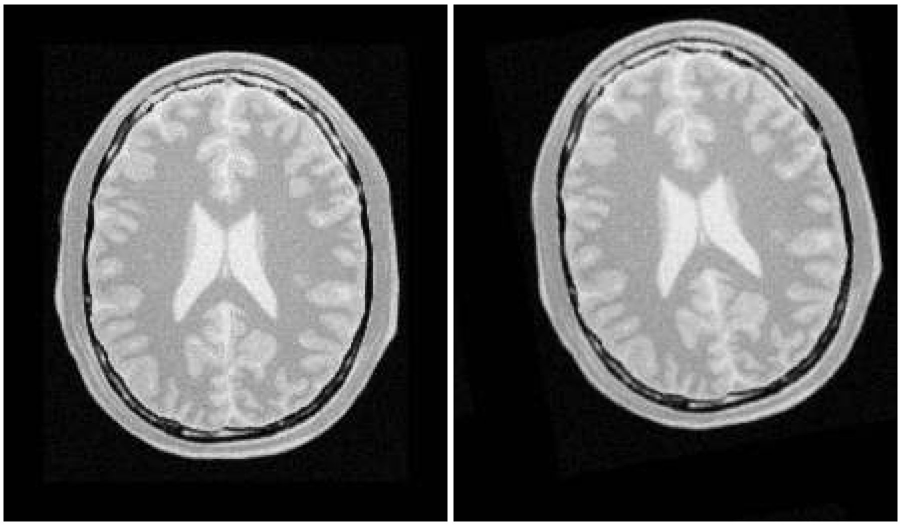


Figure 8.24: Fixed and moving images provided as input to the registration method using CenteredTransformInitializer.

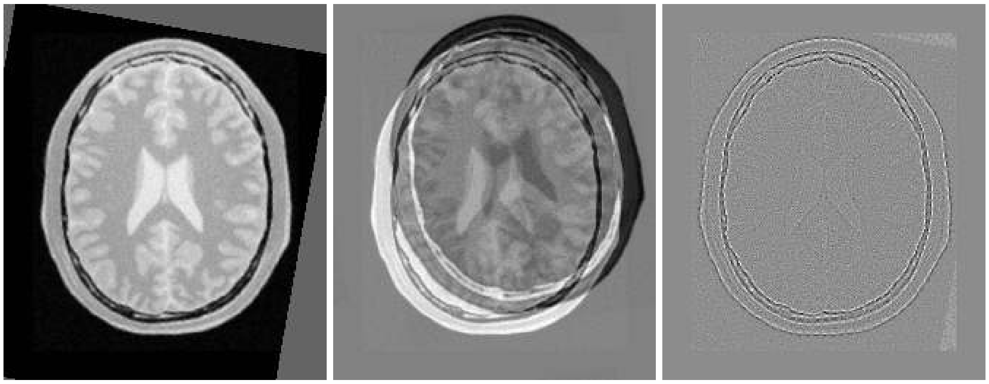


Figure 8.25: Resampled moving image (left). Differences between fixed and moving images, before registration (center) and after registration (right) with the CenteredTransformInitializer.

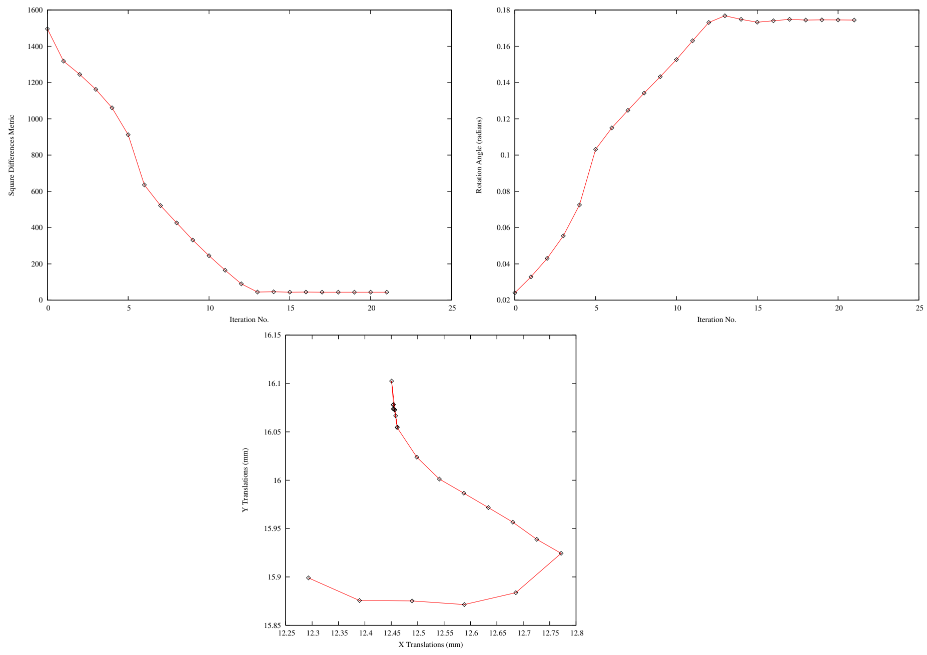


Figure 8.26: Plots of the Metric, rotation angle, center of rotation and translations during the registration using CenteredTransformInitializer.

8.6.3 Similarity Transform in 2D

The source code for this section can be found in the file
`Examples/Registration/ImageRegistration7.cxx`.

This example illustrates the use of the `itk::CenteredSimilarity2DTransform` class for performing registration in 2*D*. The of example code is for the most part identical to the code presented in Section 8.6.2. The main difference is the use of `itk::CenteredSimilarity2DTransform` here rather than the `itk::CenteredRigid2DTransform` class.

A similarity transform can be seen as a composition of rotations, translations and uniform scaling. It preserves angles and map lines into lines. This transform is implemented in the toolkit as deriving from a rigid 2*D* transform and with a scale parameter added.

When using this transform, attention should be paid to the fact that scaling and translations are not independent. In the same way that rotations can locally be seen as translations, scaling also result in local displacements. Scaling is performed in general with respect to the origin of coordinates. However, we already saw how ambiguous that could be in the case of rotations. For this reason, this transform also allows users to setup a specific center. This center is use both for rotation and scaling.

In addition to the headers included in previous examples, here the following header must be included.

```
#include "itkCenteredSimilarity2DTransform.h"
```

The Transform class is instantiated using the code below. The only template parameter of this class is the representation type of the space coordinates.

```
  typedef itk::CenteredSimilarity2DTransform< double > TransformType;
```

The transform object is constructed below and passed to the registration method.

```
  TransformType::Pointer  transform = TransformType::New();
  registration->SetTransform( transform );
```

In this example, we again use the helper class `itk::CenteredTransformInitializer` to compute a reasonable value for the initial center of rotation and the translation.

```
  typedef itk::CenteredTransformInitializer<
                                    TransformType,
                                    FixedImageType,
                                    MovingImageType >  TransformInitializerType;
```

```
TransformInitializerType::Pointer initializer = TransformInitializerType::New();

initializer->SetTransform(   transform );

initializer->SetFixedImage(  fixedImageReader->GetOutput() );
initializer->SetMovingImage( movingImageReader->GetOutput() );

initializer->MomentsOn();

initializer->InitializeTransform();
```

The remaining parameters of the transform are initialized below.

```
transform->SetScale( initialScale );
transform->SetAngle( initialAngle );
```

We now pass the parameter of the current transform as the initial parameters to be used when the registration process starts.

```
registration->SetInitialTransformParameters( transform->GetParameters() );
```

Keeping in mind that the scale of units in scaling, rotation and translation are quite different, we take advantage of the scaling functionality provided by the optimizers. We know that the first element of the parameters array corresponds to the scale factor, the second corresponds to the angle, third and forth are the center of rotation and fifth and sixth are the remaining translation. We use henceforth small factors in the scales associated with translations and the rotation center.

```
typedef OptimizerType::ScalesType       OptimizerScalesType;
OptimizerScalesType optimizerScales( transform->GetNumberOfParameters() );
const double translationScale = 1.0 / 100.0;

optimizerScales[0] = 10.0;
optimizerScales[1] =  1.0;
optimizerScales[2] =  translationScale;
optimizerScales[3] =  translationScale;
optimizerScales[4] =  translationScale;
optimizerScales[5] =  translationScale;

optimizer->SetScales( optimizerScales );
```

We set also the normal parameters of the optimization method. In this case we are using A `itk::RegularStepGradientDescentOptimizer`. Below, we define the optimization parameters like initial step length, minimal step length and number of iterations. These last two act as stopping criteria for the optimization.

```
optimizer->SetMaximumStepLength( steplength );
optimizer->SetMinimumStepLength( 0.0001 );
optimizer->SetNumberOfIterations( 500 );
```

Let's execute this example over some of the images provided in `Examples/Data`, for example:

- `BrainProtonDensitySliceBorder20.png`
- `BrainProtonDensitySliceR10X13Y17S12.png`

The second image is the result of intentionally rotating the first image by 10 degrees, scaling by $1/1.2$ and then translating by $(-13,-17)$. Both images have unit-spacing and are shown in Figure 8.27. The registration takes 16 iterations and produces:

```
[0.833222, -0.174521, 111.437, 131.741, -12.8272, -12.7862]
```

That are interpreted as

- Scale factor = 0.833222
- Angle = 0.174521 radians
- Center = $(111.437, 131.741)$ millimeters
- Translation = $(-12.8272, -12.7862)$ millimeters

These values approximate the misalignment intentionally introduced into the moving image. Since 10 degrees is about 0.174532 radians.

Figure 8.28 shows the output of the registration. The right image shows the squared magnitude of pixel differences between the fixed image and the resampled moving image.

Figure 8.29 shows the plots of the main output parameters of the registration process. The metric values at every iteration are shown on the top. The angle values are shown in the plot at left while the translation components of the registration are presented in the plot at right.

8.6.4 Rigid Transform in 3D

The source code for this section can be found in the file
`Examples/Registration/ImageRegistration8.cxx`.

This example illustrates the use of the `itk::VersorRigid3DTransform` class for performing registration of two $3D$ images. The example code is for the most part identical to the code presented in Section 8.6.1. The major difference is that this example is done in $3D$. The class

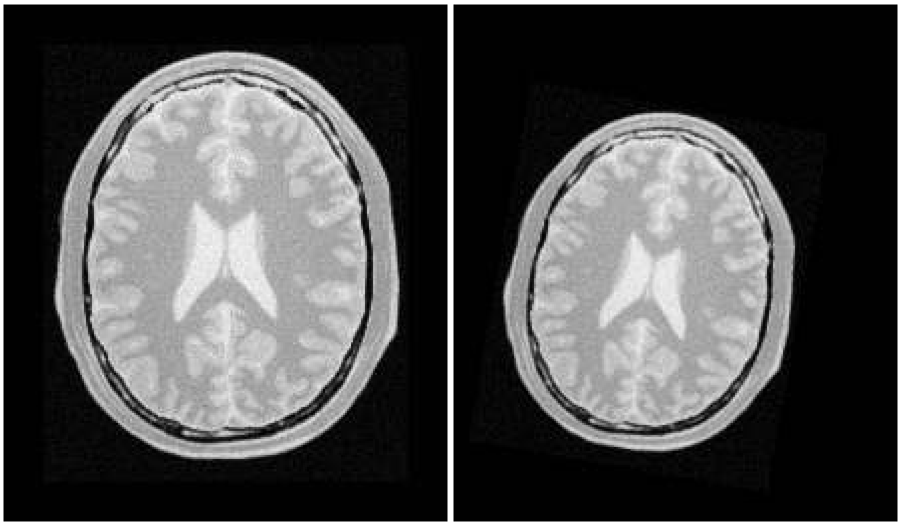


Figure 8.27: Fixed and Moving image provided as input to the registration method using the Similarity2D transform.

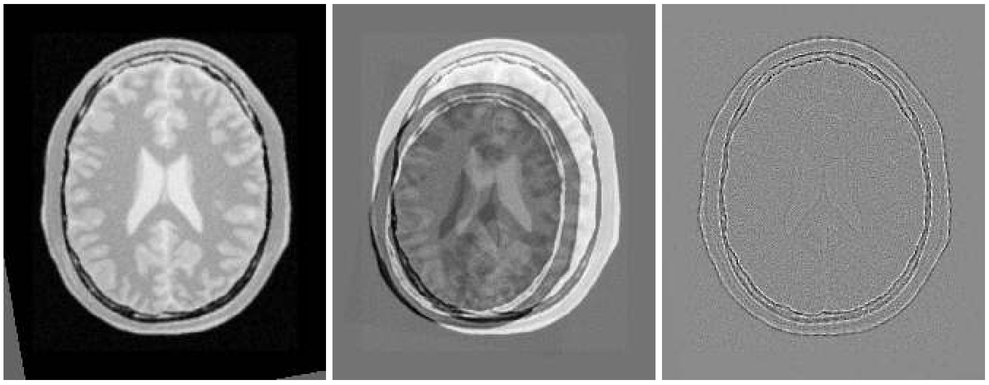


Figure 8.28: Resampled moving image (left). Differences between fixed and moving images, before (center) and after (right) registration with the Similarity2D transform.

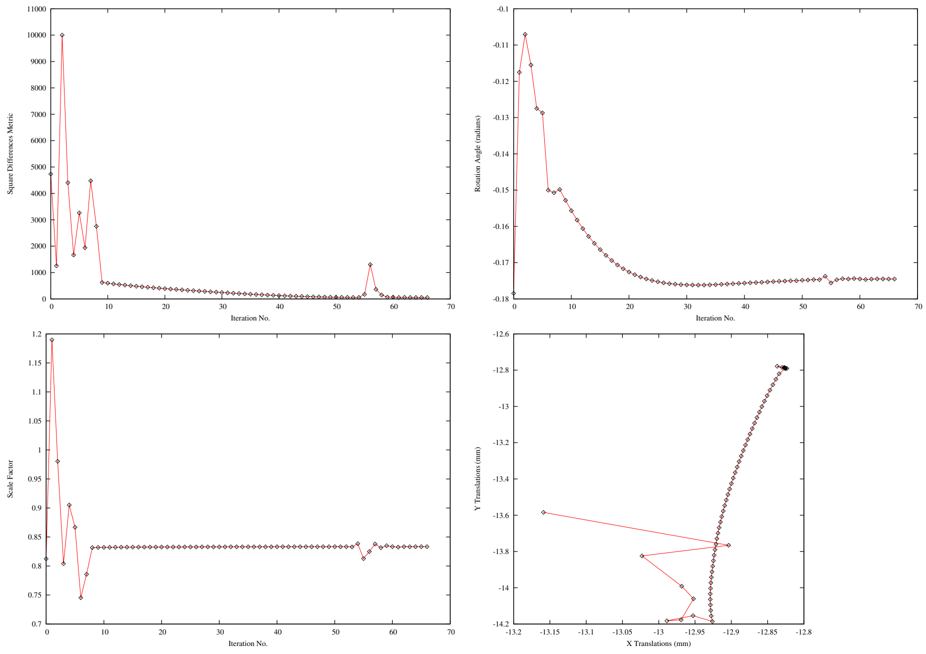


Figure 8.29: Plots of the Metric, rotation angle and translations during the registration using Similarity2D transform.

itk::CenteredTransformInitializer is used to initialize the center and translation of the transform. The case of rigid registration of 3D images is probably one of the most commonly found cases of image registration.

The following are the most relevant headers of this example.

```
#include "itkVersorRigid3DTransform.h"
#include "itkCenteredTransformInitializer.h"
```

The parameter space of the VersorRigid3DTransform is not a vector space, due to the fact that addition is not a closed operation in the space of versor components. This precludes the use of standard gradient descent algorithms for optimizing the parameter space of this transform. A special optimizer should be used in this registration configuration. The optimizer designed for this transform is the itk::VersorRigid3DTransformOptimizer. This optimizer uses Versor composition for updating the first three components of the parameters array, and Vector addition for updating the last three components of the parameters array [34, 42].

```
#include "itkVersorRigid3DTransformOptimizer.h"
```

The Transform class is instantiated using the code below. The only template parameter to this class is the representation type of the space coordinates.

```
  typedef itk::VersorRigid3DTransform< double > TransformType;
```

The transform object is constructed below and passed to the registration method.

```
  TransformType::Pointer  transform = TransformType::New();
  registration->SetTransform( transform );
```

The input images are taken from readers. It is not necessary here to explicitly call Update() on the readers since the itk::CenteredTransformInitializer will do it as part of its computations. The following code instantiates the type of the initializer. This class is templated over the fixed and moving image type as well as the transform type. An initializer is then constructed by calling the New() method and assigning the result to a smart pointer.

```
  typedef itk::CenteredTransformInitializer< TransformType,
                                             FixedImageType,
                                             MovingImageType
                                                 >  TransformInitializerType;

  TransformInitializerType::Pointer initializer =
                                        TransformInitializerType::New();
```

The initializer is now connected to the transform and to the fixed and moving images.

```
  initializer->SetTransform(   transform );
  initializer->SetFixedImage(  fixedImageReader->GetOutput() );
  initializer->SetMovingImage( movingImageReader->GetOutput() );
```

The use of the geometrical centers is selected by calling `GeometryOn()` while the use of center of mass is selected by calling `MomentsOn()`. Below we select the center of mass mode.

```
  initializer->MomentsOn();
```

Finally, the computation of the center and translation is triggered by the `InitializeTransform()` method. The resulting values will be passed directly to the transform.

```
  initializer->InitializeTransform();
```

The rotation part of the transform is initialized using a `itk::Versor` which is simply a unit quaternion. The `VersorType` can be obtained from the transform traits. The versor itself defines the type of the vector used to indicate the rotation axis. This trait can be extracted as `VectorType`. The following lines create a versor object and initialize its parameters by passing a rotation axis and an angle.

```
  typedef TransformType::VersorType  VersorType;
  typedef VersorType::VectorType     VectorType;

  VersorType     rotation;
  VectorType     axis;

  axis[0] = 0.0;
  axis[1] = 0.0;
  axis[2] = 1.0;

  const double angle = 0;

  rotation.Set(  axis, angle  );

  transform->SetRotation( rotation );
```

We now pass the parameters of the current transform as the initial parameters to be used when the registration process starts.

```
  registration->SetInitialTransformParameters( transform->GetParameters() );
```

Let's execute this example over some of the images available in the ftp site ftp://public.kitware.com/pub/itk/Data/BrainWeb

Note that the images in the ftp site are compressed in `.tgz` files. You should download these files an uncompress them in your local system. After decompressing and extracting the files you could take a pair of volumes, for example the pair:

- `brainweb165a10f17.mha`
- `brainweb165a10f17Rot10Tx15.mha`

The second image is the result of intentionally rotating the first image by 10 degrees around the origin and shifting it 15mm in X. The registration takes 24 iterations and produces:

```
[-6.03744e-05, 5.91487e-06, -0.0871932, 2.64659, -17.4637, -0.00232496]
```

That are interpreted as

- Versor = $(-6.03744e-05, 5.91487e-06, -0.0871932)$
- Translation = $(2.64659, -17.4637, -0.00232496)$ millimeters

This Versor is equivalent to a rotation of 9.98 degrees around the Z axis.

Note that the reported translation is not the translation of $(15.0, 0.0, 0.0)$ that we may be naively expecting. The reason is that the `VersorRigid3DTransform` is applying the rotation around the center found by the `CenteredTransformInitializer` and then adding the translation vector shown above.

It is more illustrative in this case to take a look at the actual rotation matrix and offset resulting form the 6 parameters.

```
transform->SetParameters( finalParameters );

TransformType::MatrixType matrix = transform->GetRotationMatrix();
TransformType::OffsetType offset = transform->GetOffset();

std::cout << "Matrix = " << std::endl << matrix << std::endl;
std::cout << "Offset = " << std::endl << offset << std::endl;
```

The output of this print statements is

```
Matrix =
    0.984795 0.173722 2.23132e-05
    -0.173722 0.984795 0.000119257
```

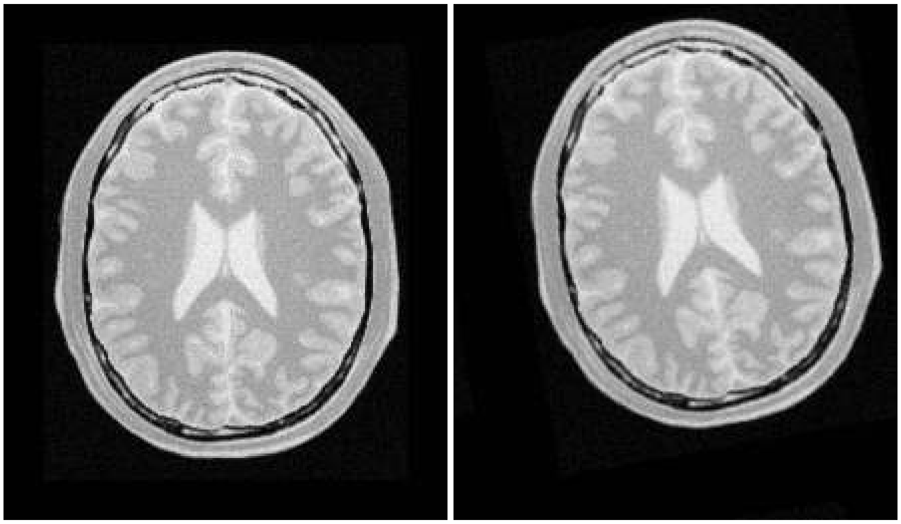


Figure 8.30: Fixed and moving image provided as input to the registration method using CenteredTransformInitializer.

```
        -1.25621e-06 -0.00012132 1

    Offset =
        [-15.0105, -0.00672343, 0.0110854]
```

From the rotation matrix it is possible to deduce that the rotation is happening in the X,Y plane and that the angle is on the order of $\arcsin(0.173722)$ which is very close to 10 degrees, as we expected.

Figure 8.31 shows the output of the registration. The center image in this figure shows the differences between the fixed image and the resampled moving image before the registration. The image on the right side presents the difference between the fixed image and the resampled moving image after the registration has been performed. Note that these images are individual slices extracted from the actual volumes. For details, look at the source code of this example, where the ExtractImageFilter is used to extract a slice from the the center of each one of the volumes. One of the main purposes of this example is to illustrate that the toolkit can perform registration on images of any dimension. The only limitations are, as usual, the amount of memory available for the images and the amount of computation time that it will take to complete the optimization process.

Figure 8.32 shows the plots of the main output parameters of the registration process. The metric values at every iteration. The Z component of the versor is plotted as an indication of how the rotation progress. The X,Y translation components of the registration are plotted at every iteration too.

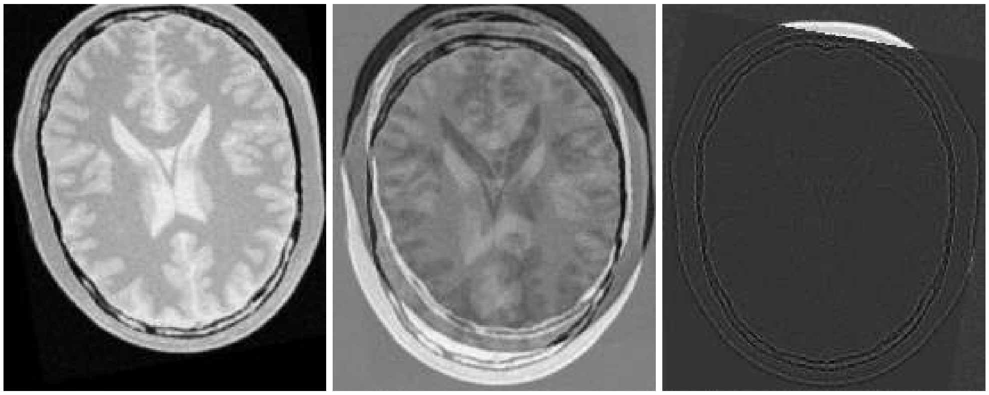


Figure 8.31: Resampled moving image (left). Differences between fixed and moving images, before (center) and after (right) registration with the CenteredTransformInitializer.

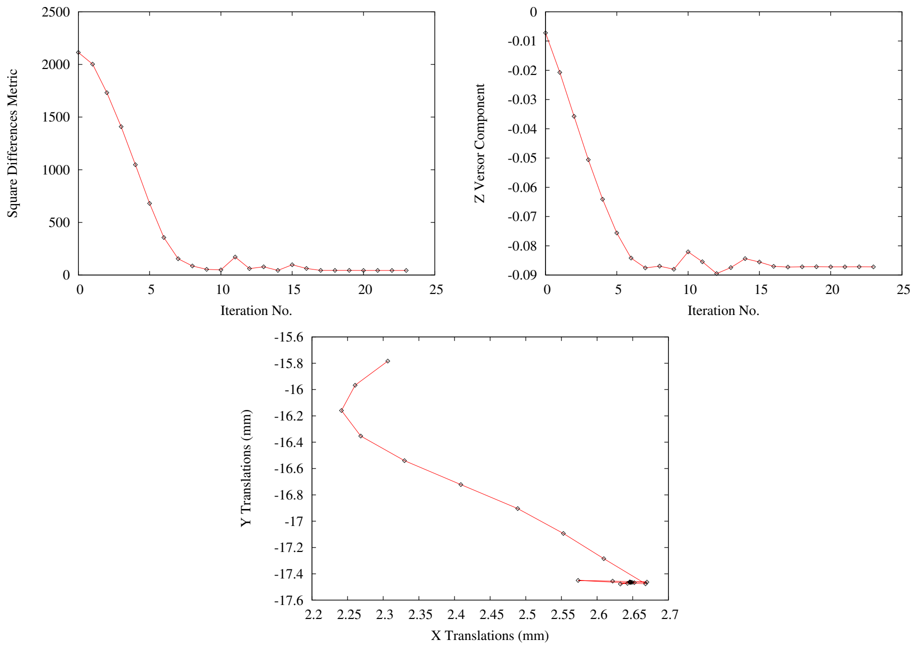


Figure 8.32: Plots of the metric, rotation angle, center of rotation and translations during the registration using CenteredTransformInitializer.

Shell and Gnuplot scripts for generating the diagrams in Figure 8.32 are available in the directory

`InsightDocuments/SoftwareGuide/Art`

You are strongly encouraged to run the example code, since only in this way you can gain a first hand experience with the behavior of the registration process. Once again, this is a simple reflection of the philosophy that we put forward in this book:

If you can not replicate it, then it does not exist!.

We have seen enough published papers with pretty pictures, presenting results that in practice are impossible to replicate. That is vanity, not science.

8.6.5 Centered Affine Transform

The source code for this section can be found in the file
`Examples/Registration/ImageRegistration9.cxx`.

This example illustrates the use of the `itk::AffineTransform` for performing registration in 2*D*. The example code is, for the most part, identical to that in 8.6.2. The main difference is the use of the AffineTransform here instead of the `itk::CenteredRigid2DTransform`. We will focus on the most relevant changes in the current code and skip the basic elements already explained in previous examples.

Let's start by including the header file of the AffineTransform.

```
#include "itkAffineTransform.h"
```

We define then the types of the images to be registered.

```
  const    unsigned int    Dimension = 2;
  typedef  float           PixelType;

  typedef itk::Image< PixelType, Dimension >  FixedImageType;
  typedef itk::Image< PixelType, Dimension >  MovingImageType;
```

The transform type is instantiated using the code below. The template parameters of this class are the representation type of the space coordinates and the space dimension.

```
  typedef itk::AffineTransform<
                                  double,
                                  Dimension  >     TransformType;
```

The transform object is constructed below and passed to the registration method.

```
TransformType::Pointer  transform = TransformType::New();
registration->SetTransform( transform );
```

In this example, we again use the itk::CenteredTransformInitializer helper class in order to compute a reasonable value for the initial center of rotation and the translation. The initializer is set to use the center of mass of each image as the initial correspondence correction.

```
typedef itk::CenteredTransformInitializer<
                                  TransformType,
                                  FixedImageType,
                                  MovingImageType >  TransformInitializerType;
TransformInitializerType::Pointer initializer = TransformInitializerType::New();
initializer->SetTransform(   transform );
initializer->SetFixedImage(  fixedImageReader->GetOutput() );
initializer->SetMovingImage( movingImageReader->GetOutput() );
initializer->MomentsOn();
initializer->InitializeTransform();
```

Now we pass the parameters of the current transform as the initial parameters to be used when the registration process starts.

```
registration->SetInitialTransformParameters(
                               transform->GetParameters() );
```

Keeping in mind that the scale of units in scaling, rotation and translation are quite different, we take advantage of the scaling functionality provided by the optimizers. We know that the first $N \times N$ elements of the parameters array correspond to the rotation matrix factor, the next N correspond to the rotation center, and the last N are the components of the translation to be applied after multiplication with the matrix is performed.

```
typedef OptimizerType::ScalesType       OptimizerScalesType;
OptimizerScalesType optimizerScales( transform->GetNumberOfParameters() );

optimizerScales[0] =  1.0;
optimizerScales[1] =  1.0;
optimizerScales[2] =  1.0;
optimizerScales[3] =  1.0;
optimizerScales[4] =  translationScale;
optimizerScales[5] =  translationScale;

optimizer->SetScales( optimizerScales );
```

We also set the usual parameters of the optimization method. In this case we are using an itk::RegularStepGradientDescentOptimizer. Below, we define the optimization parameters like initial step length, minimal step length and number of iterations. These last two act as stopping criteria for the optimization.

```
optimizer->SetMaximumStepLength( steplength );
optimizer->SetMinimumStepLength( 0.0001 );
optimizer->SetNumberOfIterations( maxNumberOfIterations );
```

We also set the optimizer to do minimization by calling the `MinimizeOn()` method.

```
optimizer->MinimizeOn();
```

Finally we trigger the execution of the registration method by calling the `Update()` method. The call is placed in a `try/catch` block in case any exceptions are thrown.

```
try
  {
  registration->StartRegistration();
  }
catch( itk::ExceptionObject & err )
  {
  std::cerr << "ExceptionObject caught !" << std::endl;
  std::cerr << err << std::endl;
  return -1;
  }
```

Once the optimization converges, we recover the parameters from the registration method. This is done with the `GetLastTransformParameters()` method. We can also recover the final value of the metric with the `GetValue()` method and the final number of iterations with the `GetCurrentIteration()` method.

```
OptimizerType::ParametersType finalParameters =
                  registration->GetLastTransformParameters();

const double finalRotationCenterX = transform->GetCenter()[0];
const double finalRotationCenterY = transform->GetCenter()[1];
const double finalTranslationX    = finalParameters[4];
const double finalTranslationY    = finalParameters[5];

const unsigned int numberOfIterations = optimizer->GetCurrentIteration();
const double bestValue = optimizer->GetValue();
```

Let's execute this example over two of the images provided in `Examples/Data`:

- `BrainProtonDensitySliceBorder20.png`
- `BrainProtonDensitySliceR10X13Y17.png`

The second image is the result of intentionally rotating the first image by 10 degrees and then translating by $(-13,-17)$. Both images have unit-spacing and are shown in Figure 8.33. We execute the code using the following parameters: step length=1.0, translation scale= 0.0001 and maximum number of iterations = 300. With these images and parameters the registration takes 98 iterations and produces

```
96 58.09 [0.986481, -0.169104, 0.166411, 0.986174, 12.461, 16.0754]
```

These results are interpreted as

- Iterations = 98
- Final Metric = 58.09
- Center = $(111.204, 131.6)$ millimeters
- Translation = $(12.461, 16.0754)$ millimeters
- Affine scales = $(1.00185, .999137)$

The second component of the matrix values is usually associated with $\sin\theta$. We obtain the rotation through SVD of the affine matrix. The value is 9.6526 degrees, which is approximately the intentional misalignment of 10.0 degrees.

Figure 8.34 shows the output of the registration. The right most image of this figure shows the squared magnitude difference between the fixed image and the resampled moving image.

Figure 8.35 shows the plots of the main output parameters of the registration process. The metric values at every iteration are shown on the top plot. The angle values are shown on the bottom left plot, while the translation components of the registration are presented on the bottom right plot. Note that the final total offset of the transform is to be computed as a combination of the shift due rotation plus the explicit translation set on the transform.

8.7 Multi-Resolution Registration

Performing image registration using a multi-resolution approach is widely used to improve speed, accuracy and robustness. The basic idea is that registration is first performed at a coarse scale where the images have fewer pixels. The spatial mapping determined at the coarse level is then used to initialize registration at the next finer scale. This process is repeated until it reaches the finest possible scale. This coarse-to-fine strategy greatly improve the registration success rate and also increases robustness by eliminating local optima at coarser scales.

The Insight Toolkit offers a multi-resolution registration framework that is directly compatible with all the registration framework components. The multi-resolution registration framework

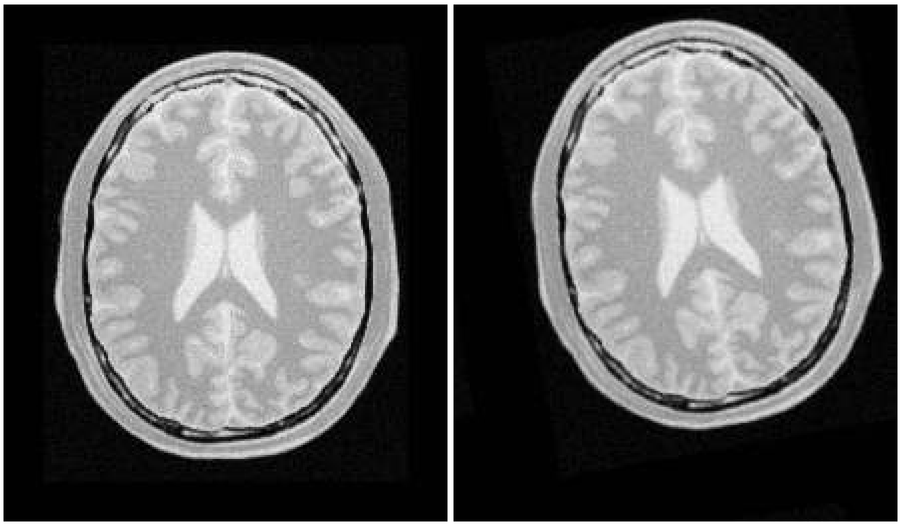


Figure 8.33: Fixed and moving images provided as input to the registration method using the AffineTransform.

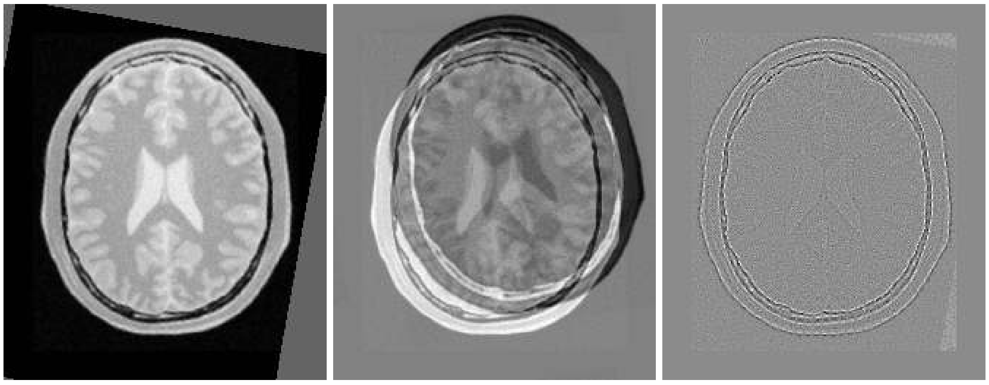


Figure 8.34: The resampled moving image (left), and the difference between the fixed and moving images before (center) and after (right) registration with the AffineTransform transform.

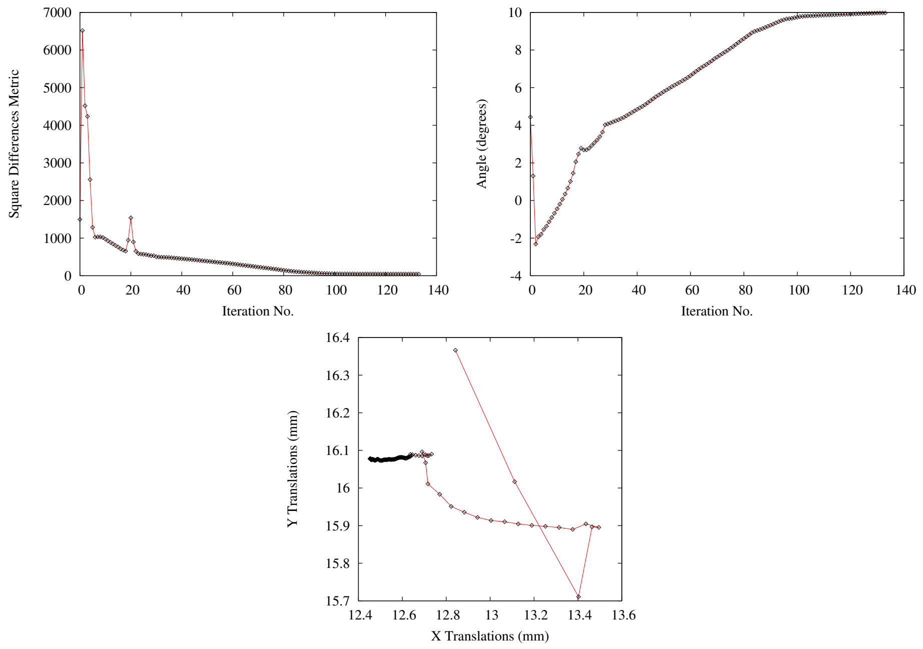


Figure 8.35: Metric values, rotation angle and translations during the registration using the AffineTransform transform.

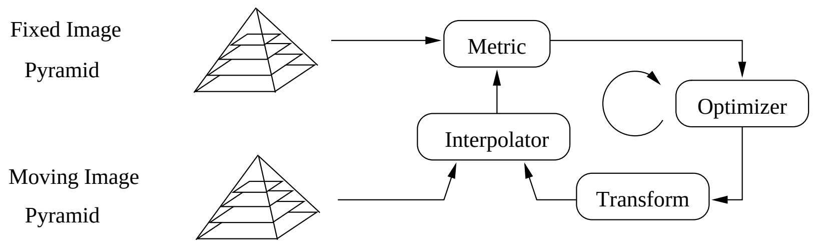


Figure 8.36: Components of the multi-resolution registration framework.

has two additional components: a pair of *image pyramids* that are used to down-sample the fixed and moving images as illustrated in Figure 8.36. The pyramids smooth and subsample the images according to user-defined scheduling of shrink factors.

We now present the main capabilities of the multi-resolution framework by way of an example.

8.7.1 Fundamentals

The source code for this section can be found in the file
`Examples/Registration/MultiResImageRegistration1.cxx`.

This example illustrates the use of the `itk::MultiResolutionImageRegistrationMethod` to solve a simple multi-modality registration problem. In addition to the two input images, a transform, a metric, an interpolator and an optimizer, the multi-resolution framework also requires two image pyramids for creating the sequence of downsampled images. To begin the example, we include the headers of the registration components we will use.

```
#include "itkMultiResolutionImageRegistrationMethod.h"
#include "itkTranslationTransform.h"
#include "itkMattesMutualInformationImageToImageMetric.h"
#include "itkLinearInterpolateImageFunction.h"
#include "itkRegularStepGradientDescentOptimizer.h"
#include "itkMultiResolutionPyramidImageFilter.h"
#include "itkImage.h"
```

The MultiResolutionImageRegistrationMethod solves a registration problem in a coarse to fine manner as illustrated in Figure 8.37. The registration is first performed at the coarsest level using the images at the first level of the fixed and moving image pyramids. The transform parameters determined by the registration are then used to initialize the registration at the next finer level using images from the second level of the pyramids. This process is repeated as we work up to the finest level of image resolution.

In a typical registration scenario, a user will tweak component settings or even swap out components between multi-resolution levels. For example, when optimizing at a coarse resolution,

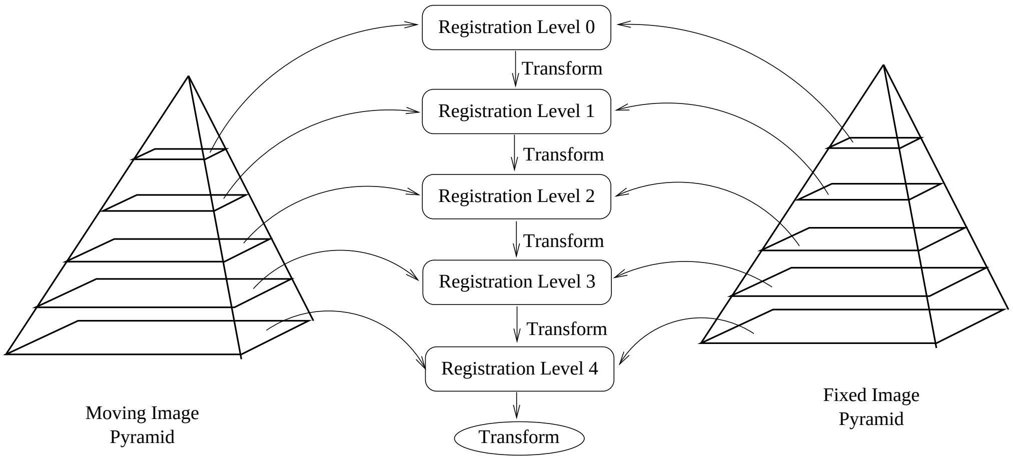


Figure 8.37: Conceptual representation of the multi-resolution registration process.

it may be possible to take more aggressive step sizes and have a more relaxed convergence criterion. Another possible scheme is to use a simple translation transform for the initial coarse registration and upgrade to an affine transform at the finer levels.

Tweaking the components between resolution levels can be done using ITK's implementation of the *Command/Observer* design pattern. Before beginning registration at each resolution level, MultiResolutionImageRegistrationMethod invokes an IterationEvent. The registration components can be changed by implementing a `itk::Command` which responds to the event. A brief description the interaction between events and commands was previously presented in Section 8.4.

We will illustrate this mechanism by changing the parameters of the optimizer between each resolution level by way of a simple interface command. First, we include the header file of the Command class.

```
#include "itkCommand.h"
```

Our new interface command class is called `RegistrationInterfaceCommand`. It derives from Command and is templated over the multi-resolution registration type.

```
template <typename TRegistration>
class RegistrationInterfaceCommand : public itk::Command
{
```

We then define `Self`, `Superclass`, `Pointer`, `New()` and a constructor in a similar fashion to the `CommandIterationUpdate` class in Section 8.4.

```
public:
  typedef  RegistrationInterfaceCommand   Self;
  typedef  itk::Command                   Superclass;
  typedef  itk::SmartPointer<Self>        Pointer;
  itkNewMacro( Self );
protected:
  RegistrationInterfaceCommand() {};
```

For convenience, we declare types useful for converting pointers in the `Execute()` method.

```
public:
  typedef   TRegistration                              RegistrationType;
  typedef   RegistrationType *                         RegistrationPointer;
  typedef   itk::RegularStepGradientDescentOptimizer   OptimizerType;
  typedef   OptimizerType *                            OptimizerPointer;
```

Two arguments are passed to the `Execute()` method: the first is the pointer to the object which invoked the event and the second is the event that was invoked.

```
  void Execute(itk::Object * object, const itk::EventObject & event)
  {
```

First we verify if that the event invoked is of the right type. If not, we return without any further action.

```
    if( !(itk::IterationEvent().CheckEvent( &event )) )
      {
      return;
      }
```

We then convert the input object pointer to a RegistrationPointer. Note that no error checking is done here to verify if the `dynamic_cast` was successful since we know the actual object is a multi-resolution registration method.

```
    RegistrationPointer registration =
                            dynamic_cast<RegistrationPointer>( object );
```

If this is the first resolution level we set the maximum step length (representing the first step size) and the minimum step length (representing the convergence criterion) to large values. At each subsequent resolution level, we will reduce the minimum step length by a factor of 10 in order to allow the optimizer to focus on progressively smaller regions. The maximum step length is set up to the current step length. In this way, when the optimizer is reinitialized at the beginning of the registration process for the next level, the step length will simply start with the last value used for the previous level. This will guarantee the continuity of the path taken by the optimizer through the parameter space.

```
    OptimizerPointer optimizer = dynamic_cast< OptimizerPointer >(
                       registration->GetOptimizer() );

    if ( registration->GetCurrentLevel() == 0 )
      {
      optimizer->SetMaximumStepLength( 16.00 );
      optimizer->SetMinimumStepLength( 2.5 );
      }
    else
      {
      optimizer->SetMaximumStepLength(
                optimizer->GetCurrentStepLength() );
      optimizer->SetMinimumStepLength(
                optimizer->GetMinimumStepLength() / 10.0 );
      }
  }
```

Another version of the `Execute()` method accepting a `const` input object is also required since this method is defined as pure virtual in the base class. This version simply returns without taking any action.

```
  void Execute(const itk::Object * , const itk::EventObject & )
    { return; }
};
```

The fixed and moving image types are defined as in previous examples. Due to the recursive nature of the process by which the downsampled images are computed by the image pyramids, the output images are required to have real pixel types. We declare this internal image type to be `InternalPixelType`:

```
  typedef   float     InternalPixelType;
  typedef itk::Image< InternalPixelType, Dimension > InternalImageType;
```

The types for the registration components are then derived using the internal image type.

```
  typedef itk::TranslationTransform< double, Dimension > TransformType;
  typedef itk::RegularStepGradientDescentOptimizer       OptimizerType;
  typedef itk::LinearInterpolateImageFunction<
                                    InternalImageType,
                                    double             > InterpolatorType;
  typedef itk::MattesMutualInformationImageToImageMetric<
                                    InternalImageType,
                                    InternalImageType >   MetricType;
  typedef itk::MultiResolutionImageRegistrationMethod<
                                    InternalImageType,
                                    InternalImageType >   RegistrationType;
```

In the multi-resolution framework, a itk::MultiResolutionPyramidImageFilter is used to create a pyramid of downsampled images. The size of each downsampled image is specified by the user in the form of a schedule of shrink factors. A description of the filter and the format of the schedules are found in Section 8.12. For this example, we will simply use the default schedules.

```
typedef itk::MultiResolutionPyramidImageFilter<
                                  InternalImageType,
                                  InternalImageType >   FixedImagePyramidType;
typedef itk::MultiResolutionPyramidImageFilter<
                                  InternalImageType,
                                  InternalImageType >   MovingImagePyramidType;
```

The fixed and moving images are read from a file. Before connecting these images to the registration we need to cast them to the internal image type using itk::CastImageFilters.

```
typedef itk::CastImageFilter<
                      FixedImageType, InternalImageType > FixedCastFilterType;
typedef itk::CastImageFilter<
                      MovingImageType, InternalImageType > MovingCastFilterType;

FixedCastFilterType::Pointer fixedCaster   = FixedCastFilterType::New();
MovingCastFilterType::Pointer movingCaster = MovingCastFilterType::New();
```

The output of the readers is connected as input to the cast filters. The inputs to the registration method are taken from the cast filters.

```
fixedCaster->SetInput(  fixedImageReader->GetOutput() );
movingCaster->SetInput( movingImageReader->GetOutput() );

registration->SetFixedImage(    fixedCaster->GetOutput()    );
registration->SetMovingImage(   movingCaster->GetOutput()   );
```

Given that the Mattes Mutual Information metric uses a random iterator in order to collect the samples from the images, it is usually convenient to initialize the seed of the random number generator.

```
metric->ReinitializeSeed( 76926294 );

optimizer->SetNumberOfIterations( 200 );


// Create the Command observer and register it with the optimizer.
//
CommandIterationUpdate::Pointer observer = CommandIterationUpdate::New();
```

```
  optimizer->AddObserver( itk::IterationEvent(), observer );



    Once all the registration components are in place we can create
    an instance of our interface command and connect it to the
    registration object using the \code{AddObserver()} method.

\small
\begin{verbatim}
  typedef RegistrationInterfaceCommand<RegistrationType> CommandType;
  CommandType::Pointer command = CommandType::New();
  registration->AddObserver( itk::IterationEvent(), command );
```

We set the number of multi-resolution levels to three and trigger the registration process by calling `StartRegistration()`.

```
  registration->SetNumberOfLevels( 3 );

  try
    {
    registration->StartRegistration();
    }
  catch( itk::ExceptionObject & err )
    {
    std::cout << "ExceptionObject caught !" << std::endl;
    std::cout << err << std::endl;
    return -1;
    }
```

Let's execute this example using the same multi-modality images as before. The registration converged at the first level after 6 iterations with translation parameters of (13.8663, 18.9939). The second level converged after 5 iterations with result of (13.1035, 17.19). Registration converged after 1 iteration at the last level with the final result being:

```
    Translation X = 13.1035
    Translation Y = 17.19
```

These values are a close match to the true misalignment of $(13, 17)$ introduced in the moving image.

The result of resampling the moving image is presented in the left image of Figure 8.38. The center and right images of the figure depict a checkerboard composite of the fixed and moving images before and after registration.

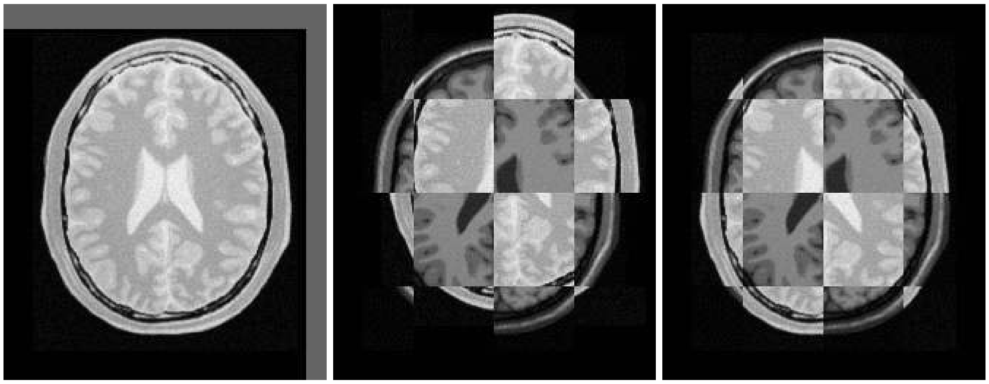


Figure 8.38: Mapped moving image (left) and composition of fixed and moving images before (center) and after (right) registration.

Figure 8.39 (left) shows the sequence of translations followed by the optimizer as it searched the parameter space. The right side of the same figure shows the sequence of metric values computed as the optimizer searched the parameter space. From the trace, we can see that with the more aggressive optimization parameters we get quite close to the optimal value within 4 iterations with the remaining iterations just doing fine adjustments. It is interesting to compare these results with the ones of the single resolution example in Section 8.5.2, where 24 iterations were required as more conservative optimization parameters had to be used.

8.7.2 Parameter Tuning

The source code for this section can be found in the file
`Examples/Registration/MultiResImageRegistration2.cxx`.

This example illustrates the use of more complex components of the registration framework. In particular, it introduces the use of the `itk::AffineTransform` and the importance of fine-tuning the scale parameters of the optimizer.

The AffineTransform is a linear transformation that maps lines into lines. It can be used to represent translations, rotations, anisotropic scaling, shearing or any combination of them. Details about the affine transform can be seen in Section 8.8.16.

In order to use the AffineTransform class, the following header must be included.

```
#include "itkAffineTransform.h"
```

The configuration of the registration method in this example closely follows the procedure in the previous section. The main changes involve the construction and initialization of the transform. The instantiation of the transform type requires only the dimension of the space and the type used for representing space coordinates.

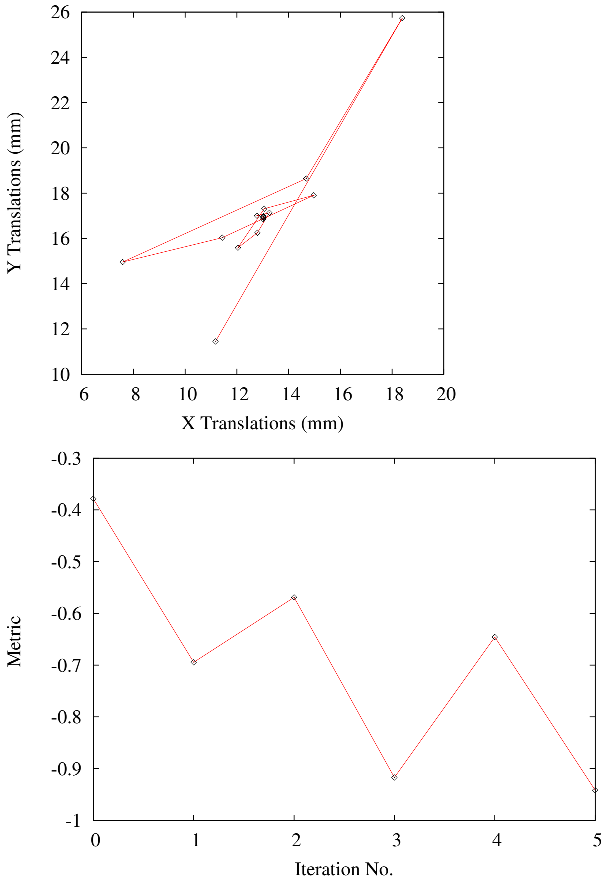


Figure 8.39: Sequence of translations and metric values at each iteration of the optimizer.

```
typedef itk::AffineTransform< double, Dimension > TransformType;
```

The transform is constructed using the standard `New()` method and assigning it to a SmartPointer.

```
TransformType::Pointer   transform  = TransformType::New();
registration->SetTransform( transform );
```

One of the easiest ways of preparing a consistent set of parameters for the transform is to use the transform itself. We can simplify the task of initialization by taking advantage of the additional convenience methods that most transforms have. In this case, we simply force the transform to be initialized as an identity transform. The method `SetIdentity()` is used to that end. Once the transform is initialized, we can invoke its `GetParameters()` method to extract the array of parameters. Finally the array is passed to the registration method using its `SetInitialTransformParameters()` method.

```
transform->SetIdentity();
registration->SetInitialTransformParameters( transform->GetParameters() );
```

The set of parameters in the AffineTransform have different dynamic ranges. Typically the parameters associated with the matrix have values around $[-1:1]$, although they are not restricted to this interval. Parameters associated with translations, on the other hand, tend to have much higher values, typically in the order of 10.0 to 100.0. This difference in dynamic range negatively affects the performance of gradient descent optimizers. ITK provides a mechanism to compensate for such differences in values among the parameters when they are passed to the optimizer. The mechanism consists of providing an array of scale factors to the optimizer. These factors re-normalize the gradient components before they are used to compute the step of the optimizer at the current iteration. In our particular case, a common choice for the scale parameters is to set to 1.0 all those associated with the matrix coefficients, that is, the first $N \times N$ factors. Then, we set the remaining scale factors to a small value. The following code sets up the scale coefficients.

```
OptimizerScalesType optimizerScales( transform->GetNumberOfParameters() );

optimizerScales[0] = 1.0; // scale for M11
optimizerScales[1] = 1.0; // scale for M12
optimizerScales[2] = 1.0; // scale for M21
optimizerScales[3] = 1.0; // scale for M22

optimizerScales[4] = 1.0 / 1000000.0; // scale for translation on X
optimizerScales[5] = 1.0 / 1000000.0; // scale for translation on Y
```

Here the affine transform is represented by the matrix **M** and the vector **T**. The transformation of a point **P** into **P′** is expressed as

$$\begin{bmatrix} P'_x \\ P'_y \end{bmatrix} = \begin{bmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \end{bmatrix} \tag{8.3}$$

The array of scales is then passed to the optimizer using the `SetScales()` method.

```
  optimizer->SetScales( optimizerScales );
```

Given that the Mattes Mutual Information metric uses a random iterator in order to collect the samples from the images, it is usually convenient to initialize the seed of the random number generator.

```
  metric->ReinitializeSeed( 76926294 );
```

```
    The step length has to be proportional to the expected values of the
    parameters in the search space. Since the expected values of the matrix
    coefficients are around $1.0$, the initial step of the optimization
    should be a small number compared to $1.0$. As a guideline, it is
    useful to think of the matrix coefficients as combinations of
    $cos(\theta)$ and $sin(\theta)$.  This leads to use values close to the
    expected rotation measured in radians. For example, a rotation of $1.0$
    degree is about $0.017$ radians. As in the previous example, the
    maximum and minimum step length of the optimizer are set by the
    \code{RegistrationInterfaceCommand} when it is called at the beginning
    of registration at each multi-resolution level.


    Let’s execute this example using the same multi-modality images as
    before.  The registration converges after $5$ iterations in the first
    level, $7$ in the second level and $4$ in the third level. The final
    results when printed as an array of parameters are

    \begin{verbatim}
   [1.00164, 0.00147688, 0.00168372, 1.0027, 12.6296, 16.4768]
```

By reordering them as coefficient of matrix **M** and vector **T** they can now be seen as

$$M = \begin{bmatrix} 1.00164 & 0.0014 \\ 0.00168 & 1.0027 \end{bmatrix} \text{ and } T = \begin{bmatrix} 12.6296 \\ 16.4768 \end{bmatrix} \tag{8.4}$$

In this form, it is easier to interpret the effect of the transform. The matrix **M** is responsible for scaling, rotation and shearing while **T** is responsible for translations. It can be seen that the translation values in this case closely match the true misalignment introduced in the moving image.

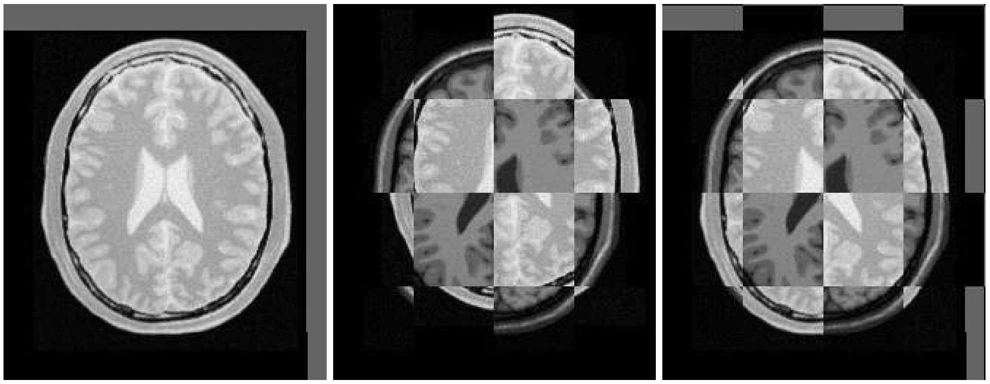


Figure 8.40: Mapped moving image (left) and composition of fixed and moving images before (center) and after (right) multi-resolution registration with the AffineTransform class.

It is important to note that once the images are registered at a sub-pixel level, any further improvement of the registration relies heavily on the quality of the interpolator. It may then be reasonable to use a coarse and fast interpolator in the lower resolution levels and switch to a high-quality but slow interpolator in the final resolution level.

The result of resampling the moving image is shown in the left image of Figure 8.40. The center and right images of the figure present a checkerboard composite of the fixed and moving images before and after registration.

Figure 8.41 (left) presents the sequence of translations followed by the optimizer as it searched the parameter space. The right side of the same figure shows the sequence of metric values computed as the optimizer explored the parameter space.

With the completion of these examples, we will now review the main features of the components forming the registration framework.

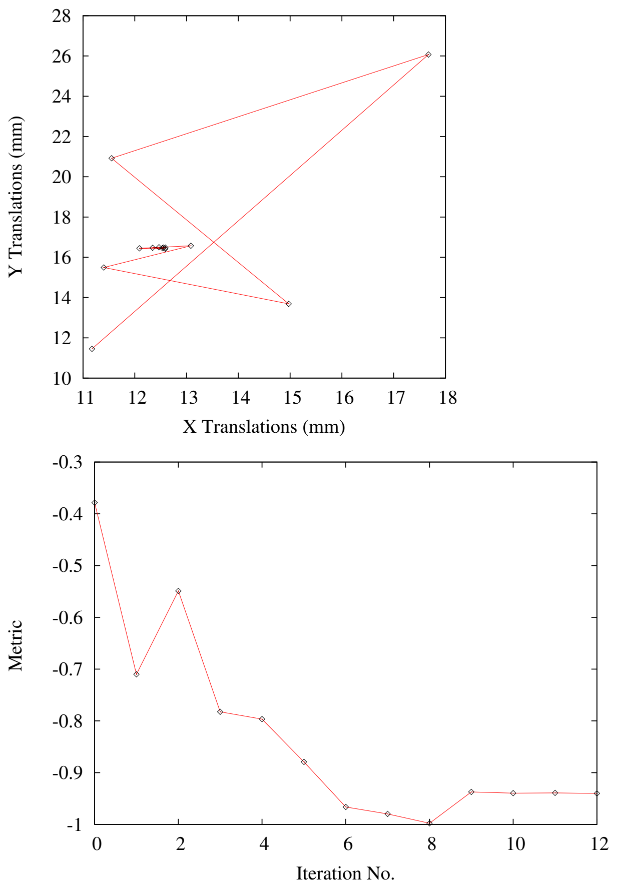


Figure 8.41: Sequence of translations and metric values at each iteration of the optimizer for multi-resolution with the AffineTransform class.

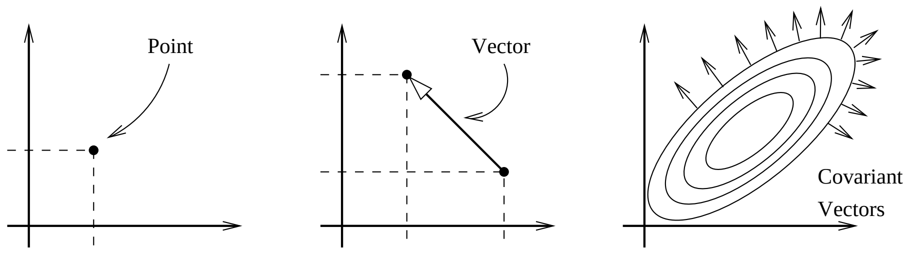


Figure 8.42: Geometric representation objects in ITK.

8.8 Transforms

In the Insight Toolkit, `itk::Transform` objects encapsulate the mapping of points and vectors from an input space to an output space. If a transform is invertible, back transform methods are also provided. Currently, ITK provides a variety of transforms from simple translation, rotation and scaling to general affine and kernel transforms. Note that, while in this section we discuss transforms in the context of registration, transforms are general and can be used for other applications. Some of the most commonly used transforms will be discussed in detail later. Let's begin by introducing the objects used in ITK for representing basic spatial concepts.

8.8.1 Geometrical Representation

ITK implements a consistent geometric representation of the space. The characteristics of classes involved in this representation are summarized in Table 8.1. In this regard, ITK takes full advantage of the capabilities of Object Oriented programming and resists the temptation of using simple arrays of `float` or `double` in order to represent geometrical objects. The use of basic arrays would have blurred the important distinction between the different geometrical concepts and would have allowed for the innumerable conceptual and programming errors that result from using a vector where a point is needed or vice versa.

Additional uses of the `itk::Point`, `itk::Vector` and `itk::CovariantVector` classes have been discussed in Chapter 4. Each one of these classes behaves differently under spatial transformations. It is therefore quite important to keep their distinction clear. Figure 8.42 illustrates the differences between these concepts.

Transform classes provide different methods for mapping each one of the basic space-representation objects. Points, vectors and covariant vectors are transformed using the methods `TransformPoint()`, `TransformVector()` and `TransformCovariantVector()` respectively.

One of the classes that deserve further comments is the `itk::Vector`. This ITK class tend to be misinterpreted as a container of elements instead of a geometrical object. This is a common misconception originated by the fact that Computer Scientist and Software Engineers misuse the term "Vector". The actual word "Vector" is relatively young. It was coined by William Hamilton in his book "*Elements of Quaternions*" published in 1886 (post-mortem)[34]. In the same text Hamilton coined the terms: "*Scalar*", "*Versor*" and "*Tensor*". Although the modern term of "*Tensor*" is used in Calculus in a different sense of what Hamilton defined in his book at the time [23].

Class	**Geometrical concept**
`itk::Point`	Position in space. In N-dimensional space it is represented by an array of N numbers associated with space coordinates.
`itk::Vector`	Relative position between two points. In N-dimensional space it is represented by an array of N numbers, each one associated with the distance along a coordinate axis. Vectors do not have a position in space. A vector is defined as the subtraction of two points.
`itk::CovariantVector`	Orthogonal direction to a $(N-1)$-dimensional manifold in space. For example, in $3D$ it corresponds to the vector orthogonal to a surface. This is the appropriate class for representing Gradients of functions. Covariant vectors do not have a position in space. Covariant vector should not be added to Points, nor to Vectors.

Table 8.1: Summary of objects representing geometrical concepts in ITK.

A "*Vector*" is, by definition, a mathematical object that embodies the concept of "direction in space". Strictly speaking, a Vector describes the relationship between two Points in space, and captures both their relative distance and orientation.

Computer scientists and software engineers misused the term vector in order to represent the concept of an "Indexed Set" [6]. Mechanical Engineers and Civil Engineers, who deal with the real world of physical objects will not commit this mistake and will keep the word "*Vector*" attached to a geometrical concept. Biologists, on the other hand, will associate "*Vector*" to a "vehicle" that allows them to direct something in a particular direction, for example, a virus that allows them to insert pieces of code into a DNA strand [50].

Textbooks in programming do not help to clarify those concepts and loosely use the term "*Vector*" for the purpose of representing an "enumerated set of common elements". STL follows this trend and continue using the word "*Vector*" for what it was not supposed to be used [6, 1]. Linear algebra separates the "*Vector*" from its notion of geometric reality and makes it an abstract set of numbers with arithmetic operations associated.

For those of you who are looking for the "*Vector*" in the Software Engineering sense, please look at the `itk::Array` and `itk::FixedArray` classes that actually provide such functionalities. Additionally, the `itk::VectorContainer` and `itk::MapContainer` classes may be of interest too. These container classes are intended for algorithms that require to insert and delete elements, and that may have large numbers of elements.

The Insight Toolkit deals with real objects that inhabit the physical space. This is particularly true in the context of the image registration framework. We chose to give the appropriate name to the mathematical objects that describe geometrical relationships in N-Dimensional space. It is for this reason that we explicitly make clear the distinction between Point, Vector and CovariantVector, despite the fact that most people would be happy with a simple use of `double[3]` for the three concepts and then will proceed to perform all sort of conceptually flawed operations such as

- Adding two Points
- Dividing a Point by a Scalar
- Adding a Covariant Vector to a Point
- Adding a Covariant Vector to a Vector

In order to enforce the correct use of the Geometrical concepts in ITK we organized these classes in a hierarchy that supports reuse of code and yet compartmentalize the behavior of the individual classes. The use of the `itk::FixedArray` as base class of the `itk::Point`, the `itk::Vector` and the `itk::CovariantVector` was a design decision based on calling things by their correct name.

An `itk::FixedArray` is an enumerated collection with a fixed number of elements. You can instantiate a fixed array of letters, or a fixed array of images, or a fixed array of transforms, or a fixed array of geometrical shapes. Therefore, the FixedArray only implements the functionality that is necessary to access those enumerated elements. No assumptions can be made at this point on any other operations required by the elements of the FixedArray, except the fact of having a default constructor.

The `itk::Point` is a type that represents the spatial coordinates of a spatial location. Based on geometrical concepts we defined the valid operations of the Point class. In particular we made sure that no `operator+()` was defined between Points, and that no `operator*( scalar )` nor `operator/( scalar )` were defined for Points.

In other words, you could do in ITK operations such as:

- Vector = Point - Point
- Point += Vector
- Point -= Vector
- Point = BarycentricCombination(Point, Point)

and you cannot (because you **should not**) do operation such as

- Point = Point * Scalar
- Point = Point + Point
- Point = Point / Scalar

The `itk::Vector` is, by Hamilton's definition, the subtraction between two points. Therefore a Vector must satisfy the following basic operations:

- Vector = Point - Point
- Point = Point + Vector
- Point = Point - Vector
- Vector = Vector + Vector
- Vector = Vector - Vector

An `itk::Vector` object is intended to be instantiated over elements that support mathematical operation such as addition, subtraction and multiplication by scalars.

8.8.2 Transform General Properties

Each transform class typically has several methods for setting its parameters. For example, `itk::Euler2DTransform` provides methods for specifying the offset, angle, and the entire rotation matrix. However, for use in the registration framework, the parameters are represented by a flat Array of doubles to facilitate communication with generic optimizers. In the case of the Euler2DTransform, the transform is also defined by three doubles: the first representing the angle, and the last two the offset. The flat array of parameters is defined using `SetParameters()`. A description of the parameters and their ordering is documented in the sections that follow.

In the context of registration, the transform parameters define the search space for optimizers. That is, the goal of the optimization is to find the set of parameters defining a transform that results in the best possible value of an image metric. The more parameters a transform has, the longer its computational time will be when used in a registration method since the dimension of the search space will be equal to the number of transform parameters.

Another requirement that the registration framework imposes on the transform classes is the computation of their Jacobians. In general, metrics require the knowledge of the Jacobian in order to compute Metric derivatives. The Jacobian is a matrix whose element are the partial derivatives of the output point with respect to the array of parameters that defines the transform:[11]

$$J = \begin{bmatrix} \frac{\partial x_1}{\partial p_1} & \frac{\partial x_1}{\partial p_2} & \cdots & \frac{\partial x_1}{\partial p_m} \\ \frac{\partial x_2}{\partial p_1} & \frac{\partial x_2}{\partial p_2} & \cdots & \frac{\partial x_2}{\partial p_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial x_n}{\partial p_1} & \frac{\partial x_n}{\partial p_2} & \cdots & \frac{\partial x_n}{\partial p_m} \end{bmatrix} \tag{8.5}$$

where $\{p_i\}$ are the transform parameters and $\{x_i\}$ are the coordinates of the output point. Within this framework, the Jacobian is represented by an `itk::Array2D` of doubles and is obtained from the transform by method `GetJacobian()`. The Jacobian can be interpreted as a matrix that indicates for a point in the input space how much its mapping on the output space will change as a response to a small variation in one of the transform parameters. Note that the values of the Jacobian matrix depend on the point in the input space. So actually the Jacobian can be noted as $J(\mathbf{X})$, where $\mathbf{X} = \{x_i\}$. The use of transform Jacobians enables the efficient computation of metric derivatives. When Jacobians are not available, metrics derivatives have to be computed using finite difference at a price of $2M$ evaluations of the metric value, where M is the number of transform parameters.

The following sections describe the main characteristics of the transform classes available in ITK.

8.8.3 Identity Transform

The identity transform `itk::IdentityTransform` is mainly used for debugging purposes. It is provided to methods that require a transform and in cases where we want to have the certainty that the transform will have no effect whatsoever in the outcome of the process. It is just a `NULL` operation. The main characteristics of the identity transform are summarized in Table 8.2

[11] Note that the term *Jacobian* is also commonly used for the matrix representing the derivatives of output point coordinates with respect to input point coordinates. Sometimes the term is loosely used to refer to the determinant of such a matrix. [23]

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Maps every point to itself, every vector to itself and every covariant vector to itself.	0	NA	Only defined when the input and output space has the same number of dimensions.

Table 8.2: Characteristics of the identity transform.

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Represents a simple translation of points in the input space and has no effect on vectors or covariant vectors.	Same as the input space dimension.	The *i*-th parameter represents the translation in the *i*-th dimension.	Only defined when the input and output space has the same number of dimensions.

Table 8.3: Characteristics of the TranslationTransform class.

8.8.4 Translation Transform

The `itk::TranslationTransform` is probably the simplest yet one of the most useful transformations. It maps all Points by adding a Vector to them. Vector and covariant vectors remain unchanged under this transformation since they are not associated with a particular position in space. Translation is the best transform to use when starting a registration method. Before attempting to solve for rotations or scaling it is important to overlap the anatomical objects in both images as much as possible. This is done by resolving the translational misalignment between the images. Translations also have the advantage of being fast to compute and having parameters that are easy to interpret. The main characteristics of the translation transform are presented in Table 8.3.

8.8.5 Scale Transform

The `itk::ScaleTransform` represents a simple scaling of the vector space. Different scaling factors can be applied along each dimension. Points are transformed by multiplying each one of their coordinates by the corresponding scale factor for the dimension. Vectors are transformed in the same way as points. Covariant vectors, on the other hand, are transformed differently since anisotropic scaling does not preserve angles. Covariant vectors are transformed by *dividing* their components by the scale factor of the corresponding dimension. In this way, if a covariant vector was orthogonal to a vector, this orthogonality will be preserved after the transformation. The following equations summarize the effect of the transform

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Points are transformed by multiplying each one of their coordinates by the corresponding scale factor for the dimension. Vectors are transformed as points. Covariant vectors are transformed by *dividing* their components by the scale factor in the corresponding dimension.	Same as the input space dimension.	The *i*-th parameter represents the scaling in the *i*-th dimension.	Only defined when the input and output space has the same number of dimensions.

Table 8.4: Characteristics of the ScaleTransform class.

on the basic geometric objects.

$$\begin{array}{lcccccccc} \text{Point} & \mathbf{P}' & = & T(\mathbf{P}) & : & \mathbf{P}'_\mathbf{i} & = & \mathbf{P_i} \cdot \mathbf{S_i} \\ \text{Vector} & \mathbf{V}' & = & T(\mathbf{V}) & : & \mathbf{V}'_\mathbf{i} & = & \mathbf{V_i} \cdot \mathbf{S_i} \\ \text{CovariantVector} & \mathbf{C}' & = & T(\mathbf{C}) & : & \mathbf{C}'_\mathbf{i} & = & \mathbf{C_i} / \mathbf{S_i} \end{array} \tag{8.6}$$

where $\mathbf{P_i}$, $\mathbf{V_i}$ and $\mathbf{C_i}$ are the point, vector and covariant vector i-th components while $\mathbf{S_i}$ is the scaling factor along dimension $i-th$. The following equation illustrates the effect of the scaling transform on a 3D point.

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} S_1 & 0 & 0 \\ 0 & S_2 & 0 \\ 0 & 0 & S_3 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{8.7}$$

Scaling appears to be a simple transformation but there are actually a number of issues to keep in mind when using different scale factors along every dimension. There are subtle effects—for example, when computing image derivatives. Since derivatives are represented by covariant vectors, their values are not intuitively modified by scaling transforms.

One of the difficulties with managing scaling transforms in a registration process is that typical optimizers manage the parameter space as a vector space where addition is the basic operation. Scaling is better treated in the frame of a logarithmic space where additions result in regular multiplicative increments of the scale. Gradient descent optimizers have trouble updating step length, since the effect of an additive increment on a scale factor diminishes as the factor grows. In other words, a scale factor variation of $(1.0+\varepsilon)$ is quite different from a scale variation of $(5.0+\varepsilon)$.

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Points are transformed by multiplying each one of their coordinates by the corresponding scale factor for the dimension. Vectors are transformed as points. Covariant vectors are transformed by *dividing* their components by the scale factor in the corresponding dimension.	Same as the input space dimension.	The *i*-th parameter represents the scaling in the *i*-th dimension.	Only defined when the input and output space has the same number of dimensions. The difference between this transform and the ScaleTransform is that here the scaling factors are passed as logarithms, in this way their behavior is closer to the one of a Vector space.

Table 8.5: Characteristics of the ScaleLogarithmicTransform class.

Registrations involving scale transforms require careful monitoring of the optimizer parameters in order to keep it progressing at a stable pace. Note that some of the transforms discussed in following sections, for example, the AffineTransform, have hidden scaling parameters and are therefore subject to the same vulnerabilities of the ScaleTransform.

In cases involving misalignments with simultaneous translation, rotation and scaling components it may be desirable to solve for these components independently. The main characteristics of the scale transform are presented in Table 8.4.

8.8.6 Scale Logarithmic Transform

The `itk::ScaleLogarithmicTransform` is a simple variation of the `itk::ScaleTransform`. It is intended to improve the behavior of the scaling parameters when they are modified by optimizers. The difference between this transform and the ScaleTransform is that the parameter factors are passed here as logarithms. In this way, multiplicative variations in the scale become additive variations in the logarithm of the scaling factors.

8.8.7 Euler2DTransform

`itk::Euler2DTransform` implements a rigid transformation in $2D$. It is composed of a plane rotation and a two-dimensional translation. The rotation is applied first, followed by the translation. The following

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Represents a $2D$ rotation and a $2D$ translation. Note that the translation component has no effect on the transformation of vectors and covariant vectors.	3	The first parameter is the angle in radians and the last two parameters are the translation in each dimension.	Only defined for two-dimensional input and output spaces.

Table 8.6: Characteristics of the Euler2DTransform class.

equation illustrates the effect of this transform on a $2D$ point,

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \end{bmatrix} \tag{8.8}$$

where θ is the rotation angle and (T_x, T_y) are the components of the translation.

A challenging aspect of this transformation is the fact that translations and rotations do not form a vector space and cannot be managed as linear independent parameters. Typical optimizers make the loose assumption that parameters exist in a vector space and rely on the step length to be small enough for this assumption to hold approximately.

In addition to the non-linearity of the parameter space, the most common difficulty found when using this transform is the difference in units used for rotations and translations. Rotations are measured in radians; hence, their values are in the range $[-\pi, \pi]$. Translations are measured in millimeters and their actual values vary depending on the image modality being considered. In practice, translations have values on the order of 10 to 100. This scale difference between the rotation and translation parameters is undesirable for gradient descent optimizers because they deviate from the trajectories of descent and make optimization slower and more unstable. In order to compensate for these differences, ITK optimizers accept an array of scale values that are used to normalize the parameter space.

Registrations involving angles and translations should take advantage of the scale normalization functionality in order to obtain the best performance out of the optimizers. The main characteristics of the Euler2DTransform class are presented in Table 8.6.

8.8.8 CenteredRigid2DTransform

`itk::CenteredRigid2DTransform` implements a rigid transformation in $2D$. The main difference between this transform and the `itk::Euler2DTransform` is that here we can specify an arbitrary center of rotation, while the Euler2DTransform always uses the origin of the coordinate system as the center of rotation. This distinction is quite important in image registration since ITK images usually have their origin in the corner of the image rather than the middle. Rotational mis-registrations usually exist, however, as rotations around the center of the image, or at least as rotations around a point in the middle of the

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Represents a $2D$ rotation around a user-provided center followed by a $2D$ translation.	5	The first parameter is the angle in radians. Second and third are the center of rotation coordinates and the last two parameters are the translation in each dimension.	Only defined for two-dimensional input and output spaces.

Table 8.7: Characteristics of the CenteredRigid2DTransform class.

anatomical structure captured by the image. Using gradient descent optimizers, it is almost impossible to solve non-origin rotations using a transform with origin rotations since the deep basin of the real solution is usually located across a high ridge in the topography of the cost function.

In practice, the user must supply the center of rotation in the input space, the angle of rotation and a translation to be applied after the rotation. With these parameters, the transform initializes a rotation matrix and a translation vector that together perform the equivalent of translating the center of rotation to the origin of coordinates, rotating by the specified angle, translating back to the center of rotation and finally translating by the user-specified vector.

As with the Euler2DTransform, this transform suffers from the difference in units used for rotations and translations. Rotations are measured in radians; hence, their values are in the range $[-\pi,\pi]$. The center of rotation and the translations are measured in millimeters, and their actual values vary depending on the image modality being considered. Registrations involving angles and translations should take advantage of the scale normalization functionality of the optimizers in order to get the best performance out of them.

The following equation illustrates the effect of the transform on an input point (x,y) that maps to the output point (x',y'),

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} x - C_x \\ y - C_y \end{bmatrix} + \begin{bmatrix} T_x + C_x \\ T_y + C_y \end{bmatrix} \tag{8.9}$$

where θ is the rotation angle, (C_x,C_y) are the coordinates of the rotation center and (T_x,T_y) are the components of the translation. Note that the center coordinates are subtracted before the rotation and added back after the rotation. The main features of the CenteredRigid2DTransform are presented in Table 8.7.

8.8.9 Similarity2DTransform

The `itk::Similarity2DTransform` can be seen as a rigid transform combined with an isotropic scaling factor. This transform preserves angles between lines. In its $2D$ implementation, the four parameters of this transformation combine the characteristics of the `itk::ScaleTransform` and

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Represents a $2D$ rotation, homogeneous scaling and a $2D$ translation. Note that the translation component has no effect on the transformation of vectors and covariant vectors.	4	The first parameter is the scaling factor for all dimensions, the second is the angle in radians, and the last two parameters are the translations in (x,y) respectively.	Only defined for two-dimensional input and output spaces.

Table 8.8: Characteristics of the Similarity2DTransform class.

itk::Euler2DTransform. In particular, those relating to the non-linearity of the parameter space and the non-uniformity of the measurement units. Gradient descent optimizers should be used with caution on such parameter spaces since the notions of gradient direction and step length are ill-defined.

The following equation illustrates the effect of the transform on an input point (x,y) that maps to the output point (x',y'),

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \lambda & 0 \\ 0 & \lambda \end{bmatrix} \cdot \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} x - C_x \\ y - C_y \end{bmatrix} + \begin{bmatrix} T_x + C_x \\ T_y + C_y \end{bmatrix} \tag{8.10}$$

where λ is the scale factor, θ is the rotation angle, (C_x, C_y) are the coordinates of the rotation center and (T_x, T_y) are the components of the translation. Note that the center coordinates are subtracted before the rotation and scaling, and they are added back afterwards. The main features of the Similarity2DTransform are presented in Table 8.8.

A possible approach for controlling optimization in the parameter space of this transform is to dynamically modify the array of scales passed to the optimizer. The effect produced by the parameter scaling can be used to steer the walk in the parameter space (by giving preference to some of the parameters over others). For example, perform some iterations updating only the rotation angle, then balance the array of scale factors in the optimizer and perform another set of iterations updating only the translations.

8.8.10 QuaternionRigidTransform

The itk::QuaternionRigidTransform class implements a rigid transformation in $3D$ space. The rotational part of the transform is represented using a quaternion while the translation is represented with a vector. Quaternions components do not form a vector space and hence raise the same concerns as the itk::Similarity2DTransform when used with gradient descent optimizers.

The itk::QuaternionRigidTransformGradientDescentOptimizer was introduced into the toolkit to address these concerns. This specialized optimizer implements a variation of a gradient descent algorithm

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Represents a 3D rotation and a 3D translation. The rotation is specified as a quaternion, defined by a set of four numbers **q**. The relationship between quaternion and rotation about vector **n** by angle θ is as follows: $\mathbf{q} = (\mathbf{n}\sin(\theta/\mathbf{2}), \cos(\theta/\mathbf{2}))$ Note that if the quaternion is not of unit length, scaling will also result.	7	The first four parameters defines the quaternion and the last three parameters the translation in each dimension.	Only defined for three-dimensional input and output spaces.

Table 8.9: Characteristics of the QuaternionRigidTransform class.

adapted for a quaternion space. This class insures that after advancing in any direction on the parameter space, the resulting set of transform parameters is mapped back into the permissible set of parameters. In practice, this comes down to normalizing the newly-computed quaternion to make sure that the transformation remains rigid and no scaling is applied. The main characteristics of the QuaternionRigidTransform are presented in Table 8.9.

The Quaternion rigid transform also accepts a user-defined center of rotation. In this way, the transform can easily be used for registering images where the rotation is mostly relative to the center of the image instead one of the corners. The coordinates of this rotation center are not subject to optimization. They only participate in the computation of the mappings for Points and in the computation of the Jacobian. The transformations for Vectors and CovariantVector are not affected by the selection of the rotation center.

8.8.11 VersorTransform

By definition, a *Versor* is the rotational part of a Quaternion. It can also be defined as a *unit-quaternion* [34, 42]. Versors only have three independent components, since they are restricted to reside in the space of unit-quaternions. The implementation of versors in the toolkit uses a set of three numbers. These three numbers correspond to the first three components of a quaternion. The fourth component of the quaternion is computed internally such that the quaternion is of unit length. The main characteristics of the `itk::VersorTransform` are presented in Table 8.10.

This transform exclusively represents rotations in 3D. It is intended to rapidly solve the rotational component of a more general misalignment. The efficiency of this transform comes from using a parameter space of reduced dimensionality. Versors are the best possible representation for rotations in 3D space. Sequences of versors allow the creation of smooth rotational trajectories; for this reason, they behave stably under optimization methods.

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Represents a 3*D* rotation. The rotation is specified by a versor or unit quaternion. The rotation is performed around a user-specified center of rotation.	3	The three parameters define the versor.	Only defined for three-dimensional input and output spaces.

Table 8.10: Characteristics of the Versor Transform

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Represents a 3*D* rotation and a 3*D* translation. The rotation is specified by a versor or unit quaternion, while the translation is represented by a vector. Users can specify the coordinates of the center of rotation.	6	The first three parameters define the versor and the last three parameters the translation in each dimension.	Only defined for three-dimensional input and output spaces.

Table 8.11: Characteristics of the VersorRigid3DTransform class.

The space formed by versor parameters is not a vector space. Standard gradient descent algorithms are not appropriate for exploring this parameter space. An optimizer specialized for the versor space is available in the toolkit under the name of `itk::VersorTransformOptimizer`. This optimizer implements versor derivatives as originally defined by Hamilton [34].

The center of rotation can be specified by the user with the `SetCenter()` method. The center is not part of the parameters to be optimized, therefore it remains the same during an optimization process. Its value is used during the computations for transforming Points and when computing the Jacobian.

8.8.12 VersorRigid3DTransform

The `itk::VersorRigid3DTransform` implements a rigid transformation in 3*D* space. It is a variant of the `itk::QuaternionRigidTransform` and the `itk::VersorTransform`. It can be seen as a `itk::VersorTransform` plus a translation defined by a vector. The advantage of this class with respect

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Represents a rigid rotation in 3*D* space. That is, a rotation followed by a 3*D* translation. The rotation is specified by three angles representing rotations to be applied around the X, Y and Z axis one after another. The translation part is represented by a Vector. Users can also specify the coordinates of the center of rotation.	6	The first three parameters are the rotation angles around X, Y and Z axis, and the last three parameters are the translations along each dimension.	Only defined for three-dimensional input and output spaces.

Table 8.12: Characteristics of the Euler3DTransform class.

to the QuaternionRigidTransform is that it exposes only six parameters, three for the versor components and three for the translational components. This reduces the search space for the optimizer to six dimensions instead of the seven dimensional used by the QuaternionRigidTransform. This transform also allows the users to set a specific center of rotation. The center coordinates are not modified during the optimization performed in a registration process. The main features of this transform are summarized in Table 8.11. This transform is probably the best option to use when dealing with rigid transformations in 3*D*.

Given that the space of Versors is not a Vector space, typical gradient descent optimizers are not well suited for exploring the parametric space of this transform. The `itk::VersorRigid3DTranformOptimizer` has been introduced in the ITK toolkit with the purpose of providing an optimizer that is aware of the Versor space properties on the rotational part of this transform, as well as the Vector space properties on the translational part of the transform.

8.8.13 Euler3DTransform

The `itk::Euler3DTransform` implements a rigid transformation in 3*D* space. It can be seen as a rotation followed by a translation. This class exposes six parameters, three for the Euler angles that represent the rotation and three for the translational components. This transform also allows the users to set a specific center of rotation. The center coordinates are not modified during the optimization performed in a registration process. The main features of this transform are summarized in Table 8.12.

The fact that the three rotational parameters are non-linear and do not behave like Vector spaces must be taken into account when selecting an optimizer to work with this transform and when fine tuning the parameters of such optimizer. It is strongly recommended to use this transform by introducing very small

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Represents a 3*D* rotation, a 3*D* translation and homogeneous scaling. The scaling factor is specified by a scalar, the rotation is specified by a versor, and the translation is represented by a vector. Users can also specify the coordinates of the center of rotation, that is the same center used for scaling.	7	The first parameter is the scaling factor, the next three parameters define the versor and the last three parameters the translation in each dimension.	Only defined for three-dimensional input and output spaces.

Table 8.13: Characteristics of the Similarity3DTransform class.

variations on the rotational components. A small rotation will be in the range of 1 degree, which in radians is approximately 0.0.1745.

You should not expect this transform to be able to compensate for large rotations just by being driven with the optimizer. In practice you must provide a reasonable initialization of the transform angles and only need to correct for residual rotations in the order of 10 or 20 degrees.

8.8.14 Similarity3DTransform

The `itk::Similarity3DTransform` implements a similarity transformation in 3*D* space. It can be seen as an homogeneous scaling followed by a `itk::VersorRigid3DTransform`. This class exposes seven parameters, one for the scaling factor, three for the versor components and three for the translational components. This transform also allows the users to set a specific center of rotation. The center coordinates are not modified during the optimization performed in a registration process. Both the rotation and scaling operations are performed with respect to the center of rotation. The main features of this transform are summarized in Table 8.13.

The fact that the scaling and rotational spaces are non-linear and do not behave like Vector spaces must be taken into account when selecting an optimizer to work with this transform and when fine tuning the parameters of such optimizer.

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Represents a rigid 3*D* transformation followed by a perspective projection. The rotation is specified by a Versor, while the translation is represented by a Vector. Users can specify the coordinates of the center of rotation. They must specifically a focal distance to be used for the perspective projection. The rotation center and the focal distance parameters are not modified during the optimization process.	6	The first three parameters define the Versor and the last three parameters the Translation in each dimension.	Only defined for three-dimensional input and two-dimensional output spaces. This is one of the few transforms where the input space has a different dimension from the output space.

Table 8.14: Characteristics of the Rigid3DPerspectiveTransform class.

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Represents an affine transform composed of rotation, scaling, shearing and translation. The transform is specified by a $N \times N$ matrix and a $N \times 1$ vector where N is the space dimension.	$(N+1) \times N$	The first $N \times N$ parameters define the matrix in column-major order (where the column index varies the fastest). The last N parameters define the translations for each dimension.	Only defined when the input and output space have the same dimension.

Table 8.15: Characteristics of the AffineTransform class.

8.8.15 Rigid3DPerspectiveTransform

The `itk::Rigid3DPerspectiveTransform` implements a rigid transformation in $3D$ space followed by a perspective projection. This transform is intended to be used in $3D/2D$ registration problems where a 3D object is projected onto a 2D plane. This is the case of Fluoroscopic images used for image guided intervention, and it is also the case for classical radiography. Users must provide a value for the focal distance to be used during the computation of the perspective transform. This transform also allows users to set a specific center of rotation. The center coordinates are not modified during the optimization performed in a registration process. The main features of this transform are summarized in Table 8.14. This transform is also used when creating Digitally Reconstructed Radiographs (DRRs).

The strategies for optimizing the parameters of this transform are the same ones used for optimizing the VersorRigid3DTransform. In particular, you can use the same VersorRigid3DTranformOptimizer in order to optimize the parameters of this class.

8.8.16 AffineTransform

The `itk::AffineTransform` is one of the most popular transformations used for image registration. Its main advantage comes from the fact that it is represented as a linear transformation. The main features of this transform are presented in Table 8.15.

The set of AffineTransform coefficients can actually be represented in a vector space of dimension $(N+1) \times N$. This makes it possible for optimizers to be used appropriately on this search space. However, the high dimensionality of the search space also implies a high computational complexity of cost-function derivatives. The best compromise in the reduction of this computational time is to use the transform's Jacobian in combination with the image gradient for computing the cost-function derivatives.

The coefficients of the $N \times N$ matrix can represent rotations, anisotropic scaling and shearing. These coefficients are usually of a very different dynamic range compared to the translation coefficients. Coefficients

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Represents a free from deformation by providing a deformation field from the interpolation of deformations in a coarse grid.	$M \times N$	Where M is the number of nodes in the BSpline grid and N is the dimension of the space.	Only defined when the input and output space have the same dimension. This transform has the advantage of allowing to compute deformable registration. It also has the disadvantage of having a very high dimensional parametric space, and therefore requiring long computation times.

Table 8.16: Characteristics of the BSplineDeformableTransform class.

in the matrix tend to be in the range $[-1:1]$, but are not restricted to this interval. Translation coefficients, on the other hand, can be on the order of 10 to 100, and are basically related to the image size and pixel spacing.

This difference in scale makes it necessary to take advantage of the functionality offered by the optimizers for rescaling the parameter space. This is particularly relevant for optimizers based on gradient descent approaches. This transform lets the user set an arbitrary center of rotation. The coordinates of the rotation center do not make part of the parameters array passed to the optimizer. Equation 8.11 illustrates the effect of applying the AffineTransform in a point in 3D space.

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} M_{00} & M_{01} & M_{02} \\ M_{10} & M_{11} & M_{12} \\ M_{20} & M_{21} & M_{22} \end{bmatrix} \cdot \begin{bmatrix} x - C_x \\ y - C_y \\ z - C_z \end{bmatrix} + \begin{bmatrix} T_x + C_x \\ T_y + C_y \\ T_z + C_z \end{bmatrix} \tag{8.11}$$

A registration based on the affine transform may be more effective when applied after simpler transformations have been used to remove the major components of misalignment. Otherwise it will incur an overwhelming computational cost. For example, using an affine transform, the first set of optimization iterations would typically focus on removing large translations. This task could instead be accomplished by a translation transform in a parameter space of size N instead of the $(N+1) \times N$ associated with the affine transform.

Tracking the evolution of a registration process that uses AffineTransforms can be challenging, since it is difficult to represent the coefficients in a meaningful way. A simple printout of the transform coefficients generally does not offer a clear picture of the current behavior and trend of the optimization. A better implementation uses the affine transform to deform wire-frame cube which is shown in a 3D visualization display.

8.8.17 BSplineDeformableTransform

The `itk::BSplineDeformableTransform` is designed to be used for solving deformable registration problems. This transform is equivalent to generation a deformation field where a deformation vector is assigned to every point in space. The deformation vectors are computed using BSpline interpolation from the deformation values of points located in a coarse grid, that is usually referred to as the BSpline grid.

The BSplineDeformableTransform is not flexible enough for accounting for large rotations or shearing, or scaling differences. In order to compensate for this limitation, it provides the functionality of being composed with an arbitrary transform. This transform is known as the *Bulk* transform and it is applied to points before they are mapped with the displacement field.

This transform do not provide functionalities for mapping Vectors nor CovariantVectors, only Points can be mapped. The reason is that the variations of a vector under a deformable transform actually depend on the location of the vector in space. In other words, Vector only make sense as the relative position between two points.

The BSplineDeformableTransform has a very large number of parameters and therefore is well suited for the `itk::LBFGSOptimizer` and `itk::LBFGSBOptimizer`. The use of this transform for was proposed in the following papers [70, 55, 56].

8.8.18 KernelTransforms

Kernel Transforms are a set of Transforms that are also suitable for performing deformable registration. These transforms compute on the fly the displacements corresponding to a deformation field. The displacement values corresponding to every point in space are computed by interpolation from the vectors defined by a set of *Source Landmarks* and a set of *Target Landmarks*.

Several variations of these transforms are available in the toolkit. They differ on the type of interpolation kernel that is used when computing the deformation in a particular point of space. Note that these transforms are computationally expensive and that their numerical complexity is proportional to the number of landmarks and the space dimension.

The following is the list of Transforms based on the KernelTransform.

- `itk::ElasticBodySplineKernelTransform`
- `itk::ElasticBodyReciprocalSplineKernelTransform`
- `itk::ThinPlateSplineKernelTransform`
- `itk::ThinPlateR2LogRSplineKernelTransform`
- `itk::VolumeSplineKernelTransform`

Details about the mathematical background of these transform can be found in the paper by Davis *et. al* [20] and the papers by Rohr *et. al* [68, 69].

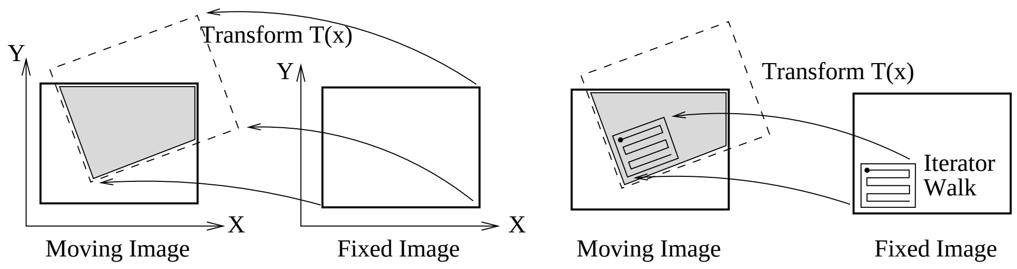


Figure 8.43: The moving image is mapped into the fixed image space under some spatial transformation. An iterator walks through the fixed image and its coordinates are mapped onto the moving image.

8.9 Interpolators

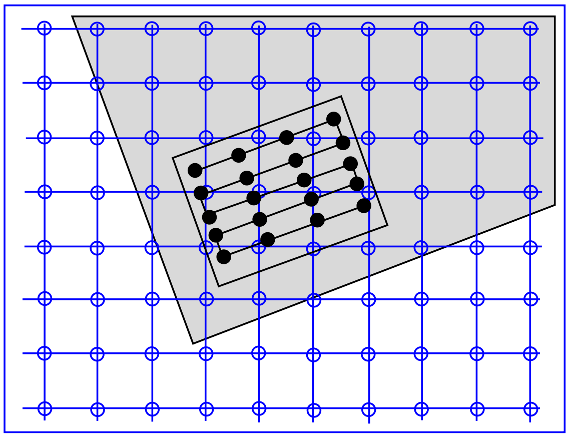


Figure 8.44: Grid positions of the fixed image map to non-grid positions of the moving image.

In the registration process, the metric typically compares intensity values in the fixed image against the corresponding values in the transformed moving image. When a point is mapped from one space to another by a transform, it will in general be mapped to a non-grid position. Therefore, interpolation is required to evaluate the image intensity at the mapped position.

Figure 8.43 (left) illustrates the mapping of the fixed image space onto the moving image space. The transform maps points from the fixed image coordinate system onto the moving image coordinate system. The figure highlights the region of overlap between the two images after the mapping. The right side illustrates how an iterator is used to walk through a region of the fixed image. Each one of the iterator positions is mapped by the transform onto the moving image space in order to find the homologous pixel.

Figure 8.44 presents a detailed view of the mapping from the fixed image to the moving image. In general, the grid positions of the fixed image will not be mapped onto grid positions of the moving image. Interpolation is needed for estimating the intensity of the moving image at these non-grid positions. The service is provided in ITK by interpolator classes that can be plugged into the registration method.

The following interpolators are available:

- itk::NearestNeighborInterpolateImageFunction
- itk::LinearInterpolateImageFunction
- itk::BSplineInterpolateImageFunction

- itk::WindowedSincInterpolateImageFunction

In the context of registration, the interpolation method affects the smoothness of the optimization search space and the overall computation time. On the other hand, interpolations are executed thousands of times in a single optimization cycle. Hence, the user has to balance the simplicity of computation with the smoothness of the optimization when selecting the interpolation scheme.

The basic input to an itk::InterpolateImageFunction is the image to be interpolated. Once an image has been defined using `SetInputImage()`, a user can interpolate either at a point using `Evaluate()` or an index using `EvaluateAtContinuousIndex()`.

Interpolators provide the method `IsInsideBuffer()` that tests whether a particular image index or a physical point falls inside the spatial domain for which image pixels exist.

8.9.1 Nearest Neighbor Interpolation

The itk::NearestNeighborInterpolateImageFunction simply uses the intensity of the nearest grid position. That is, it assumes that the image intensity is piecewise constant with jumps mid-way between grid positions. This interpolation scheme is cheap as it does not require any floating point computations.

8.9.2 Linear Interpolation

The itk::LinearInterpolateImageFunction assumes that intensity varies linearly between grid positions. Unlike nearest neighbor interpolation, the interpolated intensity is spatially continuous. However, the intensity gradient will be discontinuous at grid positions.

8.9.3 B-Spline Interpolation

The itk::BSplineInterpolateImageFunction represents the image intensity using B-spline basis functions. When an input image is first connected to the interpolator, B-spline coefficients are computed using recursive filtering (assuming mirror boundary conditions). Intensity at a non-grid position is computed by multiplying the B-spline coefficients with shifted B-spline kernels within a small support region of the requested position. Figure 8.45 illustrates on the left how the deformation values on the BSpline grid nodes are used for computing interpolated deformations in the rest of space. Note for example that when a cubic BSpline is used, the grid must have one extra node in one side of the image and two extra nodes on the other side, this along every dimension.

Currently, this interpolator supports splines of order 0 to 5. Using a spline of order 0 is almost identical to nearest neighbor interpolation; a spline of order 1 is exactly identical to linear interpolation. For splines of order greater than 1, both the interpolated value and its derivative are spatially continuous.

It is important to note that when using this scheme, the interpolated value may lie outside the range of input image intensities. This is especially important when handling unsigned data, as it is possible that the interpolated value is negative.

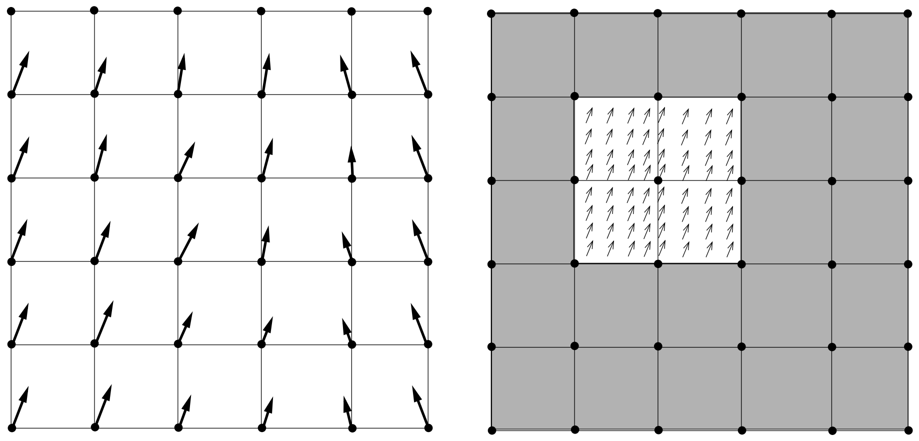


Figure 8.45: The left side illustrates the BSpline grid and the deformations that are known on those nodes. The right side illustrates the region where interpolation is possible when the BSpline is of cubic order. The small arrows represent deformation values that were interpolated from the grid deformations shown on the left side of the diagram.

8.9.4 Windowed Sinc Interpolation

The `itk::WindowedSincInterpolateImageFunction` is the best possible interpolator for data that has been digitized in a discrete grid. This interpolator has been developed based on Fourier Analysis considerations. It is well known in signal processing that the process of sampling a spatial function using a periodic discrete grid results in a replication of the spectrum of that signal in the frequency domain.

The process of recovering the continuous signal from the discrete sampling is equivalent to the removal of the replicated spectra in the frequency domain. This can be done by multiplying the spectra with a box function that will set to zero all the frequencies above the highest frequency in the original signal. Multiplying the spectrum with a box function is equivalent to convolving the spatial discrete signal with a sinc function

$$sinc(x) = \sin(x)/x \tag{8.12}$$

The sinc function has infinite support, which of course in practice can not really be implemented. Therefore, the sinc is usually truncated by multiplying it with a Window function. The Windowed Sinc interpolator is the result of such operation.

This interpolator presents a series of trade-offs in its utilization. Probably the most significant is that the larger the window, the more precise will be the resulting interpolation. However, large windows will also result in long computation times. Since the user can select the window size in this interpolator, it is up to the user to determine how much interpolation quality is required in her/his application and how much computation time can be justified. For details on the signal processing theory behind this interpolator, please refer to Meijering *et. al* [57].

The region of the image used for computing the interpolator is determined by the window *radius*. For example, in a $2D$ image where we want to interpolate the value at position (x,y) the following computation will be performed.

$$I(x,y) = \sum_{i=\lfloor x \rfloor+1-m}^{\lfloor x \rfloor+m} \sum_{j=\lfloor y \rfloor+1-m}^{\lfloor y \rfloor+m} I_{i,j} K(x-i) K(y-j) \tag{8.13}$$

where m is the *radius* of the window. Typically, values such as 3 or 4 are reasonable for the window radius. The function kernel $K(t)$ is composed by the *sinc* function and one of the windows listed above.

$$K(t) = w(t)\mathrm{sinc}(t) = w(t)\frac{\sin(\pi t)}{\pi t} \tag{8.14}$$

Some of the windows that can be used with this interpolator are

Cosinus window

$$w(x) = cos(\frac{\pi x}{2m}) \tag{8.15}$$

Hamming window

$$w(x) = 0.54 + 0.46cos(\frac{\pi x}{m}) \tag{8.16}$$

Welch window

$$w(x) = 1 - (\frac{x^2}{m^2}) \tag{8.17}$$

Lancos window

$$w(x) = \mathrm{sinc}(\frac{x}{m}) \tag{8.18}$$

Blackman window

$$w(x) = 0.42 + 0.5cos(\frac{\pi x}{m}) + 0.08cos(\frac{2\pi x}{m}) \tag{8.19}$$

The window functions listed above are available inside the itk::Function namespace. The conclusions of the referenced paper suggest to use the Welch, Cosine, Kaiser, and Lancos windows for m = 4,5. These are based on error in rotating medical images with respect to the linear interpolation method. In some cases the results achieve a 20-fold improvement in accuracy.

This filter can be used in the same way you would use any ImageInterpolationFunction. For instance, you can plug it into the ResampleImageFilter class. In order to instantiate the filter you must choose several template parameters.

```
 typedef WindowedSincInterpolateImageFunction< TInputImage, VRadius,
TWindowFunction, TBoundaryCondition, TCoordRep > InterpolatorType;
```

`TInputImage` is the image type, as for any other interpolator.

`VRadius` is the radius of the kernel, i.e., the m from the formula above.

`TWindowFunction` is the window function object, which you can choose from about five different functions defined in this header. The default is the Hamming window, which is commonly used but not optimal according to the cited paper.

`TBoundaryCondition` is the boundary condition class used to determine the values of pixels that fall off the image boundary. This class has the same meaning here as in the `itk::NeighborhoodIterator` classes.

`TCoordRep` is again standard for interpolating functions, and should be float or double.

The WindowedSincInterpolateImageFunction is probably not the interpolator that you want to use for performing registration. Its computation burden makes it too expensive for this purpose. The best use of this interpolator is for the final resampling of the image, once that the transform has been found using another less expensive interpolator in the registration process.

8.10 Metrics

In ITK, `itk::ImageToImageMetric` objects quantitatively measure how well the transformed moving image fits the fixed image by comparing the gray-scale intensity of the images. These metrics are very flexible and can work with any transform or interpolation method and do not require reduction of the gray-scale images to sparse extracted information such as edges.

The metric component is perhaps the most critical element of the registration framework. The selection of which metric to use is highly dependent on the registration problem to be solved. For example, some metrics have a large capture range while others require initialization close to the optimal position. In addition, some metrics are only suitable for comparing images obtained from the same imaging modality, while others can handle inter-modality comparisons. Unfortunately, there are no clear-cut rules as to how to choose a metric.

The basic inputs to a metric are: the fixed and moving images, a transform and an interpolator. The method `GetValue()` can be used to evaluate the quantitative criterion at the transform parameters specified in the argument. Typically, the metric samples points within a defined region of the fixed image. For each point, the corresponding moving image position is computed using the transform with the specified parameters, then the interpolator is used to compute the moving image intensity at the mapped position. Details on this mapping are illustrated in Figures 8.43 and 8.44.

The metrics also support region based evaluation. The `SetFixedImageMask()` and `SetMovingImageMask()` methods may be used to restrict evaluation of the metric within a specified region. The masks may be of any type derived from `itk::SpatialObject`.

Besides the measure value, gradient-based optimization schemes also require derivatives of the measure with respect to each transform parameter. The methods `GetDerivatives()` and `GetValueAndDerivatives()` can be used to obtain the gradient information.

The following is the list of metrics currently available in ITK:

- Mean squares
  `itk::MeanSquaresImageToImageMetric`
- Normalized correlation
  `itk::NormalizedCorrelationImageToImageMetric`
- Mean reciprocal squared difference
  `itk::MeanReciprocalSquareDifferenceImageToImageMetric`
- Mutual information by Viola and Wells
  `itk::MutualInformationImageToImageMetric`
- Mutual information by Mattes
  `itk::MattesMutualInformationImageToImageMetric`
- Kullback Liebler distance metric by Kullback and Liebler
  `itk::KullbackLeiblerCompareHistogramImageToImageMetric`
- Normalized mutual information
  `itk::NormalizedMutualInformationHistogramImageToImageMetric`
- Mean squares histogram
  `itk::MeanSquaresHistogramImageToImageMetric`
- Correlation coefficient histogram
  `itk::CorrelationCoefficientHistogramImageToImageMetric`

- Cardinality Match metric
  itk::MatchCardinalityImageToImageMetric
- Kappa Statistics metric
  itk::KappaStatisticImageToImageMetric
- Gradient Difference metric
  itk::GradientDifferenceImageToImageMetric

In the following sections, we describe each metric type in detail. For ease of notation, we will refer to the fixed image $f(\mathbf{X})$ and transformed moving image $(m \circ T(\mathbf{X}))$ as images A and B.

8.10.1 Mean Squares Metric

The itk::MeanSquaresImageToImageMetric computes the mean squared pixel-wise difference in intensity between image A and B over a user defined region:

$$MS(A,B) = \frac{1}{N} \sum_{i=1}^{N} (A_i - B_i)^2 \tag{8.20}$$

A_i is the i-th pixel of Image A
B_i is the i-th pixel of Image B
N is the number of pixels considered

The optimal value of the metric is zero. Poor matches between images A and B result in large values of the metric. This metric is simple to compute and has a relatively large capture radius.

This metric relies on the assumption that intensity representing the same homologous point must be the same in both images. Hence, its use is restricted to images of the same modality. Additionally, any linear changes in the intensity result in a poor match value.

Exploring a Metric

Getting familiar with the characteristics of the Metric as a cost function is fundamental in order to find the best way of setting up an optimization process that will use this metric for solving a registration problem. The following example illustrates a typical mechanism for studying the characteristics of a Metric. Although the example is using the Mean Squares metric, the same methodology can be applied to any of the other metrics available in the toolkit.

The source code for this section can be found in the file
Examples/Registration/MeanSquaresImageMetric1.cxx.

This example illustrates how to explore the domain of an image metric. This is a useful exercise to do before starting a registration process, since getting familiar with the characteristics of the metric is fundamental for the appropriate selection of the optimizer to be use for driving the registration process, as well as for selecting the optimizer parameters. This process makes possible to identify how noisy a metric may be in a given range of parameters, and it will also give an idea of the number of local minima or maxima in which an optimizer may get trapped while exploring the parametric space.

We start by including the headers of the basic components: Metric, Transform and Interpolator.

```
#include "itkMeanSquaresImageToImageMetric.h"
#include "itkTranslationTransform.h"
#include "itkNearestNeighborInterpolateImageFunction.h"
```

We define the dimension and pixel type of the images to be used in the evaluation of the Metric.

```
  const     unsigned int   Dimension = 2;
  typedef   unsigned char  PixelType;

  typedef itk::Image< PixelType, Dimension >   ImageType;
```

The type of the Metric is instantiated and one is constructed. In this case we decided to use the same image type for both the fixed and the moving images.

```
  typedef itk::MeanSquaresImageToImageMetric<
                            ImageType, ImageType >  MetricType;

  MetricType::Pointer metric = MetricType::New();
```

We also instantiate the transform and interpolator types, and create objects of each class.

```
  typedef itk::TranslationTransform< double, Dimension >  TransformType;

  TransformType::Pointer transform = TransformType::New();


  typedef itk::NearestNeighborInterpolateImageFunction<
                                 ImageType, double >  InterpolatorType;

  InterpolatorType::Pointer interpolator = InterpolatorType::New();
```

The classes required by the metric are connected to it. This includes the fixed and moving images, the interpolator and the transform.

```
  metric->SetTransform( transform );
  metric->SetInterpolator( interpolator );

  metric->SetFixedImage(  fixedImage  );
  metric->SetMovingImage( movingImage );
```

Finally we select a region of the parametric space to explore. In this case we are using a translation transform in 2D, so we simply select translations from a negative position to a positive position, in both *x* and *y*. For each one of those positions we invoke the GetValue() method of the Metric.

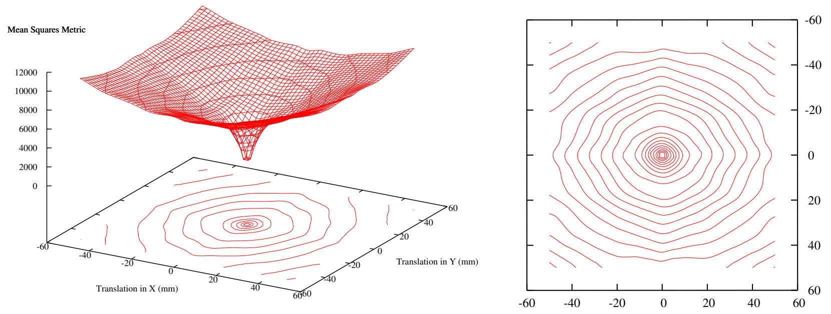


Figure 8.46: Plots of the Mean Squares Metric for an image compared to itself under multiple translations.

```
MetricType::TransformParametersType displacement( Dimension );

const int rangex = 50;
const int rangey = 50;

for( int dx = -rangex; dx <= rangex; dx++ )
  {
  for( int dy = -rangey; dy <= rangey; dy++ )
    {
    displacement[0] = dx;
    displacement[1] = dy;
    const double value = metric->GetValue( displacement );
    std::cout << dx << "   "  << dy << "   " << value << std::endl;
    }
  }
```

Running this code using the image BrainProtonDensitySlice.png as both the fixed and the moving images results in the plot shown in Figure 8.46. From this Figure, it can be seen that a gradient based optimizer will be appropriate for finding the extrema of the Metric. It is also possible to estimate a good value for the step length of a gradient-descent optimizer.

This exercise of plotting the Metric is probably the best thing to do when a registration process is not converging and when it is unclear how to fine tune the different parameters involved in the registration. This includes the optimizer parameters, the metric parameters and even options such as preprocessing the image data with smoothing filters.

The shell and Gnuplot[12] scripts used for generating the graphics in Figure 8.46 are available in the directory

`InsightDocuments/SoftwareGuide/Art`

Of course, this plotting exercise becomes more challenging when the transform has more than

[12]http://www.gnuplot.info

three parameters, and when those parameters have very different range of values. In those cases is necessary to select only a key subset of parameters from the transform and to study the behavior of the metric when those parameters are varied.

8.10.2 Normalized Correlation Metric

The itk::NormalizedCorrelationImageToImageMetric computes pixel-wise cross-correlation and normalizes it by the square root of the autocorrelation of the images:

$$NC(A,B) = -1 \times \frac{\sum_{i=1}^{N}(A_i \cdot B_i)}{\sqrt{\sum_{i=1}^{N} A_i^2 \cdot \sum_{i=1}^{N} B_i^2}} \tag{8.21}$$

A_i is the i-th pixel of Image A
B_i is the i-th pixel of Image B
N is the number of pixels considered

Note the -1 factor in the metric computation. This factor is used to make the metric be optimal when its minimum is reached. The optimal value of the metric is then minus one. Misalignment between the images results in small measure values. The use of this metric is limited to images obtained using the same imaging modality. The metric is insensitive to multiplicative factors between the two images. This metric produces a cost function with sharp peaks and well defined minima. On the other hand, it has a relatively small capture radius.

8.10.3 Mean Reciprocal Square Differences

The itk::MeanReciprocalSquareDifferenceImageToImageMetric computes pixel-wise differences and adds them after passing them through a bell-shaped function $\frac{1}{1+x^2}$:

$$PI(A,B) = \sum_{i=1}^{N} \frac{1}{1 + \frac{(A_i - B_i)^2}{\lambda^2}} \tag{8.22}$$

A_i is the i-th pixel of Image A
B_i is the i-th pixel of Image B
N is the number of pixels considered
λ controls the capture radius

The optimal value is N and poor matches results in small measure values. The characteristics of this metric have been studied by Penney and Holden [36][62]

This image metric has the advantage of producing poor values when few pixels are considered. This makes it consistent when its computation is subject to the size of the overlap region between the images. The capture radius of the metric can be regulated with the parameter λ. The profile of this metric is very peaky. The sharp peaks of the metric help to measure spatial misalignment with high precision. Note that the notion of capture radius is used here in terms of the intensity domain, not the spatial domain. In that regard, λ should be given in intensity units and be associated with the differences in intensity that will make drop the metric by 50%.

The metric is limited to images of the same image modality. The fact that its derivative is large at the central peak is a problem for some optimizers that rely on the derivative to decrease as the extrema are reached. This metric is also sensitive to linear changes in intensity.

8.10.4 Mutual Information Metric

The `itk::MutualInformationImageToImageMetric` computes the mutual information between image A and image B. Mutual information (MI) measures how much information one random variable (image intensity in one image) tells about another random variable (image intensity in the other image). The major advantage of using MI is that the actual form of the dependency does not have to be specified. Therefore, complex mapping between two images can be modeled. This flexibility makes MI well suited as a criterion of multi-modality registration [64].

Mutual information is defined in terms of entropy. Let

$$H(A) = -\int p_A(a) \log p_A(a)\, da \qquad (8.23)$$

be the entropy of random variable A, $H(B)$ the entropy of random variable B and

$$H(A,B) = \int p_{AB}(a,b) \log p_{AB}(a,b)\, da\, db \qquad (8.24)$$

be the joint entropy of A and B. If A and B are independent, then

$$p_{AB}(a,b) = p_A(a) p_B(b) \qquad (8.25)$$

and

$$H(A,B) = H(A) + H(B). \qquad (8.26)$$

However, if there is any dependency, then

$$H(A,B) < H(A) + H(B). \qquad (8.27)$$

The difference is called Mutual Information : $I(A,B)$

$$I(A,B) = H(A) + H(B) - H(A,B) \qquad (8.28)$$

Parzen Windowing

In a typical registration problem, direct access to the marginal and joint probability densities is not available and hence the densities must be estimated from the image data. Parzen windows (also known as kernel density estimators) can be used for this purpose. In this scheme, the densities are constructed by taking intensity samples S from the image and super-positioning kernel functions $K(\cdot)$ centered on the elements of S as illustrated in Figure 8.47:

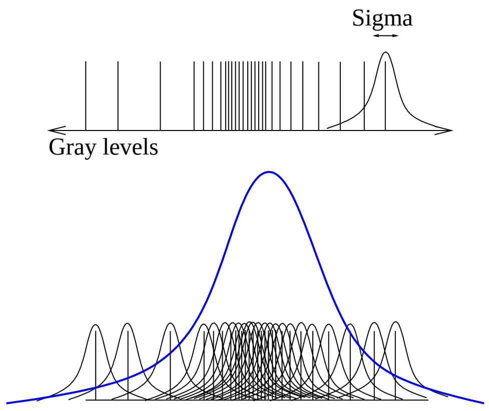


Figure 8.47: In Parzen windowing, a continuous density function is constructed by superimposing kernel functions (Gaussian function in this case) centered on the intensity samples obtained from the image.

A variety of functions can be used as the smoothing kernel with the requirement that they are smooth, symmetric, have zero mean and integrate to one. For example, boxcar, Gaussian and B-spline functions are suitable candidates. A smoothing parameter is used to scale the kernel function. The larger the smoothing parameter, the wider the kernel function used and hence the smoother the density estimate. If the parameter is too large, features such as modes in the density will get smoothed out. On the other hand, if the smoothing parameter is too small, the resulting density may be too noisy. The estimation is given by the following equation.

$$p(a) \approx P^*(a) = \frac{1}{N} \sum_{s_j \in S} K\left(a - s_j\right) \tag{8.29}$$

Choosing the optimal smoothing parameter is a difficult research problem and beyond the scope of this software guide. Typically, the optimal value of the smoothing parameter will depend on the data and the number of samples used.

Viola and Wells Implementation

The Insight Toolkit has multiple implementations of the mutual information metric. One of the most commonly used is `itk::MutualInformationImageToImageMetric` and follows the method specified by Viola and Wells in [85].

In this implementation, two separate intensity samples S and R are drawn from the image: the first to compute the density, and the second to approximate the entropy as a sample mean:

$$H(A) = \frac{1}{N} \sum_{r_j \in R} \log P^*(r_j). \tag{8.30}$$

Gaussian density is used as a smoothing kernel, where the standard deviation σ acts as the smoothing parameter.

The number of spatial samples used for computation is defined using the `SetNumberOfSpatialSamples()` method. Typical values range from 50 to 100. Note that computation involves an $N \times N$ loop and hence, the computation burden becomes very expensive when a large number of samples is used.

The quality of the density estimates depends on the choice of the standard deviation of the Gaussian kernel. The optimal choice will depend on the content of the images. In our experience with the toolkit, we have found that a standard deviation of 0.4 works well for images that have been normalized to have a mean of zero and standard deviation of 1.0. The standard deviation of the fixed image and moving image kernel can be set separately using methods `SetFixedImageStandardDeviation()` and `SetMovingImageStandardDeviation()`.

Mattes et al. Implementation

Another form of mutual information metric available in ITK follows the method specified by Mattes et al. in [55] and is implemented by the `itk::MattesMutualInformationImageToImageMetric` class.

In this implementation, only one set of intensity samples is drawn from the image. Using this set, the marginal and joint probability density function (PDF) is evaluated at discrete positions or bins uniformly spread within the dynamic range of the images. Entropy values are then computed by summing over the bins.

The number of spatial samples used is set using method `SetNumberOfSpatialSamples()`. The number of bins used to compute the entropy values is set via `SetNumberOfHistogramBins()`.

Since the fixed image PDF does not contribute to the metric derivatives, it does not need to be smooth. Hence, a zero order (boxcar) B-spline kernel is used for computing the PDF. On the other hand, to ensure smoothness, a third order B-spline kernel is used to compute the moving image intensity PDF. The advantage of using a B-spline kernel over a Gaussian kernel is that the B-spline kernel has a finite support region. This is computationally attractive, as each intensity sample only affects a small number of bins and hence does not require a $N \times N$ loop to compute the metric value.

During the PDF calculations, the image intensity values are linearly scaled to have a minimum of zero and maximum of one. This rescaling means that a fixed B-spline kernel bandwidth of one can be used to handle image data with arbitrary magnitude and dynamic range.

8.10.5 Kullback-Leibler distance metric

The `itk::KullbackLeiblerCompareHistogramImageToImageMetric` is yet another information based metric. Kullback-Leibler distance measures the relative entropy between two discrete probability distributions. The distributions are obtained from the histograms of the two input images, A and B.

The Kullback-Liebler distance between two histograms is given by

$$KL(A,B) = \sum_{i}^{N} p_A(i) \times \log \frac{p_A(i)}{p_B(i)} \tag{8.31}$$

The distance is always non-negative and is zero only if the two distributions are the same. Note that the distance is not symmetric. In other words, $KL(A,B) \neq KL(B,A)$. Nevertheless, if the distributions are not too dissimilar, the difference between $KL(A,B)$ and $KL(B,A)$ is small.

The implementation in ITK is based on [16].

8.10.6 Normalized Mutual Information Metric

Given two images, A and B, the normalized mutual information may be computed as

$$NMI(A,B) = 1 + \frac{I(A,B)}{H(A,B)} = \frac{H(A)+H(B)}{H(A,B)} \tag{8.32}$$

where the entropy of the images, $H(A)$, $H(B)$, the mutual information, $I(A,B)$ and the joint entropy $H(A,B)$ are computed as mentioned in 8.10.4. Details of the implementation may be found in the [33].

8.10.7 Mean Squares Histogram

The `itk::MeanSquaresHistogramImageToImageMetric` is an alternative implementation of the Mean Squares Metric. In this implementation the joint histogram of the fixed and the mapped moving image is built first. The user selects the number of bins to use in this joint histogram. Once the joint histogram is computed, the bins are visited with an iterator. Given that each bin is associated to a pair of intensities of the form: {fixed intensity, moving intensity}, along with the number of pixels pairs in the images that fell in this bin, it is then possible to compute the sum of square distances between the intensities of both images at the quantization levels defined by the joint histogram bins.

This metric can be represented with Equation 8.33

$$MSH = \sum_{f} \sum_{m} H(f,m)(f-m)^2 \tag{8.33}$$

where $H(f,m)$ is the count on the joint histogram bin identified with fixed image intensity f and moving image intensity m.

8.10.8 Correlation Coefficient Histogram

The `itk::CorrelationCoefficientHistogramImageToImageMetric` computes the cross correlation coefficient between the intensities in the fixed image and the intensities on the mapped moving image. This metric is intended to be used in images of the same modality where the relationship between the intensities of the fixed image and the intensities on the moving images is given by a linear equation.

The correlation coefficient is computed from the Joint histogram as

$$CC = \frac{\sum_f \sum_m H(f,m)\left(f \cdot m - \overline{f} \cdot \overline{m}\right)}{\sum_f H(f)\left((f-\overline{f})^2\right) \cdot \sum_m H(m)\left((m-\overline{m})^2\right)} \tag{8.34}$$

Where $H(f,m)$ is the joint histogram count for the bin identified with the fixed image intensity f and the moving image intensity m. The values $\overline{f}$ and $\overline{m}$ are the mean values of the fixed and moving images respectively. $H(f)$ and $H(m)$ are the histogram counts of the fixed and moving images respectively. The optimal value of the correlation coefficient is 1, which would indicate a perfect straight line in the histogram.

8.10.9 Cardinality Match Metric

The `itk::MatchCardinalityImageToImageMetric` computes cardinality of the set of pixels that match exactly between the moving and fixed images. In other words, it computes the number of pixel matches and mismatches between the two images. The match is designed for label maps. All pixel mismatches are considered equal whether they are between label 1 and label 2 or between label 1 and label 500. In other words, the magnitude of an individual label mismatch is not relevant, or the occurrence of a label mismatch is important.

The spatial correspondence between the fixed and moving images is established using a `itk::Transform` using the `SetTransform()` method and an interpolator using `SetInterpolator()`. Given that we are matching pixels with labels, it is advisable to use Nearest Neighbor interpolation.

8.10.10 Kappa Statistics Metric

The `itk::KappaStatisticImageToImageMetric` computes spatial intersection of two binary images. The metric here is designed for matching pixels in two images with the same exact value, which may be set using `SetForegroundValue()`. Given two images A and B, the κ coefficient is computed as

$$\kappa = \frac{|A| \cap |B|}{|A| + |B|} \tag{8.35}$$

where $|A|$ is the number of foreground pixels in image A. This computes the fraction of area in the two images that is common to both the images. In the computation of the metric, only foreground pixels are considered.

8.10.11 Gradient Difference Metric

This `itk::GradientDifferenceImageToImageMetric` metric evaluates the difference in the derivatives of the moving and fixed images. The derivatives are passed through a function $\frac{1}{1+x}$ and then they are added. The purpose of this metric is to focus the registration on the edges of structures in the images. In this way the borders exert larger influence on the result of the registration than do the inside of the homogeneous regions on the image.

8.11 Optimizers

Optimization algorithms are encapsulated as `itk::Optimizer` objects within ITK. Optimizers are generic and can be used for applications other than registration. Within the registration framework, subclasses of `itk::SingleValuedNonLinearOptimizer` are used to optimize the metric criterion with respect to the transform parameters.

The basic input to an optimizer is a cost function object. In the context of registration, `itk::ImageToImageMetric` classes provides this functionality. The initial parameters are set using `SetInitialPosition()` and the optimization algorithm is invoked by `StartOptimization()`. Once the optimization has finished, the final parameters can be obtained using `GetCurrentPosition()`.

Some optimizers also allow rescaling of their individual parameters. This is convenient for normalizing parameters spaces where some parameters have different dynamic ranges. For example, the first parameter of `itk::Euler2DTransform` represents an angle while the last two parameters represent translations. A unit change in angle has a much greater impact on an image than a unit change in translation. This difference in scale appears as long narrow valleys in the search space making the optimization problem more difficult. Rescaling the translation parameters can help to fix this problem. Scales are represented as an `itk::Array` of doubles and set defined using `SetScales()`.

There are two main types of optimizers in ITK. In the first type we find optimizers that are suitable for dealing with cost functions that return a single value. These are indeed the most common type of cost functions, and are known as *Single Valued* functions, therefore the corresponding optimizers are known as *Single Valued* optimizers. The second type of optimizers are those suitable for managing cost functions that return multiple values at each evaluation. These cost functions are common in model-fitting problems and are known as *Multi Valued* or *Multivariate* functions. The corresponding optimizers are therefore called *MultipleValued* optimizers in ITK.

The `itk::SingleValuedNonLinearOptimizer` is the base class for the first type of optimizers while the `itk::MultipleValuedNonLinearOptimizer` is the base class for the second type of optimizers.

The types of single valued optimizer currently available in ITK are:

- **Amoeba**: Nelder-Meade downhill simplex. This optimizer is actually implemented in the `vxl/vnl` numerics toolkit. The ITK class `itk::AmoebaOptimizer` is merely an adaptor class.
- **Conjugate Gradient**: Fletcher-Reeves form of the conjugate gradient with or without preconditioning (`itk::ConjugateGradientOptimizer`). It is also an adaptor to an optimizer in `vnl`.
- **Gradient Descent**: Advances parameters in the direction of the gradient where the step size is governed by a learning rate (`itk::GradientDescentOptimizer`).

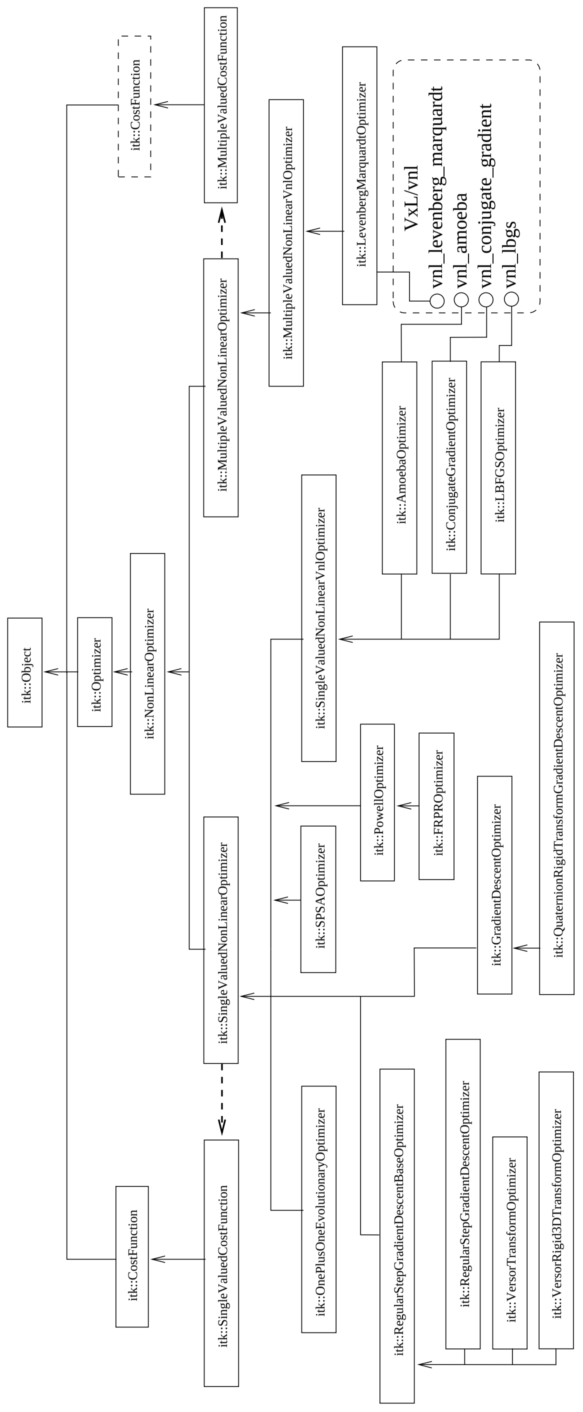

itk::Object
itk::Optimizer
itk::NonLinearOptimizer
itk::CostFunction
itk::SingleValuedCostFunction
itk::SingleValuedNonLinearOptimizer
itk::MultipleValuedNonLinearOptimizer
itk::CostFunction
itk::MultipleValuedCostFunction
itk::MultipleValuedNonLinearVnlOptimizer
itk::LevenbergMarquardtOptimizer
itk::OnePlusOneEvolutionaryOptimizer
itk::SPSAOptimizer
itk::SingleValuedNonLinearVnlOptimizer
itk::PowellOptimizer
itk::FRPROptimizer
itk::AmoebaOptimizer
itk::ConjugateGradientOptimizer
itk::LBFGSOptimizer
itk::RegularStepGradientDescentBaseOptimizer
itk::RegularStepGradientDescentOptimizer
itk::VersorTransformOptimizer
itk::VersorRigid3DTransformOptimizer
itk::GradientDescentOptimizer
itk::QuaternionRigidTransformGradientDescentOptimizer
VxL/vnl
vnl_levenberg_marquardt
vnl_amoeba
vnl_conjugate_gradient
vnl_lbgs

Figure 8.48: Class diagram of the optimizers hierarchy.

- **Quaternion Rigid Transform Gradient Descent**: A specialized version of GradientDescentOptimizer for QuaternionRigidTransform parameters, where the parameters representing the quaternion are normalized to a magnitude of one at each iteration to represent a pure rotation (`itk::QuaternionRigidTransformGradientDescent`).

- **LBFGS**: Limited memory Broyden, Fletcher, Goldfarb and Shannon minimization. It is an adaptor to an optimizer in `vnl` (`itk::LBFGSOptimizer`).

- **LBFGSB**: A modified version of the LBFGS optimizer that allows to specify bounds for the parameters in the search space. It is an adaptor to an optimizer in `netlib`. Details on this optimizer can be found in [12, 13] (`itk::LBFGSBOptimizer`).

- **One Plus One Evolutionary**: Strategy that simulates the biological evolution of a set of samples in the search space. This optimizer is mainly used in the process of bias correction of MRI images (`itk::OnePlusOneEvolutionaryOptimizer.`). Details on this optimizer can be found in [78].

- **Regular Step Gradient Descent**: Advances parameters in the direction of the gradient where a bipartition scheme is used to compute the step size (`itk::RegularStepGradientDescentOptimizer`).

- **Powell Optimizer**: Powell optimization method. For an N-dimensional parameter space, each iteration minimizes(maximizes) the function in N (initially orthogonal) directions. This optimizer is described in [67]. (`itk::PowellOptimizer`).

- **SPSA Optimizer**: Simultaneous Perturbation Stochastic Approximation Method. This optimizer is described in `http://www.jhuapl.edu/SPSA` and in [77]. (`itk::SPSAOptimizer`).

- **Versor Transform Optimizer**: A specialized version of the RegularStepGradientDescentOptimizer for VersorTransform parameters, where the current rotation is composed with the gradient rotation to produce the new rotation versor. It follows the definition of versor gradients defined by Hamilton [34] (`itk::VersorTransformOptimizer`).

- **Versor Rigid3D Transform Optimizer**: A specialized version of the RegularStepGradientDescentOptimizer for VersorRigid3DTransform parameters, where the current rotation is composed with the gradient rotation to produce the new rotation versor. The translational part of the transform parameters are updated as usually done in a vector space. (`itk::VersorRigid3DTransformOptimizer`).

A parallel hierarchy exists for optimizing multiple-valued cost functions. The base optimizer in this branch of the hierarchy is the `itk::MultipleValuedNonLinearOptimizer` whose only current derived class is:

- **Levenberg Marquardt**: Non-linear least squares minimization. Adapted to an optimizer in `vnl` (`itk::LevenbergMarquardtOptimizer`). This optimizer is described in [67].

Figure 8.48 illustrates the full class hierarchy of optimizers in ITK. Optimizers in the lower right corner are adaptor classes to optimizers existing in the vxl/vnl numerics toolkit. The optimizers interact with the itk::CostFunction class. In the registration framework this cost function is reimplemented in the form of ImageToImageMetric.

8.11.1 Registration using Match Cardinality metric

The source code for this section can be found in the file
Examples/Registration/ImageRegistration10.cxx.

This example illustrates the use of the image registration framework in Insight to align two label maps. Common structures are assumed to use the same label. The registration metric simply counts the number of corresponding pixels that have the same label.

```
#include "itkImageRegistrationMethod.h"
#include "itkTranslationTransform.h"
#include "itkMatchCardinalityImageToImageMetric.h"
#include "itkNearestNeighborInterpolateImageFunction.h"
#include "itkAmoebaOptimizer.h"
```

The transform that will map one image space into the other is defined below.

```
  typedef itk::TranslationTransform< double, Dimension > TransformType;
```

An optimizer is required to explore the parameter space of the transform in search of optimal values of the metric. The metric selected does not require analytical derivatives of its cost function.

```
  typedef itk::AmoebaOptimizer       OptimizerType;
```

The metric will compare how well the two images match each other. Metric types are usually parameterized by the image types as can be seen in the following type declaration. The metric selected here is suitable for comparing two label maps where the labels are consistent between the two maps. This metric measures the percentage of pixels that exactly match or mismatch.

```
  typedef itk::MatchCardinalityImageToImageMetric<
                                    FixedImageType,
                                    MovingImageType >    MetricType;
```

Since we are registering label maps, we use a NearestNeighborInterpolateImageFunction to ensure subpixel values are not interpolated (to labels that do not exist).

```
  typedef itk:: NearestNeighborInterpolateImageFunction<
                                    MovingImageType,
                                    double          >    InterpolatorType;
```

```
MetricType::Pointer         metric        = MetricType::New();
TransformType::Pointer      transform     = TransformType::New();
OptimizerType::Pointer      optimizer     = OptimizerType::New();
InterpolatorType::Pointer   interpolator  = InterpolatorType::New();
RegistrationType::Pointer   registration  = RegistrationType::New();
```

We are using a MatchCardinalityImageToImageMetric to compare two label maps. This metric simple counts the percentage of corresponding pixels that have the same label. This metric does not provide analytical derivatives, so we will use an AmoebaOptimizer to drive the registration. The AmoebaOptimizer can only minimize a cost function, so we set the metric to count the percentages of mismatches.

```
metric->MeasureMatchesOff();
```

It is usually desirable to fine tune the parameters of the optimizer. Each optimizer has particular parameters that must be interpreted in the context of the optimization strategy it implements.

The AmoebaOptimizer moves a simplex around the cost surface. Here we set the initial size of the simplex (5 units in each of the parameters)

```
OptimizerType::ParametersType
  simplexDelta( transform->GetNumberOfParameters() );
simplexDelta.Fill( 5.0 );

optimizer->AutomaticInitialSimplexOff();
optimizer->SetInitialSimplexDelta( simplexDelta );
```

We also adjust the tolerances on the optimizer to define convergence. Here, we used a tolerance on the parameters of 0.25 (which will be a quarter of image unit, in this case pixels). We also set the tolerance on the cost function value to define convergence. The metric we are using returns the percentage of pixels that mismatch. So we set the function convergence to be 0.1

```
optimizer->SetParametersConvergenceTolerance( 0.25 ); // quarter pixel
optimizer->SetFunctionConvergenceTolerance(0.001); // 0.1%
```

In the case where the optimizer never succeeds in reaching the desired precision tolerance, it is prudent to establish a limit on the number of iterations to be performed. This maximum number is defined with the method `SetMaximumNumberOfIterations()`.

```
optimizer->SetMaximumNumberOfIterations( 200 );
```

The example was run on two binary images. The first binary image was generated by running the confidence connected image filter (section 9.1.4) on the MRI slice of the brain. The second was generated similarly after shifting the slice by 13 pixels horizontally and 17 pixels vertically. The Amoeba optimizer converged after 34 iterations and produced the following results:

```
Translation X = 12.5
Translation Y = 16.77
```

These results are a close match to the true misalignment.

8.11.2 Registration using the One plus One Evolutionary Optimizer

The source code for this section can be found in the file
`Examples/Registration/ImageRegistration11.cxx`.

This example illustrates how to combine the MutualInformation metric with an Evolutionary algorithm for optimization. Evolutionary algorithms are naturally well-suited for optimizing the Mutual Information metric given its random and noisy behavior.

The structure of the example is almost identical o the one illustrated in ImageRegistration4. Therefore we will focus here on the setup that is specifically required for the evolutionary optimizer.

```
#include "itkImageRegistrationMethod.h"
#include "itkTranslationTransform.h"
#include "itkMattesMutualInformationImageToImageMetric.h"
#include "itkLinearInterpolateImageFunction.h"
#include "itkOnePlusOneEvolutionaryOptimizer.h"
#include "itkNormalVariateGenerator.h"
#include "itkImage.h"
```

In this example the image types and all registration components, except the metric, are declared as in Section 8.2. The Mattes mutual information metric type is instantiated using the image types.

```
  typedef itk::MattesMutualInformationImageToImageMetric<
                                          FixedImageType,
                                          MovingImageType >    MetricType;
```

Evolutionary algorithms are based on testing random variations of parameters. In order to support the computation of random values, ITK provides a family of random number generators. In this example, we use the `itk::NormalVariateGenerator` which generates values with a normal distribution.

```
  typedef itk::Statistics::NormalVariateGenerator  GeneratorType;

  GeneratorType::Pointer generator = GeneratorType::New();
```

The random number generator must be initialized with a seed.

```
  generator->Initialize(12345);
```

Another significant difference in the metric is that it computes the negative mutual information and hence we need to minimize the cost function in this case. In this example we will use the same optimization parameters as in Section 8.2.

```
  optimizer->MaximizeOff();

  optimizer->SetNormalVariateGenerator( generator );
  optimizer->Initialize( 10 );
  optimizer->SetEpsilon( 1.0 );
  optimizer->SetMaximumIteration( 4000 );
```

This example is executed using the same multi-modality images as in the previous one. The registration converges after 24 iterations and produces the following results:

```
    Translation X = 13.1719
    Translation Y = 16.9006
```

These values are a very close match to the true misalignment introduced in the moving image.

8.11.3 Registration using masks constructed with Spatial objects

The source code for this section can be found in the file
`Examples/Registration/ImageRegistration12.cxx`.

This example illustrates the use SpatialObjects as masks for selecting the pixels that should contribute to the computation of Image Metrics. This example is almost identical to ImageRegistration6 with the exception that the SpatialObject masks are created and passed to the image metric.

The most important header in this example is the one corresponding to the `itk::ImageMaskSpatialObject` class.

```
#include "itkImageMaskSpatialObject.h"
```

Here we instantiate the type of the `itk::ImageMaskSpatialObject` using the same dimension of the images to be registered.

```
  typedef itk::ImageMaskSpatialObject< Dimension >   MaskType;
```

Then we use the type for creating the spatial object mask that will restrict the registration to a reduced region of the image.

```
MaskType::Pointer  spatialObjectMask = MaskType::New();
```

The mask in this case is read from a binary file using the `ImageFileReader` instantiated for an `unsigned char` pixel type.

```
typedef itk::Image< unsigned char, Dimension >   ImageMaskType;

typedef itk::ImageFileReader< ImageMaskType >    MaskReaderType;
```

The reader is constructed and a filename is passed to it.

```
MaskReaderType::Pointer  maskReader = MaskReaderType::New();

maskReader->SetFileName( argv[3] );
```

As usual, the reader is triggered by invoking its `Update()` method. Since this may eventually throw an exception, the call must be placed in a `try/catch` block. Note that a full fledged application will place this `try/catch` block at a much higher level, probably under the control of the GUI.

```
try
  {
  maskReader->Update();
  }
catch( itk::ExceptionObject & err )
  {
  std::cerr << "ExceptionObject caught !" << std::endl;
  std::cerr << err << std::endl;
  return -1;
  }
```

The output of the mask reader is connected as input to the `ImageMaskSpatialObject`.

```
spatialObjectMask->SetImage( maskReader->GetOutput() );
```

Finally, the spatial object mask is passed to the image metric.

```
metric->SetFixedImageMask( spatialObjectMask );
```

Let’s execute this example over some of the images provided in `Examples/Data`, for example:

- `BrainProtonDensitySliceBorder20.png`

- BrainProtonDensitySliceR10X13Y17.png

The second image is the result of intentionally rotating the first image by 10 degrees and shifting it $13mm$ in X and $17mm$ in Y. Both images have unit-spacing and are shown in Figure 8.17.

```
  transform->SetParameters( finalParameters );

  TransformType::MatrixType matrix = transform->GetRotationMatrix();
  TransformType::OffsetType offset = transform->GetOffset();

  std::cout << "Matrix = " << std::endl << matrix << std::endl;
  std::cout << "Offset = " << std::endl << offset << std::endl;
```

Now we resample the moving image using the transform resulting from the registration process.

8.11.4 Rigid registrations incorporating prior knowledge

The source code for this section can be found in the file
Examples/Registration/ImageRegistration13.cxx.

This example illustrates how to do registration with a 2D Rigid Transform and with MutualInformation metric.

```
#include "itkMattesMutualInformationImageToImageMetric.h"
```

The CenteredRigid2DTransform applies a rigid transform in 2D space.

```
  typedef itk::CenteredRigid2DTransform< double > TransformType;
  typedef itk::RegularStepGradientDescentOptimizer       OptimizerType;

  typedef itk::MattesMutualInformationImageToImageMetric<
                                          FixedImageType,
                                          MovingImageType >    MetricType;

  TransformType::Pointer      transform     = TransformType::New();
  OptimizerType::Pointer      optimizer     = OptimizerType::New();
```

The `itk::CenteredRigid2DTransform` is initialized by 5 parameters, indicating the angle of rotation, the center coordinates and the translation to be applied after rotation. The initialization is done by the `itk::CenteredTransformInitializer`. The transform can operate in two modes, one assumes that the anatomical objects to be registered are centered in their respective images. Hence the best initial guess for the registration is the one that superimposes those two centers. This second approach assumes that the moments of the anatomical objects are similar for both images and hence the best initial guess for registration is to superimpose both mass centers. The center of mass is computed from the moments obtained from the gray level values. Here we adopt the first approach. The `GeometryOn()` method toggles between the approaches.

```
typedef itk::CenteredTransformInitializer<
                                  TransformType,
                                  FixedImageType,
                                  MovingImageType >  TransformInitializerType;
TransformInitializerType::Pointer initializer = TransformInitializerType::New();

initializer->SetTransform(   transform );

initializer->SetFixedImage(  fixedImageReader->GetOutput() );
initializer->SetMovingImage( movingImageReader->GetOutput() );
initializer->GeometryOn();
initializer->InitializeTransform();
```

The optimizer scales the metrics (the gradient in this case) by the scales during each iteration. Hence a large value of the center scale will prevent movement along the center during optimization. Here we assume that the fixed and moving images are likely to be related by a translation.

```
typedef OptimizerType::ScalesType       OptimizerScalesType;
OptimizerScalesType optimizerScales( transform->GetNumberOfParameters() );

const double translationScale = 1.0 / 128.0;
const double centerScale      = 1000.0; // prevents it from moving
                                        // during the optimization
optimizerScales[0] = 1.0;
optimizerScales[1] = centerScale;
optimizerScales[2] = centerScale;
optimizerScales[3] = translationScale;
optimizerScales[4] = translationScale;

optimizer->SetScales( optimizerScales );

optimizer->SetMaximumStepLength( 0.5   );
optimizer->SetMinimumStepLength( 0.0001 );
optimizer->SetNumberOfIterations( 400 );
```

Let's execute this example over some of the images provided in `Examples/Data`, for example:

- `BrainProtonDensitySlice.png`
- `BrainProtonDensitySliceBorder20.png`

The second image is the result of intentionally shifting the first image by $20mm$ in X and $20mm$ in Y. Both images have unit-spacing and are shown in Figure 8.3. The example yielded the following results.

```
Translation X = 20
Translation Y = 20
```

These values match the true misalignment introduced in the moving image.

8.12 Image Pyramids

In ITK, the `itk::MultiResolutionPyramidImageFilter` can be used to create a sequence of reduced resolution images from the input image. The down-sampling is performed according to a user defined multi-resolution schedule. The schedule is specified as an `itk::Array2D` of integers, containing shrink factors for each multi-resolution level (rows) for each dimension (columns). For example,

```
8 4 4
4 4 2
```

is a schedule for a three dimensional image for two multi-resolution levels. In the first (coarsest) level, the image is reduced by a factor of 8 in the column dimension, factor of 4 in the row dimension and a factor of 4 in the slice dimension. In the second level, the image reduced by a factor of 4 in the column dimension, 4 in the row dimension and 2 in the slice dimension.

The method `SetNumberOfLevels()` is used to set the number of resolution levels in the pyramid. This method will allocate memory for the schedule and generate a default table with the starting (coarsest) shrink factors for all dimensions set to $(M-1)^2$, where M is the number of levels. All factors are halved for all subsequent levels. For example, if we set the number of levels to 4, the default schedule is then:

```
8 8 8
4 4 4
2 2 2
1 1 1
```

The user can get a copy of the schedule using method `GetSchedule()`, make modifications, and reset it using method `SetSchedule()`. Alternatively, a user can create a default table by specifying the starting (coarsest) shrink factors using the method `SetStartingShrinkFactors()`. The factors for the subsequent levels are generated by halving the factor or setting it to one, depending on which is larger. For example, for a 4 level pyramid and starting factors of 8, 8 and 4, the generated schedule would be:

```
8 8 4
4 4 2
2 2 1
1 1 1
```

When this filter is triggered by `Update()`, M outputs are produced where the m-th output corresponds to the m-th level of the pyramid. To generate these images, Gaussian smoothing is first performed using a `itk::DiscreteGaussianImageFilter` with the variance set to $(s/2)^2$, where s is the shrink factor. The smoothed images are then sub-sampled using a `itk::ShrinkImageFilter`.

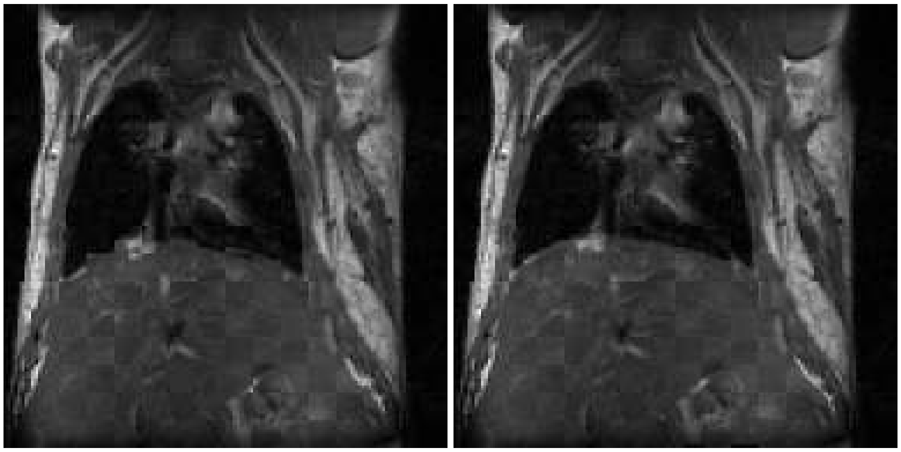


Figure 8.49: Checkerboard comparisons before and after FEM-based deformable registration.

8.13 Deformable Registration

The source code for this section can be found in the file
`Examples/Registration/DeformableRegistration1.cxx`.

The finite element (FEM) library within the Insight Toolkit can be used to solve deformable image registration problems. The first step in implementing a FEM-based registration is to include the appropriate header files.

```
#include "itkFEM.h"
#include "itkFEMRegistrationFilter.h"
```

Next, we use `typedef`s to instantiate all necessary classes. We define the image and element types we plan to use to solve a two-dimensional registration problem. We define multiple element types so that they can be used without recompiling the code.

```
typedef itk::Image<unsigned char, 2>                        fileImageType;
typedef itk::Image<float, 2>                                ImageType;
typedef itk::fem::Element2DC0LinearQuadrilateralMembrane   ElementType;
typedef itk::fem::Element2DC0LinearTriangularMembrane      ElementType2;
```

Note that in order to solve a three-dimensional registration problem, we would simply define 3D image and element types in lieu of those above. The following declarations could be used for a 3D problem:

```
typedef itk::Image<unsigned char, 3>                    fileImage3DType;
typedef itk::Image<float, 3>                               Image3DType;
typedef itk::fem::Element3DC0LinearHexahedronMembrane     Element3DType;
typedef itk::fem::Element3DC0LinearTetrahedronMembrane    Element3DType2;
```

Here, we instantiate the load types and explicitly template the load implementation type. We also define visitors that allow the elements and loads to communicate with one another.

```
typedef itk::fem::FiniteDifferenceFunctionLoad<ImageType,ImageType> ImageLoadType;
template class itk::fem::ImageMetricLoadImplementation<ImageLoadType>;

typedef ElementType::LoadImplementationFunctionPointer     LoadImpFP;
typedef ElementType::LoadType                              ElementLoadType;

typedef ElementType2::LoadImplementationFunctionPointer    LoadImpFP2;
typedef ElementType2::LoadType                             ElementLoadType2;

typedef itk::fem::VisitorDispatcher<ElementType,ElementLoadType, LoadImpFP>
                                                           DispatcherType;

typedef itk::fem::VisitorDispatcher<ElementType2,ElementLoadType2, LoadImpFP2>
                                                           DispatcherType2;
```

Once all the necessary components have been instantiated, we can instantiate the `itk::FEMRegistrationFilter`, which depends on the image input and output types.

```
typedef itk::fem::FEMRegistrationFilter<ImageType,ImageType> RegistrationType;
```

The `itk::fem::ImageMetricLoad` must be registered before it can be used correctly with a particular element type. An example of this is shown below for ElementType. Similar definitions are required for all other defined element types.

```
  ElementType::LoadImplementationFunctionPointer fp =
    &itk::fem::ImageMetricLoadImplementation<ImageLoadType>::ImplementImageMetricLoad;
  DispatcherType::RegisterVisitor((ImageLoadType*)0,fp);
```

In order to begin the registration, we declare an instance of the FEMRegistrationFilter. For simplicity, we will call it `registrationFilter`.

```
  RegistrationType::Pointer registrationFilter = RegistrationType::New();
```

Next, we call `registrationFilter->SetConfigFileName()` to read the parameter file containing information we need to set up the registration filter (image files, image sizes, etc.). A sample parameter file is shown at the end of this section, and the individual components are labeled.

In order to initialize the mesh of elements, we must first create “dummy” material and element objects and assign them to the registration filter. These objects are subsequently used to either read a predefined mesh from a file or generate a mesh using the software. The values assigned to the fields within the material object are arbitrary since they will be replaced with those specified in the parameter file. Similarly, the element object will be replaced with those from the desired mesh.

```
// Create the material properties
itk::fem::MaterialLinearElasticity::Pointer m;
m = itk::fem::MaterialLinearElasticity::New();
m->GN = 0;                  // Global number of the material
m->E = registrationFilter->GetElasticity();  // Young's modulus -- used in the membrane
m->A = 1.0;                 // Cross-sectional area
m->h = 1.0;                 // Thickness
m->I = 1.0;                 // Moment of inertia
m->nu = 0.;                 // Poisson's ratio -- DONT CHOOSE 1.0!!
m->RhoC = 1.0;              // Density

// Create the element type
ElementType::Pointer e1=ElementType::New();
e1->m_mat=dynamic_cast<itk::fem::MaterialLinearElasticity*>( m );
registrationFilter->SetElement(e1);
registrationFilter->SetMaterial(m);
```

Now we are ready to run the registration:

```
registrationFilter->RunRegistration();
```

To output the image resulting from the registration, we can call `WriteWarpedImage()`. The image is written in floating point format.

```
registrationFilter->WriteWarpedImage(
      (registrationFilter->GetResultsFileName()).c_str());
```

We can also output the displacement fields resulting from the registration, we can call `WriteDisplacementField()` with the desired vector component as an argument. For a $2D$ registration, you would want to write out both the x and y displacements, and this requires two calls to the aforementioned function.

```
if (registrationFilter->GetWriteDisplacements())
  {
  registrationFilter->WriteDisplacementField(0);
```

```
    registrationFilter->WriteDisplacementField(1);
    // If this were a 3D example, you might also want to call this line:
    // registrationFilter->WriteDisplacementField(2);

    // We can also write it as a multicomponent vector field
    registrationFilter->WriteDisplacementFieldMultiComponent();
    }
```

Figure 8.49 presents the results of the FEM-based deformable registration applied to two time-separated slices of a living rat dataset. Checkerboard comparisons of the two images are shown before registration (left) and after registration (right). Both images were acquired from the same living rat, the first after inspiration of air into the lungs and the second after exhalation. Deformation occurs due to the relaxation of the diaphragm and the intercostal muscles, both of which exert force on the lung tissue and cause air to be expelled.

The following is a documented sample parameter file that can be used with this deformable registration example. This example demonstrates the setup of a basic registration problem that does not use multi-resolution strategies. As a result, only one value for the parameters between `(# of pixels per element)` and `(maximum iterations)` is necessary. In order to use a multi-resolution strategy, you would have to specify values for those parameters at each level of the pyramid.

The source code for this section can be found in the file
`Examples/Registration/DeformableRegistration4.cxx`.

This example illustrates the use of the `itk::BSplineDeformableTransform` class for performing registration of two 2*D* images. The example code is for the most part identical to the code presented in Section 8.6.1. The major difference is that this example we replace the Transform for a more generic one endowed with a large number of degrees of freedom. Due to the large number of parameters, we will also replace the simple steepest descent optimizer with the `itk::LBFGSOptimizer`.

The following are the most relevant headers to this example.

```
#include "itkBSplineDeformableTransform.h"
#include "itkLBFGSOptimizer.h"
```

The parameter space of the `BSplineDeformableTransform` is composed by the set of all the deformations associated with the nodes of the BSpline grid. This large number of parameters makes possible to represent a wide variety of deformations, but it also has the price of requiring a significant amount of computation time.

We instantiate now the type of the `BSplineDeformableTransform` using as template parameters the type for coordinates representation, the dimension of the space, and the order of the BSpline.

```
  const unsigned int SpaceDimension = ImageDimension;
```

```
const unsigned int SplineOrder = 3;
typedef double CoordinateRepType;

typedef itk::BSplineDeformableTransform<
                          CoordinateRepType,
                          SpaceDimension,
                          SplineOrder >     TransformType;
```

The transform object is constructed below and passed to the registration method.

```
TransformType::Pointer  transform = TransformType::New();
registration->SetTransform( transform );
```

Here we define the parameters of the BSplineDeformableTransform grid. We arbitrarily decide to use a grid with 5×5 nodes within the image. The reader should note that the BSpline computation requires a finite support region (1 grid node at the lower borders and 2 grid nodes at upper borders). Therefore in this example, we set the grid size to be 8×8 and place the grid origin such that grid node (1,1) coincides with the first pixel in the fixed image.

```
typedef TransformType::RegionType RegionType;
RegionType bsplineRegion;
RegionType::SizeType   gridSizeOnImage;
RegionType::SizeType   gridBorderSize;
RegionType::SizeType   totalGridSize;

gridSizeOnImage.Fill( 5 );
gridBorderSize.Fill( 3 );    // Border for spline order = 3 ( 1 lower, 2 upper )
totalGridSize = gridSizeOnImage + gridBorderSize;

bsplineRegion.SetSize( totalGridSize );

typedef TransformType::SpacingType SpacingType;
SpacingType spacing = fixedImage->GetSpacing();

typedef TransformType::OriginType OriginType;
OriginType origin = fixedImage->GetOrigin();;

FixedImageType::SizeType fixedImageSize = fixedRegion.GetSize();

for(unsigned int r=0; r<ImageDimension; r++)
  {
  spacing[r] *= floor( static_cast<double>(fixedImageSize[r] - 1)  /
                 static_cast<double>(gridSizeOnImage[r] - 1) );
  origin[r]  -=  spacing[r];
  }
transform->SetGridSpacing( spacing );
transform->SetGridOrigin( origin );
transform->SetGridRegion( bsplineRegion );


typedef TransformType::ParametersType     ParametersType;

const unsigned int numberOfParameters =
             transform->GetNumberOfParameters();

ParametersType parameters( numberOfParameters );

parameters.Fill( 0.0 );

transform->SetParameters( parameters );
```

We now pass the parameters of the current transform as the initial parameters to be used when the registration process starts.

```
  registration->SetInitialTransformParameters( transform->GetParameters() );
```

Next we set the parameters of the LBFGS Optimizer.

```
  optimizer->SetGradientConvergenceTolerance( 0.05 );
  optimizer->SetLineSearchAccuracy( 0.9 );
  optimizer->SetDefaultStepLength( 1.5 );
  optimizer->TraceOn();
  optimizer->SetMaximumNumberOfFunctionEvaluations( 1000 );
```

Let's execute this example using the rat lung images from the previous examples.

- `RatLungSlice1.mha`
- `RatLungSlice2.mha`

```
  transform->SetParameters( finalParameters );
```

The source code for this section can be found in the file
`Examples/Registration/DeformableRegistration5.cxx`.

This example demonstrates how to use the level set motion to deformably register two images. The first step is to include the header files.

```
#include "itkLevelSetMotionRegistrationFilter.h"
#include "itkHistogramMatchingImageFilter.h"
```

```
#include "itkCastImageFilter.h"
#include "itkWarpImageFilter.h"
#include "itkLinearInterpolateImageFunction.h"
```

Second, we declare the types of the images.

```
  const unsigned int Dimension = 2;
  typedef unsigned short PixelType;

  typedef itk::Image< PixelType, Dimension >  FixedImageType;
  typedef itk::Image< PixelType, Dimension >  MovingImageType;
```

Image file readers are set up in a similar fashion to previous examples. To support the re-mapping of the moving image intensity, we declare an internal image type with a floating point pixel type and cast the input images to the internal image type.

```
  typedef float InternalPixelType;
  typedef itk::Image< InternalPixelType, Dimension > InternalImageType;
  typedef itk::CastImageFilter< FixedImageType,
                                InternalImageType > FixedImageCasterType;
  typedef itk::CastImageFilter< MovingImageType,
                                InternalImageType > MovingImageCasterType;

  FixedImageCasterType::Pointer fixedImageCaster   = FixedImageCasterType::New();
  MovingImageCasterType::Pointer movingImageCaster = MovingImageCasterType::New();

  fixedImageCaster->SetInput( fixedImageReader->GetOutput() );
  movingImageCaster->SetInput( movingImageReader->GetOutput() );
```

The level set motion algorithm relies on the assumption that pixels representing the same homologous point on an object have the same intensity on both the fixed and moving images to be registered. In this example, we will preprocess the moving image to match the intensity between the images using the `itk::HistogramMatchingImageFilter`.

The basic idea is to match the histograms of the two images at a user-specified number of quantile values. For robustness, the histograms are matched so that the background pixels are excluded from both histograms. For MR images, a simple procedure is to exclude all gray values that are smaller than the mean gray value of the image.

```
  typedef itk::HistogramMatchingImageFilter<
                                    InternalImageType,
                                    InternalImageType >   MatchingFilterType;
  MatchingFilterType::Pointer matcher = MatchingFilterType::New();
```

For this example, we set the moving image as the source or input image and the fixed image as the reference image.

```
matcher->SetInput( movingImageCaster->GetOutput() );
matcher->SetReferenceImage( fixedImageCaster->GetOutput() );
```

We then select the number of bins to represent the histograms and the number of points or quantile values where the histogram is to be matched.

```
matcher->SetNumberOfHistogramLevels( 1024 );
matcher->SetNumberOfMatchPoints( 7 );
```

Simple background extraction is done by thresholding at the mean intensity.

```
matcher->ThresholdAtMeanIntensityOn();
```

In the `itk::LevelSetMotionRegistrationFilter`, the deformation field is represented as an image whose pixels are floating point vectors.

```
typedef itk::Vector< float, Dimension >    VectorPixelType;
typedef itk::Image<  VectorPixelType, Dimension > DeformationFieldType;
typedef itk::LevelSetMotionRegistrationFilter<
                              InternalImageType,
                              InternalImageType,
                              DeformationFieldType>   RegistrationFilterType;
RegistrationFilterType::Pointer filter = RegistrationFilterType::New();
```

The input fixed image is simply the output of the fixed image casting filter. The input moving image is the output of the histogram matching filter.

```
filter->SetFixedImage( fixedImageCaster->GetOutput() );
filter->SetMovingImage( matcher->GetOutput() );
```

The level set motion registration filter has two parameters: the number of iterations to be performed and the standard deviation of the Gaussian smoothing kernel to be applied to the image prior to calculating gradients.

```
filter->SetNumberOfIterations( 50 );
filter->SetGradientSmoothingStandardDeviations(4);
```

The registration algorithm is triggered by updating the filter. The filter output is the computed deformation field.

```
filter->Update();
```

The itk::WarpImageFilter can be used to warp the moving image with the output deformation field. Like the itk::ResampleImageFilter, the WarpImageFilter requires the specification of the input image to be resampled, an input image interpolator, and the output image spacing and origin.

```
typedef itk::WarpImageFilter<
                        MovingImageType,
                        MovingImageType,
                        DeformationFieldType  >     WarperType;
typedef itk::LinearInterpolateImageFunction<
                                 MovingImageType,
                                 double          >  InterpolatorType;
WarperType::Pointer warper = WarperType::New();
InterpolatorType::Pointer interpolator = InterpolatorType::New();
FixedImageType::Pointer fixedImage = fixedImageReader->GetOutput();

warper->SetInput( movingImageReader->GetOutput() );
warper->SetInterpolator( interpolator );
warper->SetOutputSpacing( fixedImage->GetSpacing() );
warper->SetOutputOrigin( fixedImage->GetOrigin() );
```

Unlike the ResampleImageFilter, the WarpImageFilter warps or transform the input image with respect to the deformation field represented by an image of vectors. The resulting warped or resampled image is written to file as per previous examples.

```
warper->SetDeformationField( filter->GetOutput() );
```

Let's execute this example using the rat lung data from the previous example. The associated data files can be found in `Examples/Data`:

- `RatLungSlice1.mha`
- `RatLungSlice2.mha`

The result of the demons-based deformable registration is presented in Figure 8.50. The checkerboard comparison shows that the algorithm was able to recover the misalignment due to expiration.

It may be also desirable to write the deformation field as an image of vectors. This can be done with the following code.

```
typedef itk::ImageFileWriter< DeformationFieldType > FieldWriterType;
FieldWriterType::Pointer fieldWriter = FieldWriterType::New();
fieldWriter->SetFileName( argv[4] );
fieldWriter->SetInput( filter->GetOutput() );

fieldWriter->Update();
```

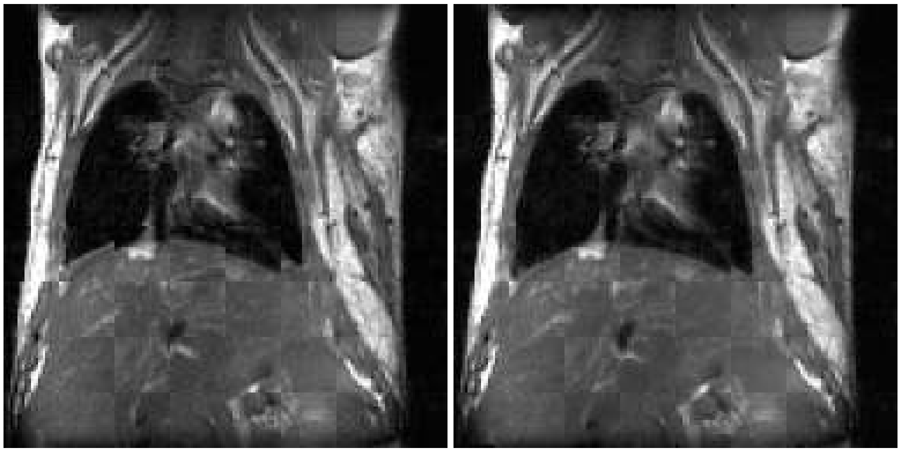


Figure 8.50: Checkerboard comparisons before and after demons-based deformable registration.

Note that the file format used for writing the deformation field must be capable of representing multiple components per pixel. This is the case for the MetaImage and VTK file formats for example.

The source code for this section can be found in the file
`Examples/Registration/DeformableRegistration6.cxx`.

This example illustrates the use of the `itk::BSplineDeformableTransform` class in a manually controlled multi-resolution scheme. Here we define two transforms at two different resolution levels. A first registration is performed with the spline grid of low resolution, and the results are then used for initializing a higher resolution grid. Since this example is quite similar to the previous example on the use of the `BSplineDeformableTransform` we omit here most of the details already discussed and will focus on the aspects related to the multi-resolution approach.

We include the header files for the transform and the optimizer.

```
#include "itkBSplineDeformableTransform.h"
#include "itkLBFGSOptimizer.h"
```

We instantiate now the type of the `BSplineDeformableTransform` using as template parameters the type for coordinates representation, the dimension of the space, and the order of the BSpline.

```
  const unsigned int SpaceDimension = ImageDimension;
  const unsigned int SplineOrder = 3;
  typedef double CoordinateRepType;
```

```
typedef itk::BSplineDeformableTransform<
                          CoordinateRepType,
                          SpaceDimension,
                          SplineOrder >     TransformType;
```

We construct two transform objects, each one will be configured for a resolution level. Notice than in this multi-resolution scheme we are not modifying the resolution of the image, but rather the flexibility of the deformable transform itself.

```
TransformType::Pointer  transformLow = TransformType::New();
registration->SetTransform( transformLow );
```

Here we define the parameters of the BSplineDeformableTransform grid. We arbitrarily decide to use a grid with 5×5 nodes within the image. The reader should note that the BSpline computation requires a finite support region (1 grid node at the lower borders and 2 grid nodes at upper borders). Therefore in this example, we set the grid size to be 8×8 and place the grid origin such that grid node (1,1) coincides with the first pixel in the fixed image.

Here we define the parameters of the BSpline transform at low resolution

```
RegionType::SizeType   gridLowSizeOnImage;
gridLowSizeOnImage.Fill( 5 );
totalGridSize = gridLowSizeOnImage + gridBorderSize;

RegionType bsplineRegion;
bsplineRegion.SetSize( totalGridSize );

typedef TransformType::SpacingType SpacingType;
SpacingType spacingLow = fixedImage->GetSpacing();

typedef TransformType::OriginType OriginType;
OriginType originLow = fixedImage->GetOrigin();;

FixedImageType::SizeType fixedImageSize = fixedRegion.GetSize();

for(unsigned int r=0; r<ImageDimension; r++)
  {
  spacingLow[r] *= floor( static_cast<double>(fixedImageSize[r] - 1)  /
                          static_cast<double>(gridLowSizeOnImage[r] - 1) );
  originLow[r]  -=  spacingLow[r];
  }

transformLow->SetGridSpacing( spacingLow );
transformLow->SetGridOrigin( originLow );
transformLow->SetGridRegion( bsplineRegion );

typedef TransformType::ParametersType     ParametersType;
```

```
const unsigned int numberOfParameters =
             transformLow->GetNumberOfParameters();

ParametersType parametersLow( numberOfParameters );

parametersLow.Fill( 0.0 );

transformLow->SetParameters( parametersLow );
```

We now pass the parameters of the current transform as the initial parameters to be used when the registration process starts.

```
registration->SetInitialTransformParameters( transformLow->GetParameters() );


optimizer->SetGradientConvergenceTolerance( 0.05 );
optimizer->SetLineSearchAccuracy( 0.9 );
optimizer->SetDefaultStepLength( 1.5 );
optimizer->TraceOn();
optimizer->SetMaximumNumberOfFunctionEvaluations( 1000 );


std::cout << "Starting Registration with low resolution transform" << std::endl;

try
  {
  registration->StartRegistration();
  }
catch( itk::ExceptionObject & err )
  {
  std::cerr << "ExceptionObject caught !" << std::endl;
  std::cerr << err << std::endl;
  return -1;
  }
```

Once the registration has finished with the low resolution grid, we proceed to instantiate a higher resolution `BSplineDeformableTransform`.

Now we need to initialize the BSpline coefficients of the higher resolution transform. This is done by first computing the actual deformation field at the higher resolution from the lower resolution BSpline coefficients. Then a BSpline decomposition is done to obtain the BSpline coefficient of the higher resolution transform.

We now pass the parameters of the high resolution transform as the initial parameters to be used in a second stage of the registration process.

```
registration->SetInitialTransformParameters( transformHigh->GetParameters() );
```

```
  registration->SetTransform( transformHigh );


    Typically, we will also want to tighten the optimizer parameters
    when we move from lower to higher resolution grid.
```

The source code for this section can be found in the file
`Examples/Registration/DeformableRegistration7.cxx`.

This example illustrates the use of the `itk::BSplineDeformableTransform` class for performing registration of two 3*D* images. The example code is for the most part identical to the code presented in Section 8.13. The major difference is that this example we set the image dimension to 3 and replace the `itk::LBFGSOptimizer` optimizer with the `itk::LBFGSBOptimizer`. This optimizer is more appropriate for performing optimization in a parametric spaces of higher dimensions.

The following are the most relevant headers to this example.

```
#include "itkBSplineDeformableTransform.h"
#include "itkLBFGSBOptimizer.h"
```

The parameter space of the `BSplineDeformableTransform` is composed by the set of all the deformations associated with the nodes of the BSpline grid. This large number of parameters makes possible to represent a wide variety of deformations, but it also has the price of requiring a significant amount of computation time.

We instantiate now the type of the `BSplineDeformableTransform` using as template parameters the type for coordinates representation, the dimension of the space, and the order of the BSpline.

```
  const unsigned int SpaceDimension = ImageDimension;
  const unsigned int SplineOrder = 3;
  typedef double CoordinateRepType;

  typedef itk::BSplineDeformableTransform<
                            CoordinateRepType,
                            SpaceDimension,
                            SplineOrder >     TransformType;
```

The transform object is constructed below and passed to the registration method.

```
  TransformType::Pointer  transform = TransformType::New();
  registration->SetTransform( transform );
```

Here we define the parameters of the BSplineDeformableTransform grid. We arbitrarily decide to use a grid with 5×5 nodes within the image. The reader should note that the BSpline computation requires a finite support region (1 grid node at the lower borders and 2 grid nodes at upper borders). Therefore in this example, we set the grid size to be 8×8 and place the grid origin such that grid node (1,1) coincides with the first pixel in the fixed image.

```
typedef TransformType::RegionType RegionType;
RegionType bsplineRegion;
RegionType::SizeType   gridSizeOnImage;
RegionType::SizeType   gridBorderSize;
RegionType::SizeType   totalGridSize;

gridSizeOnImage.Fill( 5 );
gridBorderSize.Fill( 3 );    // Border for spline order = 3 ( 1 lower, 2 upper )
totalGridSize = gridSizeOnImage + gridBorderSize;

bsplineRegion.SetSize( totalGridSize );

typedef TransformType::SpacingType SpacingType;
SpacingType spacing = fixedImage->GetSpacing();

typedef TransformType::OriginType OriginType;
OriginType origin = fixedImage->GetOrigin();;

FixedImageType::SizeType fixedImageSize = fixedRegion.GetSize();

for(unsigned int r=0; r<ImageDimension; r++)
  {
  spacing[r] *= floor( static_cast<double>(fixedImageSize[r] - 1)  /
                static_cast<double>(gridSizeOnImage[r] - 1) );
  origin[r]  -=  spacing[r];
  }

transform->SetGridSpacing( spacing );
transform->SetGridOrigin( origin );
transform->SetGridRegion( bsplineRegion );


typedef TransformType::ParametersType     ParametersType;

const unsigned int numberOfParameters =
             transform->GetNumberOfParameters();

ParametersType parameters( numberOfParameters );

parameters.Fill( 0.0 );

transform->SetParameters( parameters );
```

We now pass the parameters of the current transform as the initial parameters to be used when the registration process starts.

```
  registration->SetInitialTransformParameters( transform->GetParameters() );
```

Next we set the parameters of the LBFGSB Optimizer.

```
  transform->SetParameters( finalParameters );
```

The source code for this section can be found in the file
`Examples/Registration/DeformableRegistration8.cxx`.

This example illustrates the use of the `itk::BSplineDeformableTransform` class for performing registration of two $3D$ images and for the case of multi-modality images. The image metric of choice in this case is the `itk::MattesMutualInformationImageToImageMetric`.

The following are the most relevant headers to this example.

```
#include "itkBSplineDeformableTransform.h"
#include "itkLBFGSBOptimizer.h"
```

The parameter space of the `BSplineDeformableTransform` is composed by the set of all the deformations associated with the nodes of the B-spline grid. This large number of parameters makes possible to represent a wide variety of deformations, but it also has the price of requiring a significant amount of computation time.

We instantiate now the type of the `BSplineDeformableTransform` using as template parameters the type for coordinates representation, the dimension of the space, and the order of the B-spline.

```
  const unsigned int SpaceDimension = ImageDimension;
  const unsigned int SplineOrder = 3;
  typedef double CoordinateRepType;

  typedef itk::BSplineDeformableTransform<
                          CoordinateRepType,
                          SpaceDimension,
                          SplineOrder >     TransformType;
```

The transform object is constructed below and passed to the registration method.

```
  TransformType::Pointer  transform = TransformType::New();
  registration->SetTransform( transform );
```

Here we define the parameters of the BSplineDeformableTransform grid. We arbitrarily decide to use a grid with 5×5 nodes within the image. The reader should note that the B-spline computation requires a finite support region (1 grid node at the lower borders and 2 grid nodes at upper borders). Therefore in this example, we set the grid size to be 8×8 and place the grid origin such that grid node (1,1) coincides with the first pixel in the fixed image.

```
typedef TransformType::RegionType RegionType;
RegionType bsplineRegion;
RegionType::SizeType   gridSizeOnImage;
RegionType::SizeType   gridBorderSize;
RegionType::SizeType   totalGridSize;

gridSizeOnImage.Fill( 12 );
gridBorderSize.Fill( 3 );    // Border for spline order = 3 ( 1 lower, 2 upper )
totalGridSize = gridSizeOnImage + gridBorderSize;

bsplineRegion.SetSize( totalGridSize );

typedef TransformType::SpacingType SpacingType;
SpacingType spacing = fixedImage->GetSpacing();

typedef TransformType::OriginType OriginType;
OriginType origin = fixedImage->GetOrigin();;

FixedImageType::SizeType fixedImageSize = fixedRegion.GetSize();

for(unsigned int r=0; r<ImageDimension; r++)
  {
  spacing[r] *= floor( static_cast<double>(fixedImageSize[r] - 1)  /
                static_cast<double>(gridSizeOnImage[r] - 1) );
  origin[r]  -=  spacing[r];
  }

transform->SetGridSpacing( spacing );
transform->SetGridOrigin( origin );
transform->SetGridRegion( bsplineRegion );


typedef TransformType::ParametersType     ParametersType;

const unsigned int numberOfParameters =
             transform->GetNumberOfParameters();

ParametersType parameters( numberOfParameters );

parameters.Fill( 0.0 );

transform->SetParameters( parameters );
```

We now pass the parameters of the current transform as the initial parameters to be used when the registration process starts.

```
  registration->SetInitialTransformParameters( transform->GetParameters() );
```

Next we set the parameters of the LBFGSB Optimizer.

```
  OptimizerType::BoundSelectionType boundSelect( transform->GetNumberOfParameters() );
  OptimizerType::BoundValueType upperBound( transform->GetNumberOfParameters() );
  OptimizerType::BoundValueType lowerBound( transform->GetNumberOfParameters() );

  boundSelect.Fill( 0 );
  upperBound.Fill( 0.0 );
  lowerBound.Fill( 0.0 );

  optimizer->SetBoundSelection( boundSelect );
  optimizer->SetUpperBound( upperBound );
  optimizer->SetLowerBound( lowerBound );

  optimizer->SetCostFunctionConvergenceFactor( 1e+7 );
  optimizer->SetProjectedGradientTolerance( 1e-4 );
  optimizer->SetMaximumNumberOfIterations( 500 );
  optimizer->SetMaximumNumberOfEvaluations( 500 );
  optimizer->SetMaximumNumberOfCorrections( 12 );
```

Next we set the parameters of the Mattes Mutual Information Metric.

```
  metric->SetNumberOfHistogramBins( 50 );

  const unsigned int numberOfSamples = fixedRegion.GetNumberOfPixels() / 10;

  metric->SetNumberOfSpatialSamples( numberOfSamples );
```

Given that the Mattes Mutual Information metric uses a random iterator in order to collect the samples from the images, it is usually convenient to initialize the seed of the random number generator.

```
  metric->ReinitializeSeed( 76926294 );
```

```
  transform->SetParameters( finalParameters );
```

```
% Configuration file #1 for DeformableRegistration1.cxx
%
% This example demonstrates the setup of a basic registration
% problem that does NOT use multi-resolution strategies.  As a
% result, only one value for the parameters between
% (# of pixels per element) and (maximum iterations) is necessary.
% If you were using multi-resolution, you would have to specify
% values for those parameters at each level of the pyramid.
%
% Note: the paths in the parameters assume you have the traditional
% ITK file hierarchy as shown below:
%
% ITK/Insight/Examples/Registration/DeformableRegistration1.cxx
% ITK/Insight/Examples/Data/RatLungSlice*
% ITK/Insight-Bin/bin/DeformableRegistration1
%
% ---------------------------------------------------------
% Parameters for the single- or multi-resolution techniques
% ---------------------------------------------------------
1       % Number of levels in the multi-res pyramid (1 = single-res)
1       % Highest level to use in the pyramid
 1 1            % Scaling at lowest level of pyramid
 4              % Number of pixels per element
 1.e4           % Elasticity (E)
 1.e4           % Density x capacity (RhoC)
 1              % Image energy scaling (gamma) - sets gradient step size
 2              % NumberOfIntegrationPoints
 1              % WidthOfMetricRegion
 20             % MaximumIterations
% -------------------------------
% Parameters for the registration
% -------------------------------
0 0.99  % Similarity metric (0=mean sq, 1 = ncc, 2=pattern int, 3=MI, 5=demons)
1.0     % Alpha
0       % DescentDirection (1 = max, 0 = min)
0       % DoLineSearch (0=never, 1=always, 2=if needed)
1.e1    % TimeStep
0.5     % Landmark variance
0       % Employ regridding / enforce diffeomorphism ( >= 1 -> true)
% ----------------------------------
% Information about the image inputs
% ----------------------------------
128     % Nx (image x dimension)
128     % Ny (image y dimension)
0       % Nz (image z dimension - not used if 2D)
../../Insight/Examples/Data/RatLungSlice1.mha  % ReferenceFileName
../../Insight/Examples/Data/RatLungSlice2.mha  % TargetFileName
% -------------------------------------------------------------------
% The actions below depend on the values of the flags preceding them.
% For example, to write out the displacement fields, you have to set
% the value of WriteDisplacementField to 1.
% -------------------------------------------------------------------
```

```
0       % UseLandmarks? - read the file name below if this is true
-       % LandmarkFileName
./RatLung_result                       % ResultsFileName (prefix only)
1       % WriteDisplacementField?
./RatLung_disp                         % DisplacementsFileName (prefix only)
0       % ReadMeshFile?
-                                      % MeshFileName
END
```

The source code for this section can be found in the file
`Examples/Registration/LandmarkWarping2.cxx`.

This example illustrates how to deform an image using a KernelBase spline and two sets of landmarks.

```
#include "itkVector.h"
#include "itkImage.h"
#include "itkImageRegionIteratorWithIndex.h"
#include "itkDeformationFieldSource.h"
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
#include "itkWarpImageFilter.h"
#include "itkLinearInterpolateImageFunction.h"
```

The source code for this section can be found in the file
`Examples/Registration/BSplineWarping1.cxx`.

This example illustrates how to deform an image using a BSplineTransform.

```
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"

#include "itkImage.h"
#include "itkResampleImageFilter.h"
#include "itkLinearInterpolateImageFunction.h"

#include "itkBSplineDeformableTransform.h"

#include <fstream>


int main( int argc, char * argv[] )
{

  if( argc < 5 )
    {
```

```
    std::cerr << "Missing Parameters " << std::endl;
    std::cerr << "Usage: " << argv[0];
    std::cerr << " coefficientsFile fixedImage ";
    std::cerr << "movingImage deformedMovingImage" << std::endl;
    std::cerr << "[deformationField]" << std::endl;
    return 1;
    }

  const     unsigned int   ImageDimension = 2;

  typedef   unsigned char  PixelType;
  typedef   itk::Image< PixelType, ImageDimension >  FixedImageType;
  typedef   itk::Image< PixelType, ImageDimension >  MovingImageType;

  typedef   itk::ImageFileReader< FixedImageType  >  FixedReaderType;
  typedef   itk::ImageFileReader< MovingImageType >  MovingReaderType;

  typedef   itk::ImageFileWriter< MovingImageType >  MovingWriterType;


  FixedReaderType::Pointer fixedReader = FixedReaderType::New();
  fixedReader->SetFileName( argv[2] );

  try
    {
    fixedReader->Update();
    }
  catch( itk::ExceptionObject & excp )
    {
    std::cerr << "Exception thrown " << std::endl;
    std::cerr << excp << std::endl;
    return EXIT_FAILURE;
    }


  MovingReaderType::Pointer movingReader = MovingReaderType::New();
  MovingWriterType::Pointer movingWriter = MovingWriterType::New();

  movingReader->SetFileName( argv[3] );
  movingWriter->SetFileName( argv[4] );


  FixedImageType::ConstPointer fixedImage = fixedReader->GetOutput();


  typedef itk::ResampleImageFilter< MovingImageType,
                                    FixedImageType  >  FilterType;
```

```
FilterType::Pointer resampler = FilterType::New();

typedef itk::LinearInterpolateImageFunction<
                     MovingImageType, double >  InterpolatorType;

InterpolatorType::Pointer interpolator = InterpolatorType::New();

resampler->SetInterpolator( interpolator );

FixedImageType::SpacingType fixedSpacing = fixedImage->GetSpacing();
FixedImageType::PointType   fixedOrigin  = fixedImage->GetOrigin();

resampler->SetOutputSpacing( fixedSpacing );
resampler->SetOutputOrigin(  fixedOrigin  );


FixedImageType::RegionType fixedRegion = fixedImage->GetBufferedRegion();
FixedImageType::SizeType   fixedSize =  fixedRegion.GetSize();
resampler->SetSize( fixedSize );
resampler->SetOutputStartIndex(  fixedRegion.GetIndex() );


resampler->SetInput( movingReader->GetOutput() );

movingWriter->SetInput( resampler->GetOutput() );
```

We instantiate now the type of the `BSplineDeformableTransform` using as template parameters the type for coordinates representation, the dimension of the space, and the order of the B-spline.

```
const unsigned int SpaceDimension = ImageDimension;
const unsigned int SplineOrder = 3;
typedef double CoordinateRepType;

typedef itk::BSplineDeformableTransform<
                          CoordinateRepType,
                          SpaceDimension,
                          SplineOrder >     TransformType;

TransformType::Pointer bsplineTransform = TransformType::New();
```

Since we are using a B-spline of order 3, the coverage of the BSpling grid should exceed by one the spatial extent of the image on the lower region of image indices, and by two grid points on the upper region of image indices. We choose here to use a 8×8 B-spline grid, from which only a 5×5 sub-grid will be covering the input image. If we use an input image of size 500×500

pixels, and pixel spacing 2.0×2.0 then we need the 5×5 B-spline grid to cover a physical extent of 1000×1000 mm. This can be achieved by setting the pixel spacing of the B-spline grid to 250.0×250.0 mm. The origin of the B-spline grid must be set at one grid position away from the origin of the desired output image. All this is done with the following lines of code.

```
typedef TransformType::RegionType RegionType;
RegionType bsplineRegion;
RegionType::SizeType   size;

const unsigned int numberOfGridNodesOutsideTheImageSupport = 3;

const unsigned int numberOfGridNodesInsideTheImageSupport = 5;

const unsigned int numberOfGridNodes =
                      numberOfGridNodesInsideTheImageSupport +
                      numberOfGridNodesOutsideTheImageSupport;

const unsigned int numberOfGridCells =
                      numberOfGridNodesInsideTheImageSupport - 1;

size.Fill( numberOfGridNodes );
bsplineRegion.SetSize( size );

typedef TransformType::SpacingType SpacingType;
SpacingType spacing;
spacing[0] = floor( fixedSpacing[0] * (fixedSize[0] - 1) / numberOfGridCells );
spacing[1] = floor( fixedSpacing[1] * (fixedSize[1] - 1) / numberOfGridCells );

typedef TransformType::OriginType OriginType;
OriginType origin;
origin[0] = fixedOrigin[0] - spacing[0];
origin[1] = fixedOrigin[1] - spacing[1];

bsplineTransform->SetGridSpacing( spacing );
bsplineTransform->SetGridOrigin( origin );
bsplineTransform->SetGridRegion( bsplineRegion );


typedef TransformType::ParametersType     ParametersType;

const unsigned int numberOfParameters =
             bsplineTransform->GetNumberOfParameters();


const unsigned int numberOfNodes = numberOfParameters / SpaceDimension;

ParametersType parameters( numberOfParameters );
```

The B-spline grid should now be fed with coeficients at each node. Since this is a two dimensional grid, each node should receive two coefficients. Each coefficient pair is representing a displacement vector at this node. The coefficients can be passed to the B-spline in the form of an array where the first set of elements are the first component of the displacements for all the nodes, and the second set of elemets is formed by the second component of the displacements for all the nodes.

In this example we read such displacements from a file, but for convinience we have written this file using the pairs of (x,y) displacement for every node. The elements read from the file should therefore be reorganized when assigned to the elements of the array. We do this by storing all the odd elements from the file in the first block of the array, and all the even elements from the file in the second block of the array. Finally the array is passed to the B-spline transform using the `SetParameters()`.

```
std::ifstream infile;

infile.open( argv[1] );

for( unsigned int n=0; n < numberOfNodes; n++ )
  {
  infile >>  parameters[n];
  infile >>  parameters[n+numberOfNodes];
  }

infile.close();
```

Finally the array is passed to the B-spline transform using the `SetParameters()`.

```
bsplineTransform->SetParameters( parameters );
```

At this point we are ready to use the transform as part of the resample filter. We trigger the execution of the pipeline by invoking `Update()` on the last filter of the pipeline, in this case writer.

```
resampler->SetTransform( bsplineTransform );

try
  {
  movingWriter->Update();
  }
catch( itk::ExceptionObject & excp )
  {
  std::cerr << "Exception thrown " << std::endl;
  std::cerr << excp << std::endl;
  return EXIT_FAILURE;
  }
```

8.14 Demons Deformable Registration

For the problem of intra-modality deformable registration, the Insight Toolkit provides an implementation of Thirion's "demons" algorithm [80, 81]. In this implementation, each image is viewed as a set of iso-intensity contours. The main idea is that a regular grid of forces deform an image by pushing the contours in the normal direction. The orientation and magnitude of the displacement is derived from the instantaneous optical flow equation:

$$\mathbf{D}(\mathbf{X}) \cdot \nabla \mathbf{f}(\mathbf{X}) = -\left(\mathbf{m}(\mathbf{X}) - \mathbf{f}(\mathbf{X})\right) \tag{8.36}$$

In the above equation, $f(\mathbf{X})$ is the fixed image, $m(\mathbf{X})$ is the moving image to be registered, and $\mathbf{D}(\mathbf{X})$ is the displacement or optical flow between the images. It is well known in optical flow literature that Equation 8.36 is insufficient to specify $\mathbf{D}(\mathbf{X})$ locally and is usually determined using some form of regularization. For registration, the projection of the vector on the direction of the intensity gradient is used:

$$\mathbf{D}(\mathbf{X}) = -\frac{\left(\mathbf{m}(\mathbf{X}) - \mathbf{f}(\mathbf{X})\right) \nabla \mathbf{f}(\mathbf{X})}{\|\nabla \mathbf{f}\|^2} \tag{8.37}$$

However, this equation becomes unstable for small values of the image gradient, resulting in large displacement values. To overcome this problem, Thirion re-normalizes the equation such that:

$$\mathbf{D}(\mathbf{X}) = -\frac{\left(\mathbf{m}(\mathbf{X}) - \mathbf{f}(\mathbf{X})\right) \nabla \mathbf{f}(\mathbf{X})}{\|\nabla \mathbf{f}\|^2 + \left(\mathbf{m}(\mathbf{X}) - \mathbf{f}(\mathbf{X})\right)^2 / \mathbf{K}} \tag{8.38}$$

Where K is a normalization factor that accounts for the units imbalance between intensities and gradients. This factor is computed as the mean squared value of the pixel spacings. The inclusion of K make the force computation to be invariant to the pixel scaling of the images.

Starting with an initial deformation field $\mathbf{D^0}(\mathbf{X})$, the demons algorithm iteratively updates the field using Equation 8.38 such that the field at the N-th iteration is given by:

$$\mathbf{D^N}(\mathbf{X}) = \mathbf{D^{N-1}}(\mathbf{X}) - \frac{\left(\mathbf{m}(\mathbf{X} + \mathbf{D^{N-1}}(\mathbf{X})) - \mathbf{f}(\mathbf{X})\right) \nabla \mathbf{f}(\mathbf{X})}{\|\nabla \mathbf{f}\|^2 + \left(\mathbf{m}(\mathbf{X} + \mathbf{D^{N-1}}(\mathbf{X})) - \mathbf{f}(\mathbf{X})\right)^2} \tag{8.39}$$

Reconstruction of the deformation field is an ill-posed problem where matching the fixed and moving images has many solutions. For example, since each image pixel is free to move independently, it is possible that all pixels of one particular value in $m(\mathbf{X})$ could map to a single image pixel in $f(\mathbf{X})$ of the same value. The resulting deformation field may be unrealistic for real-world applications. An option to solve for the field uniquely is to enforce an elastic-like behavior, smoothing the deformation field with a Gaussian filter between iterations.

In ITK, the demons algorithm is implemented as part of the finite difference solver (FDS) framework and its use is demonstrated in the following example.

The source code for this section can be found in the file
`Examples/Registration/DeformableRegistration2.cxx`.

This example demonstrates how to use the "demons" algorithm to deformably register two images. The first step is to include the header files.

```
#include "itkDemonsRegistrationFilter.h"
#include "itkHistogramMatchingImageFilter.h"
#include "itkCastImageFilter.h"
#include "itkWarpImageFilter.h"
#include "itkLinearInterpolateImageFunction.h"
```

Second, we declare the types of the images.

```
  const unsigned int Dimension = 2;
  typedef unsigned short PixelType;

  typedef itk::Image< PixelType, Dimension >  FixedImageType;
  typedef itk::Image< PixelType, Dimension >  MovingImageType;
```

Image file readers are set up in a similar fashion to previous examples. To support the re-mapping of the moving image intensity, we declare an internal image type with a floating point pixel type and cast the input images to the internal image type.

```
  typedef float InternalPixelType;
  typedef itk::Image< InternalPixelType, Dimension > InternalImageType;
  typedef itk::CastImageFilter< FixedImageType,
                                InternalImageType > FixedImageCasterType;
  typedef itk::CastImageFilter< MovingImageType,
                                InternalImageType > MovingImageCasterType;

  FixedImageCasterType::Pointer fixedImageCaster   = FixedImageCasterType::New();
  MovingImageCasterType::Pointer movingImageCaster = MovingImageCasterType::New();

  fixedImageCaster->SetInput( fixedImageReader->GetOutput() );
  movingImageCaster->SetInput( movingImageReader->GetOutput() );
```

The demons algorithm relies on the assumption that pixels representing the same homologous point on an object have the same intensity on both the fixed and moving images to be registered. In this example, we will preprocess the moving image to match the intensity between the images using the `itk::HistogramMatchingImageFilter`.

The basic idea is to match the histograms of the two images at a user-specified number of quantile values. For robustness, the histograms are matched so that the background pixels are excluded from both histograms. For MR images, a simple procedure is to exclude all gray values that are smaller than the mean gray value of the image.

```
typedef itk::HistogramMatchingImageFilter<
                                  InternalImageType,
                                  InternalImageType >   MatchingFilterType;
MatchingFilterType::Pointer matcher = MatchingFilterType::New();
```

For this example, we set the moving image as the source or input image and the fixed image as the reference image.

```
matcher->SetInput( movingImageCaster->GetOutput() );
matcher->SetReferenceImage( fixedImageCaster->GetOutput() );
```

We then select the number of bins to represent the histograms and the number of points or quantile values where the histogram is to be matched.

```
matcher->SetNumberOfHistogramLevels( 1024 );
matcher->SetNumberOfMatchPoints( 7 );
```

Simple background extraction is done by thresholding at the mean intensity.

```
matcher->ThresholdAtMeanIntensityOn();
```

In the `itk::DemonsRegistrationFilter`, the deformation field is represented as an image whose pixels are floating point vectors.

```
typedef itk::Vector< float, Dimension >    VectorPixelType;
typedef itk::Image<  VectorPixelType, Dimension > DeformationFieldType;
typedef itk::DemonsRegistrationFilter<
                              InternalImageType,
                              InternalImageType,
                              DeformationFieldType>   RegistrationFilterType;
RegistrationFilterType::Pointer filter = RegistrationFilterType::New();
```

The input fixed image is simply the output of the fixed image casting filter. The input moving image is the output of the histogram matching filter.

```
filter->SetFixedImage( fixedImageCaster->GetOutput() );
filter->SetMovingImage( matcher->GetOutput() );
```

The demons registration filter has two parameters: the number of iterations to be performed and the standard deviation of the Gaussian smoothing kernel to be applied to the deformation field after each iteration.

```
filter->SetNumberOfIterations( 50 );
filter->SetStandardDeviations( 1.0 );
```

The registration algorithm is triggered by updating the filter. The filter output is the computed deformation field.

```
filter->Update();
```

The `itk::WarpImageFilter` can be used to warp the moving image with the output deformation field. Like the `itk::ResampleImageFilter`, the WarpImageFilter requires the specification of the input image to be resampled, an input image interpolator, and the output image spacing and origin.

```
typedef itk::WarpImageFilter<
                        MovingImageType,
                        MovingImageType,
                        DeformationFieldType  >     WarperType;
typedef itk::LinearInterpolateImageFunction<
                                 MovingImageType,
                                 double          >  InterpolatorType;
WarperType::Pointer warper = WarperType::New();
InterpolatorType::Pointer interpolator = InterpolatorType::New();
FixedImageType::Pointer fixedImage = fixedImageReader->GetOutput();

warper->SetInput( movingImageReader->GetOutput() );
warper->SetInterpolator( interpolator );
warper->SetOutputSpacing( fixedImage->GetSpacing() );
warper->SetOutputOrigin( fixedImage->GetOrigin() );
```

Unlike the ResampleImageFilter, the WarpImageFilter warps or transform the input image with respect to the deformation field represented by an image of vectors. The resulting warped or resampled image is written to file as per previous examples.

```
warper->SetDeformationField( filter->GetOutput() );
```

Let's execute this example using the rat lung data from the previous example. The associated data files can be found in `Examples/Data`:

- `RatLungSlice1.mha`
- `RatLungSlice2.mha`

The result of the demons-based deformable registration is presented in Figure 8.51. The checkerboard comparison shows that the algorithm was able to recover the misalignment due to expiration.

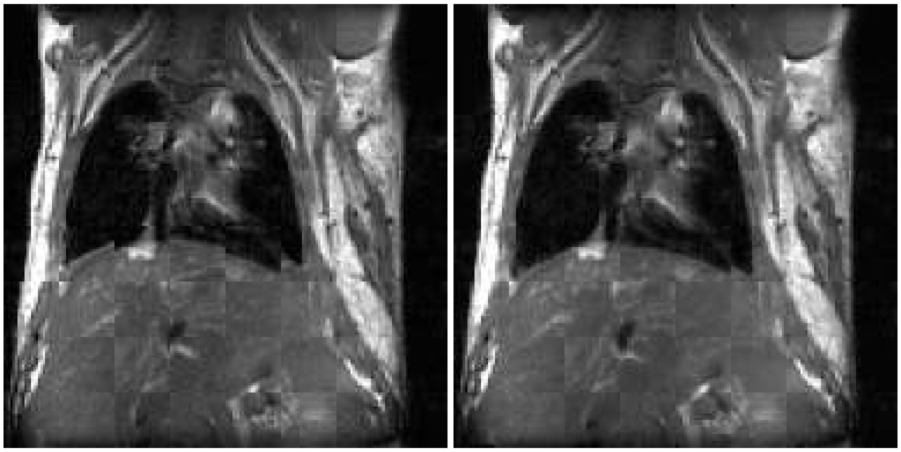


Figure 8.51: Checkerboard comparisons before and after demons-based deformable registration.

It may be also desirable to write the deformation field as an image of vectors. This can be done with the following code.

```
typedef itk::ImageFileWriter< DeformationFieldType > FieldWriterType;
FieldWriterType::Pointer fieldWriter = FieldWriterType::New();
fieldWriter->SetFileName( argv[4] );
fieldWriter->SetInput( filter->GetOutput() );

fieldWriter->Update();
```

Note that the file format used for writing the deformation field must be capable of representing multiple components per pixel. This is the case for the MetaImage and VTK file formats for example.

A variant of the force computation is also implemented in which the gradient of the deformed moving image is also involved. This provides a level of symmetry in the force calculation during one iteration of the PDE update. The equation used in this case is

$$\mathbf{D}(\mathbf{X}) = -\frac{\mathbf{2}\,(\mathbf{m}(\mathbf{X}) - \mathbf{f}(\mathbf{X}))\,(\nabla\mathbf{f}(\mathbf{X}) + \nabla\mathbf{g}(\mathbf{X}))}{\|\nabla\mathbf{f} + \nabla\mathbf{g}\|^{\mathbf{2}} + (\mathbf{m}(\mathbf{X}) - \mathbf{f}(\mathbf{X}))^{\mathbf{2}}\,/\mathbf{K}} \tag{8.40}$$

The following example illustrates the use of this deformable registration method.

The source code for this section can be found in the file
`Examples/Registration/DeformableRegistration3.cxx`.

This example demonstrates how to use a variant of the "demons" algorithm to deformably register two images. This variant uses a different formulation for computing the forces to be applied to the image in order to compute the deformation fields. The variant uses both the gradient of the fixed image and the gradient of the deformed moving image in order to compute the forces. This mechanism for computing the forces introduces a symmetry with respect to the choice of the fixed and moving images. This symmetry only holds during the computation of one iteration of the PDE updates. It is unlikely that total symmetry may be achieved by this mechanism for the entire registration process.

The first step for using this filter is to include the following header files.

```
#include "itkSymmetricForcesDemonsRegistrationFilter.h"
#include "itkHistogramMatchingImageFilter.h"
#include "itkCastImageFilter.h"
#include "itkWarpImageFilter.h"
#include "itkLinearInterpolateImageFunction.h"
```

Second, we declare the types of the images.

```
  const unsigned int Dimension = 2;
  typedef unsigned short PixelType;

  typedef itk::Image< PixelType, Dimension >  FixedImageType;
  typedef itk::Image< PixelType, Dimension >  MovingImageType;
```

Image file readers are set up in a similar fashion to previous examples. To support the re-mapping of the moving image intensity, we declare an internal image type with a floating point pixel type and cast the input images to the internal image type.

```
  typedef float InternalPixelType;
  typedef itk::Image< InternalPixelType, Dimension > InternalImageType;
  typedef itk::CastImageFilter< FixedImageType,
                                InternalImageType > FixedImageCasterType;
  typedef itk::CastImageFilter< MovingImageType,
                                InternalImageType > MovingImageCasterType;

  FixedImageCasterType::Pointer fixedImageCaster   = FixedImageCasterType::New();
  MovingImageCasterType::Pointer movingImageCaster = MovingImageCasterType::New();

  fixedImageCaster->SetInput( fixedImageReader->GetOutput() );
  movingImageCaster->SetInput( movingImageReader->GetOutput() );
```

The demons algorithm relies on the assumption that pixels representing the same homologous point on an object have the same intensity on both the fixed and moving images to be registered. In this example, we will preprocess the moving image to match the intensity between the images using the `itk::HistogramMatchingImageFilter`.

The basic idea is to match the histograms of the two images at a user-specified number of quantile values. For robustness, the histograms are matched so that the background pixels are excluded from both histograms. For MR images, a simple procedure is to exclude all gray values that are smaller than the mean gray value of the image.

```
typedef itk::HistogramMatchingImageFilter<
                                  InternalImageType,
                                  InternalImageType >   MatchingFilterType;
MatchingFilterType::Pointer matcher = MatchingFilterType::New();
```

For this example, we set the moving image as the source or input image and the fixed image as the reference image.

```
matcher->SetInput( movingImageCaster->GetOutput() );
matcher->SetReferenceImage( fixedImageCaster->GetOutput() );
```

We then select the number of bins to represent the histograms and the number of points or quantile values where the histogram is to be matched.

```
matcher->SetNumberOfHistogramLevels( 1024 );
matcher->SetNumberOfMatchPoints( 7 );
```

Simple background extraction is done by thresholding at the mean intensity.

```
matcher->ThresholdAtMeanIntensityOn();
```

In the `itk::SymmetricForcesDemonsRegistrationFilter`, the deformation field is represented as an image whose pixels are floating point vectors.

```
typedef itk::Vector< float, Dimension >    VectorPixelType;
typedef itk::Image<  VectorPixelType, Dimension > DeformationFieldType;
typedef itk::SymmetricForcesDemonsRegistrationFilter<
                              InternalImageType,
                              InternalImageType,
                              DeformationFieldType>   RegistrationFilterType;
RegistrationFilterType::Pointer filter = RegistrationFilterType::New();
```

The input fixed image is simply the output of the fixed image casting filter. The input moving image is the output of the histogram matching filter.

```
filter->SetFixedImage( fixedImageCaster->GetOutput() );
filter->SetMovingImage( matcher->GetOutput() );
```

The demons registration filter has two parameters: the number of iterations to be performed and the standard deviation of the Gaussian smoothing kernel to be applied to the deformation field after each iteration.

```
  filter->SetNumberOfIterations( 50 );
  filter->SetStandardDeviations( 1.0 );
```

The registration algorithm is triggered by updating the filter. The filter output is the computed deformation field.

```
  filter->Update();
```

The `itk::WarpImageFilter` can be used to warp the moving image with the output deformation field. Like the `itk::ResampleImageFilter`, the WarpImageFilter requires the specification of the input image to be resampled, an input image interpolator, and the output image spacing and origin.

```
  typedef itk::WarpImageFilter<
                          MovingImageType,
                          MovingImageType,
                          DeformationFieldType  >     WarperType;
  typedef itk::LinearInterpolateImageFunction<
                                   MovingImageType,
                                   double          >  InterpolatorType;
  WarperType::Pointer warper = WarperType::New();
  InterpolatorType::Pointer interpolator = InterpolatorType::New();
  FixedImageType::Pointer fixedImage = fixedImageReader->GetOutput();

  warper->SetInput( movingImageReader->GetOutput() );
  warper->SetInterpolator( interpolator );
  warper->SetOutputSpacing( fixedImage->GetSpacing() );
  warper->SetOutputOrigin( fixedImage->GetOrigin() );
```

Unlike the ResampleImageFilter, the WarpImageFilter warps or transform the input image with respect to the deformation field represented by an image of vectors. The resulting warped or resampled image is written to file as per previous examples.

```
  warper->SetDeformationField( filter->GetOutput() );
```

Let's execute this example using the rat lung data from the previous example. The associated data files can be found in `Examples/Data`:

- `RatLungSlice1.mha`
- `RatLungSlice2.mha`

The result of the demons-based deformable registration is presented in Figure 8.52. The checkerboard comparison shows that the algorithm was able to recover the misalignment due to expiration.

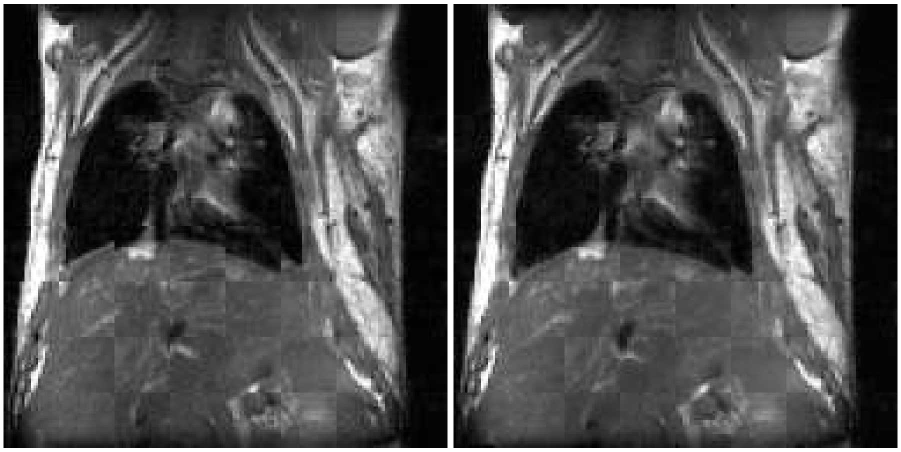


Figure 8.52: Checkerboard comparisons before and after demons-based deformable registration.

It may be also desirable to write the deformation field as an image of vectors. This can be done with the following code.

```
typedef itk::ImageFileWriter< DeformationFieldType > FieldWriterType;

FieldWriterType::Pointer fieldWriter = FieldWriterType::New();
fieldWriter->SetFileName( argv[4] );
fieldWriter->SetInput( filter->GetOutput() );

fieldWriter->Update();
```

Note that the file format used for writing the deformation field must be capable of representing multiple components per pixel. This is the case for the MetaImage and VTK file formats for example.

8.15 Visualizing Deformation fields

Vector deformation fields may be visualized using ParaView. ParaView [35] is an open-source, multi-platform visualization application and uses the Visualization Toolkit as the data processing and rendering engine and has a user interface written using a unique blend of Tcl/Tk and C++. You may download it from http://paraview.org.

8.15.1 Visualizing 2D deformation fields

Let us visualize the deformation field obtained from Demons Registration algorithm generated from Insight/Examples/Registration/DeformableRegistration2.cxx.

Load the Deformation field in Paraview. (The deformation field must be capable of handling vector data, such as MetaImages). Paraview shows a color map of the magnitudes of the deformation fields as shown in 8.53.

Covert the deformation field to 3D vector data using a *Calculator*. The Calculator may be found in the *Filter* pull down menu. A screenshot of the calculator tab is shown in Figure 8.54. Although the deformation field is a 2D vector, we will generate a 3D vector with the third component set to 0 since Paraview generates glyphs only for 3D vectors. You may now apply a glyph of arrows to the resulting 3D vector field by using *Glyph* on the menu bar. The glyphs obtained will be very dense since a glyph is generated for each point in the data set. To better visualize the deformation field, you may adopt one of the following approaches.

Reduce the number of glyphs by reducing the number in *Max. Number of Glyphs* to reasonable amount. This uniformly downsamples the number of glyphs. Alternatively, you may apply a *Threshold* filter to the *Magnitude* of the vector dataset and then glyph the vector data that lies above the threshold. This eliminates the smaller deformation fields that clutter the display. You may now reduce the number of glyphs to a reasonable value.

Figure 8.55 shows the vector field visualized using Paraview by thresholding the vector magnitudes by 2.1 and restricting the number of glyphs to 100.

8.15.2 Visualizing 3D deformation fields

Let us create a 3D deformation field. We will use Thin Plate Splines to warp a 3D dataset and create a deformation field. We will pick a set of point landmarks and translate them to provide a specification of correspondences at point landmarks. Note that the landmarks have been picked randomly for purposes of illustration and are not intended to portray a true deformation. The landmarks may be used to produce a deformation field in several ways. Most techniques minimize some regularizing functional representing the irregularity of the deformation field, which is usually some function of the spatial derivatives of the field. Here will we use *thin plate splines*. Thin plate splines minimize the regularizing functional

$$I[f(x,y)] = \iint (f_{xx}^2 + 2f_{xy}^2 + f_{yy}^2)dxdy \tag{8.41}$$

where the subscripts denote partial derivatives of f.

The code for this section can be found in Insight/Examples/Registration/ThinPlateSplineWarp.cxx

We may now proceed as before to visualize the deformation field using Paraview as shown in Figure 8.56.

Let us register the deformed volumes generated by Thin plate warping in the previous example

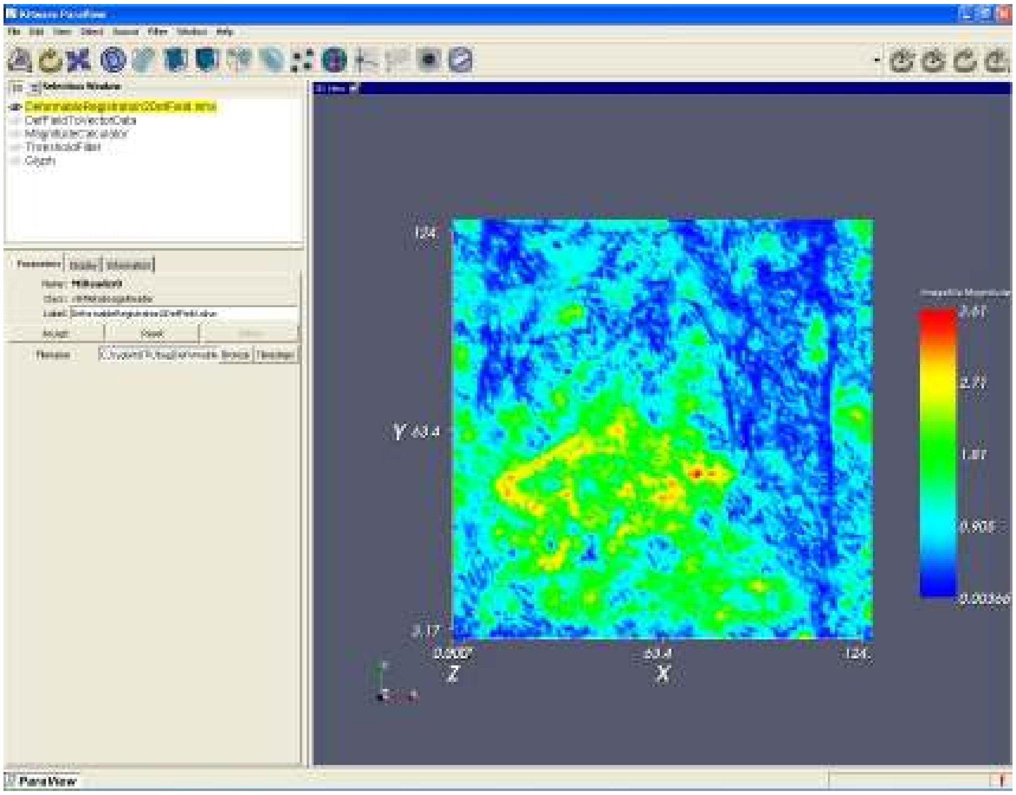


Figure 8.53: Deformation field magnitudes displayed using Paraview

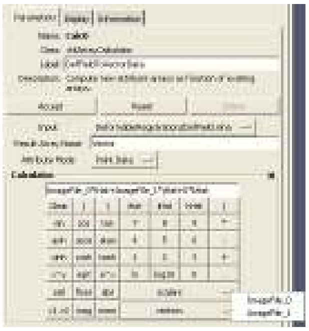

Parameters Display Information
Name:
Class: vtkArrayCalculator
Label: DefFieldToVectorData
Description: Compute new attribute arrays as function of existing arrays.
Accept Reset Delete
Input
Result Array Name
Attribute Mode
Point Data
Calculator
Clear
scalars
vectors
ImageFile_0
ImageFile_1

Figure 8.54: Calculators and filters may be used to compute the vector magnitude, compose vectors etc.

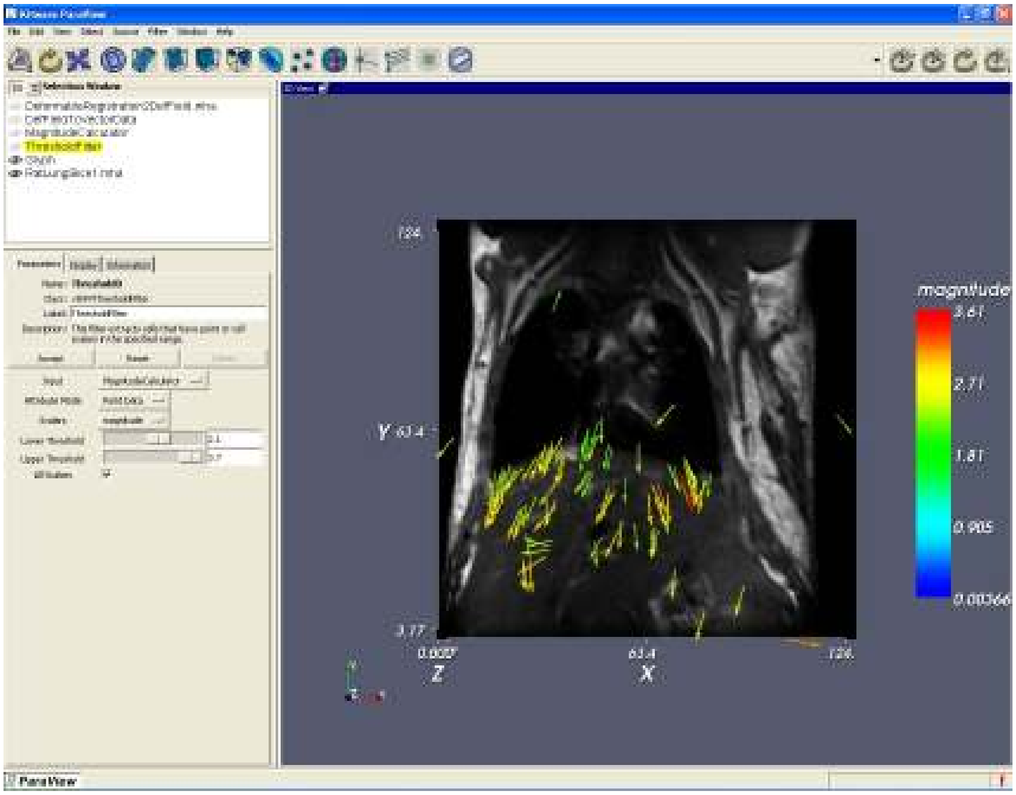


Figure 8.55: Deformation field visualized using Paraview after thresholding and subsampling.

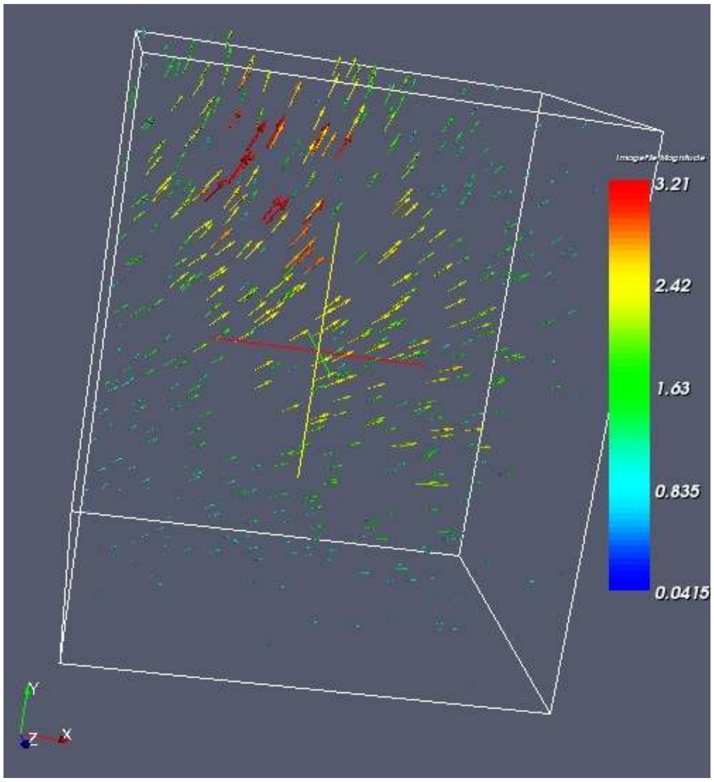


Figure 8.56: 3D Deformation field visualized using Paraview.

Iteration	**Function value**	$\|\|G\|\|$	**Step length**
1	156.981	14.911	0.202
2	68.956	11.774	1.500
3	38.146	4.802	1.500
4	26.690	2.515	1.500
5	23.295	1.106	1.500
6	21.454	1.032	1.500
7	20.322	1.557	1.500
8	19.751	0.594	1.500

Table 8.17: LBFGS Optimizer trace.

using DeformableRegistration4.cxx. Since ITK is in general N-dimensional, the only change in the example is to replace the `ImageDimension` by 3.

The registration method uses B-splines and an LBFGS optimizer. The trace in Table. 8.17 prints the trace of the optimizer through the search space.

Here $||G||$ is the norm of the gradient at the current estimate of the minimum, x. “Function Value” is the current value of the function, f(x).

The resulting deformation field that maps the moving to the fixed image is shown in 8.57. A difference image of two slices before and after registration is shown in 8.58. As can be seen from the figures, the deformation field is in close agreement to the one generated from the Thin plate spline warping.

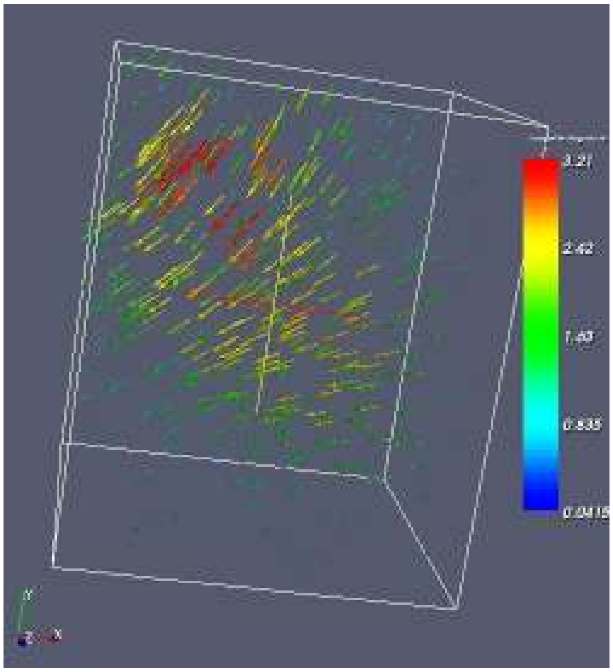


Figure 8.57: Resulting deformation field that maps the moving image to the fixed image.

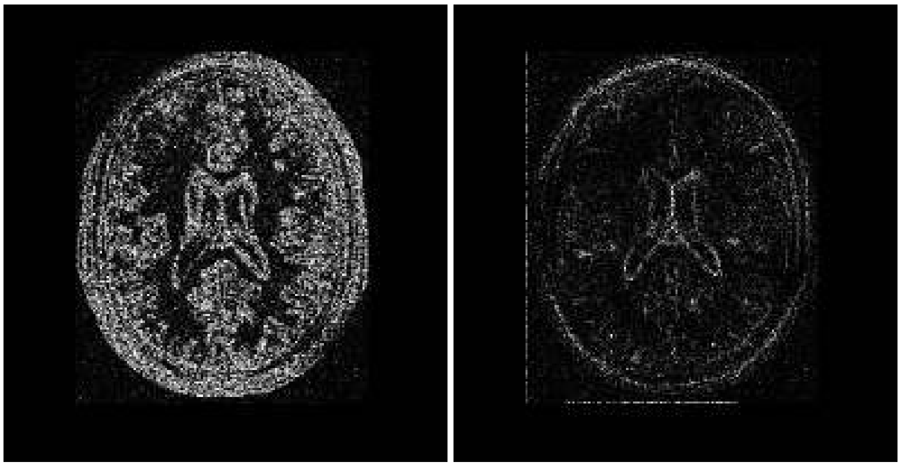


Figure 8.58: Difference image from a slice before and after registration.

8.16 Model Based Registration

This section introduces the concept of registering a geometrical model with an image. We refer to this concept as *model based registration* but this may not be the most widespread terminology. In this approach, a geometrical model is built first and a number of parameters are identified in the model. Variations of these parameters make it possible to adapt the model to the morphology of a particular patient. The task of registration is then to find the optimal combination of model parameters that will make this model a good representation of the anatomical structures contained in an image.

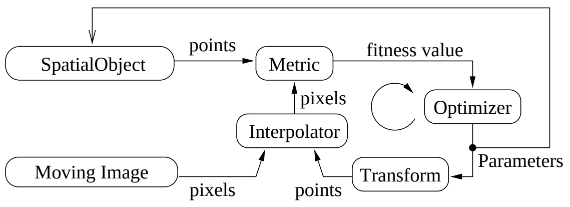


Figure 8.59: The basic components of model based registration are an image, a spatial object, a transform, a metric, an interpolator and an optimizer.

For example, let's say that in the axial view of a brain image we can roughly approximate the skull with an ellipse. The ellipse becomes our simplified geometrical model, and registration is the task of finding the best center for the ellipse, the measures of its axis lengths and its orientation in the plane. This is illustrated in Figure 8.60. If we compare this approach with the image-to-image registration problem, we can see that the main difference here is that in addition to mapping the spatial position of the model, we can also customize internal parameters that change its shape.

Figure 8.59 illustrates the major components of the registration framework in ITK when a model base registration problem is configured. The basic input data for the registration is provided by pixel data in an `itk::Image` and by geometrical data stored in a `itk::SpatialObject`. A metric has to be defined in order to evaluate the fitness between the model and the image. This fitness value can be improved by introducing variations in the spatial positioning of the SpatialObject and/or by changing its internal parameters. The search space for the optimizer is now the composition of the transform parameter and the shape internal parameters.

This same approach can be considered a segmentation technique, since once the model has been optimally superimposed on the image we could label pixels according to their associations with specific parts of the model. The applications of model to image registration/segmentation are endless. The main advantage of this approach is probably that, as opposed to image-to-image registration, it actually provides *Insight* into the anatomical structure contained in the image. The adapted model becomes a condensed representation of the essential elements of the anatomical structure.

ITK provides a hierarchy of classes intended to support the construction of shape models. This hierarchy has the SpatialObject as its base class. A number of basic functionalities are defined at this level, including the capacity to evaluate whether a given point is *inside* or *outside* of the model, form complex shapes by creating hierarchical conglomerates of basic shapes, and support basic spatial parameterizations like scale, orientation and position.

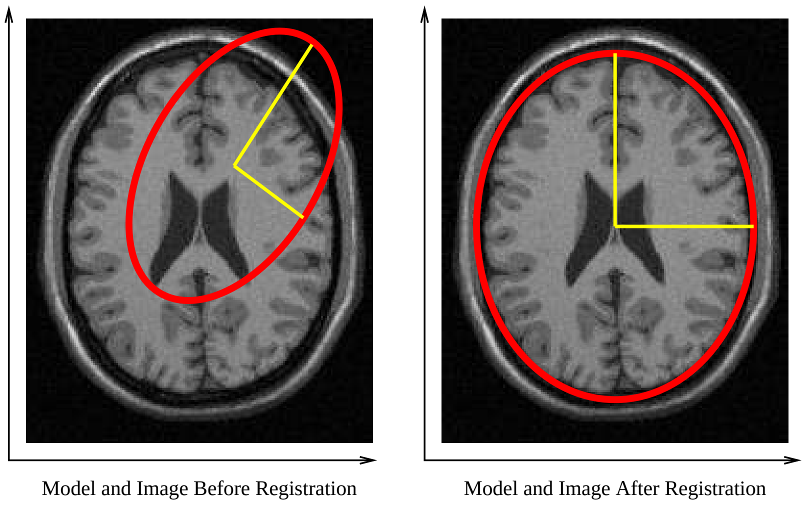

Model and Image Before Registration
Model and Image After Registration

Figure 8.60: Basic concept of Model-to-Image registration. A simplified geometrical model (ellipse) is registered against an anatomical structure (skull) by applying a spatial transform and modifying the model internal parameters. This image is not the result of an actual registration, it is shown here only with the purpose of illustrating the concept of model to image registration.

The following sections present examples of the typical uses of these powerful elements of the toolkit.

The source code for this section can be found in the file
`Examples/Registration/ModelToImageRegistration1.cxx`.

This example illustrates the use of the `itk::SpatialObject` as a component of the registration framework in order to perform model based registration. The current example creates a geometrical model composed of several ellipses. Then, it uses the model to produce a synthetic binary image of the ellipses. Next, it introduces perturbations on the position and shape of the model, and finally it uses the perturbed version as the input to a registration problem. A metric is defined to evaluate the fitness between the geometric model and the image.

Let's look first at the classes required to support SpatialObject. In this example we use the `itk::EllipseSpatialObject` as the basic shape components and we use the `itk::GroupSpatialObject` to group them together as a representation of a more complex shape. Their respective headers are included below.

```
#include "itkEllipseSpatialObject.h"
#include "itkGroupSpatialObject.h"
```

In order to generate the initial synthetic image of the ellipses, we use the `itk::SpatialObjectToImageFilter` that tests—for every pixel in the image—whether the pixel (and hence the spatial object) is *inside* or *outside* the geometric model.

```
#include <itkSpatialObjectToImageFilter.h>
```

A metric is defined to evaluate the fitness between the SpatialObject and the Image. The base class for this type of metric is the itk::ImageToSpatialObjectMetric, whose header is included below.

```
#include <itkImageToSpatialObjectMetric.h>
```

As in previous registration problems, we have to evaluate the image intensity in non-grid positions. The itk::LinearInterpolateImageFunction is used here for this purpose.

```
#include "itkLinearInterpolateImageFunction.h"
```

The SpatialObject is mapped from its own space into the image space by using a itk::Transform. In this example, we use the itk::Euler2DTransform.

```
#include "itkEuler2DTransform.h"
```

Registration is fundamentally an optimization problem. Here we include the optimizer used to search the parameter space and identify the best transformation that will map the shape model on top of the image. The optimizer used in this example is the itk::OnePlusOneEvolutionaryOptimizer that implements an evolutionary algorithm.

```
#include "itkOnePlusOneEvolutionaryOptimizer.h"
```

As in previous registration examples, it is important to track the evolution of the optimizer as it progresses through the parameter space. This is done by using the Command/Observer paradigm. The following lines of code implement the itk::Command observer that monitors the progress of the registration. The code is quite similar to what we have used in previous registration examples.

```
#include "itkCommand.h"
template < class TOptimizer >
class IterationCallback : public itk::Command
{
public:
  typedef IterationCallback   Self;
  typedef itk::Command  Superclass;
  typedef itk::SmartPointer<Self>  Pointer;
  typedef itk::SmartPointer<const Self>  ConstPointer;

  itkTypeMacro( IterationCallback, Superclass );
  itkNewMacro( Self );
```

```
/** Type defining the optimizer. */
typedef    TOptimizer     OptimizerType;

/** Method to specify the optimizer. */
void SetOptimizer( OptimizerType * optimizer )
  {
    m_Optimizer = optimizer;
    m_Optimizer->AddObserver( itk::IterationEvent(), this );
  }

/** Execute method will print data at each iteration */
void Execute(itk::Object *caller, const itk::EventObject & event)
  {
    Execute( (const itk::Object *)caller, event);
  }

void Execute(const itk::Object *, const itk::EventObject & event)
  {
    if( typeid( event ) == typeid( itk::StartEvent ) )
      {
      std::cout << std::endl << "Position              Value";
      std::cout << std::endl << std::endl;
      }
    else if( typeid( event ) == typeid( itk::IterationEvent ) )
      {
      std::cout << m_Optimizer->GetCurrentIteration() << "   ";
      std::cout << m_Optimizer->GetValue() << "   ";
      std::cout << m_Optimizer->GetCurrentPosition() << std::endl;
      }
    else if( typeid( event ) == typeid( itk::EndEvent ) )
      {
      std::cout << std::endl << std::endl;
      std::cout << "After " << m_Optimizer->GetCurrentIteration();
      std::cout << "  iterations " << std::endl;
      std::cout << "Solution is    = " << m_Optimizer->GetCurrentPosition();
      std::cout << std::endl;
      }
  }
```

This command will be invoked at every iteration of the optimizer and will print out the current combination of transform parameters.

Consider now the most critical component of this new registration approach: the metric. This component evaluates the match between the SpatialObject and the Image. The smoothness and regularity of the metric determine the difficulty of the task assigned to the optimizer. In this case, we use a very robust optimizer that should be able to find its way even in the most discontinuous cost functions. The metric to be implemented should derive from the ImageToSpatialObject-

Metric class.

The following code implements a simple metric that computes the sum of the pixels that are inside the spatial object. In fact, the metric maximum is obtained when the model and the image are aligned. The metric is templated over the type of the SpatialObject and the type of the Image.

```
template <typename TFixedImage, typename TMovingSpatialObject>
class SimpleImageToSpatialObjectMetric :
  public itk::ImageToSpatialObjectMetric<TFixedImage,TMovingSpatialObject>
{
```

The fundamental operation of the metric is its `GetValue()` method. It is in this method that the fitness value is computed. In our current example, the fitness is computed over the points of the SpatialObject. For each point, its coordinates are mapped through the transform into image space. The resulting point is used to evaluate the image and the resulting value is accumulated in a sum. Since we are not allowing scale changes, the optimal value of the sum will result when all the SpatialObject points are mapped on the white regions of the image. Note that the argument for the `GetValue()` method is the array of parameters of the transform.

```
  MeasureType    GetValue( const ParametersType & parameters ) const
    {
      double value;
      this->m_Transform->SetParameters( parameters );

      PointListType::const_iterator it = m_PointList.begin();

      typename TFixedImage::SizeType size =
        this->m_FixedImage->GetBufferedRegion().GetSize();

      itk::Index<2> index;
      itk::Index<2> start = this->m_FixedImage->GetBufferedRegion().GetIndex();

      value = 0;
      while(it != m_PointList.end())
        {
        PointType transformedPoint = this->m_Transform->TransformPoint(*it);
        this->m_FixedImage->TransformPhysicalPointToIndex(transformedPoint,index);
        if(    index[0]> start[0]
               && index[1]> start[1]
               && index[0]< static_cast< signed long >( size[0] )
               && index[1]< static_cast< signed long >( size[1] )  )
          {
          value += this->m_FixedImage->GetPixel(index);
          }
        it++;
        }
      return value;
    }
```

Having defined all the registration components we are ready to put the pieces together and implement the registration process.

First we instantiate the GroupSpatialObject and EllipseSpatialObject. These two objects are parameterized by the dimension of the space. In our current example a 2*D* instantiation is created.

```
typedef itk::GroupSpatialObject< 2 >     GroupType;
typedef itk::EllipseSpatialObject< 2 >   EllipseType;
```

The image is instantiated in the following lines using the pixel type and the space dimension. This image uses a `float` pixel type since we plan to blur it in order to increase the capture radius of the optimizer. Images of real pixel type behave better under blurring than those of integer pixel type.

```
typedef itk::Image< float, 2 >      ImageType;
```

Here is where the fun begins! In the following lines we create the EllipseSpatialObjects using their `New()` methods, and assigning the results to SmartPointers. These lines will create three ellipses.

```
EllipseType::Pointer ellipse1 = EllipseType::New();
EllipseType::Pointer ellipse2 = EllipseType::New();
EllipseType::Pointer ellipse3 = EllipseType::New();
```

Every class deriving from SpatialObject has particular parameters enabling the user to tailor its shape. In the case of the EllipseSpatialObject, `SetRadius()` is used to define the ellipse size. An additional `SetRadius(Array)` method allows the user to define the ellipse axes independently.

```
ellipse1->SetRadius(  10.0  );
ellipse2->SetRadius(  10.0  );
ellipse3->SetRadius(  10.0  );
```

The ellipses are created centered in space by default. We use the following lines of code to arrange the ellipses in a triangle. The spatial transform intrinsically associated with the object is accessed by the `GetTransform()` method. This transform can define a translation in space with the `SetOffset()` method. We take advantage of this feature to place the ellipses at particular points in space.

```
EllipseType::TransformType::OffsetType offset;
offset[ 0 ] = 100.0;
offset[ 1 ] =  40.0;

ellipse1->GetObjectToParentTransform()->SetOffset(offset);
ellipse1->ComputeObjectToWorldTransform();

offset[ 0 ] =  40.0;
offset[ 1 ] = 150.0;
ellipse2->GetObjectToParentTransform()->SetOffset(offset);
ellipse2->ComputeObjectToWorldTransform();

offset[ 0 ] = 150.0;
offset[ 1 ] = 150.0;
ellipse3->GetObjectToParentTransform()->SetOffset(offset);
ellipse3->ComputeObjectToWorldTransform();
```

Note that after a change has been made in the transform, the SpatialObject invokes the method `ComputeGlobalTransform()` in order to update its global transform. The reason for doing this is that SpatialObjects can be arranged in hierarchies. It is then possible to change the position of a set of spatial objects by moving the parent of the group.

Now we add the three EllipseSpatialObjects to a GroupSpatialObject that will be subsequently passed on to the registration method. The GroupSpatialObject facilitates the management of the three ellipses as a higher level structure representing a complex shape. Groups can be nested any number of levels in order to represent shapes with higher detail.

```
GroupType::Pointer group = GroupType::New();
group->AddSpatialObject( ellipse1 );
group->AddSpatialObject( ellipse2 );
group->AddSpatialObject( ellipse3 );
```

Having the geometric model ready, we proceed to generate the binary image representing the imprint of the space occupied by the ellipses. The SpatialObjectToImageFilter is used to that end. Note that this filter is instantiated over the spatial object used and the image type to be generated.

```
typedef itk::SpatialObjectToImageFilter< GroupType, ImageType >
  SpatialObjectToImageFilterType;
```

With the defined type, we construct a filter using the `New()` method. The newly created filter is assigned to a SmartPointer.

```
SpatialObjectToImageFilterType::Pointer imageFilter =
  SpatialObjectToImageFilterType::New();
```

The GroupSpatialObject is passed as input to the filter.

```
imageFilter->SetInput(  group  );
```

The `itk::SpatialObjectToImageFilter` acts as a resampling filter. Therefore it requires the user to define the size of the desired output image. This is specified with the `SetSize()` method.

```
ImageType::SizeType size;
size[ 0 ] = 200;
size[ 1 ] = 200;
imageFilter->SetSize( size );
```

Finally we trigger the execution of the filter by calling the `Update()` method.

```
imageFilter->Update();
```

In order to obtain a smoother metric, we blur the image using a `itk::DiscreteGaussianImageFilter`. This extends the capture radius of the metric and produce a more continuous cost function to optimize. The following lines instantiate the Gaussian filter and create one object of this type using the `New()` method.

```
typedef itk::DiscreteGaussianImageFilter< ImageType, ImageType >
  GaussianFilterType;
GaussianFilterType::Pointer   gaussianFilter =   GaussianFilterType::New();
```

The output of the SpatialObjectToImageFilter is connected as input to the DiscreteGaussianImageFilter.

```
gaussianFilter->SetInput(  imageFilter->GetOutput()  );
```

The variance of the filter is defined as a large value in order to increase the capture radius. Finally the execution of the filter is triggered using the `Update()` method.

```
const double variance = 20;
gaussianFilter->SetVariance(variance);
gaussianFilter->Update();
```

The following lines instantiate the type of the `itk::ImageToSpatialObjectRegistrationMethod` method and instantiate a registration object with the `New()` method. Note that the registration type is templated over the Image and the SpatialObject types. The spatial object in this case is the group of spatial objects.

```
typedef itk::ImageToSpatialObjectRegistrationMethod< ImageType, GroupType >
  RegistrationType;
RegistrationType::Pointer registration = RegistrationType::New();
```

Now we instantiate the metric that is templated over the image type and the spatial object type. As usual, the `New()` method is used to create an object.

```
typedef SimpleImageToSpatialObjectMetric< ImageType, GroupType > MetricType;
MetricType::Pointer metric = MetricType::New();
```

An interpolator will be needed to evaluate the image at non-grid positions. Here we instantiate a linear interpolator type.

```
typedef itk::LinearInterpolateImageFunction< ImageType, double >
  InterpolatorType;
InterpolatorType::Pointer interpolator = InterpolatorType::New();
```

The following lines instantiate the evolutionary optimizer.

```
typedef itk::OnePlusOneEvolutionaryOptimizer  OptimizerType;
OptimizerType::Pointer optimizer  = OptimizerType::New();
```

Next, we instantiate the transform class. In this case we use the Euler2DTransform that implements a rigid transform in 2*D* space.

```
typedef itk::Euler2DTransform<> TransformType;
TransformType::Pointer transform = TransformType::New();
```

Evolutionary algorithms are based on testing random variations of parameters. In order to support the computation of random values, ITK provides a family of random number generators. In this example, we use the `itk::NormalVariateGenerator` which generates values with a normal distribution.

```
itk::Statistics::NormalVariateGenerator::Pointer generator
  = itk::Statistics::NormalVariateGenerator::New();
```

The random number generator must be initialized with a seed.

```
generator->Initialize(12345);
```

The OnePlusOneEvolutionaryOptimizer is initialized by specifying the random number generator, the number of samples for the initial population and the maximum number of iterations.

```
optimizer->SetNormalVariateGenerator( generator );
optimizer->Initialize( 10 );
optimizer->SetMaximumIteration( 400 );
```

As in previous registration examples, we take care to normalize the dynamic range of the different transform parameters. In particular, the we must compensate for the ranges of the angle and translations of the Euler2DTransform. In order to achieve this goal, we provide an array of scales to the optimizer.

```
TransformType::ParametersType parametersScale;
parametersScale.set_size(3);
parametersScale[0] = 1000; // angle scale

for( unsigned int i=1; i<3; i++ )
  {
  parametersScale[i] = 2; // offset scale
  }
optimizer->SetScales( parametersScale );
```

Here we instantiate the Command object that will act as an observer of the registration method and print out parameters at each iteration. Earlier, we defined this command as a class templated over the optimizer type. Once it is created with the `New()` method, we connect the optimizer to the command.

```
typedef IterationCallback< OptimizerType >   IterationCallbackType;
IterationCallbackType::Pointer callback = IterationCallbackType::New();
callback->SetOptimizer( optimizer );
```

All the components are plugged into the ImageToSpatialObjectRegistrationMethod object. The typical `Set()` methods are used here. Note the use of the `SetMovingSpatialObject()` method for connecting the spatial object. We provide the blurred version of the original synthetic binary image as the input image.

```
registration->SetFixedImage( gaussianFilter->GetOutput() );
registration->SetMovingSpatialObject( group );
registration->SetTransform( transform );
registration->SetInterpolator( interpolator );
registration->SetOptimizer( optimizer );
registration->SetMetric( metric );
```

The initial set of transform parameters is passed to the registration method using the `SetInitialTransformParameters()` method. Note that since our original model is already registered with the synthetic image, we introduce an artificial mis-registration in order to initialize the optimization at some point away from the optimal value.

```
  TransformType::ParametersType initialParameters(
    transform->GetNumberOfParameters() );

  initialParameters[0] = 0.2;     // Angle
  initialParameters[1] = 7.0;     // Offset X
  initialParameters[2] = 6.0;     // Offset Y
  registration->SetInitialTransformParameters(initialParameters);
```

Due to the character of the metric used to evaluate the fitness between the spatial object and the image, we must tell the optimizer that we are interested in finding the maximum value of the metric. Some metrics associate low numeric values with good matching, while others associate high numeric values with good matching. The `MaximizeOn()` and `MaximizeOff()` methods allow the user to deal with both types of metrics.

```
  optimizer->MaximizeOn();
```

Finally, we trigger the execution of the registration process with the `StartRegistration()` method. We place this call in a `try/catch` block in case any exception is thrown during the process.

```
  try
    {
    registration->StartRegistration();
    }
  catch( itk::ExceptionObject & exp )
    {
    std::cerr << "Exception caught ! " << std::endl;
    std::cerr << exp << std::endl;
    }
```

The set of transform parameters resulting from the registration can be recovered with the `GetLastTransformParameters()` method. This method returns the array of transform parameters that should be interpreted according to the implementation of each transform. In our current example, the Euler2DTransform has three parameters: the rotation angle, the translation in x and the translation in y.

```
  RegistrationType::ParametersType finalParameters
    = registration->GetLastTransformParameters();

  std::cout << "Final Solution is : " << finalParameters << std::endl;
```

The results are presented in Figure 8.61. The left side shows the evolution of the angle parameter as a function of iteration numbers, while the right side shows the (x,y) translation.

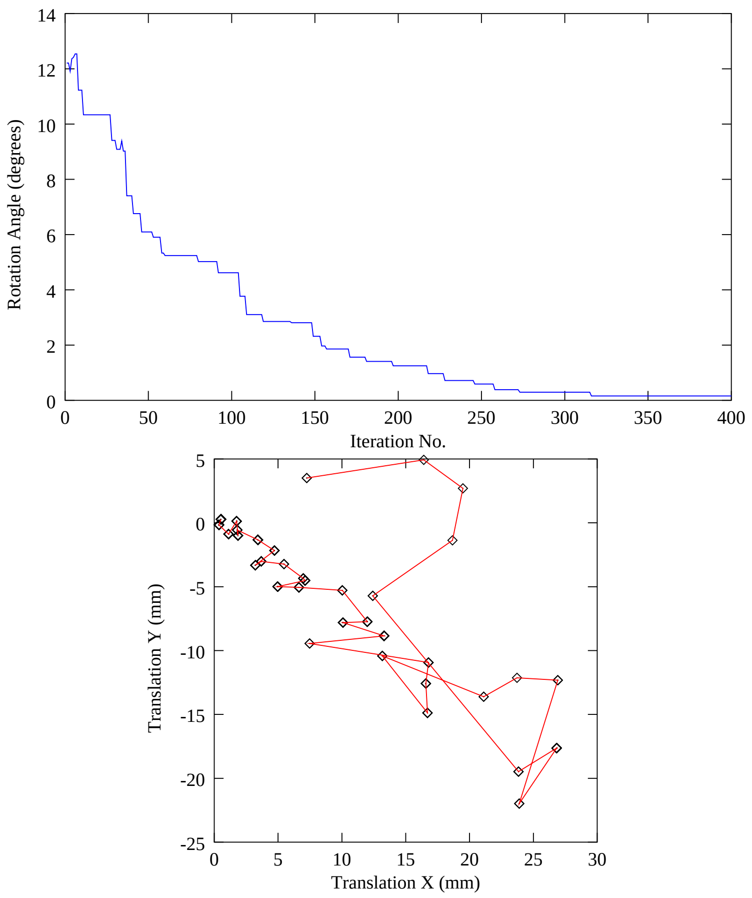


Figure 8.61: Plots of the angle and translation parameters for a registration process between an spatial object and an image.

8.17 Point Set Registration

PointSet-to-PointSet registration is a common problem in medical image analysis. It usually arises in cases where landmarks are extracted from images and are used for establishing the spatial correspondence between the images. This type of registration can be considered to be the simplest case of feature-based registration. In general terms, feature-based registration is more efficient than the intensity based method that we have presented so far. However, feature-base registration brings the new problem of identifying and extracting the features from the images, which is not a minor challenge.

The two most common scenarios in PointSet to PointSet registration are

- Two PointSets with the same number of points, and where each point in one set has a known correspondence to exactly one point in the second set.
- Two PointSets without known correspondences between the points of one set and the points of the other. In this case the PointSets may have different numbers of points.

The first case can be solved with a closed form solution when we are dealing with a Rigid or an Affine Transform [37]. This is done in ITK with the class `itk::LandmarkBasedTransformInitializer`. If we are interested in a deformable Transformation then the problem can be solved with the `itk::KernelTransform` family of classes, which includes Thin Plate Splines among others [69]. In both circumstances, the availability o f correspondences between the points make possible to apply a straight forward solution to the problem.

The classical algorithm for performing PointSet to PointSet registration is the Iterative Closest Point (ICP) algorithm. The following examples illustrate how this can be used in ITK.

The source code for this section can be found in the file
`Examples/Patented/IterativeClosestPoint1.cxx`.

This example illustrates how to perform Iterative Closest Point (ICP) registration in ITK. The main class featured in this section is the `itk::EuclideanDistancePointMetric`.

```
#include "itkTranslationTransform.h"
#include "itkEuclideanDistancePointMetric.h"
#include "itkLevenbergMarquardtOptimizer.h"
#include "itkPointSet.h"
#include "itkPointSetToPointSetRegistrationMethod.h"
#include <iostream>
#include <fstream>


int main(int argc, char * argv[] )
{
```

```
if( argc < 3 )
  {
  std::cerr << "Arguments Missing. " << std::endl;
  std::cerr
    << "Usage:  IterativeClosestPoint1   fixedPointsFile  movingPointsFile "
    << std::endl;
  return 1;
  }

const unsigned int Dimension = 2;

typedef itk::PointSet< float, Dimension >   PointSetType;

PointSetType::Pointer fixedPointSet  = PointSetType::New();
PointSetType::Pointer movingPointSet = PointSetType::New();

typedef PointSetType::PointType     PointType;

typedef PointSetType::PointsContainer  PointsContainer;

PointsContainer::Pointer fixedPointContainer  = PointsContainer::New();
PointsContainer::Pointer movingPointContainer = PointsContainer::New();

PointType fixedPoint;
PointType movingPoint;


// Read the file containing coordinates of fixed points.
std::ifstream   fixedFile;
fixedFile.open( argv[1] );
if( fixedFile.fail() )
  {
  std::cerr << "Error opening points file with name : " << std::endl;
  std::cerr << argv[1] << std::endl;
  return 2;
  }

unsigned int pointId = 0;
fixedFile >> fixedPoint;
while( !fixedFile.eof() )
  {
  fixedPointContainer->InsertElement( pointId, fixedPoint );
  fixedFile >> fixedPoint;
  pointId++;
  }
fixedPointSet->SetPoints( fixedPointContainer );
std::cout <<
  "Number of fixed Points = " <<
```

```
    fixedPointSet->GetNumberOfPoints() << std::endl;



  // Read the file containing coordinates of moving points.
  std::ifstream   movingFile;
  movingFile.open( argv[2] );
  if( movingFile.fail() )
    {
    std::cerr << "Error opening points file with name : " << std::endl;
    std::cerr << argv[2] << std::endl;
    return 2;
    }

  pointId = 0;
  movingFile >> movingPoint;
  while( !movingFile.eof() )
    {
    movingPointContainer->InsertElement( pointId, movingPoint );
    movingFile >> movingPoint;
    pointId++;
    }
  movingPointSet->SetPoints( movingPointContainer );
  std::cout << "Number of moving Points = "
    << movingPointSet->GetNumberOfPoints() << std::endl;


//-----------------------------------------------------------
// Set up  the Metric
//-----------------------------------------------------------
  typedef itk::EuclideanDistancePointMetric<
                                  PointSetType,
                                  PointSetType>
                                                  MetricType;

  typedef MetricType::TransformType               TransformBaseType;
  typedef TransformBaseType::ParametersType       ParametersType;
  typedef TransformBaseType::JacobianType         JacobianType;

  MetricType::Pointer  metric = MetricType::New();


//-----------------------------------------------------------
// Set up a Transform
//-----------------------------------------------------------

  typedef itk::TranslationTransform< double, Dimension >      TransformType;
```

```
TransformType::Pointer transform = TransformType::New();


// Optimizer Type
typedef itk::LevenbergMarquardtOptimizer OptimizerType;

OptimizerType::Pointer      optimizer     = OptimizerType::New();
optimizer->SetUseCostFunctionGradient(false);

// Registration Method
typedef itk::PointSetToPointSetRegistrationMethod<
                                          PointSetType,
                                          PointSetType >
                                                  RegistrationType;


RegistrationType::Pointer   registration  = RegistrationType::New();

// Scale the translation components of the Transform in the Optimizer
OptimizerType::ScalesType scales( transform->GetNumberOfParameters() );
scales.Fill( 0.01 );


unsigned long   numberOfIterations =  100;
double          gradientTolerance  =  1e-5;    // convergence criterion
double          valueTolerance     =  1e-5;    // convergence criterion
double          epsilonFunction    =  1e-6;   // convergence criterion


optimizer->SetScales( scales );
optimizer->SetNumberOfIterations( numberOfIterations );
optimizer->SetValueTolerance( valueTolerance );
optimizer->SetGradientTolerance( gradientTolerance );
optimizer->SetEpsilonFunction( epsilonFunction );

// Start from an Identity transform (in a normal case, the user
// can probably provide a better guess than the identity...
transform->SetIdentity();

registration->SetInitialTransformParameters( transform->GetParameters() );

//------------------------------------------------------
// Connect all the components required for Registration
//------------------------------------------------------
registration->SetMetric(        metric        );
registration->SetOptimizer(     optimizer     );
registration->SetTransform(     transform     );
registration->SetFixedPointSet( fixedPointSet );
```

```
  registration->SetMovingPointSet(   movingPointSet   );


  try
    {
    registration->StartRegistration();
    }
  catch( itk::ExceptionObject & e )
    {
    std::cout << e << std::endl;
    return EXIT_FAILURE;
    }

  std::cout << "Solution = " << transform->GetParameters() << std::endl;
```

The source code for this section can be found in the file
`Examples/Patented/IterativeClosestPoint2.cxx`.

This example illustrates how to perform Iterative Closest Point (ICP) registration in ITK using sets of 3D points.

```
#include "itkEuler3DTransform.h"
#include "itkEuclideanDistancePointMetric.h"
#include "itkLevenbergMarquardtOptimizer.h"
#include "itkPointSet.h"
#include "itkPointSetToPointSetRegistrationMethod.h"
#include <iostream>
#include <fstream>


int main(int argc, char * argv[] )
{

  if( argc < 3 )
    {
    std::cerr << "Arguments Missing. " << std::endl;
    std::cerr <<
      "Usage:  IterativeClosestPoint1   fixedPointsFile  movingPointsFile "
      << std::endl;
    return 1;
    }

  const unsigned int Dimension = 3;
```

```
typedef itk::PointSet< float, Dimension >   PointSetType;

PointSetType::Pointer fixedPointSet  = PointSetType::New();
PointSetType::Pointer movingPointSet = PointSetType::New();

typedef PointSetType::PointType     PointType;

typedef PointSetType::PointsContainer  PointsContainer;

PointsContainer::Pointer fixedPointContainer  = PointsContainer::New();
PointsContainer::Pointer movingPointContainer = PointsContainer::New();

PointType fixedPoint;
PointType movingPoint;


// Read the file containing coordinates of fixed points.
std::ifstream   fixedFile;
fixedFile.open( argv[1] );
if( fixedFile.fail() )
  {
  std::cerr << "Error opening points file with name : " << std::endl;
  std::cerr << argv[1] << std::endl;
  return 2;
  }

unsigned int pointId = 0;
fixedFile >> fixedPoint;
while( !fixedFile.eof() )
  {
  fixedPointContainer->InsertElement( pointId, fixedPoint );
  fixedFile >> fixedPoint;
  pointId++;
  }
fixedPointSet->SetPoints( fixedPointContainer );
std::cout <<
  "Number of fixed Points = " << fixedPointSet->GetNumberOfPoints()
  << std::endl;



// Read the file containing coordinates of moving points.
std::ifstream   movingFile;
movingFile.open( argv[2] );
if( movingFile.fail() )
  {
  std::cerr << "Error opening points file with name : " << std::endl;
  std::cerr << argv[2] << std::endl;
```

```
    return 2;
    }

  pointId = 0;
  movingFile >> movingPoint;
  while( !movingFile.eof() )
    {
    movingPointContainer->InsertElement( pointId, movingPoint );
    movingFile >> movingPoint;
    pointId++;
    }
  movingPointSet->SetPoints( movingPointContainer );
  std::cout <<
    "Number of moving Points = "
    << movingPointSet->GetNumberOfPoints() << std::endl;


//-----------------------------------------------------------
// Set up  the Metric
//-----------------------------------------------------------
  typedef itk::EuclideanDistancePointMetric<
                                  PointSetType,
                                  PointSetType>
                                                  MetricType;

  typedef MetricType::TransformType                TransformBaseType;
  typedef TransformBaseType::ParametersType        ParametersType;
  typedef TransformBaseType::JacobianType          JacobianType;

  MetricType::Pointer  metric = MetricType::New();


//-----------------------------------------------------------
// Set up a Transform
//-----------------------------------------------------------

  typedef itk::Euler3DTransform< double >      TransformType;

  TransformType::Pointer transform = TransformType::New();


  // Optimizer Type
  typedef itk::LevenbergMarquardtOptimizer OptimizerType;

  OptimizerType::Pointer      optimizer     = OptimizerType::New();
  optimizer->SetUseCostFunctionGradient(false);

  // Registration Method
```

```
typedef itk::PointSetToPointSetRegistrationMethod<
                                          PointSetType,
                                          PointSetType >
                                                  RegistrationType;


RegistrationType::Pointer   registration  = RegistrationType::New();

// Scale the translation components of the Transform in the Optimizer
OptimizerType::ScalesType scales( transform->GetNumberOfParameters() );

const double translationScale = 1000.0;   // dynamic range of translations
const double rotationScale    =    1.0;   // dynamic range of rotations

scales[0] = 1.0 / rotationScale;
scales[1] = 1.0 / rotationScale;
scales[2] = 1.0 / rotationScale;
scales[3] = 1.0 / translationScale;
scales[4] = 1.0 / translationScale;
scales[5] = 1.0 / translationScale;

unsigned long   numberOfIterations =  2000;
double          gradientTolerance  =  1e-4;   // convergence criterion
double          valueTolerance     =  1e-4;   // convergence criterion
double          epsilonFunction    =  1e-5;   // convergence criterion


optimizer->SetScales( scales );
optimizer->SetNumberOfIterations( numberOfIterations );
optimizer->SetValueTolerance( valueTolerance );
optimizer->SetGradientTolerance( gradientTolerance );
optimizer->SetEpsilonFunction( epsilonFunction );

// Start from an Identity transform (in a normal case, the user
// can probably provide a better guess than the identity...
transform->SetIdentity();

registration->SetInitialTransformParameters( transform->GetParameters() );

//------------------------------------------------------
// Connect all the components required for Registration
//------------------------------------------------------
registration->SetMetric(        metric        );
registration->SetOptimizer(     optimizer     );
registration->SetTransform(     transform     );
registration->SetFixedPointSet( fixedPointSet );
registration->SetMovingPointSet(   movingPointSet   );
```

```
  try
    {
    registration->StartRegistration();
    }
  catch( itk::ExceptionObject & e )
    {
    std::cout << e << std::endl;
    return EXIT_FAILURE;
    }

  std::cout << "Solution = " << transform->GetParameters() << std::endl;
```

The source code for this section can be found in the file
`Examples/Patented/IterativeClosestPoint3.cxx`.

This example illustrates how to perform Iterative Closest Point (ICP) registration in ITK using a DistanceMap in order to increase the performance. There is of course a trade-off between the time needed for computing the DistanceMap and the time saving obtained by its repeated use during the iterative computation of the point to point distances. It is then necessary in practice to ponder both factors.

`itk::EuclideanDistancePointMetric`.

```
#include "itkTranslationTransform.h"
#include "itkEuclideanDistancePointMetric.h"
#include "itkLevenbergMarquardtOptimizer.h"
#include "itkPointSet.h"
#include "itkPointSetToPointSetRegistrationMethod.h"
#include "itkDanielssonDistanceMapImageFilter.h"
#include "itkPointSetToImageFilter.h"
#include <iostream>
#include <fstream>


int main(int argc, char * argv[] )
{

  if( argc < 3 )
    {
    std::cerr << "Arguments Missing. " << std::endl;
    std::cerr <<
      "Usage:  IterativeClosestPoint3   fixedPointsFile  movingPointsFile "
      << std::endl;
    return 1;
    }

  const unsigned int Dimension = 2;

  typedef itk::PointSet< float, Dimension >   PointSetType;

  PointSetType::Pointer fixedPointSet  = PointSetType::New();
  PointSetType::Pointer movingPointSet = PointSetType::New();

  typedef PointSetType::PointType     PointType;

  typedef PointSetType::PointsContainer  PointsContainer;

  PointsContainer::Pointer fixedPointContainer  = PointsContainer::New();
  PointsContainer::Pointer movingPointContainer = PointsContainer::New();

  PointType fixedPoint;
  PointType movingPoint;


  // Read the file containing coordinates of fixed points.
  std::ifstream   fixedFile;
  fixedFile.open( argv[1] );
  if( fixedFile.fail() )
    {
    std::cerr << "Error opening points file with name : " << std::endl;
    std::cerr << argv[1] << std::endl;
    return 2;
    }

  unsigned int pointId = 0;
  fixedFile >> fixedPoint;
  while( !fixedFile.eof() )
    {
    fixedPointContainer->InsertElement( pointId, fixedPoint );
    fixedFile >> fixedPoint;
    pointId++;
    }
  fixedPointSet->SetPoints( fixedPointContainer );
  std::cout << "Number of fixed Points = "
        << fixedPointSet->GetNumberOfPoints() << std::endl;



  // Read the file containing coordinates of moving points.
  std::ifstream   movingFile;
  movingFile.open( argv[2] );
  if( movingFile.fail() )
    {
    std::cerr << "Error opening points file with name : " << std::endl;
    std::cerr << argv[2] << std::endl;
    return 2;
    }

  pointId = 0;
  movingFile >> movingPoint;
  while( !movingFile.eof() )
    {
    movingPointContainer->InsertElement( pointId, movingPoint );
    movingFile >> movingPoint;
    pointId++;
    }
  movingPointSet->SetPoints( movingPointContainer );
  std::cout << "Number of moving Points = "
      << movingPointSet->GetNumberOfPoints() << std::endl;


//-----------------------------------------------------------
// Set up  the Metric
//-----------------------------------------------------------
  typedef itk::EuclideanDistancePointMetric<
                                  PointSetType,
                                  PointSetType>
                                                  MetricType;

  typedef MetricType::TransformType                 TransformBaseType;
  typedef TransformBaseType::ParametersType         ParametersType;
  typedef TransformBaseType::JacobianType           JacobianType;

  MetricType::Pointer  metric = MetricType::New();


//-----------------------------------------------------------
// Set up a Transform
//-----------------------------------------------------------

  typedef itk::TranslationTransform< double, Dimension >      TransformType;

  TransformType::Pointer transform = TransformType::New();


  // Optimizer Type
  typedef itk::LevenbergMarquardtOptimizer OptimizerType;

  OptimizerType::Pointer      optimizer     = OptimizerType::New();
```

```
optimizer->SetUseCostFunctionGradient(false);

// Registration Method
typedef itk::PointSetToPointSetRegistrationMethod<
                                          PointSetType,
                                          PointSetType >
                                                  RegistrationType;


RegistrationType::Pointer   registration  = RegistrationType::New();

// Scale the translation components of the Transform in the Optimizer
OptimizerType::ScalesType scales( transform->GetNumberOfParameters() );
scales.Fill( 0.01 );


unsigned long   numberOfIterations =  100;
double          gradientTolerance  =  1e-5;    // convergence criterion
double          valueTolerance     =  1e-5;    // convergence criterion
double          epsilonFunction    =  1e-6;   // convergence criterion


optimizer->SetScales( scales );
optimizer->SetNumberOfIterations( numberOfIterations );
optimizer->SetValueTolerance( valueTolerance );
optimizer->SetGradientTolerance( gradientTolerance );
optimizer->SetEpsilonFunction( epsilonFunction );

// Start from an Identity transform (in a normal case, the user
// can probably provide a better guess than the identity...
transform->SetIdentity();

registration->SetInitialTransformParameters( transform->GetParameters() );

//------------------------------------------------------
// Connect all the components required for Registration
//------------------------------------------------------
registration->SetMetric(        metric        );
registration->SetOptimizer(     optimizer     );
registration->SetTransform(     transform     );
registration->SetFixedPointSet( fixedPointSet );
registration->SetMovingPointSet(   movingPointSet   );


//------------------------------------------------------
// Prepare the Distance Map in order to accelerate
// distance computations.
//------------------------------------------------------
```

```
//
//  First map the Fixed Points into a binary image.
//  This is needed because the DanielssonDistance
//  filter expects an image as input.
//
//-------------------------------------------------
typedef itk::Image< unsigned char,  Dimension >  BinaryImageType;

typedef itk::PointSetToImageFilter<
                          PointSetType,
                          BinaryImageType> PointsToImageFilterType;

PointsToImageFilterType::Pointer
                pointsToImageFilter = PointsToImageFilterType::New();

pointsToImageFilter->SetInput( fixedPointSet );

BinaryImageType::SpacingType spacing;
spacing.Fill( 1.0 );

BinaryImageType::PointType origin;
origin.Fill( 0.0 );

pointsToImageFilter->SetSpacing( spacing );
pointsToImageFilter->SetOrigin( origin   );

pointsToImageFilter->Update();

BinaryImageType::Pointer binaryImage = pointsToImageFilter->GetOutput();


typedef itk::Image< unsigned short, Dimension >  DistanceImageType;

typedef itk::DanielssonDistanceMapImageFilter<
                                        BinaryImageType,
                                        DistanceImageType> DistanceFilterType;

DistanceFilterType::Pointer distanceFilter = DistanceFilterType::New();

distanceFilter->SetInput( binaryImage );

distanceFilter->Update();

metric->SetDistanceMap( distanceFilter->GetOutput() );


try
  {
  registration->StartRegistration();
  }
catch( itk::ExceptionObject & e )
  {
  std::cout << e << std::endl;
  return EXIT_FAILURE;
  }

std::cout << "Solution = " << transform->GetParameters() << std::endl;
```

CHAPTER

NINE

Segmentation

Segmentation of medical images is a challenging task. A myriad of different methods have been proposed and implemented in recent years. In spite of the huge effort invested in this problem, there is no single approach that can generally solve the problem of segmentation for the large variety of image modalities existing today.

The most effective segmentation algorithms are obtained by carefully customizing combinations of components. The parameters of these components are tuned for the characteristics of the image modality used as input and the features of the anatomical structure to be segmented.

The Insight Toolkit provides a basic set of algorithms that can be used to develop and customize a full segmentation application. Some of the most commonly used segmentation components are described in the following sections.

9.1 Region Growing

Region growing algorithms have proven to be an effective approach for image segmentation. The basic approach of a region growing algorithm is to start from a seed region (typically one or more pixels) that are considered to be inside the object to be segmented. The pixels neighboring this region are evaluated to determine if they should also be considered part of the object. If so, they are added to the region and the process continues as long as new pixels are added to the region. Region growing algorithms vary depending on the criteria used to decide whether a pixel should be included in the region or not, the type connectivity used to determine neighbors, and the strategy used to visit neighboring pixels.

Several implementations of region growing are available in ITK. This section describes some of the most commonly used.

9.1.1 Connected Threshold

A simple criterion for including pixels in a growing region is to evaluate intensity value inside a specific interval.

The source code for this section can be found in the file
`Examples/Segmentation/ConnectedThresholdImageFilter.cxx`.

The following example illustrates the use of the `itk::ConnectedThresholdImageFilter`. This filter uses the flood fill iterator. Most of the algorithmic complexity of a region growing method comes from visiting neighboring pixels. The flood fill iterator assumes this responsibility and greatly simplifies the implementation of the region growing algorithm. Thus the algorithm is left to establish a criterion to decide whether a particular pixel should be included in the current region or not.

The criterion used by the ConnectedThresholdImageFilter is based on an interval of intensity values provided by the user. Values of lower and upper threshold should be provided. The region growing algorithm includes those pixels whose intensities are inside the interval.

$$I(\mathbf{X}) \in [\text{lower}, \text{upper}] \tag{9.1}$$

Let's look at the minimal code required to use this algorithm. First, the following header defining the ConnectedThresholdImageFilter class must be included.

```
#include "itkConnectedThresholdImageFilter.h"
```

Noise present in the image can reduce the capacity of this filter to grow large regions. When faced with noisy images, it is usually convenient to pre-process the image by using an edge-preserving smoothing filter. Any of the filters discussed in Section 6.7.3 could be used to this end. In this particular example we use the `itk::CurvatureFlowImageFilter`, hence we need to include its header file.

```
#include "itkCurvatureFlowImageFilter.h"
```

We declare the image type based on a particular pixel type and dimension. In this case the `float` type is used for the pixels due to the requirements of the smoothing filter.

```
  typedef   float           InternalPixelType;
  const     unsigned int    Dimension = 2;
  typedef itk::Image< InternalPixelType, Dimension >  InternalImageType;
```

The smoothing filter is instantiated using the image type as a template parameter.

```
  typedef itk::CurvatureFlowImageFilter< InternalImageType, InternalImageType >
    CurvatureFlowImageFilterType;
```

Then the filter is created by invoking the `New()` method and assigning the result to a `itk::SmartPointer`.

```
CurvatureFlowImageFilterType::Pointer smoothing =
                        CurvatureFlowImageFilterType::New();
```

We now declare the type of the region growing filter. In this case it is the ConnectedThresholdImageFilter.

```
typedef itk::ConnectedThresholdImageFilter< InternalImageType,
                                   InternalImageType > ConnectedFilterType;
```

Then we construct one filter of this class using the `New()` method.

```
ConnectedFilterType::Pointer connectedThreshold = ConnectedFilterType::New();
```

Now it is time to connect a simple, linear pipeline. A file reader is added at the beginning of the pipeline and a cast filter and writer are added at the end. The cast filter is required to convert `float` pixel types to integer types since only a few image file formats support `float` types.

```
smoothing->SetInput( reader->GetOutput() );
connectedThreshold->SetInput( smoothing->GetOutput() );
caster->SetInput( connectedThreshold->GetOutput() );
writer->SetInput( caster->GetOutput() );
```

The CurvatureFlowImageFilter requires a couple of parameters to be defined. The following are typical values for $2D$ images. However they may have to be adjusted depending on the amount of noise present in the input image.

```
smoothing->SetNumberOfIterations( 5 );
smoothing->SetTimeStep( 0.125 );
```

The ConnectedThresholdImageFilter has two main parameters to be defined. They are the lower and upper thresholds of the interval in which intensity values should fall in order to be included in the region. Setting these two values too close will not allow enough flexibility for the region to grow. Setting them too far apart will result in a region that engulfs the image.

```
connectedThreshold->SetLower(  lowerThreshold  );
connectedThreshold->SetUpper(  upperThreshold  );
```

The output of this filter is a binary image with zero-value pixels everywhere except on the extracted region. The intensity value set inside the region is selected with the method `SetReplaceValue()`

Structure	Seed Index	Lower	Upper	Output Image
White matter	$(60,116)$	150	180	Second from left in Figure 9.1
Ventricle	$(81,112)$	210	250	Third from left in Figure 9.1
Gray matter	$(107,69)$	180	210	Fourth from left in Figure 9.1

Table 9.1: Parameters used for segmenting some brain structures shown in Figure 9.1 with the filter itk::ConnectedThresholdImageFilter.

```
connectedThreshold->SetReplaceValue( 255 );
```

The initialization of the algorithm requires the user to provide a seed point. It is convenient to select this point to be placed in a *typical* region of the anatomical structure to be segmented. The seed is passed in the form of a itk::Index to the SetSeed() method.

```
connectedThreshold->SetSeed( index );
```

The invocation of the Update() method on the writer triggers the execution of the pipeline. It is usually wise to put update calls in a try/catch block in case errors occur and exceptions are thrown.

```
try
  {
  writer->Update();
  }
catch( itk::ExceptionObject & excep )
  {
  std::cerr << "Exception caught !" << std::endl;
  std::cerr << excep << std::endl;
  }
```

Let's run this example using as input the image BrainProtonDensitySlice.png provided in the directory Examples/Data. We can easily segment the major anatomical structures by providing seeds in the appropriate locations and defining values for the lower and upper thresholds. Figure 9.1 illustrates several examples of segmentation. The parameters used are presented in Table 9.1.

Notice that the gray matter is not being completely segmented. This illustrates the vulnerability of the region growing methods when the anatomical structures to be segmented do not have a homogeneous statistical distribution over the image space. You may want to experiment with different values of the lower and upper thresholds to verify how the accepted region will extend.

Another option for segmenting regions is to take advantage of the functionality provided by the ConnectedThresholdImageFilter for managing multiple seeds. The seeds can be passed one by one to the filter using the AddSeed() method. You could imagine a user interface in which

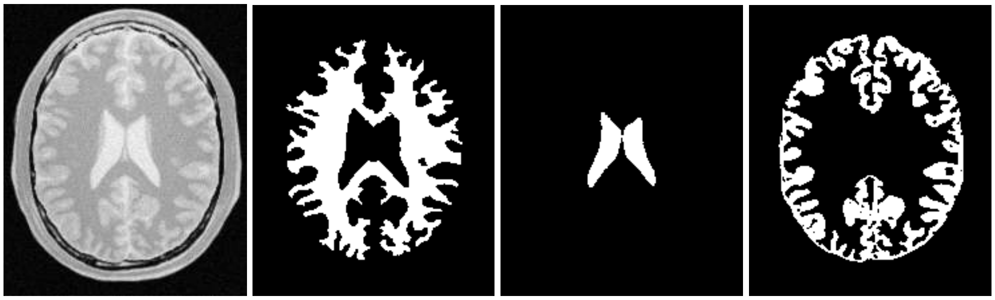


Figure 9.1: Segmentation results for the ConnectedThreshold filter for various seed points.

an operator clicks on multiple points of the object to be segmented and each selected point is passed as a seed to this filter.

9.1.2 Otsu Segmentation

Another criterion for classifying pixels is to minimize the error of misclassification. The goal is to find a threshold that classifies the image into two clusters such that we minimize the area under the histogram for one cluster that lies on the other cluster's side of the threshold. This is equivalent to minimizing the within class variance or equivalently maximizing the between class variance.

The source code for this section can be found in the file
`Examples/Filtering/OtsuThresholdImageFilter.cxx`.

This example illustrates how to use the `itk::OtsuThresholdImageFilter`.

```
#include "itkOtsuThresholdImageFilter.h"
```

The next step is to decide which pixel types to use for the input and output images.

```
  typedef  unsigned char  InputPixelType;
  typedef  unsigned char  OutputPixelType;
```

The input and output image types are now defined using their respective pixel types and dimensions.

```
  typedef itk::Image< InputPixelType,  2 >   InputImageType;
  typedef itk::Image< OutputPixelType, 2 >   OutputImageType;
```

The filter type can be instantiated using the input and output image types defined above.

```
typedef itk::OtsuThresholdImageFilter<
             InputImageType, OutputImageType >  FilterType;
```

An itk::ImageFileReader class is also instantiated in order to read image data from a file. (See Section 7 on page 263 for more information about reading and writing data.)

```
typedef itk::ImageFileReader< InputImageType >  ReaderType;
```

An itk::ImageFileWriter is instantiated in order to write the output image to a file.

```
typedef itk::ImageFileWriter< InputImageType >  WriterType;
```

Both the filter and the reader are created by invoking their `New()` methods and assigning the result to itk::SmartPointers.

```
ReaderType::Pointer reader = ReaderType::New();
FilterType::Pointer filter = FilterType::New();
```

The image obtained with the reader is passed as input to the OtsuThresholdImageFilter.

```
filter->SetInput( reader->GetOutput() );
```

The method `SetOutsideValue()` defines the intensity value to be assigned to those pixels whose intensities are outside the range defined by the lower and upper thresholds. The method `SetInsideValue()` defines the intensity value to be assigned to pixels with intensities falling inside the threshold range.

```
filter->SetOutsideValue( outsideValue );
filter->SetInsideValue(  insideValue  );
```

The method `SetNumberOfHistogramBins()` defines the number of bins to be used for computing the histogram. This histogram will be used internally in order to compute the Otsu threshold.

```
filter->SetNumberOfHistogramBins( 128 );
```

The execution of the filter is triggered by invoking the `Update()` method. If the filter's output has been passed as input to subsequent filters, the `Update()` call on any posterior filters in the pipeline will indirectly trigger the update of this filter.

```
filter->Update();
```

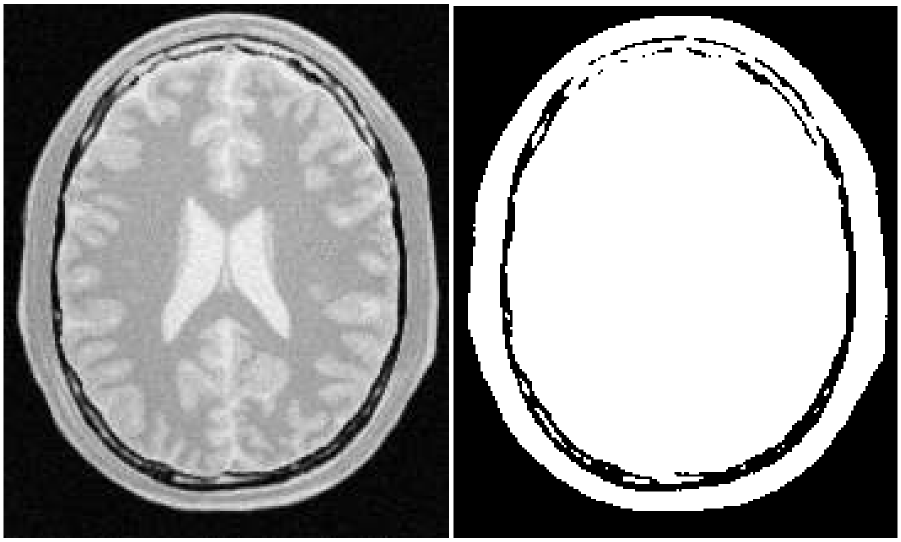


Figure 9.2: Effect of the OtsuThresholdImageFilter on a slice from a MRI proton density image of the brain.

We print out here the Threshold value that was computed internally by the filter. For this we invoke the `GetThreshold` method.

```
int threshold = filter->GetThreshold();
std::cout << "Threshold = " << threshold << std::endl;
```

Figure 9.2 illustrates the effect of this filter on a MRI proton density image of the brain. This figure shows the limitations of this filter for performing segmentation by itself. These limitations are particularly noticeable in noisy images and in images lacking spatial uniformity as is the case with MRI due to field bias.

The following classes provide similar functionality:

- `itk::ThresholdImageFilter`

The source code for this section can be found in the file
`Examples/Filtering/OtsuMultipleThresholdImageFilter.cxx`.

This example illustrates how to use the `itk::OtsuMultipleThresholdsCalculator`.

```
#include "itkOtsuMultipleThresholdsCalculator.h"
```

OtsuMultipleThresholdsCalculator calculates thresholds for a give histogram so as to maximize the between-class variance. We use ScalarImageToHistogramGenerator to generate histograms

```
typedef itk::Statistics::ScalarImageToHistogramGenerator< InputImageType >
  ScalarImageToHistogramGeneratorType;
typedef itk::OtsuMultipleThresholdsCalculator<
  ScalarImageToHistogramGeneratorType::HistogramType >   CalculatorType;
```

Once thresholds are computed we will use BinaryThresholdImageFilter to segment the input image into segments.

```
typedef itk::BinaryThresholdImageFilter< InputImageType, OutputImageType >
  FilterType;


ScalarImageToHistogramGeneratorType::Pointer scalarImageToHistogramGenerator =
  ScalarImageToHistogramGeneratorType::New();
CalculatorType::Pointer calculator = CalculatorType::New();
FilterType::Pointer filter = FilterType::New();


scalarImageToHistogramGenerator->SetNumberOfBins(128);
calculator->SetNumberOfThresholds(atoi(argv[4]));
```

The pipeline will look as follows:

```
scalarImageToHistogramGenerator->SetInput(reader->GetOutput());
calculator->SetInputHistogram(scalarImageToHistogramGenerator->GetOutput());
filter->SetInput( reader->GetOutput() );
writer->SetInput(filter->GetOutput());
```

Thresholds are obtained using the `GetOutput` method

```
const CalculatorType::OutputType &thresholdVector = calculator->GetOutput();
CalculatorType::OutputType::const_iterator itNum = thresholdVector.begin();


for(; itNum < thresholdVector.end(); itNum++)
  {
  std::cout << "OtsuThreshold["
    << (int)(itNum - thresholdVector.begin())
    << "] = " <<
    static_cast<itk::NumericTraits<CalculatorType::MeasurementType>::PrintType>
    (*itNum) << std::endl;


  }
```

9.1.3 Neighborhood Connected

The source code for this section can be found in the file
`Examples/Segmentation/NeighborhoodConnectedImageFilter.cxx`.

The following example illustrates the use of the `itk::NeighborhoodConnectedImageFilter`. This filter is a close variant of the `itk::ConnectedThresholdImageFilter`. On one hand, the ConnectedThresholdImageFilter accepts a pixel in the region if its intensity is in the interval defined by two user-provided threshold values. The NeighborhoodConnectedImageFilter, on the other hand, will only accept a pixel if **all** its neighbors have intensities that fit in the interval. The size of the neighborhood to be considered around each pixel is defined by a user-provided integer radius.

The reason for considering the neighborhood intensities instead of only the current pixel intensity is that small structures are less likely to be accepted in the region. The operation of this filter is equivalent to applying the ConnectedThresholdImageFilter followed by mathematical morphology erosion using a structuring element of the same shape as the neighborhood provided to the NeighborhoodConnectedImageFilter.

```
#include "itkNeighborhoodConnectedImageFilter.h"
```

The `itk::CurvatureFlowImageFilter` is used here to smooth the image while preserving edges.

```
#include "itkCurvatureFlowImageFilter.h"
```

We now define the image type using a particular pixel type and image dimension. In this case the `float` type is used for the pixels due to the requirements of the smoothing filter.

```
  typedef   float           InternalPixelType;
  const     unsigned int    Dimension = 2;
  typedef itk::Image< InternalPixelType, Dimension >  InternalImageType;
```

The smoothing filter type is instantiated using the image type as a template parameter.

```
  typedef   itk::CurvatureFlowImageFilter<InternalImageType, InternalImageType>
    CurvatureFlowImageFilterType;
```

Then, the filter is created by invoking the `New()` method and assigning the result to a `itk::SmartPointer`.

```
  CurvatureFlowImageFilterType::Pointer smoothing =
                       CurvatureFlowImageFilterType::New();
```

We now declare the type of the region growing filter. In this case it is the NeighborhoodConnectedImageFilter.

```
  typedef itk::NeighborhoodConnectedImageFilter<InternalImageType,
                                    InternalImageType > ConnectedFilterType;
```

One filter of this class is constructed using the `New()` method.

```
  ConnectedFilterType::Pointer neighborhoodConnected = ConnectedFilterType::New();
```

Now it is time to create a simple, linear data processing pipeline. A file reader is added at the beginning of the pipeline and a cast filter and writer are added at the end. The cast filter is required to convert `float` pixel types to integer types since only a few image file formats support `float` types.

```
  smoothing->SetInput( reader->GetOutput() );
  neighborhoodConnected->SetInput( smoothing->GetOutput() );
  caster->SetInput( neighborhoodConnected->GetOutput() );
  writer->SetInput( caster->GetOutput() );
```

The CurvatureFlowImageFilter requires a couple of parameters to be defined. The following are typical values for $2D$ images. However they may have to be adjusted depending on the amount of noise present in the input image.

```
  smoothing->SetNumberOfIterations( 5 );
  smoothing->SetTimeStep( 0.125 );
```

The NeighborhoodConnectedImageFilter requires that two main parameters are specified. They are the lower and upper thresholds of the interval in which intensity values must fall to be included in the region. Setting these two values too close will not allow enough flexibility for the region to grow. Setting them too far apart will result in a region that engulfs the image.

```
  neighborhoodConnected->SetLower(  lowerThreshold  );
  neighborhoodConnected->SetUpper(  upperThreshold  );
```

Here, we add the crucial parameter that defines the neighborhood size used to determine whether a pixel lies in the region. The larger the neighborhood, the more stable this filter will be against noise in the input image, but also the longer the computing time will be. Here we select a filter of radius 2 along each dimension. This results in a neighborhood of 5×5 pixels.

```
  InternalImageType::SizeType   radius;

  radius[0] = 2;   // two pixels along X
  radius[1] = 2;   // two pixels along Y

  neighborhoodConnected->SetRadius( radius );
```

As in the ConnectedThresholdImageFilter we must now provide the intensity value to be used for the output pixels accepted in the region and at least one seed point to define the initial region.

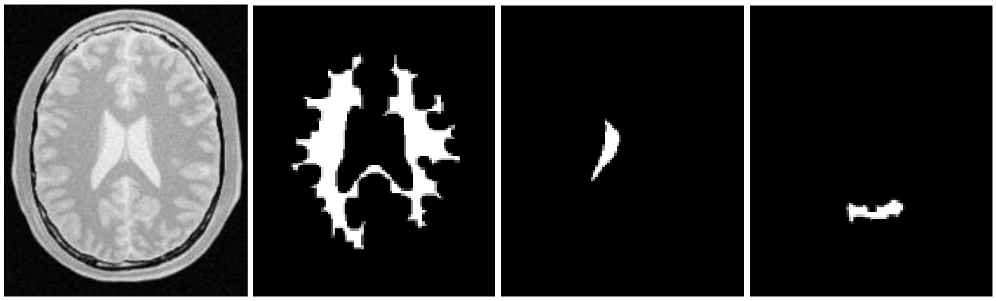


Figure 9.3: Segmentation results of the NeighborhoodConnectedImageFilter for various seed points.

```
neighborhoodConnected->SetSeed( index );
neighborhoodConnected->SetReplaceValue( 255 );
```

The invocation of the `Update()` method on the writer triggers the execution of the pipeline. It is usually wise to put update calls in a `try/catch` block in case errors occur and exceptions are thrown.

```
try
  {
  writer->Update();
  }
catch( itk::ExceptionObject & excep )
  {
  std::cerr << "Exception caught !" << std::endl;
  std::cerr << excep << std::endl;
  }
```

Now we'll run this example using the image `BrainProtonDensitySlice.png` as input available from the directory `Examples/Data`. We can easily segment the major anatomical structures by providing seeds in the appropriate locations and defining values for the lower and upper thresholds. For example

Structure	Seed Index	Lower	Upper	Output Image
White matter	$(60,116)$	150	180	Second from left in Figure 9.3
Ventricle	$(81,112)$	210	250	Third from left in Figure 9.3
Gray matter	$(107,69)$	180	210	Fourth from left in Figure 9.3

As with the ConnectedThresholdImageFilter, several seeds could be provided to the filter by using the `AddSeed()` method. Compare the output of Figure 9.3 with those of Figure 9.1 produced by the ConnectedThresholdImageFilter. You may want to play with the value of the neighborhood radius and see how it affect the smoothness of the segmented object borders, the size of the segmented region and how much that costs in computing time.

9.1.4 Confidence Connected

The source code for this section can be found in the file
`Examples/Segmentation/ConfidenceConnected.cxx`.

The following example illustrates the use of the `itk::ConfidenceConnectedImageFilter`. The criterion used by the ConfidenceConnectedImageFilter is based on simple statistics of the current region. First, the algorithm computes the mean and standard deviation of intensity values for all the pixels currently included in the region. A user-provided factor is used to multiply the standard deviation and define a range around the mean. Neighbor pixels whose intensity values fall inside the range are accepted and included in the region. When no more neighbor pixels are found that satisfy the criterion, the algorithm is considered to have finished its first iteration. At that point, the mean and standard deviation of the intensity levels are recomputed using all the pixels currently included in the region. This mean and standard deviation defines a new intensity range that is used to visit current region neighbors and evaluate whether their intensity falls inside the range. This iterative process is repeated until no more pixels are added or the maximum number of iterations is reached. The following equation illustrates the inclusion criterion used by this filter,

$$I(\mathbf{X}) \in [m - f\sigma, m + f\sigma] \tag{9.2}$$

where m and σ are the mean and standard deviation of the region intensities, f is a factor defined by the user, $I()$ is the image and $\mathbf{X}$ is the position of the particular neighbor pixel being considered for inclusion in the region.

Let's look at the minimal code required to use this algorithm. First, the following header defining the `itk::ConfidenceConnectedImageFilter` class must be included.

```
#include "itkConfidenceConnectedImageFilter.h"
```

Noise present in the image can reduce the capacity of this filter to grow large regions. When faced with noisy images, it is usually convenient to pre-process the image by using an edge-preserving smoothing filter. Any of the filters discussed in Section 6.7.3 can be used to this end. In this particular example we use the `itk::CurvatureFlowImageFilter`, hence we need to include its header file.

```
#include "itkCurvatureFlowImageFilter.h"
```

We now define the image type using a pixel type and a particular dimension. In this case the `float` type is used for the pixels due to the requirements of the smoothing filter.

```
  typedef   float           InternalPixelType;
  const     unsigned int    Dimension = 2;
  typedef itk::Image< InternalPixelType, Dimension >  InternalImageType;
```

The smoothing filter type is instantiated using the image type as a template parameter.

```
typedef itk::CurvatureFlowImageFilter< InternalImageType, InternalImageType >
  CurvatureFlowImageFilterType;
```

Next the filter is created by invoking the `New()` method and assigning the result to a `itk::SmartPointer`.

```
CurvatureFlowImageFilterType::Pointer smoothing =
                       CurvatureFlowImageFilterType::New();
```

We now declare the type of the region growing filter. In this case it is the ConfidenceConnectedImageFilter.

```
typedef itk::ConfidenceConnectedImageFilter<InternalImageType, InternalImageType>
  ConnectedFilterType;
```

Then, we construct one filter of this class using the `New()` method.

```
ConnectedFilterType::Pointer confidenceConnected = ConnectedFilterType::New();
```

Now it is time to create a simple, linear pipeline. A file reader is added at the beginning of the pipeline and a cast filter and writer are added at the end. The cast filter is required here to convert `float` pixel types to integer types since only a few image file formats support `float` types.

```
smoothing->SetInput( reader->GetOutput() );
confidenceConnected->SetInput( smoothing->GetOutput() );
caster->SetInput( confidenceConnected->GetOutput() );
writer->SetInput( caster->GetOutput() );
```

The CurvatureFlowImageFilter requires defining two parameters. The following are typical values for $2D$ images. However they may have to be adjusted depending on the amount of noise present in the input image.

```
smoothing->SetNumberOfIterations( 5 );
smoothing->SetTimeStep( 0.125 );
```

The ConfidenceConnectedImageFilter requires defining two parameters. First, the factor f that the defines how large the range of intensities will be. Small values of the multiplier will restrict the inclusion of pixels to those having very similar intensities to those in the current region. Larger values of the multiplier will relax the accepting condition and will result in more generous growth of the region. Values that are too large will cause the region to grow into neighboring regions that may actually belong to separate anatomical structures.

```
confidenceConnected->SetMultiplier( 2.5 );
```

The number of iterations is specified based on the homogeneity of the intensities of the anatomical structure to be segmented. Highly homogeneous regions may only require a couple of iterations. Regions with ramp effects, like MRI images with inhomogeneous fields, may require more iterations. In practice, it seems to be more important to carefully select the multiplier factor than the number of iterations. However, keep in mind that there is no reason to assume that this algorithm should converge to a stable region. It is possible that by letting the algorithm run for more iterations the region will end up engulfing the entire image.

```
confidenceConnected->SetNumberOfIterations( 5 );
```

The output of this filter is a binary image with zero-value pixels everywhere except on the extracted region. The intensity value to be set inside the region is selected with the method `SetReplaceValue()`

```
confidenceConnected->SetReplaceValue( 255 );
```

The initialization of the algorithm requires the user to provide a seed point. It is convenient to select this point to be placed in a *typical* region of the anatomical structure to be segmented. A small neighborhood around the seed point will be used to compute the initial mean and standard deviation for the inclusion criterion. The seed is passed in the form of a `itk::Index` to the `SetSeed()` method.

```
confidenceConnected->SetSeed( index );
```

The size of the initial neighborhood around the seed is defined with the method `SetInitialNeighborhoodRadius()`. The neighborhood will be defined as an N-dimensional rectangular region with $2r+1$ pixels on the side, where r is the value passed as initial neighborhood radius.

```
confidenceConnected->SetInitialNeighborhoodRadius( 2 );
```

The invocation of the `Update()` method on the writer triggers the execution of the pipeline. It is recommended to place update calls in a `try/catch` block in case errors occur and exceptions are thrown.

```
try
  {
  writer->Update();
  }
catch( itk::ExceptionObject & excep )
  {
```

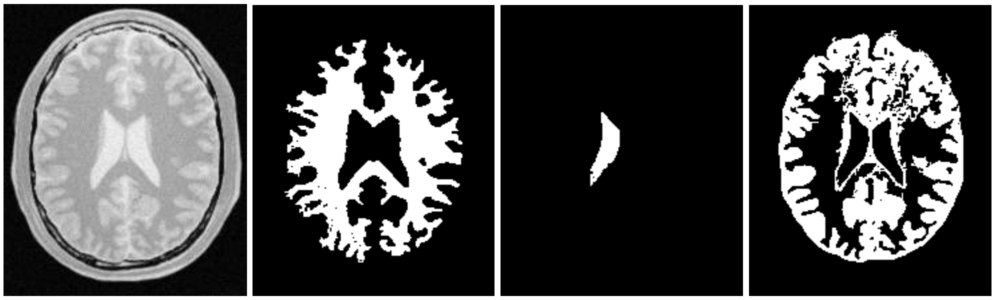


Figure 9.4: Segmentation results for the ConfidenceConnected filter for various seed points.

```
std::cerr << "Exception caught !" << std::endl;
std::cerr << excep << std::endl;
}
```

Let's now run this example using as input the image `BrainProtonDensitySlice.png` provided in the directory `Examples/Data`. We can easily segment the major anatomical structures by providing seeds in the appropriate locations. For example

Structure	Seed Index	Output Image
White matter	$(60,116)$	Second from left in Figure 9.4
Ventricle	$(81,112)$	Third from left in Figure 9.4
Gray matter	$(107,69)$	Fourth from left in Figure 9.4

Note that the gray matter is not being completely segmented. This illustrates the vulnerability of the region growing methods when the anatomical structures to be segmented do not have a homogeneous statistical distribution over the image space. You may want to experiment with different numbers of iterations to verify how the accepted region will extend.

Application of the Confidence Connected filter on the Brain Web Data

This section shows some results obtained by applying the Confidence Connected filter on the BrainWeb database. The filter was applied on a $181 \times 217 \times 181$ crosssection of the *brainweb165a10f17* dataset. The data is a MR T1 acquisition, with an intensity non-uniformity of 20% and a slice thickness 1mm. The dataset may be obtained from `http://www.bic.mni.mcgill.ca/brainweb/` or `ftp://public.kitware.com/pub/itk/Data/BrainWeb/`

The previous code was used in this example replacing the image dimension by 3. Gradient Anistropic diffusion was applied to smooth the image. The filter used 2 iterations, a time step of 0.05 and a conductance value of 3. The smoothed volume was then segmented using the Confidence Connected approach. Five seed points were used at coordinate locations (118,85,92),

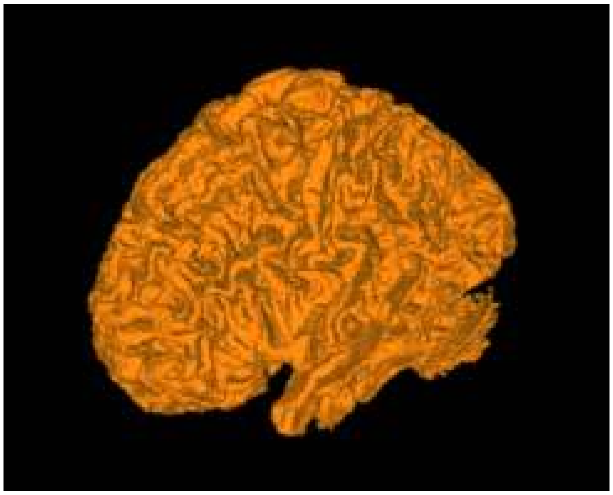


Figure 9.5: White matter segmented using Confidence Connected region growing.

(63,87,94), (63,157,90), (111,188,90), (111,50,88). The ConfidenceConnnected filter used the parameters, a neighborhood radius of 2, 5 iterations and an f of 2.5 (the same as in the previous example). The results were then rendered using VolView.

Figure 9.5 shows the rendered volume. Figure 9.6 shows an axial, saggital and a coronal slice of the volume.

9.1.5 Isolated Connected

The source code for this section can be found in the file
`Examples/Segmentation/IsolatedConnectedImageFilter.cxx`.

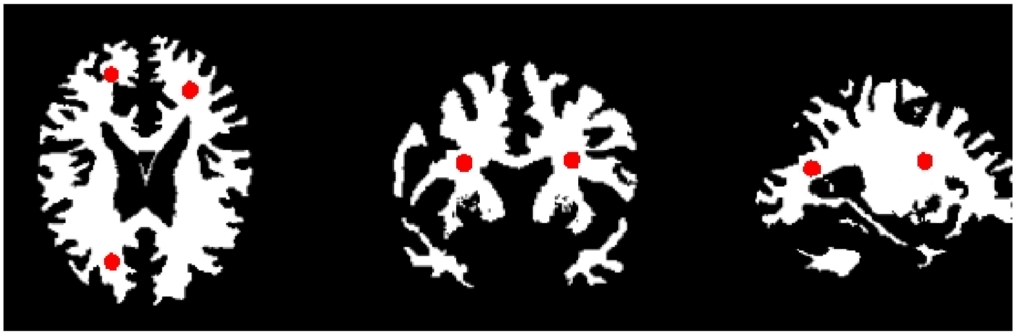


Figure 9.6: Axial, sagittal and coronal slice segmented using Confidence Connected region growing.

The following example illustrates the use of the `itk::IsolatedConnectedImageFilter`. This filter is a close variant of the `itk::ConnectedThresholdImageFilter`. In this filter two seeds and a lower threshold are provided by the user. The filter will grow a region connected to the first seed and **not connected** to the second one. In order to do this, the filter finds an intensity value that could be used as upper threshold for the first seed. A binary search is used to find the value that separates both seeds.

This example closely follows the previous ones. Only the relevant pieces of code are highlighted here.

The header of the IsolatedConnectedImageFilter is included below.

```
#include "itkIsolatedConnectedImageFilter.h"
```

We define the image type using a pixel type and a particular dimension.

```
  typedef   float           InternalPixelType;
  const     unsigned int    Dimension = 2;
  typedef itk::Image< InternalPixelType, Dimension >  InternalImageType;
```

The IsolatedConnectedImageFilter is instantiated in the lines below.

```
  typedef itk::IsolatedConnectedImageFilter<InternalImageType, InternalImageType>
    ConnectedFilterType;
```

One filter of this class is constructed using the `New()` method.

```
  ConnectedFilterType::Pointer isolatedConnected = ConnectedFilterType::New();
```

Now it is time to connect the pipeline.

```
  smoothing->SetInput( reader->GetOutput() );
  isolatedConnected->SetInput( smoothing->GetOutput() );
  caster->SetInput( isolatedConnected->GetOutput() );
  writer->SetInput( caster->GetOutput() );
```

The IsolatedConnectedImageFilter expects the user to specify a threshold and two seeds. In this example, we take all of them from the command line arguments.

```
  isolatedConnected->SetLower(  lowerThreshold  );
  isolatedConnected->SetSeed1( indexSeed1 );
  isolatedConnected->SetSeed2( indexSeed2 );
```

As in the `itk::ConnectedThresholdImageFilter` we must now specify the intensity value to be set on the output pixels and at least one seed point to define the initial region.

Adjacent Structures	Seed1	Seed2	Lower	Isolated value found
Gray matter vs White matter	$(61,140)$	$(63,43)$	150	183.31

Table 9.2: Parameters used for separating white matter from gray matter in Figure 9.7 using the IsolatedConnectedImageFilter.

```
isolatedConnected->SetReplaceValue( 255 );
```

The invocation of the `Update()` method on the writer triggers the execution of the pipeline.

```
try
  {
  writer->Update();
  }
catch( itk::ExceptionObject & excep )
  {
  std::cerr << "Exception caught !" << std::endl;
  std::cerr << excep << std::endl;
  }
```

The intensity value allowing us to separate both regions can be recovered with the method `GetIsolatedValue()`

```
std::cout << "Isolated Value Found = ";
std::cout << isolatedConnected->GetIsolatedValue()  << std::endl;
```

Let's now run this example using the image `BrainProtonDensitySlice.png` provided in the directory `Examples/Data`. We can easily segment the major anatomical structures by providing seed pairs in the appropriate locations and defining values for the lower threshold. It is important to keep in mind in this and the previous examples that the segmentation is being performed in the smoothed version of the image. The selection of threshold values should therefore be performed in the smoothed image since the distribution of intensities could be quite different from that of the input image. As a reminder of this fact, Figure 9.7 presents, from left to right, the input image and the result of smoothing with the `itk::CurvatureFlowImageFilter` followed by segmentation results.

This filter is intended to be used in cases where adjacent anatomical structures are difficult to separate. Selecting one seed in one structure and the other seed in the adjacent structure creates the appropriate setup for computing the threshold that will separate both structures. Table 9.2 presents the parameters used to obtain the images shown in Figure 9.7.

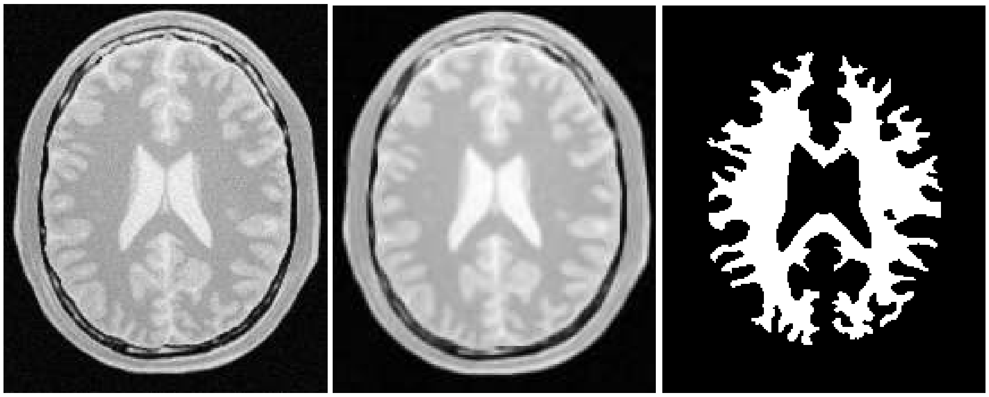


Figure 9.7: Segmentation results of the IsolatedConnectedImageFilter.

9.1.6 Confidence Connected in Vector Images

The source code for this section can be found in the file
`Examples/Segmentation/VectorConfidenceConnected.cxx`.

This example illustrates the use of the confidence connected concept applied to images with vector pixel types. The confidence connected algorithm is implemented for vector images in the class `itk::VectorConfidenceConnected`. The basic difference between the scalar and vector version is that the vector version uses the covariance matrix instead of a variance, and a vector mean instead of a scalar mean. The membership of a vector pixel value to the region is measured using the Mahalanobis distance as implemented in the class `itk::Statistics::MahalanobisDistanceThresholdImageFunction`.

```
#include "itkVectorConfidenceConnectedImageFilter.h"
```

We now define the image type using a particular pixel type and dimension. In this case the `float` type is used for the pixels due to the requirements of the smoothing filter.

```
  typedef   unsigned char                         PixelComponentType;
  typedef   itk::RGBPixel< PixelComponentType >   InputPixelType;
  const     unsigned int    Dimension = 2;
  typedef itk::Image< InputPixelType, Dimension >  InputImageType;
```

We now declare the type of the region growing filter. In this case it is the `itk::VectorConfidenceConnectedImageFilter`.

```
  typedef  itk::VectorConfidenceConnectedImageFilter< InputImageType,
                                  OutputImageType > ConnectedFilterType;
```

Then, we construct one filter of this class using the `New()` method.

```
ConnectedFilterType::Pointer confidenceConnected = ConnectedFilterType::New();
```

Next we create a simple, linear data processing pipeline.

```
confidenceConnected->SetInput( reader->GetOutput() );
writer->SetInput( confidenceConnected->GetOutput() );
```

The VectorConfidenceConnectedImageFilter requires specifying two parameters. First, the multiplier factor f defines how large the range of intensities will be. Small values of the multiplier will restrict the inclusion of pixels to those having similar intensities to those already in the current region. Larger values of the multiplier relax the accepting condition and result in more generous growth of the region. Values that are too large will cause the region to grow into neighboring regions that may actually belong to separate anatomical structures.

```
confidenceConnected->SetMultiplier( multiplier );
```

The number of iterations is typically determined based on the homogeneity of the image intensity representing the anatomical structure to be segmented. Highly homogeneous regions may only require a couple of iterations. Regions with ramp effects, like MRI images with inhomogeneous fields, may require more iterations. In practice, it seems to be more relevant to carefully select the multiplier factor than the number of iterations. However, keep in mind that there is no reason to assume that this algorithm should converge to a stable region. It is possible that by letting the algorithm run for more iterations the region will end up engulfing the entire image.

```
confidenceConnected->SetNumberOfIterations( iterations );
```

The output of this filter is a binary image with zero-value pixels everywhere except on the extracted region. The intensity value to be put inside the region is selected with the method `SetReplaceValue()`

```
confidenceConnected->SetReplaceValue( 255 );
```

The initialization of the algorithm requires the user to provide a seed point. This point should be placed in a *typical* region of the anatomical structure to be segmented. A small neighborhood around the seed point will be used to compute the initial mean and standard deviation for the inclusion criterion. The seed is passed in the form of a `itk::Index` to the `SetSeed()` method.

```
confidenceConnected->SetSeed( index );
```

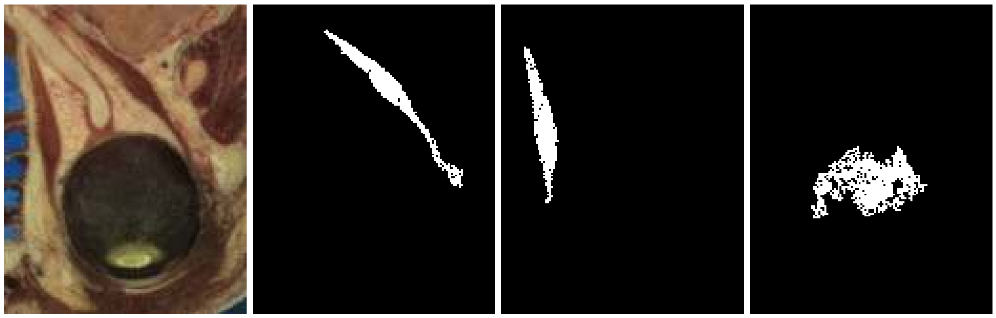


Figure 9.8: Segmentation results of the VectorConfidenceConnected filter for various seed points.

The size of the initial neighborhood around the seed is defined with the method `SetInitialNeighborhoodRadius()`. The neighborhood will be defined as an N-Dimensional rectangular region with $2r+1$ pixels on the side, where r is the value passed as initial neighborhood radius.

```
confidenceConnected->SetInitialNeighborhoodRadius( 3 );
```

The invocation of the `Update()` method on the writer triggers the execution of the pipeline. It is usually wise to put update calls in a `try/catch` block in case errors occur and exceptions are thrown.

```
try
  {
  writer->Update();
  }
catch( itk::ExceptionObject & excep )
  {
  std::cerr << "Exception caught !" << std::endl;
  std::cerr << excep << std::endl;
  }
```

Now let's run this example using as input the image `VisibleWomanEyeSlice.png` provided in the directory `Examples/Data`. We can easily segment the major anatomical structures by providing seeds in the appropriate locations. For example,

Structure	Seed Index	Multiplier	Iterations	Output Image
Rectum	$(70,120)$	7	1	Second from left in Figure 9.8
Rectum	$(23,93)$	7	1	Third from left in Figure 9.8
Vitreo	$(66,66)$	3	1	Fourth from left in Figure 9.8

The coloration of muscular tissue makes it easy to distinguish them from the surrounding anatomical structures. The optic vitrea on the other hand has a coloration that is not very homogeneous inside the eyeball and does not allow to generate a full segmentation based only on color.

The values of the final mean vector and covariance matrix used for the last iteration can be queried using the methods `GetMean()` and `GetCovariance()`.

```
typedef ConnectedFilterType::MeanVectorType   MeanVectorType;

const MeanVectorType & mean = confidenceConnected->GetMean();

std::cout << "Mean vector = " << std::endl;
std::cout << mean << std::endl;

typedef ConnectedFilterType::CovarianceMatrixType   CovarianceMatrixType;

const CovarianceMatrixType & covariance = confidenceConnected->GetCovariance();

std::cout << "Covariance matrix = " << std::endl;
std::cout << covariance << std::endl;
```

9.2 Segmentation Based on Watersheds

9.2.1 Overview

Watershed segmentation classifies pixels into regions using gradient descent on image features and analysis of weak points along region boundaries. Imagine water raining onto a landscape topology and flowing with gravity to collect in low basins. The size of those basins will grow with increasing amounts of precipitation until they spill into one another, causing small basins to merge together into larger basins. Regions (catchment basins) are formed by using local geometric structure to associate points in the image domain with local extrema in some feature measurement such as curvature or gradient magnitude. This technique is less sensitive to user-defined thresholds than classic region-growing methods, and may be better suited for fusing different types of features from different data sets. The watersheds technique is also more flexible in that it does not produce a single image segmentation, but rather a hierarchy of segmentations from which a single region or set of regions can be extracted a-priori, using a threshold, or interactively, with the help of a graphical user interface [96, 97].

The strategy of watershed segmentation is to treat an image f as a height function, i.e., the surface formed by graphing f as a function of its independent parameters, $\vec{x} \in U$. The image f is often not the original input data, but is derived from that data through some filtering, graded (or fuzzy) feature extraction, or fusion of feature maps from different sources. The assumption is that higher values of f (or $-f$) indicate the presence of boundaries in the original data. Watersheds may therefore be considered as a final or intermediate step in a hybrid segmentation method, where the initial segmentation is the generation of the edge feature map.

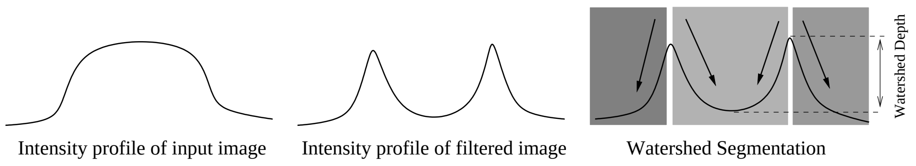

Intensity profile of input image
Intensity profile of filtered image
Watershed Segmentation
Watershed Depth

Figure 9.9: A fuzzy-valued boundary map, from an image or set of images, is segmented using local minima and catchment basins.

Gradient descent associates regions with local minima of f (clearly interior points) using the watersheds of the graph of f, as in Figure 9.9. That is, a segment consists of all points in U whose paths of steepest descent on the graph of f terminate at the same minimum in f. Thus, there are as many segments in an image as there are minima in f. The segment boundaries are "ridges" [44, 45, 25] in the graph of f. In the 1D case ($U \subset \Re$), the watershed boundaries are the local maxima of f, and the results of the watershed segmentation is trivial. For higher-dimensional image domains, the watershed boundaries are not simply local phenomena; they depend on the shape of the entire watershed.

The drawback of watershed segmentation is that it produces a region for each local minimum—in practice too many regions—and an over segmentation results. To alleviate this, we can establish a minimum watershed depth. The watershed depth is the difference in height between the watershed minimum and the lowest boundary point. In other words, it is the maximum depth of water a region could hold without flowing into any of its neighbors. Thus, a watershed segmentation algorithm can sequentially combine watersheds whose depths fall below the minimum until all of the watersheds are of sufficient depth. This depth measurement can be combined with other saliency measurements, such as size. The result is a segmentation containing regions whose boundaries and size are significant. Because the merging process is sequential, it produces a hierarchy of regions, as shown in Figure 9.10. Previous work has shown the benefit of a user-assisted approach that provides a graphical interface to this hierarchy, so that a technician can quickly move from the small regions that lie within an area of interest to the union of regions that correspond to the anatomical structure [97].

There are two different algorithms commonly used to implement watersheds: top-down and bottom-up. The top-down, gradient descent strategy was chosen for ITK because we want to consider the output of multi-scale differential operators, and the f in question will therefore have floating point values. The bottom-up strategy starts with seeds at the local minima in the image and grows regions outward and upward at discrete intensity levels (equivalent to a sequence of morphological operations and sometimes called *morphological watersheds* [73].) This limits the accuracy by enforcing a set of discrete gray levels on the image.

Figure 9.11 shows how the ITK image-to-image watersheds filter is constructed. The filter is actually a collection of smaller filters that modularize the several steps of the algorithm in a mini-pipeline. The segmenter object creates the initial segmentation via steepest descent from each pixel to local minima. Shallow background regions are removed (flattened) before segmentation using a simple minimum value threshold (this helps to minimize oversegmentation

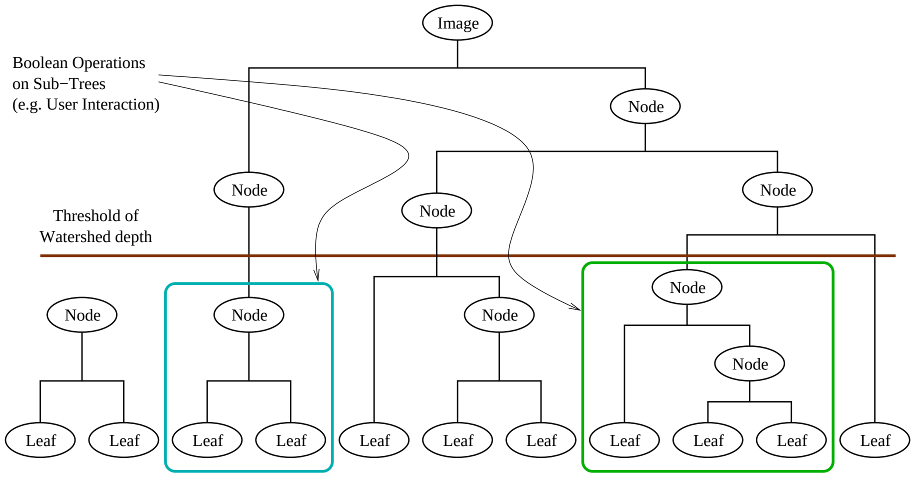


Figure 9.10: A watershed segmentation combined with a saliency measure (watershed depth) produces a hierarchy of regions. Structures can be derived from images by either thresholding the saliency measure or combining subtrees within the hierarchy.

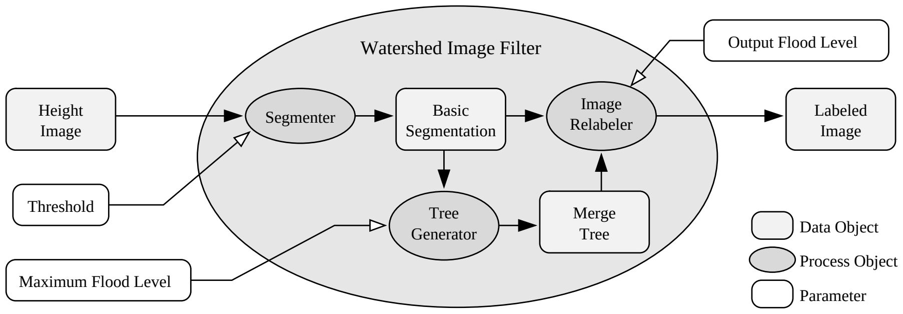


Figure 9.11: The construction of the Insight watersheds filter.

of the image). The initial segmentation is passed to a second sub-filter that generates a hierarchy of basins to a user-specified maximum watershed depth. The relabeler object at the end of the mini-pipeline uses the hierarchy and the initial segmentation to produce an output image at any scale *below* the user-specified maximum. Data objects are cached in the mini-pipeline so that changing watershed depths only requires a (fast) relabeling of the basic segmentation. The three parameters that control the filter are shown in Figure 9.11 connected to their relevant processing stages.

9.2.2 Using the ITK Watershed Filter

The source code for this section can be found in the file
`Examples/Segmentation/WatershedSegmentation1.cxx`.

The following example illustrates how to preprocess and segment images using the `itk::WatershedImageFilter`. Note that the care with which the data is preprocessed will greatly affect the quality of your result. Typically, the best results are obtained by preprocessing the original image with an edge-preserving diffusion filter, such as one of the anisotropic diffusion filters, or with the bilateral image filter. As noted in Section 9.2.1, the height function used as input should be created such that higher positive values correspond to object boundaries. A suitable height function for many applications can be generated as the gradient magnitude of the image to be segmented.

The `itk::VectorGradientMagnitudeAnisotropicDiffusionImageFilter` class is used to smooth the image and the `itk::VectorGradientMagnitudeImageFilter` is used to generate the height function. We begin by including all preprocessing filter header files and the header file for the WatershedImageFilter. We use the vector versions of these filters because the input data is a color image.

```
#include "itkVectorGradientAnisotropicDiffusionImageFilter.h"
#include "itkVectorGradientMagnitudeImageFilter.h"
#include "itkWatershedImageFilter.h"
```

We now declare the image and pixel types to use for instantiation of the filters. All of these filters expect real-valued pixel types in order to work properly. The preprocessing stages are done directly on the vector-valued data and the segmentation is done using floating point scalar data. Images are converted from RGB pixel type to numerical vector type using `itk::VectorCastImageFilter`.

```
  typedef itk::RGBPixel<unsigned char>   RGBPixelType;
  typedef itk::Image<RGBPixelType, 2>    RGBImageType;
  typedef itk::Vector<float, 3>          VectorPixelType;
  typedef itk::Image<VectorPixelType, 2> VectorImageType;
  typedef itk::Image<unsigned long, 2>   LabeledImageType;
  typedef itk::Image<float, 2>           ScalarImageType;
```

The various image processing filters are declared using the types created above and eventually used in the pipeline.

```
typedef itk::ImageFileReader<RGBImageType> FileReaderType;
typedef itk::VectorCastImageFilter<RGBImageType, VectorImageType>
  CastFilterType;
typedef itk::VectorGradientAnisotropicDiffusionImageFilter<VectorImageType,
  VectorImageType>  DiffusionFilterType;
typedef itk::VectorGradientMagnitudeImageFilter<VectorImageType>
  GradientMagnitudeFilterType;
typedef itk::WatershedImageFilter<ScalarImageType> WatershedFilterType;
```

Next we instantiate the filters and set their parameters. The first step in the image processing pipeline is diffusion of the color input image using an anisotropic diffusion filter. For this class of filters, the CFL condition requires that the time step be no more than 0.25 for two-dimensional images, and no more than 0.125 for three-dimensional images. The number of iterations and the conductance term will be taken from the command line. See Section 6.7.3 for more information on the ITK anisotropic diffusion filters.

```
DiffusionFilterType::Pointer diffusion = DiffusionFilterType::New();
diffusion->SetNumberOfIterations( atoi(argv[4]) );
diffusion->SetConductanceParameter( atof(argv[3]) );
diffusion->SetTimeStep(0.125);
```

The ITK gradient magnitude filter for vector-valued images can optionally take several parameters. Here we allow only enabling or disabling of principal component analysis.

```
GradientMagnitudeFilterType::Pointer
  gradient = GradientMagnitudeFilterType::New();
gradient->SetUsePrincipleComponents(atoi(argv[7]));
```

Finally we set up the watershed filter. There are two parameters. `Level` controls watershed depth, and `Threshold` controls the lower thresholding of the input. Both parameters are set as a percentage (0.0 - 1.0) of the maximum depth in the input image.

```
WatershedFilterType::Pointer watershed = WatershedFilterType::New();
watershed->SetLevel( atof(argv[6]) );
watershed->SetThreshold( atof(argv[5]) );
```

The output of WatershedImageFilter is an image of unsigned long integer labels, where a label denotes membership of a pixel in a particular segmented region. This format is not practical for visualization, so for the purposes of this example, we will convert it to RGB pixels. RGB images have the advantage that they can be saved as a simple png file and viewed using any standard image viewer software. The `itk::Functor::ScalarToRGBPixelFunctor` class is a

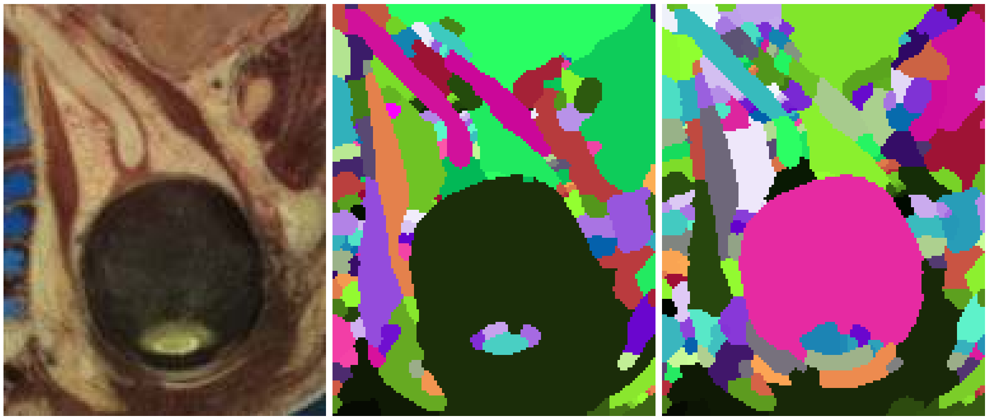


Figure 9.12: Segmented section of Visible Human female head and neck cryosection data. At left is the original image. The image in the middle was generated with parameters: conductance = 2.0, iterations = 10, threshold = 0.0, level = 0.05, principal components = on. The image on the right was generated with parameters: conductance = 2.0, iterations = 10, threshold = 0.001, level = 0.15, principal components = off.

special function object designed to hash a scalar value into an `itk::RGBPixel`. Plugging this functor into the `itk::UnaryFunctorImageFilter` creates an image filter for that converts scalar images to RGB images.

```
typedef itk::Functor::ScalarToRGBPixelFunctor<unsigned long>
  ColorMapFunctorType;
typedef itk::UnaryFunctorImageFilter<LabeledImageType,
  RGBImageType, ColorMapFunctorType> ColorMapFilterType;
ColorMapFilterType::Pointer colormapper = ColorMapFilterType::New();
```

The filters are connected into a single pipeline, with readers and writers at each end.

```
caster->SetInput(reader->GetOutput());
diffusion->SetInput(caster->GetOutput());
gradient->SetInput(diffusion->GetOutput());
watershed->SetInput(gradient->GetOutput());
colormapper->SetInput(watershed->GetOutput());
writer->SetInput(colormapper->GetOutput());
```

Tuning the filter parameters for any particular application is a process of trial and error. The *threshold* parameter can be used to great effect in controlling oversegmentation of the image. Raising the threshold will generally reduce computation time and produce output with fewer and larger regions. The trick in tuning parameters is to consider the scale level of the objects that you are trying to segment in the image. The best time/quality trade-off will be achieved when the image is smoothed and thresholded to eliminate features just below the desired scale.

Figure 9.12 shows output from the example code. The input image is taken from the Visible Human female data around the right eye. The images on the right are colorized watershed segmentations with parameters set to capture objects such as the optic nerve and lateral rectus muscles, which can be seen just above and to the left and right of the eyeball. Note that a critical difference between the two segmentations is the mode of the gradient magnitude calculation.

A note on the computational complexity of the watershed algorithm is warranted. Most of the complexity of the ITK implementation lies in generating the hierarchy. Processing times for this stage are non-linear with respect to the number of catchment basins in the initial segmentation. This means that the amount of information contained in an image is more significant than the number of pixels in the image. A very large, but very flat input take less time to segment than a very small, but very detailed input.

9.3 Level Set Segmentation

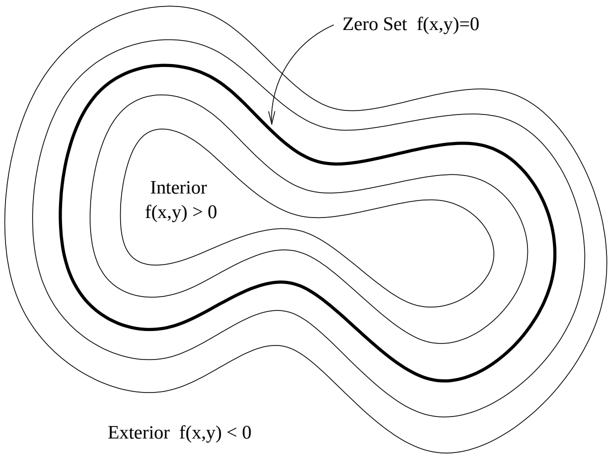


Figure 9.13: Concept of zero set in a level set.

The paradigm of the level set is that it is a numerical method for tracking the evolution of contours and surfaces. Instead of manipulating the contour directly, the contour is embedded as the zero level set of a higher dimensional function called the level-set function, $\psi(\mathbf{X},\mathbf{t})$. The level-set function is then evolved under the control of a differential equation. At any time, the evolving contour can be obtained by extracting the zero level-set $\Gamma((\mathbf{X}),\mathbf{t}) = \{\psi(\mathbf{X},\mathbf{t}) = \mathbf{0}\}$ from the output. The main advantages of using level sets is that arbitrarily complex shapes can be modeled and topological changes such as merging and splitting are handled implicitly.

Level sets can be used for image segmentation by using image-based features such as mean intensity, gradient and edges in the governing differential equation. In a typical approach, a contour is initialized by a user and is then evolved until it fits the form of an anatomical structure in the image. Many different implementations and variants of this basic concept have been published in the literature. An overview of the field has been made by Sethian [74].

The following sections introduce practical examples of some of the level set segmentation methods available in ITK. The remainder of this section describes features common to all of these filters except the `itk::FastMarchingImageFilter`, which is derived from a different code framework. Understanding these features will aid in using the filters more effectively.

Each filter makes use of a generic level-set equation to compute the update to the solution ψ of the partial differential equation.

$$\frac{d}{dt}\psi = -\alpha\mathbf{A}(\mathbf{x})\cdot\nabla\psi - \beta P(\mathbf{x}) \mid \nabla\psi \mid + \gamma Z(\mathbf{x})\kappa \mid \nabla\psi \mid \tag{9.3}$$

where $\mathbf{A}$ is an advection term, P is a propagation (expansion) term, and Z is a spatial modifier term for the mean curvature κ. The scalar constants α, β, and γ weight the relative influence of each of the terms on the movement of the interface. A segmentation filter may use all of these terms in its calculations, or it may omit one or more terms. If a term is left out of the equation, then setting the corresponding scalar constant weighting will have no effect.

All of the level-set based segmentation filters *must* operate with floating point precision to pro-

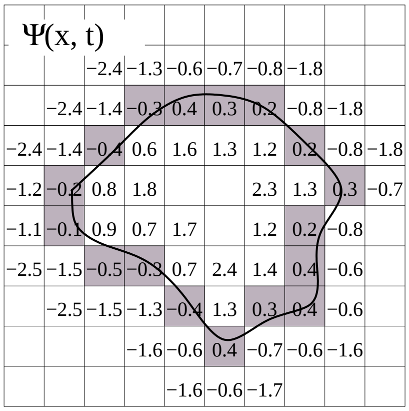


Figure 9.14: The implicit level set surface Γ is the black line superimposed over the image grid. The location of the surface is interpolated by the image pixel values. The grid pixels closest to the implicit surface are shown in gray.

duce valid results. The third, optional template parameter is the *numerical type* used for calculations and as the output image pixel type. The numerical type is `float` by default, but can be changed to `double` for extra precision. A user-defined, signed floating point type that defines all of the necessary arithmetic operators and has sufficient precision is also a valid choice. You should not use types such as `int` or `unsigned char` for the numerical parameter. If the input image pixel types do not match the numerical type, those inputs will be cast to an image of appropriate type when the filter is executed.

Most filters require two images as input, an initial model $\psi(\mathbf{X}, \mathbf{t} = \mathbf{0})$, and a *feature image*, which is either the image you wish to segment or some preprocessed version. You must specify the isovalue that represents the surface Γ in your initial model. The single image output of each filter is the function ψ at the final time step. It is important to note that the contour representing the surface Γ is the zero level-set of the output image, and not the isovalue you specified for the initial model. To represent Γ using the original isovalue, simply add that value back to the output.

The solution Γ is calculated to subpixel precision. The best discrete approximation of the surface is therefore the set of grid positions closest to the zero-crossings in the image, as shown in Figure 9.14. The `itk::ZeroCrossingImageFilter` operates by finding exactly those grid positions and can be used to extract the surface.

There are two important considerations when analyzing the processing time for any particular level-set segmentation task: the surface area of the evolving interface and the total distance that the surface must travel. Because the level-set equations are usually solved only at pixels near the surface (fast marching methods are an exception), the time taken at each iteration depends on the number of points on the surface. This means that as the surface grows, the solver will slow down proportionally. Because the surface must evolve slowly to prevent numerical instabilities

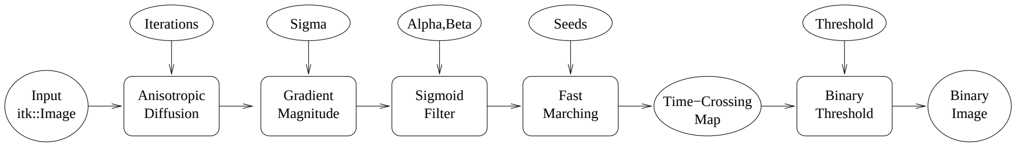


Figure 9.15: Collaboration diagram of the FastMarchingImageFilter applied to a segmentation task.

in the solution, the distance the surface must travel in the image dictates the total number of iterations required.

Some level-set techniques are relatively insensitive to initial conditions and are therefore suitable for region-growing segmentation. Other techniques, such as the `itk::LaplacianSegmentationLevelSetImageFilter`, can easily become "stuck" on image features close to their initialization and should be used only when a reasonable prior segmentation is available as the initialization. For best efficiency, your initial model of the surface should be the best guess possible for the solution. When extending the example applications given here to higher dimensional images, for example, you can improve results and dramatically decrease processing time by using a multi-scale approach. Start with a downsampled volume and work back to the full resolution using the results at each intermediate scale as the initialization for the next scale.

9.3.1 Fast Marching Segmentation

The source code for this section can be found in the file
`Examples/Segmentation/FastMarchingImageFilter.cxx`.

When the differential equation governing the level set evolution has a very simple form, a fast evolution algorithm called fast marching can be used.

The following example illustrates the use of the `itk::FastMarchingImageFilter`. This filter implements a fast marching solution to a simple level set evolution problem. In this example, the speed term used in the differential equation is expected to be provided by the user in the form of an image. This image is typically computed as a function of the gradient magnitude. Several mappings are popular in the literature, for example, the negative exponential $exp(-x)$ and the reciprocal $1/(1+x)$. In the current example we decided to use a Sigmoid function since it offers a good deal of control parameters that can be customized to shape a nice speed image.

The mapping should be done in such a way that the propagation speed of the front will be very low close to high image gradients while it will move rather fast in low gradient areas. This arrangement will make the contour propagate until it reaches the edges of anatomical structures in the image and then slow down in front of those edges. The output of the FastMarchingImageFilter is a *time-crossing map* that indicates, for each pixel, how much time it would take for the front to arrive at the pixel location.

The application of a threshold in the output image is then equivalent to taking a snapshot of the contour at a particular time during its evolution. It is expected that the contour will take a longer time to cross over the edges of a particular anatomical structure. This should result in large changes on the time-crossing map values close to the structure edges. Segmentation is performed with this filter by locating a time range in which the contour was contained for a long time in a region of the image space.

Figure 9.15 shows the major components involved in the application of the FastMarchingImageFilter to a segmentation task. It involves an initial stage of smoothing using the `itk::CurvatureAnisotropicDiffusionImageFilter`. The smoothed image is passed as the input to the `itk::GradientMagnitudeRecursiveGaussianImageFilter` and then to the `itk::SigmoidImageFilter`. Finally, the output of the FastMarchingImageFilter is passed to a `itk::BinaryThresholdImageFilter` in order to produce a binary mask representing the segmented object.

The code in the following example illustrates the typical setup of a pipeline for performing segmentation with fast marching. First, the input image is smoothed using an edge-preserving filter. Then the magnitude of its gradient is computed and passed to a sigmoid filter. The result of the sigmoid filter is the image potential that will be used to affect the speed term of the differential equation.

Let's start by including the following headers. First we include the header of the CurvatureAnisotropicDiffusionImageFilter that will be used for removing noise from the input image.

```
#include "itkCurvatureAnisotropicDiffusionImageFilter.h"
```

The headers of the GradientMagnitudeRecursiveGaussianImageFilter and SigmoidImageFilter are included below. Together, these two filters will produce the image potential for regulating the speed term in the differential equation describing the evolution of the level set.

```
#include "itkGradientMagnitudeRecursiveGaussianImageFilter.h"
#include "itkSigmoidImageFilter.h"
```

Of course, we will need the `itk::Image` class and the FastMarchingImageFilter class. Hence we include their headers.

```
#include "itkImage.h"
#include "itkFastMarchingImageFilter.h"
```

The time-crossing map resulting from the FastMarchingImageFilter will be thresholded using the BinaryThresholdImageFilter. We include its header here.

```
#include "itkBinaryThresholdImageFilter.h"
```

Reading and writing images will be done with the `itk::ImageFileReader` and `itk::ImageFileWriter`.

```
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
```

We now define the image type using a pixel type and a particular dimension. In this case the `float` type is used for the pixels due to the requirements of the smoothing filter.

```
  typedef   float           InternalPixelType;
  const     unsigned int    Dimension = 2;
  typedef itk::Image< InternalPixelType, Dimension >  InternalImageType;
```

The output image, on the other hand, is declared to be binary.

```
  typedef unsigned char OutputPixelType;
  typedef itk::Image< OutputPixelType, Dimension > OutputImageType;
```

The type of the BinaryThresholdImageFilter filter is instantiated below using the internal image type and the output image type.

```
  typedef itk::BinaryThresholdImageFilter< InternalImageType,
                        OutputImageType    >    ThresholdingFilterType;
  ThresholdingFilterType::Pointer thresholder = ThresholdingFilterType::New();
```

The upper threshold passed to the BinaryThresholdImageFilter will define the time snapshot that we are taking from the time-crossing map. In an ideal application the user should be able to select this threshold interactively using visual feedback. Here, since it is a minimal example, the value is taken from the command line arguments.

```
  thresholder->SetLowerThreshold(           0.0  );
  thresholder->SetUpperThreshold( timeThreshold  );

  thresholder->SetOutsideValue(  0  );
  thresholder->SetInsideValue(  255 );
```

We instantiate reader and writer types in the following lines.

```
  typedef  itk::ImageFileReader< InternalImageType > ReaderType;
  typedef  itk::ImageFileWriter<  OutputImageType  > WriterType;
```

The CurvatureAnisotropicDiffusionImageFilter type is instantiated using the internal image type.

```
  typedef   itk::CurvatureAnisotropicDiffusionImageFilter<
                               InternalImageType,
                               InternalImageType >  SmoothingFilterType;
```

Then, the filter is created by invoking the `New()` method and assigning the result to a `itk::SmartPointer`.

```
SmoothingFilterType::Pointer smoothing = SmoothingFilterType::New();
```

The types of the GradientMagnitudeRecursiveGaussianImageFilter and SigmoidImageFilter are instantiated using the internal image type.

```
typedef   itk::GradientMagnitudeRecursiveGaussianImageFilter<
                              InternalImageType,
                              InternalImageType >  GradientFilterType;

typedef   itk::SigmoidImageFilter<
                              InternalImageType,
                              InternalImageType >  SigmoidFilterType;
```

The corresponding filter objects are instantiated with the `New()` method.

```
GradientFilterType::Pointer  gradientMagnitude = GradientFilterType::New();
SigmoidFilterType::Pointer sigmoid = SigmoidFilterType::New();
```

The minimum and maximum values of the SigmoidImageFilter output are defined with the methods `SetOutputMinimum()` and `SetOutputMaximum()`. In our case, we want these two values to be 0.0 and 1.0 respectively in order to get a nice speed image to feed to the FastMarchingImageFilter. Additional details on the use of the SigmoidImageFilter are presented in Section 6.3.2.

```
sigmoid->SetOutputMinimum(  0.0  );
sigmoid->SetOutputMaximum(  1.0  );
```

We now declare the type of the FastMarchingImageFilter.

```
typedef  itk::FastMarchingImageFilter< InternalImageType,
                              InternalImageType >    FastMarchingFilterType;
```

Then, we construct one filter of this class using the `New()` method.

```
FastMarchingFilterType::Pointer  fastMarching = FastMarchingFilterType::New();
```

The filters are now connected in a pipeline shown in Figure 9.15 using the following lines.

```
smoothing->SetInput( reader->GetOutput() );
gradientMagnitude->SetInput( smoothing->GetOutput() );
sigmoid->SetInput( gradientMagnitude->GetOutput() );
fastMarching->SetInput( sigmoid->GetOutput() );
thresholder->SetInput( fastMarching->GetOutput() );
writer->SetInput( thresholder->GetOutput() );
```

The CurvatureAnisotropicDiffusionImageFilter class requires a couple of parameters to be defined. The following are typical values for $2D$ images. However they may have to be adjusted depending on the amount of noise present in the input image. This filter has been discussed in Section 6.7.3.

```
smoothing->SetTimeStep( 0.125 );
smoothing->SetNumberOfIterations(  5 );
smoothing->SetConductanceParameter( 9.0 );
```

The GradientMagnitudeRecursiveGaussianImageFilter performs the equivalent of a convolution with a Gaussian kernel followed by a derivative operator. The sigma of this Gaussian can be used to control the range of influence of the image edges. This filter has been discussed in Section 6.4.2

```
gradientMagnitude->SetSigma(  sigma  );
```

The SigmoidImageFilter class requires two parameters to define the linear transformation to be applied to the sigmoid argument. These parameters are passed using the `SetAlpha()` and `SetBeta()` methods. In the context of this example, the parameters are used to intensify the differences between regions of low and high values in the speed image. In an ideal case, the speed value should be 1.0 in the homogeneous regions of anatomical structures and the value should decay rapidly to 0.0 around the edges of structures. The heuristic for finding the values is the following. From the gradient magnitude image, let's call $K1$ the minimum value along the contour of the anatomical structure to be segmented. Then, let's call $K2$ an average value of the gradient magnitude in the middle of the structure. These two values indicate the dynamic range that we want to map to the interval $[0:1]$ in the speed image. We want the sigmoid to map $K1$ to 0.0 and $K2$ to 1.0. Given that $K1$ is expected to be higher than $K2$ and we want to map those values to 0.0 and 1.0 respectively, we want to select a negative value for alpha so that the sigmoid function will also do an inverse intensity mapping. This mapping will produce a speed image such that the level set will march rapidly on the homogeneous region and will definitely stop on the contour. The suggested value for beta is $(K1+K2)/2$ while the suggested value for alpha is $(K2-K1)/6$, which must be a negative number. In our simple example the values are provided by the user from the command line arguments. The user can estimate these values by observing the gradient magnitude image.

```
sigmoid->SetAlpha( alpha );
sigmoid->SetBeta(  beta  );
```

The FastMarchingImageFilter requires the user to provide a seed point from which the contour will expand. The user can actually pass not only one seed point but a set of them. A good set of seed points increases the chances of segmenting a complex object without missing parts. The use of multiple seeds also helps to reduce the amount of time needed by the front to visit a whole object and hence reduces the risk of leaks on the edges of regions visited earlier. For example, when segmenting an elongated object, it is undesirable to place a single seed at one extreme of the object since the front will need a long time to propagate to the other end of the object. Placing several seeds along the axis of the object will probably be the best strategy to ensure that the entire object is captured early in the expansion of the front. One of the important properties of level sets is their natural ability to fuse several fronts implicitly without any extra bookkeeping. The use of multiple seeds takes good advantage of this property.

The seeds are passed stored in a container. The type of this container is defined as `NodeContainer` among the FastMarchingImageFilter traits.

```
typedef FastMarchingFilterType::NodeContainer           NodeContainer;
typedef FastMarchingFilterType::NodeType                NodeType;
NodeContainer::Pointer seeds = NodeContainer::New();
```

Nodes are created as stack variables and initialized with a value and an `itk::Index` position.

```
NodeType node;
const double seedValue = 0.0;

node.SetValue( seedValue );
node.SetIndex( seedPosition );
```

The list of nodes is initialized and then every node is inserted using the `InsertElement()`.

```
seeds->Initialize();
seeds->InsertElement( 0, node );
```

The set of seed nodes is now passed to the FastMarchingImageFilter with the method `SetTrialPoints()`.

```
fastMarching->SetTrialPoints(  seeds  );
```

The FastMarchingImageFilter requires the user to specify the size of the image to be produced as output. This is done using the `SetOutputSize()`. Note that the size is obtained here from the output image of the smoothing filter. The size of this image is valid only after the `Update()` methods of this filter has been called directly or indirectly.

```
fastMarching->SetOutputSize(
         reader->GetOutput()->GetBufferedRegion().GetSize() );
```

Structure	Seed Index	σ	α	β	Threshold	Output Image from left
Left Ventricle	$(81,114)$	1.0	-0.5	3.0	100	First
Right Ventricle	$(99,114)$	1.0	-0.5	3.0	100	Second
White matter	$(56,92)$	1.0	-0.3	2.0	200	Third
Gray matter	$(40,90)$	0.5	-0.3	2.0	200	Fourth

Table 9.3: Parameters used for segmenting some brain structures shown in Figure 9.17 using the filter FastMarchingImageFilter. All of them used a stopping value of 100.

Since the front representing the contour will propagate continuously over time, it is desirable to stop the process once a certain time has been reached. This allows us to save computation time under the assumption that the region of interest has already been computed. The value for stopping the process is defined with the method `SetStoppingValue()`. In principle, the stopping value should be a little bit higher than the threshold value.

```
fastMarching->SetStoppingValue(  stoppingTime  );
```

The invocation of the `Update()` method on the writer triggers the execution of the pipeline. As usual, the call is placed in a `try/catch` block should any errors occur or exceptions be thrown.

```
try
  {
  writer->Update();
  }
catch( itk::ExceptionObject & excep )
  {
  std::cerr << "Exception caught !" << std::endl;
  std::cerr << excep << std::endl;
  }
```

Now let's run this example using the input image `BrainProtonDensitySlice.png` provided in the directory `Examples/Data`. We can easily segment the major anatomical structures by providing seeds in the appropriate locations. The following table presents the parameters used for some structures.

Figure 9.16 presents the intermediate outputs of the pipeline illustrated in Figure 9.15. They are from left to right: the output of the anisotropic diffusion filter, the gradient magnitude of the smoothed image and the sigmoid of the gradient magnitude which is finally used as the speed image for the FastMarchingImageFilter.

Notice that the gray matter is not being completely segmented. This illustrates the vulnerability of the level set methods when the anatomical structures to be segmented do not occupy extended regions of the image. This is especially true when the width of the structure is comparable to the size of the attenuation bands generated by the gradient filter. A possible workaround for

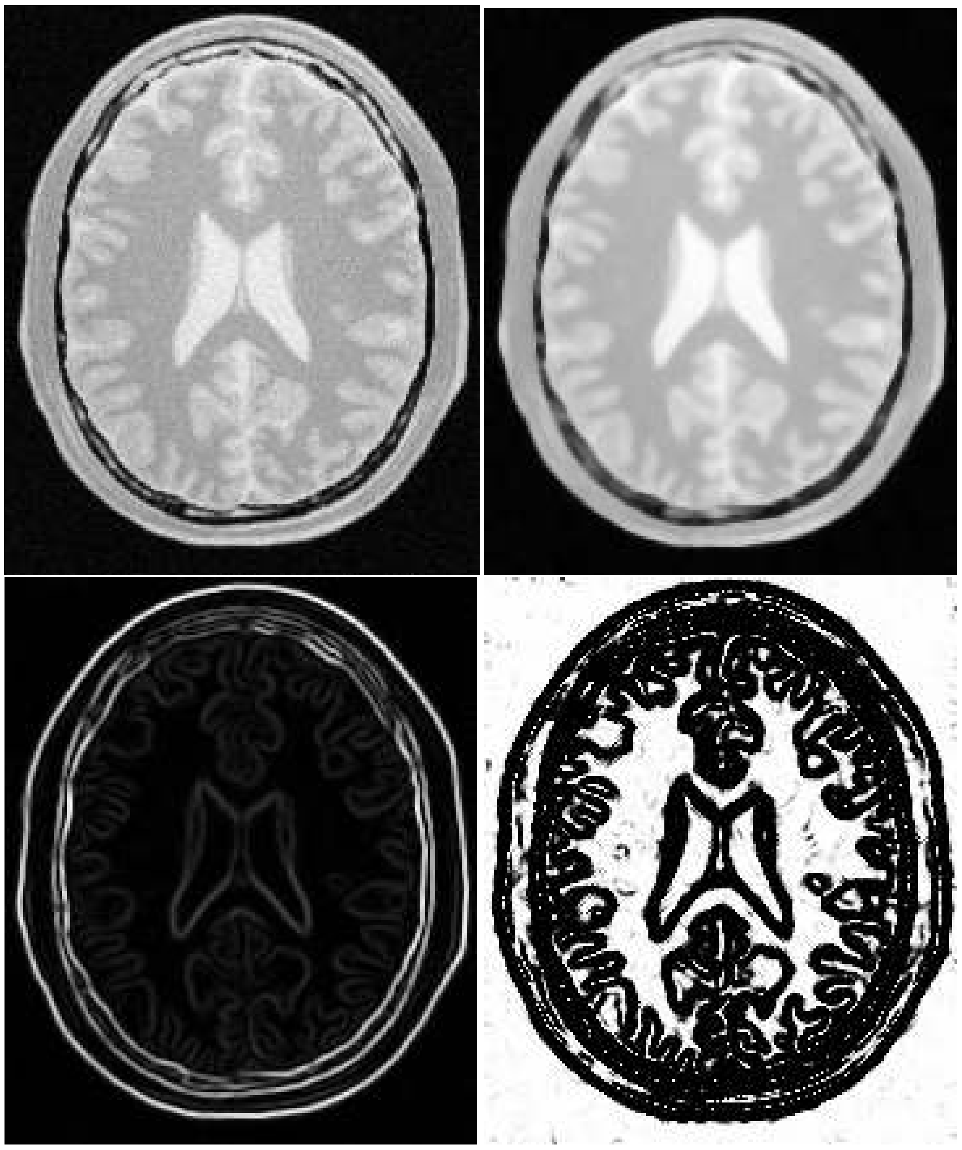


Figure 9.16: Images generated by the segmentation process based on the FastMarchingImageFilter. From left to right and top to bottom: input image to be segmented, image smoothed with an edge-preserving smoothing filter, gradient magnitude of the smoothed image, sigmoid of the gradient magnitude. This last image, the sigmoid, is used to compute the speed term for the front propagation

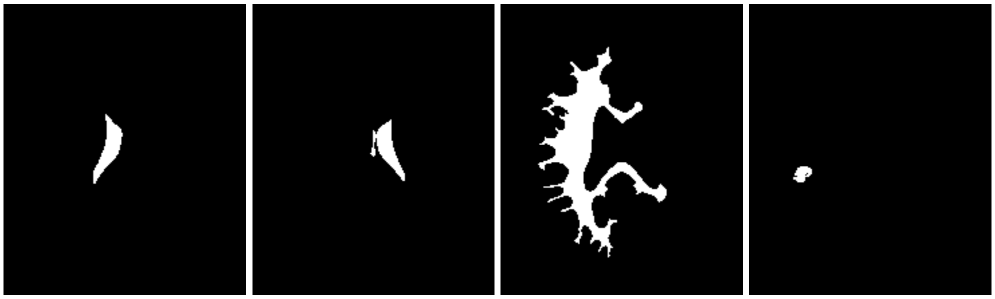


Figure 9.17: Images generated by the segmentation process based on the FastMarchingImageFilter. From left to right: segmentation of the left ventricle, segmentation of the right ventricle, segmentation of the white matter, attempt of segmentation of the gray matter.

this limitation is to use multiple seeds distributed along the elongated object. However, note that white matter versus gray matter segmentation is not a trivial task, and may require a more elaborate approach than the one used in this basic example.

9.3.2 Shape Detection Segmentation

The source code for this section can be found in the file
`Examples/Segmentation/ShapeDetectionLevelSetFilter.cxx`.

The use of the `itk::ShapeDetectionLevelSetImageFilter` is illustrated in the following example. The implementation of this filter in ITK is based on the paper by Malladi et al [54]. In this implementation, the governing differential equation has an additional curvature-based term. This term acts as a smoothing term where areas of high curvature, assumed to be due to noise, are smoothed out. Scaling parameters are used to control the tradeoff between the expansion term and the smoothing term. One consequence of this additional curvature term is that the fast marching algorithm is no longer applicable, because the contour is no longer guaranteed to always be expanding. Instead, the level set function is updated iteratively.

The ShapeDetectionLevelSetImageFilter expects two inputs, the first being an initial Level Set in the form of an `itk::Image`, and the second being a feature image. For this algorithm, the feature image is an edge potential image that basically follows the same rules applicable to the speed image used for the FastMarchingImageFilter discussed in Section 9.3.1.

In this example we use an FastMarchingImageFilter to produce the initial level set as the distance function to a set of user-provided seeds. The FastMarchingImageFilter is run with a constant speed value which enables us to employ this filter as a distance map calculator.

Figure 9.18 shows the major components involved in the application of the ShapeDetection-LevelSetImageFilter to a segmentation task. The first stage involves smoothing using the `itk::CurvatureAnisotropicDiffusionImageFilter`. The smoothed image is passed as the input for the `itk::GradientMagnitudeRecursiveGaussianImageFilter` and then to

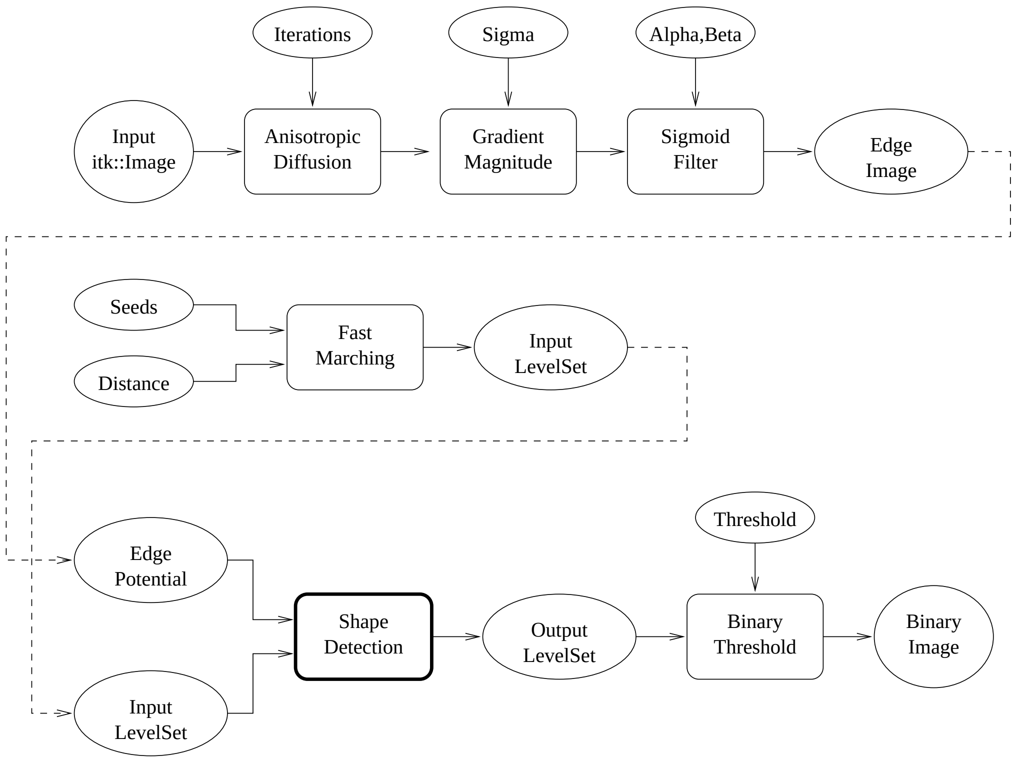


Figure 9.18: Collaboration diagram for the ShapeDetectionLevelSetImageFilter applied to a segmentation task.

the `itk::SigmoidImageFilter` in order to produce the edge potential image. A set of user-provided seeds is passed to an FastMarchingImageFilter in order to compute the distance map. A constant value is subtracted from this map in order to obtain a level set in which the *zero set* represents the initial contour. This level set is also passed as input to the ShapeDetection-LevelSetImageFilter.

Finally, the level set at the output of the ShapeDetectionLevelSetImageFilter is passed to an BinaryThresholdImageFilter in order to produce a binary mask representing the segmented object.

Let's start by including the headers of the main filters involved in the preprocessing.

```
#include "itkCurvatureAnisotropicDiffusionImageFilter.h"
#include "itkGradientMagnitudeRecursiveGaussianImageFilter.h"
#include "itkSigmoidImageFilter.h"
```

The edge potential map is generated using these filters as in the previous example.

We will need the Image class, the FastMarchingImageFilter class and the ShapeDetection-LevelSetImageFilter class. Hence we include their headers here.

```
#include "itkImage.h"
#include "itkFastMarchingImageFilter.h"
#include "itkShapeDetectionLevelSetImageFilter.h"
```

The level set resulting from the ShapeDetectionLevelSetImageFilter will be thresholded at the zero level in order to get a binary image representing the segmented object. The BinaryThresholdImageFilter is used for this purpose.

```
#include "itkBinaryThresholdImageFilter.h"
```

We now define the image type using a particular pixel type and a dimension. In this case the `float` type is used for the pixels due to the requirements of the smoothing filter.

```
  typedef   float           InternalPixelType;
  const     unsigned int    Dimension = 2;
  typedef itk::Image< InternalPixelType, Dimension >  InternalImageType;
```

The output image, on the other hand, is declared to be binary.

```
  typedef unsigned char OutputPixelType;
  typedef itk::Image< OutputPixelType, Dimension > OutputImageType;
```

The type of the BinaryThresholdImageFilter filter is instantiated below using the internal image type and the output image type.

```
typedef itk::BinaryThresholdImageFilter< InternalImageType, OutputImageType >
  ThresholdingFilterType;
ThresholdingFilterType::Pointer thresholder = ThresholdingFilterType::New();
```

The upper threshold of the BinaryThresholdImageFilter is set to 0.0 in order to display the zero set of the resulting level set. The lower threshold is set to a large negative number in order to ensure that the interior of the segmented object will appear inside the binary region.

```
thresholder->SetLowerThreshold( -1000.0 );
thresholder->SetUpperThreshold(     0.0 );

thresholder->SetOutsideValue(  0  );
thresholder->SetInsideValue(  255 );
```

The CurvatureAnisotropicDiffusionImageFilter type is instantiated using the internal image type.

```
typedef   itk::CurvatureAnisotropicDiffusionImageFilter<
                               InternalImageType,
                               InternalImageType >  SmoothingFilterType;
```

The filter is instantiated by invoking the `New()` method and assigning the result to a `itk::SmartPointer`.

```
SmoothingFilterType::Pointer smoothing = SmoothingFilterType::New();
```

The types of the GradientMagnitudeRecursiveGaussianImageFilter and SigmoidImageFilter are instantiated using the internal image type.

```
typedef   itk::GradientMagnitudeRecursiveGaussianImageFilter<
                               InternalImageType,
                               InternalImageType >  GradientFilterType;

typedef   itk::SigmoidImageFilter<
                               InternalImageType,
                               InternalImageType >  SigmoidFilterType;
```

The corresponding filter objects are created with the method `New()`.

```
GradientFilterType::Pointer  gradientMagnitude = GradientFilterType::New();
SigmoidFilterType::Pointer sigmoid = SigmoidFilterType::New();
```

The minimum and maximum values of the SigmoidImageFilter output are defined with the methods `SetOutputMinimum()` and `SetOutputMaximum()`. In our case, we want these two values to be 0.0 and 1.0 respectively in order to get a nice speed image to feed to the FastMarchingImageFilter. Additional details on the use of the SigmoidImageFilter are presented in Section 6.3.2.

```
sigmoid->SetOutputMinimum(  0.0  );
sigmoid->SetOutputMaximum(  1.0  );
```

We now declare the type of the FastMarchingImageFilter that will be used to generate the initial level set in the form of a distance map.

```
typedef  itk::FastMarchingImageFilter< InternalImageType, InternalImageType >
  FastMarchingFilterType;
```

Next we construct one filter of this class using the `New()` method.

```
FastMarchingFilterType::Pointer  fastMarching = FastMarchingFilterType::New();
```

In the following lines we instantiate the type of the ShapeDetectionLevelSetImageFilter and create an object of this type using the `New()` method.

```
typedef  itk::ShapeDetectionLevelSetImageFilter< InternalImageType,
                              InternalImageType >    ShapeDetectionFilterType;
ShapeDetectionFilterType::Pointer
  shapeDetection = ShapeDetectionFilterType::New();
```

The filters are now connected in a pipeline indicated in Figure 9.18 with the following code.

```
smoothing->SetInput( reader->GetOutput() );
gradientMagnitude->SetInput( smoothing->GetOutput() );
sigmoid->SetInput( gradientMagnitude->GetOutput() );

shapeDetection->SetInput( fastMarching->GetOutput() );
shapeDetection->SetFeatureImage( sigmoid->GetOutput() );

thresholder->SetInput( shapeDetection->GetOutput() );

writer->SetInput( thresholder->GetOutput() );
```

The CurvatureAnisotropicDiffusionImageFilter requires a couple of parameters to be defined. The following are typical values for $2D$ images. However they may have to be adjusted depending on the amount of noise present in the input image. This filter has been discussed in Section 6.7.3.

```
smoothing->SetTimeStep( 0.125 );
smoothing->SetNumberOfIterations(  5 );
smoothing->SetConductanceParameter( 9.0 );
```

The GradientMagnitudeRecursiveGaussianImageFilter performs the equivalent of a convolution with a Gaussian kernel followed by a derivative operator. The sigma of this Gaussian can be used to control the range of influence of the image edges. This filter has been discussed in Section 6.4.2

```
gradientMagnitude->SetSigma(  sigma  );
```

The SigmoidImageFilter requires two parameters that define the linear transformation to be applied to the sigmoid argument. These parameters have been discussed in Sections 6.3.2 and 9.3.1.

```
sigmoid->SetAlpha( alpha );
sigmoid->SetBeta(  beta  );
```

The FastMarchingImageFilter requires the user to provide a seed point from which the level set will be generated. The user can actually pass not only one seed point but a set of them. Note the FastMarchingImageFilter is used here only as a helper in the determination of an initial level set. We could have used the `itk::DanielssonDistanceMapImageFilter` in the same way.

The seeds are stored in a container. The type of this container is defined as `NodeContainer` among the FastMarchingImageFilter traits.

```
typedef FastMarchingFilterType::NodeContainer           NodeContainer;
typedef FastMarchingFilterType::NodeType                NodeType;
NodeContainer::Pointer seeds = NodeContainer::New();
```

Nodes are created as stack variables and initialized with a value and an `itk::Index` position. Note that we assign the negative of the value of the user-provided distance to the unique node of the seeds passed to the FastMarchingImageFilter. In this way, the value will increment as the front is propagated, until it reaches the zero value corresponding to the contour. After this, the front will continue propagating until it fills up the entire image. The initial distance is taken from the command line arguments. The rule of thumb for the user is to select this value as the distance from the seed points at which the initial contour should be.

```
NodeType node;
const double seedValue = - initialDistance;

node.SetValue( seedValue );
node.SetIndex( seedPosition );
```

The list of nodes is initialized and then every node is inserted using `InsertElement()`.

```
  seeds->Initialize();
  seeds->InsertElement( 0, node );
```

The set of seed nodes is now passed to the FastMarchingImageFilter with the method `SetTrialPoints()`.

```
  fastMarching->SetTrialPoints(  seeds  );
```

Since the FastMarchingImageFilter is used here only as a distance map generator, it does not require a speed image as input. Instead, the constant value 1.0 is passed using the `SetSpeedConstant()` method.

```
  fastMarching->SetSpeedConstant( 1.0 );
```

The FastMarchingImageFilter requires the user to specify the size of the image to be produced as output. This is done using the `SetOutputSize()`. Note that the size is obtained here from the output image of the smoothing filter. The size of this image is valid only after the `Update()` methods of this filter have been called directly or indirectly.

```
  fastMarching->SetOutputSize(
           reader->GetOutput()->GetBufferedRegion().GetSize() );
```

ShapeDetectionLevelSetImageFilter provides two parameters to control the competition between the propagation or expansion term and the curvature smoothing term. The methods `SetPropagationScaling()` and `SetCurvatureScaling()` defines the relative weighting between the two terms. In this example, we will set the propagation scaling to one and let the curvature scaling be an input argument. The larger the the curvature scaling parameter the smoother the resulting segmentation. However, the curvature scaling parameter should not be set too large, as it will draw the contour away from the shape boundaries.

```
  shapeDetection->SetPropagationScaling(  propagationScaling );
  shapeDetection->SetCurvatureScaling( curvatureScaling );
```

Once activated, the level set evolution will stop if the convergence criteria or the maximum number of iterations is reached. The convergence criteria are defined in terms of the root mean squared (RMS) change in the level set function. The evolution is said to have converged if the RMS change is below a user-specified threshold. In a real application, it is desirable to couple the evolution of the zero set to a visualization module, allowing the user to follow the evolution of the zero set. With this feedback, the user may decide when to stop the algorithm before the zero set leaks through the regions of low gradient in the contour of the anatomical structure to be segmented.

Structure	Seed Index	Distance	σ	α	β	Output Image
Left Ventricle	$(81,114)$	5.0	1.0	-0.5	3.0	First in Figure 9.20
Right Ventricle	$(99,114)$	5.0	1.0	-0.5	3.0	Second in Figure 9.20
White matter	$(56,92)$	5.0	1.0	-0.3	2.0	Third in Figure 9.20
Gray matter	$(40,90)$	5.0	0.5	-0.3	2.0	Fourth in Figure 9.20

Table 9.4: Parameters used for segmenting some brain structures shown in Figure 9.19 using the filter ShapeDetectionLevelSetFilter. All of them used a propagation scaling of 1.0 and curvature scaling of 0.05.

```
shapeDetection->SetMaximumRMSError( 0.02 );
shapeDetection->SetNumberOfIterations( 800 );
```

The invocation of the `Update()` method on the writer triggers the execution of the pipeline. As usual, the call is placed in a `try/catch` block should any errors occur or exceptions be thrown.

```
try
  {
  writer->Update();
  }
catch( itk::ExceptionObject & excep )
  {
  std::cerr << "Exception caught !" << std::endl;
  std::cerr << excep << std::endl;
  }
```

Let's now run this example using as input the image `BrainProtonDensitySlice.png` provided in the directory `Examples/Data`. We can easily segment the major anatomical structures by providing seeds in the appropriate locations. Table 9.4 presents the parameters used for some structures. For all of the examples illustrated in this table, the propagation scaling was set to 1.0, and the curvature scaling set to 0.05.

Figure 9.19 presents the intermediate outputs of the pipeline illustrated in Figure 9.18. They are from left to right: the output of the anisotropic diffusion filter, the gradient magnitude of the smoothed image and the sigmoid of the gradient magnitude which is finally used as the edge potential for the ShapeDetectionLevelSetImageFilter.

Notice that in Figure 9.20 the segmented shapes are rounder than in Figure 9.17 due to the effects of the curvature term in the driving equation. As with the previous example, segmentation of the gray matter is still problematic.

A larger number of iterations is reguired for segmenting large structures since it takes longer for the front to propagate and cover the structure. This drawback can be easily mitigated by setting many seed points in the initialization of the FastMarchingImageFilter. This will generate an initial level set much closer in shape to the object to be segmented and hence require fewer iterations to fill and reach the edges of the anatomical structure.

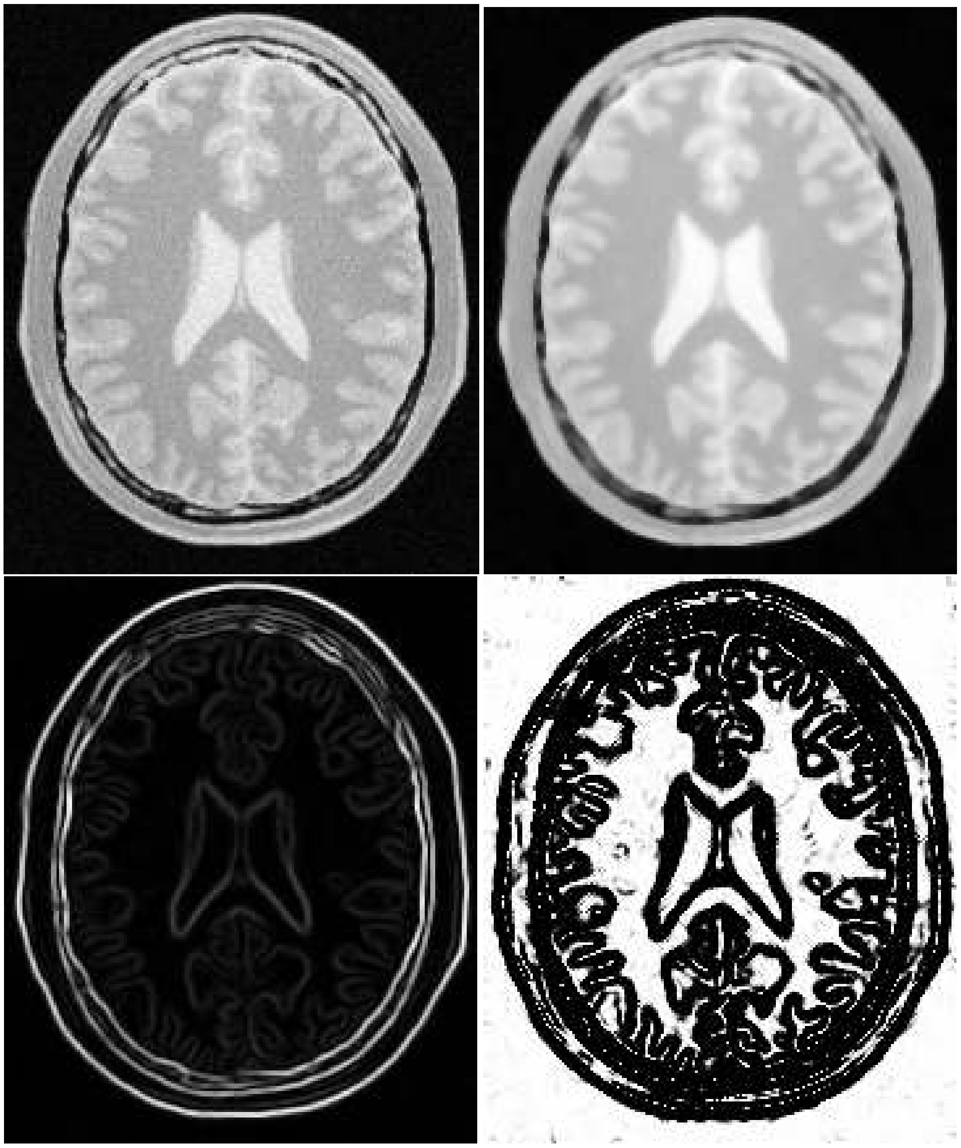


Figure 9.19: Images generated by the segmentation process based on the ShapeDetectionLevelSetImageFilter. From left to right and top to bottom: input image to be segmented, image smoothed with an edge-preserving smoothing filter, gradient magnitude of the smoothed image, sigmoid of the gradient magnitude. This last image, the sigmoid, is used to compute the speed term for the front propagation.

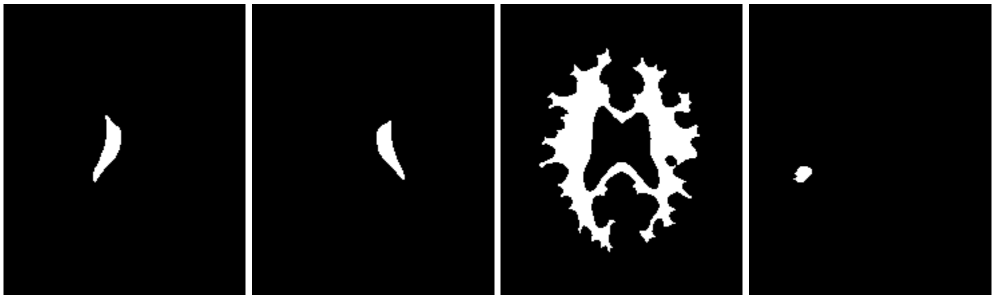


Figure 9.20: Images generated by the segmentation process based on the ShapeDetectionLevelSetImageFilter. From left to right: segmentation of the left ventricle, segmentation of the right ventricle, segmentation of the white matter, attempt of segmentation of the gray matter.

9.3.3 Geodesic Active Contours Segmentation

The source code for this section can be found in the file
`Examples/Segmentation/GeodesicActiveContourImageFilter.cxx`.

The use of the `itk::GeodesicActiveContourLevelSetImageFilter` is illustrated in the following example. The implementation of this filter in ITK is based on the paper by Caselles [14]. This implementation extends the functionality of the `itk::ShapeDetectionLevelSetImageFilter` by the addition of a third advection term which attracts the level set to the object boundaries.

GeodesicActiveContourLevelSetImageFilter expects two inputs. The first is an initial level set in the form of an `itk::Image`. The second input is a feature image. For this algorithm, the feature image is an edge potential image that basically follows the same rules used for the ShapeDetectionLevelSetImageFilter discussed in Section 9.3.2. The configuration of this example is quite similar to the example on the use of the ShapeDetectionLevelSetImageFilter. We omit most of the redundant description. A look at the code will reveal the great degree of similarity between both examples.

Figure 9.21 shows the major components involved in the application of the GeodesicActiveContourLevelSetImageFilter to a segmentation task. This pipeline is quite similar to the one used by the ShapeDetectionLevelSetImageFilter in section 9.3.2.

The pipeline involves a first stage of smoothing using the `itk::CurvatureAnisotropicDiffusionImageFilter`. The smoothed image is passed as the input to the `itk::GradientMagnitudeRecursiveGaussianImageFilter` and then to the `itk::SigmoidImageFilter` in order to produce the edge potential image. A set of user-provided seeds is passed to a `itk::FastMarchingImageFilter` in order to compute the distance map. A constant value is subtracted from this map in order to obtain a level set in which the *zero set* represents the initial contour. This level set is also passed as input to the GeodesicActiveContourLevelSetImageFilter.

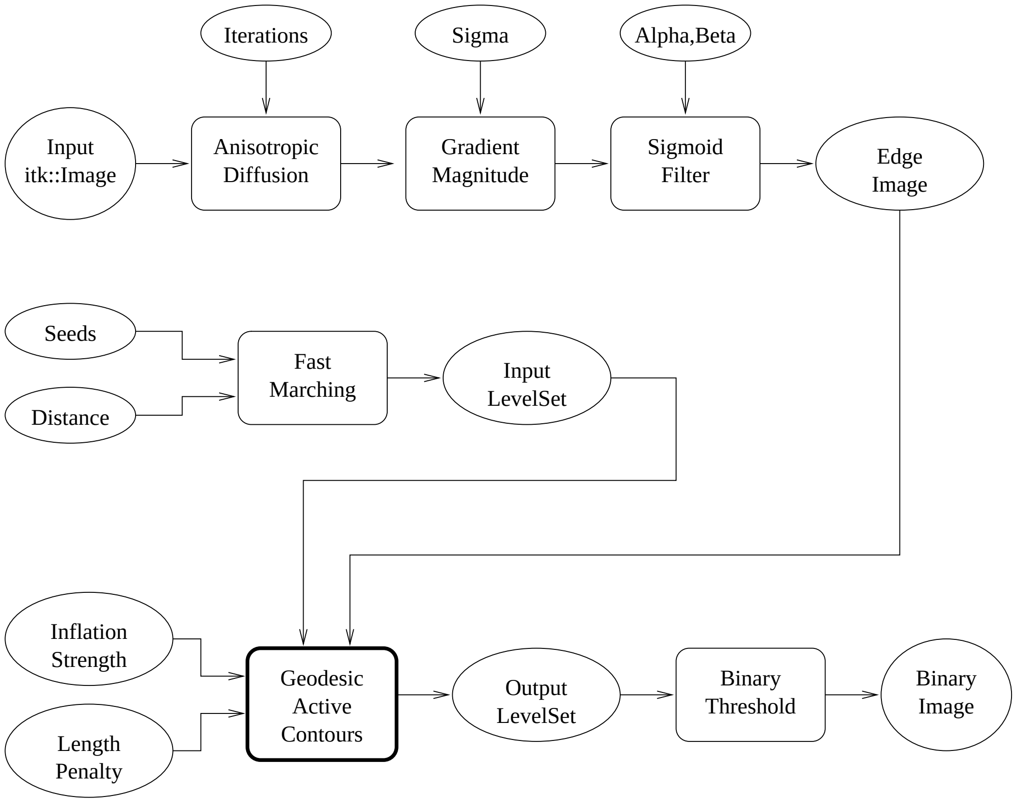


Figure 9.21: Collaboration diagram for the GeodesicActiveContourLevelSetImageFilter applied to a segmentation task.

Finally, the level set generated by the GeodesicActiveContourLevelSetImageFilter is passed to a itk::BinaryThresholdImageFilter in order to produce a binary mask representing the segmented object.

Let's start by including the headers of the main filters involved in the preprocessing.

```
#include "itkImage.h"
#include "itkGeodesicActiveContourLevelSetImageFilter.h"
```

We now define the image type using a particular pixel type and dimension. In this case the `float` type is used for the pixels due to the requirements of the smoothing filter.

```
  typedef   float           InternalPixelType;
  const     unsigned int    Dimension = 2;
  typedef itk::Image< InternalPixelType, Dimension >  InternalImageType;
```

In the following lines we instantiate the type of the GeodesicActiveContourLevelSetImageFilter and create an object of this type using the `New()` method.

```
  typedef  itk::GeodesicActiveContourLevelSetImageFilter< InternalImageType,
                InternalImageType >    GeodesicActiveContourFilterType;
  GeodesicActiveContourFilterType::Pointer geodesicActiveContour =
                                     GeodesicActiveContourFilterType::New();
```

For the GeodesicActiveContourLevelSetImageFilter, scaling parameters are used to trade off between the propagation (inflation), the curvature (smoothing) and the advection terms. These parameters are set using methods `SetPropagationScaling()`, `SetCurvatureScaling()` and `SetAdvectionScaling()`. In this example, we will set the curvature and advection scales to one and let the propagation scale be a command-line argument.

```
  geodesicActiveContour->SetPropagationScaling( propagationScaling );
  geodesicActiveContour->SetCurvatureScaling( 1.0 );
  geodesicActiveContour->SetAdvectionScaling( 1.0 );
```

The filters are now connected in a pipeline indicated in Figure 9.21 using the following lines:

```
  smoothing->SetInput( reader->GetOutput() );
  gradientMagnitude->SetInput( smoothing->GetOutput() );
  sigmoid->SetInput( gradientMagnitude->GetOutput() );

  geodesicActiveContour->SetInput(  fastMarching->GetOutput() );
  geodesicActiveContour->SetFeatureImage( sigmoid->GetOutput() );

  thresholder->SetInput( geodesicActiveContour->GetOutput() );
  writer->SetInput( thresholder->GetOutput() );
```

Structure	Seed Index	Distance	σ	α	β	Propag.	Output Image
Left Ventricle	$(81,114)$	5.0	1.0	-0.5	3.0	2.0	First
Right Ventricle	$(99,114)$	5.0	1.0	-0.5	3.0	2.0	Second
White matter	$(56,92)$	5.0	1.0	-0.3	2.0	10.0	Third
Gray matter	$(40,90)$	5.0	0.5	-0.3	2.0	10.0	Fourth

Table 9.5: Parameters used for segmenting some brain structures shown in Figure 9.23 using the filter GeodesicActiveContourLevelSetImageFilter.

The invocation of the `Update()` method on the writer triggers the execution of the pipeline. As usual, the call is placed in a `try/catch` block should any errors occur or exceptions be thrown.

```
try
  {
  writer->Update();
  }
catch( itk::ExceptionObject & excep )
  {
  std::cerr << "Exception caught !" << std::endl;
  std::cerr << excep << std::endl;
  }
```

Let's now run this example using as input the image `BrainProtonDensitySlice.png` provided in the directory `Examples/Data`. We can easily segment the major anatomical structures by providing seeds in the appropriate locations. Table 9.5 presents the parameters used for some structures.

Figure 9.22 presents the intermediate outputs of the pipeline illustrated in Figure 9.21. They are from left to right: the output of the anisotropic diffusion filter, the gradient magnitude of the smoothed image and the sigmoid of the gradient magnitude which is finally used as the edge potential for the GeodesicActiveContourLevelSetImageFilter.

Segmentations of the main brain structures are presented in Figure 9.23. The results are quite similar to those obtained with the ShapeDetectionLevelSetImageFilter in Section 9.3.2.

Note that a relatively larger propagation scaling value was required to segment the white matter. This is due to two factors: the lower contrast at the border of the white matter and the complex shape of the structure. Unfortunately the optimal value of these scaling parameters can only be determined by experimentation. In a real application we could imagine an interactive mechanism by which a user supervises the contour evolution and adjusts these parameters accordingly.

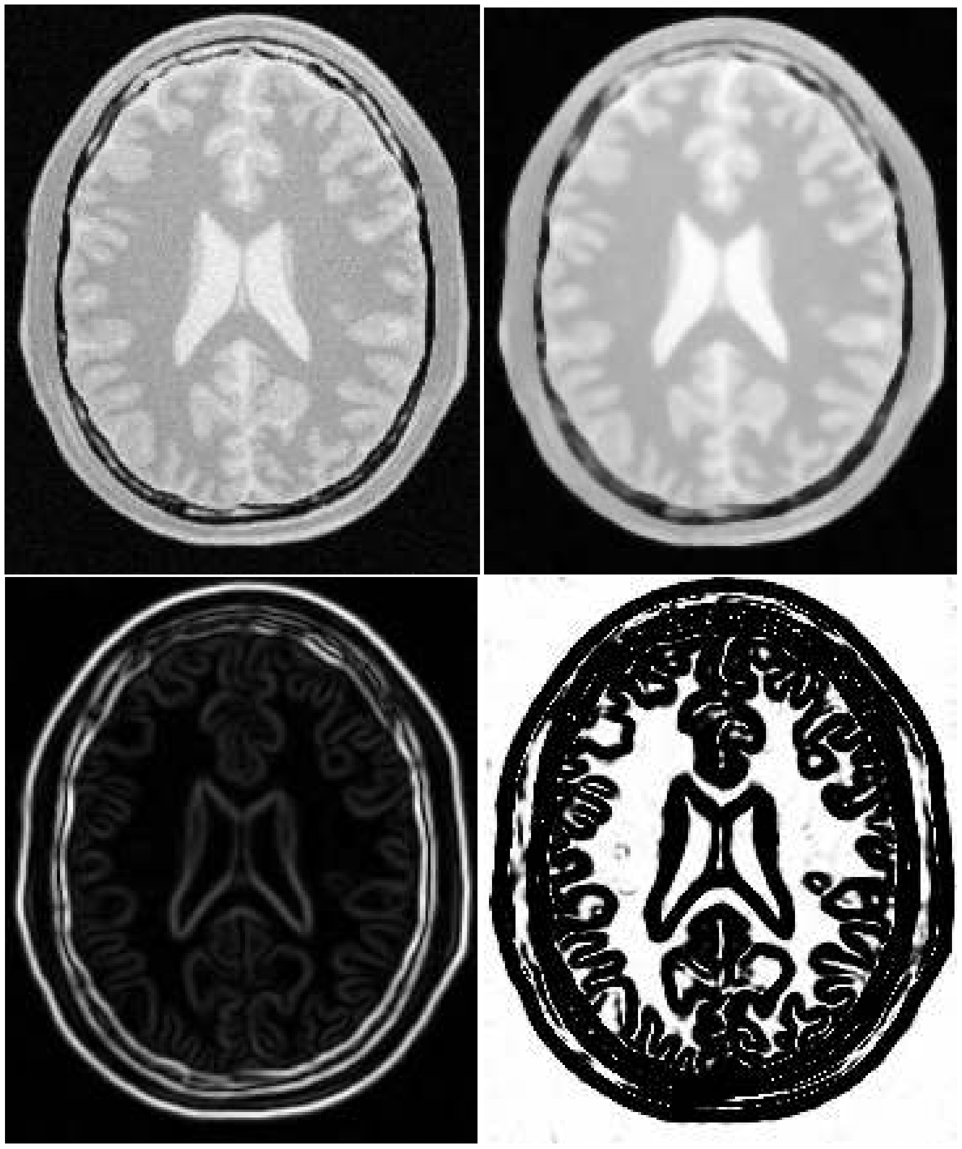


Figure 9.22: Images generated by the segmentation process based on the GeodesicActiveContourLevelSetImageFilter. From left to right and top to bottom: input image to be segmented, image smoothed with an edge-preserving smoothing filter, gradient magnitude of the smoothed image, sigmoid of the gradient magnitude. This last image, the sigmoid, is used to compute the speed term for the front propagation.

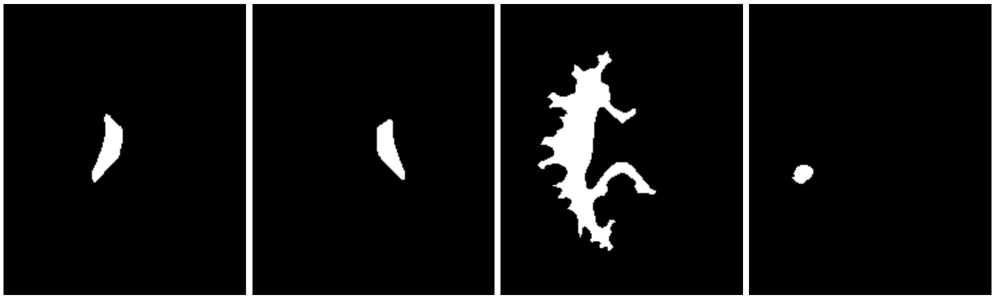


Figure 9.23: Images generated by the segmentation process based on the GeodesicActiveContourImageFilter. From left to right: segmentation of the left ventricle, segmentation of the right ventricle, segmentation of the white matter, attempt of segmentation of the gray matter.

9.3.4 Threshold Level Set Segmentation

The source code for this section can be found in the file
`Examples/Segmentation/ThresholdSegmentationLevelSetImageFilter.cxx`.

The `itk::ThresholdSegmentationLevelSetImageFilter` is an extension of the threshold connected-component segmentation to the level set framework. The goal is to define a range of intensity values that classify the tissue type of interest and then base the propagation term on the level set equation for that intensity range. Using the level set approach, the smoothness of the evolving surface can be constrained to prevent some of the "leaking" that is common in connected-component schemes.

The propagation term P from Equation 9.3 is calculated from the `FeatureImage` input g with `UpperThreshold` U and `LowerThreshold` L according to the following formula.

$$P(\mathbf{x}) = \begin{cases} g(\mathbf{x}) - L & \text{if } g(\mathbf{x}) < (U-L)/2 + L \\ U - g(\mathbf{x}) & \text{otherwise} \end{cases} \tag{9.4}$$

Figure 9.25 illustrates the propagation term function. Intensity values in g between L and H yield positive values in P, while outside intensities yield negative values in P.

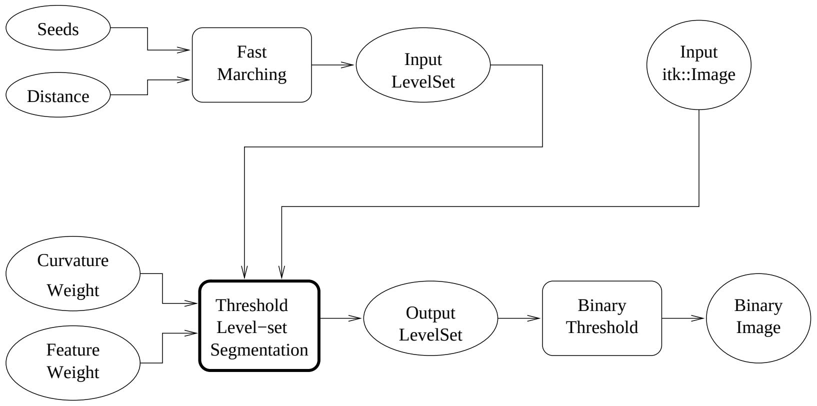


Figure 9.24: Collaboration diagram for the ThresholdSegmentationLevelSetImageFilter applied to a segmentation task.

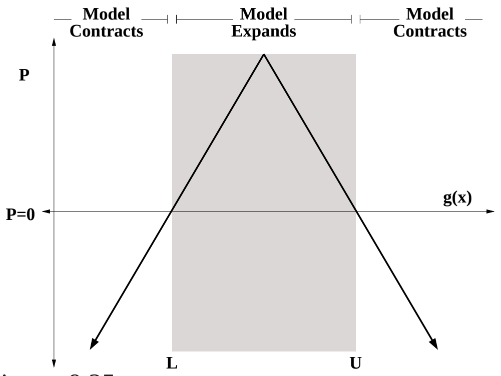


Figure 9.25: Propagation term for threshold-based level set segmentation. From Equation 9.4.

The threshold segmentation filter expects two inputs. The first is an initial level set in the form of an itk::Image. The second input is the feature image g. For many applications, this filter requires little or no preprocessing of its input. Smoothing the input image is not usually required to produce reasonable solutions, though it may still be warranted in some cases.

Figure 9.24 shows how the image processing pipeline is constructed. The initial surface is generated using the fast marching filter. The output of the segmentation filter is passed to a itk::BinaryThresholdImageFilter to create a binary representation of the segmented object. Let's start by including the appropriate header file.

```
#include "itkThresholdSegmentationLevelSetImageFilter.h"
```

We define the image type using a particular pixel type and dimension. In this case we will use 2D `float` images.

```
  typedef   float           InternalPixelType;
  const     unsigned int    Dimension = 2;
  typedef itk::Image< InternalPixelType, Dimension >  InternalImageType;
```

The following lines instantiate a ThresholdSegmentationLevelSetImageFilter using the `New()` method.

```
typedef  itk::ThresholdSegmentationLevelSetImageFilter< InternalImageType,
  InternalImageType > ThresholdSegmentationLevelSetImageFilterType;
ThresholdSegmentationLevelSetImageFilterType::Pointer thresholdSegmentation =
  ThresholdSegmentationLevelSetImageFilterType::New();
```

For the ThresholdSegmentationLevelSetImageFilter, scaling parameters are used to balance the influence of the propagation (inflation) and the curvature (surface smoothing) terms from Equation 9.3. The advection term is not used in this filter. Set the terms with methods `SetPropagationScaling()` and `SetCurvatureScaling()`. Both terms are set to 1.0 in this example.

```
thresholdSegmentation->SetPropagationScaling( 1.0 );
if ( argc > 8 )
  {
  thresholdSegmentation->SetCurvatureScaling( atof(argv[8]) );
  }
else
  {
  thresholdSegmentation->SetCurvatureScaling( 1.0 );
  }
```

The convergence criteria `MaximumRMSError` and `MaximumIterations` are set as in previous examples. We now set the upper and lower threshold values U and L, and the isosurface value to use in the initial model.

```
thresholdSegmentation->SetUpperThreshold( ::atof(argv[7]) );
thresholdSegmentation->SetLowerThreshold( ::atof(argv[6]) );
thresholdSegmentation->SetIsoSurfaceValue(0.0);
```

The filters are now connected in a pipeline indicated in Figure 9.24. Remember that before calling `Update()` on the file writer object, the fast marching filter must be initialized with the seed points and the output from the reader object. See previous examples and the source code for this section for details.

```
thresholdSegmentation->SetInput( fastMarching->GetOutput() );
thresholdSegmentation->SetFeatureImage( reader->GetOutput() );
thresholder->SetInput( thresholdSegmentation->GetOutput() );
writer->SetInput( thresholder->GetOutput() );
```

Invoking the `Update()` method on the writer triggers the execution of the pipeline. As usual, the call is placed in a `try/catch` block should any errors occur or exceptions be thrown.

```
try
```

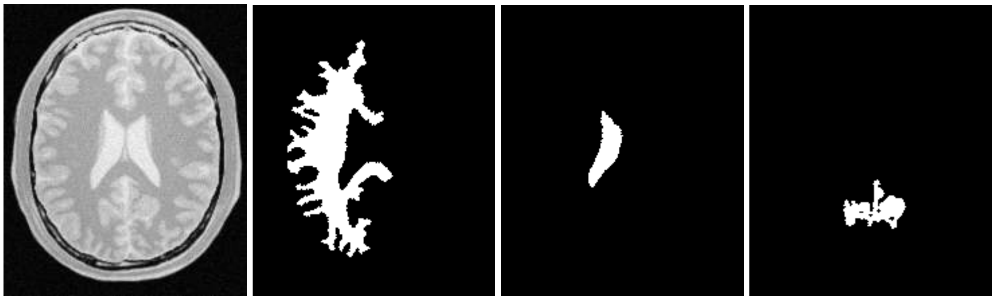


Figure 9.26: Images generated by the segmentation process based on the ThresholdSegmentationLevelSetImageFilter. From left to right: segmentation of the left ventricle, segmentation of the right ventricle, segmentation of the white matter, attempt of segmentation of the gray matter. The parameters used in this segmentations are presented in Table 9.6

Structure	Seed Index	Lower	Upper	Output Image
White matter	$(60,116)$	150	180	Second from left
Ventricle	$(81,112)$	210	250	Third from left
Gray matter	$(107,69)$	180	210	Fourth from left

Table 9.6: Segmentation results using the ThresholdSegmentationLevelSetImageFilter for various seed points. The resulting images are shown in Figure 9.26

.

```
    {
    reader->Update();
    fastMarching->SetOutputSize(
      reader->GetOutput()->GetBufferedRegion().GetSize() );
    writer->Update();
    }
  catch( itk::ExceptionObject & excep )
    {
    std::cerr << "Exception caught !" << std::endl;
    std::cerr << excep << std::endl;
    }
```

Let's run this application with the same data and parameters as the example given for `itk::ConnectedThresholdImageFilter` in Section 9.1.1. We will use a value of 5 as the initial distance of the surface from the seed points. The algorithm is relatively insensitive to this initialization. Compare the results in Figure 9.26 with those in Figure 9.1. Notice how the smoothness constraint on the surface prevents leakage of the segmentation into both ventricles, but also localizes the segmentation to a smaller portion of the gray matter.

9.3.5 Canny-Edge Level Set Segmentation

The source code for this section can be found in the file
`Examples/Segmentation/CannySegmentationLevelSetImageFilter.cxx`.

The `itk::CannySegmentationLevelSetImageFilter` defines a speed term that minimizes distance to the Canny edges in an image. The initial level set model moves through a gradient advection field until it locks onto those edges. This filter is more suitable for refining existing segmentations than as a region-growing algorithm.

The two terms defined for the CannySegmentationLevelSetImageFilter are the advection term and the propagation term from Equation 9.3. The advection term is constructed by minimizing the squared distance transform from the Canny edges.

$$\min \int D^2 \Rightarrow D \nabla D \tag{9.5}$$

where the distance transform D is calculated using a `itk::DanielssonDistanceMapImageFilter` applied to the output of the `itk::CannyEdgeDetectionImageFilter`.

For cases in which some surface expansion is to be allowed, a non-zero value may be set for the propagation term. The propagation term is simply D. As with all ITK level set segmentation filters, the curvature term controls the smoothness of the surface.

CannySegmentationLevelSetImageFilter expects two inputs. The first is an initial level set in the form of an `itk::Image`. The second input is the feature image g from which propagation and advection terms are calculated. It is generally a good idea to do some preprocessing of the feature image to remove noise.

Figure 9.27 shows how the image processing pipeline is constructed. We read two images: the image to segment and the image that contains the initial implicit surface. The goal is to refine the initial model from the second input and not to grow a new segmentation from seed points. The `feature` image is preprocessed with a few iterations of an anisotropic diffusion filter.

Let's start by including the appropriate header file.

```
#include "itkCannySegmentationLevelSetImageFilter.h"
#include "itkGradientAnisotropicDiffusionImageFilter.h"
```

We define the image type using a particular pixel type and dimension. In this case we will use 2D `float` images.

```
  typedef   float           InternalPixelType;
  const     unsigned int    Dimension = 2;
  typedef itk::Image< InternalPixelType, Dimension >  InternalImageType;
```

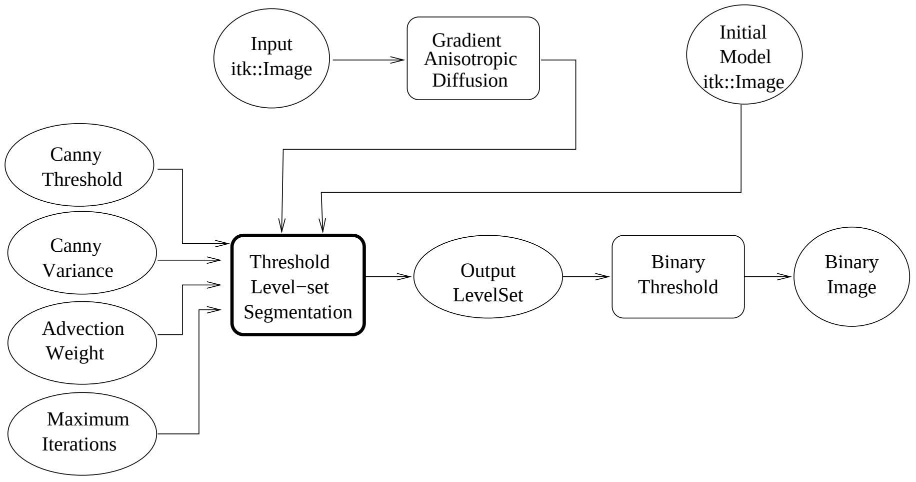


Figure 9.27: Collaboration diagram for the CannySegmentationLevelSetImageFilter applied to a segmentation task.

The input image will be processed with a few iterations of feature-preserving diffusion. We create a filter and set the appropriate parameters.

```
typedef itk::GradientAnisotropicDiffusionImageFilter< InternalImageType,
  InternalImageType> DiffusionFilterType;
DiffusionFilterType::Pointer diffusion = DiffusionFilterType::New();
diffusion->SetNumberOfIterations(5);
diffusion->SetTimeStep(0.125);
diffusion->SetConductanceParameter(1.0);
```

The following lines define and instantiate a CannySegmentationLevelSetImageFilter.

```
typedef  itk::CannySegmentationLevelSetImageFilter< InternalImageType,
              InternalImageType > CannySegmentationLevelSetImageFilterType;
CannySegmentationLevelSetImageFilterType::Pointer cannySegmentation =
              CannySegmentationLevelSetImageFilterType::New();
```

As with the other ITK level set segmentation filters, the terms of the CannySegmentation-LevelSetImageFilter level set equation can be weighted by scalars. For this application we will modify the relative weight of the advection term. The propagation and curvature term weights are set to their defaults of 0 and 1, respectively.

```
cannySegmentation->SetAdvectionScaling( ::atof(argv[6]) );
cannySegmentation->SetCurvatureScaling( 1.0 );
cannySegmentation->SetPropagationScaling( 0.0 );
```

The maximum number of iterations is specified from the command line. It may not be desirable in some applications to run the filter to convergence. Only a few iterations may be required.

```
cannySegmentation->SetMaximumRMSError( 0.01 );
cannySegmentation->SetNumberOfIterations( ::atoi(argv[8]) );
```

There are two important parameters in the CannySegmentationLevelSetImageFilter to control the behavior of the Canny edge detection. The *variance* parameter controls the amount of Gaussian smoothing on the input image. The *threshold* parameter indicates the lowest allowed value in the output image. Thresholding is used to suppress Canny edges whose gradient magnitudes fall below a certain value.

```
cannySegmentation->SetThreshold( ::atof(argv[4]) );
cannySegmentation->SetVariance(  ::atof(argv[5]) );
```

Finally, it is very important to specify the isovalue of the surface in the initial model input image. In a binary image, for example, the isosurface is found midway between the foreground and background values.

```
cannySegmentation->SetIsoSurfaceValue( ::atof(argv[7]) );
```

The filters are now connected in a pipeline indicated in Figure 9.27.

```
diffusion->SetInput( reader1->GetOutput() );
cannySegmentation->SetInput( reader2->GetOutput() );
cannySegmentation->SetFeatureImage( diffusion->GetOutput() );
thresholder->SetInput( cannySegmentation->GetOutput() );
writer->SetInput( thresholder->GetOutput() );
```

Invoking the `Update()` method on the writer triggers the execution of the pipeline. As usual, the call is placed in a `try/catch` block to handle any exceptions that may be thrown.

```
try
  {
  writer->Update();
  }
catch( itk::ExceptionObject & excep )
  {
  std::cerr << "Exception caught !" << std::endl;
  std::cerr << excep << std::endl;
  }
```

We can use this filter to make some subtle refinements to the ventricle segmentation from the previous example that used the `itk::ThresholdSegmentationLevelSetImageFilter`.

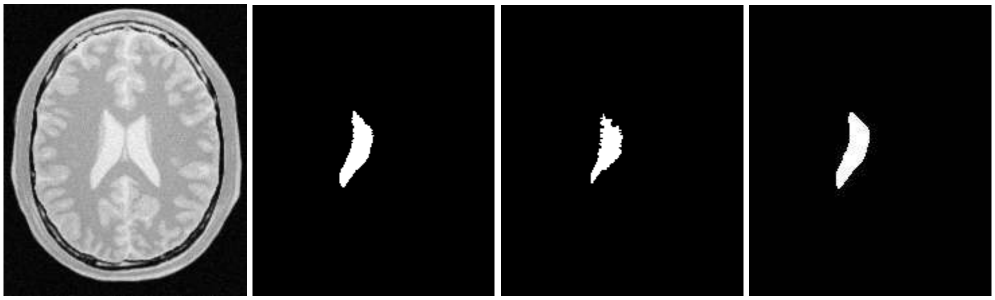


Figure 9.28: Results of applying the CannySegmentationLevelSetImageFilter to a prior ventricle segmentation. Shown from left to right are the original image, the prior segmentation of the ventricle from Figure 9.26, 15 iterations of the CannySegmentationLevelSetImageFilter, and the CannySegmentationLevelSetImageFilter run to convergence.

The application was run using `Examples/Data/BrainProtonDensitySlice.png` and `Examples/Data/VentricleModel.png` as inputs, a `threshold` of 7.0, `variance` of 0.1, `advection weight` of 10.0, and an initial isosurface value of 127.5. One case was run for 15 iterations and the second was run to convergence. Compare the results in the two rightmost images of Figure 9.28 with the ventricle segmentation from Figure 9.26 shown in the middle. Jagged edges are straightened and the small spur at the upper right-hand side of the mask has been removed.

The free parameters of this filter can be adjusted to achieve a wide range of shape variations from the original model. Finding the right parameters for your particular application is usually a process of trial and error. As with most ITK level set segmentation filters, examining the propagation (speed) and advection images can help the process of tuning parameters. These images are available using `Set/Get` methods from the filter after it has been updated.

In some cases it is interesting to take a direct look at the speed image used internally by this filter. This may help for setting the correct parameters for driving the segmentation. In order to obtain such speed image, the method `GenerateSpeedImage()` should be invoked first. Then we can recover the speed image with the `GetSpeedImage()` method as illustrated in the following lines.

```
cannySegmentation->GenerateSpeedImage();

typedef CannySegmentationLevelSetImageFilterType::SpeedImageType SpeedImageType;
typedef itk::ImageFileWriter<SpeedImageType>   SpeedWriterType;
SpeedWriterType::Pointer speedWriter = SpeedWriterType::New();

speedWriter->SetInput( cannySegmentation->GetSpeedImage() );
```

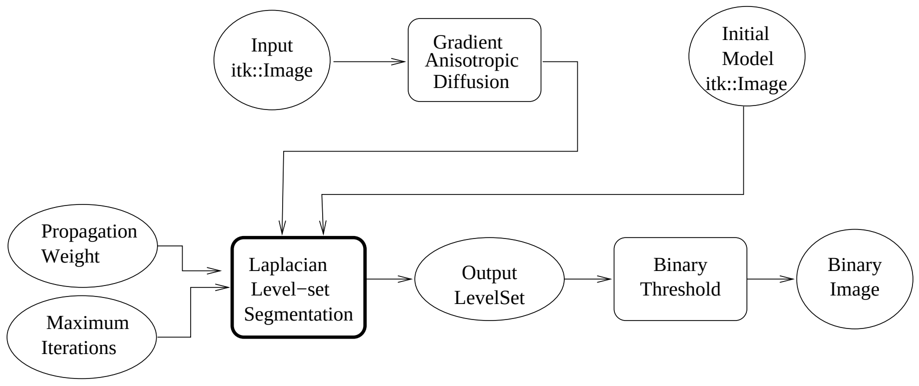


Figure 9.29: An image processing pipeline using LaplacianSegmentationLevelSetImageFilter for segmentation.

9.3.6 Laplacian Level Set Segmentation

The source code for this section can be found in the file
`Examples/Segmentation/LaplacianSegmentationLevelSetImageFilter.cxx`.

The `itk::LaplacianSegmentationLevelSetImageFilter` defines a speed term based on second derivative features in the image. The speed term is calculated as the Laplacian of the image values. The goal is to attract the evolving level set surface to local zero-crossings in the Laplacian image. Like `itk::CannySegmentationLevelSetImageFilter`, this filter is more suitable for refining existing segmentations than as a stand-alone, region growing algorithm. It is possible to perform region growing segmentation, but be aware that the growing surface may tend to become "stuck" at local edges.

The propagation (speed) term for the LaplacianSegmentationLevelSetImageFilter is constructed by applying the `itk::LaplacianImageFilter` to the input feature image. One nice property of using the Laplacian is that there are no free parameters in the calculation.

LaplacianSegmentationLevelSetImageFilter expects two inputs. The first is an initial level set in the form of an `itk::Image`. The second input is the feature image g from which the propagation term is calculated (see Equation 9.3). Because the filter performs a second derivative calculation, it is generally a good idea to do some preprocessing of the feature image to remove noise.

Figure 9.29 shows how the image processing pipeline is constructed. We read two images: the image to segment and the image that contains the initial implicit surface. The goal is to refine the initial model from the second input to better match the structure represented by the initial implicit surface (a prior segmentation). The `feature` image is preprocessed using an anisotropic diffusion filter.

Let's start by including the appropriate header files.

```
#include "itkLaplacianSegmentationLevelSetImageFilter.h"
#include "itkGradientAnisotropicDiffusionImageFilter.h"
```

We define the image type using a particular pixel type and dimension. In this case we will use 2D `float` images.

```
  typedef   float           InternalPixelType;
  const     unsigned int    Dimension = 2;
  typedef itk::Image< InternalPixelType, Dimension >  InternalImageType;
```

The input image will be processed with a few iterations of feature-preserving diffusion. We create a filter and set the parameters. The number of iterations and the conductance parameter are taken from the command line.

```
  typedef itk::GradientAnisotropicDiffusionImageFilter< InternalImageType,
    InternalImageType> DiffusionFilterType;
  DiffusionFilterType::Pointer diffusion = DiffusionFilterType::New();
  diffusion->SetNumberOfIterations( atoi(argv[4]) );
  diffusion->SetTimeStep(0.125);
  diffusion->SetConductanceParameter( atof(argv[5]) );
```

The following lines define and instantiate a LaplacianSegmentationLevelSetImageFilter.

```
  typedef itk::LaplacianSegmentationLevelSetImageFilter< InternalImageType,
             InternalImageType > LaplacianSegmentationLevelSetImageFilterType;
  LaplacianSegmentationLevelSetImageFilterType::Pointer laplacianSegmentation =
              LaplacianSegmentationLevelSetImageFilterType::New();
```

As with the other ITK level set segmentation filters, the terms of the LaplacianSegmentation-LevelSetImageFilter level set equation can be weighted by scalars. For this application we will modify the relative weight of the propagation term. The curvature term weight is set to its default of 1. The advection term is not used in this filter.

```
  laplacianSegmentation->SetCurvatureScaling( 1.0 );
  laplacianSegmentation->SetPropagationScaling( ::atof(argv[6]) );
```

The maximum number of iterations is set from the command line. It may not be desirable in some applications to run the filter to convergence. Only a few iterations may be required.

```
  laplacianSegmentation->SetMaximumRMSError( 0.002 );
  laplacianSegmentation->SetNumberOfIterations( ::atoi(argv[8]) );
```

Finally, it is very important to specify the isovalue of the surface in the initial model input image. In a binary image, for example, the isosurface is found midway between the foreground and background values.

```
  laplacianSegmentation->SetIsoSurfaceValue( ::atof(argv[7]) );
```

The filters are now connected in a pipeline indicated in Figure 9.29.

```
  diffusion->SetInput( reader1->GetOutput() );
  laplacianSegmentation->SetInput( reader2->GetOutput() );
  laplacianSegmentation->SetFeatureImage( diffusion->GetOutput() );
  thresholder->SetInput( laplacianSegmentation->GetOutput() );
  writer->SetInput( thresholder->GetOutput() );
```

Invoking the `Update()` method on the writer triggers the execution of the pipeline. As usual, the call is placed in a `try/catch` block to handle any exceptions that may be thrown.

```
  try
    {
    writer->Update();
    }
  catch( itk::ExceptionObject & excep )
    {
    std::cerr << "Exception caught !" << std::endl;
    std::cerr << excep << std::endl;
    }
```

We can use this filter to make some subtle refinements to the ventricle segmentation from the example using the filter `itk::ThresholdSegmentationLevelSetImageFilter`. This application was run using `Examples/Data/BrainProtonDensitySlice.png` and `Examples/Data/VentricleModel.png` as inputs. We used 10 iterations of the diffusion filter with a conductance of 2.0. The propagation scaling was set to 1.0 and the filter was run until convergence. Compare the results in the rightmost images of Figure 9.30 with the ventricle segmentation from Figure 9.26 shown in the middle. Jagged edges are straightened and the small spur at the upper right-hand side of the mask has been removed.

9.3.7 Geodesic Active Contours Segmentation With Shape Guidance

The source code for this section can be found in the file
`Examples/Segmentation/GeodesicActiveContourShapePriorLevelSetImageFilter.cxx`.

In medical imaging applications, the general shape, location and orientation of an anatomical structure of interest is typically known *a priori*. This information can be used to aid the segmentation process especially when image contrast is low or when the object boundary is not distinct.

In [48], Leventon *et al.* extended the geodesic active contours method with an additional shape-influenced term in the driving PDE. The

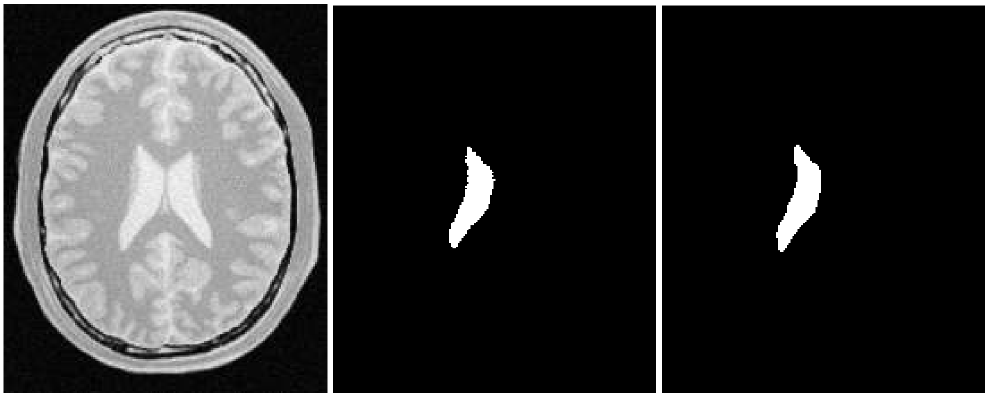


Figure 9.30: Results of applying LaplacianSegmentationLevelSetImageFilter to a prior ventricle segmentation. Shown from left to right are the original image, the prior segmentation of the ventricle from Figure 9.26, and the refinement of the prior using LaplacianSegmentationLevelSetImageFilter.

`itk::GeodesicActiveContourShapePriorLevelSetFilter` is a generalization of Leventon's approach and its use is illustrated in the following example.

To support shape-guidance, the generic level set equation (Eqn(9.3)) is extended to incorporate a shape guidance term:

$$\xi\left(\psi^{*}(\mathbf{x}) - \psi(\mathbf{x})\right) \qquad (9.6)$$

where ψ^* is the signed distance function of the "best-fit" shape with respect to a shape model. The new term has the effect of driving the contour towards the best-fit shape. The scalar ξ weights the influence of the shape term in the overall evolution. In general, the best-fit shape is not known ahead of time and has to be iteratively estimated in conjunction with the contour evolution.

As with the `itk::GeodesicActiveContourLevelSetImageFilter`, the GeodesicActiveContourShapePriorLevelSetImageFilter expects two input images: the first is an initial level set and the second a feature image that represents the image edge potential. The configuration of this example is quite similar to the example in Section 9.3.3 and hence the description will focus on the new objects involved in the segmentation process as shown in Figure 9.31.

The process pipeline begins with centering the input image using the the `itk::ChangeInformationImageFilter` to simplify the estimation of the pose of the shape, to be explained later. The centered image is then smoothed using non-linear diffusion to remove noise and the gradient magnitude is computed from the smoothed image. For simplicity, this example uses the `itk::BoundedReciprocalImageFilter` to produce the edge potential image.

The `itk::FastMarchingImageFilter` creates an initial level set using three user specified

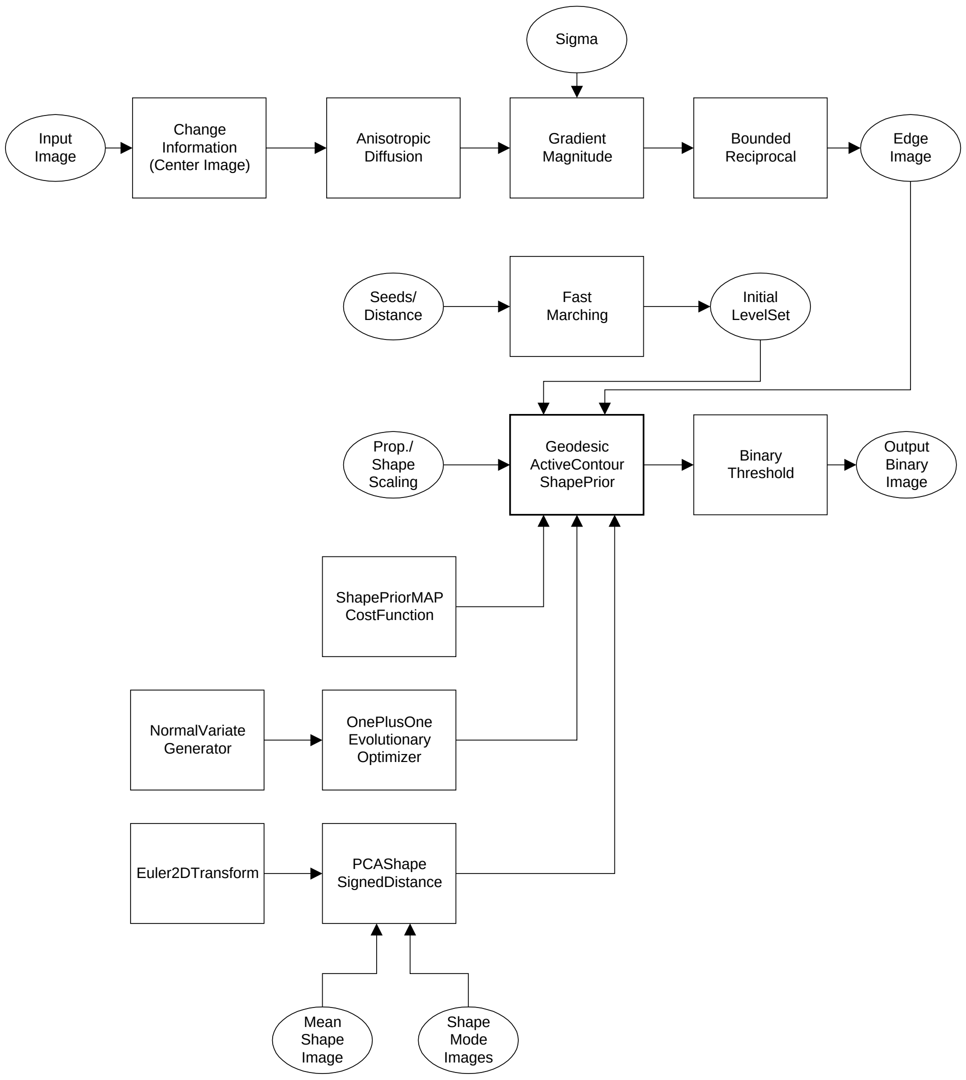


Figure 9.31: Collaboration diagram for the GeodesicActiveContourShapePriorLevelSetImageFilter applied to a segmentation task.

seed positions and a initial contour radius. Three seeds are used in this example to facilitate the segmentation of long narrow objects in a smaller number of iterations. The output of the FastMarchingImageFilter is passed as the input to the GeodesicActiveContourShapePriorLevelSetImageFilter. At then end of the segmentation process, the output level set is passed to the `itk::BinaryThresholdImageFilter` to produce a binary mask representing the segmented object.

The remaining objects in Figure 9.31 are used for shape modeling and estimation. The `itk::PCAShapeSignedDistanceFunction` represents a statistical shape model defined by a mean signed distance and the first K principal components modes; while the `itk::Euler2DTransform` is used to represent the pose of the shape. In this implementation, the best-fit shape estimation problem is reformulated as a minimization problem where the `itk::ShapePriorMAPCostFunction` is the cost function to be optimized using the `itk::OnePlusOneEvolutionaryOptimizer`.

It should be noted that, although particular shape model, transform cost function, and optimizer are used in this example, the implementation is generic, allowing different instances of these components to be plugged in. This flexibility allows a user to tailor the behavior of the segmentation process to suit the circumstances of the targeted application.

Let's start the example by including the headers of the new filters involved in the segmentation.

```
#include "itkGeodesicActiveContourShapePriorLevelSetImageFilter.h"
#include "itkChangeInformationImageFilter.h"
#include "itkBoundedReciprocalImageFilter.h"
```

Next, we include the headers of the objects involved in shape modeling and estimation.

```
#include "itkPCAShapeSignedDistanceFunction.h"
#include "itkEuler2DTransform.h"
#include "itkShapePriorMAPCostFunction.h"
#include "itkOnePlusOneEvolutionaryOptimizer.h"
#include "itkNormalVariateGenerator.h"
#include "vnl/vnl_sample.h"
#include "itkNumericSeriesFileNames.h"
```

Given the numerous parameters involved in tuning this segmentation method it is not uncommon for a segmentation process to run for several minutes and still produce an unsatisfactory result. For debugging purposes it is quite helpful to track the evolution of the segmentation as it progresses. The following defines a custom `itk::Command` class for monitoring the RMS change and shape parameters at each iteration.

```
#include "itkCommand.h"

template<class TFilter>
class CommandIterationUpdate : public itk::Command
```

```
{
public:
  typedef CommandIterationUpdate   Self;
  typedef itk::Command             Superclass;
  typedef itk::SmartPointer<Self>  Pointer;
  itkNewMacro( Self );
protected:
  CommandIterationUpdate() {};
public:

  void Execute(itk::Object *caller, const itk::EventObject & event)
    {
      Execute( (const itk::Object *) caller, event);
    }

  void Execute(const itk::Object * object, const itk::EventObject & event)
    {
      const TFilter * filter =
                  dynamic_cast< const TFilter * >( object );
      if( typeid( event ) != typeid( itk::IterationEvent ) )
        { return; }

      std::cout << filter->GetElapsedIterations() << ": ";
      std::cout << filter->GetRMSChange() << " ";
      std::cout << filter->GetCurrentParameters() << std::endl;
    }

};
```

We define the image type using a particular pixel type and dimension. In this case we will use 2D `float` images.

```
  typedef   float           InternalPixelType;
  const     unsigned int    Dimension = 2;
  typedef itk::Image< InternalPixelType, Dimension >  InternalImageType;
```

The following line instantiate a `itk::GeodesicActiveContourShapePriorLevelSetImageFilter` using the `New()` method.

```
  typedef  itk::GeodesicActiveContourShapePriorLevelSetImageFilter<
                             InternalImageType,
                             InternalImageType >   GeodesicActiveContourFilterType;
  GeodesicActiveContourFilterType::Pointer geodesicActiveContour =
                                    GeodesicActiveContourFilterType::New();
```

The `itk::ChangeInformationImageFilter` is the first filter in the preprocessing stage and is used to force the image origin to the center of the image.

```
typedef itk::ChangeInformationImageFilter<
                             InternalImageType >  CenterFilterType;

CenterFilterType::Pointer center = CenterFilterType::New();
center->CenterImageOn();
```

In this example, we will use the bounded reciprocal $1/(1+x)$ of the image gradient magnitude as the edge potential feature image.

```
typedef   itk::BoundedReciprocalImageFilter<
                             InternalImageType,
                             InternalImageType >  ReciprocalFilterType;

ReciprocalFilterType::Pointer reciprocal = ReciprocalFilterType::New();
```

In the GeodesicActiveContourShapePriorLevelSetImageFilter, scaling parameters are used to trade off between the propagation (inflation), the curvature (smoothing), the advection, and the shape influence terms. These parameters are set using methods `SetPropagationScaling()`, `SetCurvatureScaling()`, `SetAdvectionScaling()` and `SetShapePriorScaling()`. In this example, we will set the curvature and advection scales to one and let the propagation and shape prior scale be command-line arguments.

```
geodesicActiveContour->SetPropagationScaling( propagationScaling );
geodesicActiveContour->SetShapePriorScaling( shapePriorScaling );
geodesicActiveContour->SetCurvatureScaling( 1.0 );
geodesicActiveContour->SetAdvectionScaling( 1.0 );
```

Each iteration, the current “best-fit” shape is estimated from the edge potential image and the current contour. To increase speed, only information within the sparse field layers of the current contour is used in the estimation. The default number of sparse field layers is the same as the ImageDimension which does not contain enough information to get a reliable best-fit shape estimate. Thus, we override the default and set the number of layers to 4.

```
geodesicActiveContour->SetNumberOfLayers( 4 );
```

The filters are then connected in a pipeline as illustrated in Figure 9.31.

```
center->SetInput( reader->GetOutput() );
smoothing->SetInput( center->GetOutput() );
gradientMagnitude->SetInput( smoothing->GetOutput() );
reciprocal->SetInput( gradientMagnitude->GetOutput() );

geodesicActiveContour->SetInput(  fastMarching->GetOutput() );
geodesicActiveContour->SetFeatureImage( reciprocal->GetOutput() );
```

```
  thresholder->SetInput( geodesicActiveContour->GetOutput() );
  writer->SetInput( thresholder->GetOutput() );
```

Next, we define the shape model. In this example, we use an implicit shape model based on the principal components such that:

$$\psi^*(\mathbf{x}) = \mu(\mathbf{x}) + \sum_k \alpha_k u_k(\mathbf{x}) \tag{9.7}$$

where $\mu(\mathbf{x})$ is the mean signed distance computed from training set of segmented objects and $u_k(\mathbf{x})$ are the first K principal components of the offset (signed distance - mean). The coefficients $\{\alpha_k\}$ form the set of *shape* parameters.

Given a set of training data, the itk::ImagePCAShapeModelEstimator can be used to obtain the mean and principal mode shape images required by PCAShapeSignedDistanceFunction.

```
  typedef itk::PCAShapeSignedDistanceFunction<
                                double,
                                Dimension,
                                InternalImageType >     ShapeFunctionType;

  ShapeFunctionType::Pointer shape = ShapeFunctionType::New();

  shape->SetNumberOfPrincipalComponents( numberOfPCAModes );
```

In this example, we will read the mean shape and principal mode images from file. We will assume that the filenames of the mode images form a numeric series starting from index 0.

```
  ReaderType::Pointer meanShapeReader = ReaderType::New();
  meanShapeReader->SetFileName( argv[13] );
  meanShapeReader->Update();

  std::vector<InternalImageType::Pointer> shapeModeImages( numberOfPCAModes );

  itk::NumericSeriesFileNames::Pointer fileNamesCreator =
          itk::NumericSeriesFileNames::New();

  fileNamesCreator->SetStartIndex( 0 );
  fileNamesCreator->SetEndIndex( numberOfPCAModes - 1 );
  fileNamesCreator->SetSeriesFormat( argv[15] );
  const std::vector<std::string> & shapeModeFileNames =
          fileNamesCreator->GetFileNames();

  for ( unsigned int k = 0; k < numberOfPCAModes; k++ )
    {
```

```
    ReaderType::Pointer shapeModeReader = ReaderType::New();
    shapeModeReader->SetFileName( shapeModeFileNames[k].c_str() );
    shapeModeReader->Update();
    shapeModeImages[k] = shapeModeReader->GetOutput();
    }

  shape->SetMeanImage( meanShapeReader->GetOutput() );
  shape->SetPrincipalComponentImages( shapeModeImages );
```

Further we assume that the shape modes have been normalized by multiplying with the corresponding singular value. Hence, we can set the principal component standard deviations to all ones.

```
  ShapeFunctionType::ParametersType pcaStandardDeviations( numberOfPCAModes );
  pcaStandardDeviations.Fill( 1.0 );

  shape->SetPrincipalComponentStandardDeviations( pcaStandardDeviations );
```

Next, we instantiate a `itk::Euler2DTransform` and connect it to the PCASignedDistanceFunction. The transform represent the pose of the shape. The parameters of the transform forms the set of *pose* parameters.

```
  typedef itk::Euler2DTransform<double>    TransformType;
  TransformType::Pointer transform = TransformType::New();

  shape->SetTransform( transform );
```

Before updating the level set at each iteration, the parameters of the current best-fit shape is estimated by minimizing the `itk::ShapePriorMAPCostFunction`. The cost function is composed of four terms: contour fit, image fit, shape prior and pose prior. The user can specify the weights applied to each term.

```
  typedef itk::ShapePriorMAPCostFunction<
                            InternalImageType,
                            InternalPixelType >     CostFunctionType;

  CostFunctionType::Pointer costFunction = CostFunctionType::New();

  CostFunctionType::WeightsType weights;
  weights[0] =  1.0;  // weight for contour fit term
  weights[1] =  20.0; // weight for image fit term
  weights[2] =  1.0;  // weight for shape prior term
  weights[3] =  1.0;  // weight for pose prior term

  costFunction->SetWeights( weights );
```

Contour fit measures the likelihood of seeing the current evolving contour for a given set of shape/pose parameters. This is computed by counting the number of pixels inside the current contour but outside the current shape.

Image fit measures the likelihood of seeing certain image features for a given set of shape/pose parameters. This is computed by assuming that (1 - edge potential) approximates a zero-mean, unit variance Gaussian along the normal of the evolving contour. Image fit is then computed by computing the Laplacian goodness of fit of the Gaussian:

$$\sum \left(G(\psi(\mathbf{x})) - |1 - g(\mathbf{x})|\right)^2 \tag{9.8}$$

where G is a zero-mean, unit variance Gaussian and g is the edge potential feature image.

The pose parameters are assumed to have a uniform distribution and hence do not contribute to the cost function. The shape parameters are assumed to have a Gaussian distribution. The parameters of the distribution are user-specified. Since we assumed the principal modes have already been normalized, we set the distribution to zero mean and unit variance.

```
CostFunctionType::ArrayType mean(   shape->GetNumberOfShapeParameters() );
CostFunctionType::ArrayType stddev( shape->GetNumberOfShapeParameters() );

mean.Fill( 0.0 );
stddev.Fill( 1.0 );
costFunction->SetShapeParameterMeans( mean );
costFunction->SetShapeParameterStandardDeviations( stddev );
```

In this example, we will use the `itk::OnePlusOneEvolutionaryOptimizer` to optimize the cost function.

```
typedef itk::OnePlusOneEvolutionaryOptimizer    OptimizerType;
OptimizerType::Pointer optimizer = OptimizerType::New();
```

The evolutionary optimization algorithm is based on testing random permutations of the parameters. As such, we need to provide the optimizer with a random number generator. In the following lines, we create a `itk::NormalVariateGenerator`, seed it, and connect it to the optimizer.

```
typedef itk::Statistics::NormalVariateGenerator GeneratorType;
GeneratorType::Pointer generator = GeneratorType::New() ;

generator->Initialize( 20020702 ) ;

optimizer->SetNormalVariateGenerator( generator ) ;
```

The cost function has $K+3$ parameters. The first K parameters are the principal component multipliers, followed by the 2D rotation parameter (in radians) and the x- and y- translation parameters (in mm). We need to carefully scale the different types of parameters to compensate for the differences in the dynamic ranges of the parameters.

```
OptimizerType::ScalesType scales( shape->GetNumberOfParameters() );
scales.Fill( 1.0 );
for( unsigned int k = 0; k < numberOfPCAModes; k++ )
  {
  scales[k] = 20.0;  // scales for the pca mode multiplier
  }
scales[numberOfPCAModes] = 350.0;  // scale for 2D rotation
optimizer->SetScales( scales );
```

Next, we specify the initial radius, the shrink and grow mutation factors and termination criteria of the optimizer. Since the best-fit shape is re-estimated each iteration of the curve evolution, we do not need to spend too much time finding the true minimizing solution each time; we only need to head towards it. As such, we only require a small number of optimizer iterations.

```
double initRadius = 1.05;
double grow = 1.1 ;
double shrink = pow(grow, -0.25) ;
optimizer->Initialize(initRadius, grow, shrink) ;

optimizer->SetEpsilon(1.0e-6) ; // minimal search radius

optimizer->SetMaximumIteration(15) ;
```

Before starting the segmentation process we need to also supply the initial best-fit shape estimate. In this example, we start with the unrotated mean shape with the initial x- and y-translation specified through command-line arguments.

```
ShapeFunctionType::ParametersType parameters( shape->GetNumberOfParameters() );
parameters.Fill( 0.0 );
parameters[numberOfPCAModes + 1] = atof( argv[16] ); // startX
parameters[numberOfPCAModes + 2] = atof( argv[17] ); // startY
```

Finally, we connect all the components to the filter and add our observer.

```
geodesicActiveContour->SetShapeFunction( shape );
geodesicActiveContour->SetCostFunction( costFunction );
geodesicActiveContour->SetOptimizer( optimizer );
geodesicActiveContour->SetInitialParameters( parameters );

typedef CommandIterationUpdate<GeodesicActiveContourFilterType> CommandType;
CommandType::Pointer observer = CommandType::New();
geodesicActiveContour->AddObserver( itk::IterationEvent(), observer );
```

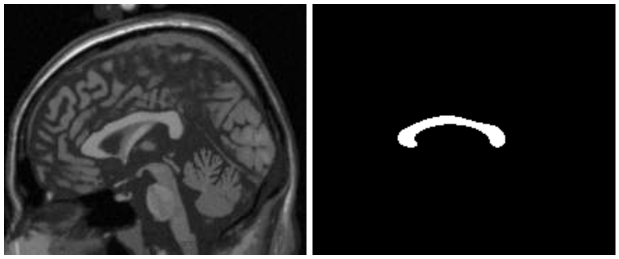


Figure 9.32: The input image to the GeodesicActiveContourShapePriorLevelSetImageFilter is a synthesized MR-T1 mid-sagittal slice (217×180 pixels, 1×1 mm spacing) of the brain (left) and the initial best-fit shape (right) chosen to roughly overlap the corpus callosum in the image to be segmented.

The invocation of the `Update()` method on the writer triggers the execution of the pipeline. As usual, the call is placed in a `try/catch` block to handle exceptions should errors occur.

```
try
  {
  writer->Update();
  }
catch( itk::ExceptionObject & excep )
  {
  std::cerr << "Exception caught !" << std::endl;
  std::cerr << excep << std::endl;
  }
```

Deviating from previous examples, we will demonstrate this example using `BrainMidSagittalSlice.png` (Figure 9.32, left) from the `Examples/Data` directory. The aim here is to segment the corpus callosum from the image using a shape model defined by `CorpusCallosumMeanShape.mha` and the first three principal components `CorpusCallosumMode0.mha`, `CorpusCallosumMode1.mha` and `CorpusCallosumMode12.mha`. As shown in Figure 9.33, the first mode captures scaling, the second mode captures the shifting of mass between the rostrum and the splenium and the third mode captures the degree of curvature. Segmentation results with and without shape guidance are shown in Figure 9.34.

A sigma value of 1.0 was used to compute the image gradient and the propagation and shape prior scaling are respectively set to 0.5 and 0.02. An initial level set was created by placing one seed point in the rostrum $(60,102)$, one in the splenium $(120,85)$ and one centrally in the body $(88,83)$ of the corpus callosum with an initial radius of 6 pixels at each seed position. The best-fit shape was initially placed with a translation of $(10,0)$mm so that it roughly overlapped the corpus callosum in the image as shown in Figure 9.32 (right).

From Figure 9.34 it can be observed that without shape guidance (left), segmentation using geodesic active contour leaks in the regions where the corpus callosum blends into the sur-

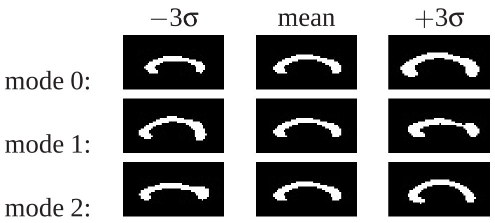


Figure 9.33: First three PCA modes of a low-resolution (58×31 pixels, 2×2 mm spacing) corpus callosum model used in the shape guided geodesic active contours example.

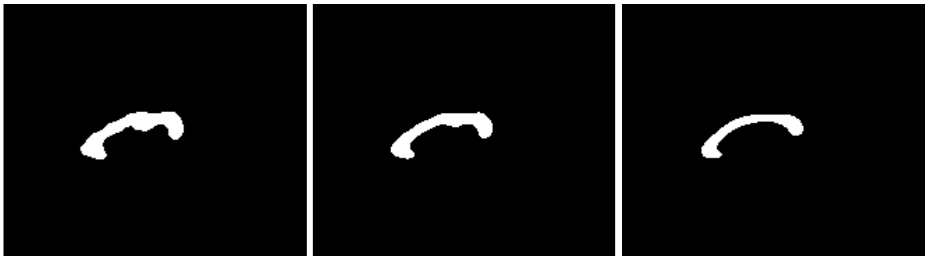


Figure 9.34: Corpus callosum segmentation using geodesic active contours without (left) and with (center) shape guidance. The image on the right represents the best-fit shape at the end of the segmentation process.

rounding brain tissues. With shape guidance (center), the segmentation is constrained by the global shape model to prevent leaking.

The final best-fit shape parameters after the segmentation process is:

```
Parameters: [-0.384988, -0.578738, 0.557793, 0.275202, 16.9992, 4.73473]
```

and is shown in Figure 9.34 (right). Note that a 0.28 radian (15.8 degree) rotation has been introduced to match the model to the corpus callosum in the image. Additionally, a negative weight for the first mode shrinks the size relative to the mean shape. A negative weight for the second mode shifts the mass to splenium, and a positive weight for the third mode increases the curvature. It can also be observed that the final segmentation is a combination of the best-fit shape with additional local deformation. The combination of both global and local shape allows the segmentation to capture fine details not represented in the shape model.

9.4 Hybrid Methods

9.4.1 Introduction

This section introduces the use of hybrid methods for segmentation of image data. Typically we are dealing with radiological patient and the Visible Human data. The hybrid segmentation approach integrates boundary-based and region-based segmentation methods that amplify the strength but reduce the weakness of both techniques. The advantage of this approach comes from combining region-based segmentation methods like the fuzzy connectedness and Voronoi diagram classification with boundary-based deformable model segmentation. The synergy between fundamentally different methodologies tends to result in robustness and higher segmentation quality. A hybrid segmentation engine can be built, as illustrated in Figure 9.35. It consists of modules representing segmentation methods and implemented as ITK filters. We can derive a variety of hybrid segmentation methods by exchanging the filter used in each module. It should be noted that under the fuzzy connectedness and deformable models modules, there are several different filters that can be used as components. Below, we describe two examples of hybrid segmentation methods, derived from the hybrid segmentation engine: integration of fuzzy connectedness and Voronoi diagram classification (hybrid method 1), and integration of Gibbs prior and deformable models (hybrid method 2). Details regarding the concepts behind these methods have been discussed in the literature [4, 83, 41, 40, 39, 38]

9.4.2 Fuzzy Connectedness and Confidence Connectedness

Probably the simplest combination of hybrid filters is the pair formed by the `itk::ConfidenceConnectednessImageFilter` and `itk::SimpleFuzzyConnectednessScalarImageFilter`. In this combination the confidence connectedness filter is used to produce a rough segmentation of an anatomical structure and to compute and estimation of the mean and variance of gray values in such structure. The values of mean and variance are then passed to the Simple Fuzzy Connectedness image filter in order to compute an affinity map.

The source code for this section can be found in the file
`Examples/Patented/FuzzyConnectednessImageFilter.cxx`.

This example illustrates the use of the `itk::SimpleFuzzyConnectednessScalarImageFilter`. This filter computes an affinity map from a seed point provided by the user. This affinity map indicates for every pixels how homogeneous is the path that will link it to the seed point.

Please note that the Fuzzy Connectedness algorithm is covered by a Patent [84]. For this reason the current example is located in the `Examples/Patented` subdirectory.

In order to use this algorithm we should first include the header files of the filter and the image class.

```
#include "itkSimpleFuzzyConnectednessScalarImageFilter.h"
```

```
#include "itkImage.h"
```

Since the FuzzyConnectednessImageFilter requires an estimation of the gray level mean and variance for the region to be segmented, we use here the `itk::ConfidenceConnectedImageFilter` as a preprocessor that produces a rough segmentation and estimates from it the values of the mean and the variance.

```
#include "itkConfidenceConnectedImageFilter.h"
```

Next, we declare the pixel type and image dimension and specify the image type to be used as input.

```
  typedef  float             InputPixelType;
  const    unsigned int     Dimension = 2;
  typedef itk::Image< InputPixelType, Dimension >  InputImageType;
```

Fuzzy connectedness computes first the affinity map and then thresholds its values in order to get a binary image as output. The type of the binary image is provided as the second template parameter of the filter.

```
  typedef   unsigned char   BinaryPixelType;
  typedef itk::Image< BinaryPixelType, Dimension >     BinaryImageType;
```

The Confidence connected filter type is instantiated using the input image type and a binary image type for output.

```
  typedef itk::ConfidenceConnectedImageFilter<
                                                InputImageType,
                                                BinaryImageType
                                                  >  ConfidenceConnectedFilterType;

  ConfidenceConnectedFilterType::Pointer confidenceConnectedFilter =
                                               ConfidenceConnectedFilterType::New();
```

The fuzzy segmentation filter type is instantiated here using the input and binary image types as template parameters.

```
  typedef itk::SimpleFuzzyConnectednessScalarImageFilter<
                                                InputImageType,
                                                BinaryImageType
                                                  >  FuzzySegmentationFilterType;
```

The fuzzy connectedness segmentation filter is created by invoking the `New()` method and assigning the result to a `itk::SmartPointer`.

```
  FuzzySegmentationFilterType::Pointer fuzzysegmenter =
                                       FuzzySegmentationFilterType::New();
```

The affinity map can be accessed through the type `FuzzySceneType`

```
  typedef FuzzySegmentationFilterType::FuzzySceneType  FuzzySceneType;
```

We instantiate reader and writer types

The output of the reader is passed as input to the ConfidenceConnected image filter. Then the filter is executed in order to obtain estimations of the mean and variance gray values for the region to be segmented.

```
  confidenceConnectedFilter->SetInput( reader->GetOutput()  );
  confidenceConnectedFilter->SetMultiplier( varianceMultiplier );
  confidenceConnectedFilter->SetNumberOfIterations( 2 );
  confidenceConnectedFilter->AddSeed( index );

  confidenceConnectedFilter->Update();
```

The input that is passed to the fuzzy segmentation filter is taken from the reader.

```
  fuzzysegmenter->SetInput( reader->GetOutput() );
```

The parameters of the fuzzy segmentation filter are defined here. A seed point is provided with the method `SetObjectsSeed()` in order to initialize the region to be grown. Estimated values for the mean and variance of the object intensities are also provided with the methods `SetMean()` and `SetVariance()`, respectively. A threshold value for generating the binary object is preset with the method `SetThreshold()`. For details describing the role of the mean and variance on the computation of the segmentation, please see [82].

```
  fuzzysegmenter->SetObjectSeed( index );
  fuzzysegmenter->SetMean( meanEstimation );
  fuzzysegmenter->SetVariance( varianceEstimation );
  fuzzysegmenter->SetThreshold( 0.5 );
```

The execution of the fuzzy segmentation filter is triggered by the `Update()` method.

```
  fuzzysegmenter->Update();

  writer->SetInput( fuzzysegmenter->GetOutput() );
  writer->Update();

  fwriter->SetInput( fuzzysegmenter->GetFuzzyScene() );
  fwriter->Update();
```

9.4.3 Fuzzy Connectedness and Voronoi Classification

In this section we present a hybrid segmentation method that requires minimal manual initialization by integrating the fuzzy connectedness and Voronoi diagram classification segmentation algorithms. We start with a fuzzy connectedness filter to generate a sample of tissue from a region to be segmented. From the sample, we automatically derive image statistics that constitute the homogeneity operator to be used in the next stage of the method. The output of the fuzzy connectedness filter is used as a prior to the Voronoi diagram classification filter. This filter performs iterative subdivision and classification of the segmented image resulting in an estimation of the boundary. The output of this filter is a 3D binary image that can be used to display the 3D result of the segmentation, or passed to another filter (e.g. deformable model) for further improvement of the final segmentation. Details describing the concepts behind these methods have been published in [4, 83, 41, 40, 39, 38]

In Figure 9.36, we describe the base class for simple fuzzy connectedness segmentation. This method is non-scale based and non-iterative, and requires only one seed to initialize it. We define affinity between two nearby elements in a image (e.g. pixels, voxels) via a degree of adjacency, similarity in their intensity values, and their similarity to the estimated object. The closer the elements and the more similar their intensities, the greater the affinity between them. We compute the strength of a path and fuzzy connectedness between each two pixels (voxels) in the segmented image from the fuzzy affinity. Computation of the fuzzy connectedness value of each pixel (voxel) is implemented by selecting a seed point and using dynamic programming. The result constitutes the fuzzy map. Thresholding of the fuzzy map gives a segmented object that is strongly connected to the seed point (for more details, see [82]). Two fuzzy connectedness filters are available in the toolkit:

- The `itk::SimpleFuzzyConnectednessScalarImageFilter`, an implementation of the fuzzy connectedness segmentation of single-channel (grayscale) image.
- The `itk::SimpleFuzzyConnectednessRGBImageFilter`, an implementation of fuzzy connectedness segmentation of a three-channel (RGB) image.

New classes can be derived from the base class by defining other affinity functions and targeting multi-channel images with an arbitrary number of channels. Note that the simple fuzzy connectedness filter can be used as a stand-alone segmentation method and does not necessarily need to be combined with other methods as indicated by Figure 9.37.

In Figure 9.38 we present the base class for Voronoi diagram classification. We initialize the method with a number of random seed points and compute the Voronoi diagram over the segmented 2D image. Each Voronoi region in the subdivision is classified as internal or external, based on the homogeneity operator derived from the fuzzy connectedness algorithm. We define boundary regions as the external regions that are adjacent to the internal regions. We further subdivide the boundary regions by adding seed points to them. We converge to the final segmentation using simple stopping criteria (for details, see [40]). Two Voronoi-based segmentation methods are available in ITK: the `itk::VoronoiSegmentationImageFilter` for processing

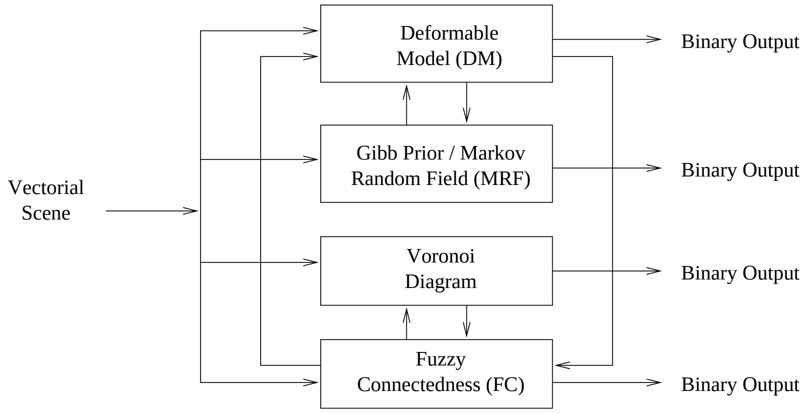


Figure 9.35: The hybrid segmentation engine.

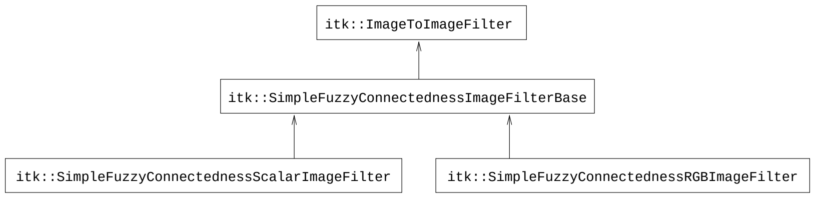

itk::ImageToImageFilter
itk::SimpleFuzzyConnectednessImageFilterBase
itk::SimpleFuzzyConnectednessScalarImageFilter
itk::SimpleFuzzyConnectednessRGBImageFilter

Figure 9.36: Inheritance diagram for the fuzzy connectedness filter.

single-channel (grayscale) images, and the `itk::VoronoiSegmentationRGBImageFilter`, for segmenting three-channel (RGB) images. New classes can be derived from the base class by defining other homogeneity measurements and targeting multichannel images with an arbitrary number of channels. The other classes that are used for computing a 2D Voronoi diagram are shown in Figure 9.39. Note that the Voronoi diagram filter can be used as a stand-alone segmentation method, as depicted in Figure 9.40.

Figures 9.41 and 9.42 illustrate hybrid segmentation methods that integrate fuzzy connectedness with Voronoi diagrams, and fuzzy connectedness, Voronoi diagrams and deformable models, respectively.

Example of a Hybrid Segmentation Method

The source code for this section can be found in the file
`Examples/Patented/HybridSegmentationFuzzyVoronoi.cxx`.

This example illustrates the use of the `itk::SimpleFuzzyConnectednessScalarImageFilter` and `itk::VoronoiSegmentationImageFilter` to build a hybrid segmentation that integrates fuzzy connectedness with the Voronoi diagram classification.

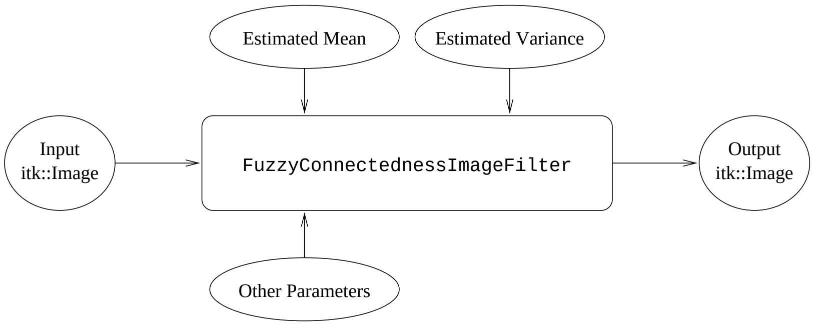


Figure 9.37: Inputs and outputs to FuzzyConnectednessImageFilter segmentation algorithm.

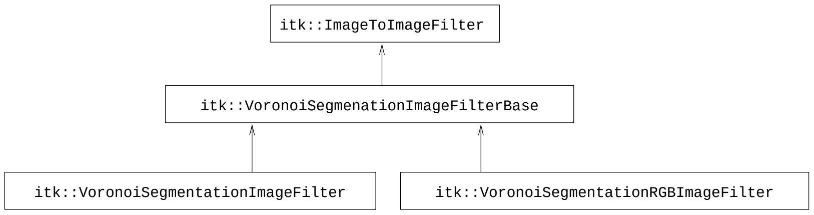

itk::ImageToImageFilter
itk::VoronoiSegmenationImageFilterBase
itk::VoronoiSegmentationImageFilter
itk::VoronoiSegmentationRGBImageFilter

Figure 9.38: Inheritance diagram for the Voronoi segmentation filters.

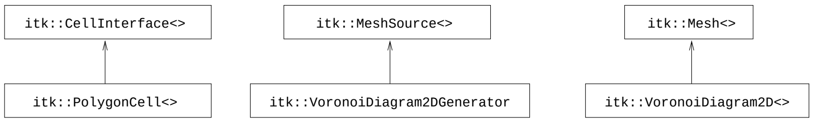


Figure 9.39: Classes used by the Voronoi segmentation filters.

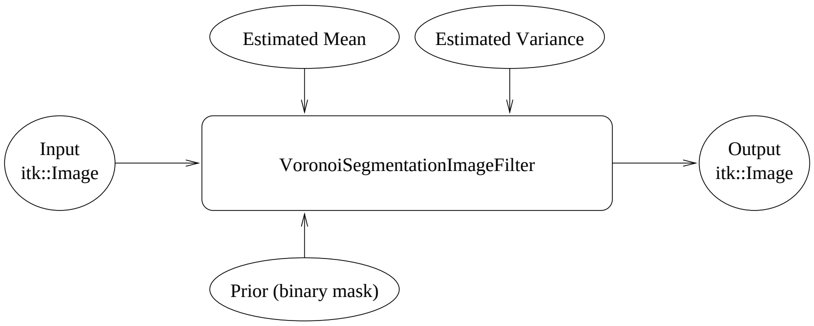


Figure 9.40: Input and output to the VoronoiSegmentationImageFilter.

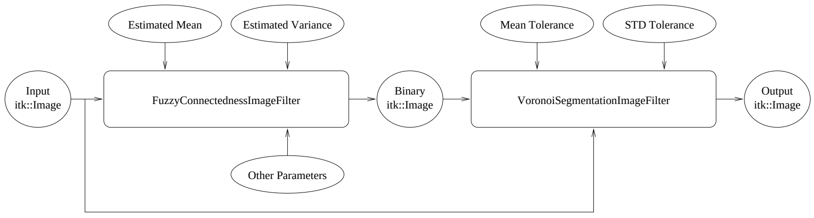


Figure 9.41: Integration of the fuzzy connectedness and Voronoi segmentation filters.

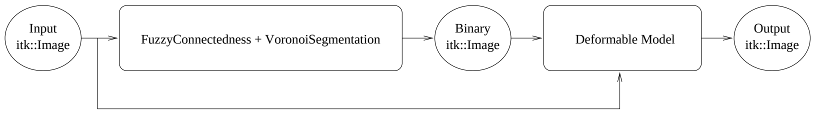


Figure 9.42: Integration of the fuzzy connectedness, Voronoi, and deformable model segmentation methods.

Please note that the Fuzzy Connectedness algorithm is covered by a Patent [84]. For this reason the current example is located in the `Examples/Patented` subdirectory.

First, we include the header files of the two filters.

```
#include "itkSimpleFuzzyConnectednessScalarImageFilter.h"
#include "itkVoronoiSegmentationImageFilter.h"
```

Next, we declare the pixel type and image dimension and specify the image type to be used as input.

```
  typedef  float     InputPixelType;
  const    unsigned int      Dimension = 2;
  typedef itk::Image< InputPixelType, Dimension >  InputImageType;
```

Fuzzy connectedness segmentation is performed first to generate a rough segmentation that yields a sample of tissue from the region to be segmented. A binary result, representing the sample, is used as a prior for the next step. Here, we use the `itk::SimpleFuzzyConnectednessScalarImageFilter`, but we may also utilize any other image segmentation filter instead. The result produced by the fuzzy segmentation filter is stored in a binary image. Below, we declare the type of the image using a pixel type and a spatial dimension.

```
  typedef unsigned char      BinaryPixelType;
  typedef itk::Image< BinaryPixelType, Dimension >      BinaryImageType;
```

The fuzzy segmentation filter type is instantiated here using the input and binary image types as template parameters.

```
typedef   itk::SimpleFuzzyConnectednessScalarImageFilter<
                                                   InputImageType,
                                                   BinaryImageType
                                                      >  FuzzySegmentationFilterType;
```

The fuzzy connectedness segmentation filter is created by invoking the `New()` method and assigning the result to a `itk::SmartPointer`.

```
FuzzySegmentationFilterType::Pointer fuzzysegmenter =
                                       FuzzySegmentationFilterType::New();
```

In the next step of the hybrid segmentation method, the prior generated from the fuzzy segmentation is used to build a homogeneity measurement for the object. A VoronoiSegmentationImageFilter uses the homogeneity measurement to drive iterative subdivision of Voronoi regions and to generate the final segmentation result (for details, please see [38]). In this example, the result of the VoronoiSegmentationImageFilter is sent to a writer. Its output type is conveniently declared as one that is compatible with the writer.

```
typedef unsigned char OutputPixelType;
typedef itk::Image< OutputPixelType, Dimension > OutputImageType;
```

The following lines instantiate the Voronoi segmentation filter.

```
typedef  itk::VoronoiSegmentationImageFilter<
                                    InputImageType,
                                    OutputImageType,
                                    BinaryImageType>
                                         VoronoiSegmentationFilterType;

VoronoiSegmentationFilterType::Pointer voronoisegmenter =
                                   VoronoiSegmentationFilterType::New();
```

The input that is passed to the fuzzy segmentation filter is taken from the reader.

```
fuzzysegmenter->SetInput( reader->GetOutput() );
```

The parameters of the fuzzy segmentation filter are defined here. A seed point is provided with the method `SetObjectSeed()` in order to initialize the region to be grown. Estimated values for the mean and variance of the object intensities are also provided with the methods `SetMean()` and `SetVariance()`, respectively. A threshold value for generating the binary object is preset with the method `SetThreshold()`. For details describing the role of the mean and variance on the computation of the segmentation, please see [82].

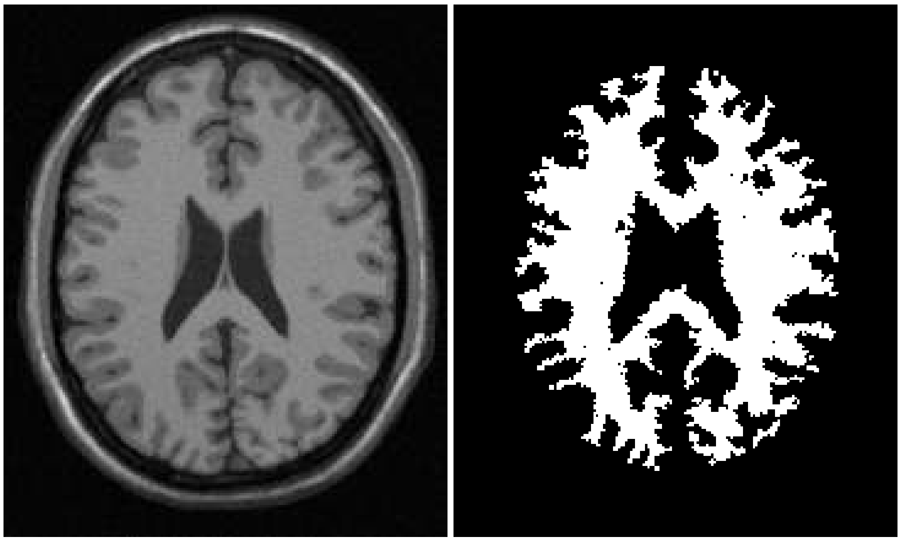


Figure 9.43: Segmentation results for the hybrid segmentation approach.

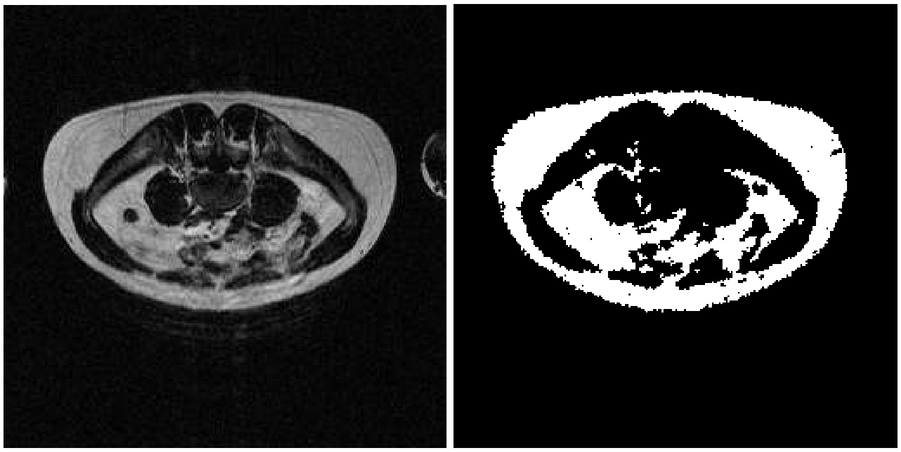


Figure 9.44: Another segmentation result for the hybrid segmentation approach.

```
fuzzysegmenter->SetObjectSeed( index );
fuzzysegmenter->SetMean( mean );
fuzzysegmenter->SetVariance( variance );
fuzzysegmenter->SetThreshold( 0.5 );
```

The execution of the fuzzy segmentation filter is triggered by the `Update()` method.

```
fuzzysegmenter->Update();
```

The input to the Voronoi diagram classification filter is obtained from the reader and the prior is obtained from the fuzzy segmentation filter.

```
voronoisegmenter->SetInput( reader->GetOutput() );
voronoisegmenter->TakeAPrior( fuzzysegmenter->GetOutput() );
```

The tolerance levels for testing the mean and standard deviation are set with the methods `SetMeanPercentError()` and `SetSTDPercentError()`. Note that the fuzzy segmentation filter uses *variance* as parameter while the Voronoi segmentation filter uses the tolerance of the *standard deviation* as a parameter. For more details on how these parameters should be selected, please see [38].

```
voronoisegmenter->SetMeanPercentError( meanTolerance );
voronoisegmenter->SetSTDPercentError(  stdTolerance );
```

The *resolution* of the Voronoi diagram classification can be chosen with the method `SetMinRegion()`.

```
voronoisegmenter->SetMinRegion( 5 );
```

The execution of the Voronoi segmentation filter is triggered with the `Update()` method.

```
voronoisegmenter->Update();
```

The output of the Voronoi diagram classification is an image mask with zeros everywhere and ones inside the segmented object. This image will appear black on many image viewers since they do not usually stretch the gray levels. Here, we add a `itk::RescaleIntensityImageFilter` in order to expand the dynamic range to more typical values.

```
typedef itk::RescaleIntensityImageFilter< OutputImageType,OutputImageType >
  ScalerFilterType;
ScalerFilterType::Pointer scaler = ScalerFilterType::New();
```

```
  scaler->SetOutputMinimum(   0 );
  scaler->SetOutputMaximum( 255 );

  scaler->SetInput( voronoisegmenter->GetOutput() );
```

The output of the rescaler is passed to the writer. The invocation of the `Update()` method on the writer triggers the execution of the pipeline.

```
  writer->SetInput( scaler->GetOutput() );
  writer->Update();
```

We execute this program on the image `BrainT1Slice.png` available in the directory `Examples/Data`. The following parameters are passed to the command line:

```
HybridSegmentationFuzzyVoronoi BrainT1Slice.png Output.png 140 125 140 25 0.2 2.0
```

$(140,125)$ specifies the index position of a seed point in the image, while 140 and 25 are the estimated mean and standard deviation, respectively, of the object to be segmented. Finally, 0.2 and 2.0 are the tolerance for the mean and standard deviation, respectively. Figure 9.43 shows the input image and the binary mask resulting from the segmentation.

Note that in order to successfully segment other images, these parameters have to be adjusted to reflect the data. For example, when segmenting the input image `FatMRISlice.png` we apply the following new set of parameters parameters.

```
HybridSegmentationFuzzyVoronoi FatMRISlice.png Output.png 80 200 140 300 0.3 3.0
```

Figure 9.44 shows the input image and the binary mask resulting from this segmentation. Note that, we can segment color (RGB) and other multi-channel images using an approach similar to this example.

9.4.4 Deformable Models and Gibbs Prior

Another combination that can be used in a hybrid segmentation method is the set of Gibbs prior filters with deformable models.

Deformable Model

The source code for this section can be found in the file
`Examples/Segmentation/DeformableModel1.cxx`.

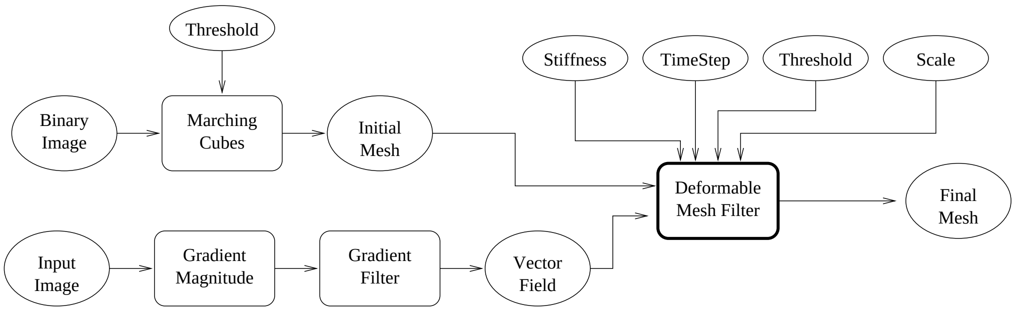


Figure 9.45: Collaboration diagram for the DeformableMesh3DFilter applied to a segmentation task.

This example illustrates the use of the itk::DeformableMesh3DFilter and itk::BinaryMask3DMeshSource in the hybrid segmentation framework.

The purpose of the DeformableMesh3DFilter is to take an initial surface described by an itk::Mesh and deform it in order to adapt it to the shape of an anatomical structure in an image. Figure 9.45 illustrates a typical setup for a segmentation method based on deformable models. First, an initial mesh is generated using a binary mask and an isocontouring algorithm (such as marching cubes) to produce an initial mesh. The binary mask used here contains a simple shape which vaguely resembles the anatomical structure that we want to segment. The application of the isocontouring algorithm produces a $3D$ mesh that has the shape of this initial structure. This initial mesh is passed as input to the deformable model which will apply forces to the mesh points in order to reshape the surface until make it fit to the anatomical structures in the image.

The forces to be applied on the surface are computed from an approximate physical model that simulates an elastic deformation. Among the forces to be applied we need one that will pull the surface to the position of the edges in the anatomical structure. This force component is represented here in the form of a vector field and is computed as illustrated in the lower left of Figure 9.45. The input image is passed to a itk::GradientMagnitudeRecursiveGaussianImageFilter, which computes the magnitude of the image gradient. This scalar image is then passed to another gradient filter (itk::GradientRecursiveGaussianImageFilter). The output of this second gradient filter is a vector field in which every vector points to the closest edge in the image and has a magnitude proportional to the second derivative of the image intensity along the direction of the gradient. Since this vector field is computed using Gaussian derivatives, it is possible to regulate the smoothness of the vector field by playing with the value of sigma assigned to the Gaussian. Large values of sigma will result in a large capture radius, but will have poor precision in the location of the edges. A reasonable strategy may involve the use of large sigmas for the initial iterations of the model and small sigmas to refine the model when it is close to the edges. A similar effect could be achieved using multiresolution and taking advantage of the image pyramid structures already illustrated in the registration framework.

We start by including the headers of the main classes required for this example. The BinaryMask3DMeshSource is used to produce an initial approximation of the shape to be segmented. This filter takes a binary image as input and produces a Mesh as output using the marching cube isocontouring algorithm.

```
#include "itkBinaryMask3DMeshSource.h"
```

Then we include the header of the DeformableMesh3DFilter that implements the deformable model algorithm.

```
#include "itkDeformableMesh3DFilter.h"
```

We also need the headers of the gradient filters that will be used for computing the vector field. In our case they are the GradientMagnitudeRecursiveGaussianImageFilter and GradientRecursiveGaussianImageFilter.

```
#include "itkGradientRecursiveGaussianImageFilter.h"
#include "itkGradientMagnitudeRecursiveGaussianImageFilter.h"
```

The main data structures required in this example are the Image and the Mesh classes. The deformable model *per se* is represented as a Mesh.

```
#include "itkImage.h"
#include "itkMesh.h"
```

The `PixelType` of the image derivatives is represented with a `itk::CovariantVector`. We include its header in the following line.

```
#include "itkCovariantVector.h"
```

The deformed mesh is converted into a binary image using the `itk::PointSetToImageFilter`.

```
#include "itkPointSetToImageFilter.h"
```

In order to read both the input image and the mask image, we need the `itk::ImageFileReader` class. We also need the `itk::ImageFileWriter` to save the resulting deformed mask image.

```
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
```

Here we declare the type of the image to be processed. This implies a decision about the `PixelType` and the dimension. The DeformableMesh3DFilter is specialized for 3*D*, so the choice is clear in our case.

```
const     unsigned int    Dimension = 3;
typedef   double          PixelType;
typedef itk::Image<PixelType, Dimension>      ImageType;
```

The input to BinaryMask3DMeshSource is a binary mask that we will read from a file. This mask could be the result of a rough segmentation algorithm applied previously to the same anatomical structure. We declare below the type of the binary mask image.

```
typedef itk::Image< unsigned char, Dimension >   BinaryImageType;
```

Then we define the type of the deformable mesh. We represent the deformable model using the Mesh class. The `double` type used as template parameter here is to be used to assign values to every point of the Mesh.

```
typedef  itk::Mesh<double>     MeshType;
```

The following lines declare the type of the gradient image:

```
typedef itk::CovariantVector< double, Dimension >  GradientPixelType;
typedef itk::Image< GradientPixelType, Dimension > GradientImageType;
```

With it we can declare the type of the gradient filter and the gradient magnitude filter:

```
typedef itk::GradientRecursiveGaussianImageFilter<ImageType, GradientImageType>
  GradientFilterType;
typedef itk::GradientMagnitudeRecursiveGaussianImageFilter<ImageType,ImageType>
  GradientMagnitudeFilterType;
```

The filter implementing the isocontouring algorithm is the BinaryMask3DMeshSource filter.

```
typedef itk::BinaryMask3DMeshSource< BinaryImageType, MeshType >  MeshSourceType;
```

Now we instantiate the type of the DeformableMesh3DFilter that implements the deformable model algorithm. Note that both the input and output types of this filter are `itk::Mesh` classes.

```
typedef itk::DeformableMesh3DFilter<MeshType,MeshType>  DeformableFilterType;
```

Let's declare two readers. The first will read the image to be segmented. The second will read the binary mask containing a first approximation of the segmentation that will be used to initialize a mesh for the deformable model.

```
typedef itk::ImageFileReader< ImageType       >  ReaderType;
typedef itk::ImageFileReader< BinaryImageType >  BinaryReaderType;
ReaderType::Pointer       imageReader   =  ReaderType::New();
BinaryReaderType::Pointer maskReader    =  BinaryReaderType::New();
```

In this example we take the filenames of the input image and the binary mask from the command line arguments.

```
imageReader->SetFileName( argv[1] );
maskReader->SetFileName(  argv[2] );
```

We create here the GradientMagnitudeRecursiveGaussianImageFilter that will be used to compute the magnitude of the input image gradient. As usual, we invoke its `New()` method and assign the result to a `itk::SmartPointer`.

```
GradientMagnitudeFilterType::Pointer  gradientMagnitudeFilter
                                        = GradientMagnitudeFilterType::New();
```

The output of the image reader is connected as input to the gradient magnitude filter. Then the value of sigma used to blur the image is selected using the method `SetSigma()`.

```
gradientMagnitudeFilter->SetInput( imageReader->GetOutput() );
gradientMagnitudeFilter->SetSigma( 1.0 );
```

In the following line, we construct the gradient filter that will take the gradient magnitude of the input image that will be passed to the deformable model algorithm.

```
GradientFilterType::Pointer gradientMapFilter = GradientFilterType::New();
```

The magnitude of the gradient is now passed to the next step of gradient computation. This allows us to obtain a second derivative of the initial image with the gradient vector pointing to the maxima of the input image gradient. This gradient map will have the properties desirable for attracting the deformable model to the edges of the anatomical structure on the image. Once again we must select the value of sigma to be used in the blurring process.

```
gradientMapFilter->SetInput( gradientMagnitudeFilter->GetOutput());
gradientMapFilter->SetSigma( 1.0 );
```

At this point, we are ready to compute the vector field. This is done simply by invoking the `Update()` method on the second derivative filter. This was illustrated in Figure 9.45.

```
gradientMapFilter->Update();
```

Now we can construct the mesh source filter that implements the isocontouring algorithm.

```
MeshSourceType::Pointer meshSource = MeshSourceType::New();
```

Then we create the filter implementing the deformable model and set its input to the output of the binary mask mesh source. We also set the vector field using the `SetGradient()` method.

```
DeformableFilterType::Pointer deformableModelFilter =
                                   DeformableFilterType::New();
deformableModelFilter->SetGradient( gradientMapFilter->GetOutput() );
```

Here we connect the output of the binary mask reader to the input of the Binary-Mask3DMeshSource that will apply the isocontouring algorithm and generate the initial mesh to be deformed. We must also select the value to be used for representing the binary object in the image. In this case we select the value 200 and pass it to the filter using its method `SetObjectValue()`.

```
BinaryImageType::Pointer mask = maskReader->GetOutput();
meshSource->SetInput( mask );
meshSource->SetObjectValue( 200 );

std::cout << "Creating mesh..." << std::endl;
try
  {
  meshSource->Update();
  }
catch( itk::ExceptionObject & excep )
  {
  std::cerr << "Exception Caught !" << std::endl;
  std::cerr << excep << std::endl;
  }

deformableModelFilter->SetInput(  meshSource->GetOutput() );
```

Next, we set the parameters of the deformable model computation. `Stiffness` defines the model stiffness in the vertical and horizontal directions on the deformable surface. `Scale` helps to accommodate the deformable mesh to gradient maps of different size.

```
typedef itk::CovariantVector<double, 2>           double2DVector;
typedef itk::CovariantVector<double, 3>           double3DVector;

double2DVector stiffness;
stiffness[0] = 0.0001;
stiffness[1] = 0.1;
```

```
  double3DVector scale;
  scale[0] = 1.0;
  scale[1] = 1.0;
  scale[2] = 1.0;

  deformableModelFilter->SetStiffness( stiffness );
  deformableModelFilter->SetScale( scale );
```

Other parameters to be set are the gradient magnitude, the time step and the step threshold. The gradient magnitude controls the magnitude of the external force. The time step controls the length of each step during deformation. Step threshold is the number of the steps the model will deform.

```
  deformableModelFilter->SetGradientMagnitude( 0.8 );
  deformableModelFilter->SetTimeStep( 0.01 );
  deformableModelFilter->SetStepThreshold( 60 );
```

Finally, we trigger the execution of the deformable model computation using the `Update()` method of the DeformableMesh3DFilter. As usual, the call to `Update()` should be placed in a `try/catch` block in case any exceptions are thrown.

```
  try
    {
    deformableModelFilter->Update();
    }
  catch( itk::ExceptionObject & excep )
    {
    std::cerr << "Exception Caught !" << std::endl;
    std::cerr << excep << std::endl;
    }
```

The `itk::PointSetToImageFilter` takes the deformed mesh and produce a binary image corresponding to the node of the mesh. Note that only the nodes are producing the image and not the cells. See the section on SpatialObjects to produce a complete binary image from cells using the `itk::MeshSpatialObject` combined with the `itk::SpatialObjectToImageFilter`. However, using SpatialObjects is computationally more expensive.

```
  typedef itk::PointSetToImageFilter<MeshType,ImageType> MeshFilterType;
  MeshFilterType::Pointer meshFilter = MeshFilterType::New();
  meshFilter->SetOrigin(mask->GetOrigin());
  meshFilter->SetSize(mask->GetLargestPossibleRegion().GetSize());
  meshFilter->SetSpacing(mask->GetSpacing());
  meshFilter->SetInput(meshSource->GetOutput());
  try
    {
```

```
    meshFilter->Update();
    }
  catch( itk::ExceptionObject & excep )
    {
    std::cerr << "Exception Caught !" << std::endl;
    std::cerr << excep << std::endl;
    }
```

The resulting deformed binary mask can be written on disk using the `itk::ImageFileWriter`.

```
  typedef itk::ImageFileWriter<ImageType> WriterType;
  WriterType::Pointer writer = WriterType::New();
  writer->SetInput(meshFilter->GetOutput());
  writer->SetFileName(argv[3]);
  writer->Update();
```

Note that in order to successfully segment images, input parameters must be adjusted to reflect the characteristics of the data. The output of the filter is an Mesh. Users can use their own visualization packages to see the segmentation results.

Gibbs Prior Image Filter

The source code for this section can be found in the file
`Examples/Segmentation/GibbsPriorImageFilter1.cxx`.

This example illustrates the use of the `itk::RGBGibbsPriorFilter`. The filter outputs a binary segmentation that can be improved by the deformable model. It is the first part of our hybrid framework.

First, we include the appropriate header file.

```
#include "itkRGBGibbsPriorFilter.h"
```

The input is a single channel 2D image; the channel number is `NUMBANDS` = 1, and `NDIMENSION` is set to 3.

```
  const unsigned short NUMBANDS = 1;
  const unsigned short NDIMENSION = 3;

  typedef itk::Image<itk::Vector<unsigned short,NUMBANDS>,NDIMENSION> VecImageType;
```

The Gibbs prior segmentation is performed first to generate a rough segmentation that yields a sample of tissue from a region to be segmented, which will be combined to form the input for the isocontouring method. We define the pixel type of the output of the Gibbs prior filter to be `unsigned short`.

```
typedef itk::Image< unsigned short, NDIMENSION > ClassImageType;
```

Then we define the classifier that is needed for the Gibbs prior model to make correct segmenting decisions.

```
typedef itk::ImageClassifierBase< VecImageType, ClassImageType > ClassifierType;
typedef itk::ClassifierBase<VecImageType>::Pointer ClassifierBasePointer;

typedef ClassifierType::Pointer ClassifierPointer;
ClassifierPointer myClassifier = ClassifierType::New();
```

After that we can define the multi-channel Gibbs prior model.

```
typedef itk::RGBGibbsPriorFilter<VecImageType,ClassImageType>
  GibbsPriorFilterType;
GibbsPriorFilterType::Pointer applyGibbsImageFilter =
  GibbsPriorFilterType::New();
```

The parameters for the Gibbs prior filter are defined below. `NumberOfClasses` indicates how many different objects are in the image. The maximum number of iterations is the number of minimization steps. `ClusterSize` sets the lower limit on the object's size. The boundary gradient is the estimate of the variance between objects and background at the boundary region.

```
applyGibbsImageFilter->SetNumberOfClasses(NUM_CLASSES);
applyGibbsImageFilter->SetMaximumNumberOfIterations(MAX_NUM_ITER);
applyGibbsImageFilter->SetClusterSize(10);
applyGibbsImageFilter->SetBoundaryGradient(6);
applyGibbsImageFilter->SetObjectLabel(1);
```

We now set the input classifier for the Gibbs prior filter and the input to the classifier. The classifier will calculate the mean and variance of the object using the class image, and the results will be used as parameters for the Gibbs prior model.

```
applyGibbsImageFilter->SetInput(vecImage);
applyGibbsImageFilter->SetClassifier( myClassifier );
applyGibbsImageFilter->SetTrainingImage(trainingimagereader->GetOutput());
```

Finally we execute the Gibbs prior filter using the Update() method.

```
applyGibbsImageFilter->Update();
```

We execute this program on the image `brainweb89.png`. The following parameters are passed to the command line:

```
GibbsGuide.exe brainweb89.png brainweb89_train.png brainweb_gp.png
```

`brainweb89train` is a training image that helps to estimate the object statistics.

Note that in order to successfully segment other images, one has to create suitable training images for them. We can also segment color (RGB) and other multi-channel images.

9.5 Feature Extraction

Extracting salient features from images is an important task on image processing. It is typically used for guiding segmentation methods, preparing data for registration methods, or as a mechanism for recognizing anatomical structures in images. The following section introduce some of the feature extraction methods available in ITK.

9.5.1 Hough Transform

The Hough transform is a widely used technique for detection of geometrical features in images. It is based on mapping the image into a parametric space in which it may be easier to identify if particular geometrical features are present in the image. The transformation is specific for each desired geometrical shape.

Line Extraction

The source code for this section can be found in the file
`Examples/Segmentation/HoughTransform2DLinesImageFilter.cxx`.

This example illustrates the use of the `itk::HoughTransform2DLinesImageFilter` to find straight lines in a 2-dimensional image.

First, we include the header files of the filter.

```
#include "itkHoughTransform2DLinesImageFilter.h"
```

Next, we declare the pixel type and image dimension and specify the image type to be used as input. We also specify the image type of the accumulator used in the Hough transform filter.

```
  typedef   unsigned char   PixelType;
  typedef   float           AccumulatorPixelType;
  const     unsigned int    Dimension = 2;

  typedef itk::Image< PixelType, Dimension >  ImageType;
  typedef itk::Image< AccumulatorPixelType, Dimension > AccumulatorImageType;
```

We setup a reader to load the input image.

```
typedef  itk::ImageFileReader< ImageType > ReaderType;
ReaderType::Pointer reader = ReaderType::New();

reader->SetFileName( argv[1] );
try
  {
  reader->Update();
  }
catch( itk::ExceptionObject & excep )
  {
  std::cerr << "Exception caught !" << std::endl;
  std::cerr << excep << std::endl;
  }
ImageType::Pointer localImage = reader->GetOutput();
```

Once the image is loaded, we apply a `itk::GradientMagnitudeImageFilter` to segment edges. This casts the input image using a `itk::CastImageFilter`.

```
typedef itk::CastImageFilter< ImageType, AccumulatorImageType >
  CastingFilterType;
CastingFilterType::Pointer caster = CastingFilterType::New();

std::cout << "Applying gradient magnitude filter" << std::endl;

typedef itk::GradientMagnitudeImageFilter<AccumulatorImageType,
             AccumulatorImageType > GradientFilterType;
GradientFilterType::Pointer gradFilter =  GradientFilterType::New();

caster->SetInput(localImage);
gradFilter->SetInput(caster->GetOutput());
gradFilter->Update();
```

The next step is to apply a threshold filter on the gradient magnitude image to keep only bright values. Only pixels with a high value will be used by the Hough transform filter.

```
std::cout << "Thresholding" << std::endl;
typedef itk::ThresholdImageFilter<AccumulatorImageType> ThresholdFilterType;
ThresholdFilterType::Pointer threshFilter = ThresholdFilterType::New();

threshFilter->SetInput( gradFilter->GetOutput());
threshFilter->SetOutsideValue(0);
unsigned char threshBelow = 0;
unsigned char threshAbove = 255;
threshFilter->ThresholdOutside(threshBelow,threshAbove);
threshFilter->Update();
```

We create the HoughTransform2DLinesImageFilter based on the pixel type of the input image (the resulting image from the ThresholdImageFilter).

```
std::cout << "Computing Hough Map" << std::endl;
typedef itk::HoughTransform2DLinesImageFilter<AccumulatorPixelType,
             AccumulatorPixelType>  HoughTransformFilterType;

HoughTransformFilterType::Pointer houghFilter = HoughTransformFilterType::New();
```

We set the input to the filter to be the output of the ThresholdImageFilter. We set also the number of lines we are looking for. Basically, the filter computes the Hough map, blurs it using a certain variance and finds maxima in the Hough map. After a maximum is found, the local neighborhood, a circle, is removed from the Hough map. SetDiscRadius() defines the radius of this disc.

The output of the filter is the accumulator.

```
houghFilter->SetInput(threshFilter->GetOutput());
houghFilter->SetNumberOfLines(atoi(argv[3]));

if(argc > 4 )
  {
  houghFilter->SetVariance(atof(argv[4]));
  }

if(argc > 5 )
  {
  houghFilter->SetDiscRadius(atof(argv[5]));
  }
houghFilter->Update();
AccumulatorImageType::Pointer localAccumulator = houghFilter->GetOutput();
```

We can also get the lines as `itk::LineSpatialObject`. The `GetLines()` function return a list of those.

```
HoughTransformFilterType::LinesListType lines;
lines = houghFilter->GetLines(atoi(argv[3]));
std::cout << "Found " << lines.size() << " line(s)." << std::endl;
```

We can then allocate an image to draw the resulting lines as binary objects.

```
typedef  unsigned char    OutputPixelType;
typedef  itk::Image< OutputPixelType, Dimension > OutputImageType;

OutputImageType::Pointer  localOutputImage = OutputImageType::New();
```

```
OutputImageType::RegionType region;
region.SetSize(localImage->GetLargestPossibleRegion().GetSize());
region.SetIndex(localImage->GetLargestPossibleRegion().GetIndex());
localOutputImage->SetRegions( region );
localOutputImage->SetOrigin(localImage->GetOrigin());
localOutputImage->SetSpacing(localImage->GetSpacing());
localOutputImage->Allocate();
localOutputImage->FillBuffer(0);
```

We iterate through the list of lines and we draw them.

```
typedef HoughTransformFilterType::LinesListType::const_iterator LineIterator;
LineIterator itLines = lines.begin();
while( itLines != lines.end() )
  {
```

We get the list of points which consists of two points to represent a straight line. Then, from these two points, we compute a fixed point u and a unit vector $\vec{v}$ to parameterize the line.

```
  typedef HoughTransformFilterType::LineType::PointListType  PointListType;

  PointListType                   pointsList = (*itLines)->GetPoints();
  PointListType::const_iterator   itPoints = pointsList.begin();

  double u[2];
  u[0] = (*itPoints).GetPosition()[0];
  u[1] = (*itPoints).GetPosition()[1];
  itPoints++;
  double v[2];
  v[0] = u[0]-(*itPoints).GetPosition()[0];
  v[1] = u[1]-(*itPoints).GetPosition()[1];

  double norm = sqrt(v[0]*v[0]+v[1]*v[1]);
  v[0] /= norm;
  v[1] /= norm;
```

We draw a white pixels in the output image to represent the line.

```
  ImageType::IndexType localIndex;
  itk::Size<2> size = localOutputImage->GetLargestPossibleRegion().GetSize();
  float diag = sqrt((float)( size[0]*size[0] + size[1]*size[1] ));

  for(int i=static_cast<int>(-diag); i<static_cast<int>(diag); i++)
    {
    localIndex[0]=(long int)(u[0]+i*v[0]);
```

```
      localIndex[1]=(long int)(u[1]+i*v[1]);

      OutputImageType::RegionType region =
                          localOutputImage->GetLargestPossibleRegion();

      if( region.IsInside( localIndex ) )
        {
        localOutputImage->SetPixel( localIndex, 255 );
        }
      }
    itLines++;
    }
```

We setup a writer to write out the binary image created.

```
  typedef  itk::ImageFileWriter<  OutputImageType  > WriterType;
  WriterType::Pointer writer = WriterType::New();
  writer->SetFileName( argv[2] );
  writer->SetInput( localOutputImage );

  try
    {
    writer->Update();
    }
  catch( itk::ExceptionObject & excep )
    {
    std::cerr << "Exception caught !" << std::endl;
    std::cerr << excep << std::endl;
    }
```

Circle Extraction

The source code for this section can be found in the file
`Examples/Segmentation/HoughTransform2DCirclesImageFilter.cxx`.

This example illustrates the use of the `itk::HoughTransform2DCirclesImageFilter` to find circles in a 2-dimensional image.

First, we include the header files of the filter.

```
#include "itkHoughTransform2DCirclesImageFilter.h"
```

Next, we declare the pixel type and image dimension and specify the image type to be used as input. We also specify the image type of the accumulator used in the Hough transform filter.

```
  typedef   unsigned char   PixelType;
```

```
typedef   float           AccumulatorPixelType;
const     unsigned int    Dimension = 2;
typedef itk::Image< PixelType, Dimension >  ImageType;
ImageType::IndexType localIndex;
typedef itk::Image< AccumulatorPixelType, Dimension > AccumulatorImageType;
```

We setup a reader to load the input image.

```
typedef  itk::ImageFileReader< ImageType > ReaderType;
ReaderType::Pointer reader = ReaderType::New();
reader->SetFileName( argv[1] );
try
  {
  reader->Update();
  }
catch( itk::ExceptionObject & excep )
  {
  std::cerr << "Exception caught !" << std::endl;
  std::cerr << excep << std::endl;
  }
ImageType::Pointer localImage = reader->GetOutput();
```

We create the HoughTransform2DCirclesImageFilter based on the pixel type of the input image (the resulting image from the ThresholdImageFilter).

```
std::cout << "Computing Hough Map" << std::endl;

typedef itk::HoughTransform2DCirclesImageFilter<PixelType,
             AccumulatorPixelType> HoughTransformFilterType;
HoughTransformFilterType::Pointer houghFilter = HoughTransformFilterType::New();
```

We set the input of the filter to be the output of the ImageFileReader. We set also the number of circles we are looking for. Basically, the filter computes the Hough map, blurs it using a certain variance and finds maxima in the Hough map. After a maximum is found, the local neighborhood, a circle, is removed from the Hough map. SetDiscRadiusRatio() defines the radius of this disc proportional to the radius of the disc found. The Hough map is computed by looking at the points above a certain threshold in the input image. Then, for each point, a Gaussian derivative function is computed to find the direction of the normal at that point. The standard deviation of the derivative function can be adjusted by SetSigmaGradient(). The accumulator is filled by drawing a line along the normal and the length of this line is defined by the minimum radius (SetMinimumRadius()) and the maximum radius (SetMaximumRadius()). Moreover, a sweep angle can be defined by SetSweepAngle() (default 0.0) to increase the accuracy of detection.

The output of the filter is the accumulator.

```
houghFilter->SetInput( reader->GetOutput() );
```

```
  houghFilter->SetNumberOfCircles( atoi(argv[3]) );
  houghFilter->SetMinimumRadius(   atof(argv[4]) );
  houghFilter->SetMaximumRadius(   atof(argv[5]) );

  if( argc > 6 )
    {
    houghFilter->SetSweepAngle( atof(argv[6]) );
    }
  if( argc > 7 )
    {
    houghFilter->SetSigmaGradient( atoi(argv[7]) );
    }
  if( argc > 8 )
    {
    houghFilter->SetVariance( atof(argv[8]) );
    }
  if( argc > 9 )
    {
    houghFilter->SetDiscRadiusRatio( atof(argv[9]) );
    }

  houghFilter->Update();
  AccumulatorImageType::Pointer localAccumulator = houghFilter->GetOutput();
```

We can also get the circles as `itk::EllipseSpatialObject`. The `GetCircles()` function return a list of those.

```
  HoughTransformFilterType::CirclesListType circles;
  circles = houghFilter->GetCircles( atoi(argv[3]) );
  std::cout << "Found " << circles.size() << " circle(s)." << std::endl;
```

We can then allocate an image to draw the resulting circles as binary objects.

```
  typedef  unsigned char    OutputPixelType;
  typedef  itk::Image< OutputPixelType, Dimension > OutputImageType;

  OutputImageType::Pointer  localOutputImage = OutputImageType::New();

  OutputImageType::RegionType region;
  region.SetSize(localImage->GetLargestPossibleRegion().GetSize());
  region.SetIndex(localImage->GetLargestPossibleRegion().GetIndex());
  localOutputImage->SetRegions( region );
  localOutputImage->SetOrigin(localImage->GetOrigin());
  localOutputImage->SetSpacing(localImage->GetSpacing());
  localOutputImage->Allocate();
  localOutputImage->FillBuffer(0);
```

We iterate through the list of circles and we draw them.

```
typedef HoughTransformFilterType::CirclesListType CirclesListType;
CirclesListType::const_iterator itCircles = circles.begin();

while( itCircles != circles.end() )
  {
  std::cout << "Center: ";
  std::cout << (*itCircles)->GetObjectToParentTransform()->GetOffset()
            << std::endl;
  std::cout << "Radius: " << (*itCircles)->GetRadius()[0] << std::endl;
```

We draw white pixels in the output image to represent each circle.

```
  for(double angle = 0;angle <= 2*vnl_math::pi; angle += vnl_math::pi/60.0 )
    {
    localIndex[0] =
       (long int)((*itCircles)->GetObjectToParentTransform()->GetOffset()[0]
                            + (*itCircles)->GetRadius()[0]*cos(angle));
    localIndex[1] =
       (long int)((*itCircles)->GetObjectToParentTransform()->GetOffset()[1]
                            + (*itCircles)->GetRadius()[0]*sin(angle));
    OutputImageType::RegionType region =
                            localOutputImage->GetLargestPossibleRegion();

    if( region.IsInside( localIndex ) )
      {
      localOutputImage->SetPixel( localIndex, 255 );
      }
    }
  itCircles++;
  }
```

We setup a writer to write out the binary image created.

```
typedef  itk::ImageFileWriter< ImageType  > WriterType;
WriterType::Pointer writer = WriterType::New();

writer->SetFileName( argv[2] );
writer->SetInput(localOutputImage );

try
  {
  writer->Update();
  }
catch( itk::ExceptionObject & excep )
```

```
{
std::cerr << "Exception caught !" << std::endl;
std::cerr << excep << std::endl;
}
```

CHAPTER

TEN

Statistics

This chapter introduces the statistics functionalities in Insight. The statistics subsystem's primary purpose is to provide general capabilities for statistical pattern classification. However, its use is not limited for classification. Users might want to use data containers and algorithms in the statistics subsystem to perform other statistical analysis or to preprocessor image data for other tasks.

The statistics subsystem mainly consists of three parts: data container classes, statistical algorithms, and the classification framework. In this chapter, we will discuss each major part in that order.

10.1 Data Containers

An `itk::Statistics::Sample` object is a data container of elements that we call *measurement vectors*. A measurement vector is an array of values (of the same type) measured on an object (In images, it can be a vector of the gray intensity value and/or the gradient value of a pixel). Strictly speaking from the design of the Sample class, a measurement vector can be any class derived from `itk::FixedArray`, including FixedArray itself.

10.1.1 Sample Interface

The source code for this section can be found in the file
`Examples/Statistics/ListSample.cxx`.

This example illustrates the common interface of the `Sample` class in Insight.

Different subclasses of `itk::Statistics::Sample` expect different sets of template arguments. In this example, we use the `itk::Statistics::ListSample` class that requires the type of measurement vectors. The ListSample uses STL `vector` to store measurement vectors. This class conforms to the common interface of Sample. Most methods of the Sample class interface are for retrieving measurement vectors, the size of a container, and the total frequency. In

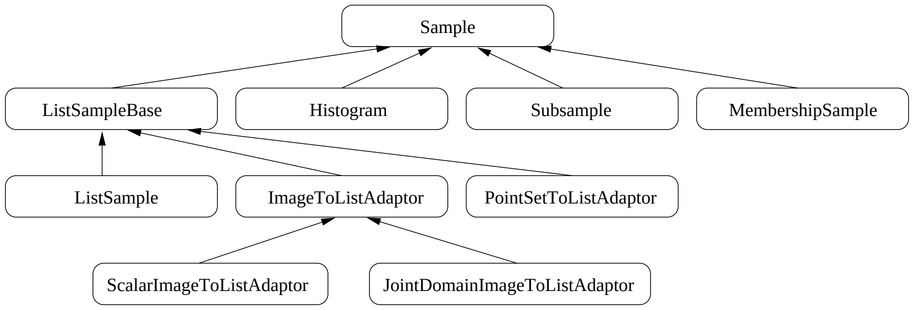


Figure 10.1: Sample class inheritance diagram.

this example, we will see those information retrieving methods in addition to methods specific to the ListSample class for data input.

To use the ListSample class, we include the header file for the class.

We need another header for measurement vectors. We are going to use the `itk::Vector` class which is a subclass of the `itk::FixedArray` class.

```
#include "itkListSample.h"
#include "itkVector.h"
```

The following code snippet defines the measurement vector type as three component `float` `itk::Vector`. The `MeasurementVectorType` is the measurement vector type in the `SampleType`. An object is instantiated at the third line.

```
  typedef itk::Vector< float, 3 > MeasurementVectorType ;
  typedef itk::Statistics::ListSample< MeasurementVectorType > SampleType ;
  SampleType::Pointer sample = SampleType::New() ;
```

In the above code snippet, the namespace specifier for ListSample is `itk::Statistics::` instead of the usual namespace specifier for other ITK classes, `itk::`.

The newly instantiated object does not have any data in it. We have two different ways of storing data elements. The first method is using the `PushBack` method.

```
  MeasurementVectorType mv ;
  mv[0] = 1.0 ;
  mv[1] = 2.0 ;
  mv[2] = 4.0 ;

  sample->PushBack(mv) ;
```

The previous code increases the size of the container by one and stores `mv` as the first data element in it.

The other way to store data elements is calling the `Resize` method and then calling the `SetMeasurementVector()` method with a measurement vector. The following code snippet increases the size of the container to three and stores two measurement vectors at the second and the third slot. The measurement vector stored using the `PushBack` method above is still at the first slot.

```
sample->Resize(3) ;

mv[0] = 2.0 ;
mv[1] = 4.0 ;
mv[2] = 5.0 ;
sample->SetMeasurementVector(1, mv) ;

mv[0] = 3.0 ;
mv[1] = 8.0 ;
mv[2] = 6.0 ;
sample->SetMeasurementVector(2, mv) ;
```

Now that we have seen how to create an ListSample object and store measurement vectors using the ListSample-specific interface. The following code shows the common interface of the Sample class. The `Size` method returns the number of measurement vectors in the sample. The primary data stored in Sample subclasses are measurement vectors. However, each measurement vector has its associated frequency of occurrence within the sample. For the ListSample and the adaptor classes (see Section 10.1.2), the frequency value is always one. `itk::Statistics::Histogram` can have a varying frequency (`float` type) for each measurement vector. We retrieve measurement vectors using the `GetMeasurementVector(unsigned long instance identifier)`, and frequency using the `GetFrequency(unsigned long instance identifier)`.

```
for ( unsigned long i = 0 ; i < sample->Size() ; ++i )
  {
  std::cout << "id = " << i
            << "\t measurement vector = "
            << sample->GetMeasurementVector(i)
            << "\t frequency = "
            << sample->GetFrequency(i)
            << std::endl ;
  }
```

The output should look like the following:

```
id = 0 measurement vector = 1 2 4 frequency = 1
id = 1 measurement vector = 2 4 5 frequency = 1
id = 2 measurement vector = 3 8 6 frequency = 1
```

We can get the same result with its iterator.

```
SampleType::Iterator iter = sample->Begin() ;

while( iter != sample->End() )
  {
  std::cout << "id = " << iter.GetInstanceIdentifier()
            << "\t measurement vector = "
            << iter.GetMeasurementVector()
            << "\t frequency = "
            << iter.GetFrequency()
            << std::endl ;
  ++iter ;
  }
```

The last method defined in the Sample class is the `GetTotalFrequency()` method that returns the sum of frequency values associated with every measurement vector in a container. In the case of ListSample and the adaptor classes, the return value should be exactly the same as that of the `Size()` method, because the frequency values are always one for each measurement vector. However, for the `itk::Statistics::Histogram`, the frequency values can vary. Therefore, if we want to develop a general algorithm to calculate the sample mean, we must use the `GetTotalFrequency()` method instead of the `Size()` method.

```
std::cout << "Size = " << sample->Size() << std::endl ;
std::cout << "Total frequency = "
          << sample->GetTotalFrequency() << std::endl ;
```

10.1.2 Sample Adaptors

There are two adaptor classes that provide the common `itk::Statistics::Sample` interfaces for `itk::Image` and `itk::PointSet`, two fundamental data container classes found in ITK. The adaptor classes do not store any real data elements themselves. These data comes from the source data container plugged into them. First, we will describe how to create an `itk::Statistics::ImageToListAdaptor` and then an `itk::statistics::PointSetToListAdaptor` object.

ImageToListAdaptor

The source code for this section can be found in the file
`Examples/Statistics/ImageToListAdaptor.cxx`.

This example shows how to instantiate an `itk::Statistics::ImageToListAdaptor` object and plug-in an `itk::Image` object as the data source for the adaptor.

In this example, we use the ImageToListAdaptor class that requires the input type of Image as the template argument. To users of the ImageToListAdaptor, the pixels of the input image are treated as measurement vectors. The ImageToListAdaptor is one of two adaptor classes among the subclasses of the itk::Statistics::Sample. That means an ImageToListAdaptor object does not store any real data. The data comes from other ITK data container classes. In this case, an instance of the Image class is the source of the data.

To use an ImageToListAdaptor object, include the header file for the class. Since we are using an adaptor, we also should include the header file for the Image class. For illustration, we use the itk::RandomImageSource that generates an image with random pixel values. So, we need to include the header file for this class. Another convenient filter is the itk::ScalarToArrayCastImageFilter which creates an image with pixels of array type from one or more input images have pixels of scalar type. Since an element of a Sample object is a measurement *vector*, you cannot plug-in an image of scalar pixels. However, if we want to use an image with scalar pixels without the help from the ScalarToArrayCastImageFilter, we can use the itk::Statistics::ScalarImageToListAdaptor class that is derived from the itk::Statistics::ImageToListAdaptor. The usage of the ScalarImageToListAdaptor is identical to that of the ImageToListAdaptor.

```
#include "itkImageToListAdaptor.h"
#include "itkImage.h"
#include "itkRandomImageSource.h"
#include "itkScalarToArrayCastImageFilter.h"
```

We assume you already know how to create an image (see Section 4.1.1). The following code snippet will create a 2D image of float pixels filled with random values.

```
  typedef itk::Image<float,2> FloatImage2DType;

  itk::RandomImageSource<FloatImage2DType>::Pointer random;
  random = itk::RandomImageSource<FloatImage2DType>::New();

  random->SetMin(    0.0 );
  random->SetMax( 1000.0 );

  unsigned long size[2] = {20, 20};
  random->SetSize( size );

  float spacing[2] = {0.7, 2.1};
  random->SetSpacing( spacing );

  float origin[2] = {15, 400};
  random->SetOrigin( origin );
```

We now have an instance of Image and need to cast it to an Image object with an array pixel type (anything derived from the itk::FixedArray class such as itk::Vector, itk::Point, itk::RGBPixel, and itk::CovariantVector).

Since in this example the image pixel type is `float`, we will use single element a `float` FixedArray as our measurement vector type. And that will also be our pixel type for the cast filter.

```
  typedef itk::FixedArray< float, 1 > MeasurementVectorType;
  typedef itk::Image< MeasurementVectorType, 2 > ArrayImageType;
  typedef itk::ScalarToArrayCastImageFilter< FloatImage2DType, ArrayImageType >
    CasterType;

  CasterType::Pointer caster = CasterType::New();
  caster->SetInput( random->GetOutput() );
  caster->Update();
```

Up to now, we have spent most of our time creating an image suitable for the adaptor. Actually, the hard part of this example is done. Now, we just define an adaptor with the image type and instantiate an object.

```
  typedef itk::Statistics::ImageToListAdaptor< ArrayImageType > SampleType;
  SampleType::Pointer sample = SampleType::New();
```

The final task is to plug in the image object to the adaptor. After that, we can use the common methods and iterator interfaces shown in Section 10.1.1.

```
  sample->SetImage( caster->GetOutput() );
```

If we are interested only in pixel values, the ScalarImageToListAdaptor (scalar pixels) and the ImageToListAdaptor (vector pixels) would be sufficient. However, if we want to perform some statistical analysis on spatial information (image index or pixel's physical location) and pixel values altogether, we want to have a measurement vector that consists of a pixel's value and physical position. In that case, we can use the `itk::Statistics::JointDomainImageToListAdaptor` class. With that class, when we call the `GetMeasurementVector()` method, the returned measurement vector is composed of the physical coordinates and pixel values. The usage is almost the same as with ImageToListAdaptor. One important difference between JointDomainImageToListAdaptor and the other two image adaptors is that the JointDomainImageToListAdaptor is the `SetNormalizationFactors()` method. Each component of a measurement vector from the JointDomainImageToListAdaptor is divided by the corresponding component value from the supplied normalization factors.

PointSetToListAdaptor

The source code for this section can be found in the file
`Examples/Statistics/PointSetToListAdaptor.cxx`.

We will describe how to use `itk::PointSet` as a `itk::Statistics::Sample` using an adaptor in this example.

The `itk::Statistics::PointSetToListAdaptor` class requires a PointSet as input. The PointSet class is an associative data container. Each point in a PointSet object can have an associated optional data value. For the statistics subsystem, the current implementation of PointSetToListAdaptor takes only the point part into consideration. In other words, the measurement vectors from a PointSetToListAdaptor object are points from the PointSet object that is plugged into the adaptor object.

To use an PointSetToListAdaptor class, we include the header file for the class.

```
#include "itkPointSetToListAdaptor.h"
```

Since we are using an adaptor, we also include the header file for the PointSet class.

```
#include "itkPointSet.h"
#include "itkVector.h"
```

Next we create a PointSet object (see Section 4.2.1 otherwise). The following code snippet will create a PointSet object that stores points (its coordinate value type is float) in 3D space.

```
  typedef itk::PointSet< short > PointSetType;
  PointSetType::Pointer pointSet = PointSetType::New();
```

Note that the `short` type used in the declaration of `PointSetType` pertains to the pixel type associated with every point, not to the type used to represent point coordinates. If we want to change the type of point in terms of the coordinate value and/or dimension, we have to modify the `TMeshTraits` (one of the optional template arguments for the `PointSet` class). The easiest way of create a custom mesh traits instance is to specialize the existing `itk::DefaultStaticMeshTraits`. By specifying the `TCoordRep` template argument, we can change the coordinate value type of a point. By specifying the `VPointDimension` template argument, we can change the dimension of the point. As mentioned earlier, a `PointSetToListAdaptor` object cares only about the points, and the type of measurement vectors is the type of points. Therefore, we can define the measurement vector type as in the following code snippet.

```
  typedef PointSetType::PointType MeasurementVectorType;
```

To make the example a little bit realistic, we add two point into the `pointSet`.

```
  PointSetType::PointType point;
  point[0] = 1.0;
  point[1] = 2.0;
  point[2] = 3.0;

  pointSet->SetPoint( 0UL, point);

  point[0] = 2.0;
  point[1] = 4.0;
  point[2] = 6.0;

  pointSet->SetPoint( 1UL, point );
```

Now we have a PointSet object that has two points in it. And the pointSet is ready to be plugged into the adaptor. First, we create an instance of the PointSetToListAdaptor class with the type of the input PointSet object.

```
  typedef itk::Statistics::PointSetToListAdaptor< PointSetType > SampleType;
  SampleType::Pointer sample = SampleType::New();
```

Second, just as we did with the ImageToListAdaptor example in Section 10.1.2, all we have to do is to plug in the PointSet object to the adaptor. After that, we can use the common methods and iterator interfaces shown in Section 10.1.1.

```
  sample->SetPointSet( pointSet );
```

The source code for this section can be found in the file
`Examples/Statistics/PointSetToAdaptor.cxx`.

We will describe how to use `itk::PointSet` as a `Sample` using an adaptor in this example.

`itk::Statistics::PointSetToListAdaptor` class requires the type of input `itk::PointSet` object. The `itk::PointSet` class is an associative data container. Each point in a `PointSet` object can have its associated data value (optional). For the statistics subsystem, current implementation of `PointSetToListAdaptor` takes only the point part into consideration. In other words, the measurement vectors from a `PointSetToListAdaptor` object are points from the `PointSet` object that is plugged-into the adaptor object.

To use, an `itk::PointSetToListAdaptor` object, we include the header file for the class.

```
#include "itkPointSetToListAdaptor.h"
```

Since, we are using an adaptor, we also include the header file for the `itk::PointSet` class.

```
#include "itkPointSet.h"
```

We assume you already know how to create an `itk::PointSet` object. The following code snippet will create a 2D image of float pixels filled with random values.

```
typedef itk::PointSet<float,2> FloatPointSet2DType ;

itk::RandomPointSetSource<FloatPointSet2DType>::Pointer random ;
random = itk::RandomPointSetSource<FloatPointSet2DType>::New() ;
random->SetMin(0.0) ;
random->SetMax(1000.0) ;

unsigned long size[2] = {20, 20} ;
random->SetSize(size) ;
float spacing[2] = {0.7, 2.1} ;
random->SetSpacing( spacing ) ;
float origin[2] = {15, 400} ;
random->SetOrigin( origin ) ;
```

We now have an `itk::PointSet` object and need to cast it to an `itk::PointSet` object with array type (anything derived from the `itk::FixedArray` class) pixels.

Since, the `itk::PointSet` object's pixel type is `float`, We will use single element `float` `itk::FixedArray` as our measurement vector type. And that will also be our pixel type for the cast filter.

```
typedef itk::FixedArray< float, 1 > MeasurementVectorType ;
typedef itk::PointSet< MeasurementVectorType, 2 > ArrayPointSetType ;
typedef itk::ScalarToArrayCastPointSetFilter< FloatPointSet2DType,
  ArrayPointSetType > CasterType ;

CasterType::Pointer caster = CasterType::New() ;
caster->SetInput( random->GetOutput() ) ;
caster->Update() ;
```

Up to now, we spend most of time to prepare an `itk::PointSet` object suitable for the adaptor. Actually, the hard part of this example is done. Now, we must define an adaptor with the image type and instantiate an object.

```
typedef itk::Statistics::PointSetToListAdaptor< ArrayPointSetType > SampleType ;
SampleType::Pointer sample = SampleType::New() ;
```

The final thing we have to is to plug-in the image object to the adaptor. After that, we can use the common methods and iterator interfaces shown in 10.1.1.

```
sample->SetPointSet( caster->GetOutput() ) ;
```

10.1.3 Histogram

The source code for this section can be found in the file
`Examples/Statistics/Histogram.cxx`.

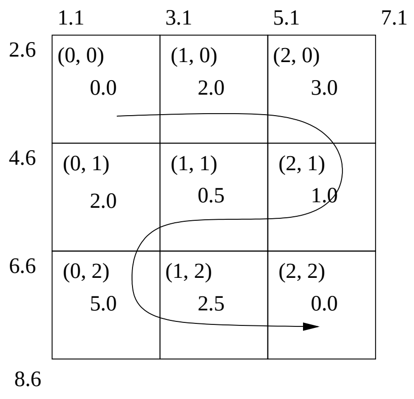


Figure 10.2: Conceptual histogram data structure.

This example shows how to create an itk::Statistics::Histogram object and use it.

We call an instance in a Histogram object a *bin*. The Histogram differs from the itk::Statistics::ListSample, itk::Statistics::ImageToListAdaptor, or itk::Statistics::PointSetToListAdaptor in significant ways. Histogram can have a variable number of values (float type) for each measurement vector, while the three other classes have a fixed value (one) for all measurement vectors. Also those array-type containers can have multiple instances (data elements) that have identical measurement vector values. However, in a Histogram object, there is one unique instance for any given measurement vector.

```
#include "itkHistogram.h"
```

Here we create a histogram with 2-component measurement vectors.

```
  typedef float MeasurementType ;
  typedef itk::Statistics::Histogram< MeasurementType, 2 > HistogramType ;
  HistogramType::Pointer histogram = HistogramType::New() ;
```

We initialize it as a 3×3 histogram with equal size intervals.

```
  HistogramType::SizeType size ;
  size.Fill(3) ;
  HistogramType::MeasurementVectorType lowerBound ;
  HistogramType::MeasurementVectorType upperBound ;
  lowerBound[0] = 1.1 ;
  lowerBound[1] = 2.6 ;
  upperBound[0] = 7.1 ;
  upperBound[1] = 8.6 ;

  histogram->Initialize(size, lowerBound, upperBound ) ;
```

Now the histogram is ready for storing frequency values. We will fill the each bin's frequency according to the Figure 10.2. There are three ways of accessing data elements in the histogram:

- using instance identifiers—just like any other Sample object;
- using n-dimensional indices—just like an Image object;
- using an iterator—just like any other Sample object.

In this example, the index $(0,0)$ refers the same bin as the instance identifier (0) refers to. The instance identifier of the index (0, 1) is (3), (0, 2) is (6), (2, 2) is (8), and so on.

```
histogram->SetFrequency(0UL, 0.0) ;
histogram->SetFrequency(1UL, 2.0) ;
histogram->SetFrequency(2UL, 3.0) ;
histogram->SetFrequency(3UL, 2.0) ;
histogram->SetFrequency(4UL, 0.5) ;
histogram->SetFrequency(5UL, 1.0) ;
histogram->SetFrequency(6UL, 5.0) ;
histogram->SetFrequency(7UL, 2.5) ;
histogram->SetFrequency(8UL, 0.0) ;
```

Let us examine if the frequency is set correctly by calling the `GetFrequency(index)` method. We can use the `GetFrequency(instance identifier)` method for the same purpose.

```
HistogramType::IndexType index ;
index[0] = 0 ;
index[1] = 2 ;
std::cout << "Frequency of the bin at index  " << index
          << " is " << histogram->GetFrequency(index)
          << ", and the bin's instance identifier is "
          << histogram->GetInstanceIdentifier(index) << std::endl ;
```

For test purposes, we create a measurement vector and an index that belongs to the center bin.

```
HistogramType::MeasurementVectorType mv ;
mv[0] = 4.1 ;
mv[1] = 5.6 ;
index.Fill(1) ;
```

We retrieve the measurement vector at the index value (1, 1), the center bin's measurement vector. The output is [4.1, 5.6].

```
std::cout << "Measurement vector at the center bin is "
          << histogram->GetMeasurementVector(index) << std::endl ;
```

Since all the measurement vectors are unique in the Histogram class, we can determine the index from a measurement vector.

```
  HistogramType::IndexType resultingIndex;
  histogram->GetIndex(mv,resultingIndex);
  std::cout << "Index of the measurement vector " << mv
            << " is " << resultingIndex << std::endl ;
```

In a similar way, we can get the instance identifier from the index.

```
  std::cout << "Instance identifier of index " << index
            << " is " << histogram->GetInstanceIdentifier(index)
            << std::endl ;
```

If we want to check if an index is a valid one, we use the method `IsIndexOutOfBounds(index)`. The following code snippet fills the index variable with (100, 100). It is obviously not a valid index.

```
  index.Fill(100) ;
  if ( histogram->IsIndexOutOfBounds(index) )
    {
    std::cout << "Index " << index << "is out of bounds." << std::endl ;
    }
```

The following code snippets show how to get the histogram size and frequency dimension.

```
  std::cout << "Number of bins = " << histogram->Size()
            << " Total frequency = " << histogram->GetTotalFrequency()
            << " Dimension sizes = " << histogram->GetSize() << std::endl ;
```

The Histogram class has a quantile calculation method, `Quantile(dimension, percent)`. The following code returns the 50th percentile along the first dimension. Note that the quantile calculation considers only one dimension.

```
  std::cout << "50th percentile along the first dimension = "
            << histogram->Quantile(0, 0.5) << std::endl ;
```

10.1.4 Subsample

The source code for this section can be found in the file
`Examples/Statistics/Subsample.cxx`.

The `itk::Statistics::Subsample` is a derived sample. In other words, it requires another `itk::Statistics::Sample` object for storing measurement vectors. The Subsample class stores a subset of instance identifiers from another Sample object. *Any* Sample's subclass can be the source Sample object. You can create a Subsample object out of another Subsample object. The Subsample class is useful for storing classification results from a test Sample object or for just extracting some part of interest in a Sample object. Another good use of Subsample is sorting a Sample object. When we use an `itk::Image` object as the data source, we do not want to change the order of data element in the image. However, we sometimes want to sort or select data elements according to their order. Statistics algorithms for this purpose accepts only Subsample objects as inputs. Changing the order in a Subsample object does not change the order of the source sample.

To use a Subsample object, we include the header files for the class itself and a Sample class. We will use the `itk::Statistics::ListSample` as the input sample.

```
#include "itkListSample.h"
#include "itkSubsample.h"
```

We need another header for measurement vectors. We are going to use the `itk::Vector` class in this example.

```
#include "itkVector.h"
```

The following code snippet will create a ListSample object with three-component float measurement vectors and put three measurement vectors into the list.

```
  typedef itk::Vector< float, 3 > MeasurementVectorType;
  typedef itk::Statistics::ListSample< MeasurementVectorType > SampleType;
  SampleType::Pointer sample = SampleType::New();
  MeasurementVectorType mv;
  mv[0] = 1.0;
  mv[1] = 2.0;
  mv[2] = 4.0;

  sample->PushBack(mv);

  mv[0] = 2.0;
  mv[1] = 4.0;
  mv[2] = 5.0;
  sample->PushBack(mv);

  mv[0] = 3.0;
  mv[1] = 8.0;
  mv[2] = 6.0;
  sample->PushBack(mv);
```

To create a Subsample instance, we define the type of the Subsample with the source sample type, in this case, the previously defined `SampleType`. As usual, after that, we call the

New() method to create an instance. We must plug in the source sample, sample, using the SetSample() method. However, with regard to data elements, the Subsample is empty. We specify which data elements, among the data elements in the Sample object, are part of the Subsample. There are two ways of doing that. First, if we want to include every data element (instance) from the sample, we simply call the InitializeWithAllInstances() method like the following:

```
     subsample->InitializeWithAllInstances();
```

This method is useful when we want to create a Subsample object for sorting all the data elements in a Sample object. However, in most cases, we want to include only a subset of a Sample object. For this purpose, we use the AddInstance(instance identifier) method in this example. In the following code snippet, we include only the first and last instance in our subsample object from the three instances of the Sample class.

```
  typedef itk::Statistics::Subsample< SampleType > SubsampleType;
  SubsampleType::Pointer subsample = SubsampleType::New();
  subsample->SetSample( sample );

  subsample->AddInstance( 0UL );
  subsample->AddInstance( 2UL );
```

The Subsample is ready for use. The following code snippet shows how to use Iterator interfaces.

```
  SubsampleType::Iterator iter = subsample->Begin();
  while ( iter != subsample->End() )
    {
    std::cout << "instance identifier = " << iter.GetInstanceIdentifier()
              << "\t measurement vector = "
              << iter.GetMeasurementVector()
              << "\t frequency = "
              << iter.GetFrequency()
              << std::endl;
    ++iter;
    }
```

As mentioned earlier, the instances in a Subsample can be sorted without changing the order in the source Sample. For this purpose, the Subsample provides an additional instance indexing scheme. The indexing scheme is just like the instance identifiers for the Sample. The index is an integer value starting at 0, and the last value is one less than the number of all instances in a Subsample. The Swap(0, 1) method, for example, swaps two instance identifiers of the first data element and the second element in the Subsample. Internally, the Swap() method changes the instance identifiers in the first and second position. Using indices, we can print out the effects of the `Swap()` method. We use the `GetMeasurementVectorByIndex(index)` to get the measurement vector at the index position. However, if we want to use the common methods of Sample that accepts instance identifiers, we call them after we get the instance identifiers using `GetInstanceIdentifier(index)` method.

```
subsample->Swap(0, 1);

for ( int index = 0 ; index < subsample->Size() ; ++index )
  {
  std::cout << "instance identifier = "
            << subsample->GetInstanceIdentifier(index)
            << "\t measurement vector = "
            << subsample->GetMeasurementVectorByIndex(index)
            << std::endl;
  }
```

Since we are using a ListSample object as the source sample, the following code snippet will return the same value (2) for the `Size()` and the `GetTotalFrequency()` methods. However, if we used a Histogram object as the source sample, the two return values might be different because a Histogram allows varying frequency values for each instance.

```
std::cout << "Size = " << subsample->Size() << std::endl;
std::cout << "Total frequency = "
          << subsample->GetTotalFrequency() << std::endl;
```

If we want to remove all instances that are associated with the Subsample, we call the `Clear()` method. After this invocation, the `Size()` and the `GetTotalFrequency()` methods return 0.

```
subsample->Clear();
std::cout << "Size = " << subsample->Size() << std::endl;
std::cout << "Total frequency = "
          << subsample->GetTotalFrequency() << std::endl;
```

10.1.5 MembershipSample

The source code for this section can be found in the file
`Examples/Statistics/MembershipSample.cxx`.

The `itk::Statistics::MembershipSample` is derived from the class `itk::Statistics::Sample` that associates a class label with each measurement vector. It needs another Sample object for storing measurement vectors. A `MembershipSample` object stores a subset of instance identifiers from another Sample object. *Any* subclass of Sample can be the source Sample object. The MembershipSample class is useful for storing classification results from a test Sample object. The MembershipSample class can be considered as an associative container that stores measurement vectors, frequency values, and *class labels*.

To use a MembershipSample object, we include the header files for the class itself and the Sample class. We will use the `itk::Statistics::ListSample` as the input sample. We need another header for measurement vectors. We are going to use the `itk::Vector` class which is a subclass of the `itk::FixedArray`.

```
#include "itkListSample.h"
#include "itkMembershipSample.h"
#include "itkVector.h"
```

The following code snippet will create a `ListSample` object with three-component float measurement vectors and put three measurement vectors in the `ListSample` object.

```
  typedef itk::Vector< float, 3 > MeasurementVectorType;
  typedef itk::Statistics::ListSample< MeasurementVectorType > SampleType;
  SampleType::Pointer sample = SampleType::New();
  MeasurementVectorType mv;

  mv[0] = 1.0;
  mv[1] = 2.0;
  mv[2] = 4.0;
  sample->PushBack(mv);

  mv[0] = 2.0;
  mv[1] = 4.0;
  mv[2] = 5.0;
  sample->PushBack(mv);

  mv[0] = 3.0;
  mv[1] = 8.0;
  mv[2] = 6.0;
  sample->PushBack(mv);
```

To create a MembershipSample instance, we define the type of the MembershipSample using the source sample type using the previously defined `SampleType`. As usual, after that, we call the `New()` method to create an instance. We must plug in the source sample, Sample, using the `SetSample()` method. We provide class labels for data instances in the Sample object using the `AddInstance()` method. As the required initialization step for the `membershipSample`, we must call the `SetNumberOfClasses()` method with the number of classes. We must add all instances in the source sample with their class labels. In the following code snippet, we set the first instance' class label to 0, the second to 0, the third (last) to 1. After this, the `membershipSample` has two `Subsample` objects. And the class labels for these two `Subsample` objects are 0 and 1. The 0 class `Subsample` object includes the first and second instances, and the 1 class includes the third instance.

```
typedef itk::Statistics::MembershipSample< SampleType >
  MembershipSampleType;

MembershipSampleType::Pointer membershipSample =
  MembershipSampleType::New();

membershipSample->SetSample(sample);
membershipSample->SetNumberOfClasses(2);

membershipSample->AddInstance(0U, 0UL );
membershipSample->AddInstance(0U, 1UL );
membershipSample->AddInstance(1U, 2UL );
```

The `Size()` and `GetTotalFrequency()` returns the same information that Sample does.

```
std::cout << "Size = " << membershipSample->Size() << std::endl;
std::cout << "Total frequency = "
          << membershipSample->GetTotalFrequency() << std::endl;
```

The `membershipSample` is ready for use. The following code snippet shows how to use the `Iterator` interface. The MembershipSample's `Iterator` has an additional method that returns the class label (`GetClassLabel()`).

```
MembershipSampleType::ConstIterator iter = membershipSample->Begin();
while ( iter != membershipSample->End() )
  {
  std::cout << "instance identifier = " << iter.GetInstanceIdentifier()
            << "\t measurement vector = "
            << iter.GetMeasurementVector()
            << "\t frequency = "
            << iter.GetFrequency()
            << "\t class label = "
            << iter.GetClassLabel()
            << std::endl;
  ++iter;
  }
```

To see the numbers of instances in each class subsample, we use the `GetClassSampleSize()` method.

```
std::cout << "class label = 0 sample size = "
          << membershipSample->GetClassSampleSize(0) << std::endl;
std::cout << "class label = 1 sample size = "
          << membershipSample->GetClassSampleSize(1) << std::endl;
```

We call the GetClassSample() method to get the class subsample in the membershipSample. The MembershipSampleType::ClassSampleType is actually a specialization of the itk::Statistics::Subsample. We print out the instance identifiers, measurement vectors, and frequency values that are part of the class. The output will be two lines for the two instances that belong to the class 0.

```
  MembershipSampleType::ClassSampleType::ConstPointer classSample =
                                  membershipSample->GetClassSample( 0 );

  MembershipSampleType::ClassSampleType::ConstIterator c_iter =
                                                  classSample->Begin();

  while ( c_iter != classSample->End() )
    {
    std::cout << "instance identifier = " << c_iter.GetInstanceIdentifier()
              << "\t measurement vector = "
              << c_iter.GetMeasurementVector()
              << "\t frequency = "
              << c_iter.GetFrequency() << std::endl;
    ++c_iter;
    }
```

10.1.6 MembershipSampleGenerator

The source code for this section can be found in the file
Examples/Statistics/MembershipSampleGenerator.cxx.

To use, an MembershipSample object, we include the header files for the class itself and a Sample class. We will use the ListSample as the input sample.

```
#include "itkListSample.h"
#include "itkMembershipSample.h"
```

We need another header for measurement vectors. We are going to use the itk::Vector class which is a subclass of the itk::FixedArray in this example.

```
#include "itkVector.h"
```

The following code snippet will create a ListSample object with three-component float measurement vectors and put three measurement vectors in the ListSample object.

```
  typedef itk::Vector< float, 3 > MeasurementVectorType ;
  typedef itk::Statistics::ListSample< MeasurementVectorType > SampleType ;
  SampleType::Pointer sample = SampleType::New() ;
  MeasurementVectorType mv ;
```

```
mv[0] = 1.0 ;
mv[1] = 2.0 ;
mv[2] = 4.0 ;
sample->PushBack(mv) ;

mv[0] = 2.0 ;
mv[1] = 4.0 ;
mv[2] = 5.0 ;
sample->PushBack(mv) ;

mv[0] = 3.0 ;
mv[1] = 8.0 ;
mv[2] = 6.0 ;
sample->PushBack(mv) ;
```

To create a `MembershipSample` instance, we define the type of the `MembershipSample` with the source sample type, in this case, previously defined `SampleType`. As usual, after that, we call `New()` method to instantiate an instance. We must plug-in the source sample, `sample` object using the `SetSample(source sample)` method. However, in regard of **class labels**, the `membershipSample` is empty. We provide class labels for data instances in the `sample` object using the `AddInstance(class label, instance identifier)` method. As the required initialization step for the `membershipSample`, we must call the `SetNumberOfClasses(number of classes)` method with the number of classes. We must add all instances in the source sample with their class labels. In the following code snippet, we set the first instance' class label to 0, the second to 0, the third (last) to 1. After this, the `membershipSample` has two `Subclass` objects. And the class labels for these two `Subclass` are 0 and 1. The **0** class `Subsample` object includes the first and second instances, and the **1** class includes the third instance.

```
typedef itk::Statistics::MembershipSample< SampleType >
  MembershipSampleType ;

MembershipSampleType::Pointer membershipSample =
  MembershipSampleType::New() ;

membershipSample->SetSample(sample) ;
membershipSample->SetNumberOfClasses(2) ;

membershipSample->AddInstance(0U, 0UL ) ;
membershipSample->AddInstance(0U, 1UL ) ;
membershipSample->AddInstance(1U, 2UL ) ;
```

The `Size()` and `GetTotalFrequency()` returns the same values as the `sample` does.

```
std::cout << "Size = " << membershipSample->Size() << std::endl ;
std::cout << "Total frequency = "
          << membershipSample->GetTotalFrequency() << std::endl ;
```

The membershipSample is ready for use. The following code snippet shows how to use Iterator interfaces. The MembershipSample' Iterator has an additional method that returns the class label (GetClassLabel()).

```
MembershipSampleType::Iterator iter = membershipSample->Begin() ;
while ( iter != membershipSample->End() )
  {
  std::cout << "instance identifier = " << iter.GetInstanceIdentifier()
            << "\t measurement vector = "
            << iter.GetMeasurementVector()
            << "\t frequency = "
            << iter.GetFrequency()
            << "\t class label = "
            << iter.GetClassLabel()
            << std::endl ;
  ++iter ;
  }
```

To see the numbers of instances in each class subsample, we use the GetClassSampleSize(class label) method.

```
std::cout << "class label = 0 sample size = "
          << membershipSample->GetClassSampleSize(0) << std::endl ;
std::cout << "class label = 1 sample size = "
          << membershipSample->GetClassSampleSize(0) << std::endl ;
```

We call the GetClassSample(class label) method to get the class subsample in the membershipSample. The MembershipSampleType::ClassSampleType is actually an specialization of the itk::Statistics::Subsample. We print out the instance identifiers, measurement vectors, and frequency values that are part of the class. The output will be two lines for the two instances that belongs to the class **0**. the GetClassSampleSize(class label) method.

```
MembershipSampleType::ClassSampleType::Pointer classSample =
  membershipSample->GetClassSample(0) ;
MembershipSampleType::ClassSampleType::Iterator c_iter =
  classSample->Begin() ;
while ( c_iter != classSample->End() )
  {
  std::cout << "instance identifier = " << c_iter.GetInstanceIdentifier()
            << "\t measurement vector = "
            << c_iter.GetMeasurementVector()
            << "\t frequency = "
            << c_iter.GetFrequency() << std::endl ;
  ++c_iter ;
  }
```

10.1.7 K-d Tree

The source code for this section can be found in the file
`Examples/Statistics/KdTree.cxx`.

The `itk::Statistics::KdTree` implements a data structure that separates samples in a *k*-dimension space. The `std::vector` class is used here as the container for the measurement vectors from a sample.

```
#include "itkVector.h"
#include "itkListSample.h"
#include "itkKdTree.h"
#include "itkKdTreeGenerator.h"
#include "itkWeightedCentroidKdTreeGenerator.h"
#include "itkEuclideanDistance.h"
```

We define the measurement vector type and instantiate a `itk::Statistics::ListSample` object, and then put 1000 measurement vectors in the object.

```
  typedef itk::Vector< float, 2 > MeasurementVectorType;

  typedef itk::Statistics::ListSample< MeasurementVectorType > SampleType;
  SampleType::Pointer sample = SampleType::New();
  sample->SetMeasurementVectorSize( 2 );

  MeasurementVectorType mv;
  for (unsigned int i = 0 ; i < 1000 ; ++i )
    {
    mv[0] = (float) i;
    mv[1] = (float) ((1000 - i) / 2 );
    sample->PushBack( mv );
    }
```

The following code snippet shows how to create two KdTree objects. The first object `itk::Statistics::KdTreeGenerator` has a minimal set of information (partition dimension, partition value, and pointers to the left and right child nodes). The second tree from the `itk::Statistics::WeightedCentroidKdTreeGenerator` has additional information such as the number of children under each node, and the vector sum of the measurement vectors belonging to children of a particular node. WeightedCentroidKdTreeGenerator and the resulting k-d tree structure were implemented based on the description given in the paper by Kanungo et al [43].

The instantiation and input variables are exactly the same for both tree generators. Using the `SetSample()` method we plug-in the source sample. The bucket size input specifies the limit on the maximum number of measurement vectors that can be stored in a terminal (leaf) node. A bigger bucket size results in a smaller number of nodes in a tree. It also affects the efficiency of search. With many small leaf nodes, we might experience slower search performance because of excessive boundary comparisons.

```
typedef itk::Statistics::KdTreeGenerator< SampleType > TreeGeneratorType;
TreeGeneratorType::Pointer treeGenerator = TreeGeneratorType::New();

treeGenerator->SetSample( sample );
treeGenerator->SetBucketSize( 16 );
treeGenerator->Update();

typedef itk::Statistics::WeightedCentroidKdTreeGenerator< SampleType >
  CentroidTreeGeneratorType;

CentroidTreeGeneratorType::Pointer centroidTreeGenerator =
                                       CentroidTreeGeneratorType::New();

centroidTreeGenerator->SetSample( sample );
centroidTreeGenerator->SetBucketSize( 16 );
centroidTreeGenerator->Update();
```

After the generation step, we can get the pointer to the kd-tree from the generator by calling the `GetOutput()` method. To traverse a kd-tree, we have to use the `GetRoot()` method. The method will return the root node of the tree. Every node in a tree can have its left and/or right child node. To get the child node, we call the `Left()` or the `Right()` method of a node (these methods do not belong to the kd-tree but to the nodes).

We can get other information about a node by calling the methods described below in addition to the child node pointers.

```
typedef TreeGeneratorType::KdTreeType TreeType;
typedef TreeType::NearestNeighbors NeighborsType;
typedef TreeType::KdTreeNodeType NodeType;

TreeType::Pointer tree = treeGenerator->GetOutput();
TreeType::Pointer centroidTree = centroidTreeGenerator->GetOutput();

NodeType* root = tree->GetRoot();

if ( root->IsTerminal() )
  {
  std::cout << "Root node is a terminal node." << std::endl;
  }
else
  {
  std::cout << "Root node is not a terminal node." << std::endl;
  }
```

```
unsigned int partitionDimension;
float partitionValue;
root->GetParameters( partitionDimension, partitionValue);
std::cout << "Dimension chosen to split the space = "
          << partitionDimension << std::endl;
std::cout << "Split point on the partition dimension = "
          << partitionValue << std::endl;

std::cout << "Address of the left chile of the root node = "
          << root->Left() << std::endl;

std::cout << "Address of the right chile of the root node = "
          << root->Right() << std::endl;

root = centroidTree->GetRoot();
std::cout << "Number of the measurement vectors under the root node"
          << " in the tree hierarchy = " << root->Size() << std::endl;

NodeType::CentroidType centroid;
root->GetWeightedCentroid( centroid );
std::cout << "Sum of the measurement vectors under the root node = "
          << centroid << std::endl;

std::cout << "Number of the measurement vectors under the left child"
          << " of the root node = " << root->Left()->Size() << std::endl;
```

In the following code snippet, we query the three nearest neighbors of the `queryPoint` on the two tree. The results and procedures are exactly the same for both. First we define the point from which distances will be measured.

```
MeasurementVectorType queryPoint;
queryPoint[0] = 10.0;
queryPoint[1] = 7.0;
```

Then we instantiate the type of a distance metric, create an object of this type and set the origin of coordinates for measuring distances. The `GetMeasurementVectorSize()` method returns the length of each measurement vector stored in the sample.

```
typedef itk::Statistics::EuclideanDistance< MeasurementVectorType >
  DistanceMetricType;
DistanceMetricType::Pointer distanceMetric = DistanceMetricType::New();

DistanceMetricType::OriginType origin( 2 );
for ( unsigned int i = 0 ; i < sample->GetMeasurementVectorSize() ; ++i )
  {
  origin[i] = queryPoint[i];
```

```
    }
  distanceMetric->SetOrigin( origin );
```

We can now set the number of neighbors to be located and the point coordinates to be used as a reference system.

```
  unsigned int numberOfNeighbors = 3;
  TreeType::InstanceIdentifierVectorType neighbors;
  tree->Search( queryPoint, numberOfNeighbors, neighbors ) ;

  std::cout << "kd-tree knn search result:" << std::endl
            << "query point = [" << queryPoint << "]" << std::endl
            << "k = " << numberOfNeighbors << std::endl;
  std::cout << "measurement vector : distance" << std::endl;
  for ( unsigned int i = 0 ; i < numberOfNeighbors ; ++i )
    {
    std::cout << "[" << tree->GetMeasurementVector( neighbors[i] )
              << "] : "
              << distanceMetric->Evaluate(
                  tree->GetMeasurementVector( neighbors[i] ))
              << std::endl;
    }
```

As previously indicated, the interface for finding nearest neighbors in the centroid tree is very similar.

```
  centroidTree->Search( queryPoint, numberOfNeighbors, neighbors ) ;
  std::cout << "weighted centroid kd-tree knn search result:" << std::endl
            << "query point = [" << queryPoint << "]" << std::endl
            << "k = " << numberOfNeighbors << std::endl;
  std::cout << "measurement vector : distance" << std::endl;
  for ( unsigned int i = 0 ; i < numberOfNeighbors ; ++i )
    {
    std::cout << "[" << centroidTree->GetMeasurementVector( neighbors[i] )
              << "] : "
              << distanceMetric->Evaluate(
                  centroidTree->GetMeasurementVector( neighbors[i]))
              << std::endl;
    }
```

KdTree also supports searching points within a hyper-spherical kernel. We specify the radius and call the `Search()` method. In the case of the KdTree, this is done with the following lines of code.

```
  double radius = 437.0;
  tree->Search( queryPoint, radius, neighbors ) ;

  std::cout << "kd-tree radius search result:" << std::endl
            << "query point = [" << queryPoint << "]" << std::endl
            << "search radius = " << radius << std::endl;
  std::cout << "measurement vector : distance" << std::endl;
  for ( unsigned int i = 0 ; i < neighbors.size() ; ++i )
    {
    std::cout << "[" << tree->GetMeasurementVector( neighbors[i] )
              << "] : "
              << distanceMetric->Evaluate(
                  tree->GetMeasurementVector( neighbors[i]))
              << std::endl;
    }
```

In the case of the centroid KdTree, the `Search()` method is used as illustrated by the following code.

```
  centroidTree->Search( queryPoint, radius, neighbors ) ;
  std::cout << "weighted centroid kd-tree radius search result:" << std::endl
            << "query point = [" << queryPoint << "]" << std::endl
            << "search radius = " << radius << std::endl;
  std::cout << "measurement vector : distance" << std::endl;
  for ( unsigned int i = 0 ; i < neighbors.size() ; ++i )
    {
    std::cout << "[" << centroidTree->GetMeasurementVector( neighbors[i] )
              << "] : "
              << distanceMetric->Evaluate(
                  centroidTree->GetMeasurementVector( neighbors[i]))
              << std::endl;
    }
```

10.2 Algorithms and Functions

In the previous section, we described the data containers in the ITK statistics subsystem. We also need data processing algorithms and statistical functions to conduct statistical analysis or statistical classification using these containers. Here we define an algorithm to be an operation over a set of measurement vectors in a sample. A function is an operation over individual measurement vectors. For example, if we implement a class (`itk::Statistics::EuclideanDistance`) to calculate the Euclidean distance between two measurement vectors, we call it a function, while if we implemented a class (`itk::Statistics::MeanCalculator`) to calculate the mean of a sample, we call it an algorithm.

10.2.1 Sample Statistics

We will show how to get sample statistics such as means and covariance from the (itk::Statistics::Sample) classes. Statistics can tells us characteristics of a sample. Such sample statistics are very important for statistical classification. When we know the form of the sample distributions and their parameters (statistics), we can conduct Bayesian classification. In ITK, sample mean and covariance calculation algorithms are implemented. Each algorithm also has its weighted version (see Section 10.2.1). The weighted versions are used in the expectation-maximization parameter estimation process.

Mean and Covariance

The source code for this section can be found in the file
`Examples/Statistics/SampleStatistics.cxx`.

We include the header file for the itk::Vector class that will be our measurement vector template in this example.

```
#include "itkVector.h"
```

We will use the itk::Statistics::ListSample as our sample template. We include the header for the class too.

```
#include "itkListSample.h"
```

The following headers are for sample statistics algorithms.

```
#include "itkMeanCalculator.h"
#include "itkCovarianceCalculator.h"
```

The following code snippet will create a ListSample object with three-component float measurement vectors and put five measurement vectors in the ListSample object.

```
  const unsigned int MeasurementVectorLength = 3;
  typedef itk::Vector< float, MeasurementVectorLength > MeasurementVectorType;
  typedef itk::Statistics::ListSample< MeasurementVectorType > SampleType;
  SampleType::Pointer sample = SampleType::New();
  sample->SetMeasurementVectorSize( MeasurementVectorLength );
  MeasurementVectorType mv;
  mv[0] = 1.0;
  mv[1] = 2.0;
  mv[2] = 4.0;

  sample->PushBack( mv );
```

```
mv[0] = 2.0;
mv[1] = 4.0;
mv[2] = 5.0;
sample->PushBack( mv );

mv[0] = 3.0;
mv[1] = 8.0;
mv[2] = 6.0;
sample->PushBack( mv );

mv[0] = 2.0;
mv[1] = 7.0;
mv[2] = 4.0;
sample->PushBack( mv );

mv[0] = 3.0;
mv[1] = 2.0;
mv[2] = 7.0;
sample->PushBack( mv );
```

To calculate the mean (vector) of a sample, we instantiate the `itk::Statistics::MeanCalculator` class that implements the mean algorithm and plug in the sample using the `SetInputSample(sample*)` method. By calling the `Update()` method, we run the algorithm. We get the mean vector using the `GetOutput()` method. The output from the `GetOutput()` method is the pointer to the mean vector.

```
typedef itk::Statistics::MeanCalculator< SampleType > MeanAlgorithmType;

MeanAlgorithmType::Pointer meanAlgorithm = MeanAlgorithmType::New();

meanAlgorithm->SetInputSample( sample );
meanAlgorithm->Update();

std::cout << "Sample mean = " << *(meanAlgorithm->GetOutput()) << std::endl;
```

To use the covariance calculation algorithm, we have two options. Since we already have the mean calculated by the MeanCalculator, we can plug-in its output to an instance of `itk::Statistics::CovarianceCalculator` using the `SetMean()` method. The other option is not to set the mean at all and just call the `Update()` method. The covariance calculation algorithm will compute the mean and covariance together in one pass. If you have already set the mean as in this example and you want to run one pass algorithm, simply pass a null pointer as the mean vector.

```
typedef itk::Statistics::CovarianceCalculator< SampleType >
  CovarianceAlgorithmType;
CovarianceAlgorithmType::Pointer covarianceAlgorithm =
  CovarianceAlgorithmType::New();

covarianceAlgorithm->SetInputSample( sample );
covarianceAlgorithm->SetMean( meanAlgorithm->GetOutput() );
covarianceAlgorithm->Update();

std::cout << "Sample covariance = " << std::endl ;
std::cout << *(covarianceAlgorithm->GetOutput()) << std::endl;

covarianceAlgorithm->SetMean( 0 );
covarianceAlgorithm->Update();

std::cout << "Using the one pass algorithm:" << std::endl;
std::cout << "Mean = " << std::endl ;
std::cout << *(covarianceAlgorithm->GetMean()) << std::endl;

std::cout << "Covariance = " << std::endl ;
std::cout << *(covarianceAlgorithm->GetOutput()) << std::endl;
```

Weighted Mean and Covariance

The source code for this section can be found in the file
`Examples/Statistics/WeightedSampleStatistics.cxx`.

We include the header file for the `itk::Vector` class that will be our measurement vector template in this example.

```
#include "itkVector.h"
```

We will use the `itk::Statistics::ListSample` as our sample template. We include the header for the class too.

```
#include "itkListSample.h"
```

The following headers are for the weighted covariance algorithms.

```
#include "itkWeightedMeanCalculator.h"
#include "itkWeightedCovarianceCalculator.h"
```

The following code snippet will create a ListSample instance with three-component float measurement vectors and put five measurement vectors in the ListSample object.

```
  typedef itk::Statistics::ListSample< MeasurementVectorType > SampleType;
  SampleType::Pointer sample = SampleType::New();
```

```
sample->SetMeasurementVectorSize( 3 );
MeasurementVectorType mv;
mv[0] = 1.0;
mv[1] = 2.0;
mv[2] = 4.0;

sample->PushBack( mv );

mv[0] = 2.0;
mv[1] = 4.0;
mv[2] = 5.0;
sample->PushBack( mv );

mv[0] = 3.0;
mv[1] = 8.0;
mv[2] = 6.0;
sample->PushBack( mv );

mv[0] = 2.0;
mv[1] = 7.0;
mv[2] = 4.0;
sample->PushBack( mv );

mv[0] = 3.0;
mv[1] = 2.0;
mv[2] = 7.0;
sample->PushBack( mv );
```

Robust versions of covariance algorithms require weight values for measurement vectors. We have two ways of providing weight values for the weighted mean and weighted covariance algorithms.

The first method is to plug in an array of weight values. The size of the weight value array should be equal to that of the measurement vectors. In both algorithms, we use the `SetWeights(weights*)`.

```
typedef itk::Statistics::WeightedMeanCalculator< SampleType >
  WeightedMeanAlgorithmType;

WeightedMeanAlgorithmType::WeightArrayType weightArray( sample->Size() );
weightArray.Fill( 0.5 );
weightArray[2] = 0.01;
weightArray[4] = 0.01;

WeightedMeanAlgorithmType::Pointer weightedMeanAlgorithm =
                                            WeightedMeanAlgorithmType::New();

weightedMeanAlgorithm->SetInputSample( sample );
```

```
  weightedMeanAlgorithm->SetWeights( &weightArray );
  weightedMeanAlgorithm->Update();

  std::cout << "Sample weighted mean = "
            << *(weightedMeanAlgorithm->GetOutput()) << std::endl;

  typedef itk::Statistics::WeightedCovarianceCalculator< SampleType >
                                            WeightedCovarianceAlgorithmType;

  WeightedCovarianceAlgorithmType::Pointer weightedCovarianceAlgorithm =
                                      WeightedCovarianceAlgorithmType::New();

  weightedCovarianceAlgorithm->SetInputSample( sample );
  weightedCovarianceAlgorithm->SetMean( weightedMeanAlgorithm->GetOutput() );
  weightedCovarianceAlgorithm->SetWeights( &weightArray );
  weightedCovarianceAlgorithm->Update();

  std::cout << "Sample weighted covariance = " << std::endl ;
  std::cout << *(weightedCovarianceAlgorithm->GetOutput()) << std::endl;
```

The second method for computing weighted statistics is to plug-in a function that returns a weight value that is usually a function of each measurement vector. Since the `weightedMeanAlgorithm` and `weightedCovarianceAlgorithm` already have the input sample plugged in, we only need to call the `SetWeightFunction(weights*)` method. For the `weightedCovarianceAlgorithm`, we replace the mean vector input with the output from the `weightedMeanAlgorithm`. If we do not provide the mean vector using the `SetMean()` method or if we pass a null pointer as the mean vector as in this example, the `weightedCovarianceAlgorithm` will perform the one pass algorithm to generate the mean vector and the covariance matrix.

```
  ExampleWeightFunction::Pointer weightFunction = ExampleWeightFunction::New();

  weightedMeanAlgorithm->SetWeightFunction( weightFunction );
  weightedMeanAlgorithm->Update();

  std::cout << "Sample weighted mean = "
            << *(weightedMeanAlgorithm->GetOutput()) << std::endl;

  weightedCovarianceAlgorithm->SetMean( weightedMeanAlgorithm->GetOutput() );
  weightedCovarianceAlgorithm->SetWeightFunction( weightFunction );
  weightedCovarianceAlgorithm->Update();

  std::cout << "Sample weighted covariance = " << std::endl ;
  std::cout << *(weightedCovarianceAlgorithm->GetOutput()) << std::endl;

  weightedCovarianceAlgorithm->SetMean( 0 );
  weightedCovarianceAlgorithm->SetWeightFunction( weightFunction );
  weightedCovarianceAlgorithm->Update();

  std::cout << "Using the one pass algorithm:" << std::endl;
  std::cout << "Sample weighted covariance = " << std::endl ;
  std::cout << *(weightedCovarianceAlgorithm->GetOutput()) << std::endl;

  std::cout << "Sample weighted mean = "
            << *(weightedCovarianceAlgorithm->GetMean()) << std::endl;
```

10.2.2 Sample Generation

ListSampleToHistogramFilter

The source code for this section can be found in the file
`Examples/Statistics/ListSampleToHistogramFilter.cxx`.

Sometimes we want to work with a histogram instead of a list of measurement vectors (e.g. `itk::Statistics::ListSample`, `itk::Statistics::ImageToListAdaptor`, or `itk::Statistics::PointSetToListSample`) to use less memory or to perform a particular type od analysis. In such cases, we can import data from a list type sample to a `itk::Statistics::Histogram` object using the `itk::Statistics::ListSampleToHistogramFilter`.

We use a ListSample object as the input for the filter. We include the header files for the ListSample and Histogram classes, as well as the filter.

```
#include "itkListSample.h"
#include "itkHistogram.h"
#include "itkListSampleToHistogramFilter.h"
```

We need another header for the type of the measurement vectors. We are going to use the `itk::Vector` class which is a subclass of the `itk::FixedArray` in this example.

```
#include "itkVector.h"
```

The following code snippet creates a ListSample object with two-component `int` measurement vectors and put the measurement vectors: [1,1] - 1 time, [2,2] - 2 times, [3,3] - 3 times, [4,4] - 4 times, [5,5] - 5 times into the `listSample`.

```
  typedef int MeasurementType;
  const unsigned int MeasurementVectorLength = 2;
  typedef itk::Vector< MeasurementType , MeasurementVectorLength >
                                                         MeasurementVectorType;
  typedef itk::Statistics::ListSample< MeasurementVectorType > ListSampleType;
  ListSampleType::Pointer listSample = ListSampleType::New();
```

```
listSample->SetMeasurementVectorSize( MeasurementVectorLength );

MeasurementVectorType mv;
for ( unsigned int i = 1 ; i < 6 ; i++ )
  {
  for ( unsigned int j = 0 ; j < 2 ; j++ )
    {
    mv[j] = ( MeasurementType ) i;
    }
  for ( unsigned int j = 0 ; j < i ; j++ )
    {
    listSample->PushBack(mv);
    }
  }
```

Here, we create a Histogram object with equal interval bins using the `Initalize()` method.

```
typedef float HistogramMeasurementType;
typedef itk::Statistics::Histogram< HistogramMeasurementType, 2 >
  HistogramType;
HistogramType::Pointer histogram = HistogramType::New();

HistogramType::SizeType size;
size.Fill(5);

HistogramType::MeasurementVectorType lowerBound;
HistogramType::MeasurementVectorType upperBound;

lowerBound[0] = 0.5;
lowerBound[1] = 0.5;

upperBound[0] = 5.5;
upperBound[1] = 5.5;

histogram->Initialize( size, lowerBound, upperBound );
```

The `Size()` and `GetTotalFrequency()` methods return the same values as the `sample` does.

```
typedef itk::Statistics::ListSampleToHistogramFilter< ListSampleType,
                         HistogramType > FilterType;
FilterType::Pointer filter = FilterType::New();

filter->SetListSample( listSample );
filter->SetHistogram( histogram );
filter->Update();

HistogramType::Iterator iter = histogram->Begin();
```

```
  while ( iter != histogram->End() )
    {
    std::cout << "Measurement vectors = " << iter.GetMeasurementVector()
              << " frequency = " << iter.GetFrequency() << std::endl;
    ++iter;
    }

  std::cout << "Size = " << histogram->Size() << std::endl;
  std::cout << "Total frequency = "
            << histogram->GetTotalFrequency() << std::endl;
```

ListSampleToHistogramGenerator

The source code for this section can be found in the file
`Examples/Statistics/ListSampleToHistogramGenerator.cxx`.

In previous sections (Section 10.2.2 we described how to import data from a `itk::Statistics::ListSample` to a `itk::Statistics::Histogram`. An alternative way of creating a histogram is to use `itk::Statistics::ListSampleToHistogramGenerator`. With this generator, we only provide the size of the histogram and the type of the measurement vectors in the histogram. The generator will automatically find the lower and upper space bound and create equal interval bins in the histogram.

We use a ListSample object as the input for the filter. We include the header files for the ListSample, Histogram, and the filter itself.

```
#include "itkListSample.h"
#include "itkHistogram.h"
#include "itkListSampleToHistogramGenerator.h"
```

We need another header for measurement vectors. We are going to use the `itk::Vector` class which is a subclass of the `itk::FixedArray` in this example.

```
#include "itkVector.h"
```

The following code snippet will create a ListSample object with two-component int measurement vectors and put the measurement vectors: [1,1] - 1 time, [2,2] - 2 times, [3,3] - 3 times, [4,4] - 4 times, [5,5] - 5 times into the ListSample.

```
  typedef int MeasurementType;
  const unsigned int MeasurementVectorLength = 2;
  typedef itk::Vector< MeasurementType , MeasurementVectorLength >
                                                             MeasurementVectorType;
  typedef itk::Statistics::ListSample< MeasurementVectorType > ListSampleType;
  ListSampleType::Pointer listSample = ListSampleType::New();
```

```
listSample->SetMeasurementVectorSize( MeasurementVectorLength );

MeasurementVectorType mv;
for ( unsigned int i = 1 ; i < 6 ; i++ )
  {
  for ( unsigned int j = 0 ; j < 2 ; j++ )
    {
    mv[j] = ( MeasurementType ) i;
    }
  for ( unsigned int j = 0 ; j < i ; j++ )
    {
    listSample->PushBack(mv);
    }
  }
```

The ListSampleToHistogramGenerator will find the lower and upper bound from the input sample and create equal interval bins. Since a Histogram object does not include the upper bound value and we want to include [5,5] measurement vector, we increase the upper-bound by the calculated bin interval/10.0 (divider). The divider is set by the `SetMarginalScale(float)` method. If you want to create a non-uniform histogram, you should use the ListSampleToHistogramFilter (see Section 10.2.2). The filter does not create a Histogram object. Instead, users should create a Histogram object with varying intervals and use the filter to fill the Histogram objects with the measurement vectors from a ListSample object.

```
typedef float HistogramMeasurementType;
typedef itk::Statistics::ListSampleToHistogramGenerator< ListSampleType,
        HistogramMeasurementType,
        itk::Statistics::DenseFrequencyContainer,
        MeasurementVectorLength >             GeneratorType;
GeneratorType::Pointer generator = GeneratorType::New();

GeneratorType::HistogramType::SizeType size;
size.Fill(5);

generator->SetListSample( listSample );
generator->SetNumberOfBins( size );
generator->SetMarginalScale( 10.0 );
generator->Update();
```

The following code prints out the content of the resulting histogram.

```
GeneratorType::HistogramType::ConstPointer histogram = generator->GetOutput();
GeneratorType::HistogramType::ConstIterator iter = histogram->Begin();
while ( iter != histogram->End() )
  {
  std::cout << "Measurement vectors = " << iter.GetMeasurementVector()
```

```
              << " frequency = " << iter.GetFrequency() << std::endl;
    ++iter;
    }
  std::cout << "Size = " << histogram->Size() << std::endl;
  std::cout << "Total frequency = "
            << histogram->GetTotalFrequency() << std::endl;
```

NeighborhoodSampler

The source code for this section can be found in the file
`Examples/Statistics/NeighborhoodSampler.cxx`.

When we want to create an `itk::Statistics::Subsample` object that includes only the measurement vectors within a radius from a center in a sample, we can use the `itk::Statistics::NeighborhoodSampler`. In this example, we will use the `itk::Statistics::ListSample` as the input sample.

We include the header files for the ListSample and the NeighborhoodSampler classes.

```
#include "itkListSample.h"
#include "itkNeighborhoodSampler.h"
```

We need another header for measurement vectors. We are going to use the `itk::Vector` class which is a subclass of the `itk::FixedArray`.

```
#include "itkVector.h"
```

The following code snippet will create a ListSample object with two-component int measurement vectors and put the measurement vectors: [1,1] - 1 time, [2,2] - 2 times, [3,3] - 3 times, [4,4] - 4 times, [5,5] - 5 times into the `listSample`.

```
  typedef int MeasurementType;
  const unsigned int MeasurementVectorLength = 2;
  typedef itk::Vector< MeasurementType , MeasurementVectorLength >
                                                  MeasurementVectorType;
  typedef itk::Statistics::ListSample< MeasurementVectorType > SampleType;
  SampleType::Pointer sample = SampleType::New();
  sample->SetMeasurementVectorSize( MeasurementVectorLength );

  MeasurementVectorType mv;
  for ( unsigned int i = 1 ; i < 6 ; i++ )
    {
    for ( unsigned int j = 0 ; j < 2 ; j++ )
      {
      mv[j] = ( MeasurementType ) i;
      }
    for ( unsigned int j = 0 ; j < i ; j++ )
      {
      sample->PushBack(mv);
      }
    }
```

We plug-in the sample to the NeighborhoodSampler using the `SetInputSample(sample*)`. The two required inputs for the NeighborhoodSampler are a center and a radius. We set these two inputs using the `SetCenter(center vector*)` and the `SetRadius(double*)` methods respectively. And then we call the `Update()` method to generate the Subsample object. This sampling procedure subsamples measurement vectors within a hyper-spherical kernel that has the center and radius specified.

```
  typedef itk::Statistics::NeighborhoodSampler< SampleType > SamplerType;
  SamplerType::Pointer sampler = SamplerType::New();

  sampler->SetInputSample( sample );
  SamplerType::CenterType center( MeasurementVectorLength );
  center[0] = 3;
  center[1] = 3;
  double radius = 1.5;
  sampler->SetCenter( &center );
  sampler->SetRadius( &radius );
  sampler->Update();

  SamplerType::OutputType::Pointer output = sampler->GetOutput();
```

The `SamplerType::OutputType` is in fact `itk::Statistics::Subsample`. The following code prints out the resampled measurement vectors.

```
  SamplerType::OutputType::Iterator iter = output->Begin();
  while ( iter != output->End() )
    {
    std::cout << "instance identifier = " << iter.GetInstanceIdentifier()
              << "\t measurement vector = "
              << iter.GetMeasurementVector()
              << "\t frequency = "
              << iter.GetFrequency() << std::endl;
    ++iter;
    }
```

SampleToHistogramProjectionFilter

The source code for this section can be found in the file
`Examples/Statistics/SampleToHistogramProjectionFilter.cxx`.

The itk::Statistics::SampleToHistogramProjectionFilter projects measurement vectors of a sample onto a vector and fills up a 1-D itk::Statistics::Histogram. The histogram will be formed around the mean value set by the `SetMean()` method. The histogram's measurement values are the distance between the mean and the projected measurement vectors normalized by the standard deviation set by the `SetStandardDeviation()` method. Such histogram can be used to analyze the multi-dimensional distribution or examine the *goodness-of-fit* of a projected distribution (histogram) with its expected distribution.

We will use the ListSample as the input sample.

```
#include "itkListSample.h"
#include "itkSampleToHistogramProjectionFilter.h"
```

We need another header for measurement vectors. We are going to use the itk::Vector class which is a subclass of the itk::FixedArray.

```
#include "itkVector.h"
```

The following code snippet will create a ListSample object with two-component int measurement vectors and put the measurement vectors: [1,1] - 1 time, [2,2] - 2 times, [3,3] - 3 times, [4,4] - 4 times, [5,5] - 5 times into the `listSample`.

```
  const unsigned int MeasurementVectorLength = 2;
  typedef int MeasurementType;
  typedef itk::Vector< MeasurementType , MeasurementVectorLength > MeasurementVectorType;
  typedef itk::Statistics::ListSample< MeasurementVectorType > SampleType;
  SampleType::Pointer sample = SampleType::New();
  sample->SetMeasurementVectorSize( MeasurementVectorLength );

  MeasurementVectorType mv;
  for ( unsigned int i = 1 ; i < 6 ; i++ )
    {
    for ( unsigned int j = 0 ; j < 2 ; j++ )
      {
      mv[j] = ( MeasurementType ) i;
      }
    for ( unsigned int j = 0 ; j < i ; j++ )
      {
      sample->PushBack(mv);
      }
    }
```

We create a histogram that has six bins. The histogram's range is [-2, 2). Since the `sample` has measurement vectors between [1, 1] and [5,5], The histogram does not seem to cover the whole range. However, the SampleToHistogramProjectionFilter normalizes the measurement vectors with the given mean and the standard deviation. Therefore, the projected value is approximately the distance between the measurement vector and the mean divided by the standard deviation.

```
typedef itk::Statistics::Histogram< float, 1 > HistogramType;
HistogramType::Pointer histogram = HistogramType::New();

HistogramType::SizeType size;
size.Fill(6);
HistogramType::MeasurementVectorType lowerBound;
HistogramType::MeasurementVectorType upperBound;
lowerBound[0] = -2;
upperBound[0] = 2;

histogram->Initialize( size, lowerBound, upperBound );
```

We use the `SetInputSample(sample*)` and the `SetHistogram(histogram*)` methods to set the input sample and the output histogram that have been created.

```
typedef itk::Statistics::SampleToHistogramProjectionFilter<SampleType, float>
  ProjectorType;
ProjectorType::Pointer projector = ProjectorType::New();

projector->SetInputSample( sample );
projector->SetHistogram( histogram );
```

As mentioned above, this class projects measurement vectors onto the projection axis with normalization using the mean and standard deviation.

$$y = \frac{\sum_{i=0}^{d}(x_i - \mu_i)\alpha_i}{\sigma} \tag{10.1}$$

where, y is the projected value, x is the ith component of the measurement vector, μ_i is the ith component of the mean vector, α_i is the ith component of the projection axis (a vector), and σ is the standard deviation.

If the bin overlap value is set by the `SetHistogramBinOverlap()` method and it is greater than 0.001, the frequency will be weighted based on its closeness of the bin boundaries. In other words, even if a measurement vector falls into a bin, depending on its closeness to the adjacent bins, the frequencies of the adjacent bins will be also updated with weights. If we do not want to use the bin overlapping function, we do not call the `SetHistogramBinOverlap(double)` method. The default value for the histogram bin overlap is zero, so without calling the method, the filter will not use bin overlapping [7] [8].

```
ProjectorType::MeanType mean( MeasurementVectorLength );
mean[0] = 3.66667;
mean[1] = 3.66667;

double standardDeviation = 3;

ProjectorType::ArrayType projectionAxis( MeasurementVectorLength );
```

```
projectionAxis[0] = 1;
projectionAxis[1] = 1;

projector->SetMean( &mean );
projector->SetStandardDeviation( &standardDeviation );
projector->SetProjectionAxis( &projectionAxis );
projector->SetHistogramBinOverlap( 0.25 );
projector->Update();
```

We print out the updated histogram after the projection.

```
float fSum = 0.0;
HistogramType::Iterator iter = histogram->Begin();
while ( iter != histogram->End() )
  {
  std::cout << "instance identifier = " << iter.GetInstanceIdentifier()
            << "\t measurement vector = "
            << iter.GetMeasurementVector()
            << "\t frequency = "
            << iter.GetFrequency() << std::endl;
  fSum += iter.GetFrequency();
  ++iter;
  }
std::cout << " sum of frequency = " << fSum << std::endl;
```

10.2.3 Sample Sorting

The source code for this section can be found in the file
`Examples/Statistics/SampleSorting.cxx`.

Sometimes we want to sort the measurement vectors in a sample. The sorted vectors may reveal some characteristics of the sample. The *insert sort*, the *heap sort*, and the *introspective sort* algorithms [59] for samples are implemented in ITK. To learn pros and cons of each algorithm, please refer to [24]. ITK also offers the *quick select* algorithm.

Among the subclasses of the `itk::Statistics::Sample`, only the class `itk::Statistics::Subsample` allows users to change the order of the measurement vector. Therefore, we must create a Subsample to do any sorting or selecting.

We include the header files for the `itk::Statistics::ListSample` and the `Subsample` classes.

```
#include "itkListSample.h"
#include "itkSubsample.h"
```

The sorting and selecting related functions are in the include file `itkStatisticsAlgorithm.h`.

```
#include "itkStatisticsAlgorithm.h"
```

We need another header for measurement vectors. We are going to use the `itk::Vector` class which is a subclass of the `itk::FixedArray` in this example.

We define the types of the measurement vectors, the sample, and the subsample.

```
#include "itkVector.h"
```

We define two functions for convenience. The first one clears the content of the subsample and fill it with the measurement vectors from the sample.

```
void initializeSubsample(SubsampleType* subsample, SampleType* sample)
{
  subsample->Clear();
  subsample->SetSample(sample);
  subsample->InitializeWithAllInstances();
}
```

The second one prints out the content of the subsample using the Subsample's iterator interface.

```
void printSubsample(SubsampleType* subsample, const char* header)
{
  std::cout << std::endl;
  std::cout << header << std::endl;
  SubsampleType::Iterator iter = subsample->Begin();
  while ( iter != subsample->End() )
    {
    std::cout << "instance identifier = " << iter.GetInstanceIdentifier()
              << "\t measurement vector = "
              << iter.GetMeasurementVector()
              << std::endl;
    ++iter;
    }
}
```

The following code snippet will create a ListSample object with two-component int measurement vectors and put the measurement vectors: [5,5] - 5 times, [4,4] - 4 times, [3,3] - 3 times, [2,2] - 2 times,[1,1] - 1 time into the `sample`.

```
  SampleType::Pointer sample = SampleType::New();

  MeasurementVectorType mv;
  for ( unsigned int i = 5 ; i > 0 ; --i )
    {
```

```
    for (unsigned int j = 0 ; j < 2 ; j++ )
      {
      mv[j] = ( MeasurementType ) i;
      }
    for ( unsigned int j = 0 ; j < i ; j++ )
      {
      sample->PushBack(mv);
      }
    }
```

We create a Subsample object and plug-in the `sample`.

```
  SubsampleType::Pointer subsample = SubsampleType::New();
  subsample->SetSample(sample);
  initializeSubsample(subsample, sample);
  printSubsample(subsample, "Unsorted");
```

The common parameters to all the algorithms are the Subsample object (`subsample`), the dimension (`activeDimension`) that will be considered for the sorting or selecting (only the component belonging to the dimension of the measurement vectors will be considered), the beginning index, and the ending index of the measurement vectors in the `subsample`. The sorting or selecting algorithms are applied only to the range specified by the beginning index and the ending index. The ending index should be the actual last index plus one.

The `itk::InsertSort` function does not require any other optional arguments. The following function call will sort the all measurement vectors in the `subsample`. The beginning index is `0`, and the ending index is the number of the measurement vectors in the `subsample`.

```
  int activeDimension = 0 ;
  itk::Statistics::InsertSort< SubsampleType >( subsample, activeDimension,
                                                0, subsample->Size() );
  printSubsample(subsample, "InsertSort");
```

We sort the `subsample` using the heap sort algorithm. The arguments are identical to those of the insert sort.

```
  initializeSubsample(subsample, sample);
  itk::Statistics::HeapSort< SubsampleType >( subsample, activeDimension,
                                              0, subsample->Size() );
  printSubsample(subsample, "HeapSort");
```

The introspective sort algorithm needs an additional argument that specifies when to stop the introspective sort loop and sort the fragment of the sample using the heap sort algorithm. Since we set the threshold value as `16`, when the sort loop reach the point where the number of measurement vectors in a sort loop is not greater than `16`, it will sort that fragment using the insert sort algorithm.

```
  initializeSubsample(subsample, sample);
  itk::Statistics::IntrospectiveSort< SubsampleType >
                    ( subsample, activeDimension, 0, subsample->Size(), 16 );
  printSubsample(subsample, "IntrospectiveSort");
```

We query the median of the measurements along the `activeDimension`. The last argument tells the algorithm that we want to get the `subsample->Size()/2`-th element along the `activeDimension`. The quick select algorithm changes the order of the measurement vectors.

```
  initializeSubsample(subsample, sample);
  SubsampleType::MeasurementType median =
          itk::Statistics::QuickSelect< SubsampleType >( subsample,
                                                         activeDimension,
                                                         0, subsample->Size(),
                                                         subsample->Size()/2 );
  std::cout << std::endl;
  std::cout << "Quick Select: median = " << median << std::endl;
```

10.2.4 Probability Density Functions

The probability density function (PDF) for a specific distribution returns the probability density for a measurement vector. To get the probability density from a PDF, we use the `Evaluate(input)` method. PDFs for different distributions require different sets of distribution parameters. Before calling the `Evaluate()` method, make sure to set the proper values for the distribution parameters.

Gaussian Distribution

The source code for this section can be found in the file
`Examples/Statistics/GaussianDensityFunction.cxx`.

The Gaussian probability density function `itk::Statistics::GaussianDensityFunction` requires two distribution parameters—the mean vector and the covariance matrix.

We include the header files for the class and the `itk::Vector`.

```
#include "itkVector.h"
#include "itkGaussianDensityFunction.h"
```

We define the type of the measurement vector that will be input to the Gaussian density function.

```
  typedef itk::Vector< float, 2 > MeasurementVectorType;
```

The instantiation of the function is done through the usual `New()` method and a smart pointer.

```
typedef itk::Statistics::GaussianDensityFunction< MeasurementVectorType >
  DensityFunctionType;
DensityFunctionType::Pointer densityFunction = DensityFunctionType::New();
```

The length of the measurement vectors in the density function, in this case a vector of length 2, is specified using the `SetMeasurementVectorSize()` method.

```
densityFunction->SetMeasurementVectorSize( 2 );
```

We create the two distribution parameters and set them. The mean is [0, 0], and the covariance matrix is a 2 x 2 matrix:

$$\begin{pmatrix} 4 & 0 \\ 0 & 4 \end{pmatrix}$$

We obtain the probability density for the measurement vector: [0, 0] using the `Evaluate(measurement vector)` method and print it out.

```
DensityFunctionType::MeanType mean( 2 );
mean.Fill( 0.0 );

DensityFunctionType::CovarianceType cov;
cov.SetSize( 2, 2 );
cov.SetIdentity();
cov *= 4;

densityFunction->SetMean( &mean );
densityFunction->SetCovariance( &cov );

MeasurementVectorType mv;
mv.Fill( 0 );

std::cout << densityFunction->Evaluate( mv ) << std::endl;
```

10.2.5 Distance Metric

Euclidean Distance

The source code for this section can be found in the file
`Examples/Statistics/EuclideanDistance.cxx`.

The Euclidean distance function (`itk::Statistics::EuclideanDistance` requires as template parameter the type of the measurement vector. We can use this function for any subclass of the `itk::FixedArray`. As a subclass of the `itk::Statistics::DistanceMetric`, it has two basic methods, the `SetOrigin(measurement vector)` and the `Evaluate(measurement vector)`. The `Evaluate()` method returns the distance between its argument (a measurement vector) and the measurement vector set by the `SetOrigin()` method.

In addition to the two methods, EuclideanDistance has two more methods that return the distance of two measurements — `Evaluate(measurement vector, measurement vector)` and the coordinate distance between two measurements (not vectors) — `Evaluate(measurement, measurement)`. The argument type of the latter method is the type of the component of the measurement vector.

We include the header files for the class and the `itk::Vector`.

```
#include "itkVector.h"
#include "itkArray.h"
#include "itkEuclideanDistance.h"
```

We define the type of the measurement vector that will be input of the Euclidean distance function. As a result, the measurement type is `float`.

```
  typedef itk::Array< float > MeasurementVectorType;
```

The instantiation of the function is done through the usual `New()` method and a smart pointer.

```
  typedef itk::Statistics::EuclideanDistance< MeasurementVectorType >
    DistanceMetricType;
  DistanceMetricType::Pointer distanceMetric = DistanceMetricType::New();
```

We create three measurement vectors, the `originPoint`, the `queryPointA`, and the `queryPointB`. The type of the `originPoint` is fixed in the `itk::Statistics::DistanceMetric` base class as `itk::Vector< double, length of the measurement vector of the each distance metric instance>`.

```
  // The Distance metric does not know about the length of the measurement vectors.
  // We must set it explicitly using the \code{SetMeasurementVectorSize()} method.

  DistanceMetricType::OriginType originPoint( 2 );
  MeasurementVectorType queryPointA( 2 );
  MeasurementVectorType queryPointB( 2 );

  originPoint[0] = 0;
  originPoint[1] = 0;

  queryPointA[0] = 2;
  queryPointA[1] = 2;

  queryPointB[0] = 3;
  queryPointB[1] = 3;
```

In the following code snippet, we show the uses of the three different `Evaluate()` methods.

```
  distanceMetric->SetOrigin( originPoint );
  std::cout << "Euclidean distance between the origin and the query point A = "
            << distanceMetric->Evaluate( queryPointA )
            << std::endl;

  std::cout << "Euclidean distance between the two query points (A and B) = "
            << distanceMetric->Evaluate( queryPointA, queryPointB )
            << std::endl;

  std::cout << "Coordinate distance between "
            << "the first components of the two query points = "
            << distanceMetric->Evaluate( queryPointA[0], queryPointB[0] )
            << std::endl;
```

10.2.6 Decision Rules

A decision rule is a function that returns the index of one data element in a vector of data elements. The index returned depends on the internal logic of each decision rule. The decision rule is an essential part of the ITK statistical classification framework. The scores from a set of membership functions (e.g. probability density functions, distance metrics) are compared by a decision rule and a class label is assigned based on the output of the decision rule. The common interface is very simple. Any decision rule class must implement the `Evaluate()` method. In addition to this method, certain decision rule class can have additional method that accepts prior knowledge about the decision task. The `itk::MaximumRatioDecisionRule` is an example of such a class.

The argument type for the `Evaluate()` method is `std::vector< double >`. The decision rule classes are part of the `itk` namespace instead of `itk::Statistics` namespace.

For a project that uses a decision rule, it must link the `itkCommon` library. Decision rules are not templated classes.

Maximum Decision Rule

The source code for this section can be found in the file
`Examples/Statistics/MaximumDecisionRule.cxx`.

The `itk::MaximumDecisionRule` returns the index of the largest discriminant score among the discriminant scores in the vector of discriminant scores that is the input argument of the `Evaluate()` method.

To begin the example, we include the header files for the class and the MaximumDecisionRule. We also include the header file for the `std::vector` class that will be the container for the discriminant scores.

```
#include "itkMaximumDecisionRule.h"
```

```
#include <vector>
```

The instantiation of the function is done through the usual `New()` method and a smart pointer.

```
  typedef itk::MaximumDecisionRule DecisionRuleType;
  DecisionRuleType::Pointer decisionRule = DecisionRuleType::New();
```

We create the discriminant score vector and fill it with three values. The `Evaluate( discriminantScores )` will return 2 because the third value is the largest value.

```
  std::vector< double > discriminantScores;
  discriminantScores.push_back( 0.1 );
  discriminantScores.push_back( 0.3 );
  discriminantScores.push_back( 0.6 );

  std::cout << "MaximumDecisionRule: The index of the chosen = "
            << decisionRule->Evaluate( discriminantScores )
            << std::endl;
```

Minimum Decision Rule

The source code for this section can be found in the file
`Examples/Statistics/MinimumDecisionRule.cxx`.

The `Evaluate()` method of the `itk::MinimumDecisionRule` returns the index of the smallest discriminant score among the vector of discriminant scores that it receives as input.

To begin this example, we include the class header file. We also include the header file for the `std::vector` class that will be the container for the discriminant scores.

```
#include "itkMinimumDecisionRule.h"
#include <vector>
```

The instantiation of the function is done through the usual `New()` method and a smart pointer.

```
  typedef itk::MinimumDecisionRule DecisionRuleType;
  DecisionRuleType::Pointer decisionRule = DecisionRuleType::New();
```

We create the discriminant score vector and fill it with three values. The call `Evaluate( discriminantScores )` will return 0 because the first value is the smallest value.

```
  std::vector< double > discriminantScores;
  discriminantScores.push_back( 0.1 );
  discriminantScores.push_back( 0.3 );
  discriminantScores.push_back( 0.6 );

  std::cout << "MinimumDecisionRule: The index of the chosen = "
            << decisionRule->Evaluate( discriminantScores )
            << std::endl;
```

Maximum Ratio Decision Rule

The source code for this section can be found in the file
`Examples/Statistics/MaximumRatioDecisionRule.cxx`.

The `Evaluate()` method of the `itk::MaximumRatioDecisionRule` returns the index, i if

$$\frac{f_i(\overrightarrow{x})}{f_j(\overrightarrow{x})} > \frac{K_j}{K_i} \text{ for all } j \neq i \tag{10.2}$$

where the i is the index of a class which has membership function f_i and its prior value (usually, the *a priori* probability of the class) is K_i

We include the header files for the class as well as the header file for the `std::vector` class that will be the container for the discriminant scores.

```
#include "itkMaximumRatioDecisionRule.h"
#include <vector>
```

The instantiation of the function is done through the usual `New()` method and a smart pointer.

```
  typedef itk::MaximumRatioDecisionRule DecisionRuleType;
  DecisionRuleType::Pointer decisionRule = DecisionRuleType::New();
```

We create the discriminant score vector and fill it with three values. We also create a vector (`aPrioris`) for the *a priori* values. The `Evaluate( discriminantScores )` will return 1.

```
  std::vector< double > discriminantScores;
  discriminantScores.push_back( 0.1 );
  discriminantScores.push_back( 0.3 );
  discriminantScores.push_back( 0.6 );

  DecisionRuleType::APrioriVectorType aPrioris;
  aPrioris.push_back( 0.1 );
  aPrioris.push_back( 0.8 );
  aPrioris.push_back( 0.1 );

  decisionRule->SetAPriori( aPrioris );
  std::cout << "MaximumRatioDecisionRule: The index of the chosen = "
            << decisionRule->Evaluate( discriminantScores )
            << std::endl;
```

10.2.7 Random Variable Generation

A random variable generation class returns a variate when the `GetVariate()` method is called. When we repeatedly call the method for "enough" times, the set of variates we will get follows the distribution form of the random variable generation class.

Normal (Gaussian) Distribution

The source code for this section can be found in the file
`Examples/Statistics/NormalVariateGenerator.cxx`.

The `itk::Statistics::NormalVariateGenerator` generates random variables according to the standard normal distribution (mean = 0, standard deviation = 1).

To use the class in a project, we must link the `itkStatistics` library to the project.

To begin the example we include the header file for the class.

```
#include "itkNormalVariateGenerator.h"
```

The NormalVariateGenerator is a non-templated class. We simply call the `New()` method to create an instance. Then, we provide the seed value using the `Initialize(seed value)`.

```
  typedef itk::Statistics::NormalVariateGenerator GeneratorType;
  GeneratorType::Pointer generator = GeneratorType::New();
  generator->Initialize( (int) 2003 );

  for ( unsigned int i = 0 ; i < 50 ; ++i )
    {
    std::cout << i << " : \t" << generator->GetVariate() << std::endl;
    }
```

10.3 Statistics applied to Images

10.3.1 Image Histograms

Scalar Image Histogram with Adaptor

The source code for this section can be found in the file
`Examples/Statistics/ImageHistogram1.cxx`.

This example shows how to compute the histogram of a scalar image. Since the statistics framework classes operate on Samples and ListOfSamples, we need to introduce a class that will make the image look like a list of samples. This class is the itk::Statistics::ScalarImageToListAdaptor. Once we have connected this adaptor to an image, we can proceed to use the itk::Statistics::ListSampleToHistogramGenerator in order to compute the histogram of the image.

First, we need to include the headers for the itk::Statistics::ScalarImageToListAdaptor and the itk::Image classes.

```
#include "itkScalarImageToListAdaptor.h"
#include "itkImage.h"
```

Now we include the headers for the ListSampleToHistogramGenerator and the reader that we will use for reading the image from a file.

```
#include "itkImageFileReader.h"
#include "itkListSampleToHistogramGenerator.h"
```

The image type must be defined using the typical pair of pixel type and dimension specification.

```
  typedef unsigned char         PixelType;
  const unsigned int            Dimension = 2;

  typedef itk::Image<PixelType, Dimension > ImageType;
```

Using the same image type we instantiate the type of the image reader that will provide the image source for our example.

```
  typedef itk::ImageFileReader< ImageType > ReaderType;

  ReaderType::Pointer reader = ReaderType::New();

  reader->SetFileName( argv[1] );
```

Now we introduce the central piece of this example, which is the use of the adaptor that will present the itk::Image as if it was a list of samples. We instantiate the type of the adaptor by using the actual image type. Then construct the adaptor by invoking its New() method and assigning the result to the corresponding smart pointer. Finally we connect the output of the image reader to the input of the adaptor.

```
  typedef itk::Statistics::ScalarImageToListAdaptor< ImageType >   AdaptorType;

  AdaptorType::Pointer adaptor = AdaptorType::New();

  adaptor->SetImage(  reader->GetOutput() );
```

You must keep in mind that adaptors are not pipeline objects. This means that they do not propagate update calls. It is therefore your responsibility to make sure that you invoke the `Update()` method of the reader before you attempt to use the output of the adaptor. As usual, this must be done inside a try/catch block because the read operation can potentially throw exceptions.

```
try
  {
  reader->Update();
  }
catch( itk::ExceptionObject & excp )
  {
  std::cerr << "Problem reading image file : " << argv[1] << std::endl;
  std::cerr << excp << std::endl;
  return -1;
  }
```

At this point, we are ready for instantiating the type of the histogram generator. Note that the adaptor type is used as template parameter of the generator. Having instantiated this type, we proceed to create one generator by invoking its `New()` method.

```
typedef PixelType        HistogramMeasurementType;

typedef itk::Statistics::ListSampleToHistogramGenerator<
                                              AdaptorType,
                                              HistogramMeasurementType
                                                              > GeneratorType;

GeneratorType::Pointer generator = GeneratorType::New();
```

We define now the characteristics of the Histogram that we want to compute. This typically includes the size of each one of the component, but given that in this simple example we are dealing with a scalar image, then our histogram will have a single component. For the sake of generality, however, we use the `HistogramType` as defined inside of the Generator type. We define also the marginal scale factor that will control the precision used when assigning values to histogram bins. Finally we invoke the `Update()` method in the generator.

```
typedef GeneratorType::HistogramType  HistogramType;

HistogramType::SizeType size;
size.Fill( 255 );

generator->SetListSample( adaptor );
generator->SetNumberOfBins( size );
generator->SetMarginalScale( 10.0 );

generator->Update();
```

Now we are ready for using the image histogram for any further processing. The histogram is obtained from the generator by invoking the `GetOutput()` method.

```
  HistogramType::ConstPointer histogram = generator->GetOutput();
```

In this current example we simply print out the frequency values of all the bins in the image histogram.

```
  const unsigned int histogramSize = histogram->Size();

  std::cout << "Histogram size " << histogramSize << std::endl;

  for( unsigned int bin=0; bin < histogramSize; bin++ )
    {
    std::cout << "bin = " << bin << " frequency = ";
    std::cout << histogram->GetFrequency( bin, 0 ) <<std::endl;
    }
```

Scalar Image Histogram with Generator

The source code for this section can be found in the file
`Examples/Statistics/ImageHistogram2.cxx`.

From the previous example you will have noticed that there is a significant number of operations to perform to compute the simple histogram of a scalar image. Given that this is a relatively common operation, it is convenient to encapsulate many of these operations in a single helper class.

The `itk::Statistics::ScalarImageToHistogramGenerator` is the result of such encapsulation. This example illustrates how to compute the histogram of a scalar image using this helper class.

We should first include the header of the histogram generator and the image class.

```
#include "itkScalarImageToHistogramGenerator.h"
#include "itkImage.h"
```

The image type must be defined using the typical pair of pixel type and dimension specification.

```
  typedef unsigned char       PixelType;
  const unsigned int          Dimension = 2;

  typedef itk::Image<PixelType, Dimension > ImageType;
```

We use now the image type in order to instantiate the type of the corresponding histogram generator class, and invoke its `New()` method in order to construct one.

```
typedef itk::Statistics::ScalarImageToHistogramGenerator<
                               ImageType >   HistogramGeneratorType;

HistogramGeneratorType::Pointer histogramGenerator =
                                      HistogramGeneratorType::New();
```

The image to be passed as input to the histogram generator is taken in this case from the output of an image reader.

```
histogramGenerator->SetInput(  reader->GetOutput() );
```

We define also the typical parameters that specify the characteristics of the histogram to be computed.

```
histogramGenerator->SetNumberOfBins( 255 );
histogramGenerator->SetMarginalScale( 10.0 );
```

Finally we trigger the computation of the histogram by invoking the `Compute()` method of the generator. Note again, that a generator is not a pipeline object and therefore it is up to you to make sure that the filters providing the input image have been updated.

```
histogramGenerator->Compute();
```

The resulting histogram can be obtained from the generator by invoking its `GetOutput()` method. It is also convenient to get the Histogram type from the traits of the generator type itself as shown in the code below.

```
typedef HistogramGeneratorType::HistogramType  HistogramType;

const HistogramType * histogram = histogramGenerator->GetOutput();
```

In this case we simply print out the frequency values of the histogram. These values can be accessed by using iterators.

```
HistogramType::ConstIterator itr = histogram->Begin();
HistogramType::ConstIterator end = histogram->End();

unsigned int binNumber = 0;
while( itr != end )
  {
  std::cout << "bin = " << binNumber << " frequency = ";
  std::cout << itr.GetFrequency() << std::endl;
  ++itr;
  ++binNumber;
  }
```

Color Image Histogram with Generator

The source code for this section can be found in the file
`Examples/Statistics/ImageHistogram3.cxx`.

By now, you are probably thinking that the statistics framework in ITK is too complex for simply computing histograms from images. Here we illustrate that the benefit for this complexity is the power that these methods provide for dealing with more complex and realistic uses of image statistics than the trivial 256-bin histogram of 8-bit images that most software packages provide. One of such cases is the computation of histograms from multi-component images such as Vector images and color images.

This example shows how to compute the histogram of an RGB image by using the helper class `ImageToHistogramGenerator`. In this first example we compute the histogram of each channel independently.

We start by including the header of the `itk::Statistics::ImageToHistogramGenerator`, as well as the headers for the image class and the RGBPixel class.

```
#include "itkImageToHistogramGenerator.h"
#include "itkImage.h"
#include "itkRGBPixel.h"
```

The type of the RGB image is defined by first instantiating a RGBPixel and then using the image dimension specification.

```
  typedef unsigned char                             PixelComponentType;

  typedef itk::RGBPixel< PixelComponentType >   RGBPixelType;

  const unsigned int                              Dimension = 2;

  typedef itk::Image< RGBPixelType, Dimension > RGBImageType;
```

Using the RGB image type we can instantiate the type of the corresponding histogram generator and construct one generator by invoking its `New()` method.

```
  typedef itk::Statistics::ImageToHistogramGenerator<
                            RGBImageType >   HistogramGeneratorType;

  HistogramGeneratorType::Pointer histogramGenerator =
                                          HistogramGeneratorType::New();
```

The parameters of the histogram must be defined now. Probably the most important one is the arrangement of histogram bins. This is provided to the histogram through a size array. The type of the array can be taken from the traits of the `HistogramGeneratorType` type. We create one instance of the size object and fill in its content. In this particular case, the three components of the size array will correspond to the number of bins used for each one of the RGB components in the color image. The following lines show how to define a histogram on the red component of the image while disregarding the green and blue components.

```
typedef HistogramGeneratorType::SizeType   SizeType;

SizeType size;

size[0] = 255;        // number of bins for the Red   channel
size[1] =   1;        // number of bins for the Green channel
size[2] =   1;        // number of bins for the Blue  channel

histogramGenerator->SetNumberOfBins( size );
```

The marginal scale must be defined in the generator. This will determine the precision in the assignment of values to the histogram bins.

```
histogramGenerator->SetMarginalScale( 10.0 );
```

The input of the generator is taken from an image reader, and the computation of the histogram is triggered by invoking the `Compute()` method of the generator.

```
histogramGenerator->SetInput(  reader->GetOutput()  );

histogramGenerator->Compute();
```

We can now access the results of the histogram computation by declaring a pointer to histogram and getting its value from the generator using the `GetOutput()` method. Note that here we use a `const HistogramType` pointer instead of a const smart pointer because we are sure that the generator is not going to be destroyed while we access the values of the histogram. Depending on what you are doing, it may be safer to assign the histogram to a const smart pointer as shown in previous examples.

```
typedef HistogramGeneratorType::HistogramType  HistogramType;

const HistogramType * histogram = histogramGenerator->GetOutput();
```

Just for the sake of exercising the experimental method [66], we verify that the resulting histogram actually have the size that we requested when we configured the generator. This can be done by invoking the `Size()` method of the histogram and printing out the result.

```
const unsigned int histogramSize = histogram->Size();

std::cout << "Histogram size " << histogramSize << std::endl;
```

Strictly speaking, the histogram computed here is the joint histogram of the three RGB components. However, given that we set the resolution of the green and blue channels to be just one bin, the histogram is in practice representing just the red channel. In the general case, we can alway access the frequency of a particular channel in a joint histogram, thanks to the fact that the histogram class offers a `GetFrequency()` method that accepts a channel as argument. This is illustrated in the following lines of code.

```
unsigned int channel = 0;  // red channel

std::cout << "Histogram of the red component" << std::endl;

for( unsigned int bin=0; bin < histogramSize; bin++ )
  {
  std::cout << "bin = " << bin << " frequency = ";
  std::cout << histogram->GetFrequency( bin, channel ) << std::endl;
  }
```

In order to reinforce the concepts presented above, we modify now the setup of the histogram generator in order to compute the histogram of the green channel instead of the red one. This is done by simply changing the number of bins desired on each channel and invoking the computation of the generator again by calling the `Compute()` method.

```
size[0] =   1;  // number of bins for the Red   channel
size[1] = 255;  // number of bins for the Green channel
size[2] =   1;  // number of bins for the Blue  channel

histogramGenerator->SetNumberOfBins( size );

histogramGenerator->Compute();
```

The result can be verified now by setting the desired channel to green and invoking the `GetFrequency()` method.

```
channel = 1;  // green channel

std::cout << "Histogram of the green component" << std::endl;

for( unsigned int bin=0; bin < histogramSize; bin++ )
  {
  std::cout << "bin = " << bin << " frequency = ";
  std::cout << histogram->GetFrequency( bin, channel ) << std::endl;
  }
```

To finalize the example, we do the same computation for the case of the blue channel.

```
  size[0] =   1;  // number of bins for the Red   channel
  size[1] =   1;  // number of bins for the Green channel
  size[2] = 255;  // number of bins for the Blue  channel

  histogramGenerator->SetNumberOfBins( size );

  histogramGenerator->Compute();
```

and verify the output.

```
  channel = 2;  // blue channel

  std::cout << "Histogram of the blue component" << std::endl;

  for( unsigned int bin=0; bin < histogramSize; bin++ )
    {
    std::cout << "bin = " << bin << " frequency = ";
    std::cout << histogram->GetFrequency( bin, channel ) << std::endl;
    }
```

Color Image Histogram Writing

The source code for this section can be found in the file
`Examples/Statistics/ImageHistogram4.cxx`.

The statistics framework in ITK has been designed for managing multi-variate statistics in a natural way. The `itk::Statistics::Histogram` class reflects this concept clearly since it is a N-variable joint histogram. This nature of the Histogram class is exploited in the following example in order to build the joint histogram of a color image encoded in RGB values.

Note that the same treatment could be applied further to any vector image thanks to the generic programming approach used in the implementation of the statistical framework.

The most relevant class in this example is the `itk::Statistics::ImageToHistogramGenerator`. This class will take care of adapting the `itk::Image` to a list of samples and then to a histogram generator. The user is only bound to provide the desired resolution on the histogram bins for each one of the image components.

In this example we compute the joint histogram of the three channels of an RGB image. Our output histogram will be equivalent to a 3D array of bins. This histogram could be used further for feeding a segmentation method based on statistical pattern recognition. Such method was actually used during the generation of the image in the cover of the Software Guide.

The first step is to include the header files for the histogram generator, the RGB pixel type and the Image.

```
#include "itkImageToHistogramGenerator.h"
```

```
#include "itkImage.h"
#include "itkRGBPixel.h"
```

We declare now the type used for the components of the RGB pixel, instantiate the type of the RGBPixel and instantiate the image type.

```
  typedef unsigned char                          PixelComponentType;

  typedef itk::RGBPixel< PixelComponentType >   RGBPixelType;

  const unsigned int                             Dimension = 2;

  typedef itk::Image< RGBPixelType, Dimension > RGBImageType;
```

Using the type of the color image, and in general of any vector image, we can now instantiate the type of the histogram generator class. We then use that type for constructing an instance of the generator by invoking its `New()` method and assigning the result to a smart pointer.

```
  typedef itk::Statistics::ImageToHistogramGenerator<
                               RGBImageType >   HistogramGeneratorType;

  HistogramGeneratorType::Pointer histogramGenerator =
                                        HistogramGeneratorType::New();
```

The resolution at which the statistics of each one of the color component will be evaluated is defined by setting the number of bins along every component in the joint histogram. For this purpose we take the `SizeType` trait from the generator and use it to instantiate a `size` variable. We set in this variable the number of bins to use for each component of the color image.

```
  typedef HistogramGeneratorType::SizeType   SizeType;

  SizeType size;

  size[0] = 255;  // number of bins for the Red   channel
  size[1] = 255;  // number of bins for the Green channel
  size[2] = 255;  // number of bins for the Blue  channel

  histogramGenerator->SetNumberOfBins( size );
```

The input to the histogram generator is taken from the output of an image reader. Of course, the output of any filter producing an RGB image could have been used instead of a reader.

```
  histogramGenerator->SetInput(  reader->GetOutput()  );
```

The marginal scale is defined in the histogram generator. This value will define the precision in the assignment of values to the histogram bins.

```
histogramGenerator->SetMarginalScale( 10.0 );
```

Finally, the computation of the histogram is triggered by invoking the `Compute()` method of the generator. Note that generators are not pipeline objects. It is therefore your responsibility to make sure that you update the filter that provides the input image to the generator.

```
histogramGenerator->Compute();
```

At this point, we can recover the histogram by calling the `GetOutput()` method of the generator. The result is assigned to a variable that is instantiated using the `HistogramType` trait of the generator type.

```
typedef HistogramGeneratorType::HistogramType  HistogramType;

const HistogramType * histogram = histogramGenerator->GetOutput();
```

We can verify that the computed histogram has the requested size by invoking its `Size()` method.

```
const unsigned int histogramSize = histogram->Size();

std::cout << "Histogram size " << histogramSize << std::endl;
```

The values of the histogram can now be saved into a file by walking through all of the histogram bins and pushing them into a std::ofstream.

```
std::ofstream histogramFile;
histogramFile.open( argv[2] );

HistogramType::ConstIterator itr = histogram->Begin();
HistogramType::ConstIterator end = histogram->End();

typedef HistogramType::FrequencyType FrequencyType;

while( itr != end )
  {
  const FrequencyType frequency = itr.GetFrequency();
  histogramFile.write( (const char *)(&frequency), sizeof(frequency) );
  ++itr;
  }

histogramFile.close();
```

Note that here the histogram is saved as a block of memory in a raw file. At this point you can use visualization software in order to explore the histogram in a display that would be equivalent to a scatter plot of the RGB components of the input color image.

10.3.2 Image Information Theory

Many concepts from Information Theory have been used successfully in the domain of image processing. This section introduces some of such concepts and illustrates how the statistical framework in ITK can be used for computing measures that have some relevance in terms of Information Theory [75, 76, 47].

Computing Image Entropy

The concept of Entropy has been introduced into image processing as a crude mapping from its application in Communications. The notions of Information Theory can be deceiving and misleading when applied to images because their language from Communication Theory does not necessarily maps to what people in the Imaging Community use.

For example, it is commonly said that

"The Entropy of an image is a measure of the amount of information contained in an image".

This statement is fundamentally **incorrect**.

The way the notion of Entropy is commonly measured in images is by first assuming that the spatial location of a pixel in an image is irrelevant! That is, we simply take the statistical distribution of the pixel values as it can be evaluated in a histogram and from that histogram we estimate the frequency of the value associated to each bin. In other words, we simply assume that the image is a set of pixels that are passing through a channel, just as things are commonly considered for communication purposes.

Once the frequency of every pixel value has been estimated, Information Theory defines that the amount of uncertainty that an observer will lose by taking one pixel and finding its real value to be the one associated with the i-th bin of the histogram, is given by $-\log_2(p_i)$, where p_i is the frequency in that histogram bin. Since a reduction in uncertainty is equivalent to an increase in the amount of information in the observer, we conclude that measuring one pixel and finding its level to be in the i-th bin results in an acquisition of $-\log_2(p_i)$ bits of information[1].

Since we could have picked any pixel at random, our chances or picking the ones that are associated to the i-th histogram bin are given by p_i. Therefore, the expected reduction in uncertainty

[1]Note that **bit** is the unit of amount of information. Our modern culture has vulgarized the bit and its multiples, the Byte, KiloByte, MegaByte, GigaByte and so on as simple measures of the amount of RAM memory and capacity of a hard drive in a computer. In that sense, a confusion is created between the encoding of a piece of data and its actual amount of information. For example a file composed of one million letters will take one million bytes in a hard disk, but it does not necessarily has one million bytes of information, since in many cases parts of the file can be predicted from others. This is the reason why data compression can manage to compact files.

that we can get from measuring the value of one pixel is given by

$$H = -\sum_i p_i \cdot \log_2 (p_i) \tag{10.3}$$

This quantity H is what is usually defined as the *Entropy of the Image*. It would be more accurate to call it the Entropy of the random variable associated to the intensity value of *one* pixel. The fact that H is unrelated to the spatial arrangement of the pixels in an image shows how little of the real *Image Information* is H actually representing. The Entropy of an image, as measured above, is only a crude indication of how the intensity values are spread in the dynamic range of intensities. For example, an image with maximum entropy will be the one that has a large dynamic range and every value in that range is equally probable.

The common acceptation of H as a representation of image information has terribly undermined the enormous potential on the application of Information Theory to image processing and analysis.

The real concepts of Information Theory would require that we define the amount of information in an image based on our expectations and prior knowledge from that image. In particular, the *Amount of Information* provided by an image should measure the number of features that we are not able to predict based on our prior knowledge about that image. For example, if we know that we are going to analyze a CT scan of the abdomen of an adult human male in the age range of 40 to 45, there is already a good deal that we could predict about the content of that image. The real amount of information in the image is the representation of the features in the image that we could not predict from knowing that it is a CT scan from a human adult male.

The application of Information Theory to image analysis is still in its early infancy and it is an exciting an promising field to be explored further. All that being said, let's now look closer at how the concept of Entropy (which is not the amount of information in an image) can be measured with the ITK statistics framework.

The source code for this section can be found in the file
`Examples/Statistics/ImageEntropy1.cxx`.

This example shows how to compute the entropy of an image. More formally this should be said : The reduction in uncertainty gained when we measure the intensity of *one* randomly selected pixel in this image, given that we already know the statistical distribution of the image intensity values.

In practice it is almost never possible to know the real statistical distribution of intensities and we are force to estimate it from the evaluation of the histogram from one or several images of similar nature. We can use the counts in histogram bins in order to compute frequencies and then consider those frequencies to be estimations of the probablility of a new value to belong to the intensity range of that bin.

Since the first stage in estimating the entropy of an image is to compute its histogram, we must start by including the headers of the classes that will perform such computation. In this case, we are going to use a scalar image as input, therefore we need the itk::Statistics::ScalarImageToHistogramGenerator class, as well as the image class.

```
#include "itkScalarImageToHistogramGenerator.h"
#include "itkImage.h"
```

The pixel type and dimension of the image are explicitly declared and then used for instantiating the image type.

```
  typedef unsigned char       PixelType;
  const   unsigned int        Dimension = 2;

  typedef itk::Image< PixelType, Dimension > ImageType;
```

The image type is used as template parameter for instantiating the histogram generator.

```
  typedef itk::Statistics::ScalarImageToHistogramGenerator<
                                    ImageType >   HistogramGeneratorType;

  HistogramGeneratorType::Pointer histogramGenerator =
                                    HistogramGeneratorType::New();
```

The parameters of the desired histogram are defined. In particular, the number of bins and the marginal scale. For convenience in this example, we read the number of bins from the command line arguments. In this way we can easily experiment with different values for the number of bins and see how that choice affects the computation of the entropy.

```
  const unsigned int numberOfHistogramBins = atoi( argv[2] );

  histogramGenerator->SetNumberOfBins( numberOfHistogramBins );
  histogramGenerator->SetMarginalScale( 10.0 );
```

We can then connect as input the output image from a reader and trigger the histogram computation by invoking the `Compute()` method in the generator.

```
  histogramGenerator->SetInput(  reader->GetOutput() );

  histogramGenerator->Compute();
```

The resulting histogram can be recovered from the generator by using the `GetOutput()` method. A histogram class can be declared using the `HistogramType` trait from the generator.

```
  typedef HistogramGeneratorType::HistogramType  HistogramType;

  const HistogramType * histogram = histogramGenerator->GetOutput();
```

We proceed now to compute the *estimation* of entropy given the histogram. The first conceptual jump to be done here is that we assume that the histogram, which is the simple count of frequency of occurrence for the gray scale values of the image pixels, can be normalized in order to estimate the probability density function **PDF** of the actual statistical distribution of pixel values.

First we declare an iterator that will visit all the bins in the histogram. Then we obtain the total number of counts using the `GetTotalFrequency()` method, and we initialize the entropy variable to zero.

```
HistogramType::ConstIterator itr = histogram->Begin();
HistogramType::ConstIterator end = histogram->End();

double Sum = histogram->GetTotalFrequency();

double Entropy = 0.0;
```

We start now visiting every bin and estimating the probability of a pixel to have a value in the range of that bin. The base 2 logarithm of that probability is computed, and then weighted by the probability in order to compute the expected amount of information for any given pixel. Note that a minimum value is imposed for the probability in order to avoid computing logarithms of zeros.

Note that the $\log(2)$ factor is used to convert the natural logarithm in to a logarithm of base 2, and make possible to report the entropy in its natural unit: the bit.

```
while( itr != end )
  {
  const double probability = itr.GetFrequency() / Sum;

  if( probability > 0.99 / Sum )
    {
    Entropy += - probability * log( probability ) / log( 2.0 );
    }
  ++itr;
  }
```

The result of this sum is considered to be our estimation of the image entropy. Note that the Entrpy value will change depending on the number of histogram bins that we use for computing the histogram. This is particularly important when dealing with images whose pixel values have dynamic ranges so large that our number of bins will always underestimate the variability of the data.

```
std::cout << "Image entropy = " << Entropy << " bits " << std::endl;
```

As an illustration, the application of this program to the image

- `Examples/Data/BrainProtonDensitySlice.png`

results in the following values of entropy for different values of number of histogram bins.

Number of Histogram Bins	16	32	64	128	255
Estimated Entropy (bits)	3.02	3.98	4.92	5.89	6.88

This table highlights the importance of carefully considering the characteristics of the histograms used for estimating Information Theory measures such as the entropy.

Computing Images Mutual Information

The source code for this section can be found in the file
`Examples/Statistics/ImageMutualInformation1.cxx`.

This example illustrates how to compute the Mutual Information between two images using classes from the Statistics framework. Note that you could also use for this purpose the Image-Metrics designed for the image registration framework.

For example, you could use:

- `itk::MutualInformationImageToImageMetric`
- `itk::MattesMutualInformationImageToImageMetric`
- `itk::MutualInformationHistogramImageToImageMetric`
- `itk::MutualInformationImageToImageMetric`
- `itk::NormalizedMutualInformationHistogramImageToImageMetric`
- `itk::KullbackLeiblerCompareHistogramImageToImageMetric`

Mutual Information as computed in this example, and as commonly used in the context of image registration provides a measure of how much uncertainty on the value of a pixel in one image is reduced by measuring the homologous pixel in the other image. Note that Mutual Information as used here does not measures the amount of information that one image provides on the other image, such measure would have required to take into account the spatial structures in the images as well as the semantics of the image context in terms of an observer.

This implies that there is still an enormous unexploited potential on the use of the Mutual Information concept in the domain of medical images. Probably the most interesting of which would be the semantic description of image on terms of anatomical structures.

In this particular example we make use of classes from the Statistics framework in order to compute the measure of Mutual Information between two images. We assume that both images have the same number of pixels along every dimension and that they have the same origin and spacing. Therefore the pixels from one image are perfectly aligned with those of the other image.

We must start by including the header files of the image, histogram generator, reader and Join image filter. We will read both images and use the Join image filter in order to compose an image of two components using the information of each one of the input images in one component. This is the natural way of using the Statistics framework in ITK given that the fundamental statistical classes are expecting to receive multi-valued measures.

```
#include "itkImage.h"
#include "itkImageFileReader.h"
#include "itkJoinImageFilter.h"
#include "itkImageToHistogramGenerator.h"
```

We define the pixel type and dimension of the images to be read.

```
  typedef unsigned char                                 PixelComponentType;
  const unsigned int                                    Dimension = 2;

  typedef itk::Image< PixelComponentType, Dimension >   ImageType;
```

Using the image type we proceed to instantiate the readers for both input images. Then, we take their filenames from the command line arguments.

```
  typedef itk::ImageFileReader< ImageType >             ReaderType;

  ReaderType::Pointer reader1 = ReaderType::New();
  ReaderType::Pointer reader2 = ReaderType::New();

  reader1->SetFileName( argv[1] );
  reader2->SetFileName( argv[2] );
```

Using the `itk::JoinImageFilter` we use the two input images and put them together in an image of two components.

```
  typedef itk::JoinImageFilter< ImageType, ImageType >  JoinFilterType;

  JoinFilterType::Pointer joinFilter = JoinFilterType::New();

  joinFilter->SetInput1( reader1->GetOutput() );
  joinFilter->SetInput2( reader2->GetOutput() );
```

At this point we trigger the execution of the pipeline by invoking the `Update()` method on the Join filter. We must put the call inside a try/catch block because the Update() call may potentially result in exceptions being thrown.

```
  try
    {
    joinFilter->Update();
    }
  catch( itk::ExceptionObject & excp )
    {
    std::cerr << excp << std::endl;
    return -1;
    }
```

We prepare now the types to be used for the computation of the Joint histogram. For this purpose, we take the type of the image resulting from the JoinImageFilter and use it as template argument of the itk::ImageToHistogramGenerator. We then construct one by invoking the New() method.

```
  typedef JoinFilterType::OutputImageType               VectorImageType;

  typedef itk::Statistics::ImageToHistogramGenerator<
                                       VectorImageType >  HistogramGeneratorType;

  HistogramGeneratorType::Pointer histogramGenerator = HistogramGeneratorType::New();
```

We pass the multiple components image as input to the histogram generator, and setup the marginal scale value that will define the precision to be used for classifying values into the histogram bins.

```
  histogramGenerator->SetInput(  joinFilter->GetOutput()  );

  histogramGenerator->SetMarginalScale( 10.0 );
```

We must now define the number of bins to use for each one of the components in the joint image. For this purpose we take the SizeType from the traits of the histogram generator type. The array of number of bins is passed to the generator and we can then invoke the Compute() method in order to trigger the computation of the joint histogram.

```
  typedef HistogramGeneratorType::SizeType   SizeType;

  SizeType size;

  size[0] = 255;  // number of bins for the first  channel
  size[1] = 255;  // number of bins for the second channel

  histogramGenerator->SetNumberOfBins( size );
  histogramGenerator->Compute();
```

The histogram can be recovered from the generator by creating a variable with the histogram type taken from the generator traits.

```
typedef HistogramGeneratorType::HistogramType  HistogramType;

const HistogramType * histogram = histogramGenerator->GetOutput();
```

We now walk over all the bins of the joint histogram and compute their contribution to the value of the joint Entropy. For this purpose we use histogram iterators, and the `Begin()` and `End()` methods. Since the values returned from the histogram are measuring frequency we must convert them to an estimation of probability by dividing them over the total sum of frequencies returned by the `GetTotalFrequency()` method.

```
HistogramType::ConstIterator itr = histogram->Begin();
HistogramType::ConstIterator end = histogram->End();

const double Sum = histogram->GetTotalFrequency();
```

We initialize to zero the variable to use for accumulating the value of the joint entropy, and then use the iterator for visiting all the bins of the joint histogram. For every bin we compute their contribution to the reduction of uncertainty. Note that in order to avoid logarithmic operations on zero values, we skip over those bins that have less than one count. The entropy contribution must be computed using logarithms in base two in order to be able express entropy in **bits**.

```
double JointEntropy = 0.0;

while( itr != end )
  {
  const double count = itr.GetFrequency();
  if( count > 0.0 )
    {
    const double probability = count / Sum;
    JointEntropy += - probability * log( probability ) / log( 2.0 );
    }
  ++itr;
  }
```

Now that we have the value of the joint entropy we can proceed to estimate the values of the entropies for each image independently. This can be done by simply changing the number of bins and then recomputing the histogram.

```
size[0] = 255;  // number of bins for the first  channel
size[1] =   1;  // number of bins for the second channel

histogramGenerator->SetNumberOfBins( size );
histogramGenerator->Compute();
```

We initialize to zero another variable in order to start accumulating the entropy contributions from every bin.

```
itr = histogram->Begin();
end = histogram->End();

double Entropy1 = 0.0;

while( itr != end )
  {
  const double count = itr.GetFrequency();
  if( count > 0.0 )
    {
    const double probability = count / Sum;
    Entropy1 += - probability * log( probability ) / log( 2.0 );
    }
  ++itr;
  }
```

The same process is used for computing the entropy of the other component. Simply by swapping the number of bins in the histogram.

```
size[0] =   1;  // number of bins for the first channel
size[1] = 255;  // number of bins for the second channel

histogramGenerator->SetNumberOfBins( size );
histogramGenerator->Compute();
```

The entropy is computed in a similar manner, just by visiting all the bins on the histogram and accumulating their entropy contributions.

```
itr = histogram->Begin();
end = histogram->End();

double Entropy2 = 0.0;

while( itr != end )
  {
  const double count = itr.GetFrequency();
  if( count > 0.0 )
    {
    const double probability = count / Sum;
    Entropy2 += - probability * log( probability ) / log( 2.0 );
    }
  ++itr;
  }
```

At this point we can compute any of the popular measures of Mutual Information. For example

```
double MutualInformation = Entropy1 + Entropy2 - JointEntropy;
```

or Normalized Mutual Information, where the value of Mutual Information gets divided by the mean entropy of the input images.

```
double NormalizedMutualInformation1 =
                    2.0 * MutualInformation / ( Entropy1 + Entropy2 );
```

A second form of Normalized Mutual Information has been defined as the mean entropy of the two images divided by their joint entropy.

```
double NormalizedMutualInformation2 = ( Entropy1 + Entropy2 ) / JointEntropy;
```

You probably will find very interesting how the value of Mutual Information is strongly dependent on the number of bins over which the histogram is defined.

10.4 Classification

In statistical classification, each object is represented by d features (a measurement vector), and the goal of classification becomes finding compact and disjoint regions (decision regions[24]) for classes in a d-dimensional feature space. Such decision regions are defined by decision rules that are known or can be trained. The simplest configuration of a classification consists of a decision rule and multiple membership functions; each membership function represents a class. Figure 10.3 illustrates this general framework.

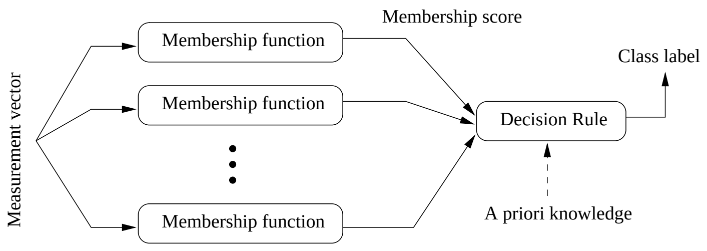


Figure 10.3: Simple conceptual classifier.

This framework closely follows that of Duda and Hart[24]. The classification process can be described as follows:

1. A measurement vector is input to each membership function.

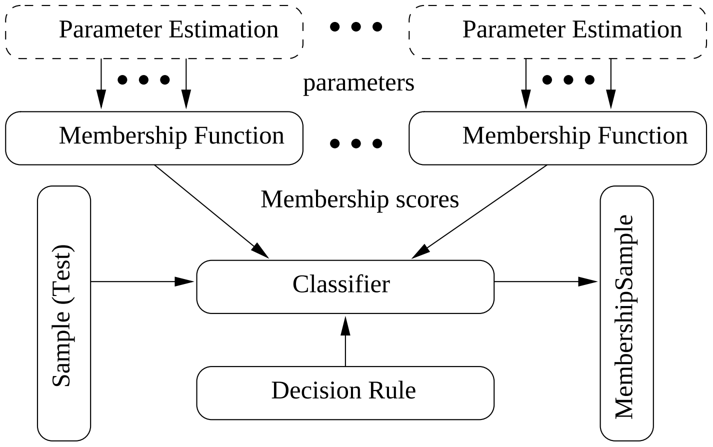


Figure 10.4: Statistical classification framework.

2. Membership functions feed the membership scores to the decision rule.
3. A decision rule compares the membership scores and returns a class label.

This simple configuration can be used to formulated various classification tasks by using different membership functions and incorporating task specific requirements and prior knowledge into the decision rule. For example, instead of using probability density functions as membership functions, through distance functions and a minimum value decision rule (which assigns a class from the distance function that returns the smallest value) users can achieve a least squared error classifier. As another example, users can add a rejection scheme to the decision rule so that even in a situation where the membership scores suggest a “winner”, a measurement vector can be flagged as ill defined. Such a rejection scheme can avoid risks of assigning a class label without a proper win margin.

10.4.1 k-d Tree Based k-Means Clustering

The source code for this section can be found in the file
`Examples/Statistics/KdTreeBasedKMeansClustering.cxx`.

K-means clustering is a popular clustering algorithm because it is simple and usually converges to a reasonable solution. The k-means algorithm works as follows:

1. Obtains the initial k means input from the user.
2. Assigns each measurement vector in a sample container to its closest mean among the k number of means (i.e., update the membership of each measurement vectors to the nearest of the k clusters).

3. Calculates each cluster's mean from the newly assigned measurement vectors (updates the centroid (mean) of k clusters).

4. Repeats step 2 and step 3 until it meets the termination criteria.

The most common termination criteria is that if there is no measurement vector that changes its cluster membership from the previous iteration, then the algorithm stops.

The itk::Statistics::KdTreeBasedKmeansEstimator is a variation of this logic. The k-means clustering algorithm is computationally very expensive because it has to recalculate the mean at each iteration. To update the mean values, we have to calculate the distance between k means and each and every measurement vector. To reduce the computational burden, the KdTreeBasedKmeansEstimator uses a special data structure: the k-d tree (itk::Statistics::KdTree) with additional information. The additional information includes the number and the vector sum of measurement vectors under each node under the tree architecture.

With such additional information and the k-d tree data structure, we can reduce the computational cost of the distance calculation and means. Instead of calculating each measurement vectors and k means, we can simply compare each node of the k-d tree and the k means. This idea of utilizing a k-d tree can be found in multiple articles [2] [61] [43]. Our implementation of this scheme follows the article by the Kanungo et al [43].

We use the itk::Statistics::ListSample as the input sample, the itk::Vector as the measurement vector. The following code snippet includes their header files.

```
#include "itkVector.h"
#include "itkListSample.h"
```

Since our k-means algorithm requires a itk::Statistics::KdTree object as an input, we include the KdTree class header file. As mentioned above, we need a k-d tree with the vector sum and the number of measurement vectors. Therefore we use the itk::Statistics::WeightedCentroidKdTreeGenerator instead of the itk::Statistics::KdTreeGenerator that generate a k-d tree without such additional information.

```
#include "itkKdTree.h"
#include "itkWeightedCentroidKdTreeGenerator.h"
```

The KdTreeBasedKmeansEstimator class is the implementation of the k-means algorithm. It does not create k clusters. Instead, it returns the mean estimates for the k clusters.

```
#include "itkKdTreeBasedKmeansEstimator.h"
```

To generate the clusters, we must create k instances of itk::Statistics::EuclideanDistance function as the membership functions for each cluster and plug that—along with a sample—into an `itk::Statistics::SampleClassifier` object to get a `itk::Statistics::MembershipSample` that stores pairs of measurement vectors and their associated class labels (k labels).

```
#include "itkMinimumDecisionRule.h"
#include "itkEuclideanDistance.h"
#include "itkSampleClassifier.h"
```

We will fill the sample with random variables from two normal distribution using the `itk::Statistics::NormalVariateGenerator`.

```
#include "itkNormalVariateGenerator.h"
```

Since the `NormalVariateGenerator` class only supports 1-D, we define our measurement vector type as one component vector. We then, create a `ListSample` object for data inputs. Each measurement vector is of length 1. We set this using the `SetMeasurementVectorSize()` method.

```
  typedef itk::Vector< double, 1 > MeasurementVectorType;
  typedef itk::Statistics::ListSample< MeasurementVectorType > SampleType;
  SampleType::Pointer sample = SampleType::New();
  sample->SetMeasurementVectorSize( 1 );
```

The following code snippet creates a NormalVariateGenerator object. Since the random variable generator returns values according to the standard normal distribution (The mean is zero, and the standard deviation is one), before pushing random values into the `sample`, we change the mean and standard deviation. We want two normal (Gaussian) distribution data. We have two for loops. Each for loop uses different mean and standard deviation. Before we fill the `sample` with the second distribution data, we call `Initialize(random seed)` method, to recreate the pool of random variables in the `normalGenerator`.

To see the probability density plots from the two distribution, refer to the Figure 10.5.

```
  typedef itk::Statistics::NormalVariateGenerator NormalGeneratorType;
  NormalGeneratorType::Pointer normalGenerator = NormalGeneratorType::New();

  normalGenerator->Initialize( 101 );

  MeasurementVectorType mv;
  double mean = 100;
  double standardDeviation = 30;
  for ( unsigned int i = 0 ; i < 100 ; ++i )
    {
    mv[0] = ( normalGenerator->GetVariate() * standardDeviation ) + mean;
    sample->PushBack( mv );
    }
```

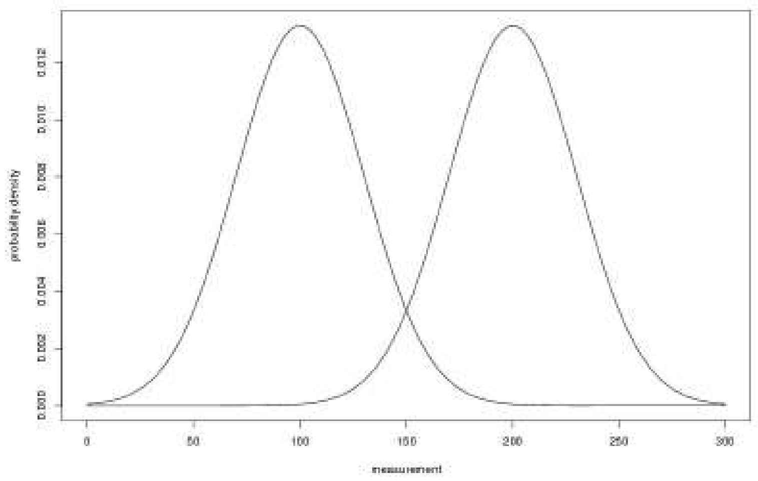


Figure 10.5: Two normal distributions' probability density plot (The means are 100 and 200, and the standard deviation is 30)

```
normalGenerator->Initialize( 3024 );
mean = 200;
standardDeviation = 30;
for ( unsigned int i = 0 ; i < 100 ; ++i )
  {
  mv[0] = ( normalGenerator->GetVariate() * standardDeviation ) + mean;
  sample->PushBack( mv );
  }
```

We create a k-d tree. To see the details on the k-d tree generation, see the Section 10.1.7.

```
typedef itk::Statistics::WeightedCentroidKdTreeGenerator< SampleType >
  TreeGeneratorType;
TreeGeneratorType::Pointer treeGenerator = TreeGeneratorType::New();

treeGenerator->SetSample( sample );
treeGenerator->SetBucketSize( 16 );
treeGenerator->Update();
```

Once we have the k-d tree, it is a simple procedure to produce k mean estimates.

We create the KdTreeBasedKmeansEstimator. Then, we provide the initial mean values using the `SetParameters()`. Since we are dealing with two normal distribution in a 1-D space, the size of the mean value array is two. The first element is the first mean value, and the second is the second mean value. If we used two normal distributions in a 2-D space, the size of array would be four, and the first two elements would be the two components of the first normal distribution's mean vector. We plug-in the k-d tree using the `SetKdTree()`.

The remaining two methods specify the termination condition. The estimation process stops when the number of iterations reaches the maximum iteration value set by the `SetMaximumIteration()`, or the distances between the newly calculated mean (centroid) values and previous ones are within the threshold set by the `SetCentroidPositionChangesThreshold()`. The final step is to call the `StartOptimization()` method.

The for loop will print out the mean estimates from the estimation process.

```
typedef TreeGeneratorType::KdTreeType TreeType;
typedef itk::Statistics::KdTreeBasedKmeansEstimator<TreeType> EstimatorType;
EstimatorType::Pointer estimator = EstimatorType::New();

EstimatorType::ParametersType initialMeans(2);
initialMeans[0] = 0.0;
initialMeans[1] = 0.0;

estimator->SetParameters( initialMeans );
estimator->SetKdTree( treeGenerator->GetOutput() );
estimator->SetMaximumIteration( 200 );
estimator->SetCentroidPositionChangesThreshold(0.0);
estimator->StartOptimization();

EstimatorType::ParametersType estimatedMeans = estimator->GetParameters();

for ( unsigned int i = 0 ; i < 2 ; ++i )
  {
  std::cout << "cluster[" << i << "] " << std::endl;
  std::cout << "    estimated mean : " << estimatedMeans[i] << std::endl;
  }
```

If we are only interested in finding the mean estimates, we might stop. However, to illustrate how a classifier can be formed using the statistical classification framework. We go a little bit further in this example.

Since the k-means algorithm is an minimum distance classifier using the estimated k means and the measurement vectors. We use the EuclideanDistance class as membership functions. Our choice for the decision rule is the `itk::Statistics::MinimumDecisionRule` that returns the index of the membership functions that have the smallest value for a measurement vector.

After creating a SampleClassifier object and a MinimumDecisionRule object, we plug-in the `decisionRule` and the `sample` to the classifier. Then, we must specify the number of classes that will be considered using the `SetNumberOfClasses()` method.

The remainder of the following code snippet shows how to use user-specified class labels. The classification result will be stored in a MembershipSample object, and for each measurement vector, its class label will be one of the two class labels, 100 and 200 (`unsigned int`).

```
typedef itk::Statistics::EuclideanDistance< MeasurementVectorType >
  MembershipFunctionType;
typedef itk::MinimumDecisionRule DecisionRuleType;
DecisionRuleType::Pointer decisionRule = DecisionRuleType::New();

typedef itk::Statistics::SampleClassifier< SampleType > ClassifierType;
ClassifierType::Pointer classifier = ClassifierType::New();

classifier->SetDecisionRule( (itk::DecisionRuleBase::Pointer) decisionRule);
classifier->SetSample( sample );
classifier->SetNumberOfClasses( 2 );

std::vector< unsigned int > classLabels;
classLabels.resize( 2 );
classLabels[0] = 100;
classLabels[1] = 200;

classifier->SetMembershipFunctionClassLabels( classLabels );
```

The `classifier` is almost ready to do the classification process except that it needs two membership functions that represents two clusters respectively.

In this example, the two clusters are modeled by two Euclidean distance functions. The distance function (model) has only one parameter, its mean (centroid) set by the `SetOrigin()` method. To plug-in two distance functions, we call the `AddMembershipFunction()` method. Then invocation of the `Update()` method will perform the classification.

```
std::vector< MembershipFunctionType::Pointer > membershipFunctions;
MembershipFunctionType::OriginType origin( sample->GetMeasurementVectorSize() );
int index = 0;
for ( unsigned int i = 0 ; i < 2 ; i++ )
  {
  membershipFunctions.push_back( MembershipFunctionType::New() );
  for ( unsigned int j = 0 ; j < sample->GetMeasurementVectorSize(); j++ )
    {
    origin[j] = estimatedMeans[index++];
    }
  membershipFunctions[i]->SetOrigin( origin );
  classifier->AddMembershipFunction( membershipFunctions[i].GetPointer() );
  }

classifier->Update();
```

The following code snippet prints out the measurement vectors and their class labels in the `sample`.

```
  ClassifierType::OutputType* membershipSample = classifier->GetOutput();
  ClassifierType::OutputType::ConstIterator iter = membershipSample->Begin();

  while ( iter != membershipSample->End() )
    {
    std::cout << "measurement vector = " << iter.GetMeasurementVector()
              << "class label = " << iter.GetClassLabel()
              << std::endl;
    ++iter;
    }
```

10.4.2 K-Means Classification

The source code for this section can be found in the file
`Examples/Statistics/ScalarImageKmeansClassifier.cxx`.

This example shows how to use the KMeans model for classifying the pixel of a scalar image.

The `itk::Statistics::ScalarImageKmeansImageFilter` is used for taking a scalar image and applying the K-Means algorithm in order to define classes that represents statistical distributions of intensity values in the pixels. The classes are then used in this filter for generating a labeled image where every pixel is assigned to one of the classes.

```
#include "itkImage.h"
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
#include "itkScalarImageKmeansImageFilter.h"
```

First we define the pixel type and dimension of the image that we intend to classify. With this image type we can =also declare the `itk::ImageFileReader` needed for reading the input image, create one and set its input filename.

```
  typedef signed short       PixelType;
  const unsigned int          Dimension = 2;

  typedef itk::Image<PixelType, Dimension > ImageType;

  typedef itk::ImageFileReader< ImageType > ReaderType;
  ReaderType::Pointer reader = ReaderType::New();
  reader->SetFileName( inputImageFileName );
```

With the `ImageType` we instantiate the type of the `itk::ScalarImageKmeansImageFilter` that will compute the K-Means model and then classify the image pixels.

```
typedef itk::ScalarImageKmeansImageFilter< ImageType > KMeansFilterType;

KMeansFilterType::Pointer kmeansFilter = KMeansFilterType::New();

kmeansFilter->SetInput( reader->GetOutput() );

const unsigned int numberOfInitialClasses = atoi( argv[4] );
```

In general the classification will produce as output an image whose pixel values are integers associated to the labels of the classes. Since typically these integers will be generated in order (0,1,2,...N), the output image will tend to look very dark when displayed with naive viewers. It is therefore convenient to have the option of spreading the label values over the dynamic range of the output image pixel type. When this is done, the dynamic range of the pixels is divide by the number of classes in order to define the increment between labels. For example, an output image of 8 bits will have a dynamic range of [0:256], and when it is used for holding four classes, the non-contiguous labels will be (0,64,128,192). The selection of the mode to use is done with the method `SetUseContiguousLabels()`.

```
const unsigned int useNonContiguousLabels = atoi( argv[3] );

kmeansFilter->SetUseNonContiguousLabels( useNonContiguousLabels );
```

For each one of the classes we must provide a tentative initial value for the mean of the class. Given that this is a scalar image, each one of the means is simply a scalar value. Note however that in a general case of K-Means, the input image would be a vector image and therefore the means will be vectors of the same dimension as the image pixels.

```
for( unsigned k=0; k < numberOfInitialClasses; k++ )
  {
  const double userProvidedInitialMean = atof( argv[k+argoffset] );
  kmeansFilter->AddClassWithInitialMean( userProvidedInitialMean );
  }
```

The `itk::ScalarImageKmeansImageFilter` is predefined for producing an 8 bits scalar image as output. This output image contains labels associated to each one of the classes in the K-Means algorithm. In the following lines we use the `OutputImageType` in order to instantiate the type of a `itk::ImageFileWriter`. Then create one, and connect it to the output of the classification filter.

```
typedef KMeansFilterType::OutputImageType  OutputImageType;

typedef itk::ImageFileWriter< OutputImageType > WriterType;

WriterType::Pointer writer = WriterType::New();
```

```
writer->SetInput( kmeansFilter->GetOutput() );

writer->SetFileName( outputImageFileName );
```

We are now ready for triggering the execution of the pipeline. This is done by simply invoking the `Update()` method in the writer. This call will propagate the update request to the reader and then to the classifier.

```
try
  {
  writer->Update();
  }
catch( itk::ExceptionObject & excp )
  {
  std::cerr << "Problem encountered while writing image file : " << argv[2] << std::endl;
  std::cerr << excp << std::endl;
  return EXIT_FAILURE;
  }
```

At this point the classification is done, the labeled image is saved in a file, and we can take a look at the means that were found as a result of the model estimation performed inside the classifier filter.

```
KMeansFilterType::ParametersType estimatedMeans = kmeansFilter->GetFinalMeans();

const unsigned int numberOfClasses = estimatedMeans.Size();

for ( unsigned int i = 0 ; i < numberOfClasses ; ++i )
  {
  std::cout << "cluster[" << i << "] ";
  std::cout << "    estimated mean : " << estimatedMeans[i] << std::endl;
  }
```

Figure 10.6 illustrates the effect of this filter with three classes. The means were estimated by ScalarImageKmeansModelEstimator.cxx.

10.4.3 Bayesian Plug-In Classifier

The source code for this section can be found in the file
`Examples/Statistics/BayesianPluginClassifier.cxx`.

In this example, we present a system that places measurement vectors into two Gaussian classes. The Figure 10.7 shows all the components of the classifier system and the data flow. This system differs with the previous k-means clustering algorithms in several ways. The biggest difference

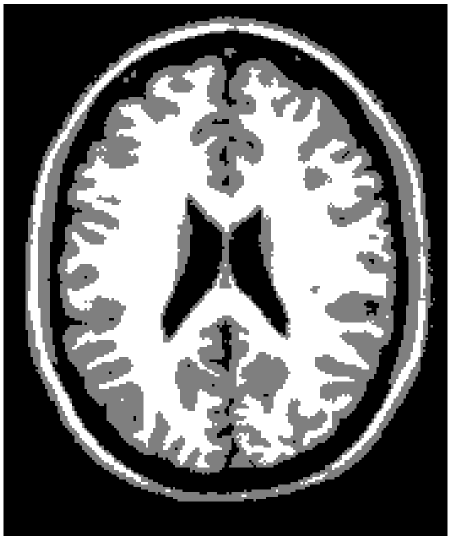


Figure 10.6: Effect of the KMeans classifier on a T1 slice of the brain.

is that this classifier uses the `itk::Statistics::GaussianDensityFunction`s as membership functions instead of the `itk::Statistics::EuclideanDistance`. Since the membership function is different, the membership function requires a different set of parameters, mean vectors and covariance matrices. We choose the `itk::Statistics::MeanCalculator` (sample mean) and the `itk::Statistics::CovarianceCalculator` (sample covariance) for the estimation algorithms of the two parameters. If we want more robust estimation algorithm, we can replace these estimation algorithms with more alternatives without changing other components in the classifier system.

It is a bad idea to use the same sample for test and training (parameter estimation) of the parameters. However, for simplicity, in this example, we use a sample for test and training.

We use the `itk::Statistics::ListSample` as the sample (test and training). The `itk::Vector` is our measurement vector class. To store measurement vectors into two separate sample containers, we use the `itk::Statistics::Subsample` objects.

```
#include "itkVector.h"
#include "itkListSample.h"
#include "itkSubsample.h"
```

The following two files provides us the parameter estimation algorithms.

```
#include "itkMeanCalculator.h"
#include "itkCovarianceCalculator.h"
```

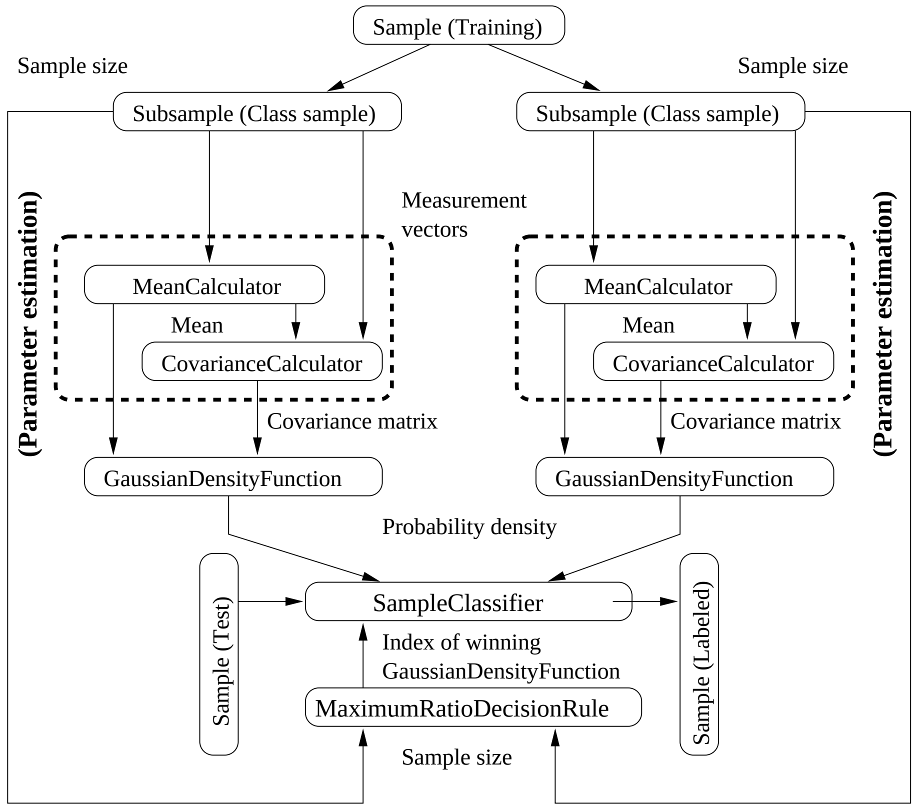


Figure 10.7: Bayesian plug-in classifier for two Gaussian classes.

The following files define the components required by ITK statistical classification framework: the decision rule, the membership function, and the classifier.

```
#include "itkMaximumRatioDecisionRule.h"
#include "itkGaussianDensityFunction.h"
#include "itkSampleClassifier.h"
```

We will fill the sample with random variables from two normal distribution using the `itk::Statistics::NormalVariateGenerator`.

```
#include "itkNormalVariateGenerator.h"
```

Since the NormalVariateGenerator class only supports 1-D, we define our measurement vector type as a one component vector. We then, create a ListSample object for data inputs.

We also create two Subsample objects that will store the measurement vectors in `sample` into two separate sample containers. Each Subsample object stores only the measurement vectors belonging to a single class. This class sample will be used by the parameter estimation algorithms.

```
  typedef itk::Vector< double, 1 > MeasurementVectorType;
  typedef itk::Statistics::ListSample< MeasurementVectorType > SampleType;
  SampleType::Pointer sample = SampleType::New();
  sample->SetMeasurementVectorSize( 1 ); // length of measurement vectors
                                         // in the sample.

  typedef itk::Statistics::Subsample< SampleType > ClassSampleType;
  std::vector< ClassSampleType::Pointer > classSamples;
  for ( unsigned int i = 0 ; i < 2 ; ++i )
    {
    classSamples.push_back( ClassSampleType::New() );
    classSamples[i]->SetSample( sample );
    }
```

The following code snippet creates a NormalVariateGenerator object. Since the random variable generator returns values according to the standard normal distribution (the mean is zero, and the standard deviation is one) before pushing random values into the `sample`, we change the mean and standard deviation. We want two normal (Gaussian) distribution data. We have two for loops. Each for loop uses different mean and standard deviation. Before we fill the `sample` with the second distribution data, we call `Initialize(random seed)` method, to recreate the pool of random variables in the `normalGenerator`. In the second for loop, we fill the two class samples with measurement vectors using the `AddInstance()` method.

To see the probability density plots from the two distributions, refer to Figure 10.5.

```
typedef itk::Statistics::NormalVariateGenerator NormalGeneratorType;
NormalGeneratorType::Pointer normalGenerator = NormalGeneratorType::New();

normalGenerator->Initialize( 101 );

MeasurementVectorType mv;
double mean = 100;
double standardDeviation = 30;
SampleType::InstanceIdentifier id = 0UL;
for ( unsigned int i = 0 ; i < 100 ; ++i )
  {
  mv.Fill( (normalGenerator->GetVariate() * standardDeviation ) + mean);
  sample->PushBack( mv );
  classSamples[0]->AddInstance( id );
  ++id;
  }

normalGenerator->Initialize( 3024 );
mean = 200;
standardDeviation = 30;
for ( unsigned int i = 0 ; i < 100 ; ++i )
  {
  mv.Fill( (normalGenerator->GetVariate() * standardDeviation ) + mean);
  sample->PushBack( mv );
  classSamples[1]->AddInstance( id );
  ++id;
  }
```

In the following code snippet, notice that the template argument for the MeanCalculator and CovarianceCalculator is `ClassSampleType` (i.e., type of Subsample) instead of SampleType (i.e., type of ListSample). This is because the parameter estimation algorithms are applied to the class sample.

```
typedef itk::Statistics::MeanCalculator< ClassSampleType > MeanEstimatorType;
typedef itk::Statistics::CovarianceCalculator< ClassSampleType >
  CovarianceEstimatorType;

std::vector< MeanEstimatorType::Pointer > meanEstimators;
std::vector< CovarianceEstimatorType::Pointer > covarianceEstimators;

for ( unsigned int i = 0 ; i < 2 ; ++i )
  {
  meanEstimators.push_back( MeanEstimatorType::New() );
  meanEstimators[i]->SetInputSample( classSamples[i] );
  meanEstimators[i]->Update();

  covarianceEstimators.push_back( CovarianceEstimatorType::New() );
```

```
    covarianceEstimators[i]->SetInputSample( classSamples[i] );
    covarianceEstimators[i]->SetMean( meanEstimators[i]->GetOutput() );
    covarianceEstimators[i]->Update();
    }
```

We print out the estimated parameters.

```
  for ( unsigned int i = 0 ; i < 2 ; ++i )
    {
    std::cout << "class[" << i << "] " << std::endl;
    std::cout << "    estimated mean : "
              << *(meanEstimators[i]->GetOutput())
              << "    covariance matrix : "
              << *(covarianceEstimators[i]->GetOutput()) << std::endl;
    }
```

After creating a SampleClassifier object and a MaximumRatioDecisionRule object, we plug in the `decisionRule` and the `sample` to the classifier. Then, we specify the number of classes that will be considered using the `SetNumberOfClasses()` method.

The MaximumRatioDecisionRule requires a vector of *a priori* probability values. Such *a priori* probability will be the $P(\omega_i)$ of the following variation of the Bayes decision rule:

$$\text{Decide } \omega_i \text{ if } \frac{p(\overrightarrow{x}|\omega_i)}{p(\overrightarrow{x}|\omega_j)} > \frac{P(\omega_j)}{P(\omega_i)} \text{ for all } j \neq i \qquad (10.4)$$

The remainder of the code snippet shows how to use user-specified class labels. The classification result will be stored in a MembershipSample object, and for each measurement vector, its class label will be one of the two class labels, 100 and 200 (`unsigned int`).

```
  typedef itk::Statistics::GaussianDensityFunction< MeasurementVectorType >
    MembershipFunctionType;
  typedef itk::MaximumRatioDecisionRule DecisionRuleType;
  DecisionRuleType::Pointer decisionRule = DecisionRuleType::New();

  DecisionRuleType::APrioriVectorType aPrioris;
  aPrioris.push_back( classSamples[0]->GetTotalFrequency()
                      / sample->GetTotalFrequency() ) ;
  aPrioris.push_back( classSamples[1]->GetTotalFrequency()
                      / sample->GetTotalFrequency() ) ;
  decisionRule->SetAPriori( aPrioris );

  typedef itk::Statistics::SampleClassifier< SampleType > ClassifierType;
  ClassifierType::Pointer classifier = ClassifierType::New();

  classifier->SetDecisionRule( (itk::DecisionRuleBase::Pointer) decisionRule);
```

```
classifier->SetSample( sample );
classifier->SetNumberOfClasses( 2 );

std::vector< unsigned int > classLabels;
classLabels.resize( 2 );
classLabels[0] = 100;
classLabels[1] = 200;
classifier->SetMembershipFunctionClassLabels(classLabels);
```

The `classifier` is almost ready to perform the classification except that it needs two membership functions that represent the two clusters.

In this example, we can imagine that the two clusters are modeled by two Euclidean distance functions. The distance function (model) has only one parameter, the mean (centroid) set by the `SetOrigin()` method. To plug-in two distance functions, we call the `AddMembershipFunction()` method. Then invocation of the `Update()` method will perform the classification.

```
std::vector< MembershipFunctionType::Pointer > membershipFunctions;
for ( unsigned int i = 0 ; i < 2 ; i++ )
  {
  membershipFunctions.push_back(MembershipFunctionType::New());
  membershipFunctions[i]->SetMean( meanEstimators[i]->GetOutput() );
  membershipFunctions[i]->
    SetCovariance( covarianceEstimators[i]->GetOutput() );
  classifier->AddMembershipFunction(membershipFunctions[i].GetPointer());
  }

classifier->Update();
```

The following code snippet prints out pairs of a measurement vector and its class label in the `sample`.

```
ClassifierType::OutputType* membershipSample = classifier->GetOutput();
ClassifierType::OutputType::ConstIterator iter = membershipSample->Begin();

while ( iter != membershipSample->End() )
  {
  std::cout << "measurement vector = " << iter.GetMeasurementVector()
            << "class label = " << iter.GetClassLabel() << std::endl;
  ++iter;
  }
```

10.4.4 Expectation Maximization Mixture Model Estimation

The source code for this section can be found in the file
`Examples/Statistics/ExpectationMaximizationMixtureModelEstimator.cxx`.

In this example, we present an implementation of the expectation maximization (EM) process to generates parameter estimates for a two Gaussian component mixture model.

The Bayesian plug-in classifier example (see Section 10.4.3) used two Gaussian probability density functions (PDF) to model two Gaussian distribution classes (two models for two class). However, in some cases, we want to model a distribution as a mixture of several different distributions. Therefore, the probability density function ($p(x)$) of a mixture model can be stated as follows :

$$p(x) = \sum_{i=0}^{c} \alpha_i f_i(x) \tag{10.5}$$

where i is the index of the component, c is the number of components, α_i is the proportion of the component, and f_i is the probability density function of the component.

Now the task is to find the parameters(the component PDF's parameters and the proportion values) to maximize the likelihood of the parameters. If we know which component a measurement vector belongs to, the solutions to this problem is easy to solve. However, we don't know the membership of each measurement vector. Therefore, we use the expectation of membership instead of the exact membership. The EM process splits into two steps:

1. E step: calculate the expected membership values for each measurement vector to each classes.

2. M step: find the next parameter sets that maximize the likelihood with the expected membership values and the current set of parameters.

The E step is basically a step that calculates the *a posteriori* probability for each measurement vector.

The M step is dependent on the type of each PDF. Most of distributions belonging to exponential family such as Poisson, Binomial, Exponential, and Normal distributions have analytical solutions for updating the parameter set. The `itk::Statistics::ExpectationMaximizationMixtureModelEstimator` class assumes that such type of components.

In the following example we use the `itk::Statistics::ListSample` as the sample (test and training). The `itk::Vector::i`s our measurement vector class. To store measurement vectors into two separate sample container, we use the `itk::Statistics::Subsample` objects.

```
#include "itkVector.h"
#include "itkListSample.h"
```

The following two files provides us the parameter estimation algorithms.

```
#include "itkGaussianMixtureModelComponent.h"
#include "itkExpectationMaximizationMixtureModelEstimator.h"
```

We will fill the sample with random variables from two normal distribution using the `itk::Statistics::NormalVariateGenerator`.

```
#include "itkNormalVariateGenerator.h"
```

Since the NormalVariateGenerator class only supports 1-D, we define our measurement vector type as a one component vector. We then, create a ListSample object for data inputs.

We also create two Subsample objects that will store the measurement vectors in the `sample` into two separate sample containers. Each Subsample object stores only the measurement vectors belonging to a single class. This *class sample* will be used by the parameter estimation algorithms.

```
  unsigned int numberOfClasses = 2;
  typedef itk::Vector< double, 1 > MeasurementVectorType;
  typedef itk::Statistics::ListSample< MeasurementVectorType > SampleType;
  SampleType::Pointer sample = SampleType::New();
  sample->SetMeasurementVectorSize( 1 ); // length of measurement vectors
                                         // in the sample.
```

The following code snippet creates a NormalVariateGenerator object. Since the random variable generator returns values according to the standard normal distribution (the mean is zero, and the standard deviation is one) before pushing random values into the `sample`, we change the mean and standard deviation. We want two normal (Gaussian) distribution data. We have two for loops. Each for loop uses different mean and standard deviation. Before we fill the `sample` with the second distribution data, we call `Initialize()` method to recreate the pool of random variables in the `normalGenerator`. In the second for loop, we fill the two class samples with measurement vectors using the `AddInstance()` method.

To see the probability density plots from the two distribution, refer to Figure 10.5.

```
  typedef itk::Statistics::NormalVariateGenerator NormalGeneratorType;
  NormalGeneratorType::Pointer normalGenerator = NormalGeneratorType::New();

  normalGenerator->Initialize( 101 );

  MeasurementVectorType mv;
  double mean = 100;
  double standardDeviation = 30;
  for ( unsigned int i = 0 ; i < 100 ; ++i )
```

```
    {
    mv[0] = ( normalGenerator->GetVariate() * standardDeviation ) + mean;
    sample->PushBack( mv );
    }

  normalGenerator->Initialize( 3024 );
  mean = 200;
  standardDeviation = 30;
  for ( unsigned int i = 0 ; i < 100 ; ++i )
    {
    mv[0] = ( normalGenerator->GetVariate() * standardDeviation ) + mean;
    sample->PushBack( mv );
    }
```

In the following code snippet notice that the template argument for the MeanCalculator and CovarianceCalculator is `ClassSampleType` (i.e., type of Subsample) instead of `SampleType` (i.e., type of ListSample). This is because the parameter estimation algorithms are applied to the class sample.

```
  typedef itk::Array< double > ParametersType;
  ParametersType params( 2 );

  std::vector< ParametersType > initialParameters( numberOfClasses );
  params[0] = 110.0;
  params[1] = 800.0;
  initialParameters[0] = params;

  params[0] = 210.0;
  params[1] = 850.0;
  initialParameters[1] = params;

  typedef itk::Statistics::GaussianMixtureModelComponent< SampleType >
    ComponentType;

  std::vector< ComponentType::Pointer > components;
  for ( unsigned int i = 0 ; i < numberOfClasses ; i++ )
    {
    components.push_back( ComponentType::New() );
    (components[i])->SetSample( sample );
    (components[i])->SetParameters( initialParameters[i] );
    }
```

We run the estimator.

```
  typedef itk::Statistics::ExpectationMaximizationMixtureModelEstimator<
                           SampleType > EstimatorType;
  EstimatorType::Pointer estimator = EstimatorType::New();
```

```
estimator->SetSample( sample );
estimator->SetMaximumIteration( 200 );

itk::Array< double > initialProportions(numberOfClasses);
initialProportions[0] = 0.5;
initialProportions[1] = 0.5;

estimator->SetInitialProportions( initialProportions );

for ( unsigned int i = 0 ; i < numberOfClasses ; i++)
  {
  estimator->AddComponent( (ComponentType::Superclass*)
                           (components[i]).GetPointer() );
  }

estimator->Update();
```

We then print out the estimated parameters.

```
for ( unsigned int i = 0 ; i < numberOfClasses ; i++ )
  {
  std::cout << "Cluster[" << i << "]" << std::endl;
  std::cout << "    Parameters:" << std::endl;
  std::cout << "         " << (components[i])->GetFullParameters()
            << std::endl;
  std::cout << "    Proportion: ";
  std::cout << "         " << (*estimator->GetProportions())[i] << std::endl;
  }
```

10.4.5 Classification using Markov Random Field

Markov Random Fields are probabilistic models that use the correlation between pixels in a neighborhood to decide the object region. The `itk::Statistics::MRFImageFilter` uses the maximum a posteriori (MAP) estimates for modeling the MRF. The object traverses the data set and uses the model generated by the Mahalanobis distance classifier to gets the the distance between each pixel in the data set to a set of known classes, updates the distances by evaluating the influence of its neighboring pixels (based on a MRF model) and finally, classifies each pixel to the class which has the minimum distance to that pixel (taking the neighborhood influence under consideration). The energy function minimization is done using the iterated conditional modes (ICM) algorithm [9].

The source code for this section can be found in the file
`Examples/Statistics/ScalarImageMarkovRandomField1.cxx`.

This example shows how to use the Markov Random Field approach for classifying the pixel of a scalar image.

The `itk::Statistics::MRFImageFilter` is used for refining an initial classification by introducing the spatial coherence of the labels. The user should provide two images as input. The first image is the one to be classified while the second image is an image of labels representing an initial classification.

The following headers are related to reading input images, writing the output image, and making the necessary conversions between scalar and vector images.

```
#include "itkImage.h"
#include "itkFixedArray.h"
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
#include "itkScalarToArrayCastImageFilter.h"
```

The following headers are related to the statistical classification classes.

```
#include "itkMRFImageFilter.h"
#include "itkDistanceToCentroidMembershipFunction.h"
#include "itkMinimumDecisionRule.h"
#include "itkImageClassifierBase.h"
```

First we define the pixel type and dimension of the image that we intend to classify. With this image type we can also declare the `itk::ImageFileReader` needed for reading the input image, create one and set its input filename. In this particular case we choose to use `signed short` as pixel type, which is typical for MicroMRI and CT data sets.

```
  typedef signed short        PixelType;
  const unsigned int          Dimension = 2;

  typedef itk::Image<PixelType, Dimension > ImageType;

  typedef itk::ImageFileReader< ImageType > ReaderType;
  ReaderType::Pointer reader = ReaderType::New();
  reader->SetFileName( inputImageFileName );
```

As a second step we define the pixel type and dimension of the image of labels that provides the initial classification of the pixels from the first image. This initial labeled image can be the output of a K-Means method like the one illustrated in section 10.4.2.

```
  typedef unsigned char       LabelPixelType;

  typedef itk::Image<LabelPixelType, Dimension > LabelImageType;

  typedef itk::ImageFileReader< LabelImageType > LabelReaderType;
  LabelReaderType::Pointer labelReader = LabelReaderType::New();
  labelReader->SetFileName( inputLabelImageFileName );
```

Since the Markov Random Field algorithm is defined in general for images whose pixels have multiple components, that is, images of vector type, we must adapt our scalar image in order to satisfy the interface expected by the MRFImageFilter. We do this by using the itk::ScalarToArrayCastImageFilter. With this filter we will present our scalar image as a vector image whose vector pixels contain a single component.

```
typedef itk::FixedArray<LabelPixelType,1>  ArrayPixelType;

typedef itk::Image< ArrayPixelType, Dimension > ArrayImageType;

typedef itk::ScalarToArrayCastImageFilter<
                   ImageType, ArrayImageType > ScalarToArrayFilterType;

ScalarToArrayFilterType::Pointer
  scalarToArrayFilter = ScalarToArrayFilterType::New();
scalarToArrayFilter->SetInput( reader->GetOutput() );
```

With the input image type ImageType and labeled image type LabelImageType we instantiate the type of the itk::MRFImageFilter that will apply the Markov Random Field algorithm in order to refine the pixel classification.

```
typedef itk::MRFImageFilter< ArrayImageType, LabelImageType > MRFFilterType;

MRFFilterType::Pointer mrfFilter = MRFFilterType::New();

mrfFilter->SetInput( scalarToArrayFilter->GetOutput() );
```

We set now some of the parameters for the MRF filter. In particular, the number of classes to be used during the classification, the maximum number of iterations to be run in this filter and the error tolerance that will be used as a criterion for convergence.

```
mrfFilter->SetNumberOfClasses( numberOfClasses );
mrfFilter->SetMaximumNumberOfIterations( numberOfIterations );
mrfFilter->SetErrorTolerance( 1e-7 );
```

The smoothing factor represents the tradeoff between fidelity to the observed image and the smoothness of the segmented image. Typical smoothing factors have values between 1 5. This factor will multiply the weights that define the influence of neighbors on the classification of a given pixel. The higher the value, the more uniform will be the regions resulting from the classification refinement.

```
mrfFilter->SetSmoothingFactor( smoothingFactor );
```

Given that the MRF filter need to continually relabel the pixels, it needs access to a set of membership functions that will measure to what degree every pixel belongs to a particular class.

The classification is performed by the `itk::ImageClassifierBase` class, that is instantiated using the type of the input vector image and the type of the labeled image.

```
typedef itk::ImageClassifierBase<
                            ArrayImageType,
                            LabelImageType >   SupervisedClassifierType;

SupervisedClassifierType::Pointer classifier =
                                       SupervisedClassifierType::New();
```

The classifier need a decision rule to be set by the user. Note that we must use `GetPointer()` in the call of the `SetDecisionRule()` method because we are passing a SmartPointer, and smart pointer cannot perform polymorphism, we must then extract the raw pointer that is associated to the smart pointer. This extraction is done with the GetPointer() method.

```
typedef itk::MinimumDecisionRule DecisionRuleType;

DecisionRuleType::Pointer  classifierDecisionRule = DecisionRuleType::New();

classifier->SetDecisionRule( classifierDecisionRule.GetPointer() );
```

We now instantiate the membership functions. In this case we use the `itk::Statistics::DistanceToCentroidMembershipFunction` class templated over the pixel type of the vector image, that in our example happens to be a vector of dimension 1.

```
typedef itk::Statistics::DistanceToCentroidMembershipFunction<
                                                  ArrayPixelType >
                                                     MembershipFunctionType;

typedef MembershipFunctionType::Pointer MembershipFunctionPointer;


double meanDistance = 0;
vnl_vector<double> centroid(1);
for( unsigned int i=0; i < numberOfClasses; i++ )
  {
  MembershipFunctionPointer membershipFunction =
                                     MembershipFunctionType::New();

  centroid[0] = atof( argv[i+numberOfArgumentsBeforeMeans] );

  membershipFunction->SetCentroid( centroid );

  classifier->AddMembershipFunction( membershipFunction );
  meanDistance += static_cast< double > (centroid[0]);
  }
meanDistance /= numberOfClasses;
```

We set the Smoothing factor. This factor will multiply the weights that define the influence of neighbors on the classification of a given pixel. The higher the value, the more uniform will be the regions resulting from the classification refinement.

```
mrfFilter->SetSmoothingFactor( smoothingFactor );
```

and we set the neighborhood radius that will define the size of the clique to be used in the computation of the neighbors' influence in the classification of any given pixel. Note that despite the fact that we call this a radius, it is actually the half size of an hypercube. That is, the actual region of influence will not be circular but rather an N-Dimensional box. For example, a neighborhood radius of 2 in a 3D image will result in a clique of size 5x5x5 pixels, and a radius of 1 will result in a clique of size 3x3x3 pixels.

```
mrfFilter->SetNeighborhoodRadius( 1 );
```

We should now set the weights used for the neighbors. This is done by passing an array of values that contains the linear sequence of weights for the neighbors. For example, in a neighborhood of size 3x3x3, we should provide a linear array of 9 weight values. The values are packaged in a `std::vector` and are supposed to be `double`. The following lines illustrate a typical set of values for a 3x3x3 neighborhood. The array is arranged and then passed to the filter by using the method `SetMRFNeighborhoodWeight()`.

```
std::vector< double > weights;
weights.push_back(1.5);
weights.push_back(2.0);
weights.push_back(1.5);
weights.push_back(2.0);
weights.push_back(0.0); // This is the central pixel
weights.push_back(2.0);
weights.push_back(1.5);
weights.push_back(2.0);
weights.push_back(1.5);
```

We now scale weights so that the smoothing function and the image fidelity functions have comparable value. This is necessary since the label image and the input image can have different dynamic ranges. The fidelity function is usually computed using a distance function, such as the `itk::DistanceToCentroidMembershipFunction` or one of the other membership functions. They tend to have values in the order of the means specified.

```
double totalWeight = 0;
for(std::vector< double >::const_iterator wcIt = weights.begin();
    wcIt != weights.end(); ++wcIt )
  {
  totalWeight += *wcIt;
  }
for(std::vector< double >::iterator wIt = weights.begin();
    wIt != weights.end(); wIt++ )
  {
  *wIt = static_cast< double > ( (*wIt) * meanDistance / (2 * totalWeight));
  }

mrfFilter->SetMRFNeighborhoodWeight( weights );
```

Finally, the classifier class is connected to the Markof Random Fields filter.

```
  mrfFilter->SetClassifier( classifier );
```

The output image produced by the `itk::MRFImageFilter` has the same pixel type as the labeled input image. In the following lines we use the `OutputImageType` in order to instantiate the type of a `itk::ImageFileWriter`. Then create one, and connect it to the output of the classification filter after passing it through an intensity rescaler to rescale it to an 8 bit dynamic range

```
  typedef MRFFilterType::OutputImageType  OutputImageType;

  typedef itk::ImageFileWriter< OutputImageType > WriterType;

  WriterType::Pointer writer = WriterType::New();

  writer->SetInput( intensityRescaler->GetOutput() );

  writer->SetFileName( outputImageFileName );
```

We are now ready for triggering the execution of the pipeline. This is done by simply invoking the `Update()` method in the writer. This call will propagate the update request to the reader and then to the MRF filter.

```
  try
    {
    writer->Update();
    }
  catch( itk::ExceptionObject & excp )
    {
    std::cerr << "Problem encountered while writing image file : " << argv[2] << std::endl;
    std::cerr << excp << std::endl;
    return EXIT_FAILURE;
    }
```

Figure 10.8 illustrates the effect of this filter with three classes. In this example the filter was run with a smoothing factor of 3. The labeled image was produced by ScalarImageKmeansClassifier.cxx and the means were estimated by ScalarImageKmeansModelEstimator.cxx.

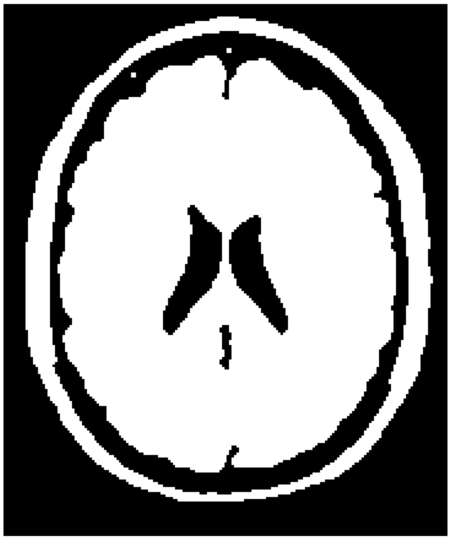


Figure 10.8: Effect of the MRF filter on a T1 slice of the brain.

Part III

Developer’s Guide

CHAPTER

ELEVEN

Iterators

This chapter introduces the *image iterator*, an important generic programming construct for image processing in ITK. An iterator is a generalization of the familiar C programming language pointer used to reference data in memory. ITK has a wide variety of image iterators, some of which are highly specialized to simplify common image processing tasks.

The next section is a brief introduction that defines iterators in the context of ITK. Section 11.2 describes the programming interface common to most ITK image iterators. Sections 11.3–11.4 document specific ITK iterator types and provide examples of how they are used.

11.1 Introduction

Generic programming models define functionally independent components called *containers* and *algorithms*. Container objects store data and algorithms operate on data. To access data in containers, algorithms use a third class of objects called *iterators*. An iterator is an abstraction of a memory pointer. Every container type must define its own iterator type, but all iterators are written to provide a common interface so that algorithm code can reference data in a generic way and maintain functional independence from containers.

The iterator is so named because it is used for *iterative*, sequential access of container values. Iterators appear in `for` and `while` loop constructs, visiting each data point in turn. A C pointer, for example, is a type of iterator. It can be moved forward (incremented) and backward (decremented) through memory to sequentially reference elements of an array. Many iterator implementations have an interface similar to a C pointer.

In ITK we use iterators to write generic image processing code for images instantiated with different combinations of pixel type, pixel container type, and dimensionality. Because ITK image iterators are specifically designed to work with *image* containers, their interface and implementation is optimized for image processing tasks. Using the ITK iterators instead of accessing data directly through the `itk::Image` interface has many advantages. Code is more compact and often generalizes automatically to higher dimensions, algorithms run much faster, and iterators simplify tasks such as multithreading and neighborhood-based image processing.

11.2 Programming Interface

This section describes the standard ITK image iterator programming interface. Some specialized image iterators may deviate from this standard or provide additional methods.

11.2.1 Creating Iterators

All image iterators have at least one template parameter that is the image type over which they iterate. There is no restriction on the dimensionality of the image or on the pixel type of the image.

An iterator constructor requires at least two arguments, a smart pointer to the image to iterate across, and an image region. The image region, called the *iteration region*, is a rectilinear area in which iteration is constrained. The iteration region must be wholly contained within the image. More specifically, a valid iteration region is any subregion of the image within the current `BufferedRegion`. See Section 4.1 for more information on image regions.

There is a const and a non-const version of most ITK image iterators. A non-const iterator cannot be instantiated on a non-const image pointer. Const versions of iterators may read, but may not write pixel values.

Here is a simple example that defines and constructs a simple image iterator for an `itk::Image`.

```
typedef itk::Image<float, 3> ImageType;
typedef itk::ImageRegionConstIterator< ImageType > ConstIteratorType;
typedef itk::ImageRegionIterator< ImageType > IteratorType;

ImageType::Pointer image = SomeFilter->GetOutput();

ConstIteratorType constIterator( image, image->GetRequestedRegion() );
IteratorType iterator( image, image->GetRequestedRegion() );
```

11.2.2 Moving Iterators

An iterator is described as *walking* its iteration region. At any time, the iterator will reference, or “point to”, one pixel location in the N-dimensional (ND) image. *Forward iteration* goes from the beginning of the iteration region to the end of the iteration region. *Reverse iteration*, goes from just past the end of the region back to the beginning. There are two corresponding starting positions for iterators, the *begin* position and the *end* position. An iterator can be moved directly to either of these two positions using the following methods.

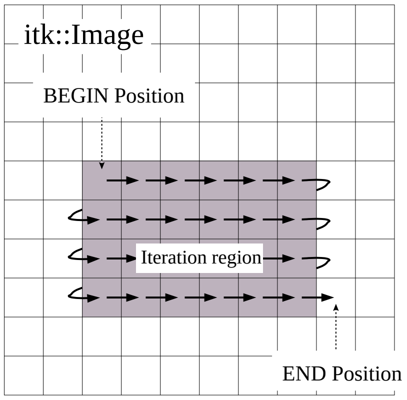


Figure 11.1: Normal path of an iterator through a 2D image. The iteration region is shown in a darker shade. An arrow denotes a single iterator step, the result of one ++ operation.

- **`GoToBegin()`** Points the iterator to the first valid data element in the region.
- **`GoToEnd()`** Points the iterator to *one position past* the last valid element in the region.

Note that the end position is not actually located within the iteration region. This is important to remember because attempting to dereference an iterator at its end position will have undefined results.

ITK iterators are moved back and forth across their iterations using the decrement and increment operators.

- **`operator++()`** Increments the iterator one position in the positive direction. Only the prefix increment operator is defined for ITK image iterators.
- **`operator--()`** Decrements the iterator one position in the negative direction. Only the prefix decrement operator is defined for ITK image iterators.

Figure 11.1 illustrates typical iteration over an image region. Most iterators increment and decrement in the direction of the fastest increasing image dimension, wrapping to the first position in the next higher dimension at region boundaries. In other words, an iterator first moves across columns, then down rows, then from slice to slice, and so on.

In addition to sequential iteration through the image, some iterators may define random access operators. Unlike the increment operators, random access operators may not be optimized for speed and require some knowledge of the dimensionality of the image and the extent of the iteration region to use properly.

- **`operator+=( OffsetType )`** Moves the iterator to the pixel position at the current index plus specified `itk::Offset`.

- **`operator-=( OffsetType )`** Moves the iterator to the pixel position at the current index minus specified Offset.
- **`SetPosition( IndexType )`** Moves the iterator to the given `itk::Index` position.

The `SetPosition()` method may be extremely slow for more complicated iterator types. In general, it should only be used for setting a starting iteration position, like you would use `GoToBegin()` or `GoToEnd()`.

Some iterators do not follow a predictable path through their iteration regions and have no fixed beginning or ending pixel locations. A conditional iterator, for example, visits pixels only if they have certain values or connectivities. Random iterators, increment and decrement to random locations and may even visit a given pixel location more than once.

An iterator can be queried to determine if it is at the end or the beginning of its iteration region.

- **`bool IsAtEnd()`** True if the iterator points to *one position past* the end of the iteration region.
- **`bool IsAtBegin()`** True if the iterator points to the first position in the iteration region. The method is typically used to test for the end of reverse iteration.

An iterator can also report its current image index position.

- **`IndexType GetIndex()`** Returns the Index of the image pixel that the iterator currently points to.

For efficiency, most ITK image iterators do not perform bounds checking. It is possible to move an iterator outside of its valid iteration region. Dereferencing an out-of-bounds iterator will produce undefined results.

11.2.3 Accessing Data

ITK image iterators define two basic methods for reading and writing pixel values.

- **`PixelType Get()`** Returns the value of the pixel at the iterator position.
- **`void Set( PixelType )`** Sets the value of the pixel at the iterator position. Not defined for const versions of iterators.

The `Get()` and `Set()` methods are inlined and optimized for speed so that their use is equivalent to dereferencing the image buffer directly. There are a few common cases, however, where using `Get()` and `Set()` do incur a penalty. Consider the following code, which fetches, modifies, and then writes a value back to the same pixel location.

```
it.Set( it.Get() + 1 );
```

As written, this code requires one more memory dereference than is necessary. Some iterators define a third data access method that avoids this penalty.

- **`PixelType &Value()`** Returns a reference to the pixel at the iterator position.

The `Value()` method can be used as either an lval or an rval in an expression. It has all the properties of `operator*`. The `Value()` method makes it possible to rewrite our example code more efficiently.

```
it.Value()++;
```

Consider using the `Value()` method instead of `Get()` or `Set()` when a call to `operator=` on a pixel is non-trivial, such as when working with vector pixels, and operations are done in-place in the image. The disadvantage of using `Value` is that it cannot support image adapters (see Section 12 on page 745 for more information about image adaptors).

11.2.4 Iteration Loops

Using the methods described in the previous sections, we can now write a simple example to do pixel-wise operations on an image. The following code calculates the squares of all values in an input image and writes them to an output image.

```
ConstIteratorType in( inputImage,   inputImage->GetRequestedRegion() );
IteratorType out( outputImage, inputImage->GetRequestedRegion() );

for ( in.GoToBegin(), out.GoToBegin(); !in.IsAtEnd(); ++in, ++out )
  {
  out.Set( in.Get() * in.Get() );
  }
```

Notice that both the input and output iterators are initialized over the same region, the `RequestedRegion` of `inputImage`. This is good practice because it ensures that the output iterator walks exactly the same set of pixel indices as the input iterator, but does not require that the output and input be the same size. The only requirement is that the input image must contain a region (a starting index and size) that matches the `RequestedRegion` of the output image.

Equivalent code can be written by iterating through the image in reverse. The syntax is slightly more awkward because the *end* of the iteration region is not a valid position and we can only test whether the iterator is strictly *equal* to its beginning position. It is often more convenient to write reverse iteration in a `while` loop.

```
  in.GoToEnd();
  out.GoToEnd();
  while ( ! in.IsAtBegin() )
    {
    --in;
    --out;
    out.Set( in.Get() * in.Get() );
    }
```

11.3 Image Iterators

This section describes iterators that walk rectilinear image regions and reference a single pixel at a time. The itk::ImageRegionIterator is the most basic ITK image iterator and the first choice for most applications. The rest of the iterators in this section are specializations of ImageRegionIterator that are designed make common image processing tasks more efficient or easier to implement.

11.3.1 ImageRegionIterator

The source code for this section can be found in the file
`Examples/Iterators/ImageRegionIterator.cxx`.

The itk::ImageRegionIterator is optimized for iteration speed and is the first choice for iterative, pixel-wise operations when location in the image is not important. ImageRegionIterator is the least specialized of the ITK image iterator classes. It implements all of the methods described in the preceding section.

The following example illustrates the use of itk::ImageRegionConstIterator and ImageRegionIterator. Most of the code constructs introduced apply to other ITK iterators as well. This simple application crops a subregion from an image by copying its pixel values into to a second, smaller image.

We begin by including the appropriate header files.

```
#include "itkImageRegionConstIterator.h"
#include "itkImageRegionIterator.h"
```

Next we define a pixel type and corresponding image type. ITK iterator classes expect the image type as their template parameter.

```
  const unsigned int Dimension = 2;

  typedef unsigned char PixelType;
```

```
typedef itk::Image< PixelType, Dimension >  ImageType;

typedef itk::ImageRegionConstIterator< ImageType > ConstIteratorType;
typedef itk::ImageRegionIterator< ImageType>       IteratorType;
```

Information about the subregion to copy is read from the command line. The subregion is defined by an `itk::ImageRegion` object, with a starting grid index and a size (Section 4.1).

```
ImageType::RegionType inputRegion;

ImageType::RegionType::IndexType inputStart;
ImageType::RegionType::SizeType  size;

inputStart[0] = ::atoi( argv[3] );
inputStart[1] = ::atoi( argv[4] );

size[0]  = ::atoi( argv[5] );
size[1]  = ::atoi( argv[6] );

inputRegion.SetSize( size );
inputRegion.SetIndex( inputStart );
```

The destination region in the output image is defined using the input region size, but a different start index. The starting index for the destination region is the corner of the newly generated image.

```
ImageType::RegionType outputRegion;

ImageType::RegionType::IndexType outputStart;

outputStart[0] = 0;
outputStart[1] = 0;

outputRegion.SetSize( size );
outputRegion.SetIndex( outputStart );
```

After reading the input image and checking that the desired subregion is, in fact, contained in the input, we allocate an output image. It is fundamental to set valid values to some of the basic image information during the copying process. In particular, the starting index of the output region is now filled up with zero values and the coordinates of the physical origin are computed as a shift from the origin of the input image. This is quite important since it will allow us to later register the extracted region against the original image.

```
ImageType::Pointer outputImage = ImageType::New();
outputImage->SetRegions( outputRegion );
```

```
const ImageType::SpacingType& spacing = reader->GetOutput()->GetSpacing();
const ImageType::PointType& inputOrigin = reader->GetOutput()->GetOrigin();
double   outputOrigin[ Dimension ];

for(unsigned int i=0; i< Dimension; i++)
  {
  outputOrigin[i] = inputOrigin[i] + spacing[i] * inputStart[i];
  }

outputImage->SetSpacing( spacing );
outputImage->SetOrigin(  outputOrigin );
outputImage->Allocate();
```

The necessary images and region definitions are now in place. All that is left to do is to create the iterators and perform the copy. Note that image iterators are not accessed via smart pointers so they are light-weight objects that are instantiated on the stack. Also notice how the input and output iterators are defined over the *same corresponding region*. Though the images are different sizes, they both contain the same target subregion.

```
ConstIteratorType inputIt(   reader->GetOutput(), inputRegion  );
IteratorType      outputIt(  outputImage,         outputRegion );

for ( inputIt.GoToBegin(), outputIt.GoToBegin(); !inputIt.IsAtEnd();
      ++inputIt, ++outputIt)
  {
  outputIt.Set(  inputIt.Get()  );
  }
```

The `for` loop above is a common construct in ITK. The beauty of these four lines of code is that they are equally valid for one, two, three, or even ten dimensional data, and no knowledge of the size of the image is necessary. Consider the ugly alternative of ten nested `for` loops for traversing an image.

Let's run this example on the image `FatMRISlice.png` found in `Examples/Data`. The command line arguments specify the input and output file names, then the *x*, *y* origin and the *x*, *y* size of the cropped subregion.

```
ImageRegionIterator FatMRISlice.png ImageRegionIteratorOutput.png 20 70 210 140
```

The output is the cropped subregion shown in Figure 11.2.

11.3.2 ImageRegionIteratorWithIndex

The source code for this section can be found in the file
`Examples/Iterators/ImageRegionIteratorWithIndex.cxx`.

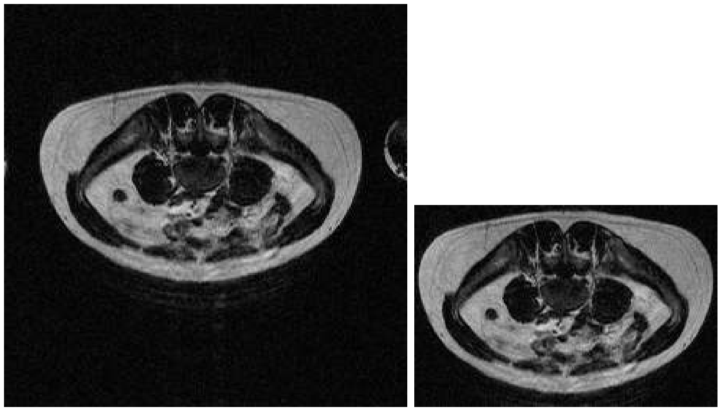


Figure 11.2: Cropping a region from an image. The original image is shown at left. The image on the right is the result of applying the ImageRegionIterator example code.

The "WithIndex" family of iterators was designed for algorithms that use both the value and the location of image pixels in calculations. Unlike `itk::ImageRegionIterator`, which calculates an index only when asked for, `itk::ImageRegionIteratorWithIndex` maintains its index location as a member variable that is updated during the increment or decrement process. Iteration speed is penalized, but the index queries are more efficient.

The following example illustrates the use of ImageRegionIteratorWithIndex. The algorithm mirrors a 2D image across its *x*-axis (see `itk::FlipImageFilter` for an ND version). The algorithm makes extensive use of the `GetIndex()` method.

We start by including the proper header file.

```
#include "itkImageRegionIteratorWithIndex.h"
```

For this example, we will use an RGB pixel type so that we can process color images. Like most other ITK image iterator, ImageRegionIteratorWithIndex class expects the image type as its single template parameter.

```
  const unsigned int Dimension = 2;

  typedef itk::RGBPixel< unsigned char > RGBPixelType;
  typedef itk::Image< RGBPixelType, Dimension >  ImageType;

  typedef itk::ImageRegionIteratorWithIndex< ImageType > IteratorType;
```

An `ImageType` smart pointer called `inputImage` points to the output of the image reader. After updating the image reader, we can allocate an output image of the same size, spacing, and origin as the input image.

```
ImageType::Pointer outputImage = ImageType::New();
outputImage->SetRegions( inputImage->GetRequestedRegion() );
outputImage->CopyInformation( inputImage );
outputImage->Allocate();
```

Next we create the iterator that walks the output image. This algorithm requires no iterator for the input image.

```
IteratorType outputIt( outputImage, outputImage->GetRequestedRegion() );
```

This axis flipping algorithm works by iterating through the output image, querying the iterator for its index, and copying the value from the input at an index mirrored across the *x*-axis.

```
ImageType::IndexType requestedIndex =
              outputImage->GetRequestedRegion().GetIndex();
ImageType::SizeType requestedSize =
              outputImage->GetRequestedRegion().GetSize();

for ( outputIt.GoToBegin(); !outputIt.IsAtEnd(); ++outputIt)
  {
  ImageType::IndexType idx = outputIt.GetIndex();
  idx[0] =  requestedIndex[0] + requestedSize[0] - 1 - idx[0];
  outputIt.Set( inputImage->GetPixel(idx) );
  }
```

Let's run this example on the image `VisibleWomanEyeSlice.png` found in the `Examples/Data` directory. Figure 11.3 shows how the original image has been mirrored across its *x*-axis in the output.

11.3.3 ImageLinearIteratorWithIndex

The source code for this section can be found in the file
`Examples/Iterators/ImageLinearIteratorWithIndex.cxx`.

The `itk::ImageLinearIteratorWithIndex` is designed for line-by-line processing of an image. It walks a linear path along a selected image direction parallel to one of the coordinate axes of the image. This iterator conceptually breaks an image into a set of parallel lines that span the selected image dimension.

Like all image iterators, movement of the ImageLinearIteratorWithIndex is constrained within an image region R. The line ℓ through which the iterator moves is defined by selecting a direction and an origin. The line ℓ extends from the origin to the upper boundary of R. The origin can be moved to any position along the lower boundary of R.

Several additional methods are defined for this iterator to control movement of the iterator along the line ℓ and movement of the origin of ℓ.

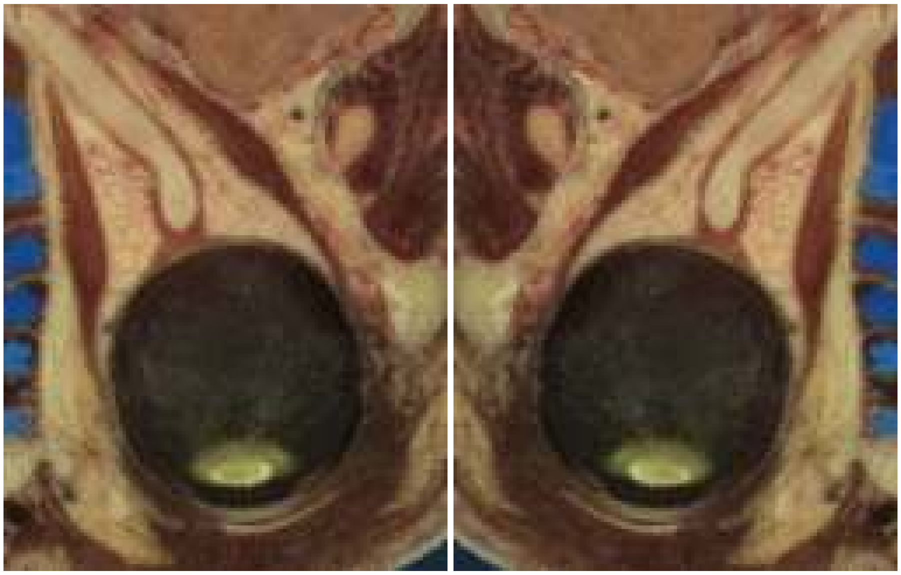


Figure 11.3: Results of using ImageRegionIteratorWithIndex to mirror an image across an axis. The original image is shown at left. The mirrored output is shown at right.

- **`NextLine()`** Moves the iterator to the beginning pixel location of the next line in the image. The origin of the next line is determined by incrementing the current origin along the fastest increasing dimension of the subspace of the image that excludes the selected dimension.
- **`PreviousLine()`** Moves the iterator to the *last valid pixel location* in the previous line. The origin of the previous line is determined by decrementing the current origin along the fastest increasing dimension of the subspace of the image that excludes the selected dimension.
- **`GoToBeginOfLine()`** Moves the iterator to the beginning pixel of the current line.
- **`GoToEndOfLine()`** Move the iterator to *one past* the last valid pixel of the current line.
- **`IsAtReverseEndOfLine()`** Returns true if the iterator points to *one position before* the beginning pixel of the current line.
- **`IsAtEndOfLine()`** Returns true if the iterator points to *one position past* the last valid pixel of the current line.

The following code example shows how to use the ImageLinearIteratorWithIndex. It implements the same algorithm as in the previous example, flipping an image across its *x*-axis. Two line iterators are iterated in opposite directions across the *x*-axis. After each line is traversed, the iterator origins are stepped along the *y*-axis to the next line.

Headers for both the const and non-const versions are needed.

```
#include "itkImageLinearConstIteratorWithIndex.h"
#include "itkImageLinearIteratorWithIndex.h"
```

The RGB image and pixel types are defined as in the previous example. The ImageLinearIteratorWithIndex class and its const version each have single template parameters, the image type.

```
  typedef itk::ImageLinearIteratorWithIndex< ImageType >       IteratorType;
  typedef itk::ImageLinearConstIteratorWithIndex< ImageType >  ConstIteratorType;
```

After reading the input image, we allocate an output image that of the same size, spacing, and origin.

```
  ImageType::Pointer outputImage = ImageType::New();
  outputImage->SetRegions( inputImage->GetRequestedRegion() );
  outputImage->CopyInformation( inputImage );
  outputImage->Allocate();
```

Next we create the two iterators. The const iterator walks the input image, and the non-const iterator walks the output image. The iterators are initialized over the same region. The direction of iteration is set to 0, the *x* dimension.

```
  ConstIteratorType inputIt( inputImage, inputImage->GetRequestedRegion() );
  IteratorType outputIt( outputImage, inputImage->GetRequestedRegion() );

  inputIt.SetDirection(0);
  outputIt.SetDirection(0);
```

Each line in the input is copied to the output. The input iterator moves forward across columns while the output iterator moves backwards.

```
  for ( inputIt.GoToBegin(),  outputIt.GoToBegin(); ! inputIt.IsAtEnd();
        outputIt.NextLine(),  inputIt.NextLine())
    {
    inputIt.GoToBeginOfLine();
    outputIt.GoToEndOfLine();
    --outputIt;
    while ( ! inputIt.IsAtEndOfLine() )
      {
      outputIt.Set( inputIt.Get() );
      ++inputIt;
      --outputIt;
      }
    }
```

Running this example on `VisibleWomanEyeSlice.png` produces the same output image shown in Figure 11.3.

The source code for this section can be found in the file
`Examples/Iterators/ImageLinearIteratorWithIndex2.cxx`.

This example shows how to use the `itk::ImageLinearIteratorWithIndex` for computing the mean across time of a 4D image where the first three dimensions correspond to spatial coordinates and the fourth dimension corresponds to time. The result of the mean across time is to be stored in a 3D image.

```
#include "itkImageLinearConstIteratorWithIndex.h"
```

First we declare the types of the images

```
  typedef unsigned char               PixelType;
  typedef itk::Image< PixelType, 3 >  Image3DType;
  typedef itk::Image< PixelType, 4 >  Image4DType;

  typedef itk::ImageFileReader< Image4DType > Reader4DType;
  typedef itk::ImageFileWriter< Image3DType > Writer3DType;

  Reader4DType::Pointer reader4D = Reader4DType::New();
  reader4D->SetFileName( argv[1] );

  try
    {
    reader4D->Update();
    }
  catch( itk::ExceptionObject & excp )
    {
    std::cerr << "Error writing the image" << std::endl;
    std::cerr << excp << std::endl;
    return EXIT_FAILURE;
    }



  Image4DType::ConstPointer image4D = reader4D->GetOutput();

  Image3DType::Pointer image3D = Image3DType::New();
  typedef Image3DType::IndexType    Index3DType;
  typedef Image3DType::SizeType     Size3DType;
  typedef Image3DType::RegionType   Region3DType;
  typedef Image3DType::SpacingType  Spacing3DType;
  typedef Image3DType::PointType    Origin3DType;

  typedef Image4DType::IndexType    Index4DType;
```

```
typedef Image4DType::SizeType     Size4DType;
typedef Image4DType::RegionType   Region4DType;
typedef Image4DType::SpacingType  Spacing4DType;
typedef Image4DType::PointType    Origin4DType;

Index3DType       index3D;
Size3DType        size3D;
Spacing3DType     spacing3D;
Origin3DType      origin3D;

Image4DType::RegionType region4D = image4D->GetBufferedRegion();

Index4DType       index4D   = region4D.GetIndex();
Size4DType        size4D    = region4D.GetSize();
Spacing4DType     spacing4D = image4D->GetSpacing();
Origin4DType      origin4D  = image4D->GetOrigin();

for( unsigned int i=0; i < 3; i++)
  {
  size3D[i]    = size4D[i];
  index3D[i]   = index4D[i];
  spacing3D[i] = spacing4D[i];
  origin3D[i]  = origin4D[i];
  }

image3D->SetSpacing( spacing3D );
image3D->SetOrigin(  origin3D  );

Region3DType region3D;
region3D.SetIndex( index3D );
region3D.SetSize( size3D );

image3D->SetRegions( region3D  );
image3D->Allocate();


typedef itk::NumericTraits< PixelType >::AccumulateType    SumType;
typedef itk::NumericTraits< SumType   >::RealType          MeanType;

const unsigned int timeLength = region4D.GetSize()[3];

typedef itk::ImageLinearConstIteratorWithIndex<
                                Image4DType > IteratorType;

IteratorType it( image4D, region4D );
it.SetDirection( 3 ); // Walk along time dimension
it.GoToBegin();
while( !it.IsAtEnd() )
```

```
{
SumType sum = itk::NumericTraits< SumType >::Zero;
it.GoToBeginOfLine();
index4D = it.GetIndex();
while( !it.IsAtEndOfLine() )
  {
   sum += it.Get();
   ++it;
  }
MeanType mean = static_cast< MeanType >( sum ) /
                static_cast< MeanType >( timeLength );

index3D[0] = index4D[0];
index3D[1] = index4D[1];
index3D[2] = index4D[2];

image3D->SetPixel( index3D, static_cast< PixelType >( mean ) );
it.NextLine();
}
```

As you can see, we avoid to use a 3D iterator to walk over the mean image. The reason is that there is no guarantee that the 3D iterator will walk in the same order as the 4D. Iterators just adhere to their contract of visiting all the pixel, but do not enforce any particular order for the visits. The linear iterator guarantees to visit the pixels along a line of the image in the order in which they are placed in the line, but do not states in what order one line will be visited with respect to other lines. Here we simply take advantage of knowing the first three components of the 4D iterator index, and use them to place the resulting mean value in the output 3D image.

11.3.4 ImageSliceIteratorWithIndex

The source code for this section can be found in the file
`Examples/Iterators/ImageSliceIteratorWithIndex.cxx`.

The `itk::ImageSliceIteratorWithIndex` class is an extension of `itk::ImageLinearIteratorWithIndex` from iteration along lines to iteration along both lines *and planes* in an image. A *slice* is a 2D plane spanned by two vectors pointing along orthogonal coordinate axes. The slice orientation of the slice iterator is defined by specifying its two spanning axes.

- **`SetFirstDirection()`** Specifies the first coordinate axis direction of the slice plane.
- **`SetSecondDirection()`** Specifies the second coordinate axis direction of the slice plane.

Several new methods control movement from slice to slice.

- **`NextSlice()`** Moves the iterator to the beginning pixel location of the next slice in the image. The origin of the next slice is calculated by incrementing the current origin index along the fastest increasing dimension of the image subspace which excludes the first and second dimensions of the iterator.
- **`PreviousSlice()`** Moves the iterator to the *last valid pixel location* in the previous slice. The origin of the previous slice is calculated by decrementing the current origin index along the fastest increasing dimension of the image subspace which excludes the first and second dimensions of the iterator.
- **`IsAtReverseEndOfSlice()`** Returns true if the iterator points to *one position before* the beginning pixel of the current slice.
- **`IsAtEndOfSlice()`** Returns true if the iterator points to *one position past* the last valid pixel of the current slice.

The slice iterator moves line by line using `NextLine()` and `PreviousLine()`. The line direction is parallel to the *second* coordinate axis direction of the slice plane (see also Section 11.3.3).

The next code example calculates the maximum intensity projection along one of the coordinate axes of an image volume. The algorithm is straightforward using ImageSliceIteratorWithIndex because we can coordinate movement through a slice of the 3D input image with movement through the 2D planar output.

Here is how the algorithm works. For each 2D slice of the input, iterate through all the pixels line by line. Copy a pixel value to the corresponding position in the 2D output image if it is larger than the value already contained there. When all slices have been processed, the output image is the desired maximum intensity projection.

We include a header for the const version of the slice iterator. For writing values to the 2D projection image, we use the linear iterator from the previous section. The linear iterator is chosen because it can be set to follow the same path in its underlying 2D image that the slice iterator follows over each slice of the 3D image.

```
#include "itkImageSliceConstIteratorWithIndex.h"
#include "itkImageLinearIteratorWithIndex.h"
```

The pixel type is defined as `unsigned short`. For this application, we need two image types, a 3D image for the input, and a 2D image for the intensity projection.

```
  typedef unsigned short PixelType;
  typedef itk::Image< PixelType, 2 >  ImageType2D;
  typedef itk::Image< PixelType, 3 >  ImageType3D;
```

A slice iterator type is defined to walk the input image.

```
typedef itk::ImageLinearIteratorWithIndex< ImageType2D > LinearIteratorType;
typedef itk::ImageSliceConstIteratorWithIndex< ImageType3D >  SliceIteratorType;
```

The projection direction is read from the command line. The projection image will be the size of the 2D plane orthogonal to the projection direction. Its spanning vectors are the two remaining coordinate axes in the volume. These axes are recorded in the `direction` array.

```
unsigned int projectionDirection =
  static_cast<unsigned int>( ::atoi( argv[3] ) );

unsigned int i, j;
unsigned int direction[2];
for (i = 0, j = 0; i < 3; ++i )
  {
  if (i != projectionDirection)
    {
    direction[j] = i;
    j++;
    }
  }
```

The `direction` array is now used to define the projection image size based on the input image size. The output image is created so that its common dimension(s) with the input image are the same size. For example, if we project along the *x* axis of the input, the size and origin of the *y* axes of the input and output will match. This makes the code slightly more complicated, but prevents a counter-intuitive rotation of the output.

```
ImageType2D::RegionType region;
ImageType2D::RegionType::SizeType size;
ImageType2D::RegionType::IndexType index;

ImageType3D::RegionType requestedRegion = inputImage->GetRequestedRegion();

index[ direction[0] ]    = requestedRegion.GetIndex()[ direction[0] ];
index[ 1- direction[0] ] = requestedRegion.GetIndex()[ direction[1] ];
size[ direction[0] ]     = requestedRegion.GetSize()[  direction[0] ];
size[ 1- direction[0] ]  = requestedRegion.GetSize()[  direction[1] ];

region.SetSize( size );
region.SetIndex( index );

ImageType2D::Pointer outputImage = ImageType2D::New();

outputImage->SetRegions( region );
outputImage->Allocate();
```

Next we create the necessary iterators. The const slice iterator walks the 3D input image, and the non-const linear iterator walks the 2D output image. The iterators are initialized to walk the same linear path through a slice. Remember that the *second* direction of the slice iterator defines the direction that linear iteration walks within a slice.

```
SliceIteratorType  inputIt(  inputImage,  inputImage->GetRequestedRegion() );
LinearIteratorType outputIt( outputImage, outputImage->GetRequestedRegion() );

inputIt.SetFirstDirection(  direction[1] );
inputIt.SetSecondDirection( direction[0] );

outputIt.SetDirection( 1 - direction[0] );
```

Now we are ready to compute the projection. The first step is to initialize all of the projection values to their nonpositive minimum value. The projection values are then updated row by row from the first slice of the input. At the end of the first slice, the input iterator steps to the first row in the next slice, while the output iterator, whose underlying image consists of only one slice, rewinds to its first row. The process repeats until the last slice of the input is processed.

```
outputIt.GoToBegin();
while ( ! outputIt.IsAtEnd() )
  {
  while ( ! outputIt.IsAtEndOfLine() )
    {
    outputIt.Set( itk::NumericTraits<unsigned short>::NonpositiveMin() );
    ++outputIt;
    }
  outputIt.NextLine();
  }

inputIt.GoToBegin();
outputIt.GoToBegin();

while( !inputIt.IsAtEnd() )
  {
  while ( !inputIt.IsAtEndOfSlice() )
    {
    while ( !inputIt.IsAtEndOfLine() )
      {
      outputIt.Set( vnl_math_max( outputIt.Get(), inputIt.Get() ));
      ++inputIt;
      ++outputIt;
      }
    outputIt.NextLine();
    inputIt.NextLine();

    }
```

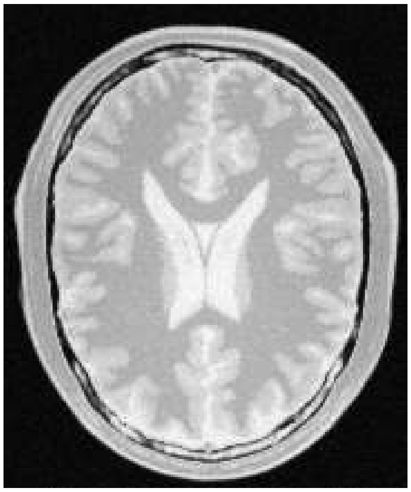


Figure 11.4: The maximum intensity projection through three slices of a volume.

```
    outputIt.GoToBegin();
    inputIt.NextSlice();
    }
```

Running this example code on the 3D image `Examples/Data/BrainProtonDensity3Slices.mha` using the *z*-axis as the axis of projection gives the image shown in Figure 11.4.

11.3.5 ImageRandomConstIteratorWithIndex

The source code for this section can be found in the file
`Examples/Iterators/ImageRandomConstIteratorWithIndex.cxx`.

`itk::ImageRandomConstIteratorWithIndex` was developed to randomly sample pixel values. When incremented or decremented, it jumps to a random location in its image region.

The user must specify a sample size when creating this iterator. The sample size, rather than a specific image index, defines the end position for the iterator. `IsAtEnd()` returns `true` when the current sample number equals the sample size. `IsAtBegin()` returns `true` when the current sample number equals zero. An important difference from other image iterators is that ImageRandomConstIteratorWithIndex may visit the same pixel more than once.

Let's use the random iterator to estimate some simple image statistics. The next example calculates an estimate of the arithmetic mean of pixel values.

First, include the appropriate header and declare pixel and image types.

```
#include "itkImageRandomConstIteratorWithIndex.h"
```

	Sample Size			
	10	**100**	**1000**	**10000**
RatLungSlice1.mha	50.5	52.4	53.0	52.4
RatLungSlice2.mha	46.7	47.5	47.4	47.6
BrainT1Slice.png	47.2	64.1	68.0	67.8

Table 11.1: Estimates of mean image pixel value using the ImageRandomConstIteratorWithIndex at different sample sizes.

```
const unsigned int Dimension = 2;

typedef unsigned short  PixelType;
typedef itk::Image< PixelType, Dimension >  ImageType;
typedef itk::ImageRandomConstIteratorWithIndex< ImageType >  ConstIteratorType;
```

The input image has been read as `inputImage`. We now create an iterator with a number of samples set by command line argument. The call to `ReinitializeSeed` seeds the random number generator. The iterator is initialized over the entire valid image region.

```
ConstIteratorType inputIt(  inputImage,  inputImage->GetRequestedRegion() );
inputIt.SetNumberOfSamples( ::atoi( argv[2]) );
inputIt.ReinitializeSeed();
```

Now take the specified number of samples and calculate their average value.

```
float mean = 0.0f;
for ( inputIt.GoToBegin(); ! inputIt.IsAtEnd(); ++inputIt)
  {
  mean += static_cast<float>( inputIt.Get() );
  }
mean = mean / ::atof( argv[2] );
```

Table 11.3.5 shows the results of running this example on several of the data files from `Examples/Data` with a range of sample sizes.

11.4 Neighborhood Iterators

In ITK, a pixel neighborhood is loosely defined as a small set of pixels that are locally adjacent to one another in an image. The size and shape of a neighborhood, as well the connectivity among pixels in a neighborhood, may vary with the application.

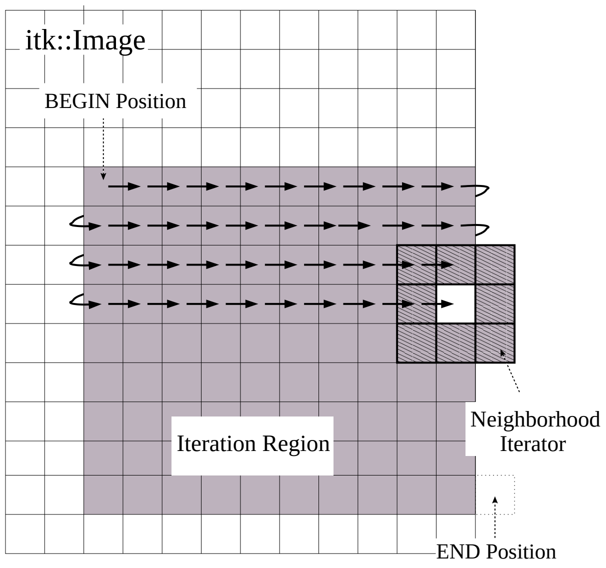


Figure 11.5: Path of a 3x3 neighborhood iterator through a 2D image region. The extent of the neighborhood is indicated by the hashing around the iterator position. Pixels that lie within this extent are accessible through the iterator. An arrow denotes a single iterator step, the result of one ++ operation.

Many image processing algorithms are neighborhood-based, that is, the result at a pixel i is computed from the values of pixels in the ND neighborhood of i. Consider finite difference operations in 2D. A derivative at pixel index $i = (j,k)$, for example, is taken as a weighted difference of the values at $(j+1,k)$ and $(j-1,k)$. Other common examples of neighborhood operations include convolution filtering and image morphology.

This section describes a class of ITK image iterators that are designed for working with pixel neighborhoods. An ITK neighborhood iterator walks an image region just like a normal image iterator, but instead of only referencing a single pixel at each step, it simultaneously points to the entire ND neighborhood of pixels. Extensions to the standard iterator interface provide read and write access to all neighborhood pixels and information such as the size, extent, and location of the neighborhood.

Neighborhood iterators use the same operators defined in Section 11.2 and the same code constructs as normal iterators for looping through an image. Figure 11.5 shows a neighborhood iterator moving through an iteration region. This iterator defines a 3x3 neighborhood around each pixel that it visits. The *center* of the neighborhood iterator is always positioned over its current index and all other neighborhood pixel indices are referenced as offsets from the center index. The pixel under the center of the neighborhood iterator and all pixels under the shaded area, or *extent*, of the iterator can be dereferenced.

In addition to the standard image pointer and iteration region (Section 11.2), neighborhood

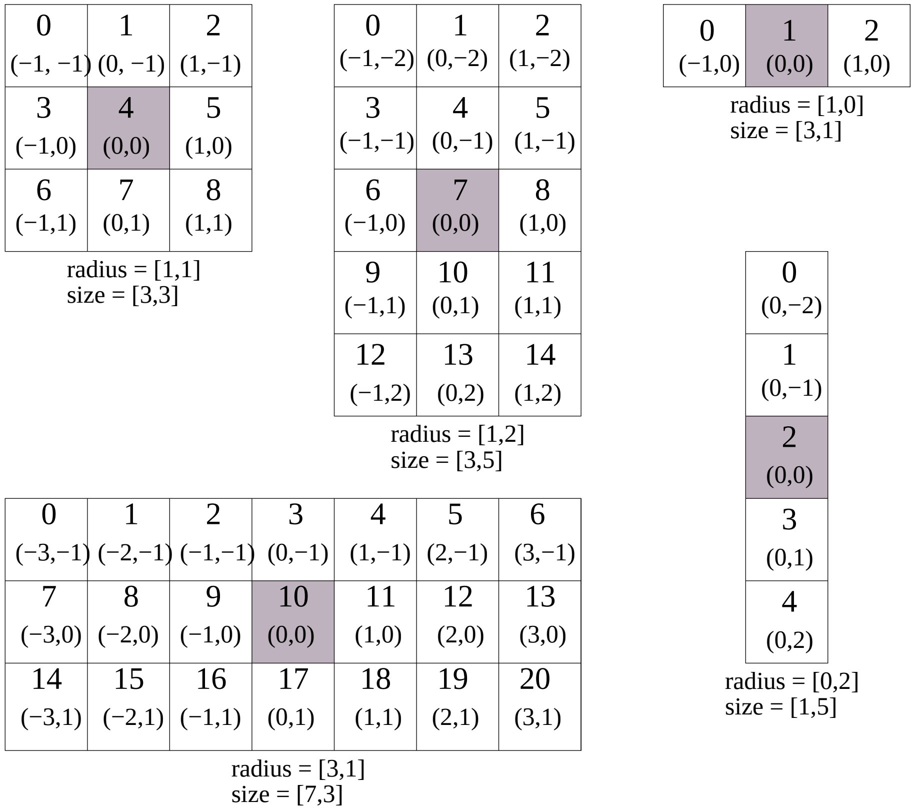


Figure 11.6: Several possible 2D neighborhood iterator shapes are shown along with their radii and sizes. A neighborhood pixel can be dereferenced by its integer index (top) or its offset from the center (bottom). The center pixel of each iterator is shaded.

iterator constructors require an argument that specifies the extent of the neighborhood to cover. Neighborhood extent is symmetric across its center in each axis and is given as an array of N distances that are collectively called the *radius*. Each element d of the radius, where $0 < d < N$ and N is the dimensionality of the neighborhood, gives the extent of the neighborhood in pixels for dimension N. The length of each face of the resulting ND hypercube is $2d + 1$ pixels, a distance of d on either side of the single pixel at the neighbor center. Figure 11.6 shows the relationship between the radius of the iterator and the size of the neighborhood for a variety of 2D iterator shapes.

The radius of the neighborhood iterator is queried after construction by calling the `GetRadius()` method. Some other methods provide some useful information about the iterator and its underlying image.

- **`SizeType GetRadius()`** Returns the ND radius of the neighborhood as an itk::Size.

- **const ImageType *GetImagePointer()** Returns the pointer to the image referenced by the iterator.
- **unsigned long Size()** Returns the size in number of pixels of the neighborhood.

The neighborhood iterator interface extends the normal ITK iterator interface for setting and getting pixel values. One way to dereference pixels is to think of the neighborhood as a linear array where each pixel has a unique integer index. The index of a pixel in the array is determined by incrementing from the upper-left-forward corner of the neighborhood along the fastest increasing image dimension: first column, then row, then slice, and so on. In Figure 11.6, the unique integer index is shown at the top of each pixel. The center pixel is always at position $n/2$, where n is the size of the array.

- **PixelType GetPixel(const unsigned int i)** Returns the value of the pixel at neighborhood position `i`.
- **void SetPixel(const unsigned int i, PixelType p)** Sets the value of the pixel at position `i` to `p`.

Another way to think about a pixel location in a neighborhood is as an ND offset from the neighborhood center. The upper-left-forward corner of a 3*x*3*x*3 neighborhood, for example, can be described by offset $(-1,-1,-1)$. The bottom-right-back corner of the same neighborhood is at offset $(1,1,1)$. In Figure 11.6, the offset from center is shown at the bottom of each neighborhood pixel.

- **PixelType GetPixel(const OffsetType &o)** Get the value of the pixel at the position offset `o` from the neighborhood center.
- **void SetPixel(const OffsetType &o, PixelType p)** Set the value at the position offset `o` from the neighborhood center to the value `p`.

The neighborhood iterators also provide a shorthand for setting and getting the value at the center of the neighborhood.

- **PixelType GetCenterPixel()** Gets the value at the center of the neighborhood.
- **void SetCenterPixel(PixelType p)** Sets the value at the center of the neighborhood to the value `p`

There is another shorthand for setting and getting values for pixels that lie some integer distance from the neighborhood center along one of the image axes.

- **`PixelType GetNext(unsigned int d)`** Get the value immediately adjacent to the neighborhood center in the positive direction along the `d` axis.
- **`void SetNext(unsigned int d, PixelType p)`** Set the value immediately adjacent to the neighborhood center in the positive direction along the `d` axis to the value `p`.
- **`PixelType GetPrevious(unsigned int d)`** Get the value immediately adjacent to the neighborhood center in the negative direction along the `d` axis.
- **`void SetPrevious(unsigned int d, PixelType p)`** Set the value immediately adjacent to the neighborhood center in the negative direction along the `d` axis to the value `p`.
- **`PixelType GetNext(unsigned int d, unsigned int s)`** Get the value of the pixel located `s` pixels from the neighborhood center in the positive direction along the `d` axis.
- **`void SetNext(unsigned int d, unsigned int s, PixelType p)`** Set the value of the pixel located `s` pixels from the neighborhood center in the positive direction along the `d` axis to value `p`.
- **`PixelType GetPrevious(unsigned int d, unsigned int s)`** Get the value of the pixel located `s` pixels from the neighborhood center in the positive direction along the `d` axis.
- **`void SetPrevious(unsigned int d, unsigned int s, PixelType p)`** Set the value of the pixel located `s` pixels from the neighborhood center in the positive direction along the `d` axis to value `p`.

It is also possible to extract or set all of the neighborhood values from an iterator at once using a regular ITK neighborhood object. This may be useful in algorithms that perform a particularly large number of calculations in the neighborhood and would otherwise require multiple dereferences of the same pixels.

- **`NeighborhoodType GetNeighborhood()`** Return a `itk::Neighborhood` of the same size and shape as the neighborhood iterator and contains all of the values at the iterator position.
- **`void SetNeighborhood(NeighborhoodType &N)`** Set all of the values in the neighborhood at the iterator position to those contained in Neighborhood `N`, which must be the same size and shape as the iterator.

Several methods are defined to provide information about the neighborhood.

- **`IndexType GetIndex()`** Return the image index of the center pixel of the neighborhood iterator.

- **`IndexType GetIndex(OffsetType o)`** Return the image index of the pixel at offset `o` from the neighborhood center.
- **`IndexType GetIndex(unsigned int i)`** Return the image index of the pixel at array position `i`.
- **`OffsetType GetOffset(unsigned int i)`** Return the offset from the neighborhood center of the pixel at array position `i`.
- **`unsigned long GetNeighborhoodIndex(OffsetType o)`** Return the array position of the pixel at offset `o` from the neighborhood center.
- **`std::slice GetSlice(unsigned int n)`** Return a `std::slice` through the iterator neighborhood along axis `n`.

A neighborhood-based calculation in a neighborhood close to an image boundary may require data that falls outside the boundary. The iterator in Figure 11.5, for example, is centered on a boundary pixel such that three of its neighbors actually do not exist in the image. When the extent of a neighborhood falls outside the image, pixel values for missing neighbors are supplied according to a rule, usually chosen to satisfy the numerical requirements of the algorithm. A rule for supplying out-of-bounds values is called a *boundary condition*.

ITK neighborhood iterators automatically detect out-of-bounds dereferences and will return values according to boundary conditions. The boundary condition type is specified by the second, optional template parameter of the iterator. By default, neighborhood iterators use a Neumann condition where the first derivative across the boundary is zero. The Neumann rule simply returns the closest in-bounds pixel value to the requested out-of-bounds location. Several other common boundary conditions can be found in the ITK toolkit. They include a periodic condition that returns the pixel value from the opposite side of the data set, and is useful when working with periodic data such as Fourier transforms, and a constant value condition that returns a set value v for all out-of-bounds pixel dereferences. The constant value condition is equivalent to padding the image with value v.

Bounds checking is a computationally expensive operation because it occurs each time the iterator is incremented. To increase efficiency, a neighborhood iterator automatically disables bounds checking when it detects that it is not necessary. A user may also explicitly disable or enable bounds checking. Most neighborhood based algorithms can minimize the need for bounds checking through clever definition of iteration regions. These techniques are explored in Section 11.4.1.

- **`void NeedToUseBoundaryConditionOn()`** Explicitly turn bounds checking on. This method should be used with caution because unnecessarily enabling bounds checking may result in a significant performance decrease. In general you should allow the iterator to automatically determine this setting.
- **`void NeedToUseBoundaryConditionOff()`** Explicitly disable bounds checking. This method should be used with caution because disabling bounds checking when it is needed will result in out-of-bounds reads and undefined results.

- **`void OverrideBoundaryCondition(BoundaryConditionType *b)`** Overrides the templated boundary condition, using boundary condition object `b` instead. Object `b` should not be deleted until it has been released by the iterator. This method can be used to change iterator behavior at run-time.
- **`void ResetBoundaryCondition()`** Discontinues the use of any run-time specified boundary condition and returns to using the condition specified in the template argument.
- **`void SetPixel(unsigned int i, PixelType p, bool status)`** Sets the value at neighborhood array position `i` to value `p`. If the position `i` is out-of-bounds, `status` is set to `false`, otherwise `status` is set to `true`.

The following sections describe the two ITK neighborhood iterator classes, `itk::NeighborhoodIterator` and `itk::ShapedNeighborhoodIterator`. Each has a const and a non-const version. The shaped iterator is a refinement of the standard NeighborhoodIterator that supports an arbitrarily-shaped (non-rectilinear) neighborhood.

11.4.1 NeighborhoodIterator

The standard neighborhood iterator class in ITK is the `itk::NeighborhoodIterator`. Together with its `const` version, `itk::ConstNeighborhoodIterator`, it implements the complete API described above. This section provides several examples to illustrate the use of NeighborhoodIterator.

Basic neighborhood techniques: edge detection

The source code for this section can be found in the file
`Examples/Iterators/NeighborhoodIterators1.cxx`.

This example uses the `itk::NeighborhoodIterator` to implement a simple Sobel edge detection algorithm [30]. The algorithm uses the neighborhood iterator to iterate through an input image and calculate a series of finite difference derivatives. Since the derivative results cannot be written back to the input image without affecting later calculations, they are written instead to a second, output image. Most neighborhood processing algorithms follow this read-only model on their inputs.

We begin by including the proper header files. The `itk::ImageRegionIterator` will be used to write the results of computations to the output image. A const version of the neighborhood iterator is used because the input image is read-only.

```
#include "itkConstNeighborhoodIterator.h"
#include "itkImageRegionIterator.h"
```

The finite difference calculations in this algorithm require floating point values. Hence, we define the image pixel type to be `float` and the file reader will automatically cast fixed-point data to `float`.

We declare the iterator types using the image type as the template parameter. The second template parameter of the neighborhood iterator, which specifies the boundary condition, has been omitted because the default condition is appropriate for this algorithm.

```
typedef float PixelType;
typedef itk::Image< PixelType, 2 >  ImageType;
typedef itk::ImageFileReader< ImageType > ReaderType;

typedef itk::ConstNeighborhoodIterator< ImageType > NeighborhoodIteratorType;
typedef itk::ImageRegionIterator< ImageType>        IteratorType;
```

The following code creates and executes the ITK image reader. The `Update` call on the reader object is surrounded by the standard `try/catch` blocks to handle any exceptions that may be thrown by the reader.

```
ReaderType::Pointer reader = ReaderType::New();
reader->SetFileName( argv[1] );
try
  {
  reader->Update();
  }
catch ( itk::ExceptionObject &err)
  {
  std::cout << "ExceptionObject caught !" << std::endl;
  std::cout << err << std::endl;
  return -1;
  }
```

We can now create a neighborhood iterator to range over the output of the reader. For Sobel edge-detection in 2D, we need a square iterator that extends one pixel away from the neighborhood center in every dimension.

```
NeighborhoodIteratorType::RadiusType radius;
radius.Fill(1);
NeighborhoodIteratorType it( radius, reader->GetOutput(),
                             reader->GetOutput()->GetRequestedRegion() );
```

The following code creates an output image and iterator.

```
ImageType::Pointer output = ImageType::New();
output->SetRegions(reader->GetOutput()->GetRequestedRegion());
```

```
output->Allocate();

IteratorType out(output, reader->GetOutput()->GetRequestedRegion());
```

Sobel edge detection uses weighted finite difference calculations to construct an edge magnitude image. Normally the edge magnitude is the root sum of squares of partial derivatives in all directions, but for simplicity this example only calculates the *x* component. The result is a derivative image biased toward maximally vertical edges.

The finite differences are computed from pixels at six locations in the neighborhood. In this example, we use the iterator `GetPixel()` method to query the values from their offsets in the neighborhood. The example in Section 11.4.1 uses convolution with a Sobel kernel instead.

Six positions in the neighborhood are necessary for the finite difference calculations. These positions are recorded in `offset1` through `offset6`.

```
NeighborhoodIteratorType::OffsetType offset1 = {{-1,-1}};
NeighborhoodIteratorType::OffsetType offset2 = {{1,-1}};
NeighborhoodIteratorType::OffsetType offset3 = {{-1,0 }};
NeighborhoodIteratorType::OffsetType offset4 = {{1,0}};
NeighborhoodIteratorType::OffsetType offset5 = {{-1,1}};
NeighborhoodIteratorType::OffsetType offset6 = {{1,1}};
```

It is equivalent to use the six corresponding integer array indices instead. For example, the offsets `(-1,-1)` and `(1, -1)` are equivalent to the integer indices `0` and `2`, respectively.

The calculations are done in a `for` loop that moves the input and output iterators synchronously across their respective images. The `sum` variable is used to sum the results of the finite differences.

```
for (it.GoToBegin(), out.GoToBegin(); !it.IsAtEnd(); ++it, ++out)
  {
  float sum;
  sum = it.GetPixel(offset2) - it.GetPixel(offset1);
  sum += 2.0 * it.GetPixel(offset4) - 2.0 * it.GetPixel(offset3);
  sum += it.GetPixel(offset6) - it.GetPixel(offset5);
  out.Set(sum);
  }
```

The last step is to write the output buffer to an image file. Writing is done inside a `try/catch` block to handle any exceptions. The output is rescaled to intensity range $[0,255]$ and cast to unsigned char so that it can be saved and visualized as a PNG image.

```
typedef unsigned char WritePixelType;
typedef itk::Image< WritePixelType, 2 > WriteImageType;
typedef itk::ImageFileWriter< WriteImageType > WriterType;
```

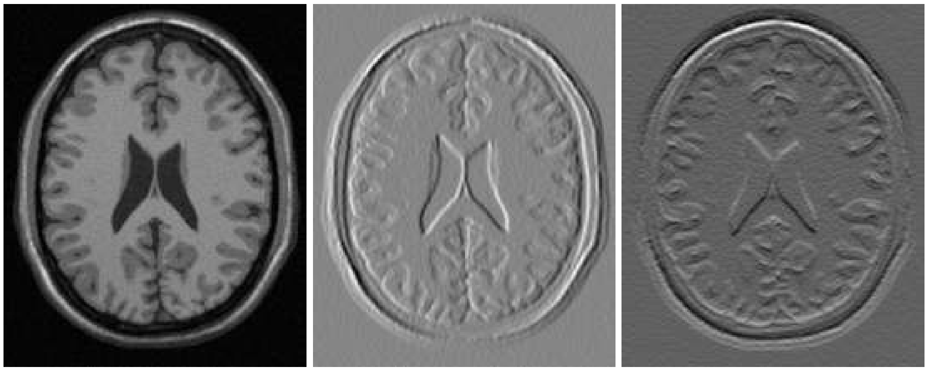


Figure 11.7: Applying the Sobel operator in different orientations to an MRI image (left) produces x (center) and y (right) derivative images.

```
typedef itk::RescaleIntensityImageFilter<
             ImageType, WriteImageType > RescaleFilterType;

RescaleFilterType::Pointer rescaler = RescaleFilterType::New();

rescaler->SetOutputMinimum(   0 );
rescaler->SetOutputMaximum( 255 );
rescaler->SetInput(output);

WriterType::Pointer writer = WriterType::New();
writer->SetFileName( argv[2] );
writer->SetInput(rescaler->GetOutput());
try
  {
  writer->Update();
  }
catch ( itk::ExceptionObject &err)
  {
  std::cout << "ExceptionObject caught !" << std::endl;
  std::cout << err << std::endl;
  return -1;
  }
```

The center image of Figure 11.7 shows the output of the Sobel algorithm applied to `Examples/Data/BrainT1Slice.png`.

Convolution filtering: Sobel operator

The source code for this section can be found in the file
`Examples/Iterators/NeighborhoodIterators2.cxx`.

In this example, the Sobel edge-detection routine is rewritten using convolution filtering. Convolution filtering is a standard image processing technique that can be implemented numerically as the inner product of all image neighborhoods with a convolution kernel [30] [15]. In ITK, we use a class of objects called *neighborhood operators* as convolution kernels and a special function object called `itk::NeighborhoodInnerProduct` to calculate inner products.

The basic ITK convolution filtering routine is to step through the image with a neighborhood iterator and use NeighborhoodInnerProduct to find the inner product of each neighborhood with the desired kernel. The resulting values are written to an output image. This example uses a neighborhood operator called the `itk::SobelOperator`, but all neighborhood operators can be convolved with images using this basic routine. Other examples of neighborhood operators include derivative kernels, Gaussian kernels, and morphological operators. `itk::NeighborhoodOperatorImageFilter` is a generalization of the code in this section to ND images and arbitrary convolution kernels.

We start writing this example by including the header files for the Sobel kernel and the inner product function.

```
#include "itkSobelOperator.h"
#include "itkNeighborhoodInnerProduct.h"
```

Refer to the previous example for a description of reading the input image and setting up the output image and iterator.

The following code creates a Sobel operator. The Sobel operator requires a direction for its partial derivatives. This direction is read from the command line. Changing the direction of the derivatives changes the bias of the edge detection, i.e. maximally vertical or maximally horizontal.

```
  itk::SobelOperator<PixelType, 2> sobelOperator;
  sobelOperator.SetDirection( ::atoi(argv[3]) );
  sobelOperator.CreateDirectional();
```

The neighborhood iterator is initialized as before, except that now it takes its radius directly from the radius of the Sobel operator. The inner product function object is templated over image type and requires no initialization.

```
  NeighborhoodIteratorType::RadiusType radius = sobelOperator.GetRadius();
  NeighborhoodIteratorType it( radius, reader->GetOutput(),
                               reader->GetOutput()->GetRequestedRegion() );

  itk::NeighborhoodInnerProduct<ImageType> innerProduct;
```

Using the Sobel operator, inner product, and neighborhood iterator objects, we can now write a very simple `for` loop for performing convolution filtering. As before, out-of-bounds pixel values are supplied automatically by the iterator.

```
  for (it.GoToBegin(), out.GoToBegin(); !it.IsAtEnd(); ++it, ++out)
    {
    out.Set( innerProduct( it, sobelOperator ) );
    }
```

The output is rescaled and written as in the previous example. Applying this example in the *x* and *y* directions produces the images at the center and right of Figure 11.7. Note that x-direction operator produces the same output image as in the previous example.

Optimizing iteration speed

The source code for this section can be found in the file
`Examples/Iterators/NeighborhoodIterators3.cxx`.

This example illustrates a technique for improving the efficiency of neighborhood calculations by eliminating unnecessary bounds checking. As described in Section 11.4, the neighborhood iterator automatically enables or disables bounds checking based on the iteration region in which it is initialized. By splitting our image into boundary and non-boundary regions, and then processing each region using a different neighborhood iterator, the algorithm will only perform bounds-checking on those pixels for which it is actually required. This trick can provide a significant speedup for simple algorithms such as our Sobel edge detection, where iteration speed is a critical.

Splitting the image into the necessary regions is an easy task when you use the `itk::ImageBoundaryFacesCalculator`. The face calculator is so named because it returns a list of the "faces" of the ND dataset. Faces are those regions whose pixels all lie within a distance *d* from the boundary, where *d* is the radius of the neighborhood stencil used for the numerical calculations. In other words, faces are those regions where a neighborhood iterator of radius *d* will always overlap the boundary of the image. The face calculator also returns the single *inner* region, in which out-of-bounds values are never required and bounds checking is not necessary.

The face calculator object is defined in `itkNeighborhoodAlgorithm.h`. We include this file in addition to those from the previous two examples.

```
#include "itkNeighborhoodAlgorithm.h"
```

First we load the input image and create the output image and inner product function as in the previous examples. The image iterators will be created in a later step. Next we create a face calculator object. An empty list is created to hold the regions that will later on be returned by the face calculator.

```
typedef itk::NeighborhoodAlgorithm
  ::ImageBoundaryFacesCalculator< ImageType > FaceCalculatorType;

FaceCalculatorType faceCalculator;
FaceCalculatorType::FaceListType faceList;
```

The face calculator function is invoked by passing it an image pointer, an image region, and a neighborhood radius. The image pointer is the same image used to initialize the neighborhood iterator, and the image region is the region that the algorithm is going to process. The radius is the radius of the iterator.

Notice that in this case the image region is given as the region of the *output* image and the image pointer is given as that of the *input* image. This is important if the input and output images differ in size, i.e. the input image is larger than the output image. ITK image filters, for example, operate on data from the input image but only generate results in the `RequestedRegion` of the output image, which may be smaller than the full extent of the input.

```
faceList = faceCalculator(reader->GetOutput(), output->GetRequestedRegion(),
                          sobelOperator.GetRadius());
```

The face calculator has returned a list of $2N+1$ regions. The first element in the list is always the inner region, which may or may not be important depending on the application. For our purposes it does not matter because all regions are processed the same way. We use an iterator to traverse the list of faces.

```
FaceCalculatorType::FaceListType::iterator fit;
```

We now rewrite the main loop of the previous example so that each region in the list is processed by a separate iterator. The iterators `it` and `out` are reinitialized over each region in turn. Bounds checking is automatically enabled for those regions that require it, and disabled for the region that does not.

```
IteratorType out;
NeighborhoodIteratorType it;

for ( fit=faceList.begin(); fit != faceList.end(); ++fit)
  {
  it = NeighborhoodIteratorType( sobelOperator.GetRadius(),
                                 reader->GetOutput(), *fit );
  out = IteratorType( output, *fit );

  for (it.GoToBegin(), out.GoToBegin(); ! it.IsAtEnd(); ++it, ++out)
    {
    out.Set( innerProduct(it, sobelOperator) );
    }
  }
```

The output is written as before. Results for this example are the same as the previous example. You may not notice the speedup except on larger images. When moving to 3D and higher dimensions, the effects are greater because the volume to surface area ratio is usually larger. In other words, as the number of interior pixels increases relative to the number of face pixels, there is a corresponding increase in efficiency from disabling bounds checking on interior pixels.

Separable convolution: Gaussian filtering

The source code for this section can be found in the file
`Examples/Iterators/NeighborhoodIterators4.cxx`.

We now introduce a variation on convolution filtering that is useful when a convolution kernel is separable. In this example, we create a different neighborhood iterator for each axial direction of the image and then take separate inner products with a 1D discrete Gaussian kernel. The idea of using several neighborhood iterators at once has applications beyond convolution filtering and may improve efficiency when the size of the whole neighborhood relative to the portion of the neighborhood used in calculations becomes large.

The only new class necessary for this example is the Gaussian operator.

```
#include "itkGaussianOperator.h"
```

The Gaussian operator, like the Sobel operator, is instantiated with a pixel type and a dimensionality. Additionally, we set the variance of the Gaussian, which has been read from the command line as standard deviation.

```
  itk::GaussianOperator< PixelType, 2 > gaussianOperator;
  gaussianOperator.SetVariance( ::atof(argv[3]) * ::atof(argv[3]) );
```

The only further changes from the previous example are in the main loop. Once again we use the results from face calculator to construct a loop that processes boundary and non-boundary image regions separately. Separable convolution, however, requires an additional, outer loop over all the image dimensions. The direction of the Gaussian operator is reset at each iteration of the outer loop using the new dimension. The iterators change direction to match because they are initialized with the radius of the Gaussian operator.

Input and output buffers are swapped at each iteration so that the output of the previous iteration becomes the input for the current iteration. The swap is not performed on the last iteration.

```
  ImageType::Pointer input = reader->GetOutput();
  for (unsigned int i = 0; i < ImageType::ImageDimension; ++i)
    {
    gaussianOperator.SetDirection(i);
    gaussianOperator.CreateDirectional();

    faceList = faceCalculator(input, output->GetRequestedRegion(),
```

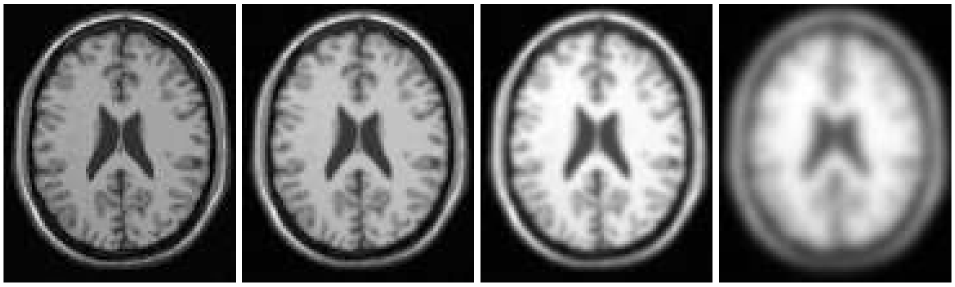


Figure 11.8: Results of convolution filtering with a Gaussian kernel of increasing standard deviation σ (from left to right, $\sigma = 0$, $\sigma = 1$, $\sigma = 2$, $\sigma = 5$). Increased blurring reduces contrast and changes the average intensity value of the image, which causes the image to appear brighter when rescaled.

```
                                gaussianOperator.GetRadius());

  for ( fit=faceList.begin(); fit != faceList.end(); ++fit )
    {
    it = NeighborhoodIteratorType( gaussianOperator.GetRadius(),
                                   input, *fit );

    out = IteratorType( output, *fit );

    for (it.GoToBegin(), out.GoToBegin(); ! it.IsAtEnd(); ++it, ++out)
      {
      out.Set( innerProduct(it, gaussianOperator) );
      }
    }

  // Swap the input and output buffers
  if (i != ImageType::ImageDimension - 1)
    {
    ImageType::Pointer tmp = input;
    input = output;
    output = tmp;
    }
  }
```

The output is rescaled and written as in the previous examples. Figure 11.8 shows the results of Gaussian blurring the image `Examples/Data/BrainT1Slice.png` using increasing kernel widths.

Slicing the neighborhood

The source code for this section can be found in the file
`Examples/Iterators/NeighborhoodIterators5.cxx`.

This example introduces slice-based neighborhood processing. A slice, in this context, is a 1D path through an ND neighborhood. Slices are defined for generic arrays by the `std::slice` class as a start index, a step size, and an end index. Slices simplify the implementation of certain neighborhood calculations. They also provide a mechanism for taking inner products with subregions of neighborhoods.

Suppose, for example, that we want to take partial derivatives in the *y* direction of a neighborhood, but offset those derivatives by one pixel position along the positive *x* direction. For a 3×3, 2D neighborhood iterator, we can construct an `std::slice`, `(start = 2, stride = 3, end = 8)`, that represents the neighborhood offsets $(1,-1)$, $(1,0)$, $(1,1)$ (see Figure 11.6). If we pass this slice as an extra argument to the `itk::NeighborhoodInnerProduct` function, then the inner product is taken only along that slice. This "sliced" inner product with a 1D `itk::DerivativeOperator` gives the desired derivative.

The previous separable Gaussian filtering example can be rewritten using slices and slice-based inner products. In general, slice-based processing is most useful when doing many different calculations on the same neighborhood, where defining multiple iterators as in Section 11.4.1 becomes impractical or inefficient. Good examples of slice-based neighborhood processing can be found in any of the ND anisotropic diffusion function objects, such as `itk::CurvatureNDAnisotropicDiffusionFunction`.

The first difference between this example and the previous example is that the Gaussian operator is only initialized once. Its direction is not important because it is only a 1D array of coefficients.

```
itk::GaussianOperator< PixelType, 2 > gaussianOperator;
gaussianOperator.SetDirection(0);
gaussianOperator.SetVariance( ::atof(argv[3]) * ::atof(argv[3]) );
gaussianOperator.CreateDirectional();
```

Next we need to define a radius for the iterator. The radius in all directions matches that of the single extent of the Gaussian operator, defining a square neighborhood.

```
NeighborhoodIteratorType::RadiusType radius;
radius.Fill( gaussianOperator.GetRadius()[0] );
```

The inner product and face calculator are defined for the main processing loop as before, but now the iterator is reinitialized each iteration with the square `radius` instead of the radius of the operator. The inner product is taken using a slice along the axial direction corresponding to the current iteration. Note the use of `GetSlice()` to return the proper slice from the iterator itself. `GetSlice()` can only be used to return the slice along the complete extent of the axial direction of a neighborhood.

```
ImageType::Pointer input = reader->GetOutput();
faceList = faceCalculator(input, output->GetRequestedRegion(), radius);

for (unsigned int i = 0; i < ImageType::ImageDimension; ++i)
  {
  for ( fit=faceList.begin(); fit != faceList.end(); ++fit )
    {
    it = NeighborhoodIteratorType( radius, input, *fit );
    out = IteratorType( output, *fit );
    for (it.GoToBegin(), out.GoToBegin(); ! it.IsAtEnd(); ++it, ++out)
      {
      out.Set( innerProduct(it.GetSlice(i), it, gaussianOperator) );
      }
    }

  // Swap the input and output buffers
  if (i != ImageType::ImageDimension - 1)
    {
    ImageType::Pointer tmp = input;
    input = output;
    output = tmp;
    }
  }
```

This technique produces exactly the same results as the previous example. A little experimentation, however, will reveal that it is less efficient since the neighborhood iterator is keeping track of extra, unused pixel locations for each iteration, while the previous example only references those pixels that it needs. In cases, however, where an algorithm takes multiple derivatives or convolution products over the same neighborhood, slice-based processing can increase efficiency and simplify the implementation.

Random access iteration

The source code for this section can be found in the file
`Examples/Iterators/NeighborhoodIterators6.cxx`.

Some image processing routines do not need to visit every pixel in an image. Flood-fill and connected-component algorithms, for example, only visit pixels that are locally connected to one another. Algorithms such as these can be efficiently written using the random access capabilities of the neighborhood iterator.

The following example finds local minima. Given a seed point, we can search the neighborhood of that point and pick the smallest value *m*. While *m* is not at the center of our current neighborhood, we move in the direction of *m* and repeat the analysis. Eventually we discover a local minimum and stop. This algorithm is made trivially simple in ND using an ITK neighborhood iterator.

To illustrate the process, we create an image that descends everywhere to a single minimum: a positive distance transform to a point. The details of creating the distance transform are not relevant to the discussion of neighborhood iterators, but can be found in the source code of this example. Some noise has been added to the distance transform image for additional interest.

The variable `input` is the pointer to the distance transform image. The local minimum algorithm is initialized with a seed point read from the command line.

```
  ImageType::IndexType index;
  index[0] = ::atoi(argv[2]);
  index[1] = ::atoi(argv[3]);
```

Next we create the neighborhood iterator and position it at the seed point.

```
  NeighborhoodIteratorType::RadiusType radius;
  radius.Fill(1);
  NeighborhoodIteratorType it(radius, input, input->GetRequestedRegion());

  it.SetLocation(index);
```

Searching for the local minimum involves finding the minimum in the current neighborhood, then shifting the neighborhood in the direction of that minimum. The `for` loop below records the `itk::Offset` of the minimum neighborhood pixel. The neighborhood iterator is then moved using that offset. When a local minimum is detected, `flag` will remain false and the `while` loop will exit. Note that this code is valid for an image of any dimensionality.

```
  bool flag = true;
  while ( flag == true )
    {
    NeighborhoodIteratorType::OffsetType nextMove;
    nextMove.Fill(0);

    flag = false;

    PixelType min = it.GetCenterPixel();
    for (unsigned i = 0; i < it.Size(); i++)
      {
      if ( it.GetPixel(i) < min )
        {
        min = it.GetPixel(i);
        nextMove = it.GetOffset(i);
        flag = true;
        }
      }
    it.SetCenterPixel( 255.0 );
    it += nextMove;
    }
```

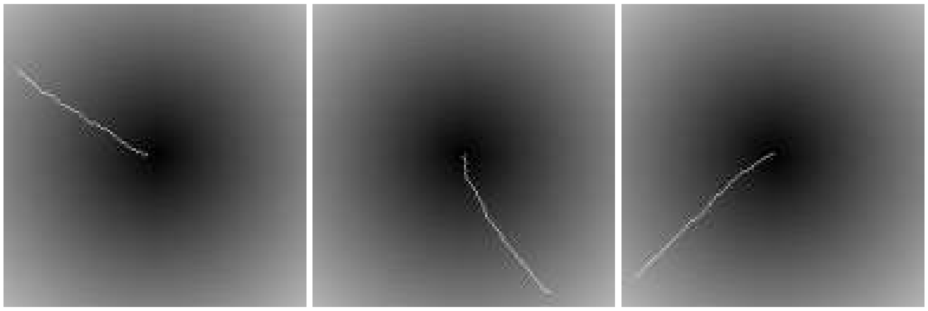


Figure 11.9: Paths traversed by the neighborhood iterator from different seed points to the local minimum. The true minimum is at the center of the image. The path of the iterator is shown in white. The effect of noise in the image is seen as small perturbations in each path.

Figure 11.9 shows the results of the algorithm for several seed points. The white line is the path of the iterator from the seed point to the minimum in the center of the image. The effect of the additive noise is visible as the small perturbations in the paths.

11.4.2 ShapedNeighborhoodIterator

This section describes a variation on the neighborhood iterator called a *shaped* neighborhood iterator. A shaped neighborhood is defined like a bit mask, or *stencil*, with different offsets in the rectilinear neighborhood of the normal neighborhood iterator turned off or on to create a pattern. Inactive positions (those not in the stencil) are not updated during iteration and their values cannot be read or written. The shaped iterator is implemented in the class `itk::ShapedNeighborhoodIterator`, which is a subclass of `itk::NeighborhoodIterator`. A const version, `itk::ConstShapedNeighborhoodIterator`, is also available.

Like a regular neighborhood iterator, a shaped neighborhood iterator must be initialized with an ND radius object, but the radius of the neighborhood of a shaped iterator only defines the set of *possible* neighbors. Any number of possible neighbors can then be activated or deactivated. The shaped neighborhood iterator defines an API for activating neighbors. When a neighbor location, defined relative to the center of the neighborhood, is activated, it is placed on the *active list* and is then part of the stencil. An iterator can be "reshaped" at any time by adding or removing offsets from the active list.

- **`void ActivateOffset(OffsetType &o)`** Include the offset `o` in the stencil of active neighborhood positions. Offsets are relative to the neighborhood center.
- **`void DeactivateOffset(OffsetType &o)`** Remove the offset `o` from the stencil of active neighborhood positions. Offsets are relative to the neighborhood center.

- **`void ClearActiveList()`** Deactivate all positions in the iterator stencil by clearing the active list.
- **`unsigned int GetActiveIndexListSize()`** Return the number of pixel locations that are currently active in the shaped iterator stencil.

Because the neighborhood is less rigidly defined in the shaped iterator, the set of pixel access methods is restricted. Only the `GetPixel()` and `SetPixel()` methods are available, and calling these methods on an inactive neighborhood offset will return undefined results.

For the common case of traversing all pixel offsets in a neighborhood, the shaped iterator class provides an iterator through the active offsets in its stencil. This *stencil iterator* can be incremented or decremented and defines `Get()` and `Set()` for reading and writing the values in the neighborhood.

- **`ShapedNeighborhoodIterator::Iterator Begin()`** Return a const or non-const iterator through the shaped iterator stencil that points to the first valid location in the stencil.
- **`ShapedNeighborhoodIterator::Iterator End()`** Return a const or non-const iterator through the shaped iterator stencil that points *one position past* the last valid location in the stencil.

The functionality and interface of the shaped neighborhood iterator is best described by example. We will use the ShapedNeighborhoodIterator to implement some binary image morphology algorithms (see [30], [15], et al.). The examples that follow implement erosion and dilation.

Shaped neighborhoods: morphological operations

The source code for this section can be found in the file
`Examples/Iterators/ShapedNeighborhoodIterators1.cxx`.

This example uses `itk::ShapedNeighborhoodIterator` to implement a binary erosion algorithm. If we think of an image I as a set of pixel indices, then erosion of I by a smaller set E, called the *structuring element*, is the set of all indices at locations x in I such that when E is positioned at x, every element in E is also contained in I.

This type of algorithm is easy to implement with shaped neighborhood iterators because we can use the iterator itself as the structuring element E and move it sequentially through all positions x. The result at x is obtained by checking values in a simple iteration loop through the neighborhood stencil.

We need two iterators, a shaped iterator for the input image and a regular image iterator for writing results to the output image.

```
#include "itkConstShapedNeighborhoodIterator.h"
#include "itkImageRegionIterator.h"
```

Since we are working with binary images in this example, an `unsigned char` pixel type will do. The image and iterator types are defined using the pixel type.

```
  typedef unsigned char PixelType;
  typedef itk::Image< PixelType, 2 >  ImageType;

  typedef itk::ConstShapedNeighborhoodIterator<
                                          ImageType
                                            > ShapedNeighborhoodIteratorType;

  typedef itk::ImageRegionIterator< ImageType> IteratorType;
```

Refer to the examples in Section 11.4.1 or the source code of this example for a description of how to read the input image and allocate a matching output image.

The size of the structuring element is read from the command line and used to define a radius for the shaped neighborhood iterator. Using the method developed in section 11.4.1 to minimize bounds checking, the iterator itself is not initialized until entering the main processing loop.

```
  unsigned int element_radius = ::atoi( argv[3] );
  ShapedNeighborhoodIteratorType::RadiusType radius;
  radius.Fill(element_radius);
```

The face calculator object introduced in Section 11.4.1 is created and used as before.

```
  typedef itk::NeighborhoodAlgorithm::ImageBoundaryFacesCalculator<
                                                ImageType > FaceCalculatorType;

  FaceCalculatorType faceCalculator;
  FaceCalculatorType::FaceListType faceList;
  FaceCalculatorType::FaceListType::iterator fit;

  faceList = faceCalculator( reader->GetOutput(),
                             output->GetRequestedRegion(),
                             radius );
```

Now we initialize some variables and constants.

```
  IteratorType out;

  const PixelType background_value = 0;
  const PixelType foreground_value = 255;
  const float rad = static_cast<float>(element_radius);
```

The outer loop of the algorithm is structured as in previous neighborhood iterator examples. Each region in the face list is processed in turn. As each new region is processed, the input and output iterators are initialized on that region.

The shaped iterator that ranges over the input is our structuring element and its active stencil must be created accordingly. For this example, the structuring element is shaped like a circle of radius `element_radius`. Each of the appropriate neighborhood offsets is activated in the double `for` loop.

```
for ( fit=faceList.begin(); fit != faceList.end(); ++fit)
  {
  ShapedNeighborhoodIteratorType it( radius, reader->GetOutput(), *fit );
  out = IteratorType( output, *fit );

  // Creates a circular structuring element by activating all the pixels less
  // than radius distance from the center of the neighborhood.

  for (float y = -rad; y <= rad; y++)
    {
    for (float x = -rad; x <= rad; x++)
      {
      ShapedNeighborhoodIteratorType::OffsetType off;

      float dis = ::sqrt( x*x + y*y );
      if (dis <= rad)
        {
        off[0] = static_cast<int>(x);
        off[1] = static_cast<int>(y);
        it.ActivateOffset(off);
        }
      }
    }
```

The inner loop, which implements the erosion algorithm, is fairly simple. The `for` loop steps the input and output iterators through their respective images. At each step, the active stencil of the shaped iterator is traversed to determine whether all pixels underneath the stencil contain the foreground value, i.e. are contained within the set I. Note the use of the stencil iterator, `ci`, in performing this check.

```
  // Implements erosion
  for (it.GoToBegin(), out.GoToBegin(); !it.IsAtEnd(); ++it, ++out)
    {
    ShapedNeighborhoodIteratorType::ConstIterator ci;

    bool flag = true;
    for (ci = it.Begin(); ci != it.End(); ci++)
      {
      if (ci.Get() == background_value)
        {
        flag = false;
        break;
```

```
          }
        }
      if (flag == true)
        {
        out.Set(foreground_value);
        }
      else
        {
        out.Set(background_value);
        }
      }
    }
```

The source code for this section can be found in the file
`Examples/Iterators/ShapedNeighborhoodIterators2.cxx`.

The logic of the inner loop can be rewritten to perform dilation. Dilation of the set I by E is the set of all x such that E positioned at x contains at least one element in I.

```
    // Implements dilation
    for (it.GoToBegin(), out.GoToBegin(); !it.IsAtEnd(); ++it, ++out)
      {
      ShapedNeighborhoodIteratorType::ConstIterator ci;

      bool flag = false;
      for (ci = it.Begin(); ci != it.End(); ci++)
        {
        if (ci.Get() != background_value)
          {
          flag = true;
          break;
          }
        }
      if (flag == true)
        {
        out.Set(foreground_value);
        }
      else
        {
        out.Set(background_value);
        }
      }
    }
```

The output image is written and visualized directly as a binary image of `unsigned chars`. Figure 11.10 illustrates some results of erosion and dilation on the image `Examples/Data/BinaryImage.png`. Applying erosion and dilation in sequence effects the morphological operations of opening and closing.

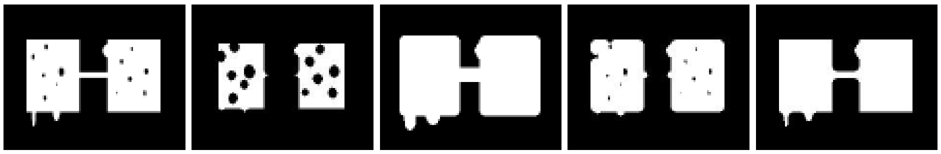


Figure 11.10: The effects of morphological operations on a binary image using a circular structuring element of size 4. From left to right are the original image, erosion, dilation, opening, and closing. The opening operation is erosion of the image followed by dilation. Closing is dilation of the image followed by erosion.

CHAPTER

TWELVE

Image Adaptors

The purpose of an *image adaptor* is to make one image appear like another image, possibly of a different pixel type. A typical example is to take an image of pixel type `unsigned char` and present it as an image of pixel type `float`. The motivation for using image adaptors in this case is to avoid the extra memory resources required by using a casting filter. When we use the `itk::CastImageFilter` for the conversion, the filter creates a memory buffer large enough to store the `float` image. The `float` image requires four times the memory of the original image and contains no useful additional information. Image adaptors, on the other hand, do not require the extra memory as pixels are converted only when they are read using image iterators (see Chapter 11).

Image adaptors are particularly useful when there is infrequent pixel access, since the actual conversion occurs on the fly during the access operation. In such cases the use of image adaptors may reduce overall computation time as well as reduce memory usage. The use of image adaptors, however, can be disadvantageous in some situations. For example, when the downstream filter is executed multiple times, a CastImageFilter will cache its output after the first execution and will not re-execute when the filter downstream is updated. Conversely, an image adaptor will compute the cast every time.

Another application for image adaptors is to perform lightweight pixel-wise operations replacing the need for a filter. In the toolkit, adaptors are defined for many single valued and single parameter functions such as trigonometric, exponential and logarithmic functions. For example,

- `itk::ExpImageAdaptor`
- `itk::SinImageAdaptor`
- `itk::CosImageAdaptor`

The following examples illustrate common applications of image adaptors.

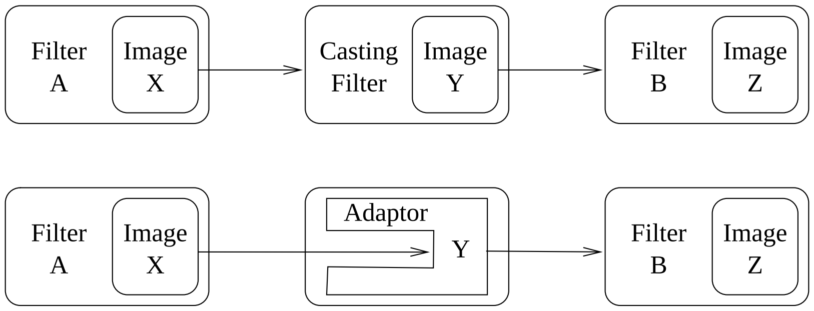


Figure 12.1: The difference between using a CastImageFilter and an ImageAdaptor. ImageAdaptors convert pixel values when they are accessed by iterators. Thus, they do not produces an intermediate image. In the example illustrated by this figure, the *Image Y* is not created by the ImageAdaptor; instead, the image is simulated on the fly each time an iterator from the filter downstream attempts to access the image data.

12.1 Image Casting

The source code for this section can be found in the file
`Examples/DataRepresentation/Image/ImageAdaptor1.cxx`.

This example illustrates how the `itk::ImageAdaptor` can be used to cast an image from one pixel type to another. In particular, we will *adapt* an `unsigned char` image to make it appear as an image of pixel type `float`.

We begin by including the relevant headers.

```
#include "itkImage.h"
#include "itkImageAdaptor.h"
```

First, we need to define a *pixel accessor* class that does the actual conversion. Note that in general, the only valid operations for pixel accessors are those that only require the value of the input pixel. As such, neighborhood type operations are not possible. A pixel accessor must provide methods `Set()` and `Get()`, and define the types of `InternalPixelType` and `ExternalPixelType`. The `InternalPixelType` corresponds to the pixel type of the image to be adapted (`unsigned char` in this example). The `ExternalPixelType` corresponds to the pixel type we wish to emulate with the ImageAdaptor (`float` in this case).

```
class CastPixelAccessor
{
public:
  typedef unsigned char InternalType;
  typedef float         ExternalType;
  static void Set(InternalType & output, const ExternalType & input)
    {
      output = static_cast<InternalType>( input );
    }

  static ExternalType Get( const InternalType & input )
    {
      return static_cast<ExternalType>( input );
    }
};
```

The CastPixelAccessor class simply applies a `static_cast` to the pixel values. We now use this pixel accessor to define the image adaptor type and create an instance using the standard `New()` method.

```
  typedef unsigned char  InputPixelType;
  const   unsigned int   Dimension = 2;
  typedef itk::Image< InputPixelType, Dimension >   ImageType;

  typedef itk::ImageAdaptor< ImageType, CastPixelAccessor > ImageAdaptorType;
  ImageAdaptorType::Pointer adaptor = ImageAdaptorType::New();
```

We also create an image reader templated over the input image type and read the input image from file.

```
  typedef itk::ImageFileReader< ImageType >   ReaderType;
  ReaderType::Pointer reader = ReaderType::New();
```

The output of the reader is then connected as the input to the image adaptor.

```
  adaptor->SetImage( reader->GetOutput() );
```

In the following code, we visit the image using an iterator instantiated using the adapted image type and compute the sum of the pixel values.

```
  typedef itk::ImageRegionIteratorWithIndex< ImageAdaptorType >  IteratorType;
  IteratorType  it( adaptor, adaptor->GetBufferedRegion() );

  double sum = 0.0;
  it.GoToBegin();
  while( !it.IsAtEnd() )
    {
    float value = it.Get();
    sum += value;
    ++it;
    }
```

Although in this example, we are just performing a simple summation, the key concept is that access to pixels is performed as if the pixel is of type `float`. Additionally, it should be noted that the adaptor is used as if it was an actual image and not as a filter. ImageAdaptors conform to the same API as the `itk::Image` class.

12.2 Adapting RGB Images

The source code for this section can be found in the file
`Examples/DataRepresentation/Image/ImageAdaptor2.cxx`.

This example illustrates how to use the `itk::ImageAdaptor` to access the individual components of an RGB image. In this case, we create an ImageAdaptor that will accept a RGB image as input and presents it as a scalar image. The pixel data will be taken directly from the red channel of the original image.

As with the previous example, the bulk of the effort in creating the image adaptor is associated with the definition of the pixel accessor class. In this case, the accessor converts a RGB vector to a scalar containing the red channel component. Note that in the following, we do not need to define the `Set()` method since we only expect the adaptor to be used for reading data from the image.

```
class RedChannelPixelAccessor
{
public:
  typedef itk::RGBPixel<float>   InternalType;
  typedef               float    ExternalType;

  static ExternalType Get( const InternalType & input )
    {
      return static_cast<ExternalType>( input.GetRed() );
    }
};
```

The `Get()` method simply calls the `GetRed()` method defined in the `itk::RGBPixel` class.

Now we use the internal pixel type of the pixel accessor to define the input image type, and then proceed to instantiate the ImageAdaptor type.

```
  typedef RedChannelPixelAccessor::InternalType  InputPixelType;
  const   unsigned int   Dimension = 2;
  typedef itk::Image< InputPixelType, Dimension >   ImageType;

  typedef itk::ImageAdaptor<  ImageType,
                              RedChannelPixelAccessor > ImageAdaptorType;

  ImageAdaptorType::Pointer adaptor = ImageAdaptorType::New();
```

We create an image reader and connect the output to the adaptor as before.

```
typedef itk::ImageFileReader< ImageType >   ReaderType;
ReaderType::Pointer reader = ReaderType::New();

adaptor->SetImage( reader->GetOutput() );
```

We create an itk::RescaleIntensityImageFilter and an itk::ImageFileWriter to rescale the dynamic range of the pixel values and send the extracted channel to an image file. Note that the image type used for the rescaling filter is the `ImageAdaptorType` itself. That is, the adaptor type is used in the same context as an image type.

```
typedef itk::Image< unsigned char, Dimension >   OutputImageType;
typedef itk::RescaleIntensityImageFilter< ImageAdaptorType,
                                          OutputImageType
                                             >   RescalerType;

RescalerType::Pointer rescaler = RescalerType::New();
typedef itk::ImageFileWriter< OutputImageType >   WriterType;
WriterType::Pointer writer = WriterType::New();
```

Now we connect the adaptor as the input to the rescaler and set the parameters for the intensity rescaling.

```
rescaler->SetOutputMinimum(  0  );
rescaler->SetOutputMaximum( 255 );

rescaler->SetInput( adaptor );
writer->SetInput( rescaler->GetOutput() );
```

Finally, we invoke the `Update()` method on the writer and take precautions to catch any exception that may be thrown during the execution of the pipeline.

```
try
  {
  writer->Update();
  }
catch( itk::ExceptionObject & excp )
  {
  std::cerr << "Exception caught " << excp << std::endl;
  return 1;
  }
```

ImageAdaptors for the green and blue channels can easily be implemented by modifying the pixel accessor of the red channel and then using the new pixel accessor for instantiating the type of an image adaptor. The following define a green channel pixel accessor.

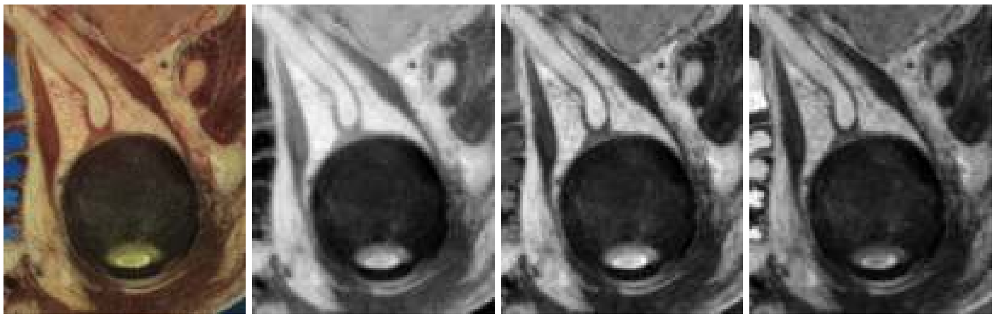


Figure 12.2: Using ImageAdaptor to extract the components of an RGB image. The image on the left is a subregion of the Visible Woman cryogenic data set. The red, green and blue components are shown from left to right as scalar images extracted with an ImageAdaptor.

```
class GreenChannelPixelAccessor
{
public:
  typedef itk::RGBPixel<float>   InternalType;
  typedef               float    ExternalType;

  static ExternalType Get( const InternalType & input )
    {
      return static_cast<ExternalType>( input.GetGreen() );
    }
};
```

A blue channel pixel accessor is similarly defined.

```
class BlueChannelPixelAccessor
{
public:
  typedef itk::RGBPixel<float>   InternalType;
  typedef               float    ExternalType;

  static ExternalType Get( const InternalType & input )
    {
      return static_cast<ExternalType>( input.GetBlue() );
    }
};
```

Figure 12.2 shows the result of extracting the red, green and blue components from a region of the Visible Woman cryogenic data set.

12.3 Adapting Vector Images

The source code for this section can be found in the file
`Examples/DataRepresentation/Image/ImageAdaptor3.cxx`.

This example illustrates the use of `itk::ImageAdaptor` to obtain access to the components of a vector image. Specifically, it shows how to manage pixel accessors containing internal parameters. In this example we create an image of vectors by using a gradient filter. Then, we use an image adaptor to extract one of the components of the vector image. The vector type used by the gradient filter is the `itk::CovariantVector` class.

We start by including the relevant headers.

```
#include "itkCovariantVector.h"
#include "itkGradientRecursiveGaussianImageFilter.h"
```

A pixel accessors class may have internal parameters that affect the operations performed on input pixel data. Image adaptors support parameters in their internal pixel accessor by using the assignment operator. Any pixel accessor which has internal parameters must therefore implement the assignment operator. The following defines a pixel accessor for extracting components from a vector pixel. The `m_Index` member variable is used to select the vector component to be returned.

```
class VectorPixelAccessor
{
public:
  typedef itk::CovariantVector<float,2>   InternalType;
  typedef                      float      ExternalType;

  void operator=( const VectorPixelAccessor & vpa )
    {
      m_Index = vpa.m_Index;
    }
  ExternalType Get( const InternalType & input ) const
    {
      return static_cast<ExternalType>( input[ m_Index ] );
    }
  void SetIndex( unsigned int index )
    {
      m_Index = index;
    }
private:
  unsigned int m_Index;
};
```

The `Get()` method simply returns the *i*-th component of the vector as indicated by the index.

The assignment operator transfers the value of the index member variable from one instance of the pixel accessor to another.

In order to test the pixel accessor, we generate an image of vectors using the itk::GradientRecursiveGaussianImageFilter. This filter produces an output image of itk::CovariantVector pixel type. Covariant vectors are the natural representation for gradients since they are the equivalent of normals to iso-values manifolds.

```
typedef unsigned char  InputPixelType;
const   unsigned int   Dimension = 2;
typedef itk::Image< InputPixelType,  Dimension  >   InputImageType;
typedef itk::CovariantVector< float, Dimension  >   VectorPixelType;
typedef itk::Image< VectorPixelType, Dimension  >   VectorImageType;
typedef itk::GradientRecursiveGaussianImageFilter< InputImageType,
                                      VectorImageType> GradientFilterType;

GradientFilterType::Pointer gradient = GradientFilterType::New();
```

We instantiate the ImageAdaptor using the vector image type as the first template parameter and the pixel accessor as the second template parameter.

```
typedef itk::ImageAdaptor<  VectorImageType,
                            VectorPixelAccessor > ImageAdaptorType;

ImageAdaptorType::Pointer adaptor = ImageAdaptorType::New();
```

The index of the component to be extracted is specified from the command line. In the following, we create the accessor, set the index and connect the accessor to the image adaptor using the `SetPixelAccessor()` method.

```
VectorPixelAccessor  accessor;
accessor.SetIndex( atoi( argv[3] ) );
adaptor->SetPixelAccessor( accessor );
```

We create a reader to load the image specified from the command line and pass its output as the input to the gradient filter.

```
typedef itk::ImageFileReader< InputImageType >   ReaderType;
ReaderType::Pointer reader = ReaderType::New();
gradient->SetInput( reader->GetOutput() );

reader->SetFileName( argv[1] );
gradient->Update();
```

We now connect the output of the gradient filter as input to the image adaptor. The adaptor emulates a scalar image whose pixel values are taken from the selected component of the vector image.

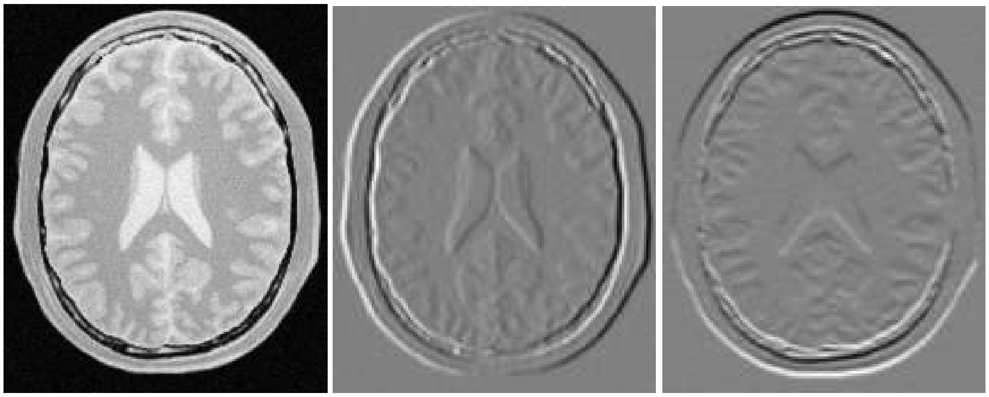


Figure 12.3: Using ImageAdaptor to access components of a vector image. The input image on the left was passed through a gradient image filter and the two components of the resulting vector image were extracted using an image adaptor.

```
  adaptor->SetImage( gradient->GetOutput() );
```

As in the previous example, we rescale the scalar image before writing the image out to file. Figure 12.3 shows the result of applying the example code for extracting both components of a two dimensional gradient.

12.4 Adaptors for Simple Computation

The source code for this section can be found in the file
`Examples/DataRepresentation/Image/ImageAdaptor4.cxx`.

Image adaptors can also be used to perform simple pixel-wise computations on image data. The following example illustrates how to use the `itk::ImageAdaptor` for image thresholding.

A pixel accessor for image thresholding requires that the accessor maintain the threshold value. Therefore, it must also implement the assignment operator to set this internal parameter.

```
class ThresholdingPixelAccessor
{
public:
  typedef unsigned char      InternalType;
  typedef unsigned char      ExternalType;

  ExternalType Get( const InternalType & input ) const
    {
      return (input > m_Threshold) ? 1 : 0;
    }
  void SetThreshold( const InternalType threshold )
    {
      m_Threshold = threshold;
    }

  void operator=( const ThresholdingPixelAccessor & vpa )
    {
      m_Threshold = vpa.m_Threshold;
    }
private:
  InternalType m_Threshold;
};
```

The `Get()` method returns one if the input pixel is above the threshold and zero otherwise. The assignment operator transfers the value of the threshold member variable from one instance of the pixel accessor to another.

To create an image adaptor, we first instantiate an image type whose pixel type is the same as the internal pixel type of the pixel accessor.

```
  typedef ThresholdingPixelAccessor::InternalType     PixelType;
  const   unsigned int   Dimension = 2;
  typedef itk::Image< PixelType,  Dimension >   ImageType;
```

We instantiate the ImageAdaptor using the image type as the first template parameter and the pixel accessor as the second template parameter.

```
  typedef itk::ImageAdaptor<  ImageType,
                              ThresholdingPixelAccessor > ImageAdaptorType;

  ImageAdaptorType::Pointer adaptor = ImageAdaptorType::New();
```

The threshold value is set from the command line. A threshold pixel accessor is created and connected to the image adaptor in the same manner as in the previous example.

```
  ThresholdingPixelAccessor  accessor;
  accessor.SetThreshold( atoi( argv[3] ) );
  adaptor->SetPixelAccessor( accessor );
```

We create a reader to load the input image and connect the output of the reader as the input to the adaptor.

```
  typedef itk::ImageFileReader< ImageType >   ReaderType;
  ReaderType::Pointer reader = ReaderType::New();
```

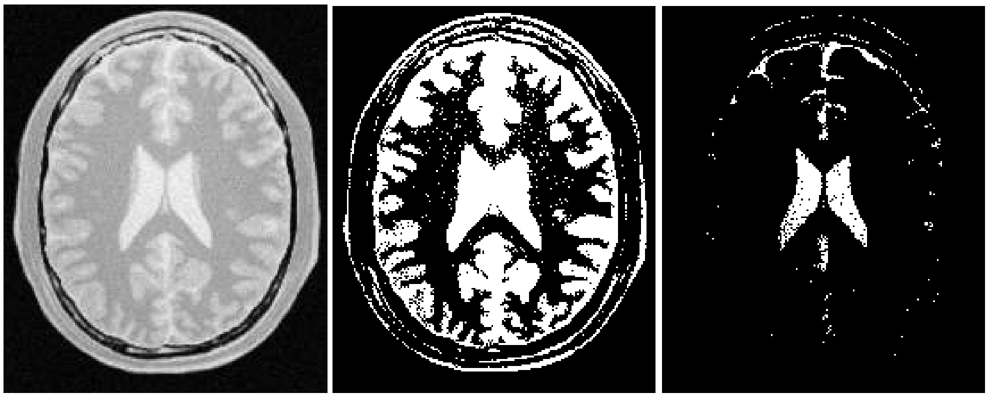


Figure 12.4: Using ImageAdaptor to perform a simple image computation. An ImageAdaptor is used to perform binary thresholding on the input image on the left. The center image was created using a threshold of 180, while the image on the right corresponds to a threshold of 220.

```
  reader->SetFileName( argv[1] );
  reader->Update();

  adaptor->SetImage( reader->GetOutput() );
```

As before, we rescale the emulated scalar image before writing it out to file. Figure 12.4 illustrates the result of applying the thresholding adaptor to a typical gray scale image using two different threshold values. Note that the same effect could have been achieved by using the `itk::BinaryThresholdImageFilter` but at the price of holding an extra copy of the image in memory.

12.5 Adaptors and Writers

Image adaptors will not behave correctly when connected directly to a writer. The reason is that writers tend to get direct access to the image buffer from their input, since image adaptors do not have a real buffer their behavior in this circumstances is incorrect. You should avoid instantiating the `ImageFileWriter` or the `ImageSeriesWriter` over an image adaptor type.

CHAPTER

THIRTEEN

How To Write A Filter

This purpose of this chapter is help developers create their own filter (process object). This chapter is divided into four major parts. An initial definition of terms is followed by an overview of the filter creation process. Next, data streaming is discussed. The way data is streamed in ITK must be understood in order to write correct filters. Finally, a section on multithreading describes what you must do in order to take advantage of shared memory parallel processing.

13.1 Terminology

The following is some basic terminology for the discussion that follows. Chapter 3 provides additional background information.

- The **data processing pipeline** is a directed graph of **process** and **data objects**. The pipeline inputs, operators on, and outputs data.
- A **filter**, or **process object**, has one or more inputs, and one or more outputs.
- A **source**, or source process object, initiates the data processing pipeline, and has one or more outputs.
- A **mapper**, or mapper process object, terminates the data processing pipeline. The mapper has one or more outputs, and may write data to disk, interface with a display system, or interface to any other system.
- A **data object** represents and provides access to data. In ITK, the data object (ITK class `itk::DataObject`) is typically of type `itk::Image` or `itk::Mesh`.
- A **region** (ITK class `itk::Region`) represents a piece, or subset of the entire data set.
- An **image region** (ITK class `itk::ImageRegion`) represents a structured portion of data. ImageRegion is implemented using the `itk::Index` and `itk::Size` classes

- A **mesh region** (ITK class `itk::MeshRegion`) represents an unstructured portion of data.
- The **LargestPossibleRegion** is the theoretical single, largest piece (region) that could represent the entire dataset. The LargestPossibleRegion is used in the system as the measure of the largest possible data size.
- The **BufferedRegion** is a contiguous block of memory that is less than or equal to in size to the LargestPossibleRegion. The buffered region is what has actually been allocated by a filter to hold its output.
- The **RequestedRegion** is the piece of the dataset that a filter is required to produce. The RequestedRegion is less than or equal in size to the BufferedRegion. The RequestedRegion may differ in size from the BufferedRegion due to performance reasons. The RequestedRegion may be set by a user, or by an application that needs just a portion of the data.
- The **modified time** (represented by ITK class `itk::TimeStamp`) is a monotonically increasing integer value that characterizes a point in time when an object was last modified.

- **Downstream** is the direction of dataflow, from sources to mappers.
- **Upstream** is the opposite of downstream, from mappers to sources.
- The **pipeline modified time** for a particular data object is the maximum modified time of all upstream data objects and process objects.
- The term **information** refers to metadata that characterizes data. For example, index and dimensions are information characterizing an image region.

13.2 Overview of Filter Creation

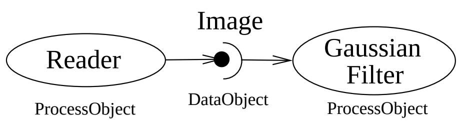


Figure 13.1: Relationship between DataObject and ProcessObject.

Filters are defined with respect to the type of data they input (if any), and the type of data they output (if any). The key to writing a ITK filter is to identify the number and types of input and output. Having done so, there are often superclasses that simplify this task via class derivation. For example, most filters in ITK take a single image as input, and produce a single image on output. The superclass `itk::ImageToImageFilter` is a convenience class that provide most of the functionality needed for such a filter.

Some common base classes for new filters include:

- `ImageToImageFilter`: the most common filter base for segmentation algorithms. Takes an image and produces a new image, by default of the same dimensions. Override `GenerateOutputInformation` to produce a different size.
- `UnaryFunctorImageFilter`: used when defining a filter that applies a function to an image.
- `BinaryFunctorImageFilter`: used when defining a filter that applies an operation to two images.
- `ImageFunction`: a functor that can be applied to an image, evaluating $f(x)$ at each point in the image.
- `MeshToMeshFilter`: a filter that transforms meshes, such as tessellation, polygon reduction, and so on.
- `LightObject`: abstract base for filters that don't fit well anywhere else in the class hierarchy. Also useful for "calculator" filters; ie. a sink filter that takes an input and calculates a result which is retrieved using a `Get()` method.

Once the appropriate superclass is identified, the filter writer implements the class defining the methods required by most all ITK objects: `New()`, `PrintSelf()`, and protected constructor, copy constructor, delete, and operator=, and so on. Also, don't forget standard typedefs like `Self`, `Superclass`, `Pointer`, and `ConstPointer`. Then the filter writer can focus on the most important parts of the implementation: defining the API, data members, and other implementation details of the algorithm. In particular, the filter writer will have to implement either a `GenerateData()` (non-threaded) or `ThreadedGenerateData()` method. (See Section 3.2.7 for an overview of multi-threading in ITK.)

An important note: the GenerateData() method is required to allocate memory for the output. The ThreadedGenerateData() method is not. In default implementation (see `itk::ImageSource`, a superclass of `itk::ImageToImageFilter`) `GenerateData()` allocates memory and then invokes `ThreadedGenerateData()`.

One of the most important decisions that the developer must make is whether the filter can stream data; that is, process just a portion of the input to produce a portion of the output. Often superclass behavior works well: if the filter processes the input using single pixel access, then the default behavior is adequate. If not, then the user may have to a) find a more specialized superclass to derive from, or b) override one or more methods that control how the filter operates during pipeline execution. The next section describes these methods.

13.3 Streaming Large Data

The data associated with multi-dimensional images is large and becoming larger. This trend is due to advances in scanning resolution, as well as increases in computing capability. Any

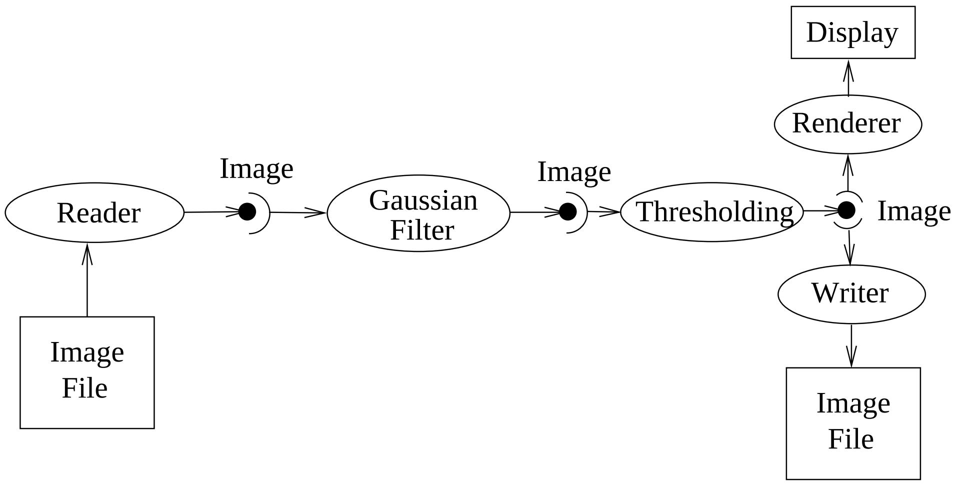


Figure 13.2: The Data Pipeline

practical segmentation and registration software system must address this fact in order to be useful in application. ITK addresses this problem via its data streaming facility.

In ITK, streaming is the process of dividing data into pieces, or regions, and then processing this data through the data pipeline. Recall that the pipeline consists of process objects that generate data objects, connected into a pipeline topology. The input to a process object is a data object (unless the process initiates the pipeline and then it is a source process object). These data objects in turn are consumed by other process objects, and so on, until a directed graph of data flow is constructed. Eventually the pipeline is terminated by one or more mappers, that may write data to storage, or interface with a graphics or other system. This is illustrated in figures 13.1 and 13.2.

A significant benefit of this architecture is that the relatively complex process of managing pipeline execution is designed into the system. This means that keeping the pipeline up to date, executing only those portions of the pipeline that have changed, multithreading execution, managing memory allocation, and streaming is all built into the architecture. However, these features do introduce complexity into the system, the bulk of which is seen by class developers. The purpose of this chapter is to describe the pipeline execution process in detail, with a focus on data streaming.

13.3.1 Overview of Pipeline Execution

The pipeline execution process performs several important functions.

1. It determines which filters, in a pipeline of filters, need to execute. This prevents redundant execution and minimizes overall execution time.

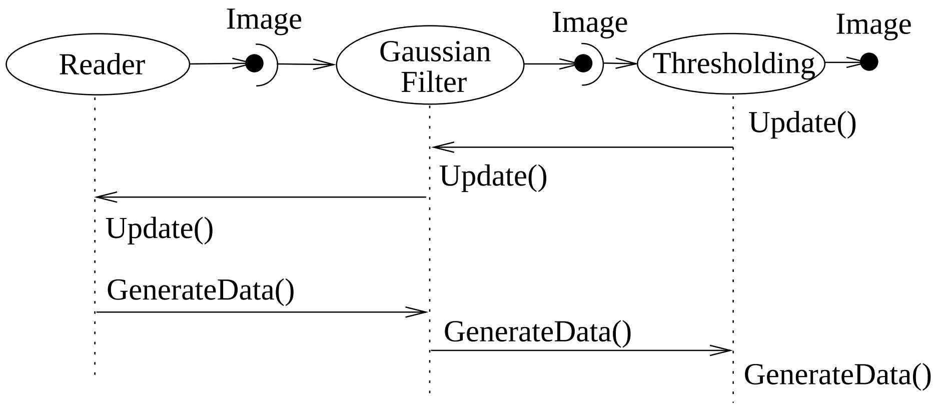


Figure 13.3: Sequence of the Data Pipeline updating mechanism

2. It initializes the (filter's) output data objects, preparing them for new data. In addition, it determines how much memory each filter must allocate for its output, and allocates it.
3. The execution process determines how much data a filter must process in order to produce an output of sufficient size for downstream filters; it also takes into account any limits on memory or special filter requirements. Other factors include the size of data processing kernels, that affect how much data input data (extra padding) is required.
4. It subdivides data into subpieces for multithreading. (Note that the division of data into subpieces is exactly same problem as dividing data into pieces for streaming; hence multithreading comes for free as part of the streaming architecture.)
5. It may free (or release) output data if filters no longer need it to compute, and the user requests that data is to be released. (Note: a filter's output data object may be considered a "cache". If the cache is allowed to remain (`ReleaseDataFlagOff()`) between pipeline execution, and the filter, or the input to the filter, never changes, then process objects downstream of the filter just reuse the filter's cache to re-execute.)

To perform these functions, the execution process negotiates with the filters that define the pipeline. Only each filter can know how much data is required on input to produce a particular output. For example, a shrink filter with a shrink factor of two requires an image twice as large (in terms of its x-y dimensions) on input to produce a particular size output. An image convolution filter would require extra input (boundary padding) depending on the size of the convolution kernel. Some filters require the entire input to produce an output (for example, a histogram), and have the option of requesting the entire input. (In this case streaming does not work unless the developer creates a filter that can request multiple pieces, caching state between each piece to assemble the final output.)

Ultimately the negotiation process is controlled by the request for data of a particular size (i.e., region). It may be that the user asks to process a region of interest within a large image, or that memory limitations result in processing the data in several pieces. For example, an application may compute the memory required by a pipeline, and then use `itk::StreamingImageFilter` to break the data processing into several pieces. The data request is propagated through the pipeline in the upstream direction, and the negotiation process configures each filter to produce output data of a particular size.

The secret to creating a streaming filter is to understand how this negotiation process works, and how to override its default behavior by using the appropriate virtual functions defined in `itk::ProcessObject`. The next section describes the specifics of these methods, and when to override them. Examples are provided along the way to illustrate concepts.

13.3.2 Details of Pipeline Execution

Typically pipeline execution is initiated when a process object receives the `ProcessObject::Update()` method invocation. This method is simply delegated to the output of the filter, invoking the `DataObject::Update()` method. Note that this behavior is typical of the interaction between ProcessObject and DataObject: a method invoked on one is eventually delegated to the other. In this way the data request from the pipeline is propagated upstream, initiating data flow that returns downstream.

The `DataObject::Update()` method in turn invokes three other methods:

- `DataObject::UpdateOutputInformation()`
- `DataObject::PropagateRequestedRegion()`
- `DataObject::UpdateOutputData()`

UpdateOutputInformation()

The `UpdateOutputInformation()` method determines the pipeline modified time. It may set the RequestedRegion and the LargestPossibleRegion depending on how the filters are configured. (The RequestedRegion is set to process all the data, i.e., the LargestPossibleRegion, if it has not been set.) The UpdateOutputInformation() propagates upstream through the entire pipeline and terminates at the sources.

During `UpdateOutputInformation()`, filters have a chance to override the `ProcessObject::GenerateOutputInformation()` method (`GenerateOutputInformation()` is invoked by `UpdateOutputInformation()`). The default behavior is for the `GenerateOutputInformation()` to copy the metadata describing the input to the output (via `DataObject::CopyInformation()`). Remember, information is metadata describing the output, such as the origin, spacing, and LargestPossibleRegion (i.e., largest possible size) of an image.

A good example of this behavior is `itk::ShrinkImageFilter`. This filter takes an input image and shrinks it by some integral value. The result is that the spacing and LargestPossibleRegion of the output will be different to that of the input. Thus, `GenerateOutputInformation()` is overloaded.

PropagateRequestedRegion()

The `PropagateRequestedRegion()` call propagates upstream to satisfy a data request. In typical application this data request is usually the LargestPossibleRegion, but if streaming is necessary, or the user is interested in updating just a portion of the data, the RequestedRegion may be any valid region within the LargestPossibleRegion.

The function of `PropagateRequestedRegion()` is, given a request for data (the amount is specified by RequestedRegion), propagate upstream configuring the filter's input and output process object's to the correct size. Eventually, this means configuring the BufferedRegion, that is the amount of data actually allocated.

The reason for the buffered region is this: the output of a filter may be consumed by more than one downstream filter. If these consumers each request different amounts of input (say due to kernel requirements or other padding needs), then the upstream, generating filter produces the data to satisfy both consumers, that may mean it produces more data than one of the consumers needs.

The `ProcessObject::PropagateRequestedRegion()` method invokes three methods that the filter developer may choose to overload.

- `EnlargeOutputRequestedRegion(DataObject *output)` gives the (filter) subclass a chance to indicate that it will provide more data than required for the output. This can happen, for example, when a source can only produce the whole output (i.e., the LargestPossibleRegion).

- `GenerateOutputRequestedRegion(DataObject *output)` gives the subclass a chance to define how to set the requested regions for each of its outputs, given this output's requested region. The default implementation is to make all the output requested regions the same. A subclass may need to override this method if each output is a different resolution. This method is only overridden if a filter has multiple outputs.

- `GenerateInputRequestedRegion()` gives the subclass a chance to request a larger requested region on the inputs. This is necessary when, for example, a filter requires more data at the "internal" boundaries to produce the boundary values - due to kernel operations or other region boundary effects.

`itk::RGBGibbsPriorFilter` is an example of a filter that needs to invoke `EnlargeOutputRequestedRegion()`. The designer of this filter decided that the filter should operate on all the data. Note that a subtle interplay between this method and `GenerateInputRequestedRegion()` is occurring here. The default behavior of `GenerateInputRequestedRegion()` (at least for `itk::ImageToImageFilter`) is to set the input RequestedRegion to the output's ReqestedRegion. Hence, by overriding the method `EnlargeOutputRequestedRegion()` to set the output to the LargestPossibleRegion, effectively sets the input to this filter to the LargestPossibleRegion (and probably causing all upstream filters to process their LargestPossibleRegion as well. This means that the filter, and therefore the pipeline, does not stream. This could be fixed by reimplementing the filter with the notion of streaming built in to the algorithm.)

`itk::GradientMagnitudeImageFilter` is an example of a filter that needs to invoke `GenerateInputRequestedRegion()`. It needs a larger input requested region because a kernel is required to compute the gradient at a pixel. Hence the input needs to be "padded out" so the filter has enough data to compute the gradient at each output pixel.

UpdateOutputData()

`UpdateOutputData()` is the third and final method as a result of the `Update()` method. The purpose of this method is to determine whether a particular filter needs to execute in order to bring its output up to date. (A filter executes when its `GenerateData()` method is invoked.) Filter execution occurs when a) the filter is modified as a result of modifying an instance variable; b) the input to the filter changes; c) the input data has been released; or d) an invalid RequestedRegion was set previously and the filter did not produce data. Filters execute in order in the downstream direction. Once a filter executes, all filters downstream of it must also execute.

`DataObject::UpdateOutputData()` is delegated to the DataObject's source (i.e., the ProcessObject that generated it) only if the DataObject needs to be updated. A comparison of modified time, pipeline time, release data flag, and valid requested region is made. If any one of these conditions indicate that the data needs regeneration, then the source's `ProcessObject::UpdateOutputData()` is invoked. These calls are made recursively up the pipeline until a source filter object is encountered, or the pipeline is determined to be up to date and valid. At this point, the recursion unrolls, and the execution of the filter proceeds. (This means that the output data is initialized, StartEvent is invoked, the filters `GenerateData()` is called, EndEvent is invoked, and input data to this filter may be released, if requested. In addition, this filter's InformationTime is updated to the current time.)

The developer will never override `UpdateOutputData()`. The developer need only write the `GenerateData()` method (non-threaded) or `ThreadedGenerateData()` method. A discussion of threading follows in the next section.

13.4 Threaded Filter Execution

Filters that can process data in pieces can typically multi-process using the data parallel, shared memory implementation built into the pipeline execution process. To create a multithreaded filter, simply define and implement a `ThreadedGenerateData()` method. For example, a itk::ImageToImageFilter would create the method:

```
    void ThreadedGenerateData(const OutputImageRegionType&
                              outputRegionForThread, int threadId)
```

The key to threading is to generate output for the output region given (as the first parameter in the argument list above). In ITK, this is simple to do because an output iterator can be created using the region provided. Hence the output can be iterated over, accessing the corresponding input pixels as necessary to compute the value of the output pixel.

Multi-threading requires caution when performing I/O (including using cout or cerr) or invoking events. A safe practice is to allow only thread id zero to perform I/O or generate events. (The thread id is passed as argument into ThreadedGenerateData()). If more than one thread tries to write to the same place at the same time, the program can behave badly, and possibly even deadlock or crash.

13.5 Filter Conventions

In order to fully participate in the ITK pipeline, filters are expected to follow certain conventions, and provide certain interfaces. This section describes the minimum requirements for a filter to integrate into the ITK framework.

The class declaration for a filter should include the macro ITK_EXPORT, so that on certain platforms an export declaration can be included.

A filter should define public types for the class itself (Self) and its Superclass, and const and non-const smart pointers, thus:

```
  typedef ExampleImageFilter                    Self;
  typedef ImageToImageFilter<TImage,TImage> Superclass;
  typedef SmartPointer<Self>                    Pointer;
  typedef SmartPointer<const Self>              ConstPointer;
```

The Pointer type is particularly useful, as it is a smart pointer that will be used by all client code to hold a reference-counted instantiation of the filter.

Once the above types have been defined, you can use the following convenience macros, which permit your filter to participate in the object factory mechanism, and to be created using the canonical ::New():

```
  /** Method for creation through the object factory. */
  itkNewMacro(Self);

  /** Run-time type information (and related methods). */
  itkTypeMacro(ExampleImageFilter, ImageToImageFilter);
```

The default constructor should be `protected`, and provide sensible defaults (usually zero) for all parameters. The copy constructor and assignment operator should be declared `private` and not implemented, to prevent instantiating the filter without the factory methods (above).

Finally, the template implementation code (in the `.txx` file) should be included, bracketed by a test for manual instantiation, thus:

```
#ifndef ITK_MANUAL_INSTANTIATION
#include "itkExampleFilter.txx"
#endif
```

13.5.1 Optional

A filter can be printed to an `std::ostream` (such as `std::cout`) by implementing the following method:

```
  void PrintSelf( std::ostream& os, Indent indent ) const;
```

and writing the name-value pairs of the filter parameters to the supplied output stream. This is particularly useful for debugging.

13.5.2 Useful Macros

Many convenience macros are provided by ITK, to simplify filter coding. Some of these are described below:

itkStaticConstMacro Declares a static variable of the given type, with the specified initial value.

itkGetMacro Defines an accessor method for the specified scalar data member. The convention is for data members to have a prefix of `m_`.

itkSetMacro Defines a mutator method for the specified scalar data member, of the supplied type. This will automatically set the `Modified` flag, so the filter stage will be executed on the next `Update()`.

itkBooleanMacro Defines a pair of `OnFlag` and `OffFlag` methods for a boolean variable `m_Flag`.

itkGetObjectMacro, itkSetObjectMacro Defines an accessor and mutator for an ITK object. The Get form returns a smart pointer to the object.

Much more useful information can be learned from browsing the source in `Code/Common/itkMacro.h` and for the `itk::Object` and `itk::LightObject` classes.

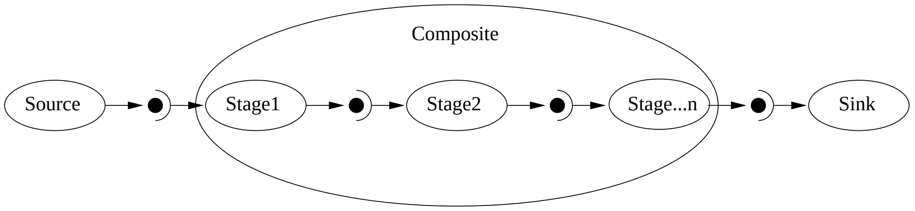


Figure 13.4: A Composite filter encapsulates a number of other filters.

13.6 How To Write A Composite Filter

In general, most ITK filters implement one particular algorithm, whether it be image filtering, an information metric, or a segmentation algorithm. In the previous section, we saw how to write new filters from scratch. However, it is often very useful to be able to make a new filter by combining two or more existing filters, which can then be used as a building block in a complex pipeline. This approach follows the Composite pattern [28], whereby the composite filter itself behaves just as a regular filter, providing its own (potentially higher level) interface and using other filters (whose detail is hidden to users of the class) for the implementation. This composite structure is shown in Figure 13.4, where the various `Stage-n` filters are combined into one by the `Composite` filter. The `Source` and `Sink` filters only see the interface published by the `Composite`. Using the Composite pattern, a composite filter can encapsulate a pipeline of arbitrary complexity. These can in turn be nested inside other pipelines.

13.6.1 Implementing a Composite Filter

There are a few considerations to take into account when implementing a composite filter. All the usual requirements for filters apply (as discussed above), but the following guidelines should be considered:

1. The template arguments it takes must be sufficient to instantiate all of the component filters. Each component filter needs a type supplied by either the implementor or the enclosing class. For example, an `ImageToImageFilter` normally takes an input and output image type (which may be the same). But if the output of the composite filter is a classified image, we need to either decide on the output type inside the composite filter, or restrict the choices of the user when she/he instantiates the filter.
2. The types of the component filters should be declared in the header, preferably with `protected` visibility. This is because the internal structure normally should not be visible to users of the class, but should be to descendent classes that may need to modify or customize the behavior.
3. The component filters should be private data members of the composite class, as in `FilterType::Pointer`.

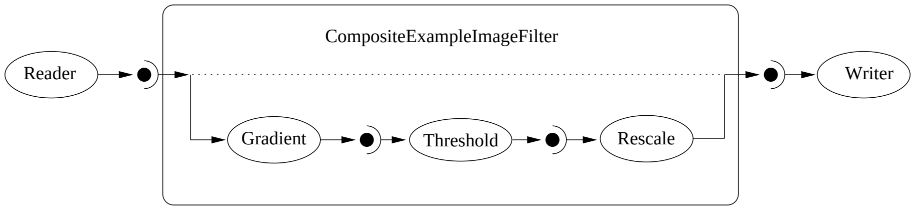


Figure 13.5: Example of a typical composite filter. Note that the output of the last filter in the internal pipeline must be grafted into the output of the composite filter.

4. The default constructor should build the pipeline by creating the stages and connect them together, along with any default parameter settings, as appropriate.
5. The input and output of the composite filter need to be grafted on to the head and tail (respectively) of the component filters.

This grafting process is illustrated in Figure 13.5.

13.6.2 A Simple Example

The source code for this section can be found in the file
`Examples/Filtering/CompositeFilterExample.cxx`.

The composite filter we will build combines three filters: a gradient magnitude operator, which will calculate the first-order derivative of the image; a thresholding step to select edges over a given strength; and finally a rescaling filter, to ensure the resulting image data is visible by scaling the intensity to the full spectrum of the output image type.

Since this filter takes an image and produces another image (of identical type), we will specialize the ImageToImageFilter:

```
#include "itkImageToImageFilter.h"
```

Next we include headers for the component filters:

```
#include "itkGradientMagnitudeImageFilter.h"
#include "itkThresholdImageFilter.h"
#include "itkRescaleIntensityImageFilter.h"
```

Now we can declare the filter itself. It is within the ITK namespace, and we decide to make it use the same image type for both input and output, thus the template declaration needs only one parameter. Deriving from `ImageToImageFilter` provides default behavior for several important aspects, notably allocating the output image (and making it the same dimensions as the input).

```
namespace itk {

template <class TImageType>
class ITK_EXPORT CompositeExampleImageFilter :
    public ImageToImageFilter<TImageType, TImageType>
{
public:
```

Next we have the standard declarations, used for object creation with the object factory:

```
  typedef CompositeExampleImageFilter               Self;
  typedef ImageToImageFilter<TImageType,TImageType> Superclass;
  typedef SmartPointer<Self>                        Pointer;
  typedef SmartPointer<const Self>                  ConstPointer;
```

Here we declare an alias (to save typing) for the image's pixel type, which determines the type of the threshold value. We then use the convenience macros to define the Get and Set methods for this parameter.

```
  typedef typename TImageType::PixelType PixelType;

  itkGetMacro( Threshold, PixelType);
  itkSetMacro( Threshold, PixelType);
```

Now we can declare the component filter types, templated over the enclosing image type:

```
protected:

  typedef ThresholdImageFilter< TImageType > ThresholdType;
  typedef GradientMagnitudeImageFilter< TImageType, TImageType > GradientType;
  typedef RescaleIntensityImageFilter< TImageType, TImageType > RescalerType;
```

The component filters are declared as data members, all using the smart pointer types.

```
  typename GradientType::Pointer     m_GradientFilter;
  typename ThresholdType::Pointer    m_ThresholdFilter;
  typename RescalerType::Pointer     m_RescaleFilter;

  PixelType m_Threshold;
};

} /* namespace itk */
```

The constructor sets up the pipeline, which involves creating the stages, connecting them together, and setting default parameters.

```
template <class TImageType>
CompositeExampleImageFilter<TImageType>
::CompositeExampleImageFilter()
{
  m_GradientFilter = GradientType::New();
  m_ThresholdFilter = ThresholdType::New();
  m_RescaleFilter = RescalerType::New();

  m_ThresholdFilter->SetInput( m_GradientFilter->GetOutput() );
  m_RescaleFilter->SetInput( m_ThresholdFilter->GetOutput() );

  m_Threshold = 1;

  m_RescaleFilter->SetOutputMinimum(NumericTraits<PixelType>::NonpositiveMin());
  m_RescaleFilter->SetOutputMaximum(NumericTraits<PixelType>::max());
}
```

The `GenerateData()` is where the composite magic happens. First, we connect the first component filter to the inputs of the composite filter (the actual input, supplied by the upstream stage). Then we graft the output of the last stage onto the output of the composite, which ensures the filter regions are updated. We force the composite pipeline to be processed by calling `Update()` on the final stage, then graft the output back onto the output of the enclosing filter, so it has the result available to the downstream filter.

```
template <class TImageType>
void
CompositeExampleImageFilter<TImageType>::
GenerateData()
{
  m_GradientFilter->SetInput( this->GetInput() );

  m_ThresholdFilter->ThresholdBelow( this->m_Threshold );

  m_RescaleFilter->GraftOutput( this->GetOutput() );
  m_RescaleFilter->Update();
  this->GraftOutput( m_RescaleFilter->GetOutput() );
}
```

Finally we define the `PrintSelf` method, which (by convention) prints the filter parameters. Note how it invokes the superclass to print itself first, and also how the indentation prefixes each line.

```
template <class TImageType>
void
CompositeExampleImageFilter<TImageType>::
PrintSelf( std::ostream& os, Indent indent ) const
{
  Superclass::PrintSelf(os,indent);

  os
    << indent << "Threshold:" << this->m_Threshold
    << std::endl;
}

} /* end namespace itk */
```

It is important to note that in the above example, none of the internal details of the pipeline were exposed to users of the class. The interface consisted of the Threshold parameter (which happened to change the value in the component filter) and the regular ImageToImageFilter interface. This example pipeline is illustrated in Figure 13.5.

CHAPTER

FOURTEEN

Software Process

An outstanding feature of ITK is the software process used to develop, maintain and test the toolkit. The Insight Toolkit software continues to evolve rapidly due to the efforts of developers and users located around the world, so the software process is essential to maintaining its quality. If you are planning to contribute to ITK, or use the CVS source code repository, you need to know something about this process (see 1.3.1 on page 4 to learn more about obtaining ITK using CVS). This information will help you know when and how to update and work with the software as it changes. The following sections describe key elements of the process.

14.1 CVS Source Code Repository

The Concurrent Versions System (CVS) is a tool for version control [27]. It is a very valuable resource for software projects involving multiple developers. The primary purpose of CVS is to keep track of changes to software. CVS date and version stamps every addition to files in the repository. Additionally, a user may set a tag to mark a particular of the whole software. Thus, it is possible to return to a particular state or point of time whenever desired. The differences between any two points is represented by a “diff” file, that is a compact, incremental representation of change. CVS supports concurrent development so that two developers can edit the same file at the same time, that are then (usually) merged together without incident (and marked if there is a conflict). In addition, branches off of the main development trunk provide parallel development of software.

Developers and users can check out the software from the CVS repository. When developers introduce changes in the system, CVS facilitates to update the local copies of other developers and users by downloading only the differences between their local copy and the version on the repository. This is an important advantage for those who are interested in keeping up to date with the leading edge of the toolkit. Bug fixes can be obtained in this way as soon as they have been checked into the system.

ITK source code, data, and examples are maintained in a CVS repository. The principal advantage of a system like CVS is that it frees developers to try new ideas and introduce changes without fear of losing a previous working version of the software. It also provides a simple way to incrementally update code as new features are added to the repository.

14.2 DART Regression Testing System

One of the unique features of the ITK software process is its use of the DART regression testing system (`http://public.kitware.com/Dart`). In a nutshell, what DART does is to provide quantifiable feedback to developers as they check in new code and make changes. The feedback consists of the results of a variety of tests, and the results are posted on a publicly-accessible Web page (to which we refer as a *dashboard*) as shown in Figure 14.1. The most recent dashboard is accessible from `http://www.itk.org/HTML/Testing.htm`). Since all users and developers of ITK can view the Web page, the DART dashboard serves as a vehicle for developer communication, especially when new additions to the software is found to be faulty. The dashboard should be consulted before considering updating software via CVS.

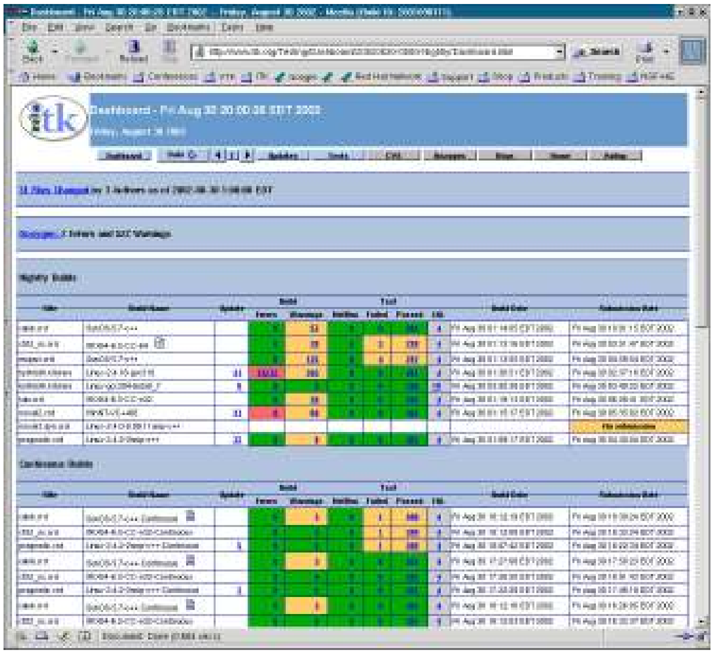


Figure 14.1: On-line presentation of the quality dashboard generated by DART.

Note that DART is independent of ITK and can be used to manage quality control for any software project. It is itself an open-source package and can be obtained from

`http://public.kitware.com/Dart/HTML/Index.shtml`

DART supports a variety of test types. These include the following.

Compilation. All source and test code is compiled and linked. Any resulting errors and warnings are reported on the dashboard.

Regression. Some ITK tests produce images as output. Testing requires comparing each tests output against a valid baseline image. If the images match then the test passes. The comparison must be performed carefully since many 3D graphics systems (e.g., OpenGL) produce slightly different results on different platforms.

Memory. Problems relating to memory such as leaks, uninitialized memory reads, and reads/ writes beyond allocated space can cause unexpected results and program crashes. ITK checks run-time memory access and management using Purify, a commercial package produced by Rational. (Other memory checking programs will be added in the future.)

PrintSelf. All classes in ITK are expected to print out all their instance variables (i.e., those with associated Set and Get methods) correctly. This test checks to make sure that this is the case.

Unit. Each class in ITK should have a corresponding unit test where the class functionalities are exercised and quantitatively compared against expected results. These tests are typically written by the class developer and should endeavor to cover all lines of code including `Set/Get` methods and error handling.

Coverage. There is a saying among ITK developers: *If it isn't covered, then it's broke.* What this means is that code that is not executed during testing is likely to be wrong. The coverage tests identify lines that are not executed in the Insight Toolkit test suite, reporting a total percentage covered at the end of the test. While it is nearly impossible to bring the coverage to 100% because of error handling code and similar constructs that are rarely encountered in practice, the coverage numbers should be 75% or higher. Code that is not covered well enough requires additional tests.

Figure 14.1 shows the top-level dashboard web page. Each row in the dashboard corresponds to a particular platform (hardware + operating system + compiler). The data on the row indicates the number of compile errors and warnings as well as the results of running hundreds of small test programs. In this way the toolkit is tested both at compile time and run time.

When a user or developer decides to update ITK source code from CVS it is important to first verify that the current dashboard is in good shape. This can be rapidly judged by the general coloration of the dashboard. A green state means that the software is building correctly and it is a good day to start with ITK or to get an upgrade. A red state, on the other hand, is an indication of instability on the system and hence users should refrain from checking out or upgrading the source code.

Another nice feature of DART is that it maintains a history of changes to the source code (by coordinating with CVS) and summarizes the changes as part of the dashboard. This is useful for tracking problems and keeping up to date with new additions to ITK.

14.3 Working The Process

The ITK software process functions across three cycles—the continuous cycle, the daily cycle, and the release cycle.

The continuous cycle revolves around the actions of developers as they check code into CVS. When changed or new code is checked into CVS, the DART continuous testing process kicks in. A small number of tests are performed (including compilation), and if something breaks, email is sent to all developers who checked code in during the continuous cycle. Developers are expected to fix the problem immediately.

The daily cycle occurs over a 24-hour period. Changes to the source base made during the day are extensively tested by the nightly DART regression testing sequence. These tests occur on different combinations of computers and operating systems located around the world, and the results are posted every day to the DART dashboard. Developers who checked in code are expected to visit the dashboard and ensure their changes are acceptable—that is, they do not introduce compilation errors or warnings, or break any other tests including regression, memory, PrintSelf, and Set/Get. Again, developers are expected to fix problems immediately.

The release cycle occurs a small number of times a year. This requires tagging and branching the CVS repository, updating documentation, and producing new release packages. Although additional testing is performed to insure the consistency of the package, keeping the daily CVS build error free minimizes the work required to cut a release.

ITK users typically work with releases, since they are the most stable. Developers work with the CVS repository, or sometimes with periodic release snapshots, in order to take advantage of newly-added features. It is extremely important that developers watch the dashboard carefully, and *update their software only when the dashboard is in good condition (i.e., is "green")*. Failure to do so can cause significant disruption if a particular day's software release is unstable.

14.4 The Effectiveness of the Process

The effectiveness of this process is profound. By providing immediate feedback to developers through email and Web pages (e.g., the dashboard), the quality of ITK is exceptionally high, especially considering the complexity of the algorithms and system. Errors, when accidently introduced, are caught quickly, as compared to catching them at the point of release. To wait to the point of release is to wait too long, since the causal relationship between a code change or addition and a bug is lost. The process is so powerful that it routinely catches errors in vendor's graphics drivers (e.g., OpenGL drivers) or changes to external subsystems such as the VXL/VNL numerics library. All of these tools that make up the process (CMake, CVS, and DART) are open-source. Many large and small systems such as VTK (The Visualization Toolkit `http://www.vtk.org`) use the same process with similar results. We encourage the adoption of the process in your environment.

BIBLIOGRAPHY

[1] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Professional Computing Series. Addison-Wesley, 2001. 7.8.3, 7.10, 8.8.1

[2] K. Alsabti, S. Ranka, and V. Singh. An efficient k-means clustering algorithm. In *First Workshop on High-Performance Data Mining*, 1998. 10.4.1

[3] L. Alvarez and J.-M. Morel. *A Morphological Approach To Multiscale Analysis: From Principles to Equations*, pages 229–254. Kluwer Academic Publishers, 1994. 6.7.3

[4] E. Angelini, C. Imielinska, J. Jin, and A. Laine. Improving Statistics for Hybrid Segmentation of High-Resolution Multichannel Images. In *SPIE Medical Imaging 2002*, San Diego, 2002. 9.4.1, 9.4.3

[5] ANSI-ISO. *Programming Languages - C++*. American National Standards Institue, 1998. 7.9

[6] M. H. Austern. *Generic Programming and the STL:*. Professional Computing Series. Addison-Wesley, 1999. 3.2.1, 7.8.3, 7.10, 8.8.1

[7] Stephen Aylward. *Gaussian Goodness of Fit Cores*. PhD thesis, Department of Computer Science, University of North Carolina at Chapel Hill, 1997. 10.2.2

[8] Stephen Aylward and Steve Pizer. Continuous gaussian mixture modeling. In *Information Processing in Medical Imaging 1997 (IPMI'97)*, pages 176–189, 1997. 10.2.2

[9] J. Besag. On the statistical analysis of dirty pictures. *J. Royal Statist. Soc. B.*, 48:259–302, 1986. 10.4.5

[10] R. N. Bracewell. *The Fourier Transform and its Applications*. McGraw-Hill, 1999. 6.10.1

[11] R. N. Bracewell. *Fourier Analysis and Imaging*. Plenum US, 2004. 6.10.1

[12] R. H. Byrd, P. Lu, and J. Nocedal. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific and Statistical Computing*, 16(5):1190–1208, 1995. 8.11

[13] R. H. Byrd C. Zhu and J. Nocedal. L-bfgs-b: Algorithm 778: L-bfgs-b, fortran routines for large scale bound constrained optimization. *ACM Transactions on Mathematical Software*, 23(4):550–560, November 1997. 8.11

[14] V. Caselles, R. Kimmel, and G. Sapiro. Geodesic active contours. *International Journal on Computer Vision*, 22(1):61–97, 1997. 9.3.3

[15] K.R. Castleman. *Digital Image Processing*. Prentice Hall, Upper Saddle River, NJ, 1996. 11.4.1, 11.4.2

[16] A. Chung, W. Wells, A. Norbash, and W. Grimson. Multi-modal image registration by minimising kullback-leibler distance. In *MICCAI'02 Medical Image Computing and Computer-Assisted Intervention*, Lecture Notes in Computer Science, pages 525–532, 2002. 8.10.5

[17] A. Collignon, F. Maes, D. Delaere, D. Vandermeulen, P. Suetens, and G. Marchal. Automated multimodality image registration based on information theory. In *Information Processing in Medical Imaging 1995*, pages 263–274. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1995. 8.5

[18] P. E. Danielsson. Euclidean distance mapping. *Computer Graphics and Image Processing*, 14:227–248, 1980. 6.8

[19] C. Darwin. *On the Origin of Species*. http://www.gutenberg.org, sixth edition, 1999. 8.6

[20] M. H. Davis, A. Khotanzad, D. P. Flamig, and S. E. Harms. A physics-based coordinate transformation for 3-d image matching. *IEEE Transactions on Medical Imaging*, 16(3), June 1997. 8.8.18

[21] R. Deriche. Fast algorithms for low level vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(1):78–87, 1990. 6.4.2, 6.7.1, 6.7.1

[22] R. Deriche. Recursively implementing the gaussian and its derivatives. Technical Report 1893, Unite de recherche INRIA Sophia-Antipolis, avril 1993. Research Repport. 6.4.2, 6.7.1, 6.7.1

[23] C. Dodson and T. Poston. *Tensor Geometry: The Geometric Viewpoint and its Uses*. Springer, 1997. 8.8.1, 11

[24] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern classification*. A Wiley-Interscience Publication, second edition, 2000. 10.2.3, 10.4, 10.4

[25] David Eberly. *Ridges in Image and Data Analysis*. Kluwer Academic Publishers, Dordrecht, 1996. 9.2.1

[26] Benoit Regrain Eric Boix, Mathieu Malaterre and Jean-Pierre Roux. *The GDCM Library*. CNRS, INSERM, INSA Lyon, UCB Lyon, http://www-creatis.insa-lyon.fr/Public/Gdcm/. 7.12.1

[27] K. Fogel. *Open Source Development with CVS*. Corolis, 1999. 1.4.2, 14.1

[28] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995. 3.2.6, 4.3.9, 7.2, 8.4, 13.6

[29] G. Gerig, O. Kübler, R. Kikinis, and F. A. Jolesz. Nonlinear anisotropic filtering of MRI data. *IEEE Transactions on Medical Imaging*, 11(2):221–232, June 1992. 6.7.3

[30] R.C. Gonzalez and R.E. Woods. *Digital Image Processing*. Addison-Wesley, Reading, MA, 1993. 11.4.1, 11.4.1, 11.4.2

[31] H. Gray. *Gray's Anatomy*. Merchant Book Company, sixteenth edition, 2003. 4.1.5

[32] Stephen Grossberg. Neural dynamics of brightness perception: Features, boundaries, diffusion, and resonance. *Perception and Psychophysics*, 36(5):428–456, 1984. 6.7.3

[33] J. Hajnal, D. J. Hawkes, and D. Hill. *Medical Image Registration*. CRC Press, 2001. 8.5, 8.10.6

[34] W. R. Hamilton. *Elements of Quaternions*. Chelsea Publishing Company, 1969. 8.6.4, 8.8.1, 8.8.11, 8.11

[35] A. Hendersen. *The Paraview Guide*. Kitware, Inc, 2004. 8.15

[36] M. Holden, D. L. G. Hill, E. R. E. Denton, J. M. Jarosz, T. C. S. Cox, and D. J. Hawkes. Voxel similarity measures for 3d serial mr brain image registration. In A. Kuba, M. Samal, and A. Todd-Pkropek, editors, *Information Processing in Medical Imaging 1999 (IPMI'99)*, pages 472–477. Springer, 1999. 8.10.3

[37] B. K. Horn. Closed-form solution of absolute orientation using unit quaternions. *Journal of the Optical Society of America*, 4:629–642, April 1987. 8.17

[38] C. Imielinska, M. Downes, and W. Yuan. Semi-Automated Color Segmentation of Anatomical Tissue. *Journal of Computerized Medical Imaging and Graphics*, 24:173–180, April 2000. 9.4.1, 9.4.3, 9.4.3, 9.4.3

[39] C. Imielinska, D. Metaxas, J. Udupa, Y. Jin, and T. Chen. Hybrid Segmentation of the Visible Human Data. In *Proceedings of the Third Visible Human Project Conference*, Bethesda, MD, 5 October 2000. 9.4.1, 9.4.3

[40] C. Imielinska, D. Metaxas, J.K. Udupa, Y. Jin, and T. Chen. Hybrid Segmentation of Anatomical Data. In *Proceedings MICCAI*, pages 1048–1057, Utrecht, The Netherlands, 2001. 9.4.1, 9.4.3

[41] Y. Jin, C. Imielinska, and A. Laine. A Homogeneity-Based Speed Term for Level-set Shape Detection. In *SPIE Medical Imaging*, San Diego, 2002. 9.4.1, 9.4.3

[42] C. J. Joly. *A Manual of Quaternions*. MacMillan and Co. Limited, 1905. 8.6.4, 8.8.11

[43] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine Piatko, Ruth Silverman, and Angela Y. Wu. An efficient k-means clustering algorithm: Analysis and implementation. 10.1.7, 10.4.1

[44] J. Koënderink and A. van Doorn. The Structure of Two-Dimensional Scalar Fields with Applications to Vision. *Biol. Cybernetics*, 33:151–158, 1979. 9.2.1

[45] J. Koenderink and A. van Doorn. Local features of smooth shapes: Ridges and courses. *SPIE Proc. Geometric Methods in Computer Vision II*, 2031:2–13, 1993. 9.2.1

[46] L. Kohn, J. Corrigan, and M.Donaldson, editors. *To Err is Human: Building a safer health system*. National Academy Press, 2001. 7.12.4

[47] S. Kullback. *Information Theory and Statistics*. Dover Publications, 1997. 10.3.2

[48] M. Leventon, W. Grimson, and O. Faugeras. Statistical shape influence in geodesic active contours. In *Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, volume 1, pages 316–323, 2000. 9.3.7

[49] T. Lindeberg. *Scale-Space Theory in Computer Science*. Kluwer Academic Publishers, 1994. 6.7.1

[50] H. Lodish, A. Berk, S. Zipursky, P. Matsudaira, D. Baltimore, and J. Darnell. *Molecular Cell Biology*. W. H. Freeman and Company, 2000. 4.1.5, 8.8.1

[51] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics*, 21(4):163–169, July 1987. 6.11.1

[52] F. Maes, A. Collignon, D. Meulen, G. Marchal, and P. Suetens. Multi-modality image registration by maximization of mutual information. *IEEE Trans. on Med. Imaging*, 16:187–198, 1997. 8.5

[53] D. Malacara. *Color Vision and Colorimetry: Theory and Applications*. SPIE PRESS, 2002. 4.1.5, 4.1.5

[54] R. Malladi, J. A. Sethian, and B. C. Vermuri. Shape modeling with front propagation: A level set approach. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 17(2):158–174, 1995. 9.3.2

[55] D. Mattes, D. R. Haynor, H. Vesselle, T. K. Lewellen, and W. Eubank. Non-rigid multimodality image registration. In *Medical Imaging 2001: Image Processing*, pages 1609–1620, 2001. 8.8.17, 8.10.4

[56] D. Mattes, D. R. Haynor, H. Vesselle, T. K. Lewellen, and W. Eubank. PET-CT image registration in the chest using free-form deformations. *IEEE Trans. on Medical Imaging*, 22(1):120–128, January 2003. 8.5.2, 8.8.17

[57] E. H. Meijering, W. J. Niessen, J. P. Pluim, and M. A. Viergever. Quantitative comparision os sinc-approximating kernels for medical image interpolation. In W. M. Wells, A. Colchester, and S. Delp, editors, *MICCAI'98 First International Conference on Medical Image Computing and Computer-Assisted Intervention*, Lecture Notes in Computer Science, pages 972–980. Springer Verlag, September 1999. 8.9.4

[58] D. Musser and A. Saini. *STL Tutorial and Reference Guide*. Professional Computing Series. Addison-Wesley, 1996. 3.2.1

[59] David R. Musser. Introspective sorting and selection algorithms. *Software–Practice and Experience*, 8:983–993, 1997. 10.2.3

[60] NEMA. The dicom standard. Technical report, NEMA, http://medial.nema.org/, 2004. 7.12.1

[61] Dan Pelleg and Andrew Moore. Accelerating exact k -means algorithms with geometric reasoning. In *Fifth ACM SIGKDD International Conference On Knowledge Discovery and Data Mining*, pages 277–281, 1999. 10.4.1

[62] G. P. Penney, J. Weese, J. A. Little, P. Desmedt, D. L. G. Hill, and D. J. Hawkes. A comparision of similarity measures for use in 2d-3d medical image registration. *IEEE Transactions on Medical Imaging*, 17(4):586–595, August 1998. 8.10.3

[63] P. Perona and J. Malik. Scale-space and edge detection using anisotropic diffusion. *IEEE Transactions on Pattern Analysis Machine Intelligence*, 12:629–639, 1990. 6.7.3, 6.7.3, 6.7.3

[64] J. P. Pluim, J. B. A. Maintz, and M. A. Viergever. Mutual-Information-Based Registration of Medical Images: A Survey. *IEEE Transactions on Medical Imaging*, 22(8):986–1004, August 2003. 8.5, 8.10.4

[65] K. Popper. *Open Society and Its Enemies*. Princenton University Press, 1971. 8.5.1

[66] K. Popper. *The Logic of Scientific Discovery*. Routledge, 2002. 8.5.1, 10.3.1

[67] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C*. Cambridge University Press, second edition, 1992. 8.11

[68] K. Rohr, M. Fornefett, and H. S. Stiehl. Approximating thin-plate splines for elastic registration: Integration of landmark errors and orientation attributes. In A. Kuba, M. Samal, and A. Todd-Pkropek, editors, *Information Processing in Medical Imaging 1999 (IPMI'99)*, pages 252–265. Springer, 1999. 8.8.18

[69] K. Rohr, H. S. Stiehl, R. Sprengel, T. M. Buzug, J. Weese, and M. H Kuhn. Landmark-based elastic registration using approximating thin-plate splines. *IEEE Transactions on Medical Imaging*, 20(6):526–534, June 1997. 8.8.18, 8.17

[70] D. Rueckert, L. I. Sonoda, C. Hayes, D. L. G. Hill, M. O. Leach, and D. J. Hawkes. Non-rigid registration using free-form deformations: Application to breast mr images. *IEEE Transaction on Medical Imaging*, 18(8):712–721, 1999. 8.8.17

[71] G. Sapiro and D. Ringach. Anisotropic diffusion of multivalued images with applications to color filtering. *IEEE Trans. on Image Processing*, 5:1582–1586, 1996. 6.7.3

[72] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit, An Object Oriented Approach to 3D Graphics*. Kitware Inc, 1998. 6.11.1

[73] J. P. Serra. *Image Analysis and Mathematical Morphology*. Academic Press Inc., 1982. 6.6.3, 9.2.1

[74] J.A. Sethian. *Level Set Methods and Fast Marching Methods*. Cambridge University Press, 1996. 9.3

[75] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, July 1948. 6.9.4, 10.3.2

[76] C. E. Shannon and W. Weaver. *The Mathematical Theory of Communication*. University of Illinois Press, 1948. 6.9.4, 10.3.2

[77] J. C. Spall. An overview of the simultaneous perturbation method for efficient optimization. *Johns Hopkins APL Technical Digest*, 19:482–492, 1998. 8.11

[78] M. Styner, C. Brehbuhler, G. Szekely, and G. Gerig. Parametric estimate of intensity homogeneities applied to MRI. *IEEE Trans. Medical Imaging*, 19(3):153–165, March 2000. 8.11

[79] Baart M. ter Haar Romeny, editor. *Geometry-Driven Diffusion in Computer Vision*. Kluwer Academic Publishers, 1994. 6.7.3

[80] J. P. Thirion. Fast non-rigid matching of 3D medical image. Technical report, Research Report RR-2547, Epidure Project, INRIA Sophia, May 1995. 8.14

[81] J.-P. Thirion. Image matching as a diffusion process: an analogy with maxwell's demons. *Medical Image Analysis*, 2(3):243–260, 1998. 8.14

[82] J. Udupa and S. Samarasekera. Fuzzy connectedness and object definition: Theory, algorithms, and applications in image segmentation. *Graphical Models and Image Processing*, 58(3):246–261, 1996. 9.4.2, 9.4.3, 9.4.3

[83] J. K. Udupa, V.R. Leblanc, H. Schmidt, C. Imielinska, K.P. Saha, G.J. Grevera, Y. Zhuge, P. Molholt, L. Currie, and Y. Jin. A Methodology for Evaluating Image Segmentation Algorithms. In *SPIE Medical Imaging*, San Diego, 2002. 9.4.1, 9.4.3

[84] J. K. Udupa and S. Samarasekera. Extraction of fuzzy object information in multidimensional images for quantifying ms lesions of the brain. Technical Report 5,812,691, United States Patent Office http://www.uspto.gov, 1998. 9.4.2, 9.4.3

[85] P. Viola and W. M. Wells III. Alignment by maximization of mutual information. *IJCV*, 24(2):137–154, 1997. 8.5, 8.10.4

[86] J. Weickert, B.M. ter Haar Romeny, and M.A. Viergever. Conservative image transformations with restoration and scale-space properties. In *Proc. 1996 IEEE International Conference on Image Processing (ICIP-96, Lausanne, Sept. 16-19, 1996)*, pages 465–468, 1996. 6.7.3

[87] R. T. Whitaker and G. Gerig. *Vector-Valued Diffusion*, pages 93–134. Kluwer Academic Publishers, 1994. 6.7.3, 6.7.3

[88] R. T. Whitaker and X. Xue. Variable-Conductance, Level-Set Curvature for Image Processing. In *International Conference on Image Processing*, pages 142–145. IEEE, 2001. 6.7.3

[89] Ross T. Whitaker. Characterizing first and second order patches using geometry-limited diffusion. In *Information Processing in Medical Imaging 1993 (IPMI'93)*, pages 149–167, 1993. 6.7.3

[90] Ross T. Whitaker. *Geometry-Limited Diffusion*. PhD thesis, The University of North Carolina, Chapel Hill, North Carolina 27599-3175, 1993. 6.7.3, 6.7.3

[91] Ross T. Whitaker. Geometry-limited diffusion in the characterization of geometric patches in images. *Computer Vision, Graphics, and Image Processing: Image Understanding*, 57(1):111–120, January 1993. 6.7.3

[92] Ross T. Whitaker and Stephen M. Pizer. Geometry-based image segmentation using anisotropic diffusion. In Ying-Lie O, A. Toet, H.J.A.M Heijmans, D.H. Foster, and P. Meer, editors, *Shape in Picture: The mathematical description of shape in greylevel images*. Springer Verlag, Heidelberg, 1993. 6.7.3

[93] Ross T. Whitaker and Stephen M. Pizer. A multi-scale approach to nonuniform diffusion. *Computer Vision, Graphics, and Image Processing: Image Understanding*, 57(1):99–110, January 1993. 6.7.3

[94] G. Wyszecki. *Color Science: Concepts and Methods, Quantitative Data and Formulae*. Wiley-Interscience, 2000. 4.1.5, 4.1.5

[95] Terry S. Yoo and James M. Coggins. Using statistical pattern recognition techniques to control variable conductance diffusion. In *Information Processing in Medical Imaging 1993 (IPMI'93)*, pages 459–471, 1993. 6.7.3

[96] T.S. Yoo, U. Neumann, H. Fuchs, S.M. Pizer, T. Cullip, J. Rhoades, and R.T. Whitaker. Direct visualization of volume data. *IEEE Computer Graphics and Applications*, 12(4):63–71, 1992. 9.2.1

[97] T.S. Yoo, S.M. Pizer, H. Fuchs, T. Cullip, J. Rhoades, and R. Whitaker. Achieving direct volume visualization with interactive semantic region selection. In *Information Processing in Medical Images*. Springer Verlag, 1991. 9.2.1, 9.2.1

INDEX