
The ITK Software Guide
Book 1: Introduction and Development
Guidelines
Fourth Edition
Updated for ITK version 4.10

Hans J. Johnson, Matthew M. McCormick, Luis Ibáñez,
and the *Insight Software Consortium*

May 27, 2016

<https://itk.org>
Email: community@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 a cross-platform software. It uses a build environment known as [CMake](#) to manage platform-specific project generation and compilation process in a platform-independent way. ITK is implemented in C++. ITK's implementation style employs generic programming, which involves the use of templates to generate, at compile-time, code that can be applied *generically* to any class or data-type that supports the operations used by the template. The use of C++ templating means that the code is highly efficient and many issues are discovered at compile-time, rather than at run-time during program execution. It also means that many of ITK's algorithms can be applied to arbitrary spatial dimensions and pixel types.

An automated wrapping system integrated with ITK generates an interface between C++ and a high-level programming language [Python](#). This enables rapid prototyping and faster exploration of ideas by shortening the edit-compile-execute cycle. In addition to automated wrapping, the [SimpleITK](#) project provides a streamlined interface to ITK that is available for C++, Python, Java, CSharp, R, Tcl and Ruby.

Developers from around the world can use, debug, maintain, and extend the software because ITK is an open-source project. ITK uses a model of software development known as Extreme Programming. Extreme Programming collapses the usual software development methodology into a simultaneous 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. An extensive testing process supported by the system known as [CDash](#) measures the quality of ITK code on a daily basis. The ITK Testing Dashboard is updated continuously, reflecting the quality of

the code at any moment.

The most recent version of this document is available online at <https://itk.org/ItkSoftwareGuide.pdf>. This book is a guide to developing software with ITK; it is the first of two companion books. This book covers building and installation, general architecture and design, as well as the process of contributing in the ITK community. The second book covers detailed design and functionality for reading and writing images, filtering, registration, segmentation, and performing statistical analysis.

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 <https://itk.org/ITK/project/parti.html> for the names of additional contributors, as well as checking the GIT 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 L^AT_EX 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.

Since the software guide is generated in part from the ITK source code itself, many ITK developers have been involved in updating and extending the ITK documentation. These include **David Doria**, **Bradley Lowekamp**, **Mark Foskey**, **Gaëtan Lehmann**, **Andreas Schuh**, **Tom Vercauteren**, **Cory Quammen**, **Daniel Blezek**, **Paul Huggett**, **Matthew McCormick**, **Josh Cates**, **Arnaud Gelas**, **Jim Miller**, **Brad King**, **Gabe Hart**, **Hans Johnson**.

Hans Johnson, **Kent Williams**, **Constantine Zakkaroff**, **Xiaoxiao Liu**, **Ali Ghayoor**, and **Matthew McCormick** updated the documentation for the initial ITK Version 4 release.

Luis Ibáñez and **Sébastien Barré** designed the original Book 1 cover. **Matthew McCormick** and **Brad King** updated the code to produce the Book 1 cover for ITK 4 and VTK 6. **Xiaoxiao Liu**, **Bill Lorensen**, **Luis Ibáñez**, and **Matthew McCormick** created the 3D printed anatomical objects that were photographed by **Sébastien Barré** for the Book 2 cover. **Steve Jordan** designed the layout of the covers.

Lisa Avila, **Hans Johnson**, **Matthew McCormick**, **Sandy McKenzie**, **Christopher Mullins**, **Katie Osterdahl**, and **Michka Popoff** prepared the book for the 4.7 print release.

CONTENTS

I	Introduction	1
1	Welcome	3
1.1	Organization	3
1.2	How to Learn ITK	4
1.3	Obtaining the Software	5
1.3.1	Downloading Packaged Releases	5
1.3.2	Downloading From Git	6
1.3.3	Data	6
1.4	Software Organization	6
1.5	The Insight Community and Support	8
1.6	A Brief History of ITK	9
2	Configuring and Building ITK	11
2.1	Using CMake for Configuring and Building ITK	12
2.1.1	Preparing CMake	12
2.1.2	Configuring ITK	14
2.1.3	Advanced Module Configuration	15
2.1.4	Compiling ITK	16
2.1.5	Installing ITK on Your System	17
2.2	Getting Started With ITK	18

2.2.1	Hello World!	19
II	Architecture	21
3	System Overview	23
3.1	System Organization	23
3.2	Essential System Concepts	24
3.2.1	Generic Programming	24
3.2.2	Include Files and Class Definitions	25
3.2.3	Object Factories	25
3.2.4	Smart Pointers and Memory Management	26
3.2.5	Error Handling and Exceptions	27
3.2.6	Event Handling	28
3.2.7	Multi-Threading	29
3.3	Numerics	29
3.4	Data Representation	30
3.5	Data Processing Pipeline	31
3.6	Spatial Objects	32
3.7	Wrapping	33
3.7.1	Python Setup	36
4	Data Representation	39
4.1	Image	39
4.1.1	Creating an Image	39
4.1.2	Reading an Image from a File	41
4.1.3	Accessing Pixel Data	42
4.1.4	Defining Origin and Spacing	43
4.1.5	RGB Images	48
4.1.6	Vector Images	50
4.1.7	Importing Image Data from a Buffer	51
4.2	PointSet	54
4.2.1	Creating a PointSet	54

4.2.2	Getting Access to Points	56
4.2.3	Getting Access to Data in Points	58
4.2.4	RGB as Pixel Type	60
4.2.5	Vectors as Pixel Type	62
4.2.6	Normals as Pixel Type	64
4.3	Mesh	66
4.3.1	Creating a Mesh	66
4.3.2	Inserting Cells	68
4.3.3	Managing Data in Cells	71
4.3.4	Customizing the Mesh	73
4.3.5	Topology and the K-Complex	77
4.3.6	Representing a PolyLine	83
4.3.7	Simplifying Mesh Creation	86
4.3.8	Iterating Through Cells	89
4.3.9	Visiting Cells	91
4.3.10	More on Visiting Cells	93
4.4	Path	97
4.4.1	Creating a PolyLineParametricPath	97
4.5	Containers	98
5	Spatial Objects	103
5.1	Introduction	103
5.2	Hierarchy	104
5.3	SpatialObject Tree Container	106
5.4	Transformations	107
5.5	Types of Spatial Objects	111
5.5.1	ArrowSpatialObject	111
5.5.2	BlobSpatialObject	112
5.5.3	CylinderSpatialObject	113
5.5.4	EllipseSpatialObject	114
5.5.5	GaussianSpatialObject	116
5.5.6	GroupSpatialObject	117

5.5.7	ImageSpatialObject	118
5.5.8	ImageMaskSpatialObject	119
5.5.9	LandmarkSpatialObject	121
5.5.10	LineSpatialObject	122
5.5.11	MeshSpatialObject	124
5.5.12	SurfaceSpatialObject	126
5.5.13	TubeSpatialObject	127
	VesselTubeSpatialObject	129
	DTITubeSpatialObject	131
5.6	SceneSpatialObject	133
5.7	Read/Write SpatialObjects	135
5.8	Statistics Computation via SpatialObjects	136

III Development Guidelines 139

6	Iterators	141
6.1	Introduction	141
6.2	Programming Interface	142
6.2.1	Creating Iterators	142
6.2.2	Moving Iterators	142
6.2.3	Accessing Data	144
6.2.4	Iteration Loops	145
6.3	Image Iterators	146
6.3.1	ImageRegionIterator	146
6.3.2	ImageRegionIteratorWithIndex	148
6.3.3	ImageLinearIteratorWithIndex	150
6.3.4	ImageSliceIteratorWithIndex	154
6.3.5	ImageRandomConstIteratorWithIndex	158
6.4	Neighborhood Iterators	159
6.4.1	NeighborhoodIterator	165
	Basic neighborhood techniques: edge detection	165
	Convolution filtering: Sobel operator	168

Optimizing iteration speed	169
Separable convolution: Gaussian filtering	171
Slicing the neighborhood	172
Random access iteration	174
6.4.2 ShapedNeighborhoodIterator	176
Shaped neighborhoods: morphological operations	177
7 Image Adaptors	181
7.1 Image Casting	182
7.2 Adapting RGB Images	184
7.3 Adapting Vector Images	187
7.4 Adaptors for Simple Computation	189
7.5 Adaptors and Writers	191
8 How To Write A Filter	193
8.1 Terminology	193
8.2 Overview of Filter Creation	194
8.3 Streaming Large Data	195
8.3.1 Overview of Pipeline Execution	196
8.3.2 Details of Pipeline Execution	198
UpdateOutputInformation()	198
PropagateRequestedRegion()	199
UpdateOutputData()	200
8.4 Threaded Filter Execution	200
8.5 Filter Conventions	201
8.5.1 Optional	202
8.5.2 Useful Macros	202
8.6 How To Write A Composite Filter	202
8.6.1 Implementing a Composite Filter	203
8.6.2 A Simple Example	204
9 How To Create A Module	209
9.1 Name and dependencies	209

9.1.1	CMakeLists.txt	210
9.1.2	itk-module.cmake	210
9.2	Headers	212
9.3	Libraries	212
9.4	Tests	213
9.5	Wrapping	215
9.5.1	CMakeLists.txt	215
9.5.2	Class wrap files	216
	Wrapping Variables	217
	Wrapping Macros	218
9.6	Third-Party Dependencies	223
9.6.1	itk-module-init.cmake	224
9.6.2	CMakeList.txt	224
10	Software Process	227
10.1	Git Source Code Repository	227
10.2	CDash Regression Testing System	228
10.3	Working The Process	230
10.4	The Effectiveness of the Process	230
Appendices		233
A	Licenses	235
A.1	Insight Toolkit License	235
A.2	Third Party Licenses	240
A.2.1	DICOM Parser	240
A.2.2	Double Conversion	241
A.2.3	Expat	242
A.2.4	GDCM	242
A.2.5	GIFTI	243
A.2.6	HDF5	243
A.2.7	JPEG	246
A.2.8	KWSys	247

A.2.9	MetaIO	248
A.2.10	Netlib's SLATEC	250
A.2.11	NIFTI	250
A.2.12	NrrdIO	250
A.2.13	OpenJPEG	253
A.2.14	PNG	254
A.2.15	TIFF	257
A.2.16	VNL	257
A.2.17	ZLIB	258

LIST OF FIGURES

2.1	CMake user interface	13
2.2	ITK Group Configuration	16
2.3	Default ITK Configuration	17
4.1	ITK Image Geometrical Concepts	44
4.2	PointSet with Vectors as PixelType	62
5.1	SpatialObject Transformations	108
5.2	SpatialObject Transform Computations	110
6.1	ITK image iteration	143
6.2	Copying an image subregion using ImageRegionIterator	149
6.3	Using the ImageRegionIteratorWithIndex	150
6.4	Maximum intensity projection using ImageSliceIteratorWithIndex	158
6.5	Neighborhood iterator	160
6.6	Some possible neighborhood iterator shapes	161
6.7	Sobel edge detection results	168
6.8	Gaussian blurring by convolution filtering	173
6.9	Finding local minima	176
6.10	Binary image morphology	180

7.1	ImageAdaptor concept	182
7.2	Image Adaptor to RGB Image	186
7.3	Image Adaptor to Vector Image	189
7.4	Image Adaptor for performing computations	191
8.1	Relationship between DataObjects and ProcessObjects	194
8.2	The Data Pipeline	196
8.3	Sequence of the Data Pipeline updating mechanism	197
8.4	Composite Filter Concept	203
8.5	Composite Filter Example	204
10.1	CDash Quality Dashboard	229

LIST OF TABLES

6.1	ImageRandomConstIteratorWithIndex usage	159
9.1	Wrapping Configuration Variables	216
9.2	Wrapping CMake Mangling Variables for PODs	219
9.3	Wrapping CMake Mangling Variables for other ITK pixel types.	220
9.4	Wrapping CMake Mangling Variables for Basic ITK types.	221

Part I

Introduction

CHAPTER
ONE

WELCOME

Welcome to the *Insight Segmentation and Registration Toolkit (ITK) Software Guide*. This book has been updated for ITK 4.10 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.

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, take advantage of the many resources available¹, including example materials, which provide cookbook recipes that concisely demonstrate how to achieve a given task, the Doxygen pages, which document the specific algorithm parameters, and the knowledge of the many ITK community members (see Section 1.5 on page 8.)

The Insight Toolkit is an open-source software system. This means that the community surrounding ITK has a great impact on the evolution of the software. The community can make significant contributions to ITK by providing code reviews, bug patches, feature patches, new classes, documentation, and discussions. Please feel free to contribute your ideas through the ITK community mailing list.

1.1 Organization

This software guide is divided into three parts. Part I is a general introduction to ITK, with a description of how to install the Insight Toolkit on your computer. This includes how to build the library from its source code. Part II introduces basic system concepts such as an overview of the

¹<https://www.itk.org/ITK/help/documentation.html>

system architecture, and how to build applications in the C++ and Python programming languages. Part II also describes the design of data structures and application of analysis methods within the system. Part III is for the ITK contributor and explains how to create your own classes, extend the system, and be an active participant in the project.

1.2 How to Learn ITK

The key to learning how to use ITK is to become familiar with its palette of objects and the ways to combine them. There are three categories of documentation to help with the learning process: high level guidance material (the Software Guide), "cookbook" demonstrations on how to achieve concrete objectives (the examples), and detailed descriptions of the application programming interface (the Doxygen² documentation). These resources are combined in the three recommended stages for learning ITK.

In the first stage, thoroughly read this introduction, which provides an overview of some of the key concepts of the system. It also provides guidance on how to build and install the software. After running your first "hello world" program, you are well on your way to advanced computational image analysis!

The next stage is to execute a few examples and gain familiarity with the available documentation. By running the examples, one can gain confidence in achieving results and is introduced the mechanics of the software system. There are three example resources,

1. the Examples directory of the ITK source code repository ³.
2. the Examples pages on the ITK Wiki ⁴
3. the Sphinx documented ITK Examples ⁵

To gain familiarity with the available documentation, browse the sections available in Part II and Part III of this guide. Also, browse the Doxygen application programming interface (API) documentation for the classes applied in the examples.

Finally, mastery of ITK involves integration of information from multiple sources. the second companion book is a reference to algorithms available, and Part III introduces how to extend them to your needs and participate in the community. Individual examples are a detailed starting point to achieve certain tasks. In practice, the Doxygen documentation becomes a frequent reference as an index of the classes available, their descriptions, and the syntax and descriptions of their methods. When examples and Doxygen documentation are insufficient, the software unit tests thoroughly demonstrate how the code is utilized. Last, but not least, the source code itself is an extremely valuable resource.

²<https://itk.org/Doxygen/index.html>

³1.3

⁴<https://itk.org/Wiki/ITK/Examples>

⁵<https://itk.org/ITKExamples>

The code is the most detailed, up-to-date, and definitive description of the software. A great deal of attention and effort is directed to the code's readability, and its value cannot be understated.

The following sections describe how to obtain the software, summarize the software functionality in each directory, and how to locate data.

1.3 Obtaining the Software

There are two different ways to access the ITK source code:

Periodic releases Official releases are available on the ITK web site⁶. They are released twice a year, and announced on the ITK web pages and mailing list. However, they may not provide the latest and greatest features of the toolkit.

Continuous repository checkout Direct access to the Git source code repository⁷ provides immediate availability to the latest toolkit additions. But, on any given day the source code may not be stable as compared to the official releases.

This software guide assumes that you are using the current released version of ITK, available on the ITK web site. If you are a new user, we recommend the released version of the software. It is stable, consistent, and better tested than the code available from the Git repository. When working from the repository, please be aware of the ITK quality testing dashboard. The Insight Toolkit is heavily tested using the open-source CDash regression testing system⁸. Before updating the repository, make sure that the dashboard is *green*, indicating stable code. (Learn more about the ITK dashboard and quality assurance process in Section 10.2 on page 228.)

1.3.1 Downloading Packaged Releases

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

<https://www.itk.org/ITK/resources/software.html>

On the web page, choose the tarball that better fits your system. The options are .zip and .tar.gz files. The first type is better suited for Microsoft-Windows, while the second one is the preferred format for UNIX systems.

Once you unzip or untar the file a directory called `InsightToolkit-4.10.0` will be created in your disk and you will be ready to start the configuration process described in Section 2.1.1 on page 12.

⁶<https://itk.org/>

⁷<https://itk.org/ITK.git>

⁸<http://open.cdash.org/index.php?project=Insight>

1.3.2 Downloading From Git

Git is a free and open source distributed version control system. For more information about Git please see Section 10.1 on page 227. (Note: please make sure that you access the software via Git only when the ITK quality dashboard indicates that the code is stable.)

Access ITK via Git using the following commands (under a Git Bash shell):

```
git clone git://itk.org/ITK.git
```

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

```
git pull
```

Once you obtain the software you are ready to configure and compile it (see Section 2.1.1 on page 12). First, however, we recommend reading the following sections that describe the organization of the software and joining the mailing list.

1.3.3 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:

<https://www.itk.org/ITK/resources/links.html>
<ftp://public.kitware.com/pub/itk/Data/>.

1.4 Software Organization

To begin your ITK odyssey, you will first need to know something about ITK's software organization and directory structure. It is helpful to know enough to navigate through the code base to find examples, code, and documentation.

ITK resources are organized into multiple Git repositories. The ITK library source code are in the `ITK`⁹ Git repository. The sphinx Examples are in the `ITKExamples`¹⁰ repository. Fairly complex applications using ITK (and other systems such as VTK, Qt, and FLTK) are available from `InsightApplications`¹¹ repository. The sources for this guide are in the `ITKSoftwareGuide`¹²

⁹<https://itk.org/ITK.git>

¹⁰<https://itk.org/ITKExamples.git>

¹¹<https://itk.org/ITKApps.git>

¹²<https://itk.org/ITKSoftwareGuide.git>

repository.

The ITK repository contains the following subdirectories:

- ITK/Modules — the heart of the software; the location of the majority of the source code.
- ITK/Documentation — migration guides and Doxygen infrastructure.
- ITK/Examples — a suite of simple, well-documented examples used by this guide, illustrating important ITK concepts.
- ITK/Testing — a collection of the MD5 files, which are used to link with the ITK data servers to download test data. This test data is used by tests in ITK/Modules to produce the ITK Quality Dashboard using CDash. (see Section 10.2 on page 228.)
- Insight/Utilities — the scripts that support source code development. For example, CTest and Doxygen support.
- Insight/Wrapping — the wrapping code to build interfaces between the C++ library and various interpreted languages (currently Python is supported).

The source code directory structure—found in ITK/Modules—is the most important to understand.

- ITK/Modules/Core — core classes, macro definitions, typedefs, and other software constructs central to ITK. The classes in Core are the only ones always compiled as part of ITK.
- ITK/Modules/ThirdParty — various third-party libraries that are used to implement image file I/O and mathematical algorithms. (Note: ITK’s mathematical library is based on the VXL/VNL software package¹³.)
- ITK/Modules/Filtering — image processing filters.
- ITK/Modules/IO — classes that support the reading and writing of images, transforms, and geometry.
- ITK/Modules/Bridge — classes used to connect with the other analysis libraries or visualization libraries, such as OpenCV¹⁴ and VTK¹⁵.
- ITK/Modules/Registration — classes for registration of images or other data structures to each other.
- ITK/Modules/Segmentation — classes for segmentation of images or other data structures.
- ITK/Modules/Video — classes for input, output and processing of static and real-time data with temporal components.

¹³<http://vxl.sourceforge.net>

¹⁴<http://opencv.org>

¹⁵<http://www.vtk.org>

- ITK/Modules/Compatibility — collects together classes for backwards compatibility with ITK Version 3, and classes that are deprecated – i.e. scheduled for removal from future versions of ITK.
- ITK/Modules/Remote — a group of modules distributed outside of the main ITK source repository (most of them are hosted on github.com) whose source code can be downloaded via CMake when configuring ITK.
- ITK/Modules/External — a directory to place in development or non-publicized modules.
- ITK/Modules/Numerics — a collection of numeric modules, including FEM, Optimization, Statistics, Neural Networks, etc.

The Doxygen documentation is an essential resource when working with ITK, but it is not contained in a separate repository. Each ITK class is implemented with a .h and .cxx/.hxx file (.hxx 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. Doxygen uses this header documentation to produce its HTML output.

The extensive Doxygen web pages describe in detail every class and method in the system. It also contains inheritance and collaboration diagrams, listing of event invocations, and data members, heavily hyper-linked to other classes and to the source code. The nightly generated Doxygen documentation is online at <https://itk.org/Doxygen/html/>. Archived versions for each feature release are also available online; for example, the documentation for the 4.4.0 release are available at <https://itk.org/Doxygen44/html/>.

The ITKApps contains large, relatively complex examples of ITK usage. See the web pages at <https://www.itk.org/ITK/resources/applications.html> for a description. Some of these applications require GUI toolkits such as Qt and FLTK or other packages such as VTK (*The Visualization Toolkit*¹⁶). It is recommended to set the CMake source directory to ITKApps/Superbuild to build the dependent third-party applications.

1.5 The Insight Community and Support

Joining the community mailing list is strongly recommended. This is one of the primary resources for guidance and help regarding the use of the toolkit. You can subscribe to the community list online at

<https://www.itk.org/ITK/help/mailing.html>

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. For details on participation, see Part III of this book.

¹⁶<http://www.vtk.org>

- Interaction with other community members is encouraged on the mailing lists by both asking as answering questions. When issues are discovered, patches submitted to the code review system are welcome. Performing code reviews, even by novice members, is encouraged. Improvements and extensions to the documentation are also welcome.
- 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 ¹⁷. Kitware also offers short ITK courses either at a site of your choice or periodically at Kitware offices.
- 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. Check the Wiki¹⁸ for a listing.

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 Ibañez 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 Mollholt 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. The NLM has funded maintenance of the toolkit over the years, and a major funding effort was started in July 2010 that culminated with the release of ITK 4.0.0 in December 2011. 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.

¹⁷<http://www.kitware.com>

¹⁸<https://itk.org/Wiki/ITK/Documentation>

CONFIGURING AND BUILDING ITK

This chapter describes the process for configuring and compiling ITK on your system. Keep in mind that ITK is a toolkit, and as such, once it is installed on your computer it does not provide an application to run. What ITK does provide is a large set of libraries which can be used to create your own applications. Besides the toolkit proper, ITK also includes an extensive set of examples and tests that introduce ITK concepts and show how to use ITK in your own projects.

Some of the examples distributed with ITK depend on third party libraries, some of which may need to be installed separately. For the initial build of ITK, you may want to ignore these extra libraries and just compile the toolkit itself.

ITK has been developed and tested across different combinations of operating systems, compilers, and hardware platforms including Microsoft Windows, Linux on various architectures, Solaris/UNIX, Mac OSX, and Cygwin. Kitware is committed to support the following compilers for building ITK:

- GCC 4.x
- Visual Studio 8 SP 1 (until 2015), 9 (until 2018), 10 (until 2020)
- Intel Compiler Suite 11.x, 12.x (including Mac OS X release)
- Darwin-c++-4.2 PPC (until 2015), x86_64
- Win32-mingw-gcc-4.5
- Clang 3.3 and later

If you are currently using an outdated compiler this may be an excellent excuse for upgrading this old piece of software! Support for different platforms is evident on the ITK quality dashboard (see Section [10.2](#) on page [228](#)).

2.1 Using CMake for Configuring and Building ITK

The challenge of supporting ITK across platforms has been solved through the use of CMake¹, a cross-platform, open-source build system. CMake controls the software compilation process with simple platform and compiler-independent configuration files. CMake is quite sophisticated—it supports complex environments requiring system introspection, compiler feature testing, and code generation.

CMake generates native Makefiles or workspaces to be used with the corresponding development environment of your choice. For example, on UNIX and Cygwin systems, CMake generates Makefiles; under Microsoft Windows CMake generates Visual Studio workspaces; CMake is also capable of generating appropriate build files for other development environments, e.g., Eclipse. The information used by CMake is provided in `CMakeLists.txt` files that are present in every directory of the ITK source tree. Along with the specification of project structure and code dependencies these files specify the information that need to be provided to CMake by the user during project configuration stage. Typical configuration options specified by the user include paths to utilities installed on your system and selection of software features to be included.

An ITK build requires only CMake and a C++ compiler. ITK ships with all the third party library dependencies required, and these dependencies are used during compilation unless the use of a system version is requested during CMake configuration.

2.1.1 Preparing CMake

CMake can be downloaded at no cost from

<http://www.cmake.org/cmake/resources/software.html>

You can download binary versions for most of the popular platforms including Microsoft Windows, Mac OSX, Linux, PowerPC and IRIX. Alternatively you can download the source code and build CMake on your system. Follow the instructions provided on the CMake web page for downloading and installing the software. The minimum version of CMake has been evolving along with the version of ITK. For example, the current version of ITK (4.10) requires the minimum CMake version to be 2.8.8.

CMake provides a terminal-based interface (Figure 2.1) on platforms support the `curses` library. For most platforms CMake also provides a GUI based on the `Qt` library. Figure 2.1 shows the terminal-based CMake interface for Linux and CMake GUI for Microsoft Windows.

Running CMake to configure and prepare for compilation a new project initially requires two pieces of information: where the source code directory is located, and where the compiled code is to be produced. These are referred to as the *source directory* and the *binary directory* respectively. We

¹www.cmake.org

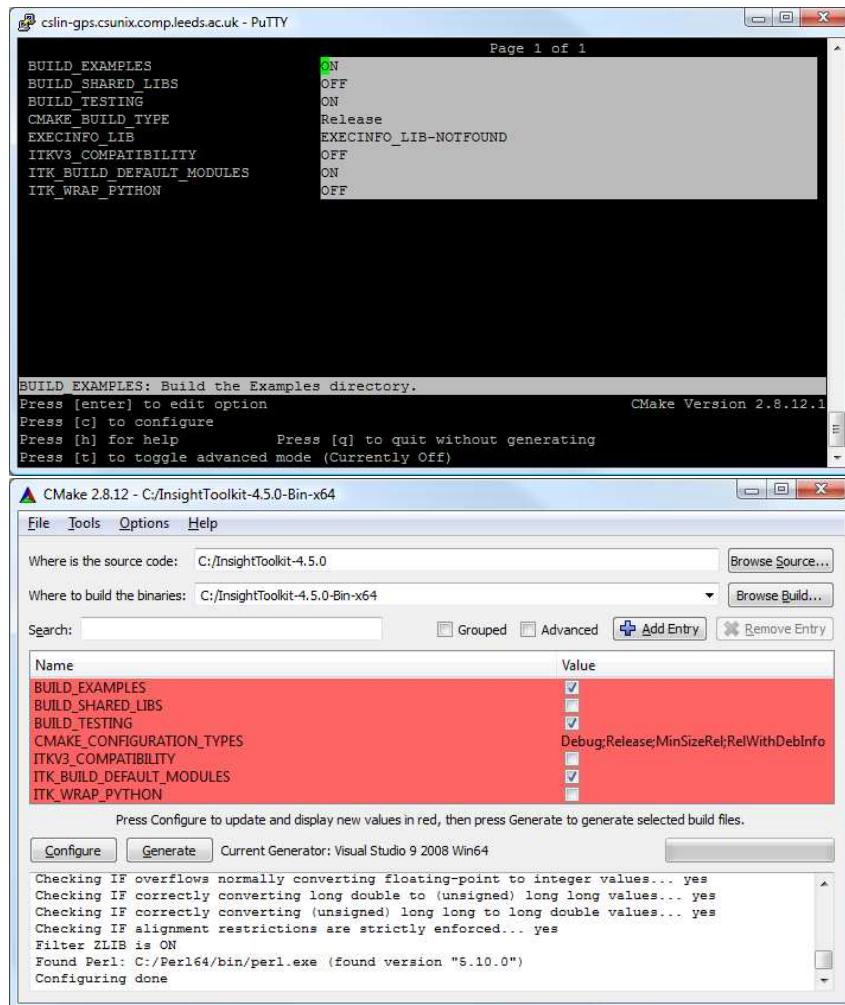


Figure 2.1: CMake user interfaces: at the top is the interface based on the `curses` library supported by UNIX/Linux systems, below is the Microsoft Windows version of the CMake GUI based on the Qt library (CMake GUI is also available on UNIX/Linux systems).

recommend setting the binary directory to be different than the source directory in order to produce an *out-of-source* build.

If you choose to use the terminal-based version of CMake (`ccmake`) the binary directory needs to be created first and then CMake is invoked from the binary directory with the path to the source directory. For example:

```
mkdir ITK-build  
cd ITK-build  
ccmake .. /ITK
```

In the GUI version of CMake (`cmake-gui`) the source and binary directories are specified in the appropriate input fields (Figure 2.1) and the application will request a confirmation to create a new binary directory if it does not exist.

CMake runs in an interactive mode which allows iterative selection of options followed by configuration according to the updated options. This iterative process proceeds until no more options remain to be specified. At this point, a generation step produces the appropriate build files for your configuration.

This interactive configuration process can be better understood by imagining the traversal of a path in a decision tree. Every selected option 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

Start terminal-based CMake interface `ccmake` on Linux and UNIX, or the graphical user interface `cmake-gui` on Microsoft Windows. Remember to run `ccmake` from the binary directory on Linux and UNIX. On Windows, specify the source and binary directories in the GUI, then set and modify the configuration and build option in the interface as necessary.

The examples distributed with the toolkit provide a helpful resource for learning how to use ITK components but are not essential for compiling the toolkit itself. The testing section of the source tree includes a large number of small programs that exercise the capabilities of ITK classes. Enabling the compilation of the examples and unit tests will considerably increase the build time. In order to speed up the build process, you can disable the compilation of the unit tests and examples. This is done by setting the variables `BUILD_TESTING` and `BUILD_EXAMPLES` to OFF.

Most CMake variables in ITK have sensible default values. Each time a CMake variable is changed, it is necessary to re-run the configuration step. In the terminal-based version of the interface the configuration step is triggered by hitting the “c” key. In the GUI version this is done by clicking on the “Configure” button.

When no new options appear highlighted in CMake, you can proceed to generate Makefiles, a Visual Studio workspace or other appropriate build files depending on your preferred development environment. This is done in the GUI interface by clicking on the “Generate” button. In the terminal-based version this is done by hitting the “g” key. After the generation process the terminal-based version of CMake will quit silently. The GUI window of CMake can be left open for further refinement of configuration options as described in the next section. With this scenario it is important to generate new build files to reflect the latest configuration changes. In addition, the new build files need to be

reloaded if the project is open in the integrated development environment such as Visual Studio or Eclipse.

2.1.3 Advanced Module Configuration

Following the default configuration introduced in [2.1.2](#), the majority of the toolkit will be built. The modern modular structure of the toolkit makes it possible to customize the ITK library by choosing which modules to include in the build. ITK was officially modularized in version 4.0.0 released in December of 2011. Developers have been testing and improving the modular structure since then. The toolkit currently contains more than 100 regular/internal modules and many remote modules, while new ITK modules are being developed.

`ITK_BUILD_DEFAULT_MODULES` is the CMake option to build all default modules in the toolkit, by default this option is `ON` as shown in [Figure 2.1](#). The default modules include most internal ITK modules except the ones that depend on external third party libraries (such as `ITKVtkGlue`, `ITKBridgeOpenCV`, `ITKBridgeVXL`, etc.) and several modules containing legacy code (`ITKReview`, `ITKDeprecated` and `ITKv3Compatibility`).

Apart from the default mode of selecting the modules for building the ITK library there are two other approaches module selection: the group mode, and the advanced module mode. When `ITK_BUILD_DEFAULT_MODULES` is set to `OFF`, the selection of modules to be included in the ITK library can be customized by changing the variables enabling group and advanced module selection.

`ITKGroup_{group name}` variables for group module selection are visible when `ITK_BUILD_DEFAULT_MODULES` is `OFF`. The ITK source code tree is organized in such way that a group of modules characterised by close relationships or similar functionalities stay in one subdirectory. Currently there are 11 groups (excluding the External and Remote groups). The CMake `ITKGroup_{group name}` options are created for the convenient enabling or disabling of multiple modules at once. The `ITKGroup_Core` group is selected by default as shown in [Figure 2.2](#). When a group is selected, all modules in the group and their depending modules are enabled. When a group variable is set to `OFF`, all modules in the group, except the ones that are required by other enabled modules, are disabled.

If you are not sure about which groups to turn on, but you do have a list of specific modules to be included in your ITK library, you can certainly skip the Group options and use the `Module_{module name}` options only. Whatever modules you select, their dependent modules are automatically enabled. In the advanced mode of the CMake GUI, you can manually toggle the build of the non-default modules via the `Module_{module name}` variables. In [Figure 2.3](#) all default modules' `Module_{module name}` variables are shown disabled for toggling since they are enabled via the `ITK_BUILD_DEFAULT_MODULES` set to `ON` variable.

However, not all modules will be visible in the CMake GUI at all times due to the various levels of controls in the previous two modes. If some modules are already enabled by other modes, these modules are set as internal variables and are hidden in the CMake GUI. For example, `Module_ITKFoo` variable is hidden when the module `ITKFoo` is enabled in either of the following scenarios:

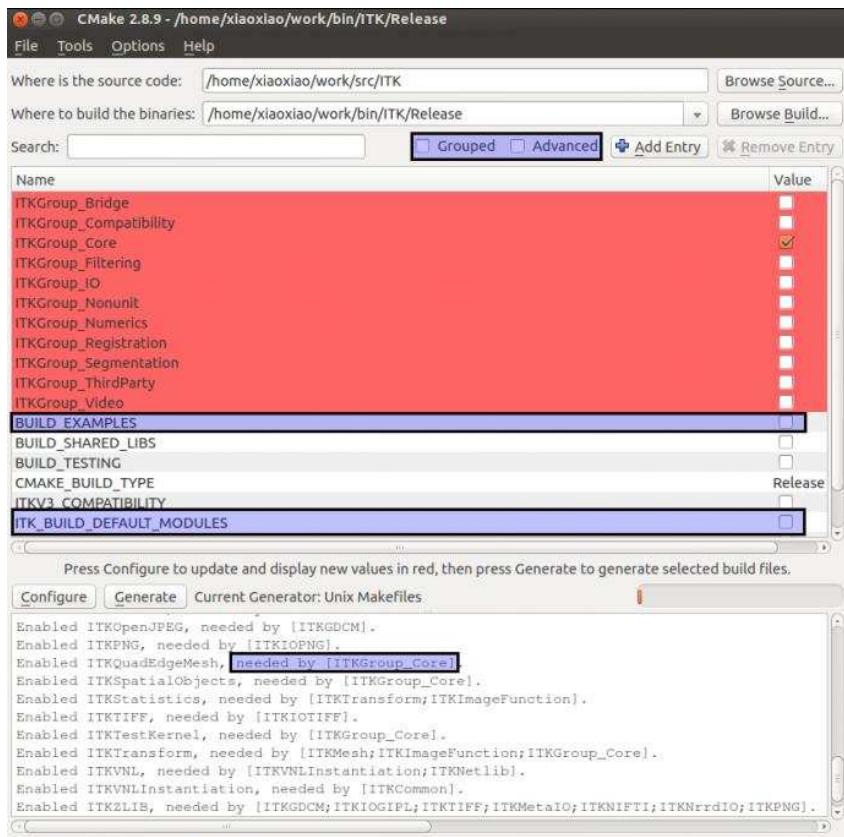


Figure 2.2: CMake GUI shows the ITK Group options.

1. module `ITKBar` is enabled and depends on `ITKFoo`,
2. `ITKFoo` belongs to the group `ITKGroup_FooAndBar` and the group is enabled
3. `ITK_BUILD_DEFAULT_MODULES` is ON and `ITKFoo` is a default module.

To find out why a particular module is enabled, check the CMake configuration messages where the information about enabling or disabling the modules is displayed (Figure 2.3); these messages are sorted in alphabetical order by module names.

2.1.4 Compiling ITK

To initiate the build process after generating the build files on Linux or UNIX, simply type `make` in the terminal if the current directory is set to the ITK binary directory. If using Visual Studio,

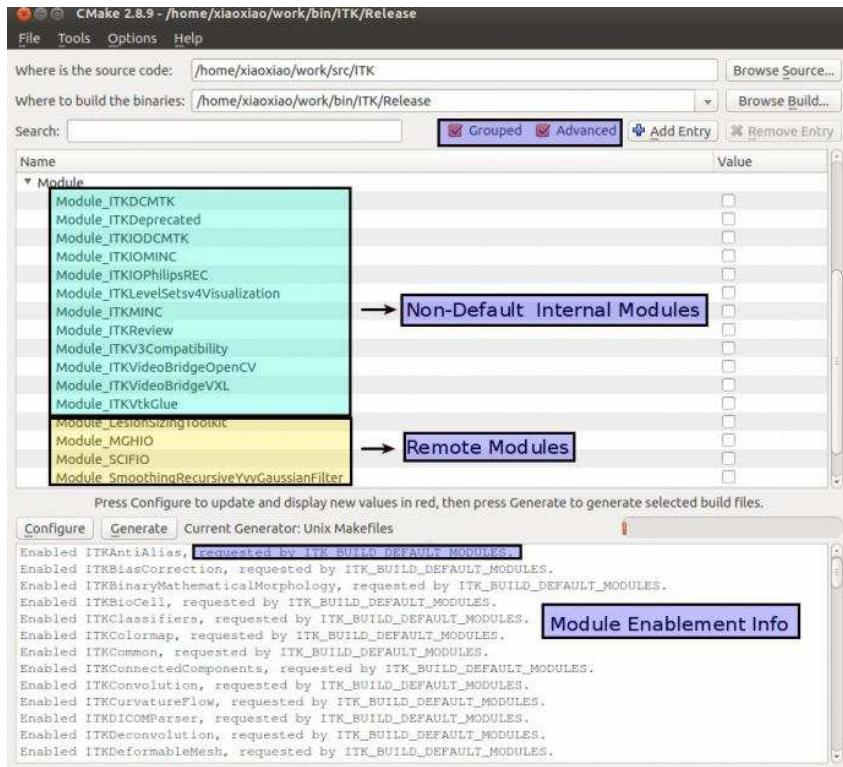


Figure 2.3: CMake GUI for configuring ITK: the advanced mode shows options for non-default ITK Modules.

first load the workspace named `ITK.sln` from the binary directory specified in the CMake GUI and then start the build by selecting “Build Solution” from the “Build” menu or right-clicking on the `ALL_BUILD` target in the Solution Explorer pane and selecting the “Build” context menu item.

The build process can take anywhere from 15 minutes to a couple of hours, depending on the build configuration and the performance of your system. If testing is enabled as part of the normal build process, about 2400 test programs will be compiled. In this case, you will then need to run `ctest` to verify that all the components of ITK have been correctly built on your system.

2.1.5 Installing ITK on Your System

When the build process is complete an ITK binary distribution package can be generated for installation on your system or on a system with compatible specifications (such as hardware platform and operating system) as well as suitable development environment components (such as C++ compiler and CMake). The default prefix for installation destination directory needs to be specified during CMake configuration process prior to compiling ITK. The installation destination prefix can be

set through the CMake cache variable `CMAKE_INSTALL_PREFIX`.

Typically distribution packages are generated to provide a “clean” form of the software which is isolated from the details of the build process (separate from the source and build trees). Due to the intended use of ITK as a toolkit for software development the step of generating ITK binary packages for installing ITK on other systems has limited application and thus it can be treated as optional. However, the step for generating binary distribution packages has a much wide application for distributing software developed with ITK. Further details on configuring and generating binary packages with CMake can be found in the CMake tutorial².

2.2 Getting Started With ITK

The simplest way to create a new project with ITK is to create two new directories somewhere in your disk, one to hold the source code and one to hold the binaries and other files that are created in the build process. For this example, create a `HelloWorldITK` directory to hold the source and a `HelloWorldITK-build` directory to hold the binaries. The first file to place in the source directory is a `CMakeLists.txt` file that will be used by CMake to generate a Makefile (if you are using Linux or UNIX) or a Visual Studio workspace (if you are using Microsoft Windows). The second source file to be created 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/Linux, you can `cd` to your newly created binary directory and launch the terminal-based version of CMake by entering “`ccmake/HelloWorldITK`” in the terminal. Note the “`..../HelloWorldITK`” in the command line to indicate that the `CMakeLists.txt` file is up one directory and in `HelloWorldITK`. In CMake GUI which can be used under Microsoft Windows and UNIX/Linux, the source and binary directories will have to be specified prior to the configuration and build file generation process.

Both the terminal-based and GUI versions of CMake will require you to specify the directory where ITK was built in the CMake variable `ITK_DIR`. The ITK binary directory will contain a file named `ITKConfig.cmake` generated during ITK configuration process with CMake. From this file, CMake will recover all information required to configure your new ITK project.

After generating the build files, on UNIX/Linux systems the project can be compiled by typing `make` in the terminal provided the current directory is set to the project’s binary directory. In Visual Studio on Microsoft Windows the project can be built by loading the workspace named `HelloWorldITK.sln` from the binary directory specified in the CMake GUI and selecting “Build Solution” from the “Build” menu or by right-clicking on the `ALL_BUILD` target in the Solution Explorer pane and selecting the “Build” context menu item.

The resulting executable, which will be called `HelloWorld`, can be executed on the command line. If on Microsoft Windows, please note that double-clicking on the icon of the executable will quickly

² http://www.cmake.org/cmake/help/cmake_tutorial.html

launch a command line window, run the executable and close the window right away, not giving you time to see the output. It is therefore preferable to run the executable from the DOS command line by starting the cmd.exe shell first.

2.2.1 Hello World!

This section provides and explains the contents of the two files which need to be created for your new project. These two files can be found in the ITK/Examples/Installation directory.

The CMakeLists.txt file contains the following lines:

```
project(HelloWorld)

find_package(ITK REQUIRED)
include(${ITK_USE_FILE})

add_executable(HelloWorld HelloWorld.cxx)

target_link_libraries(HelloWorld ${ITK_LIBRARIES})
```

The first line defines the name of your project as it appears in Visual Studio or Eclipse; this line will have no effect with UNIX/Linux Makefiles. The second line loads a CMake file with a predefined strategy for finding ITK. If the strategy for finding ITK fails, CMake will report an error which can be corrected by providing the location of the directory where ITK was compiled or installed on your system. In this case the path to the ITK's binary/installation directory needs to be specified as the value of the `ITK_DIR` CMake variable. The line `include(${USE_ITK_FILE})` loads the `UseITK.cmake` file which contains the configuration information about the specified ITK build. The line starting with `add_executable` call defines as its first argument the name of the executable that will be produced as result of this project. The remaining argument(s) of `add_executable` are the names of the source files to be compiled. Finally, the `target_link_libraries` call specifies which ITK libraries will be linked against this project. Further details on creating and configuring CMake projects can be found in the CMake tutorial³ and CMake online documentation⁴.

The source code for this section can be found in the file `HelloWorld.cxx`.

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

³ http://www.cmake.org/cmake/help/cmake_tutorial.html

⁴ <http://www.cmake.org/cmake/help/documentation.html>

```
#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 EXIT_SUCCESS;
}
```

This code instantiates a 3D image⁵ 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 SmartPointers 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.

By this point you have successfully configured and compiled ITK, and created your first simple program! If you have experienced any difficulties while following the instructions provided in this section, please join the community mailing list (see Section 1.5 on page 8) and post questions there.

⁵Also known as a *volume*.

Part II

Architecture

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.

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 routines¹.

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 `itk::Histogram` and `itk::SpatialObject`.

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).

¹<http://www.netlib.org>

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 Python. The CastXML² tool is used to produce an XML description of arbitrarily complex C++ code. An interface generator script is then used to transform the XML description into wrappers using the SWIG³ package.

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 [8]. The idea is to make software that is capable of “plugging together” in

²<https://github.com/CastXML/CastXML>

³<http://www.swig.org/>

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 [1].

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., a `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 the analysis performed at compile time increases the time to build an application. Also, the increased complexity may produce difficult to decipher error messages due to even the simplest syntax errors. For those unfamiliar 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 file (`.h`) and an implementation file (`.cxx`) if defining a non-templated class, and a `.hxx` file if defining 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 `Modules/Core/Common/include` directory and defines standard system-wide macros (such as `Set/Get`, constants, and other parameters).

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

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 stack. (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 is not derived from `LightObject`. In this case instances may be created on the stack. An example of such a class is the `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, a particular application may wish to deploy its own class implemented using specialized image processing hardware (i.e., to realize a performance gain). By using the object factory mechanism, it is possible to replace the creation of a particular ITK filter at run-time with such a custom class. (Of course, the class must provide the exact same API as the one it is replacing.). For this, the user compiles his 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 `Modules/Core/Common/include/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, 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 causes memory leaks (or excessive memory consumption). 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 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 automatically 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 in 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
    // "interp" SmartPointer.
    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. Raw pointers are preferred for multi-threaded sections of code.

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
}
```

A particular class may throw an exception 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, all ITK exceptions derive 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 [3] (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 using an 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 `Modules/Core/Common/include/itkCommand.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 classes 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_lbfq` 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.⁴ The National Institute of Standards and Technology (NIST) provides an excellent search interface to this repository in its *Guide to Available Mathematical Software (GAMS)*,⁵ 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 `itk::Image` and `itk::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 31 for more information).

`itk::Image` represents an n -dimensional, regular sampling of data. The sampling direction is parallel to direction matrix 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, most applications will use a C++ primitive type (e.g., `int`, `float`) or a pre-defined pixel type and will rarely create a new type of pixel class.

⁴<http://www.netlib.org/>

⁵<http://gams.nist.gov>

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 `itk::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<FloatImage2DType>::Pointer writer;
writer = itk::ImageFileWriter<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 to execute 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, the `itk::TubeSpatialObject` indicates the point where it is connected with its parent tube.

There are a limited number of spatial objects 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 segmentations of a liver.

3.7 Wrapping

While the core of ITK is implemented in C++, Python bindings can be automatically generated and ITK programs can be created using Python. The wrapping process in ITK is capable of handling generic programming (i.e., extensive use of C++ templates). Systems like VTK, which use their own wrapping facility, are non-templated and customized to the coding methodology found in the system, like object ownership conventions. 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. CastXML is a Clang-based tool that produces an XML description of an input C++ program.
2. The `igenerator.py` script in the ITK source tree processes XML information produced by CastXML and generates standard input files (*.i files) to the next tool (SWIG), indicating what is to be wrapped and how to wrap it.

3. SWIG produces the appropriate Python bindings.

To learn more about the wrapping process, please see the section on module wrapping, Section 9.5. The wrapping process is orchestrated by a number of CMake macros found in the Wrapping directory. The result of the wrapping process is a set of shared libraries (.so in Linux or .dlls on Windows) that can be used by interpreted languages.

There is almost a direct translation from C++, with the differences being the particular syntactical requirements of each language. For example, to dilate an image using a custom structuring element using the Python wrapping:

```
inputImage = sys.argv[1]
outputImage = sys.argv[2]
radiusValue = int(sys.argv[3])

PixelType = itk.UC
Dimension = 2
ImageType = itk.Image[PixelType, Dimension]

reader = itk.ImageFileReader[ImageType].New()
reader.SetFileName(inputImage)

StructuringElementType = itk.FlatStructuringElement[Dimension]
structuringElement = StructuringElementType.Ball(radiusValue)

dilateFilter = itk.BinaryDilateImageFilter[
    ImageType, ImageType, StructuringElementType].New()
dilateFilter.SetInput(reader.GetOutput())
dilateFilter.SetKernel(structuringElement)
```

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

```
const char * inputImage = argv[1];
const char * outputImage = argv[2];
const unsigned int radiusValue = atoi( argv[3] );

typedef unsigned char PixelType;
const unsigned int Dimension = 2;

typedef itk::Image< PixelType, Dimension >     ImageType;
typedef itk::ImageFileReader< ImageType >      ReaderType;
ReaderType::Pointer reader = ReaderType::New();
reader->SetFileName( inputImage );

typedef itk::FlatStructuringElement< Dimension >
    StructuringElementType;
StructuringElementType::RadiusType radius;
radius.Fill( radiusValue );
StructuringElementType structuringElement =
    StructuringElementType::Ball( radius );

typedef itk::BinaryDilateImageFilter< ImageType, ImageType,
    StructuringElementType > BinaryDilateImageFilterType;

BinaryDilateImageFilterType::Pointer dilateFilter =
    BinaryDilateImageFilterType::New();
dilateFilter->SetInput( reader->GetOutput() );
dilateFilter->SetKernel( structuringElement );
```

This example demonstrates an important difference between C++ and a wrapped language such as Python. 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 `ImageFileReader[ImageType]` indicates that this class, implementing an image source, has been instantiated using an input and output image type of two-dimensional unsigned char values (i.e., UC). To see the types available for a given filter, use the `.GetTypes()` method.

```
print(itk.ImageFileReader.GetTypes())
```

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, re-run the wrapping process to produce new libraries. Some high-level options for these types, such as common pixels types and image dimensions, are specified during CMake configuration. The types of specific classes that should be instantiated, based on these basic options, are defined by the `*.wrap` files in the `wrapping` directory of a module.

Conversion of common, basic wrapped ITK classes to native Python types is supported. For example, conversion between the `itk::Index` and Python list or tuple is possible:

```
Dimesion = 3
index = itk.Index[Dimension]()
index_as_tuple = tuple(index)
index_as_list = list(index)
```

```
region = itk.ImageRegion[Dimension]()
region.SetIndex((0, 2, 0))
```

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 functionalities. For example, the Python ecosystem provides a variety of powerful 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. Other languages like Java, Ruby, Tcl could also be wrapped in the future.

3.7.1 Python Setup

In order to access the Python interface of ITK, make sure to compile with the CMake `ITK_WRAP__PYTHON` option. In addition, choose which pixel types and dimensions to build into the wrapped interface. Supported pixel types are represented in the CMake configuration as variables named `ITK__WRAP_<pixel type>`. Supported image dimensions are enumerated in the semicolon-delimited list `ITK_WRAP_DIMS`, the default value of which is `2;3` indicating support for 2- and 3-dimensional images. The Release CMake build configuration is recommended.

After configuration, check to make sure that the values of the following variables are set correctly:

- `PYTHON_INCLUDE_DIR`
- `PYTHON_LIBRARY`
- `PYTHON_EXECUTABLE`

particularly if there are multiple Python installations on the system.

Python wrappers can be accessed from the build tree without installing the library. An environment to access the `itk` Python module can be configured using the Python `virtualenv` tool, which provides an isolated working copy of Python without interfering with Python installed at the system level. Once the `virtualenv` package is installed on your system, create the virtual environment within the directory ITK was built in. Copy the `WrapITK.pth` file to the `lib/python2.7/site-packages` on Unix and `Lib/site-packages` on Windows, of the `virtualenv`. For example,

```
virtualenv --system-site-packages wrapitk-venv
cd wrapitk-venv/lib/python2.7/site-packages
cp /path/to/ITK-Wrapped/Wrapping/Generators/Python/WrapITK.pth .
cd ../../../../wrapitk-venv/bin
./python /usr/bin/ipython
import itk
```

On Windows, it is also necessary to add the ITK build directory containing the `.dll` files to your `PATH` environmental variable if ITK is built with the CMake option `BUILD_SHARED_LIBS` enabled.

For example, the directory containing .dll files for an ITK build at `C:\ITK-build` when built with Visual Studio in the `Release` configuration is `C:\ITK-build\bin\Release`.

CHAPTER
FOUR

DATA REPRESENTATION

This chapter introduces the basic classes responsible for representing data in ITK. The most common classes are `itk::Image`, `itk::Mesh` and `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 `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 is defined by the `Index`. The extent, or size, of the region is defined by the `Size`. 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 `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.

```
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 as the image (using the `itk::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 `itk::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 `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 minimal 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 using the `itk::ImageIOBase` class (a list of possibilities can be found in the inheritance diagram of this class.).

```
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 `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 6 on page 141 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 dimension of the image. The `IndexType` is automatically defined by the image and can be accessed using the scope operator `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 no SmartPointer is used to access the `Index`. This is because `Index` is a lightweight 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.

```
const ImageType::IndexType pixelIndex = {{27,29,37}}; // Position of {X,Y,Z}
```

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 `Image4.cxx`.

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, voxel directions (i.e. orientation), and spacing are fundamental to many applications. Registration, for example, is performed in physical coordinates. Improperly defined spacing, direction, 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,

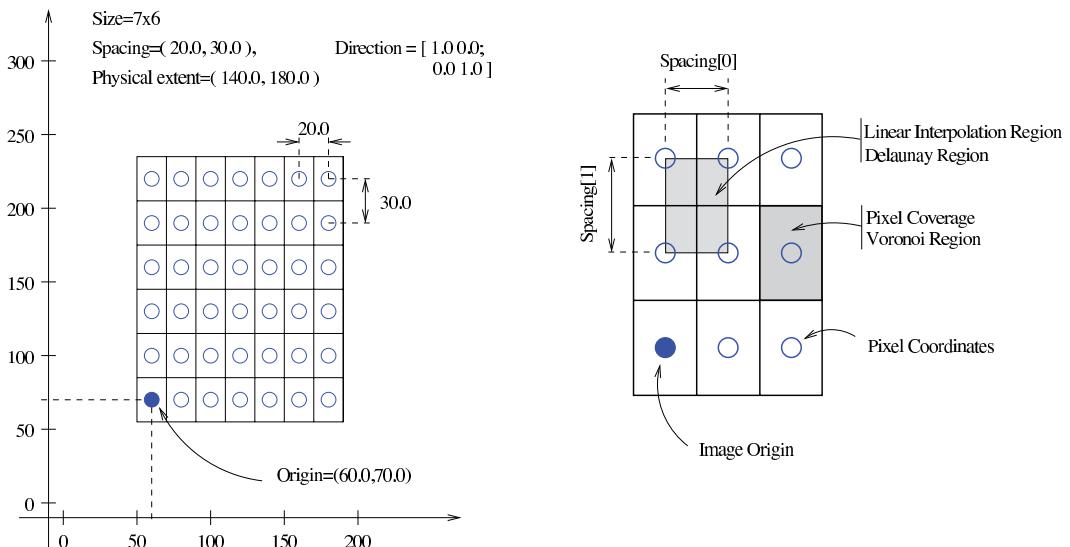


Figure 4.1: Geometrical concepts associated with the ITK image.

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. For this simplified example, the voxel lattice is perfectly aligned with physical space orientation, and the image direction is therefore an identity mapping. If the voxel lattice samples were rotated with respect to physical space, then the image direction would contain a rotation matrix.

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 `itk::Image` class for dealing with spacing and origin.

```
ImageType::SpacingType spacing;

// 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.

```
// coordinates of the center of the first pixel in N-D
ImageType::PointType newOrigin;
newOrigin.Fill(0.0);
image->SetOrigin( newOrigin );
```

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 & origin = image->GetOrigin();

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

The image direction matrix represents the orientation relationships between the image samples and physical space coordinate systems. The image direction matrix is an orthonormal matrix that describes the possible permutation of image index values and the rotational aspects that are needed to properly reconcile image index organization with physical space axis. The image directions is a $N \times N$ matrix where N is the dimension of the image. An identity image direction indicates that increasing values of the 1st, 2nd, 3rd index element corresponds to increasing values of the 1st, 2nd and 3rd physical space axis respectively, and that the voxel samples are perfectly aligned with the physical space axis.

The following code illustrates the creation and assignment of a variable suitable for initializing the image direction with an identity.

```
// coordinates of the center of the first pixel in N-D
ImageType::DirectionType direction;
direction.SetIdentity();
image->SetDirection( direction );
```

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

```
const ImageType::DirectionType& direct = image->GetDirection();

std::cout << "Direction = " << std::endl;
std::cout << direct << std::endl;
```

Once the spacing, origin, and direction of the image samples 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 `itk::Point` class, like an `itk::Index`, is a relatively small and simple object. This means that no `itk::SmartPointer` is used here and the objects are simply declared as instances, like 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 $\{0, \text{Dimension} - 1\}$.

```
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.

```
const 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.

The following example illustrates the mathematical relationships between image index locations and its corresponding physical point representation for a given Image.

Let us imagine that a graphical user interface exists where the end user manually selects the voxel index location of the left eye in a volume with a mouse interface. We need to convert that index location to a physical location so that laser guided surgery can be accurately performed. The `TransformIndexToPhysicalPoint` method can be used for this.

```
const ImageType::IndexType LeftEyeIndex = GetIndexFromMouseClick();
ImageType::PointType LeftEyePoint;
image->TransformIndexToPhysicalPoint(LeftEyeIndex, LeftEyePoint);
```

For a given index I_{3X1} , the physical location P_{3X1} is calculated as following:

$$P_{3X1} = O_{3X1} + D_{3X3} * \text{diag}(S_{3X1})_{3x3} * I_{3X1} \quad (4.1)$$

where D is an orthonormal direction cosines matrix and S is the image spacing diagonal matrix.

In matlab syntax the conversions are:

```
% Non-identity Spacing and Direction
spacing=diag( [0.9375, 0.9375, 1.5] );
direction=[0.998189, 0.0569345, -0.0194113;
0.0194429, -7.38061e-08, 0.999811;
0.0569237, -0.998378, -0.00110704];
point = origin + direction * spacing * LeftEyeIndex
```

A corresponding mathematical expansion of the C/C++ code is:

```

typedef itk::Matrix<double, Dimension, Dimension> MatrixType;
MatrixType SpacingMatrix;
SpacingMatrix.Fill( 0.0F );

const ImageType::SpacingType & ImageSpacing = image->GetSpacing();
SpacingMatrix( 0,0 ) = ImageSpacing[0];
SpacingMatrix( 1,1 ) = ImageSpacing[1];
SpacingMatrix( 2,2 ) = ImageSpacing[2];

const ImageType::DirectionType & ImageDirectionCosines =
    image->GetDirection();
const ImageType::PointType & ImageOrigin = image->GetOrigin();

typedef itk::Vector< double, Dimension > VectorType;
VectorType LeftEyeIndexVector;
LeftEyeIndexVector[0]= LeftEyeIndex[0];
LeftEyeIndexVector[1]= LeftEyeIndex[1];
LeftEyeIndexVector[2]= LeftEyeIndex[2];

ImageType::PointType LeftEyePointByHand =
    ImageOrigin + ImageDirectionCosines * SpacingMatrix * LeftEyeIndexVector;

```

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 [7, 9]. The human retina possess different types of light sensitive cells. Three of them, known as *cones*, are sensitive to color [5] 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 circadian² 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 [6].

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

$$ColorA = ColorB - ColorC \quad (4.2)$$

is just a way of saying that we can produce *ColorB* by combining *ColorA* and *ColorC*. However,

¹The human eye is capable of perceiving a single isolated photon.

²The term *Circadian* refers to the cycle of day and night, that is, events that are repeated with 24 hours intervals.

we must be aware that (at least in emitted light) it is not possible to *subtract light*. So when we mention Equation 4.2 we actually mean

$$\text{ColorB} = \text{ColorA} + \text{ColorC} \quad (4.3)$$

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 [9].

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 [9].

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` lacks many of the operators that may be naively expected from it. In particular, there are no defined operations for subtraction or addition.

When you intend to find the “Mean” of two `RGBType` pixels, you are assuming that the color in the visual “middle” of the two input pixels can be calculated through a linear operation on 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 [7].

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
`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 `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; // z 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 `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.

Here 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 `itk::ImportImageFilter` class must be included.

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

Next, we select the data type used 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.

```
const itk::SpacePrecisionType origin[ Dimension ] = { 0.0, 0.0, 0.0 };
importFilter->SetOrigin( origin );
```

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

```
// spacing isotropic volumes to 1.0
const itk::SpacePrecisionType spacing[ Dimension ] = { 1.0, 1.0, 1.0 };
importFilter->SetSpacing( spacing );
```

Next we allocate the memory block containing the pixel data to be passed to the

`ImportImageFilter`. 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 an `itk::ImageIterator` that supports 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()` method. 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.

```
typedef itk::ImageFileWriter< ImageType > WriterType;
WriterType::Pointer writer = WriterType::New();

writer->SetFileName( argv[1] );
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 `PointSet1.cxx`.

The `itk::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::itk::Mesh` providing the methods necessary to manipulate sets of points. 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 the traits of the `PointSet` class is `PointType`, which is 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 $\{0, \text{Dimension} - 1\}$. 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 number_of_points = pointsSet->GetNumberOfPoints();
std::cout << number_of_points << 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 `PointSet2.cxx`.

The `itk::PointSet` class uses an internal container to manage the storage of `itk::Points`. 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 declares it in the global namespace.

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

The actual type of `PointsContainer` depends on what style of `PointSet` is being used. The dynamic `PointSet` uses `itk::MapContainer` while the static `PointSet` uses `itk::VectorContainer`. The vector and map containers are basically ITK wrappers around the STL classes `std::map` and `std::vector`. By default, `PointSet` uses a static style, and therefore the default type of point container is `VectorContainer`. Both map and vector containers are templated over the type of element they contain. In this case they are templated over `PointType`. Containers are reference counted objects, created with the `New()` method and assigned to a `itk::SmartPointer`. 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 assigned to 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.

```
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.

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

```

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 `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 `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 generates a circle and assigns 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 * itk::Math::pi / 180.0;
    point[0] = radius * std::sin( angle );
    point[1] = radius * std::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 )
{
    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 )
{
    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 `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 the 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"

```

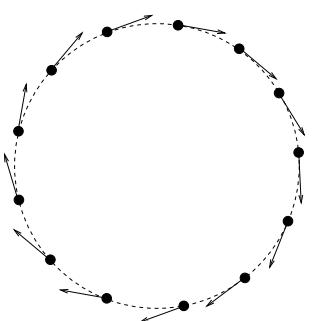


Figure 4.2: Vectors as PixelType.

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`.

```

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 * itk::Math::pi / 180.0;
    point[0] = radius * std::sin( angle );
    point[1] = radius * std::cos( angle );
    point[2] = 1.0; // flat on the Z plane
    tangent[0] = std::cos(angle);
    tangent[1] = -std::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 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 `PointSetWithCovariantVectors.cxx`.

It is common to represent geometric objects 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 it behaves 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 example demonstrates how a `CovariantVector` can be used as the `PixelType` for the `PointSet` class. The example illustrates how a deformable model could move under the influence of the gradient of a 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 circle 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 * std::atan(1.0) / 45.0;
    point[0] = radius * std::sin( angle );
    point[1] = radius * std::cos( angle );
    point[2] = 1.0; // flat on the Z plane
    gradient[0] = std::sin(angle);
    gradient[1] = std::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 )
{
    point      = pointIterator.Value();
    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 `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.

As with `PointSet`, a `Mesh` object may be *static* or *dynamic*. The first is used when the number of points in the set is known in advance and 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. In addition to point management, the distinction facilitates

optimization of performance and memory 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 PixelType, spatial dimension, and (optionally) a parameter set called MeshTraits. PixelType is the type of the value associated with each point (just as is done with PointSet). The following illustrates a typical instantiation of Mesh.

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

Meshes typically require large amounts of memory. For this reason, they are reference counted objects, managed using [itk::SmartPointers](#). The following line illustrates how a mesh is created by invoking the `New()` method on MeshType and assigning the result to a SmartPointer.

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

Management of points in a Mesh is identical to that in a PointSet. The type of 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 into the Mesh using the `SetPoint()` method. Note that points are copied into the mesh structure, meaning 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 a 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 was done in the previous section for the PointSet.

```
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 is used to advance the iterator from one point to the next. The value associated with the `Point` to which the iterator is pointing is 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 illustrates the typical loop for walking through the points of a mesh.

```
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 `Mesh2.cxx`.

A `itk::Mesh` can contain a variety of cell types. Typical cells are the `itk::LineCell`, `itk::TriangleCell`, `itk::QuadrilateralCell`, `itk::TetrahedronCell`, and `itk::PolygonCell`. 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 must be included.

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

For consistency with 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 as pointers in the `CellsContainer`. 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 adds 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 whether the `CellAutoPointer` is responsible for releasing the cell memory when the time comes for its own destruction. It is said that a `CellAutoPointer` *owns* the cell when it is responsible for its destruction. At any given time many `CellAutoPointers` can point to the same cell, but only **one** `CellAutoPointer` can own the cell.

The `CellAutoPointer` trait is defined in the `MeshType` and can be extracted as follows.

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

Note that the `CellAutoPointer` points to a generic cell type. It is not aware of the actual type of the cell, which could be (for example) a `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 `LineType`. 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 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 `CellAutoPointer` 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.

```
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.

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

```
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 value returned is a pointer to the generic `CellType`. This pointer must be downcast 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 );
    if(line == ITK_NULLPTR)
    {
        continue;
    }
    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 `Mesh3.cxx`.

Just as 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. We add the `vnl_math::eps` value in order to avoid numerical errors when the point id is zero. The value of `vnl_math::eps` is the difference between 1.0 and the least value greater than 1.0 that is representable in this computer.

```
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] = std::log( static_cast<double>( id ) + itk::Math::eps ); // 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 = static_cast<PixelType>(0.0);
    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 to get the value associated with 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 `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 `itk::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 value 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 spacial coordinates.

InterpolationWeightType. The type used to represent interpolation weights.

CellPixelType. The value type associated with every cell.

Let's define types and values for each one of those elements. For example, the following code uses points in 3D space as nodes of the Mesh. The maximum dimension of the cells will be two, meaning that this is a 2D manifold better known 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-class 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 example, we simply store a `CellDataType` dummy variable named `value`.

```
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 identical to that already illustrated for accessing 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 `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 "itkLineCell.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 `GetBoundaryAssignment()`. 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 `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 "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 `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 "itkTriangleCell.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 itk::AutomaticTopologyMeshSource< MeshType > MeshSourceType;
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 `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 `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 [3]. 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;
    void Visit(unsigned long cellId, TriangleType * t )
    {
        std::cout << "Cell # " << cellId << " is a TriangleType ";
        std::cout << t->GetNumberOfPoints() << std::endl;
    }
    CustomTriangleVisitor() {}
    virtual ~CustomTriangleVisitor() {}
};
```

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 what 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 `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;
    }
    virtual ~CustomVertexVisitor() {}
};

```

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 ) {}
    virtual ~CustomLineVisitor() {}

    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;
        }
    }
    virtual ~CustomTriangleVisitor() {}
};
```

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;
    }
    virtual ~CustomTetrahedronVisitor() {}
};

```

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 `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 example the image and path are two-dimensional.

```
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 `TreeContainer.cxx`.

This example demonstrates use of the `itk::TreeContainer` class and associated TreeIterators. `TreeContainer` implements the notion of a tree, which is a branching data structure composed of nodes and edges, where the edges indicate a parent/child relationship between nodes. Each node may have exactly one parent, except for the root node, which has none. A tree must have exactly one root node, and a node may not be its own parent. To round out the vocabulary used to discuss this data structure, two nodes sharing the same parent node are called “siblings,” a childless node is termed a “leaf,” and a “forest” is a collection of disjoint trees. Note that in the present implementation, it is the user’s responsibility to enforce these relationships, as no checking is done to ensure a cycle-free tree. `TreeContainer` is templated over the type of node, affording the user great flexibility in using the structure for their particular problem.

Let’s begin by including the appropriate header files.

```
#include "itkTreeContainer.h"
#include "itkChildTreeIterator.h"
#include "itkLeafTreeIterator.h"
#include "itkLevelOrderTreeIterator.h"
#include "itkInOrderTreeIterator.h"
#include "itkPostOrderTreeIterator.h"
#include "itkRootTreeIterator.h"
#include "itkTreeIteratorClone.h"
```

We first instantiate a tree with `int` node type.

```
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);
```

Nodes may be added to the tree using the `Add()` method, where the first argument is the value of the new node, and the second argument is the value of the parent node.

```
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);
```

If two nodes have the same value, it is ambiguous which node is intended to be the parent of the new node; in this case, the first node with that value is selected. As will be demonstrated shortly, this ambiguity can be avoided by constructing the tree with `TreeIterators`.

Let's begin by defining a `itk::ChildTreeIterator`.

```
itk::ChildTreeIterator<TreeType> childIt(tree);
```

Before discussing the particular features of this iterator, however, we will illustrate features common to all `TreeIterators`, which inherit from `itk::TreeIteratorBase`. Basic use follows the convention of other iterators in ITK, relying on the `GoToBegin()` and `IsAtEnd()` methods. The iterator is advanced using the prefix increment `++` operator, whose behavior naturally depends on the particular iterator being used.

```
for (childIt.GoToBegin(); !childIt.IsAtEnd(); ++childIt)
{
    std::cout << childIt.Get() << std::endl;
}
std::cout << std::endl;
```

Note that, though not illustrated here, trees may also be traversed using the `GoToParent()` and `GoToChild()` methods.

`TreeIterators` have a number of useful functions for testing properties of the current node. For example, `GetType()` returns an enumerated type corresponding to the type of the particular iterator being used. These types are as follows:

UNDEFIND, PREORDER, INORDER, POSTORDER, LEVELORDER, CHILD, ROOT, and LEAF.

In the following snippet, we test whether the iterator is of type CHILD, and return from the program indicating failure if the test returns false.

```
if(childIt.GetType() != itk::TreeIteratorBase<TreeType>::CHILD)
{
    std::cerr << "Error: The iterator was not of type CHILD." << std::endl;
    return EXIT_FAILURE;
}
```

The value associated with the node can be retrieved and modified using `Get()` and `Set()` methods:

```
int oldValue = childIt.Get();
std::cout << "The node's value is " << oldValue << std::endl;
int newValue = 2;
childIt.Set(newValue);
std::cout << "Now, the node's value is " << childIt.Get() << std::endl;
```

A number of member functions are defined allowing the user to query information about the current node's parent/child relationships:

```
std::cout << "Is this a leaf node? " << childIt.IsLeaf() << std::endl;
std::cout << "Is this the root node? " << childIt.IsRoot() << std::endl;
std::cout << "Does this node have a parent? " << childIt.HasParent()
    << std::endl;
std::cout << "How many children does this node have? "
    << childIt.CountChildren() << std::endl;
std::cout << "Does this node have a child 1? " << childIt.HasChild(1)
    << std::endl;
```

In addition to traversing the tree and querying for information, `TreeIterators` can alter the structure of the tree itself. For example, a node can be added using the `Add()` methods, child nodes can be removed using the `RemoveChild()` method, and the current node can be removed using the `Remove()` method. Each of these methods returns a bool indicating whether the alteration was successful.

To illustrate this, in the following snippet we clear the tree of all nodes, and then repopulate it using the iterator.

```
tree->Clear();

itk::PreOrderTreeIterator<TreeType> it(tree);

it.GoToBegin();

it.Add(0);
it.Add(1);
it.Add(2);
it.Add(3);
it.GoToChild(2);
it.Add(4);
it.Add(5);
```

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.

```
itk::TreeIteratorBase<TreeType>* childItClone = childIt.Clone();
delete childItClone;
```

Alternatively, `itk::TreeIteratorClone` can be used to create a generic copy of an iterator.

```
typedef itk::TreeIteratorBase<TreeType> IteratorType;
typedef itk::TreeIteratorClone<IteratorType> IteratorCloneType;
IteratorCloneType anotherChildItClone = childIt;
```

We now turn our attention to features of the specific TreeIterator specializations. ChildTreeIterator, for example, provides a way to iterate through all the children of a node.

```
for (childIt.GoToBegin(); !childIt.IsAtEnd(); ++childIt)
{
    std::cout << childIt.Get();
}
std::cout << std::endl;
```

The `itk::LeafTreeIterator` iterates through the leaves of the tree.

```
itk::LeafTreeIterator<TreeType> leafIt(tree);
for (leafIt.GoToBegin(); !leafIt.IsAtEnd(); ++leafIt)
{
    std::cout << leafIt.Get() << std::endl;
}
std::cout << std::endl;
```

`itk::LevelOrderTreeIterator` takes three arguments in its constructor: the tree to be traversed, the maximum depth (or ‘level’), and the starting node. Naturally, this iterator provides a method for returning the current level.

```
itk::LevelOrderTreeIterator<TreeType> levelIt(tree,10,tree->GetNode(0));
for (levelIt.GoToBegin(); !levelIt.IsAtEnd(); ++levelIt)
{
    std::cout << levelIt.Get()
        << " ("<< levelIt.GetLevel() << ")"
        << std::endl;
}
std::cout << std::endl;
```

`itk::InOrderTreeIterator` iterates through the tree from left to right.

```
itk::InOrderTreeIterator<TreeType> inOrderIt(tree);
for (inOrderIt.GoToBegin(); !inOrderIt.IsAtEnd(); ++inOrderIt)
{
    std::cout << inOrderIt.Get() << std::endl;
}
std::cout << std::endl;
```

`itk::PreOrderTreeIterator` iterates through the tree from left to right but do a depth first search.

```
itk::PreOrderTreeIterator<TreeType> preOrderIt(tree);
for (preOrderIt.GoToBegin(); !preOrderIt.IsAtEnd(); ++preOrderIt)
{
    std::cout << preOrderIt.Get() << std::endl;
}
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);
for (postOrderIt.GoToBegin(); !postOrderIt.IsAtEnd(); ++postOrderIt)
{
    std::cout << postOrderIt.Get() << std::endl;
}
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(4));
for (rootIt.GoToBegin(); !rootIt.IsAtEnd(); ++rootIt)
{
    std::cout << rootIt.Get() << std::endl;
}
std::cout << std::endl;
```

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
`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 `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 `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");

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

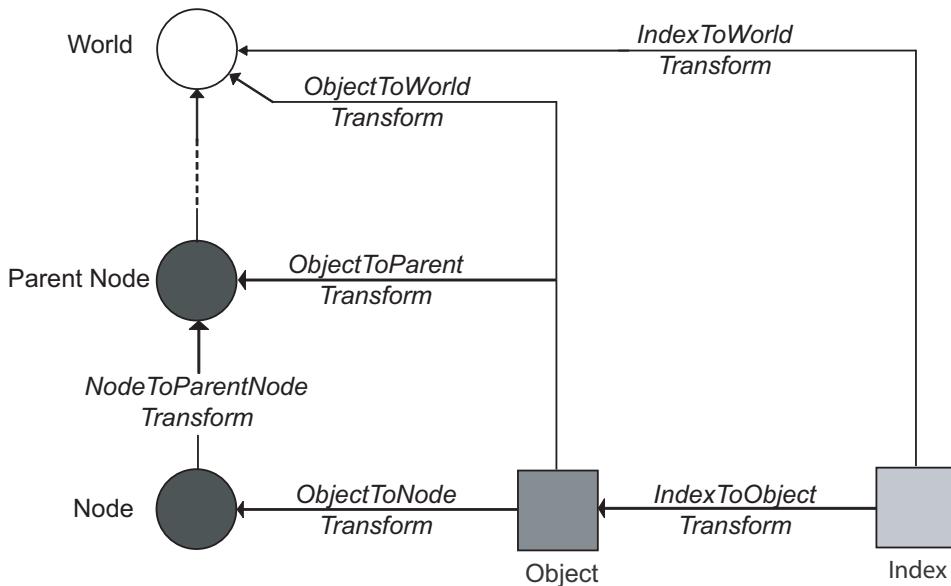


Figure 5.1: Set of transformations associated with a Spatial Object

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 `NodeToParentNodeTransform` is. Therefore, if one sets the `ObjectToParentTransform`, the `NodeToParentNodeTransform` is actually set.

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

$$X' = R \cdot (S \cdot X - C) + C + V \quad (5.1)$$

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:

$$M = R \cdot S \quad (5.2)$$

$$T = C + V - R \cdot C \quad (5.3)$$

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.

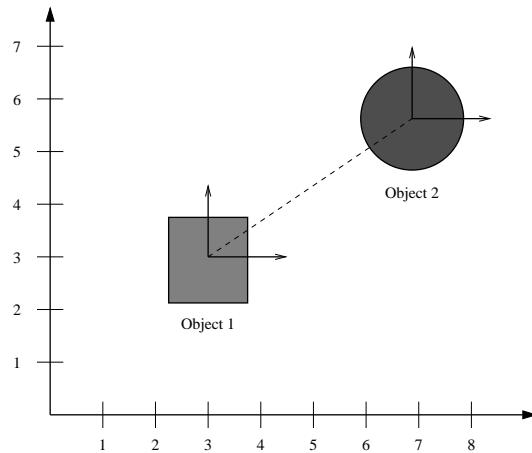


Figure 5.2: Physical positions of the two objects in the world frame (shapes are merely for illustration purposes).

```
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:

```
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 `ArrowSpatialObject.cxx`.

This example shows how to create an `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()` method. By default the length is set to 1.

```
myArrow->SetLength(2);
```

The direction of the arrow can be set using the `SetDirection()` method. Calling `SetDirection()` 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 `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 `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 `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();
Elliptype::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 `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 `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 `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;

for(it.GoToBegin(); !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(0);

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 `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::RGBOPixel`, 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(true); // initialize buffer to zero

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();
```

We then 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 `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);
    pSetColor(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.

```
size_t 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 `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"
```

`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

`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 write the mesh to disk.

Let's begin by including the appropriate header file.

```
#include "itkSpatialObjectToImageFilter.h"
#include "itkMeshSpatialObject.h"
#include "itkSpatialObjectReader.h"
#include "itkSpatialObjectWriter.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} };

MeshType::PointIdentifier 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, just like any other `SpatialObjects`.

```
myMeshSpatialObject->GetMesh();
```

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`. In order 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 `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"
```

`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("Surfacel");
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 `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"
```

`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 `GetNormal2()` 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 `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, and normals.

Let's start by including the appropriate header file.

```
#include "itkVesselTubeSpatialObject.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);
    pSetBranchness(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 `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 center-line 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"
```

`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;
    pSetColor(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 `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 `ITK_NULLPTR`). 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

`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 "itkSpatialObjectReader.h"  
#include "itkSpatialObjectWriter.h"  
#include "itkEllipseSpatialObject.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 a 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 `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();
ImageType::SizeValueType 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);
```

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();

```


Part III

Development Guidelines

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 6.2 describes the programming interface common to most ITK image iterators. Sections 6.3–6.4 document specific ITK iterator types and provide examples of how they are used.

6.1 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.

6.2 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.

6.2.1 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() );
```

6.2.2 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.

- **GoToBegin()** Points the iterator to the first valid data element in the region.

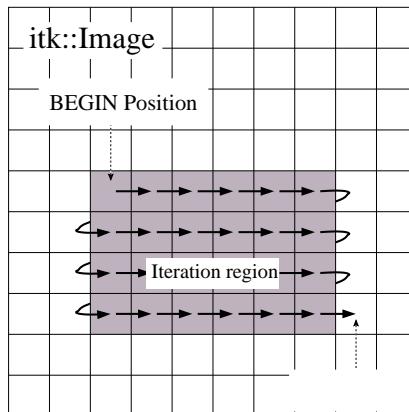


Figure 6.1: Normal path of an iterator through a 2D image. The iteration region is shown in a darker shade. An arrow denotes a single iterator step, the result of one `++` operation.

- **`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 6.1 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.

6.2.3 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 7 on page 181 for more information about image adaptors).

6.2.4 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() );
}

```

6.3 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.

6.3.1 `ImageRegionIterator`

The source code for this section can be found in the file `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 "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 );

inputIt.GoToBegin();
outputIt.GoToBegin();

while( !inputIt.IsAtEnd() )
{
    outputIt.Set( inputIt.Get() );
    ++inputIt;
    ++outputIt;
}
```

The while 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 6.2.

6.3.2 ImageRegionIteratorWithIndex

The source code for this section can be found in the file `ImageRegionIteratorWithIndex.cxx`.

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.

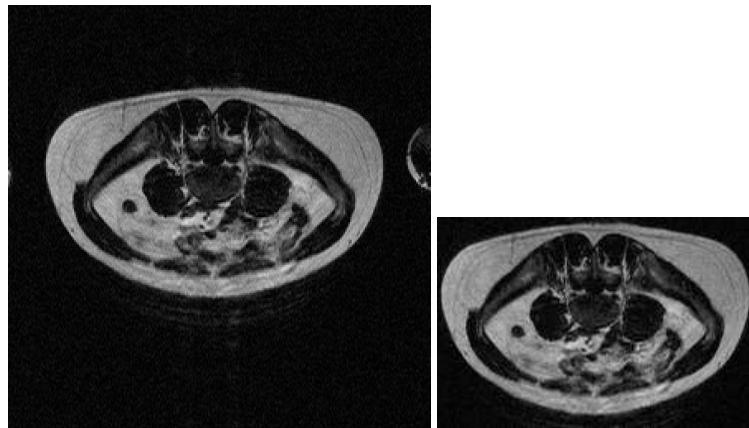


Figure 6.2: Cropping a region from an image. The original image is shown at left. The image on the right is the result of applying the `ImageRegionIterator` example code.

```
#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 `outputImage` of the same size, spacing, and origin as the `inputImage`.

```
ImageType::Pointer outputImage = ImageType::New();
outputImage->SetRegions( inputImage->GetRequestedRegion() );
outputImage->CopyInformation( inputImage );
outputImage->Allocate();
```

Next we create the iterator that walks the `outputImage`. This algorithm requires no iterator for the `inputImage`.

```
IteratorType outputIt( outputImage, outputImage->GetRequestedRegion() );
```

This axis flipping algorithm works by iterating through the `outputImage`, querying the iterator for

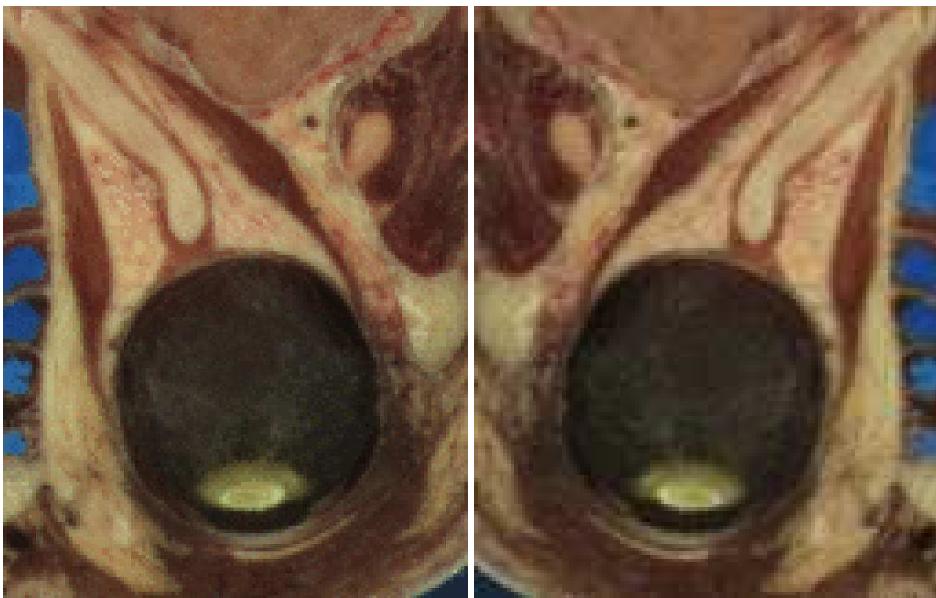


Figure 6.3: Results of using `ImageRegionIteratorWithIndex` to mirror an image across an axis. The original image is shown at left. The mirrored output is shown at right.

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 6.3 shows how the original image has been mirrored across its *x*-axis in the output.

6.3.3 `ImageLinearIteratorWithIndex`

The source code for this section can be found in the file `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 .

- **`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()`** Moves the iterator to *one past* the last valid pixel of the current line.
- **`GoToReverseBeginOfLine()`** Moves the iterator to *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 "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() );

.SetDirection(0);
.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();
    while ( ! inputIt.IsAtEndOfLine() )
    {
        --outputIt;
        outputIt.Set( inputIt.Get() );
        ++inputIt;
    }
}
```

Running this example on `VisibleWomanEyeSlice.png` produces the same output image shown in Figure 6.3.

The source code for this section can be found in the file `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, the 3D and 4D readers.

```

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;

```

Next, define the necessary types for indices, points, spacings, and size.

```

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::SpacingType Spacing4DType;
typedef Image4DType::PointType   Origin4DType;

```

Here we make sure that the values for our resultant 3D mean image match up with the input 4D image.

```

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();

```

Next we iterate over time in the input image series, compute the average, and store that value in the corresponding pixel of the output 3D image.

```

IteratorType it( image4D, region4D );
it.SetDirection( 3 ); // Walk along time dimension
it.GoToBegin();
while( !it.IsAtEnd() )
{
    SumType sum = itk::NumericTraits< SumType >::ZeroValue();
    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 every pixel, but do not enforce any particular order for the visits. The linear iterator guarantees it will visit the pixels along a line of the image in the order in which they are placed in the line, but does not state 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.

6.3.4 ImageSliceIteratorWithIndex

The source code for this section can be found in the file `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 6.3.3).

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( std::max( outputIt.Get(), inputIt.Get() ) );
            ++inputIt;
            ++outputIt;
        }
        outputIt.NextLine();
        inputIt.NextLine();
    }
    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 6.4.

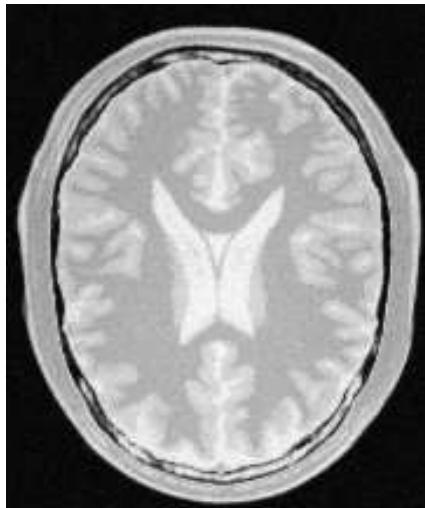


Figure 6.4: The maximum intensity projection through three slices of a volume.

6.3.5 ImageRandomConstIteratorWithIndex

The source code for this section can be found in the file `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"
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

	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 6.1: Estimates of mean image pixel value using the `ImageRandomConstIteratorWithIndex` at different sample sizes.

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] );
```

The following table shows the results of running this example on several of the data files from Examples/Data with a range of sample sizes.

6.4 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.

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

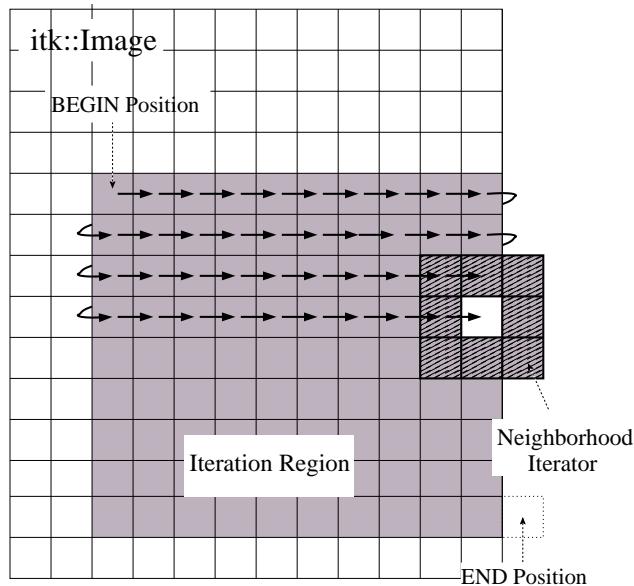


Figure 6.5: Path of a 3×3 neighborhood iterator through a 2D image region. The extent of the neighborhood is indicated by the hashing around the iterator position. Pixels that lie within this extent are accessible through the iterator. An arrow denotes a single iterator step, the result of one `++` operation.

all neighborhood pixels and information such as the size, extent, and location of the neighborhood.

Neighborhood iterators use the same operators defined in Section 6.2 and the same code constructs as normal iterators for looping through an image. Figure 6.5 shows a neighborhood iterator moving through an iteration region. This iterator defines a 3×3 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 6.2), neighborhood 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 6.6 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

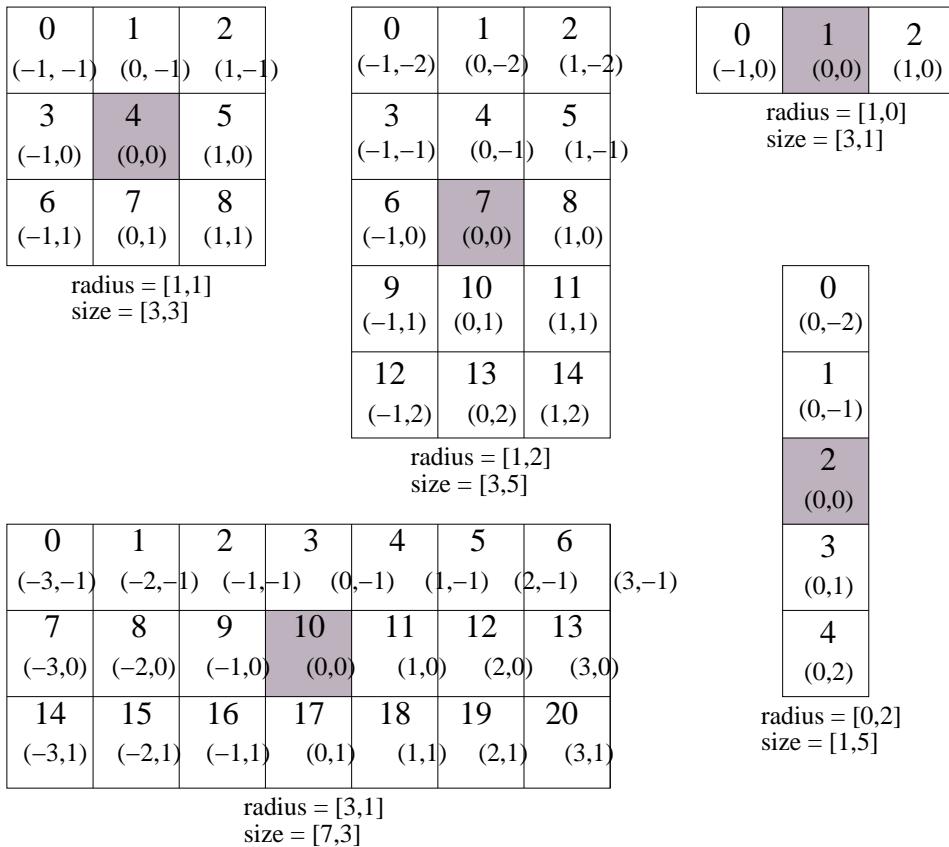


Figure 6.6: Several possible 2D neighborhood iterator shapes are shown along with their radii and sizes. A neighborhood pixel can be dereferenced by its integer index (top) or its offset from the center (bottom). The center pixel of each iterator is shaded.

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 6.6, 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 $3 \times 3 \times 3$ 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 6.6, 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 6.5, 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 6.4.1.

- **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.

6.4.1 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 `NeighborhoodIterators1.cxx`.

This example uses the `itk::NeighborhoodIterator` to implement a simple Sobel edge detection algorithm [4]. 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::cerr << "ExceptionObject caught !" << std::endl;
    std::cerr << err << std::endl;
    return EXIT_FAILURE;
}
```

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 6.4.1 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;

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::cerr << "ExceptionObject caught !" << std::endl;
    std::cerr << err << std::endl;
    return EXIT_FAILURE;
}
```

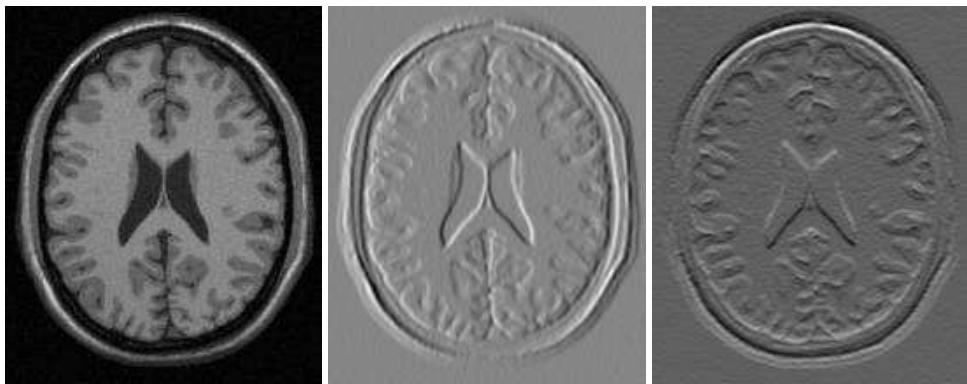


Figure 6.7: Applying the Sobel operator in different orientations to an MRI image (left) produces x (center) and y (right) derivative images.

The center image of Figure 6.7 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 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 [4] [2]. 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 6.7. 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 `NeighborhoodIterators3.cxx`.

This example illustrates a technique for improving the efficiency of neighborhood calculations by eliminating unnecessary bounds checking. As described in Section 6.4, 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 `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(),
                              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 6.8 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 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.

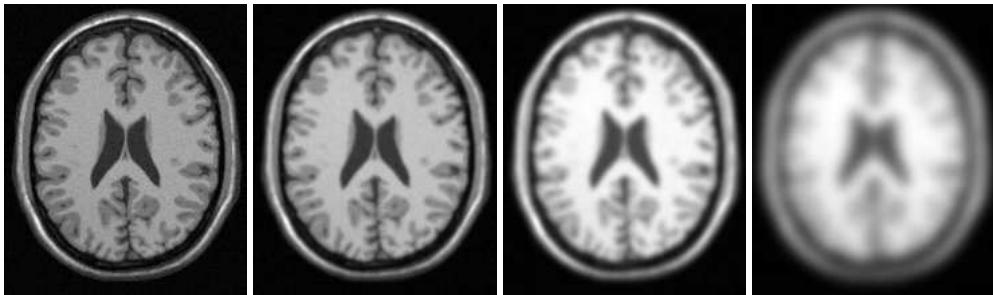


Figure 6.8: Results of convolution filtering with a Gaussian kernel of increasing standard deviation σ (from left to right, $\sigma = 0$, $\sigma = 1$, $\sigma = 2$, $\sigma = 5$). Increased blurring reduces contrast and changes the average intensity value of the image, which causes the image to appear brighter when rescaled.

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 6.6). 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 6.4.1 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 itera-

tion. 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 `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 6.9 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.



Figure 6.9: 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.

6.4.2 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 [4], [2], 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 `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 6.4.1 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 6.4.1 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 6.4.1 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 = std::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);
    }
}

```



Figure 6.10: 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.

The source code for this section can be found in the file
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 6.10 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.

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 6).

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.

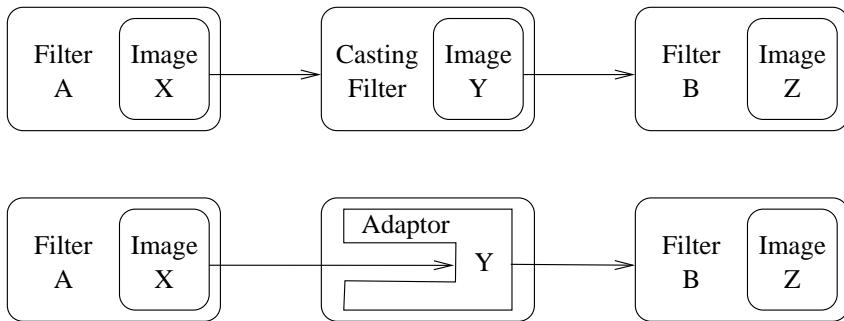


Figure 7.1: The difference between using a `CastImageFilter` and an `ImageAdaptor`. `ImageAdaptors` convert pixel values when they are accessed by iterators. Thus, they do not produce an intermediate image. In the example illustrated by this figure, the `Image Y` is not created by the `ImageAdaptor`; instead, the image is simulated on the fly each time an iterator from the filter downstream attempts to access the image data.

7.1 Image Casting

The source code for this section can be found in the file `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 "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.

7.2 Adapting RGB Images

The source code for this section can be found in the file `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 EXIT_FAILURE;
}
```

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.

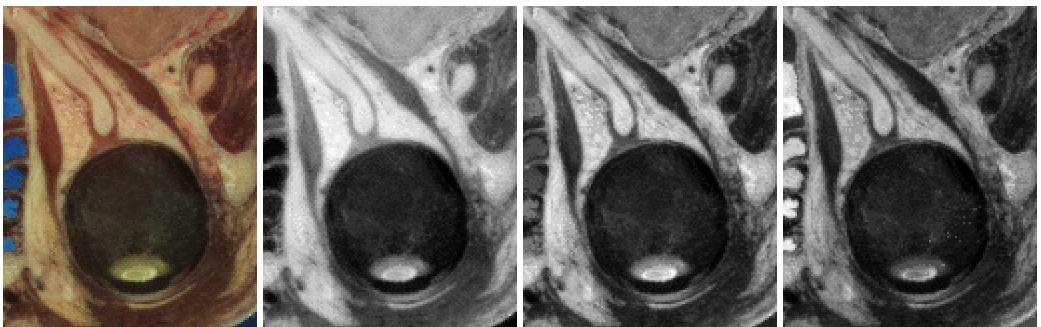


Figure 7.2: Using ImageAdaptor to extract the components of an RGB image. The image on the left is a subregion of the Visible Woman cryogenic data set. The red, green and blue components are shown from left to right as scalar images extracted with an ImageAdaptor.

```
class GreenChannelPixelAccessor
{
public:
    typedef itk::RGBPixel<float> InternalType;
    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;
    float ExternalType;

    static ExternalType Get( const InternalType & input )
    {
        return static_cast<ExternalType>( input.GetBlue() );
    }
};
```

Figure 7.2 shows the result of extracting the red, green and blue components from a region of the Visible Woman cryogenic data set.

7.3 Adapting Vector Images

The source code for this section can be found in the file `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 "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;

    VectorPixelAccessor() : m_Index(0) {}

    VectorPixelAccessor & operator=( const VectorPixelAccessor & vpa )
    {
        m_Index = vpa.m_Index;
        return *this;
    }
    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,
                           itk::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.

```
itk::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.

```
adaptor->SetImage( gradient->GetOutput() );
```

As in the previous example, we rescale the scalar image before writing the image out to file. Figure 7.3 shows the result of applying the example code for extracting both components of a two dimensional gradient.

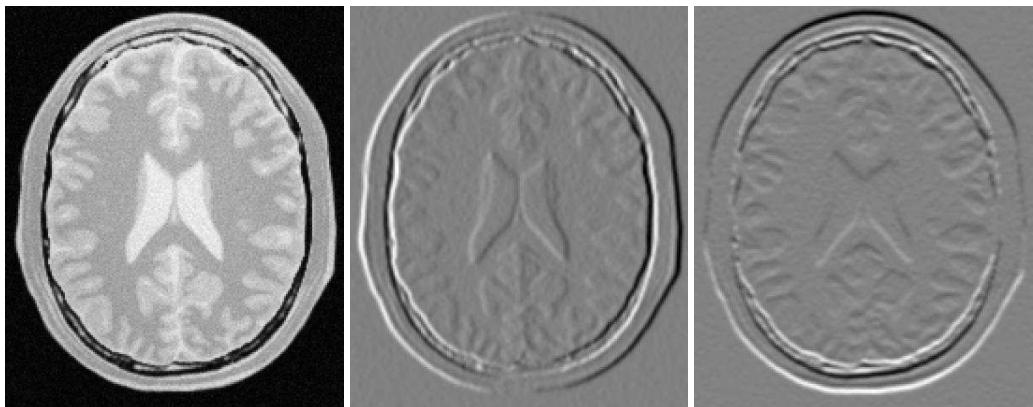


Figure 7.3: Using `ImageAdaptor` to access components of a vector image. The input image on the left was passed through a gradient image filter and the two components of the resulting vector image were extracted using an image adaptor.

7.4 Adaptors for Simple Computation

The source code for this section can be found in the file `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;

    ThresholdingPixelAccessor() : m_Threshold(0) {}

    ExternalType Get( const InternalType & input ) const
    {
        return (input > m_Threshold) ? 1 : 0;
    }
    void SetThreshold( const InternalType threshold )
    {
        m_Threshold = threshold;
    }

    ThresholdingPixelAccessor &
    operator=( const ThresholdingPixelAccessor & vpa )
    {
        m_Threshold = vpa.m_Threshold;
        return *this;
    }

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 itk::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,
                           itk::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.

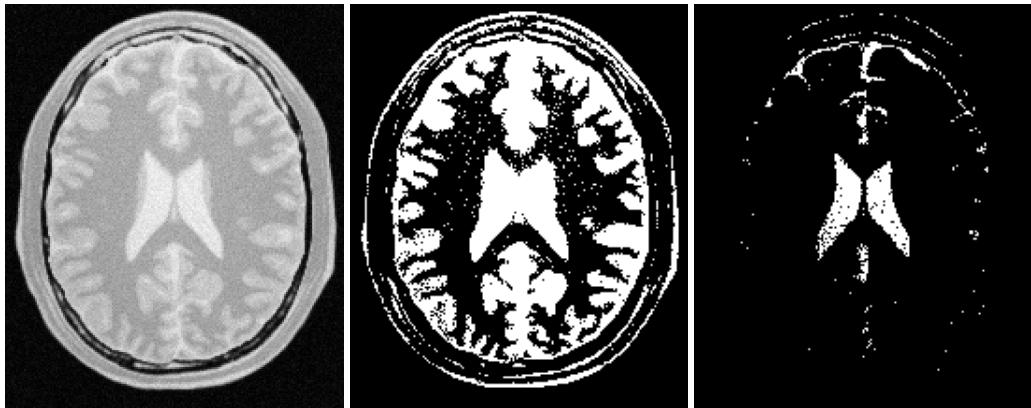


Figure 7.4: Using ImageAdaptor to perform a simple image computation. An ImageAdaptor is used to perform binary thresholding on the input image on the left. The center image was created using a threshold of 180, while the image on the right corresponds to a threshold of 220.

```
itk::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();
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 7.4 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.

7.5 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.

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.

8.1 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.

8.2 Overview of Filter Creation

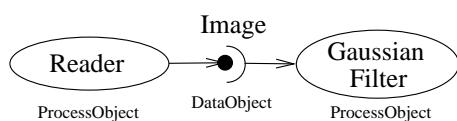


Figure 8.1: Relationship between DataObject and ProcessObject.

single image on output. The superclass `itk::ImageToImageFilter` is a convenience class that provides 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.

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

- `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.

8.3 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 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

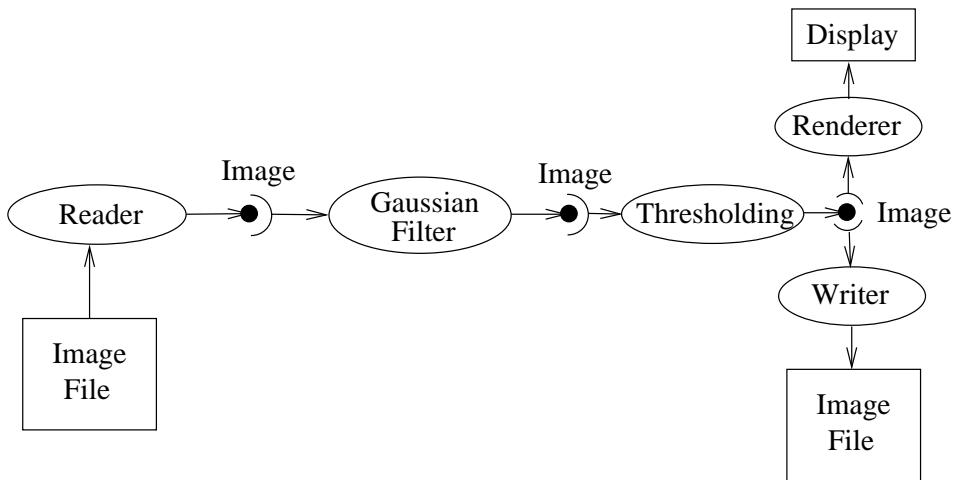


Figure 8.2: The Data Pipeline

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 8.1 and 8.2.

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.

8.3.1 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.
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

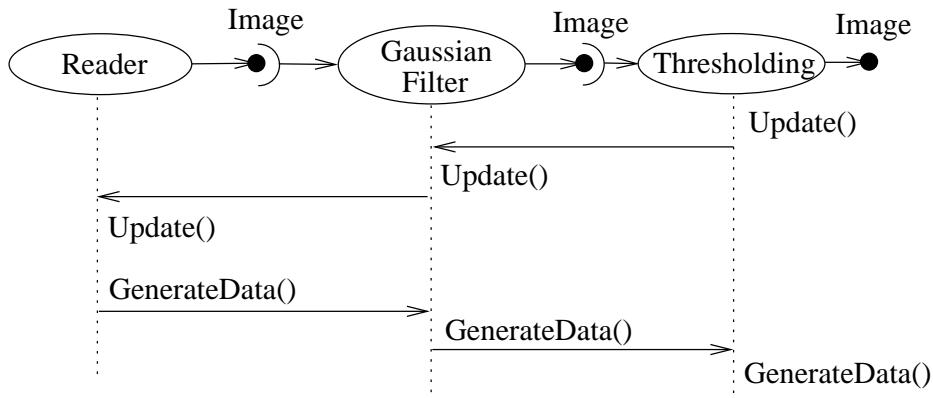


Figure 8.3: Sequence of the Data Pipeline updating mechanism

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.

8.3.2 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 `RequestedRegion`. 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.

8.4 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:

```
virtual void ThreadedGenerateData( const OutputImageRegionType&
outputRegionForThread, ThreadIdType threadId ) ITK_OVERRIDE;
```

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.

8.5 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.

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).

Use the macros `ITK_OVERRIDE`, `ITK_NULLPTR`, and `ITK_NOEXCEPT`. These expand to the C++11 keywords `override`, `nullptr`, and `noexcept`, respectively, when built with a compiler using the C++11 standard or newer.

Finally, the template implementation code (in the `.hxx` file) should be included, bracketed by a test for manual instantiation, thus:

```
#ifndef ITK_MANUAL_INSTANTIATION
#include "itkExampleFilter.hxx"
#endif
```

8.5.1 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.

8.5.2 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.

8.6 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

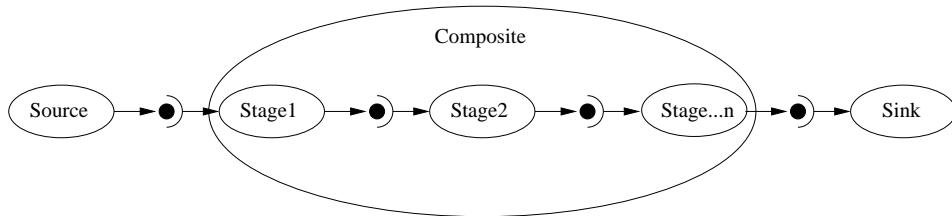


Figure 8.4: A Composite filter encapsulates a number of other filters.

two or more existing filters, which can then be used as a building block in a complex pipeline. This approach follows the Composite pattern [3], 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 8.4, 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.

8.6.1 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`.
4. The default constructor should build the pipeline by creating the stages and connect them together, along with any default parameter settings, as appropriate.

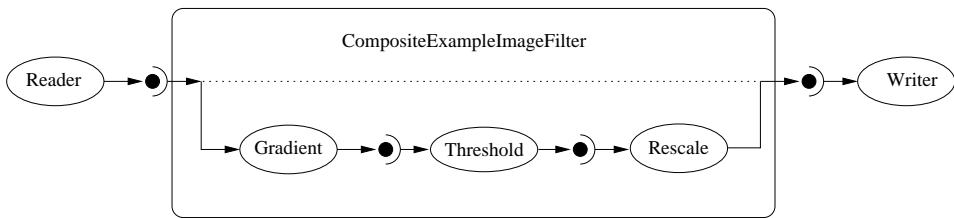


Figure 8.5: Example of a typical composite filter. Note that the output of the last filter in the internal pipeline must be grafted into the output of the composite filter.

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 8.5.

8.6.2 A Simple Example

The source code for this section can be found in the file `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`:

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, so that 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 < typename TImage >
class CompositeExampleImageFilter :
    public ImageToImageFilter< TImage, TImage >
{
public:
```

Next we have the standard declarations, used for object creation with the object factory:

```
typedef CompositeExampleImageFilter           Self;
typedef ImageToImageFilter< TImage, TImage > 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 TImage           ImageType;
typedef typename ImageType::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< ImageType >           ThresholdType;
typedef GradientMagnitudeImageFilter< ImageType, ImageType > GradientType;
typedef RescaleIntensityImageFilter< ImageType, ImageType > 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;
};

} // end namespace itk
```

The constructor sets up the pipeline, which involves creating the stages, connecting them together, and setting default parameters.

```
template< typename TImage >
CompositeExampleImageFilter< TImage >
::CompositeExampleImageFilter()
{
    m_Threshold = 1;
    m_GradientFilter = GradientType::New();
    m_ThresholdFilter = ThresholdType::New();
    m_ThresholdFilter->SetInput( m_GradientFilter->GetOutput() );
    m_RescaleFilter = RescalerType::New();
    m_RescaleFilter->SetInput( m_ThresholdFilter->GetOutput() );
    m_RescaleFilter->SetOutputMinimum(
        NumericTraits<PixelType>::NonpositiveMin());
    m_RescaleFilter->SetOutputMaximum(NumericTraits<PixelType>::max());
}
```

The `GenerateData()` is where the composite magic happens.

First, connect the first component filter to the inputs of the composite filter (the actual input, supplied by the upstream stage). At a filter's `GenerateData()` stage, the input image's information and pixel buffer content have been updated by the pipeline. To prevent the mini-pipeline update from propagating upstream, the input image is disconnected from the pipeline by grafting its contents to a new `itk::Image` pointer.

This implies that the composite filter must implement pipeline methods that indicate the `itk::ImageRegion`'s it requires and generates, like `GenerateInputRequestedRegion()`, `GenerateOutputRequestedRegion()`, `GenerateOutputInformation()` and `EnlargeOutputRequestedRegion()`, according to the behavior of its component filters.

Next, graft the output of the last stage onto the output of the composite, which ensures the requested region is updated and the last stage populates the output buffer allocated by the composite filter. 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< typename TImage >
void
CompositeExampleImageFilter< TImage >
::GenerateData()
{
    typename ImageType::Pointer input = ImageType::New();
    input->Graft( const_cast< ImageType * >( this->GetInput() ) );
    m_GradientFilter->SetInput( input );

    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< typename TImage >
void
CompositeExampleImageFilter< TImage >
::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 8.5.

HOW TO CREATE A MODULE

The Insight Toolkit is organized into logical units of coherent functionality called modules. These modules are self-contained in a directory, whose components are organized into subdirectories with standardized names. A module usually has dependencies on other modules, which it declares. A module is defined with CMake scripts that inform the build system of its contents and dependencies.

Modules are organized into:

- The **top level** directory.
- The **include** directory.
- The **src** directory.
- The **test** directory.
- The **wrapping** directory.

This chapter describes how to create a new module. The following sections are organized by the different directory components of the module. The chapter concludes with a section on how to add a third-party library dependency to a module.

9.1 Name and dependencies

The top level directory of a module is used to define a module's name and its dependencies. Two files are required:

1. CMakeLists.txt
2. itk-module.cmake

9.1.1 CMakeLists.txt

When CMake starts processing a module, it begins with the top level CMakeLists.txt file. At a minimum, the CMakeLists.txt should contain

```
cmake_minimum_required(VERSION 2.9)
project(MyModule)

set(MyModule_LIBRARIES MyModule)

if(NOT ITK_SOURCE_DIR)
  find_package(ITK REQUIRED)
  list(APPEND CMAKE_MODULE_PATH ${ITK_CMAKE_DIR})
  include(ITKModuleExternal)
else()
  itk_module_impl()
endif()
```

where MyModule is the name of the module.

The CMake variable <module-name>_LIBRARIES should be set to the names of the libraries, if any, that clients of the module need to link. This will be the same name as the library generated with the add_library command in a module's src directory, described in further detail in the Libraries Section 9.3.

The path if(NOT ITK_SOURCE_DIR) is used when developing a module outside of the ITK source tree, i.e. an External module. An External module can be made available to the community by adding it to Modules/Remote/*.remote.cmake Remote module index in the ITK repository per Section 10.1.

The CMake macro itk_module_impl is defined in the file CMake/ITKModuleMacros.cmake. It will initiate processing of the remainder of a module's CMake scripts. The script ITKModuleExternal calls itk_module_impl internally.

9.1.2 itk-module.cmake

The itk-module.cmake is also a required CMake script at the top level of a module, but this file is used to declare

1. The module name.
2. Dependencies on other modules.
3. Modules properties.
4. A description of the module.

In this file, first set a CMake variable with the module's description followed by a call to the `itk_module` macro, which is already defined by the time the script is read. For example, `itk-module.cmake` for the `ITKCommon` module is

```
set(DOCUMENTATION "This module contains the central classes of the ITK
toolkit. They include, basic data structures \such as Points, Vectors,
Images, Regions\ the core of the process objects \such as base
classes for image filters\ the pipeline infrastructure classes, the support
for multi-threading, and a collection of classes that isolate ITK from
platform specific features. It is anticipated that most other ITK modules will
depend on this one.")

itk_module(ITKCommon
  ENABLE_SHARED
  PRIVATE_DEPENDS
    ITKDoubleConversion
  COMPILE_DEPENDS
    ITKKWSys
    ITKVNLInstantiation
  TEST_DEPENDS
    ITKTestKernel
    ITKMesh
    ITKImageIntensity
    ITKIOImageBase
  DESCRIPTION
    "${DOCUMENTATION}"
)
```

The description for the module should be escaped as a CMake string, and it should be formatted with Doxygen markup. This description is added to ITK's generated Doxygen documentation when the module is added to the Remote module index. The description should describe the purpose and content of the module and reference an Insight Journal article for further information.

A module name is the only required positional argument to the `itk_module` macro. Named options that take one or argument are:

DEPENDS Modules that will be publicly linked to this module. The header's used are added to `include/*.{h,hxx}` files.

PRIVATE_DEPENDS Modules that will be privately linked to this module. The header's used are only added to `src/*.cxx` files.

COMPILE_DEPENDS Modules that are needed at compile time by this module. The header's used are added to `include/*{h,hxx}` files but there is not a library to link against.

TEST_DEPENDS Modules that are needed by this modules testing executables. The header's used are added to `test/*.cxx` files.

DESCRIPTION Free text description of the module.

Public dependencies are added to the module's `INTERFACE_LINK_LIBRARIES`, which is a list of transitive link dependencies. When this module is linked to by another target, the libraries listed (and

recursively, their link interface libraries) will be provided to the target also. Private dependencies are linked to by this module, but not added to INTERFACE_LINK_LIBRARIES.

Compile Dependencies are added to CMake's list of dependencies for the current module, ensuring that they are built before the current module, but they will not be linked either publicly or privately. They are only used to support the building of the current module.

The following additional options take no arguments:

EXCLUDE_FROM_DEFAULT Exclude this module from collection of modules enabled with the `ITK_BUILD_DEFAULT_MODULES` CMake option.

ENABLE_SHARED Build this module as a shared library if the `BUILD_SHARED_LIBS` CMake option is set.

All External and Remote modules should set the `EXCLUDE_FROM_DEFAULT` option.

9.2 Headers

Headers for the module, both `*.h` declaration headers and `*.hxx` template definition headers, should be added to the `include` directory. No other explicit CMake configuration is required.

This path will automatically be added to the build `include` directory paths for libraries (9.3) and tests (9.4) in the module and when another module declares this module as a dependency.

When a module is installed, headers are installed into a single directory common to all ITK header files.

When `BUILD_TESTING` is enabled, a header test is automatically created. This test simply builds a simple executable that `#includes` all header files in the `include` directory. This ensures that all included headers can be found, which tests the module's dependency specification per Section 9.1.

9.3 Libraries

Libraries generated by a module are created from source files with the `.cxx` extension in a module's `src` directory. Some modules are header-only, and they will not generate any libraries; in this case, the `src` directory is omitted. When present, the `src` directory should contain a `CMakeLists.txt` file that describes how to build the library. A minimal `CMakeLists.txt` file is as follows.

```
set(AModuleName_SRCS
    itkFooClass.cxx
    itkBarClass.cxx
)
add_library(AModuleName ${AModuleName_SRCS})
itk_module_link_dependencies()
itk_module_target(AModuleName)
```

The `itk_module_link_dependencies` macro will link the library to the libraries defined by the module dependency specification per Section 9.1. The `itk_module_target` macro will set CMake target properties associated with the current module to the given target.

If the `ENABLE_SHARED` option is set on a module, a shared library will be generated when the CMake option `BUILD_SHARED_LIBS` is enabled. A library symbol export specification header is also generated for the module. For a module with the name `AModuleName`, the generated header will have the name `AModuleNameExport.h`. Include the export header in the module source headers, and add the export specification macro to the contained classes. The macro name in this case would be called `AModuleName_EXPORT`. For example, the file `itkFooClass.h` would contain

```
#include "AModuleNameExport.h"

namespace itk
{
class AModuleName_EXPORT FooClass
{
...
}
```

9.4 Tests

Regression tests for a module are placed in the `test` directory. This directory will contain a `CMakeLists.txt` with the CMake configuration, test sources, and optional `Input` and `Baseline` directories, which contain test input and baseline image datasets, respectively.

An example CMake configuration for a test directory is shown below.

```
itk_module_test()

set(ModuleTemplateTests
    itkMinimalStandardRandomVariateGeneratorTest.cxx
    itkLogNormalDistributionImageSourceTest.cxx
)

CreateTestDriver(ModuleTemplate "${ModuleTemplate-Test_LIBRARIES}" "${ModuleTemplateTests}")

itk_add_test(NAME itkMinimalStandardRandomVariateGeneratorTest
    COMMAND ModuleTemplateTestDriver itkMinimalStandardRandomVariateGeneratorTest
)

itk_add_test(NAME itkLogNormalDistributionImageSourceTest
    COMMAND ModuleTemplateTestDriver --without-threads
    --compare
    ${ITK_TEST_OUTPUT_DIR}/itkLogNormalDistributionImageSourceTestOutput.mha
    DATA{Baseline/itkLogNormalDistributionImageSourceTestOutput.mha}
    itkLogNormalDistributionImageSourceTest
    ${ITK_TEST_OUTPUT_DIR}/itkLogNormalDistributionImageSourceTestOutput.mha
)
```

The CMakeLists.txt file should start with a call to the `itk_module_test` macro. Next, the test sources are listed. The naming convention for unit test files is `itk<ClassName>Test.cxx`. Each test file should be written like a command line executable, but the name of the `main` function should be replaced with the name of the test. The function should accept `int argc, char * argv[]` as arguments. To reduce the time required for linking and to provide baseline comparison functionality, all tests are linked to into a single test driver executable. To generate the executable, call the `CreateTestDriver` macro.

Tests are defined with the `itk_add_test` macro. This is a wrapper around the CMake `add_test` command that will resolve content links in the `DATA` macro. Testing data paths are given inside the `DATA` macro. Content link files, stored in the source code directory, are replaced by actual content files in the build directory when CMake downloads the target at build time. A content link file has the same name as its target, but a `.md5` extension is added, and the `.md5` file's contents are only the MD5SUM hash of its target. Content links for data files in a Git distributed version control repository prevent repository bloat. To obtain content links, register an account with the ITK community at <https://midas3.kitware.com> and request upload permissions on the ITK mailing list.

Test commands should call the test driver executable, followed by options for the test, followed by the test function name, followed by arguments that are passed to the test. The test driver accepts options like `--compare` to compare output images to baselines or options that modify tolerances on comparisons.

9.5 Wrapping

Wrapping for programming languages like Python can be added to a module through a simple configuration in the module's wrapping directory. While wrapping is almost entirely automatic, configuration is necessary to add two pieces of information,

1. The types with which to instantiate templated classes.
2. Class dependencies which must be wrapped before a given class.

When wrapping a class, dependencies, like the base class and other types used in the wrapped class's interface, should also be wrapped. The wrapping system will emit a warning when a base class or other required type is not already wrapped to ensure proper wrapping coverage. Since module dependencies are wrapped by the build system before the current module, class wrapping build order is already correct module-wise. However, it may be required to wrap classes within a module in a specific order; this order can be specified in the `wrapping/CMakeLists.txt` file.

Many ITK classes are templated, which allows an algorithm to be written once yet compiled into optimized binary code for numerous pixel types and spatial dimensions. When wrapping these templated classes, the template instantiations to wrap must be chosen at build time. The template that should be used are configured in a module's `*.wrap` files. Wrapping is configured by calling CMake macros defined in the `ITK/Wrapping/TypedefMacros.cmake` file.

9.5.1 CMakeLists.txt

The `wrapping/CMakeLists.txt` file calls three macros, and optionally set a variable, `WRAPPER_SUBMODULE_ORDER`. The following example is from the `ITKImageFilterBase` module:

```
itk_wrap_module(ITKImageFilterBase)

set(WRAPPER_SUBMODULE_ORDER
    itkRecursiveSeparableImageFilter
    itkFlatStructuringElement
    itkKernelImageFilter
    itkMovingHistogramImageFilterBase
)
itk_auto_load_submodules()
itk_end_wrap_module()
```

The `itk_wrap_module` macro takes the current module name as an argument. In some cases, classes defined in the `*.wrap` files within a module may depend each other. The `WRAPPER_SUBMODULE_ORDER` variable is used to declare which submodules should be wrapped first and the order they should be wrapped.

CMake variable	Wrapping shorthand value
ITK_WRAP_IMAGE_DIMS	List of unsigned integers
ITK_WRAP_VECTOR_COMPONENTS	List of unsigned integers
ITK_WRAP_double	D
ITK_WRAP_float	F
ITK_WRAP_complex_double	CD
ITK_WRAP_complex_float	CF
ITK_WRAP_vector_double	VD
ITK_WRAP_vector_float	VF
ITK_WRAP_covariate_vector_double	CVD
ITK_WRAP_covariate_vector_float	CVF
ITK_WRAP_signed_char	SC
ITK_WRAP_signed_short	SS
ITK_WRAP_signed_long	SL
ITK_WRAP_unsigned_char	UC
ITK_WRAP_unsigned_short	US
ITK_WRAP_unsigned_long	UL
ITK_WRAP_rgb_unsigned_char	RGBUC
ITK_WRAP_rgb_unsigned_short	RGBUS
ITK_WRAP_rgba_unsigned_char	RGBAUC
ITK_WRAP_rgba_unsigned_short	RGBASU

Table 9.1: CMake wrapping type configuration variables and their shorthand value in the wrapping configuration.

9.5.2 Class wrap files

Wrapping specification for classes is written in the module's `*.wrap` CMake script files. These files call wrapping CMake macros, and they specify which classes to wrap, whether smart pointer's should be wrapped for the class, and which template instantiations to wrap for a class.

Overall toolkit class template instantiations are parameterized by the CMake build configuration variables shown in Table 9.1. The wrapping configuration refers to these settings with the shorthand values listed in the second column.

Class wrap files call sets of wrapping macros for the class to be wrapped. The macros are often called in loops over the wrapping variables to instantiate the desired types. The following example demonstrates wrapping the `itk::ImportImageFilter` class, taken from the `ITK/Modules/Core/Common/wrapping/itkImportImageFilter.wrap` file.

```
itk_wrap_class("itk::ImportImageFilter" POINTER)

foreach(d ${ITK_WRAP_IMAGE_DIMS})
    foreach(t ${WRAP_ITK_SCALAR})
        itk_wrap_template("${ITKM_${t}}${d}" "${ITKT_${t}},${d}")
    endforeach()
endforeach()

itk_end_wrap_class()
```

Wrapping Variables

Instantiations for classes are determined by looping over CMake lists that collect sets of shorthand wrapping values, namely,

- ITK_WRAP_IMAGE_DIMS
- ITK_WRAP_IMAGE_DIMS_INCREMENTED
- ITK_WRAP_IMAGE_VECTOR_COMPONENTS
- ITK_WRAP_IMAGE_VECTOR_COMPONENTS_INCREMENTED
- WRAP_ITK_USIGN_INT
- WRAP_ITK_SIGN_INT
- WRAP_ITK_INT
- WRAP_ITK_REAL
- WRAP_ITK_COMPLEX_REAL
- WRAP_ITK_SCALAR
- WRAP_ITK_VECTOR_REAL
- WRAP_ITK_COV_VECTOR_REAL
- WRAP_ITK_VECTOR
- WRAP_ITK_RGB

- WRAP_ITK_RGBA
- WRAP_ITK_COLOR
- WRAP_ITK_ALL_TYPES

Templated classes are wrapped as typedefs for particular instantiations. The typedefs are named with a name mangling scheme for the template parameter types. The mangling of common types are stored in CMake variables listed in Table 9.2, Table 9.3, and Table 9.4. Mangling variables start with the prefix `ITKM_` and their corresponding C++ type variables start with the prefix `ITKT_`.

Wrapping Macros

There are a number of wrapping macros called in the `wrapping/*.wrap` files. Macros are specialized for classes that use `itk::SmartPointers` and templated classes.

For non-templated classes, the `itk_wrap_simple_class` is used. This macro takes fully qualified name of the class as an argument. Lastly, the macro takes an optional argument that can have the values `POINTER`, `POINTER_WITH_CONST_POINTER`, or `POINTER_WITH_SUPERCLASS`. If this argument is passed, then the typedefs `classname::Pointer`, `classname::ConstPointer`, or `classname::Superclass::Pointer` are wrapped. Thus, the wrapping configuration for `itk::Object` is

```
itk_wrap_simple_class("itk::Object" POINTER)
```

When wrapping templated classes, three or more macro calls are required. First, `itk_wrap_class` is called. Again, its arguments are the fully qualified followed by an option argument that can have the value `POINTER`, `POINTER_WITH_CONST_POINTER`, `POINTER_WITH_SUPERCLASS`, `POINTER_WITH_2_SUPERCLASSES`, `EXPLICIT_SPECIALIZATION`, `POINTER_WITH_EXPLICIT_SPECIALIZATION`, `ENUM`, or `AUTOPOINTER`. Next, a series of calls are made to macros that declare which templates to instantiate. Finally, the `itk_end_wrap_class` macro is called, which has no arguments.

The most general template wrapping macro is `itk_wrap_template`. Two arguments are required. The first argument is a mangled suffix to be added to the class name, which uniquely identifies the instantiation. This argument is usually specified at least partially with `ITKM_` mangling variables. The second argument is the template instantiation in C++ form. This argument is usually specified at least partially with `ITKT_` C++ type variables. For example, wrapping for `itk::ImageSpatialObject`, which templated a dimension and pixel type, is configured as

	CMake Variable	Value
Mangling	ITKM_B	B
C++ Type	ITKT_B	bool
Mangling	ITKM_UC	UC
C++ Type	ITKT_UC	unsigned char
Mangling	ITKM_US	US
C++ Type	ITKT_US	unsigned short
Mangling	ITKM_UI	UI
C++ Type	ITKT_UI	unsigned integer
Mangling	ITKM_UL	UL
C++ Type	ITKT_UL	unsigned long
Mangling	ITKM_SC	SC
C++ Type	ITKT_SC	signed char
Mangling	ITKM_SS	SS
C++ Type	ITKT_SS	signed short
Mangling	ITKM_SI	SI
C++ Type	ITKT_SI	signed integer
Mangling	ITKM_UL	UL
C++ Type	ITKT_UL	signed long
Mangling	ITKM_F	F
C++ Type	ITKT_F	float
Mangling	ITKM_D	D
C++ Type	ITKT_D	double

Table 9.2: CMake wrapping mangling variables, their values, and the corresponding CMake C++ type variables and their values for plain old datatypes (PODS).

	CMake Variable	Value
Mangling	ITKM_C\${type}	C\${type}
C++ Type	ITKT_CS\${type}	std::complex<\${type}>
Mangling	ITKM_A\${type}	A\${type}
C++ Type	ITKT_A\${type}	itk::Array<\${type}>
Mangling	ITKM_FA\${ITKM_-\$type} \${dim}	FA\${ITKM_-\$type} \${dim}
C++ Type	ITKT_FA\${ITKM_-\$type} \${dim}	itk::FixedArray<\${ITKT_-\$type}, \${dim}>
Mangling	ITKM_RGB\${dim}	RGB\${dim}
C++ Type	ITKT_RGB\${dim}	itk::RGBPixel<\${dim}>
Mangling	ITKM_RGBA\${dim}	RGBA\${dim}
C++ Type	ITKT_RGBA\${dim}	itk::RGBAPixel<\${dim}>
Mangling	ITKM_V\${ITKM_-\$type} \${dim}	V\${ITKM_-\$type} \${dim}
C++ Type	ITKT_V\${ITKM_-\$type} \${dim}	itk::Vector<\${ITKT_-\$type}, \${dim}>
Mangling	ITKM_CV\${ITKM_-\$type} \${dim}	CV\${ITKM_-\$type} \${dim}
C++ Type	ITKT_CV\${ITKM_-\$type} \${dim}	itk::CovariantVector<\${ITKT_-\$type}, \${dim}>
Mangling	ITKM_VLV\${ITKM_-\$type} \${dim}	VLV\${ITKM_-\$type} \${dim}
C++ Type	ITKT_VLV\${ITKM_-\$type} \${dim}	itk::VariableLengthVector<\${ITKT_-\$type}, \${dim}>
Mangling	ITKM_SSRT\${ITKM_-\$type} \${dim}	SSRT\${ITKM_-\$type} \${dim}
C++ Type	ITKT_SSRT\${ITKM_-\$type} \${dim}	itk::SymmetricSecondRankTensor<\${ITKT_-\$type}, \${dim}>

Table 9.3: CMake wrapping mangling variables, their values, and the corresponding CMake C++ type variables and their values for other ITK pixel types.

	CMake Variable	Value
Mangling	ITKM_O\${dim}	O\${dim}
C++ Type	ITKT_O\${dim}	itk::Offset<\${dim}>
Mangling	ITKM_CI\${ITKM_-\$type}\${dim}	CI\${ITKM_-\$type}\${dim}
C++ Type	ITKT_CI\${ITKM_-\$type}\${dim}	itk::ContinuousIndex<\${ITKT_-\$type}, \${dim}>
Mangling	ITKM_P\${ITKM_-\$type}\${dim}	P\${ITKM_-\$type}\${dim}
C++ Type	ITKT_P\${ITKM_-\$type}\${dim}	itk::Point<\${ITKT_-\$type}, \${dim}>
Mangling	ITKM_I\${ITKM_-\$type}\${dim}	I\${ITKM_-\$type}\${dim}
C++ Type	ITKT_I\${ITKM_-\$type}\${dim}	itk::Image<\${ITKT_-\$type}, \${dim}>
Mangling	ITKM_VI\${ITKM_-\$type}\${dim}	VI\${ITKM_-\$type}\${dim}
C++ Type	ITKT_VI\${ITKM_-\$type}\${dim}	itk::VectorImage<\${ITKT_-\$type}, \${dim}>
Mangling	ITKM_SO\${dim}	SO\${dim}
C++ Type	ITKT_SO\${dim}	itk::SpatialObject<\${dim}>
Mangling	ITKM_SE\${dim}	SE\${dim}
C++ Type	ITKT_SE\${dim}	itk::FlatStructuringElement<\${dim}>
Mangling	ITKM_H\${ITKM_-\$type}	H\${ITKM_-\$type}
C++ Type	ITKT_H\${ITKM_-\$type}	itk::Statistics::Histogram<\${ITKT_-\$type}>

Table 9.4: CMake wrapping mangling variables, their values, and the corresponding CMake C++ type variables and their values for basic ITK types.

```
itk_wrap_class("itk::ImageSpatialObject" POINTER)
# unsigned char required for the ImageMaskSpatialObject
UNIQUE(types "UC;${WRAP_ITK_SCALAR}")

foreach(d ${ITK_WRAP_IMAGE_DIMS})
    foreach(t ${types})
        itk_wrap_template("${d}${ITKM_${t}}" "${d},${ITKT_${t}}")
    endforeach()
endforeach()
itk_end_wrap_class()
```

In addition to `itk_wrap_template`, there are template wrapping macros specialized for wrapping image filters. The highest level macro is `itk_wrap_image_filter`, which is used for wrapping image filters that need one or more image parameters of the same type. This macro has two required arguments. The first argument is a semicolon delimited CMake list of pixel types. The second argument is the number of image template arguments for the filter. An optional third argument is a dimensionality condition to restrict the dimensions that the filter can be instantiated. The dimensionality condition can be a number indicating the dimension allowed, a semicolon delimited CMake list of dimensions, or a string of the form `n+`, where `n` is a number, to indicate that instantiations are allowed for dimension `n` and above. The wrapping specification for `itk::ThresholdMaximumConnectedComponentsImageFilter` is

```
itk_wrap_class("itk::ThresholdMaximumConnectedComponentsImageFilter" POINTER)
    itk_wrap_image_filter("${WRAP_ITK_INT}" 1 2+)
itk_end_wrap_class()
```

If it is desirable or required to instantiate an image filter with different image types, the `itk_wrap_image_filter_combinations` macro is applicable. This macro takes a variable number of parameters, where each parameter is a list of the possible image pixel types for the corresponding filter template parameters. A condition to restrict dimensionality may again be optionally passed as the last argument. For example, wrapping for `itk::VectorMagnitudeImageFilter` is specified with

```
itk_wrap_class("itk::VectorMagnitudeImageFilter" POINTER_WITH_SUPERCLASS)
    itk_wrap_image_filter_combinations("${WRAP_ITK_COV_VECTOR_REAL}" "${WRAP_ITK_SCALAR}")
itk_end_wrap_class()
```

The final template wrapping macro is `itk_wrap_image_filter_types`. This macro takes a variable number of arguments that should correspond to the image pixel types in the filter's template parameter list. Again, an optional dimensionality condition can be specified as the last argument. For example, wrapping for `itk::RGBToLuminanceImageFilter` is specified with

```

itk_wrap_class("itk::RGBToLuminanceImageFilter" POINTER_WITH_SUPERCLASS)
if(ITK_WRAP_rgb_unsigned_char AND ITK_WRAP_unsigned_char)
    itk_wrap_image_filter_types(RGBUC UC)
endif(ITK_WRAP_rgb_unsigned_char AND ITK_WRAP_unsigned_char)

if(ITK_WRAP_rgb_unsigned_short AND ITK_WRAP_unsigned_short)
    itk_wrap_image_filter_types(RGBUS US)
endif(ITK_WRAP_rgb_unsigned_short AND ITK_WRAP_unsigned_short)

if(ITK_WRAP_rgba_unsigned_char AND ITK_WRAP_unsigned_char)
    itk_wrap_image_filter_types(RGBAUC UC)
endif(ITK_WRAP_rgba_unsigned_char AND ITK_WRAP_unsigned_char)

if(ITK_WRAP_rgba_unsigned_short AND ITK_WRAP_unsigned_short)
    itk_wrap_image_filter_types(RGBAUS US)
endif(ITK_WRAP_rgba_unsigned_short AND ITK_WRAP_unsigned_short)
itk_end_wrap_class()

```

In some cases, it is necessary to specify the headers required to build wrapping sources for a class. To specify additional headers to include in the generated wrapping C++ source, use the **itk_wrap_include** macro. This macro takes the name of the header to include, and it can be called multiple times.

By default, the class wrapping macros include a header whose filename corresponds to the name of the class to be wrapped according to ITK naming conventions. To override the default behavior, set the CMake variable `WRAPPER_AUTO_INCLUDE_HEADERS` to OFF before calling `itk_wrap_class`. For example,

```

set(WRAPPER_AUTO_INCLUDE_HEADERS OFF)
itk_wrap_include("itkTransformFileReader.h")
itk_wrap_class("itk::TransformFileReaderTemplate" POINTER)
foreach(t ${WRAP_ITK_REAL})
    itk_wrap_template("${ITKM_${t}}" "${ITKT_${t}}")
endforeach()
itk_end_wrap_class()

```

There are a number of convenience CMake macros available to manipulate lists of template parameters. These macros take the variable name to populate with their output as the first argument followed by input arguments. The **itk_wrap_filter_dims** macro will process the dimensionality condition previously described for the filter template wrapping macros. **DECREMENT**, **INCREMENT** are macros that operate on dimensions. The **INTERSECTION** macro finds the intersection of two list arguments. Finally, the **UNIQUE** macro removes duplicates from the given list.

9.6 Third-Party Dependencies

When an ITK module depends on another ITK module, it simply lists its dependencies as described in Section 9.1. A module can also depend on non-ITK third-party libraries. This third-party library can be encapsulated in an ITK module – see examples in the `ITK/Modules/ThirdParty` directory.

Or, the dependency can be built or installed on the system and found with CMake. This section describes how to add the CMake configuration to a module for it to find and use a third-party library dependency.

9.6.1 itk-module-init.cmake

The `itk-module-init.cmake` file, if present, is found in the top level directory of the module next to the `itk-module.cmake` file. This file informs CMake of the build configuration and location of the third-party dependency. To inform CMake about the OpenCV library, use the `find_package` command,

```
find_package(OpenCV REQUIRED)
```

9.6.2 CMakeList.txt

A few additions are required to the top level `CMakeLists.txt` of the module.

First, the `itk-module-init.cmake` file should be explicitly included when building the module externally against an existing ITK build tree.

```
if(NOT ITK_SOURCE_DIR)
  include(itk-module-init.cmake)
endif()
project(ITKVideoBridgeOpenCV)
```

Optionally, the dependency libraries are added to the `<module-name>_LIBRARIES` variable. Alternatively, if the module creates a library, publically link to the dependency libraries. Our `ITKVideoBridgeOpenCV` module example creates its own library, named `ITKVideoBridgeOpenCV`, and publically links to the OpenCV libraries.

`CMakeLists.txt:` `set(ITKVideoBridgeOpenCV_LIBRARIES ITKVideoBridgeOpenCV)`

`src/CMakeLists.txt:` `target_link_libraries(ITKVideoBridgeOpenCV LINK_PUBLIC ${OpenCV_LIBS})`

Next, CMake export code is created. This code is loaded by CMake when another project uses this module. The export code stores where the dependency was located when the module was built, and how CMake should find it. Two versions are required for the build tree and for the install tree.

```
# When this module is loaded by an app, load OpenCV too.
set(ITKVideoBridgeOpenCV_EXPORT_CODE_INSTALL "
set(OpenCV_DIR \"${OpenCV_DIR}\")
find_package(OpenCV REQUIRED)
")
set(ITKVideoBridgeOpenCV_EXPORT_CODE_BUILD "
if(NOT ITK_BINARY_DIR)
    set(OpenCV_DIR \"${OpenCV_DIR}\")
    find_package(OpenCV REQUIRED)
endif()
")
```

Finally, set the `<module-name>_SYSTEM_INCLUDE_DIRS` and `<module-name>_SYSTEM_LIBRARY_DIRS`, if required, to append compilation header directories and library linking directories for this module.

```
set(ITKVideoBridgeOpenCV_SYSTEM_INCLUDE_DIRS ${OpenCV_INCLUDE_DIRS})
set(ITKVideoBridgeOpenCV_SYSTEM_LIBRARY_DIRS ${OpenCV_LIB_DIR})
```


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 Git source code repository, you need to know something about this process (see [1.3](#) on page [5](#) to learn more about obtaining ITK using Git). 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.

10.1 Git Source Code Repository

Git) is a tool for version control. It is a valuable resource for software projects involving multiple developers. The primary purpose of Git is to keep track of changes to software. Git 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. Git 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 Git repository. When developers introduce changes in the system, Git 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 Git repository. The principal advantage of a system like Git 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.

The ITK community use Git, and the Google web software tool Gerrit (<http://review.source.kitware.com>) to facilitate a structured, orderly method for developers to contribute new code and bug fixes to ITK. The Gerrit review process allows anyone to submit a proposed change to ITK, after which it will be reviewed by other developers before being approved and merged into ITK. For more information, see <https://www.itk.org/Wiki/ITK/Git/Develop>.

10.2 CDash Regression Testing System

One of the unique features of the ITK software process is its use of the CDash regression testing system (<http://www.cdash.org>). In a nutshell, what CDash 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 10.1. The most recent dashboard is accessible from <https://www.itk.org/ITK/resources/testing.html>). Since all users and developers of ITK can view the Web page, the CDash 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 Git.

Note that CDash 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://www.cdash.org>

CDash 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 test's 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 runtime 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.

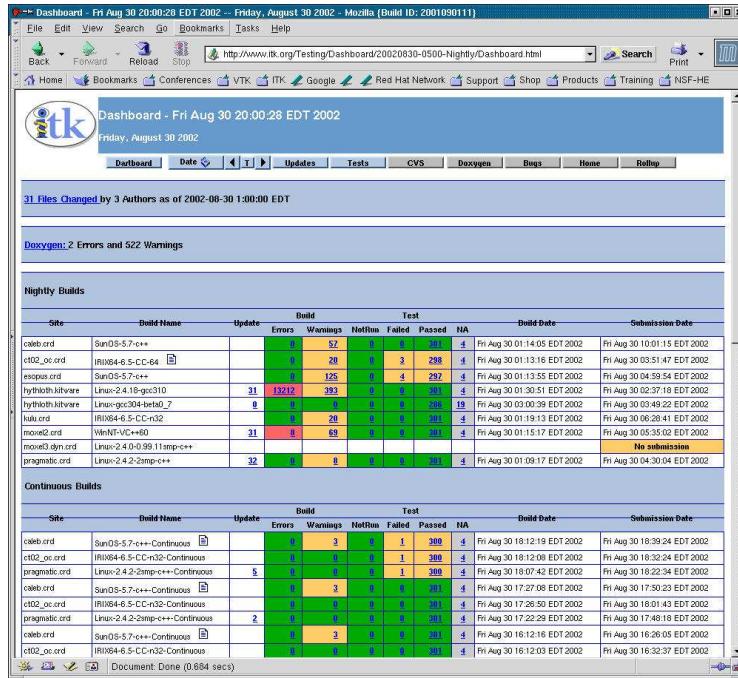


Figure 10.1: On-line presentation of the quality dashboard generated by CDash.

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 10.1 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 Git 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 CDash is that it maintains a history of changes to the source code (by coordinating with Git) 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.

10.3 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 Git. When changed or new code is checked into Git, the CDash 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 CDash 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 CDash 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 Git repository, updating documentation, and producing new release packages. Although additional testing is performed to insure the consistency of the package, keeping the daily Git 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 Git 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.

10.4 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 accidentally 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, Git, and CDash) 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.

Appendices

APPENDIX
ONE

LICENSES

A.1 Insight Toolkit License

Apache License
Version 2.0, January 2004
<http://www.apache.org/licenses/>

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation

source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licenser or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licenser for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the

Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.
4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:
 - (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and
 - (b) You must cause any modified files to carry prominent notices stating that You changed the files; and
 - (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
 - (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or,

within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.
6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.
7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.
8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be

liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

See the License for the specific language governing permissions and limitations under the License.

A.2 Third Party Licenses

The Insight Toolkit bundles a number of third party libraries that are used internally. The licenses of these libraries are as follows.

A.2.1 DICOM Parser

```
/*=====
```

```
Program:  DICOMParser
Module:   Copyright.txt
Language: C++
Date:     $Date$
Version:  $Revision$
```

Copyright (c) 2003 Matt Turek
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- * The name of Matt Turek nor the names of any contributors may be used to endorse or promote products derived from this software without specific prior written permission.
- * Modified source versions must be plainly marked as such, and must not be misrepresented as being the original software.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS ''AS IS'', AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE

IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHORS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

=====*/

A.2.2 Double Conversion

Copyright 2006-2011, the V8 project authors. All rights reserved.
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:

- * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- * Neither the name of Google Inc. nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

A.2.3 Expat

Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

A.2.4 GDCM

/*=====

Program: GDCM (Grassroots DICOM). A DICOM library

Copyright (c) 2006-2016 Mathieu Malaterre
Copyright (c) 1993-2005 CREATIS
(CREATIS = Centre de Recherche et d'Applications en Traitement de l'Image)
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright notice,

this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

- * Neither name of Mathieu Malaterre, or CREATIS, nor the names of any contributors (CNRS, INSERM, UCB, Universite Lyon I), may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS 'AS IS' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHORS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

=====*/

A.2.5 GIFTI

The gifticlib code is released into the public domain. Developers are encouraged to incorporate the library into their application, and to contribute changes or enhancements to gifticlib.

Author: Richard Reynolds, SSCC, DIRP, NIMH, National Institutes of Health
May 13, 2008 (release version 1.0.0)

<http://www.nitrc.org/projects/gifti>

A.2.6 HDF5

Copyright Notice and License Terms for
HDF5 (Hierarchical Data Format 5) Software Library and Utilities

HDF5 (Hierarchical Data Format 5) Software Library and Utilities
Copyright 2006-2011 by The HDF Group.

NCSA HDF5 (Hierarchical Data Format 5) Software Library and Utilities

Copyright 1998-2006 by the Board of Trustees of the University of Illinois.

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted for any purpose (including commercial purposes) provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the following disclaimer in the documentation and/or materials provided with the distribution.
3. In addition, redistributions of modified forms of the source or binary code must carry prominent notices stating that the original code was changed and the date of the change.
4. All publications or advertising materials mentioning features or use of this software are asked, but not required, to acknowledge that it was developed by The HDF Group and by the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign and credit the contributors.
5. Neither the name of The HDF Group, the name of the University, nor the name of any Contributor may be used to endorse or promote products derived from this software without specific prior written permission from The HDF Group, the University, or the Contributor, respectively.

DISCLAIMER:

THIS SOFTWARE IS PROVIDED BY THE HDF GROUP AND THE CONTRIBUTORS "AS IS" WITH NO WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED. In no event shall The HDF Group or the Contributors be liable for any damages suffered by the users arising out of the use of this software, even if advised of the possibility of such damage.

Contributors: National Center for Supercomputing Applications (NCSA) at the University of Illinois, Fortner Software, Unidata Program Center (netCDF), The Independent JPEG Group (JPEG), Jean-loup Gailly and Mark Adler (gzip),

and Digital Equipment Corporation (DEC).

Portions of HDF5 were developed with support from the Lawrence Berkeley National Laboratory (LBNL) and the United States Department of Energy under Prime Contract No. DE-AC02-05CH11231.

Portions of HDF5 were developed with support from the University of California, Lawrence Livermore National Laboratory (UC LLNL).

The following statement applies to those portions of the product and must be retained in any redistribution of source code, binaries, documentation, and/or accompanying materials:

This work was partially produced at the University of California, Lawrence Livermore National Laboratory (UC LLNL) under contract no. W-7405-ENG-48 (Contract 48) between the U.S. Department of Energy (DOE) and The Regents of the University of California (University) for the operation of UC LLNL.

DISCLAIMER:

This work was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately-owned rights. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

A.2.7 JPEG

The authors make NO WARRANTY or representation, either express or implied, with respect to this software, its quality, accuracy, merchantability, or fitness for a particular purpose. This software is provided "AS IS", and you, its user, assume the entire risk as to its quality and accuracy.

This software is copyright (C) 1991-2010, Thomas G. Lane, Guido Vollbeding. All Rights Reserved except as specified below.

Permission is hereby granted to use, copy, modify, and distribute this software (or portions thereof) for any purpose, without fee, subject to these conditions:

- (1) If any part of the source code for this software is distributed, then this README file must be included, with this copyright and no-warranty notice unaltered; and any additions, deletions, or changes to the original files must be clearly indicated in accompanying documentation.
- (2) If only executable code is distributed, then the accompanying documentation must state that "this software is based in part on the work of the Independent JPEG Group".
- (3) Permission for use of this software is granted only if the user accepts full responsibility for any undesirable consequences; the authors accept NO LIABILITY for damages of any kind.

These conditions apply to any software derived from or based on the IJG code, not just to the unmodified library. If you use our work, you ought to acknowledge us.

Permission is NOT granted for the use of any IJG author's name or company name in advertising or publicity relating to this software or products derived from it. This software may be referred to only as "the Independent JPEG Group's software".

We specifically permit and encourage the use of this software as the basis of commercial products, provided that all warranty or liability claims are assumed by the product vendor.

ansi2knr.c is included in this distribution by permission of L. Peter Deutsch, sole proprietor of its copyright holder, Aladdin Enterprises of Menlo Park, CA. ansi2knr.c is NOT covered by the above copyright and conditions, but instead by the usual distribution terms of the Free Software Foundation; principally, that you must include source code if you redistribute it. (See the file

ansi2knr.c for full details.) However, since ansi2knr.c is not needed as part of any program generated from the IJG code, this does not limit you more than the foregoing paragraphs do.

The Unix configuration script "configure" was produced with GNU Autoconf. It is copyright by the Free Software Foundation but is freely distributable. The same holds for its supporting scripts (config.guess, config.sub, ltmain.sh). Another support script, install-sh, is copyright by X Consortium but is also freely distributable.

The IJG distribution formerly included code to read and write GIF files. To avoid entanglement with the Unisys LZW patent, GIF reading support has been removed altogether, and the GIF writer has been simplified to produce "uncompressed GIFs". This technique does not use the LZW algorithm; the resulting GIF files are larger than usual, but are readable by all standard GIF decoders.

We are required to state that

"The Graphics Interchange Format(c) is the Copyright property of CompuServe Incorporated. GIF(sm) is a Service Mark property of CompuServe Incorporated."

A.2.8 KWSys

KWSys - Kitware System Library
Copyright 2000-2009 Kitware, Inc., Insight Software Consortium
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- * Neither the names of Kitware, Inc., the Insight Software Consortium, nor the names of their contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

A.2.9 MetaIO

MetaIO - Medical Image I/O

The following license applies to all code, without exception, in the MetaIO library.

```
/*=====
```

Copyright 2000-2014 Insight Software Consortium
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- * Neither the name of the Insight Software Consortium nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT

OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

=====*/

/*=====

Copyright (c) 1999-2007 Insight Software Consortium
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- * The name of the Insight Software Consortium, nor the names of any consortium members, nor of any contributors, may be used to endorse or promote products derived from this software without specific prior written permission.
- * Modified source versions must be plainly marked as such, and must not be misrepresented as being the original software.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDER AND CONTRIBUTORS ''AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHORS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

=====*/

A.2.10 Netlib's SLATEC

This code is in the public domain. From <http://www.netlib.org/slatec/guide>:

SECTION 4. OBTAINING THE LIBRARY

The Library is in the public domain and distributed by the Energy Science and Technology Software Center.

Energy Science and Technology Software Center
P.O. Box 1020
Oak Ridge, TN 37831

Telephone 615-576-2606
E-mail estsc%al.adonis.mrouter@zeus.osti.gov

A.2.11 NIFTI

Niftilib has been developed by members of the NIFTI DFWG and volunteers in the neuroimaging community and serves as a reference implementation of the nifti-1 file format.

<http://nifti.nimh.nih.gov/>

Nifticlib code is released into the public domain, developers are encouraged to incorporate niftilib code into their applications, and, to contribute changes and enhancements to niftilib.

A.2.12 NrrdIO

License -----

NrrdIO: stand-alone code for basic nrrd functionality
Copyright (C) 2013, 2012, 2011, 2010, 2009 University of Chicago
Copyright (C) 2008, 2007, 2006, 2005 Gordon Kindlmann
Copyright (C) 2004, 2003, 2002, 2001, 2000, 1999, 1998 University of Utah

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
 2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
 3. This notice may not be removed or altered from any source distribution.
-
- General information -----

** NOTE: These source files have been copied and/or modified from Teem,
** <<http://teem.sf.net>>. Teem is licensed under a weakened GNU Lesser Public
** License (the weakening is to remove burdens on those releasing binaries
** that statically link against Teem) . The non-reciprocal licensing defined
** above applies to only the source files in the NrrdIO distribution, and not
** to Teem.

NrrdIO is a modified and highly abbreviated version of the Teem. NrrdIO contains only the source files (or portions thereof) required for creating and destroying nrrds, and for getting them into and out of files. The NrrdIO sources are created from the Teem sources by using GNU Make (pre-GNUMakefile in the NrrdIO distribution).

NrrdIO makes it very easy to add support for the NRRD file format to your program, which is a good thing considering and design and flexibility of the NRRD file format, and the existence of the "unu" command-line tool for operating on nrrds. Using NrrdIO requires exactly one header file, "NrrdIO.h", and exactly one library, libNrrdIO.

Currently, the API presented by NrrdIO is a strict subset of the Teem API.

There is no additional encapsulation or abstraction. This could be annoying in the sense that you still have to deal with the biff (for error messages) and the air (for utilities) library function calls. Or it could be good and sane in the sense that code which uses NrrdIO can be painlessly "upgraded" to use more of Teem. Also, the API documentation for the same functionality in Teem will apply directly to NrrdIO.

NrrdIO was originally created with the help of Josh Cates in order to add support for the NRRD file format to the Insight Toolkit (ITK).

NrrdIO API crash course

Please read <<http://teem.sourceforge.net/nrrd/lib.html>>. The functions that are explained in detail are all present in NrrdIO. Be aware, however, that NrrdIO currently supports ONLY the NRRD file format, and not: PNG, PNM, VTK, or EPS.

The functionality in Teem's nrrd library which is NOT in NrrdIO is basically all those non-trivial manipulations of the values in the nrrd, or their ordering in memory. Still, NrrdIO can do a fair amount, namely all the functions listed in these sections of the "Overview of rest of API" in the above web page:

- Basic "methods"
 - Manipulation of per-axis meta-information
 - Utility functions
 - Comments in nrrd
 - Key/value pairs
 - Endianness (byte ordering)
 - Getting/Setting values (crude!)
 - Input from, Output to files
-
-

Files comprising NrrdIO

NrrdIO.h: The single header file that declares all the functions and variables that NrrdIO provides.

sampleIO.c: Tiny little command-line program demonstrating the basic NrrdIO API. Read this for examples of how NrrdIO is used to read and write NRRD

files.

CMakeLists.txt: to build NrrdIO with CMake

pre-GNUmakefile: how NrrdIO sources are created from the Teem sources. Requires that TEEM_SRC_ROOT be set, and uses the following two files.

tail.pl, unteem.pl: used to make small modifications to the source files to convert them from Teem to NrrdIO sources

mangle.pl: used to generate a #include file for name-mangling the external symbols in the NrrdIO library, to avoid possible problems with programs that link with both NrrdIO and the rest of Teem.

preamble.c: the preamble describing the non-copyleft licensing of NrrdIO.

qnanhibit.c: discover a variable which, like endianness, is architecture dependent and which is required for building NrrdIO (as well as Teem), but unlike endianness, is completely obscure and unheard of.

encodingBzip2.c, formatEPS.c, formatPNG.c, formatPNM.c, formatText.c, formatVTK.c: These files create stubs for functionality which is fully present in Teem, but which has been removed from NrrdIO in the interest of simplicity. The filenames are in fact unfortunately misleading, but they should be understood as listing the functionality that is MISSING in NrrdIO.

All other files: copied/modified from the air, biff, and nrrd libraries of Teem.

A.2.13 OpenJPEG

```
/*
 * Copyright (c) 2002-2012, Communications and Remote Sensing Laboratory,
 * Universite catholique de Louvain (UCL), Belgium
 * Copyright (c) 2002-2012, Professor Benoit Macq
 * Copyright (c) 2003-2012, Antonin Descampe
 * Copyright (c) 2003-2009, Francois-Olivier Devaux
 * Copyright (c) 2005, Herve Drolon, FreeImage Team
 * Copyright (c) 2002-2003, Yannick Verschueren
 * Copyright (c) 2001-2003, David Janssens
 * Copyright (c) 2011-2012, Centre National d'Etudes Spatiales (CNES), France
 * Copyright (c) 2012, CS Systemes d'Information, France
 */
```

```
* All rights reserved.  
*  
* Redistribution and use in source and binary forms, with or without  
* modification, are permitted provided that the following conditions  
* are met:  
* 1. Redistributions of source code must retain the above copyright  
* notice, this list of conditions and the following disclaimer.  
* 2. Redistributions in binary form must reproduce the above copyright  
* notice, this list of conditions and the following disclaimer in the  
* documentation and/or other materials provided with the distribution.  
*  
* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS 'AS IS'  
* AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE  
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE  
* ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE  
* LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR  
* CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF  
* SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS  
* INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN  
* CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)  
* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE  
* POSSIBILITY OF SUCH DAMAGE.  
*/
```

A.2.14 PNG

This copy of the libpng notices is provided for your convenience. In case of any discrepancy between this copy and the notices in the file png.h that is included in the libpng distribution, the latter shall prevail.

COPYRIGHT NOTICE, DISCLAIMER, and LICENSE:

If you modify libpng you may insert additional notices immediately following this sentence.

This code is released under the libpng license.

libpng versions 1.2.6, August 15, 2004, through 1.6.9, February 6, 2014, are Copyright (c) 2004, 2006-2014 Glenn Randers-Pehrson, and are distributed according to the same disclaimer and license as libpng-1.2.5 with the following individual added to the list of Contributing Authors

Cosmin Truta

libpng versions 1.0.7, July 1, 2000, through 1.2.5 - October 3, 2002, are Copyright (c) 2000-2002 Glenn Randers-Pehrson, and are distributed according to the same disclaimer and license as libpng-1.0.6 with the following individuals added to the list of Contributing Authors

Simon-Pierre Cadieux
Eric S. Raymond
Gilles Vollant

and with the following additions to the disclaimer:

There is no warranty against interference with your enjoyment of the library or against infringement. There is no warranty that our efforts or the library will fulfill any of your particular purposes or needs. This library is provided with all faults, and the entire risk of satisfactory quality, performance, accuracy, and effort is with the user.

libpng versions 0.97, January 1998, through 1.0.6, March 20, 2000, are Copyright (c) 1998, 1999 Glenn Randers-Pehrson, and are distributed according to the same disclaimer and license as libpng-0.96, with the following individuals added to the list of Contributing Authors:

Tom Lane
Glenn Randers-Pehrson
Willem van Schaik

libpng versions 0.89, June 1996, through 0.96, May 1997, are Copyright (c) 1996, 1997 Andreas Dilger
Distributed according to the same disclaimer and license as libpng-0.88, with the following individuals added to the list of Contributing Authors:

John Bowler
Kevin Bracey
Sam Bushell
Magnus Holmgren
Greg Roelofs
Tom Tanner

libpng versions 0.5, May 1995, through 0.88, January 1996, are Copyright (c) 1995, 1996 Guy Eric Schalnat, Group 42, Inc.

For the purposes of this copyright and license, "Contributing Authors" is defined as the following set of individuals:

Andreas Dilger
Dave Martindale
Guy Eric Schalnat
Paul Schmidt
Tim Wegner

The PNG Reference Library is supplied "AS IS". The Contributing Authors and Group 42, Inc. disclaim all warranties, expressed or implied, including, without limitation, the warranties of merchantability and of fitness for any purpose. The Contributing Authors and Group 42, Inc. assume no liability for direct, indirect, incidental, special, exemplary, or consequential damages, which may result from the use of the PNG Reference Library, even if advised of the possibility of such damage.

Permission is hereby granted to use, copy, modify, and distribute this source code, or portions hereof, for any purpose, without fee, subject to the following restrictions:

1. The origin of this source code must not be misrepresented.
2. Altered versions must be plainly marked as such and must not be misrepresented as being the original source.
3. This Copyright notice may not be removed or altered from any source or altered source distribution.

The Contributing Authors and Group 42, Inc. specifically permit, without fee, and encourage the use of this source code as a component to supporting the PNG file format in commercial products. If you use this source code in a product, acknowledgment is not required but would be appreciated.

A "png_get_copyright" function is available, for convenient use in "about" boxes and the like:

```
printf("%s",png_get_copyright(NULL));
```

Also, the PNG logo (in PNG format, of course) is supplied in the files "pngbar.png" and "pngbar.jpg" (88x31) and "pngnow.png" (98x31).

Libpng is OSI Certified Open Source Software. OSI Certified Open Source is a certification mark of the Open Source Initiative.

Glenn Randers-Pehrson
glenrrp at users.sourceforge.net
February 6, 2014

A.2.15 TIFF

Copyright (c) 1988-1997 Sam Leffler
Copyright (c) 1991-1997 Silicon Graphics, Inc.

Permission to use, copy, modify, distribute, and sell this software and its documentation for any purpose is hereby granted without fee, provided that (i) the above copyright notices and this permission notice appear in all copies of the software and related documentation, and (ii) the names of Sam Leffler and Silicon Graphics may not be used in any advertising or publicity relating to the software without the specific, prior written permission of Sam Leffler and Silicon Graphics.

THE SOFTWARE IS PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EXPRESS, IMPLIED OR OTHERWISE, INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

IN NO EVENT SHALL SAM LEFFLER OR SILICON GRAPHICS BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES OF ANY KIND, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER OR NOT ADVISED OF THE POSSIBILITY OF DAMAGE, AND ON ANY THEORY OF LIABILITY, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

A.2.16 VNL

```
#ifndef vxl_copyright_h_
#define vxl_copyright_h_

// Copyright 2000-2013 VXL Contributors
// All rights reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions
```

```
// are met:  
//  
// * Redistributions of source code must retain the above copyright  
//   notice, this list of conditions and the following disclaimer.  
//  
// * Redistributions in binary form must reproduce the above copyright  
//   notice, this list of conditions and the following disclaimer in the  
//   documentation and/or other materials provided with the distribution.  
//  
// * Neither the names of the copyright holders nor the names of their  
//   contributors may be used to endorse or promote products derived  
//   from this software without specific prior written permission.  
//  
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS  
// "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT  
// LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS  
// FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE  
// COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT,  
// INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES  
// (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR  
// SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)  
// HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,  
// STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)  
// ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED  
// OF THE POSSIBILITY OF SUCH DAMAGE.
```

```
#endif // vxl_copyright_h_
```

A.2.17 ZLIB

Acknowledgments:

The deflate format used by zlib was defined by Phil Katz. The deflate and zlib specifications were written by L. Peter Deutsch. Thanks to all the people who reported problems and suggested various improvements in zlib; they are too numerous to cite here.

Copyright notice:

(C) 1995-2004 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages

arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly
jloup@gzip.org

Mark Adler
madler@alumni.caltech.edu

If you use the zlib library in a product, we would appreciate *not* receiving lengthy legal documents to sign. The sources are provided for free but without warranty of any kind. The library has been entirely written by Jean-loup Gailly and Mark Adler; it does not include third-party code.

If you redistribute modified sources, we would appreciate that you include in the file ChangeLog history information documenting your changes. Please read the FAQ for more information on the distribution of modified source versions.

BIBLIOGRAPHY

- [1] M. H. Austern. *Generic Programming and the STL*. Professional Computing Series. Addison-Wesley, 1999. [3.2.1](#)
- [2] K.R. Castleman. *Digital Image Processing*. Prentice Hall, Upper Saddle River, NJ, 1996. [6.4.1](#), [6.4.2](#)
- [3] 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](#), [8.6](#)
- [4] R.C. Gonzalez and R.E. Woods. *Digital Image Processing*. Addison-Wesley, Reading, MA, 1993. [6.4.1](#), [6.4.1](#), [6.4.2](#)
- [5] H. Gray. *Gray's Anatomy*. Merchant Book Company, sixteenth edition, 2003. [4.1.5](#)
- [6] 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](#)
- [7] D. Malacara. *Color Vision and Colorimetry: Theory and Applications*. SPIE PRESS, 2002. [4.1.5](#), [4.1.5](#)
- [8] D. Musser and A. Saini. *STL Tutorial and Reference Guide*. Professional Computing Series. Addison-Wesley, 1996. [3.2.1](#)
- [9] G. Wyszecki. *Color Science: Concepts and Methods, Quantitative Data and Formulae*. Wiley-Interscience, 2000. [4.1.5](#), [4.1.5](#)

INDEX

Accept()
 itk::Mesh, [97](#)
AddVisitor()
 itk::Mesh, [97](#)

BoundaryFeature, [81](#)
BufferedRegion, [194](#)

CDash, [228](#)
CellAutoPointer, [69](#)
 TakeOwnership(), [70, 72, 75, 78, 84](#)
CellBoundaryFeature, [81](#)
CellDataContainer
 Begin(), [73, 76](#)
 ConstIterator, [73, 76](#)
 End(), [73, 76](#)
 Iterator, [73, 76](#)
CellDataIterator
 increment, [73, 76](#)
 Value(), [73, 76](#)
CellInterface
 iterating points, [95](#)
 PointIdsBegin(), [95](#)
 PointIdsEnd(), [95](#)
CellInterfaceVisitor, [92, 93](#)
 requirements, [92, 94](#)
 Visit(), [92, 94](#)
CellIterator
 increment, [71](#)

 Value(), [71](#)
CellMultiVisitorType, [97](#)
CellsContainer
 Begin(), [71, 80, 85, 89](#)
 End(), [71, 80, 85, 89](#)
CellType
 creation, [70, 72, 75, 78, 84](#)
 GetNumberOfPoints(), [71](#)
 PointIdIterator, [80, 86](#)
 PointIdsBegin(), [80, 86](#)
 PointIdsEnd(), [80, 86](#)
 Print(), [71](#)
CellVisitor, [92, 93, 96](#)
CMake, [12](#)
 downloading, [12](#)
Command/Observer design pattern, [28](#)
const-correctness, [62, 63](#)
ConstIterator, [62, 63](#)
convolution
 kernels, [168](#)
 operators, [168](#)
convolution filtering, [168](#)

Dashboard, [228](#)
data object, [30, 193](#)
data processing pipeline, [31, 193](#)
down casting, [71](#)
Downloading, [5](#)

- edge detection, 165
- error handling, 27
- event handling, 28
- exceptions, 27
- factory, 25
- filter, 31, 193
 - overview of creation, 194
- forward iteration, 142
- garbage collection, 26
- Gaussian blurring, 171
- Generic Programming, 141
- generic programming, 24, 141
- GetBoundaryAssignment()
 - itk::Mesh, 83
- GetNumberOfBoundaryFeatures()
 - itk::Mesh, 82
- GetNumberOfFaces()
 - TetrahedronCell, 95
- GetPointId(), 94
- Git, 227
- Hello World, 19
- image region, 193
- ImageAdaptor
 - RGB blue channel, 186
 - RGB green channel, 185
 - RGB red channel, 184
- ImageAdaptors, 181
- ImageLinearIteratorWithIndex
 - 4D images, 152
- InvokeEvent(), 28
- iteration region, 142
- Iterators
 - advantages of, 141
 - and 4D images, 152
 - and bounds checking, 144
 - and image lines, 151
 - and image regions, 142, 145–147
 - and image slices, 154
 - const, 142
 - construction of, 142, 147
- definition of, 141
- Get(), 144
- GetIndex(), 144
- GoToBegin(), 142
- GoToEnd(), 142
- image, 141–180
- image dimensionality, 148
- IsAtBegin(), 144
- IsAtEnd(), 144
- neighborhood, 159–180
- operator++, 143
- operator+=(), 143
- operator-, 143
- operator-=(), 143
- programming interface, 142–146
- Set(), 144
- SetPosition(), 144
- speed, 146, 148
- Value(), 145
- iterators
 - neighborhood
 - and convolution, 168
- ITK
 - advanced configuration, 15
 - building, 16
 - configuration, 14
 - downloading release, 5
 - Git repository, 6, 227
 - history, 9
 - installation, 17
 - mailing list, 8
 - modules, 15
 - itk::ArrowSpatialObject, 111
 - itk::AutomaticTopologyMeshSource, 86
 - AddPoint(), 87
 - AddTetrahedron(), 87
 - header, 86
 - IdentifierArrayType, 86
 - IdentifierType, 86
 - itk::AutoPointer, 69
 - TakeOwnership(), 70, 72, 75, 78, 84
 - itk::BlobSpatialObject, 112
 - itk::Cell

- CellAutoPointer, 69
- itk::CellInterface
 - GetPointId(), 94
- itk::Command, 28
- itk::CovariantVector, 66
 - Header, 64
 - Instantiation, 64
 - itk::PointSet, 64
- itk::CylinderSpatialObject, 113
- itk::DefaultStaticMeshTraits
 - Header, 74
 - Instantiation, 74
- itk::DTITubeSpatialObject, 131
- itk::EllipseSpatialObject, 114
- itk::GaussianSpatialObject, 116
- itk::GroupSpatialObject, 117
- itk::Image, 30
 - Allocate(), 41
 - direction, 46
 - GetPixel(), 43, 50
 - Header, 39
 - Index, 40, 47
 - IndexType, 40
 - Instantiation, 39
 - itk::ImageRegion, 40
 - New(), 40
 - origin, 45
 - PhysicalPoint, 47
 - Pointer, 40
 - read, 41
 - RegionType, 40
 - SetDirection(), 46
 - SetOrigin(), 45
 - SetPixel(), 43
 - SetRegions(), 41
 - SetSpacing(), 45
 - Size, 40
 - SizeType, 40
 - Spacing, 44
 - TransformPhysicalPointToIndex(), 47
 - Vector pixel, 51
- itk::ImageRandomConstIteratorWithIndex, 158–159
 - and statistics, 158
 - begin and end positions, 158
 - example of using, 158–159
 - ReinitializeSeed(), 159
 - sample size, 158
 - SetNumberOfSamples(), 159
- itk::ImageSliceIteratorWithIndex
 - example of using, 155–157
 - IsAtEndOfSlice(), 155
 - IsAtReverseEndOfSlice(), 155
 - NextSlice(), 154
 - PreviousSlice(), 155
 - SetFirstDirection(), 154
 - SetSecondDirection(), 154
- itk::ImageAdaptor
 - Header, 182, 184, 187, 189
 - Instantiation, 182, 184, 187, 189
 - performing computation, 189
 - RGB blue channel, 186
 - RGB green channel, 185
 - RGB red channel, 184
- itk::ImageFileReader
 - GetOutput(), 42
 - Instantiation, 41
 - New(), 41
 - Pointer, 41
 - RGB Image, 50
 - SetFileName(), 42
 - Update(), 42
- itk::ImageLinearIteratorWithIndex, 150–154
 - example of using, 151–152
 - GoToBeginOfLine(), 151
 - GoToEndOfLine(), 151
 - GoToReverseBeginOfLine(), 151
 - IsAtEndOfLine(), 151
 - IsAtReverseEndOfLine(), 151
 - NextLine(), 151
 - PreviousLine(), 151
- itk::ImageMaskSpatialObject, 119
- itk::ImageRegionIterator, 146–148
 - example of using, 146–148
- itk::ImageRegionIteratorWithIndex, 148–150

example of using, 148–150
 itk::ImageSliceIteratorWithIndex, 154–157
 itk::ImageSpatialObject, 118
 itk::ImportImageFilter
 Header, 51
 Instantiation, 51, 52
 New(), 52
 Pointer, 52
 SetRegion(), 52
 itk::LandmarkSpatialObject, 121
 itk::LineCell
 Header, 68
 header, 77, 83
 Instantiation, 69, 72, 75, 77, 79, 84
 SetPointId(), 79, 84
 itk::LineSpatialObject, 122
 itk::MapContainer
 InsertElement(), 57, 59
 itk::Mesh, 31, 67
 Accept(), 93, 97
 AddVisitor(), 93, 97
 BoundaryFeature, 81
 Cell data, 71
 CellInterfaceVisitorImplementation, 92,
 96
 CellAutoPointer, 69
 CellFeatureCount, 82
 CellInterfaceVisitor, 92–94, 96
 CellIterator, 85, 89
 CellsContainer, 80, 85, 89
 CellsIterators, 80
 CellType, 68
 CellType casting, 71
 CellVisitor, 92, 93, 96
 Dynamic, 67
 GetBoundaryAssignment(), 83
 GetCellData(), 73, 76
 GetCells(), 71, 80, 85, 89
 GetNumberOfBoundaryFeatures(), 82
 GetNumberOfCells(), 70
 GetNumberOfPoints(), 67
 GetPoints(), 68, 80, 85
 Header file, 67
 Inserting cells, 70
 Instantiation, 67, 71, 77, 83
 Iterating cell data, 73, 76
 Iterating cells, 71
 K-Complex, 77, 86
 MultiVisitor, 97
 New(), 67, 69, 72, 75, 77, 84
 PixelType, 71, 77, 83
 Pointer, 72, 75, 77, 84
 Pointer(), 67
 PointIterator, 85
 PointsContainer, 80, 85
 PointsIterators, 80
 PointType, 67, 69, 72, 75, 77, 84
 PolyLine, 83
 SetBoundaryAssignment(), 81
 SetCell(), 70, 72, 75, 78, 84
 SetPoint(), 67, 69, 72, 75, 77, 84
 Static, 67
 traits, 68
 itk::MeshSpatialObject, 124
 itk::PixelAccessor
 performing computation, 189
 with parameters, 187, 189
 itk::PointSet, 54
 data iterator, 61
 Dynamic, 54
 GetNumberOfPoints(), 55, 58
 GetPoint(), 55
 GetPointData(), 59–61, 63
 GetPoints(), 57, 58, 61, 63
 Instantiation, 54
 iterating point data, 61
 iterating points, 61
 itk::CovariantVector, 64
 New(), 55
 PixelType, 58
 PointDataContainer, 59
 PointDataIterator, 65
 Pointer, 55
 PointIterator, 63, 64
 points iterator, 61
 PointsContainer, 56

PointType, 55
RGBPixel, 60
SetPoint(), 55, 61, 63, 65
SetPointData(), 58, 59, 61, 63, 65
SetPoints(), 57
Static, 54
Vector pixels, 62
itk::ReadWriteSpatialObject, 135
itk::RGBPixel, 49
GetBlue(), 50
GetGreen(), 50
GetRed(), 50
header, 49
Image, 49
Instantiation, 50, 61
itk::SceneSpatialObject, 133
itk::SpatialObjectToImageStatistics-
Calculator,
136
itk::SpatialObjectHierarchy, 104
itk::SpatialObjectToImageFilter
 Update(), 126
itk::SpatialObjectTransforms, 107
itk::SpatialObjectTreeContainer, 106
itk::SurfaceSpatialObject, 126
itk::TetrahedronCell
 header, 77
 Instantiation, 77, 78
 SetPointId(), 78
itk::TreeContainer, 98
itk::TriangleCell
 header, 77
 Instantiation, 77, 78
 SetPointId(), 78
itk::TubeSpatialObject, 128
itk::Vector, 51
 header, 51
 Instantiation, 51
 itk::Image, 51
 itk::PointSet, 62
itk::VectorContainer
 InsertElement(), 57, 59
itk::VertexCell
 header, 77, 83
 Instantiation, 77, 84
itk::VesselTubeSpatialObject, 129
LargestPossibleRegion, 194
LineCell
 GetNumberOfPoints(), 71
 Print(), 71
 mailing list, 8
 mapper, 31, 193
 mesh region, 194
 modified time, 194
 module, 209
 include, 212
 src, 212
 test, 213
 third-party, 223
 top level, 209
 wrapping, 215
MultiVisitor, 97
Neighborhood iterators
 active neighbors, 176
 as stencils, 176
 boundary conditions, 164
 bounds checking, 164
 construction of, 160
 examples, 165
 inactive neighbors, 176
 radius of, 160
 shaped, 176
NeighborhoodIterator
 examples, 165
 GetCenterPixel(), 162
 GetImagePointer(), 161
 GetIndex(), 163
 GetNeighborhood(), 163
 GetNeighborhoodIndex(), 163
 GetNext(), 162
 GetOffset(), 163
 GetPixel(), 162
 GetPrevious(), 162
 GetRadius(), 161

GetSlice(), 164
 NeedToUseBoundaryConditionOff(), 164
 NeedToUseBoundaryConditionOn(), 164
 OverrideBoundaryCondition(), 164
 ResetBoundaryCondition(), 164
 SetCenterPixel(), 162
 SetNeighborhood(), 163
 SetNext(), 162
 SetPixel(), 162, 164
 SetPrevious(), 162
 Size(), 161
 NeighborhoodIterators, 162, 163
 numerics, 29
 object factory, 25
 pipeline
 downstream, 194
 execution details, 198
 information, 194
 modified time, 194
 overview of execution, 196
 PropagateRequestedRegion, 199
 streaming large data, 195
 ThreadedFilterExecution, 200
 UpdateOutputData, 200
 UpdateOutputInformation, 198
 upstream, 194
 PixelAccessor
 RGB blue channel, 186
 RGB green channel, 185
 RGB red channel, 184
 PointDataContainer
 Begin(), 60
 End(), 60
 increment ++, 60
 InsertElement(), 59
 Iterator, 60
 New(), 59
 Pointer, 59
 PointIdIterator, 80, 86
 PointIdsBegin(), 80, 86, 95
 PointIdsEnd(), 80, 86, 95
 PointsContainer
 Begin(), 57, 68, 80, 85
 End(), 57, 68, 80, 85
 InsertElement(), 57
 Iterator, 57, 68
 New(), 56
 Pointer, 56, 57
 Size(), 58
 Print(), 71
 process object, 31, 193
 ProgressEvent(), 28
 Python, 33
 Quality Dashboard, 228
 reader, 31
 region, 193
 RequestedRegion, 194
 reverse iteration, 142, 145
 scene graph, 32
 SetBoundaryAssignment()
 itk::Mesh, 81
 SetCell()
 itk::Mesh, 70
 ShapedNeighborhoodIterator, 176
 ActivateOffset(), 176
 ClearActiveList(), 176
 DeactivateOffset(), 176
 examples of, 177
 GetActiveIndexListSize(), 176
 Iterator::Begin(), 177
 Iterator::End(), 177
 smart pointer, 26
 Sobel operator, 165, 168
 source, 31, 193
 spatial object, 32
 streaming, 31
 template, 24
 TetrahedronCell
 GetNumberOfFaces(), 95
 VNL, 29
 wrapping, 33

The ITK Software Guide
Book 2: Design and Functionality
Fourth Edition
Updated for ITK version 4.10

Hans J. Johnson, Matthew M. McCormick, Luis Ibáñez,
and the *Insight Software Consortium*

May 27, 2016

<https://itk.org>
Email: community@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 a cross-platform software. It uses a build environment known as [CMake](#) to manage platform-specific project generation and compilation process in a platform-independent way. ITK is implemented in C++. ITK's implementation style employs generic programming, which involves the use of templates to generate, at compile-time, code that can be applied *generically* to any class or data-type that supports the operations used by the template. The use of C++ templating means that the code is highly efficient and many issues are discovered at compile-time, rather than at run-time during program execution. It also means that many of ITK's algorithms can be applied to arbitrary spatial dimensions and pixel types.

An automated wrapping system integrated with ITK generates an interface between C++ and a high-level programming language [Python](#). This enables rapid prototyping and faster exploration of ideas by shortening the edit-compile-execute cycle. In addition to automated wrapping, the [SimpleITK](#) project provides a streamlined interface to ITK that is available for C++, Python, Java, CSharp, R, Tcl and Ruby.

Developers from around the world can use, debug, maintain, and extend the software because ITK is an open-source project. ITK uses a model of software development known as Extreme Programming. Extreme Programming collapses the usual software development methodology into a simultaneous 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. An extensive testing process supported by the system known as [CDash](#) measures the quality of ITK code on a daily basis. The ITK Testing Dashboard is updated continuously, reflecting the quality of

the code at any moment.

The most recent version of this document is available online at <https://itk.org/ItkSoftwareGuide.pdf>. This book is a guide to developing software with ITK; it is the second of two companion books. This book covers detailed design and functionality for reading and writing images, filtering, registration, segmentation, and performing statistical analysis. The first book covers building and installation, general architecture and design, as well as the process of contributing in the ITK community.

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 <https://itk.org/ITK/project/parti.html> for the names of additional contributors, as well as checking the GIT 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 L^AT_EX 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.

Since the software guide is generated in part from the ITK source code itself, many ITK developers have been involved in updating and extending the ITK documentation. These include **David Doria**, **Bradley Lowekamp**, **Mark Foskey**, **Gaëtan Lehmann**, **Andreas Schuh**, **Tom Vercauteren**, **Cory Quammen**, **Daniel Blezek**, **Paul Huggett**, **Matthew McCormick**, **Josh Cates**, **Arnaud Gelas**, **Jim Miller**, **Brad King**, **Gabe Hart**, **Hans Johnson**.

Hans Johnson, **Kent Williams**, **Constantine Zakkaroff**, **Xiaoxiao Liu**, **Ali Ghayoor**, and **Matthew McCormick** updated the documentation for the initial ITK Version 4 release.

Luis Ibáñez and **Sébastien Barré** designed the original Book 1 cover. **Matthew McCormick** and **Brad King** updated the code to produce the Book 1 cover for ITK 4 and VTK 6. **Xiaoxiao Liu**, **Bill Lorensen**, **Luis Ibáñez**, and **Matthew McCormick** created the 3D printed anatomical objects that were photographed by **Sébastien Barré** for the Book 2 cover. **Steve Jordan** designed the layout of the covers.

Lisa Avila, **Hans Johnson**, **Matthew McCormick**, **Sandy McKenzie**, **Christopher Mullins**, **Katie Osterdahl**, and **Michka Popoff** prepared the book for the 4.7 print release.

CONTENTS

1	Reading and Writing Images	1
1.1	Basic Example	1
1.2	Pluggable Factories	5
1.3	Using ImageIO Classes Explicitly	5
1.4	Reading and Writing RGB Images	7
1.5	Reading, Casting and Writing Images	8
1.6	Extracting Regions	10
1.7	Extracting Slices	12
1.8	Reading and Writing Vector Images	14
1.8.1	The Minimal Example	14
1.8.2	Producing and Writing Covariant Images	16
1.8.3	Reading Covariant Images	18
1.9	Reading and Writing Complex Images	20
1.10	Extracting Components from Vector Images	21
1.11	Reading and Writing Image Series	23
1.11.1	Reading Image Series	24
1.11.2	Writing Image Series	25
1.11.3	Reading and Writing Series of RGB Images	27
1.12	Reading and Writing DICOM Images	30
1.12.1	Foreword	30

1.12.2	Reading and Writing a 2D Image	31
1.12.3	Reading a 2D DICOM Series and Writing a Volume	34
1.12.4	Reading a 2D DICOM Series and Writing a 2D DICOM Series	38
1.12.5	Printing DICOM Tags From One Slice	41
1.12.6	Printing DICOM Tags From a Series	44
1.12.7	Changing a DICOM Header	47
2	Filtering	51
2.1	Thresholding	51
2.1.1	Binary Thresholding	51
2.1.2	General Thresholding	54
2.2	Edge Detection	57
2.2.1	Canny Edge Detection	57
2.3	Casting and Intensity Mapping	58
2.3.1	Linear Mappings	59
2.3.2	Non Linear Mappings	61
2.4	Gradients	64
2.4.1	Gradient Magnitude	64
2.4.2	Gradient Magnitude With Smoothing	66
2.4.3	Derivative Without Smoothing	68
2.5	Second Order Derivatives	69
2.5.1	Second Order Recursive Gaussian	69
2.5.2	Laplacian Filters	74
	Laplacian Filter Recursive Gaussian	74
2.6	Neighborhood Filters	79
2.6.1	Mean Filter	79
2.6.2	Median Filter	81
2.6.3	Mathematical Morphology	82
	Binary Filters	83
	Grayscale Filters	86
2.6.4	Voting Filters	88
	Binary Median Filter	88

Hole Filling Filter	90
Iterative Hole Filling Filter	94
2.7 Smoothing Filters	95
2.7.1 Blurring	97
Discrete Gaussian	97
Binomial Blurring	99
Recursive Gaussian IIR	100
2.7.2 Local Blurring	103
Gaussian Blur Image Function	104
2.7.3 Edge Preserving Smoothing	104
Introduction to Anisotropic Diffusion	104
Gradient Anisotropic Diffusion	105
Curvature Anisotropic Diffusion	107
Curvature Flow	110
MinMaxCurvature Flow	112
Bilateral Filter	115
2.7.4 Edge Preserving Smoothing in Vector Images	117
Vector Gradient Anisotropic Diffusion	118
Vector Curvature Anisotropic Diffusion	119
2.7.5 Edge Preserving Smoothing in Color Images	122
Gradient Anisotropic Diffusion	122
Curvature Anisotropic Diffusion	123
2.8 Distance Map	125
2.9 Geometric Transformations	130
2.9.1 Filters You Should be Afraid to Use	130
2.9.2 Change Information Image Filter	130
2.9.3 Flip Image Filter	130
2.9.4 Resample Image Filter	131
Introduction	131
Importance of Spacing and Origin	137
A Complete Example	143

Rotating an Image	147
Rotating and Scaling an Image	149
Resampling using a deformation field	151
Subsampling and image in the same space	152
Resampling an Anisotropic image to make it Isotropic	155
2.10 Frequency Domain	160
2.10.1 Computing a Fast Fourier Transform (FFT)	160
2.10.2 Filtering on the Frequency Domain	163
2.11 Extracting Surfaces	166
2.11.1 Surface extraction	166
3 Registration	169
3.1 Registration Framework	169
3.2 "Hello World" Registration	171
3.3 Features of the Registration Framework	181
3.4 Monitoring Registration	184
3.5 Multi-Modality Registration	189
3.5.1 Mattes Mutual Information	189
3.6 Centered Transforms	196
3.6.1 Rigid Registration in 2D	196
3.6.2 Initializing with Image Moments	203
3.6.3 Similarity Transform in 2D	210
3.6.4 Rigid Transform in 3D	214
3.6.5 Centered Affine Transform	219
3.7 Multi-Resolution Registration	222
3.7.1 Fundamentals	224
3.8 Multi-Stage Registration	230
3.8.1 Fundamentals	231
3.8.2 Cascaded Multistage Registration	239
3.9 Transforms	244
3.9.1 Geometrical Representation	244
3.9.2 Transform General Properties	247

3.9.3	Identity Transform	248
3.9.4	Translation Transform	248
3.9.5	Scale Transform	249
3.9.6	Scale Logarithmic Transform	251
3.9.7	Euler2DTransform	251
3.9.8	CenteredRigid2DTransform	252
3.9.9	Similarity2DTransform	254
3.9.10	QuaternionRigidTransform	255
3.9.11	VersorTransform	256
3.9.12	VersorRigid3DTransform	256
3.9.13	Euler3DTransform	257
3.9.14	Similarity3DTransform	258
3.9.15	Rigid3DPerspectiveTransform	259
3.9.16	AffineTransform	259
3.9.17	BSplineDeformableTransform	261
3.9.18	KernelTransforms	262
3.10	Interpolators	264
3.10.1	Nearest Neighbor Interpolation	265
3.10.2	Linear Interpolation	265
3.10.3	B-Spline Interpolation	265
3.10.4	Windowed Sinc Interpolation	266
3.11	Metrics	269
3.11.1	Mean Squares Metric	271
Exploring a Metric	271	
3.11.2	Normalized Correlation Metric	274
3.11.3	Mutual Information Metric	274
Parzen Windowing	275	
Mattes et al. Implementation	276	
3.11.4	Normalized Mutual Information Metric	276
3.11.5	Demons metric	276
3.11.6	ANTS neighborhood correlation metric	277

3.12 Optimizers	278
3.12.1 Registration using the One plus One Evolutionary Optimizer	280
3.12.2 Registration using masks constructed with Spatial objects	282
3.12.3 Rigid registrations incorporating prior knowledge	284
3.13 Deformable Registration	287
3.13.1 FEM-Based Image Registration	287
3.13.2 BSplines Image Registration	291
3.13.3 Level Set Motion for Deformable Registration	294
3.13.4 BSplines Multi-Grid Image Registration	297
3.13.5 BSplines Multi-Grid Image Registration in 3D	300
3.13.6 Image Warping with Kernel Splines	302
3.13.7 Image Warping with BSplines	303
3.14 Demons Deformable Registration	307
3.14.1 Asymmetrical Demons Deformable Registration	308
3.14.2 Symmetrical Demons Deformable Registration	311
3.15 Visualizing Deformation fields	315
3.15.1 Visualizing 2D deformation fields	315
3.15.2 Visualizing 3D deformation fields	316
3.16 Model Based Registration	321
3.17 Point Set Registration	331
3.17.1 Point Set Registration in 2D	331
3.17.2 Point Set Registration in 3D	335
3.17.3 Point Set to Distance Map Metric	337
3.18 Registration Troubleshooting	339
3.18.1 Too many samples outside moving image buffer	339
3.18.2 General heuristics for parameter fine-tuning	339
4 Segmentation	343
4.1 Region Growing	343
4.1.1 Connected Threshold	344
4.1.2 Otsu Segmentation	347
4.1.3 Neighborhood Connected	351

4.1.4	Confidence Connected	354
	Application of the Confidence Connected filter on the Brain Web Data	357
4.1.5	Isolated Connected	359
4.1.6	Confidence Connected in Vector Images	361
4.2	Segmentation Based on Watersheds	364
4.2.1	Overview	364
4.2.2	Using the ITK Watershed Filter	366
4.3	Level Set Segmentation	370
4.3.1	Fast Marching Segmentation	372
4.3.2	Shape Detection Segmentation	379
4.3.3	Geodesic Active Contours Segmentation	387
4.3.4	Threshold Level Set Segmentation	390
4.3.5	Canny-Edge Level Set Segmentation	395
4.3.6	Laplacian Level Set Segmentation	399
4.3.7	Geodesic Active Contours Segmentation With Shape Guidance	402
4.4	Feature Extraction	413
4.4.1	Hough Transform	413
	Line Extraction	413
	Circle Extraction	417
5	Statistics	421
5.1	Data Containers	421
5.1.1	Sample Interface	422
5.1.2	Sample Adaptors	424
	ImageToListSampleAdaptor	424
	PointSetToListSampleAdaptor	426
5.1.3	Histogram	429
5.1.4	Subsample	432
5.1.5	MembershipSample	435
5.1.6	MembershipSampleGenerator	438
5.1.7	K-d Tree	440
5.2	Algorithms and Functions	445

5.2.1	Sample Statistics	446
Mean and Covariance	446	
Weighted Mean and Covariance	448	
5.2.2	Sample Generation	451
SampleToHistogramFilter	451	
NeighborhoodSampler	453	
5.2.3	Sample Sorting	455
5.2.4	Probability Density Functions	457
Gaussian Distribution	458	
5.2.5	Distance Metric	459
Euclidean Distance	459	
5.2.6	Decision Rules	460
Maximum Decision Rule	461	
Minimum Decision Rule	461	
Maximum Ratio Decision Rule	462	
5.2.7	Random Variable Generation	463
Normal (Gaussian) Distribution	463	
5.3	Statistics applied to Images	464
5.3.1	Image Histograms	464
Scalar Image Histogram with Adaptor	464	
Scalar Image Histogram with Generator	466	
Color Image Histogram with Generator	468	
Color Image Histogram Writing	472	
5.3.2	Image Information Theory	475
Computing Image Entropy	475	
Computing Images Mutual Information	479	
5.4	Classification	484
5.4.1	k-d Tree Based k-Means Clustering	485
5.4.2	K-Means Classification	491
5.4.3	Bayesian Plug-In Classifier	494
5.4.4	Expectation Maximization Mixture Model Estimation	500

5.4.5 Classification using Markov Random Field	504
--	-----

LIST OF FIGURES

1.1	Collaboration diagram of the ImageIO classes	3
1.2	Use cases of ImageIO factories	4
1.3	Class diagram of ImageIO factories	4
2.1	BinaryThresholdImageFilter transfer function	52
2.2	BinaryThresholdImageFilter output	53
2.3	ThresholdImageFilter using the threshold-below mode.	54
2.4	ThresholdImageFilter using the threshold-above mode	54
2.5	ThresholdImageFilter using the threshold-outside mode	55
2.6	Sigmoid Parameters	62
2.7	Effect of the Sigmoid filter.	63
2.8	GradientMagnitudeImageFilter output	66
2.9	GradientMagnitudeRecursiveGaussianImageFilter output	68
2.10	Effect of the Derivative filter.	70
2.11	Output of the LaplacianRecursiveGaussianImageFilter.	77
2.12	Effect of the MedianImageFilter	81
2.13	Effect of the Median filter.	83
2.14	Effect of erosion and dilation in a binary image.	85
2.15	Effect of erosion and dilation in a grayscale image.	88
2.16	Effect of the BinaryMedian filter.	90

2.17 Effect of many iterations on the BinaryMedian filter.	91
2.18 Effect of the VotingBinaryHoleFilling filter.	93
2.19 Effect of the VotingBinaryIterativeHoleFilling filter.	96
2.20 DiscreteGaussianImageFilter Gaussian diagram.	97
2.21 DiscreteGaussianImageFilter output	99
2.22 BinomialBlurImageFilter output.	100
2.23 Output of the SmoothingRecursiveGaussianImageFilter.	103
2.24 GradientAnisotropicDiffusionImageFilter output	107
2.25 CurvatureAnisotropicDiffusionImageFilter output	109
2.26 CurvatureFlowImageFilter output	112
2.27 MinMaxCurvatureFlow computation	113
2.28 MinMaxCurvatureFlowImageFilter output	115
2.29 BilateralImageFilter output	118
2.30 VectorGradientAnisotropicDiffusionImageFilter output	120
2.31 VectorCurvatureAnisotropicDiffusionImageFilter output	121
2.32 VectorGradientAnisotropicDiffusionImageFilter on RGB	124
2.33 VectorCurvatureAnisotropicDiffusionImageFilter output on RGB	126
2.34 Various Anisotropic Diffusion compared	126
2.35 DanielssonDistanceMapImageFilter output	128
2.36 SignedDanielssonDistanceMapImageFilter output	130
2.37 Effect of the FlipImageFilter	132
2.38 Effect of the Resample filter	134
2.39 Analysis of resampling in common coordinate system	135
2.40 ResampleImageFilter with a translation by $(-30, -50)$	136
2.41 ResampleImageFilter. Analysis of a translation by $(-30, -50)$	136
2.42 ResampleImageFilter highlighting image borders	137
2.43 ResampleImageFilter selecting the origin of the output image	139
2.44 ResampleImageFilter origin in the output image	140
2.45 ResampleImageFilter selecting the origin of the input image	140
2.46 ResampleImageFilter use of naive viewers	141
2.47 ResampleImageFilter and output image spacing	142

2.48 ResampleImageFilter naive viewers	143
2.49 ResampleImageFilter with non-unit spacing	144
2.50 Effect of a rotation on the resampling filter.	145
2.51 Input and output image placed in a common reference system	145
2.52 Effect of the Resample filter rotating an image	149
2.53 Effect of the Resample filter rotating and scaling an image	151
3.1 Image Registration Concept	169
3.2 A Typical Registration Framework Components	170
3.3 Registration Framework Components	170
3.4 Fixed and Moving images in registration framework	177
3.5 HelloWorld registration output images	178
3.6 Pipeline structure of the registration example	179
3.7 Trace of translations and metrics during registration	181
3.8 Registration Coordinate Systems	182
3.9 Command/Observer and the Registration Framework	187
3.10 Multi-Modality Registration Inputs	192
3.11 MattesMutualInformationImageToImageMetricv4 output images	193
3.12 MattesMutualInformationImageToImageMetricv4 output plots	194
3.13 MattesMutualInformationImageToImageMetricv4 number of bins	195
3.14 Rigid2D Registration input images	201
3.15 Rigid2D Registration output images	201
3.16 Rigid2D Registration output plots	202
3.17 Rigid2D Registration input images	203
3.18 Rigid2D Registration output images	204
3.19 Rigid2D Registration output plots	204
3.20 Effect of changing the center of rotation	208
3.21 CenteredTransformInitializer input images	209
3.22 CenteredTransformInitializer output images	209
3.23 CenteredTransformInitializer output plots	210
3.24 Fixed and Moving image registered with CenteredSimilarity2DTransform	213
3.25 Output of the CenteredSimilarity2DTransform registration	213

3.26 CenteredSimilarity2DTransform registration plots	214
3.27 CenteredTransformInitializer input images	218
3.28 CenteredTransformInitializer output images	218
3.29 CenteredTransformInitializer output plots	219
3.30 AffineTransform registration	223
3.31 AffineTransform output images	223
3.32 AffineTransform output plots	224
3.33 Conceptual representation of Multi-Resolution registration	225
3.34 Multi-Resolution registration input images	229
3.35 Multi-Resolution registration output images	230
3.36 AffineTransform registration	238
3.37 Multistage registration input images	239
3.38 Multistage registration input images	243
3.39 Geometrical representation objects in ITK	244
3.40 Mapping moving image to fixed image in Registration	264
3.41 Need for interpolation in Registration	264
3.42 BSpline Interpolation Concepts	266
3.43 Parzen Windowing in Mutual Information	269
3.44 Mean Squares Metric Plots	273
3.45 Class diagram of the Optimizers hierarchy in ITKv4	279
3.46 FEM-based deformable registration results	287
3.47 Demon's deformable registration output	297
3.48 Demon's deformable registration output	311
3.49 Demon's deformable registration output	315
3.50 Deformation field magnitudes	316
3.51 Calculator	317
3.52 Visualized Def field	317
3.53 Visualized Def field4	318
3.54 Deformation field output	320
3.55 Difference image	320
3.56 Model to Image Registration Framework Components	321

3.57 Model to Image Registration Framework Concept	322
3.58 SpatialObject to Image Registration results	332
4.1 ConnectedThreshold segmentation results	346
4.2 OtsuThresholdImageFilter output	349
4.3 NeighborhoodConnected segmentation results	353
4.4 ConfidenceConnected segmentation results	357
4.5 Whitematter Confidence Connected segmentation.	358
4.6 Axial, sagittal, and coronal slice of Confidence Connected segmentation.	358
4.7 IsolatedConnected segmentation results	361
4.8 VectorConfidenceConnected segmentation results	363
4.9 Watershed Catchment Basins	365
4.10 Watersheds Hierarchy of Regions	365
4.11 Watersheds filter composition	366
4.12 Watershed segmentation output	369
4.13 Zero Set Concept	370
4.14 Grid position of the embedded level-set surface.	371
4.15 FastMarchingImageFilter collaboration diagram	372
4.16 FastMarchingImageFilter intermediate output	378
4.17 FastMarchingImageFilter segmentations	379
4.18 ShapeDetectionLevelSetImageFilter collaboration diagram	380
4.19 ShapeDetectionLevelSetImageFilter intermediate output	386
4.20 ShapeDetectionLevelSetImageFilter segmentations	387
4.21 GeodesicActiveContourLevelSetImageFilter collaboration diagram	388
4.22 GeodesicActiveContourLevelSetImageFilter intermediate output	391
4.23 GeodesicActiveContourImageFilter segmentations	392
4.24 ThresholdSegmentationLevelSetImageFilter collaboration diagram	393
4.25 Propagation term for threshold-based level set segmentation	393
4.26 ThresholdSegmentationLevelSet segmentations	395
4.27 CannySegmentationLevelSetImageFilter collaboration diagram	397
4.28 Segmentation results of CannyLevelSetImageFilter	399
4.29 LaplacianSegmentationLevelSetImageFilter collaboration diagram	400

4.30 Segmentation results of LaplacianLevelSetImageFilter	402
4.31 GeodesicActiveContourShapePriorLevelSetImageFilter collaboration diagram	404
4.32 GeodesicActiveContourShapePriorImageFilter input image and initial model	412
4.33 Corpus callosum PCA modes	412
4.34 GeodesicActiveContourShapePriorImageFilter segmentations	413
5.1 Sample class inheritance tree	421
5.2 Histogram	430
5.3 Simple conceptual classifier	484
5.4 Statistical classification framework	485
5.5 Two normal distributions plot	487
5.6 Output of the KMeans classifier	494
5.7 Bayesian plug-in classifier for two Gaussian classes	495
5.8 Output of the ScalarImageMarkovRandomField	509

LIST OF TABLES

3.1	Geometrical Elementary Objects	245
3.2	Identity Transform Characteristics	248
3.3	Translation Transform Characteristics	249
3.4	Scale Transform Characteristics	250
3.5	Scale Logarithmic Transform Characteristics	251
3.6	Euler2D Transform Characteristics	252
3.7	CenteredRigid2D Transform Characteristics	253
3.8	Similarity2D Transform Characteristics	254
3.9	QuaternionRigid Transform Characteristics	255
3.10	Vesor Transform Characteristics	256
3.11	Vesor Rigid3D Transform Characteristics	257
3.12	Euler3D Transform Characteristics	258
3.13	Similarity3D Transform Characteristics	259
3.14	Rigid3DPerspective Transform Characteristics	260
3.15	Affine Transform Characteristics	260
3.16	BSpline Deformable Transform Characteristics	262
3.17	LBFGS Optimizer trace	319
4.1	ConnectedThreshold example parameters	346
4.2	IsolatedConnectedImageFilter example parameters	360

4.3	FastMarching segmentation example parameters	379
4.4	ShapeDetection example parameters	387
4.5	GeodesicActiveContour segmentation example parameters	390
4.6	ThresholdSegmentationLevelSet segmentation parameters	396

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.

1.1 Basic Example

The source code for this section can be found in the file `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 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 intend to apply on the images.

A typical selection for medical images is illustrated in the following lines.

```
typedef short      PixelType;
const  unsigned int Dimension = 2;
typedef itk::Image<PixelType, Dimension>    ImageType;
```

Note that the dimension of the image in memory should match that of the image in the 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 assign 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 to 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 glance this may look like 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.

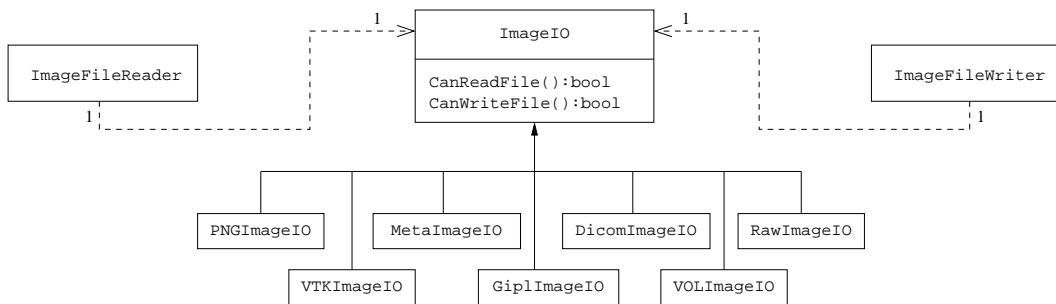


Figure 1.1: Collaboration diagram of the ImageIO classes.

```

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 in 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 Image.FileWriter to determine (at run-time) 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 of the appropriate `itk::ImageIO` subclass.

For historical reasons and as a convenience to the user, the `itk::Image.FileWriter` 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 1.1, 1.2, and 1.3.

The following section describes the internals of the IO architecture provided in the toolkit.

¹In this example no file format is specified; this program can be used as a general file conversion utility.

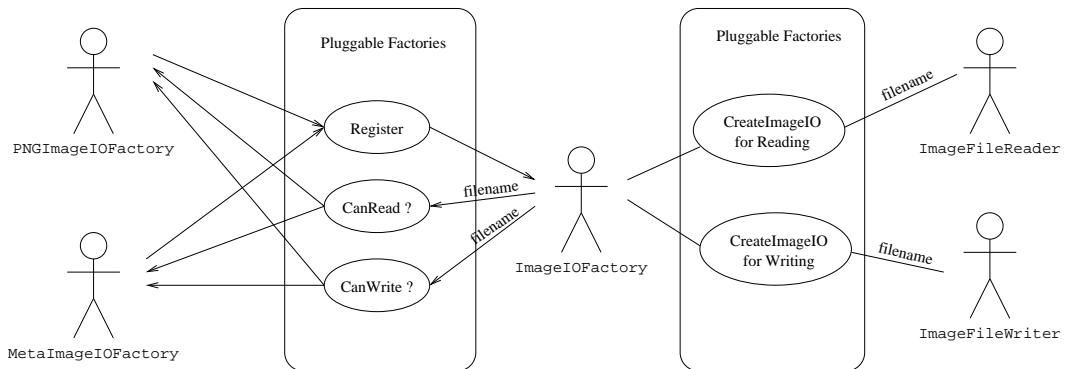


Figure 1.2: Use cases of ImageIO factories.

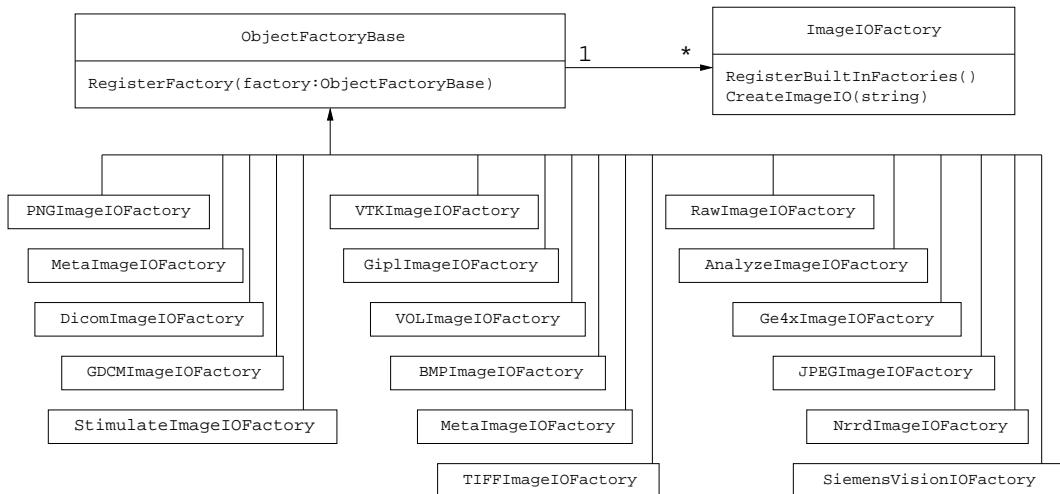


Figure 1.3: Class diagram of the ImageIO factories.

1.2 Pluggable Factories

The principle behind the input/output mechanism used in ITK is known as *pluggable-factories* [21]. This concept is illustrated in the UML diagram in Figure 1.1. 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 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 1.2. The ImageIOFactory acts here as a dispatcher that helps locate the actual IO factory classes corresponding to each file format.

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 ObjectFactoryBase class as illustrated in Figure 1.3.

For example, in order to read PNG files, a `PNGImageIOFactory` is created and registered with the central `ImageIOFactory` singleton² class as illustrated in the left side of Figure 1.2. 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 if 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 `ImageFileReader` and `ImageFileWriter`. It is possible, however, to explicitly select the type of `ImageIO` object to use. This is illustrated by the following example.

1.3 Using ImageIO Classes Explicitly

The source code for this section can be found in the file
`ImageReadExportVTK.cxx`.

²Singleton 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 provided by ImageIO.

The following example illustrates explicit instantiation 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 files 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](#).

1.4 Reading and Writing RGB Images

The source code for this section can be found in the file `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 specific to RGB images. All the actions required to support color images are implemented internally in the `itk::ImageIO` objects.

1.5 Reading, Casting and Writing Images

The source code for this section can be found in the file `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 as 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 the 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;
}

```

1.6 Extracting Regions

The source code for this section can be found in the file `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 an 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 assign 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 );
```

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;
}

```

1.7 Extracting Slices

The source code for this section can be found in the file `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 an 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 assign 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 `itk::SmartPointer`.

```
typedef itk::ExtractImageFilter< InputImageType,
                                OutputImageType > FilterType;
FilterType::Pointer filter = FilterType::New();
filter->InPlaceOn();
filter->SetDirectionCollapseToSubmatrix();
```

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` indicating 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 `UpdateOutputInformation()` is being called first on the reader. This method updates the metadata in the output image without actually reading in the bulk-data.

```
reader->UpdateOutputInformation();
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 is 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;
}
```

1.8 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.

1.8.1 The Minimal Example

The source code for this section can be found in the file `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();
```

A 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] );
```

This being 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.

1.8.2 Producing and Writing Covariant Images

The source code for this section can be found in the file `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::CovariantVectors` as opposed to Vectors. In this way, gradients are transformed correctly under `itk::AffineTransforms` 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 read an image of signed short pixels and compute the gradient to produce an image of `CovariantVectors` 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 σ for this filter is specified in millimeters.

```
filter->SetSigma( 1.5 );      // Sigma in millimeters
```

Below, we create the reader and writer using the `New()` method and assign 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 to 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;
}
```

1.8.3 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
CovariantVectorImageRead.cxx.

This example illustrates how to read an image whose pixel type is `itk::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 a gradient image from a file (written in the previous example) and computing its magnitude using the `itk::VectorMagnitudeImageFilter`. Note that this filter is different from the `itk::GradientMagnitudeImageFilter` which actually takes a scalar image as input and computes the magnitude of its gradient. The `VectorMagnitudeImageFilter` 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 "itkVectorMagnitudeImageFilter.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 `VectorMagnitudeImageFilter` 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::VectorMagnitudeImageFilter<
    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 us to define a number of type-related constants in a generic way. The use of traits is a fundamental characteristic of generic programming [5, 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 `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 );

```

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;
}
```

1.9 Reading and Writing Complex Images

The source code for this section can be found in the file `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 [4].

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();
```

File names should be provided for both the reader and the writer. In this particular example we take those file names 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 file format 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 [2.10](#) in the context of Fourier analysis.

1.10 Extracting Components from Vector Images

The source code for this section can be found in the file `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 one 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 an 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 requires 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 us to define a number of type-related constants in a generic way. The use of traits is a fundamental characteristic of generic programming [5, 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 `itk::SmartPointer`.

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

The name of the file to be read or written is passed to 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;
}
```

1.11 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.

1.11.1 Reading Image Series

The source code for this section can be found in the file `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 a 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 filename generator type and create one instance of it.

```
typedef itk::NumericSeriesFileNames    NameGeneratorType;

NameGeneratorType::Pointer nameGenerator = NameGeneratorType::New();
```

The filename 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;  
}
```

1.11.2 Writing Image Series

The source code for this section can be found in the file `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 ar-

guments 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. These numerical values are then passed to the filename 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 filename 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.

1.11.3 Reading and Writing Series of RGB Images

The source code for this section can be found in the file `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 we will 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 "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 ImageSeries-Reader.

```
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 filename 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 filename 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 to RGB images. All the actions required to support color images are implemented

internally in the `itk::ImageIO` objects.

1.12 Reading and Writing DICOM Images

1.12.1 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 [44].

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 in a clinical setting are stored and transported using the DICOM standard.

³<http://www.acr.org>

⁴<http://www.nema.org>

DICOM functionalities in ITK are provided by the GDCM library. This open source library was developed by the CREATIS Team⁵ at INSA-Lyon [7]. Although originally this library was distributed under a LGPL License⁶, the CREATIS Team was lucid enough to understand the limitations of that license and agreed to adopt the more open BSD-like License⁷. This change in their licensing made possible to distribute GDCM along with ITK.

GDCM is now maintained by Mathieu Malaterre and the GDCM community. The version distributed with ITK gets updated with major releases of the GDCM library.

1.12.2 Reading and Writing a 2D Image

The source code for this section can be found in the file `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 the `itk::GDCMImageIO` class which 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] );
```

`GDCMImageIO` is an `ImageIO` class for reading and writing DICOM v3 and ACR/NEMA images.

⁵<http://www.creatis.insa-lyon.fr>

⁶<http://www.gnu.org/copyleft/lesser.html>

⁷<http://www.opensource.org/licenses/bsd-license.php>

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 now have the image in memory and can get access to it using the `GetOutput()` method of the reader. In the remainder of this current example, we focus on showing how to 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. Since in this example we write the image in different ways, and in each case use a different writer, we enumerated 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 using the RescaleIntensityImageFilter. 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 new file, which is 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 in 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;
}
```

1.12.3 Reading a 2D DICOM Series and Writing a Volume

The source code for this section can be found in the file `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 3-dimensional 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 stored in 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 "itkImage.h"
#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.

The image orientation information contained in the direction cosines of the DICOM header are read in and passed correctly down the image processing pipeline.

```
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: 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, 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 GDCMSeriesFileNames 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(). By default SetUseSeriesDetails(true) will use the following DICOM tags to sub-refine a set of files into multiple series:

0020 0011 Series Number

0018 0024 Sequence Name

0018 0050 Slice Thickness

0028 0010 Rows

0028 0011 Columns

If this is not enough for your specific case you can always add some more restrictions using the `AddSeriesRestriction()` method. In this example we will use the DICOM Tag: 0008 0021 DA 1 Series Date, to sub-refine each series. The format for passing the argument is a string containing first the group then the element of the DICOM tag, separated by a pipe (|) sign.

```
typedef itk::GDCMSeriesFileNames NamesGeneratorType;
NamesGeneratorType::Pointer nameGenerator = NamesGeneratorType::New();

nameGenerator->SetUseSeriesDetails( true );
nameGenerator->AddSeriesRestriction("0008|0021");

nameGenerator->SetDirectory( argv[1] );
```

The `GDCMSeriesFileNames` object first identifies the list of DICOM series present in the given directory. We receive that list in a reference to a container of strings and then we can do things like print 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 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.

1.12.4 Reading a 2D DICOM Series and Writing a 2D DICOM Series

The source code for this section can be found in the file `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 norms⁸. You would probably find similar legislation in every country.

When saving datasets in DICOM format it must be made clear whether these 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 from a 16-bits/pixel to a 8-bits/pixel representation may make it impossible to detect certain pathologies and as a result will expose the patient to the risk of 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 [32]. 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 `GDCMImageIO` class, the GDCM filenames generator, as well as the series reader and writer.

```
#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.

⁸The Health Insurance Portability and Accountability Act of 1996. <http://www.cms.hhs.gov/hipaa/>

```
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 used 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 have the volumetric data loaded in memory and we can access 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 exists, using the cross-platform tools: `itksys::SystemTools`. In this case we choose to create the directory if it does not exist yet.

```
itksys::SystemTools::MakeDirectory( outputDirectory );
```

We explicitly instantiate 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. This step is important because 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 generating DICOM files which have the appearance of being produced by a scanner. It should be clear from the directory or filenames that these data were the result of the execution of some sort of algorithm. This will prevent your dataset from being used as scanner data by accident.

1.12.5 Printing DICOM Tags From One Slice

The source code for this section can be found in the file `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 consistency checking, 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 `GDCMImageIO` object, the `MetaDataDictionary` and its entry element, the `MetaDataObject`.

```
#include "itkImageFileReader.h"
#include "itkGDCMImageIO.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 a 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 `MetaDataDictionary` 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 with the tag that we are providing in the `tagkey` variable. If the label is found, it is returned in `labelId` variable. The method itself returns 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 succeeds, 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.c_str() << std::endl;
}
```

Another way to read a specific tag is to use the encapsulation above MetaDataDictionary. Note that this is strictly equivalent to the above code.

```

std::string tagkey = "0008|1050";
std::string labelId;
if( itk::GDCMImageIO::GetLabelFromTag( tagkey, labelId ) )
{
    std::string value;
    std::cout << labelId << " (" << tagkey << "): ";
    if( dicomIO->GetValueFromTag(tagkey, value) )
    {
        std::cout << value;
    }
    else
    {
        std::cout << "(No Value Found in File)";
    }
    std::cout << std::endl;
}
else
{
    std::cerr << "Trying to access inexistant DICOM tag." << std::endl;
}

```

For a full description of the DICOM dictionary please look at the file.

`Insight/Utilities/gdcm/Dicts/dicomV3.dic`

The following piece of code will print out the proper pixel type / component for instantiating an `itk::ImageFileReader` that can properly import the printed DICOM file.

```

itk::ImageIOBase::IOPixelType pixelType
    = reader->GetImageIO()->GetPixelType();
itk::ImageIOBase::IOComponentType componentType
    = reader->GetImageIO()->GetComponentType();
std::cout << "PixelType: " << reader->GetImageIO()
    ->GetPixelTypeAsString(pixelType) << std::endl;
std::cout << "Component Type: " << reader->GetImageIO()
    ->GetComponentTypeAsString(componentType) << std::endl;

```

1.12.6 Printing DICOM Tags From a Series

The source code for this section can be found in the file
`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"

```

Next, 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 = 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 obtains all the DICOM tags from the file headers. The tag values are stored in the `itk::MetaDataDictionary` which is a general-purpose container for `{key,value}` pairs. The Metadata dictionary can be recovered from any ImageIO class by invok-

ing 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. Therefore, we declare a `itk::MetaDataObject` of string type in order to receive those particular values.

```
typedef itk::MetaDataObject< std::string > MetaDataStringType;
```

The metadata 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 into a string entry by using the `dynamic_cast` based on RTTI information⁹. The dictionary is organized like a `std::map` structure, so we should use 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;
}
```

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 in the dictionary.

⁹Run Time Type Information

```
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;
    return EXIT_FAILURE;
}
```

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, we can then 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;
}
else
{
    std::cerr << "Entry was not of string type" << std::endl;
    return EXIT_FAILURE;
}
```

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 following file.

`Insight/Utilities/gdcm/Dicts/dicomV3.dic`

1.12.7 Changing a DICOM Header

The source code for this section can be found in the file `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 being delivered to the patient.

We must start by including the relevant header files. Here we include the image reader, image writer, the image, the metadata dictionary and its entries, the metadata objects and the GDCMImageIO. The metadata 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 "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 metadata 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 metadata 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 `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 with 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
TWO

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 the Data Processing Pipeline section in Book 1 for more information.)

2.1 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.

2.1.1 Binary Thresholding

The source code for this section can be found in the file `BinaryThresholdImageFilter.cxx`.

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 2.1. 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"
```

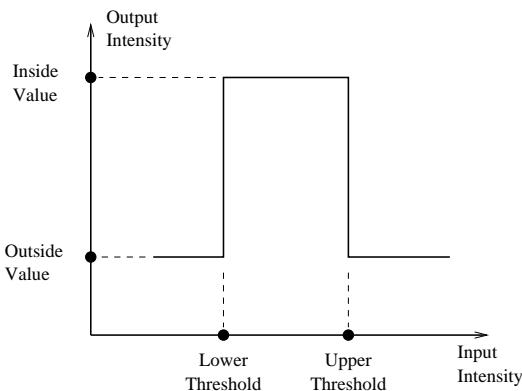


Figure 2.1: Transfer function of the `BinaryThresholdImageFilter`.

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 1 on page 1 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< OutputImageType > 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`.

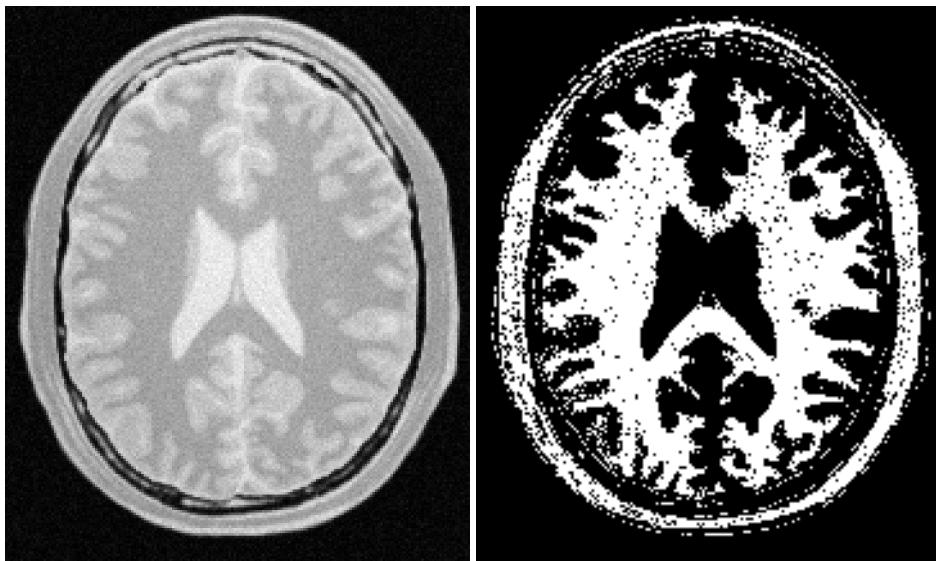


Figure 2.2: Effect of the `BinaryThresholdImageFilter` on a slice from a MRI proton density image of the brain.

```
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 downstream filters in the pipeline will indirectly trigger the update of this filter.

```
filter->Update();
```

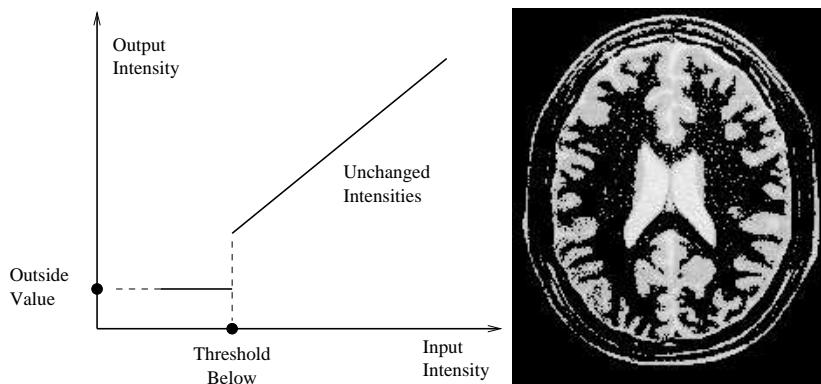


Figure 2.3: ThresholdImageFilter using the threshold-below mode.

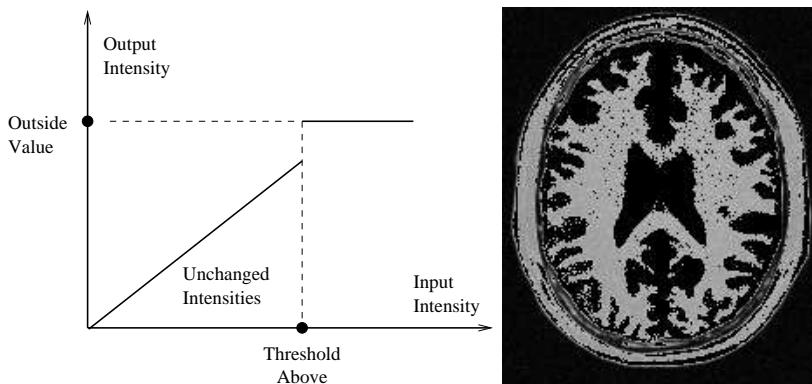


Figure 2.4: ThresholdImageFilter using the threshold-above mode.

Figure 2.2 illustrates the effect of this filter on a MRI proton density image of the brain. This figure shows the limitations of the 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](#)

2.1.2 General Thresholding

The source code for this section can be found in the file `ThresholdImageFilter.cxx`.

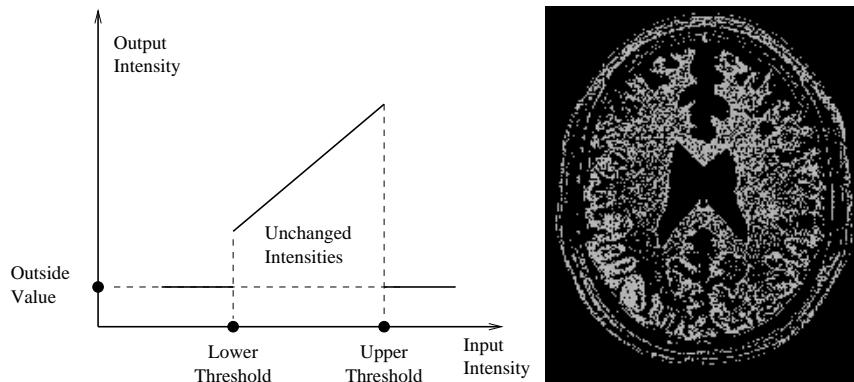


Figure 2.5: `ThresholdImageFilter` using the threshold-outside mode.

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 2.3.
- 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 2.4.
- 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 2.5.

The following methods choose among the three operating modes of the filter.

- `ThresholdBelow()`
- `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 2.3. 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 2.4. 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 2.5.

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`

2.2 Edge Detection

2.2.1 Canny Edge Detection

The source code for this section can be found in the file `CannyEdgeDetectionImageFilter.cxx`.

This example introduces the use of the `itk::CannyEdgeDetectionImageFilter`. Canny edge detection is widely used for edge detection since it is the optimal solution satisfying the constraints of good sensitivity, localization and noise robustness. To achieve this end, Canny edge detection is implemented internally as a multi-stage algorithm, which involves Gaussian smoothing to remove noise, calculation of gradient magnitudes to localize edge features, non-maximum suppression to remove spurious features, and finally thresholding to yield a binary image. Though the specifics of this internal pipeline are largely abstracted from the user of the class, it is nonetheless beneficial to have a general understanding of these components so that parameters can be appropriately adjusted.

The first step required for using this filter is to include its header file.

```
#include "itkCannyEdgeDetectionImageFilter.h"
```

In this example, images are read and written with `unsigned char` pixel type. However, Canny edge detection requires floating point pixel types in order to avoid numerical errors. For this reason, a separate internal image type with pixel type `double` is defined for edge detection.

```

const unsigned int Dimension = 2;
typedef unsigned char CharPixelType; // IO
typedef double RealPixelType; // Operations

typedef itk::Image< CharPixelType, Dimension > CharImageType;
typedef itk::Image< RealPixelType, Dimension > RealImageType;

```

The CharImageType `image` is cast to and from RealImageType using `itk::CastImageFilter` and `RescaleIntensityImageFilter`, respectively; both the input and output of `CannyEdgeDetectionImageFilter` are RealImageType.

```

typedef itk::CastImageFilter< CharImageType, RealImageType >
    CastToRealFilterType;
typedef itk::CannyEdgeDetectionImageFilter< RealImageType, RealImageType >
    CannyFilterType;
typedef itk::RescaleIntensityImageFilter< RealImageType, CharImageType >
    RescaleFilterType;

```

In this example, three parameters of the Canny edge detection filter may be set via the `SetVariance()`, `SetUpperThreshold()`, and `SetLowerThreshold()` methods. Based on the previous discussion of the steps in the internal pipeline, we understand that variance adjusts the amount of Gaussian smoothing and upperThreshold and lowerThreshold control which edges are selected in the final step.

```

cannyFilter->SetVariance( variance );
cannyFilter->SetUpperThreshold( upperThreshold );
cannyFilter->SetLowerThreshold( lowerThreshold );

```

Finally, `Update()` is called on `writer` to trigger execution of the pipeline. As usual, the call is wrapped in a `try/catch` block.

```

try
{
    writer->Update();
}
catch( itk::ExceptionObject & err )
{
    std::cout << "ExceptionObject caught !" << std::endl;
    std::cout << err << std::endl;
    return EXIT_FAILURE;
}

```

2.3 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.

2.3.1 Linear Mappings

The source code for this section can be found in the file `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 for 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

$$\text{outputPixel} = (\text{inputPixel} - \text{inpMin}) \times \frac{(\text{outMax} - \text{outMin})}{(\text{inpMax} - \text{inpMin})} + \text{outMin}$$

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

$$\text{outputPixel} = (\text{inputPixel} + \text{Shift}) \times \text{Scale}$$

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 "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()` method and assigning the result to `itk::SmartPointers`.

```
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 `Nor-`

malizeImageFilter 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();
```

2.3.2 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 `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.

$$I' = (Max - Min) \cdot \frac{1}{\left(1 + e^{-\left(\frac{I-\beta}{\alpha}\right)}\right)} + Min \quad (2.1)$$

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 2.6 illustrates the significance of each parameter.

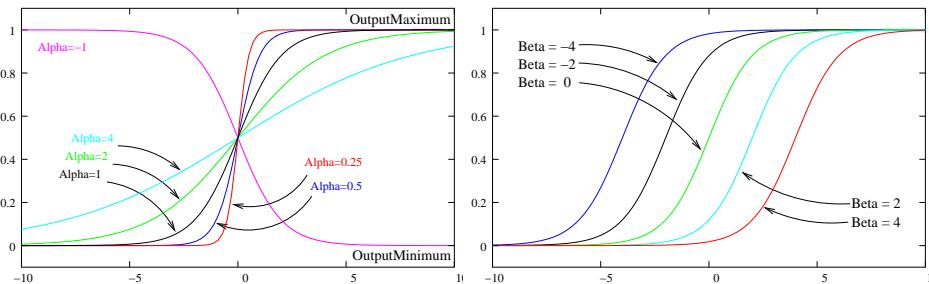


Figure 2.6: Effects of the various parameters in the `SigmoidImageFilter`. The alpha parameter defines the width of the intensity window. The beta parameter defines the center of the intensity window.

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;
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 2.6. 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.

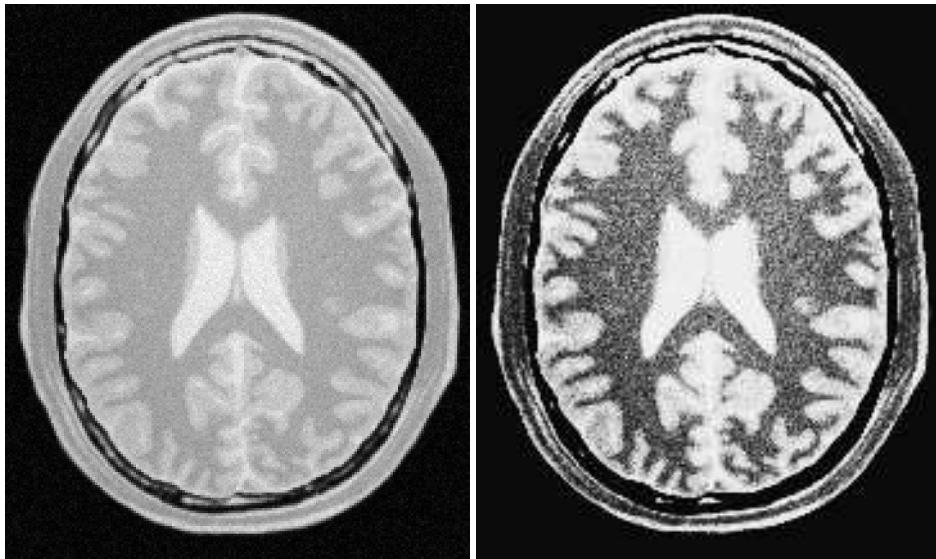


Figure 2.7: Effect of the Sigmoid filter on a slice from a MRI proton density brain image.

```
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.

```
sigmoidFilter->SetInput( reader->GetOutput() );
writer->SetInput( sigmoidFilter->GetOutput() );
writer->Update();
```

Figure 2.7 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 2.6. An application of the Sigmoid filter as preprocessing for segmentation is presented in Section 4.3.1.

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.

2.4 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.

2.4.1 Gradient Magnitude

The source code for this section can be found in the file `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

-1	0	1

-1
0
1

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 2.8 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 the best choice for scale-space analysis.

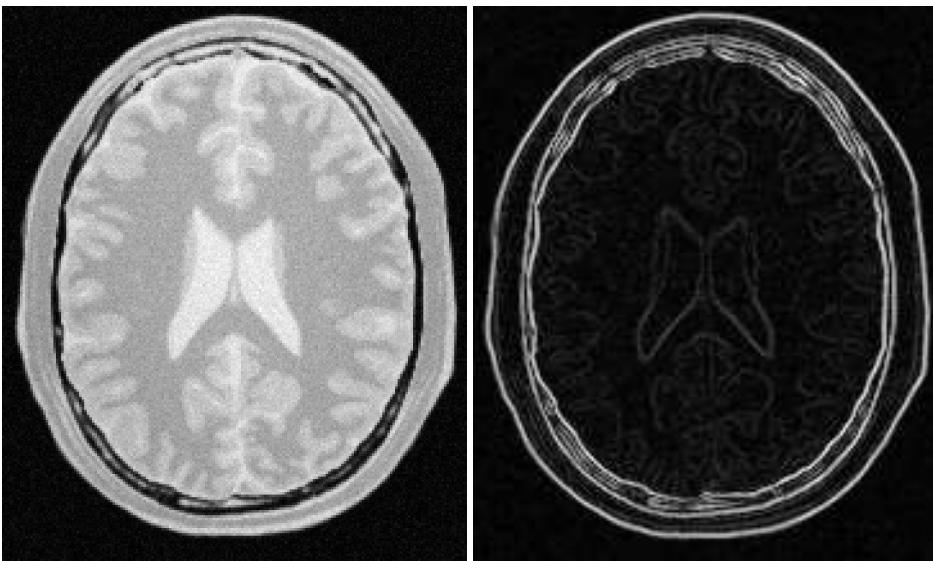


Figure 2.8: Effect of the `GradientMagnitudeImageFilter` on a slice from a MRI proton density image of the brain.

2.4.2 Gradient Magnitude With Smoothing

The source code for this section can be found in the file `GradientMagnitudeRecursiveGaussianImageFilter.cxx`.

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 IIR¹ 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 σ s [16, 17].

`GradientMagnitudeRecursiveGaussianImageFilter` will work on images of any dimension by taking advantage of the natural separability of the Gaussian kernel and its derivatives.

¹ Infinite Impulse Response

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.

```
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 2.9 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

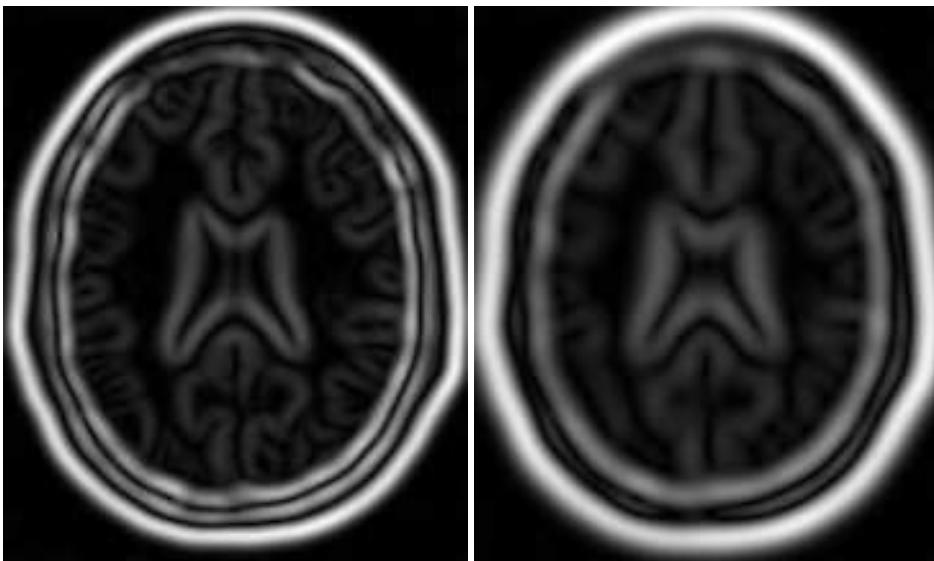


Figure 2.9: Effect of the `GradientMagnitudeRecursiveGaussianImageFilter` on a slice from a MRI proton density image of the brain.

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.

2.4.3 Derivative Without Smoothing

The source code for this section can be found in the file `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.

```
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 2.10 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.

2.5 Second Order Derivatives

2.5.1 Second Order Recursive Gaussian

The source code for this section can be found in the file `SecondDerivativeRecursiveGaussianImageFilter.cxx`.

This example illustrates how to compute second derivatives of a 3D image using the `itk::RecursiveGaussianImageFilter`.

It's good to be able to compute the raw derivative without any smoothing, but this can be problematic in a medical imaging scenario, when images will often have a certain amount of noise. It's almost always more desirable to include a smoothing step first, where an image is convolved with a Gaussian kernel in whichever directions the user desires a derivative. The nature of the Gaussian

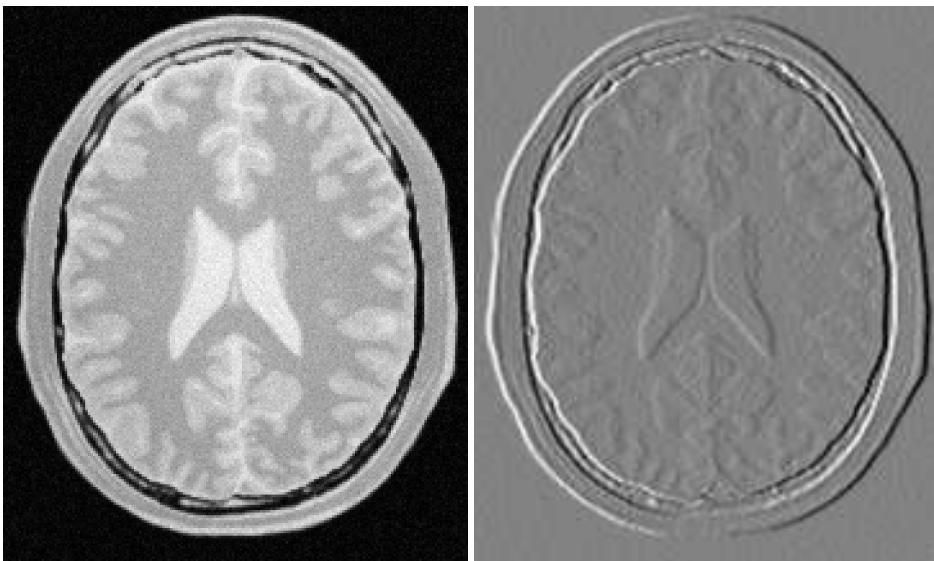


Figure 2.10: Effect of the Derivative filter on a slice from a MRI proton density brain image.

kernel makes it easy to combine these two steps into one, using an infinite impulse response (IIR) filter. 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 (a square matrix of second-order derivatives of an image, which is useful in many image processing techniques).

First, we will include the relevant header files: the `itkRecursiveGaussianImageFilter`, the image reader, writer, and duplicator.

```
#include "itkRecursiveGaussianImageFilter.h"
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
#include "itkImageDuplicator.h"
#include <string>
```

Next, we declare our pixel type and output pixel type to be floats, and our image dimension to be 3.

```
typedef float          PixelType;
typedef float          OutputPixelType;

const unsigned int Dimension = 3;
```

Using these definitions, define the image types, reader and writer types, and duplicator types, which are templated over the pixel types and dimension. Then, instantiate the reader, writer, and duplicator with the `New()` method.

```
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();
```

Here we create three new filters. For each derivative we take, we will want to smooth in that direction first. So after the filters are created, each is given a dimension, and set to (in this example) the same sigma. Note that here, σ represents the standard deviation, whereas the [itk::DiscreteGaussianImageFilter](#) exposes the SetVariance method.

```
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 );
}
```

First we will compute the second derivative of the z -direction. In order to do this, we smooth in the x - and y - directions, and finally smooth and compute the derivative in the z -direction. Taking the zero-order derivative is equivalent to simply smoothing in that direction. This result is commonly notated I_{zz} .

```

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->GetModifiableOutput();

```

Recall that `gc` is the filter responsible for taking the second derivative. We can now take advantage of the pipeline architecture and, without much hassle, switch the direction of `gc` and `gb`, so that `gc` now takes the derivatives in the y -direction. Now we only need to call `Update()` on `gc` to re-run the entire pipeline from `ga` to `gc`, obtaining the second-order derivative in the y -direction, which is commonly notated I_{yy} .

```

gc->SetDirection( 1 ); // gc now works along Y
gb->SetDirection( 2 ); // gb now works along Z

gc->Update();
duplicator->Update();

ImageType::Pointer Iyy = duplicator->GetModifiableOutput();

```

Now we switch the directions of `gc` with that of `ga` in order to take the derivatives in the x -direction. This will give us I_{xx} .

```

gc->SetDirection( 0 ); // gc now works along X
ga->SetDirection( 1 ); // ga now works along Y

gc->Update();
duplicator->Update();

ImageType::Pointer Ixx = duplicator->GetModifiableOutput();

```

Now we can reset the directions to their original values, and compute first derivatives in different directions. Since we set both `gb` and `gc` to compute first derivatives, and `ga` to zero-order (which is only smoothing) we will obtain I_{yz} .

```

ga->SetDirection( 0 );
gb->SetDirection( 1 );
gc->SetDirection( 2 );

ga->SetZeroOrder();
gb->SetFirstOrder();
gc->SetFirstOrder();

gc->Update();
duplicator->Update();

ImageType::Pointer Iyz = duplicator->GetModifiableOutput();
```

Here is how you may easily obtain I_{xz} .

```

ga->SetDirection( 1 );
gb->SetDirection( 0 );
gc->SetDirection( 2 );

ga->SetZeroOrder();
gb->SetFirstOrder();
gc->SetFirstOrder();

gc->Update();
duplicator->Update();

ImageType::Pointer Ixz = duplicator->GetModifiableOutput();
```

For the sake of completeness, here is how you may compute I_{xz} and I_{xy} .

```

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->GetModifiableOutput();

writer->SetInput( Ixy );
outputFileName = outputPrefix + "-Ixy.mhd";
writer->SetFileName( outputFileName.c_str() );
writer->Update();
```

2.5.2 Laplacian Filters

Laplacian Filter Recursive Gaussian

The source code for this section can be found in the file `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.

$$\frac{1}{\sigma\sqrt{2\pi}} \quad (2.2)$$

In applications that use the Gaussian as a solution of the diffusion equation it is desirable to use a normalization that preserves the integral of the signal. This last approach can be seen as a conservation of mass principle. This is represented by the following equation.

$$\frac{1}{\sigma^2\sqrt{2\pi}} \quad (2.3)$$

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 when 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.

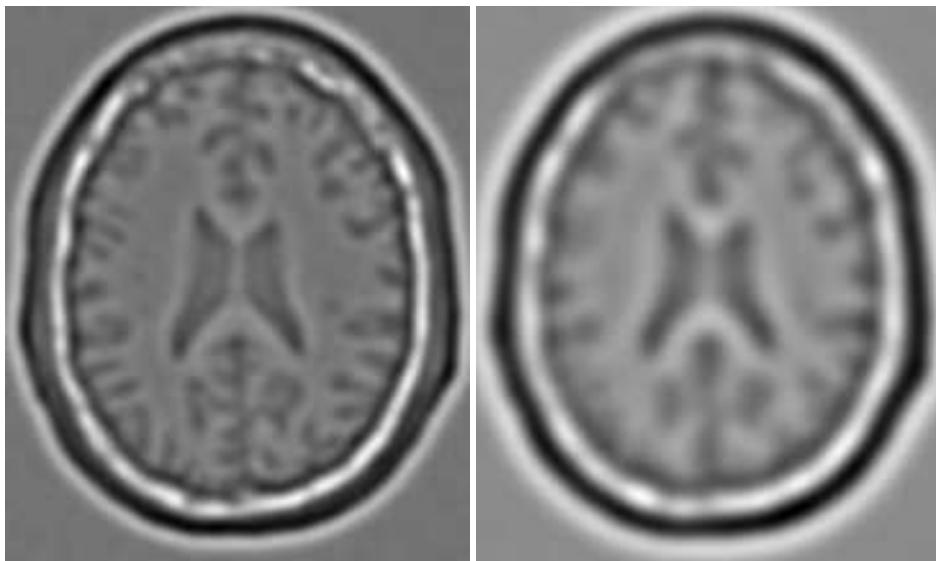


Figure 2.11: Effect of the `LaplacianRecursiveGaussianImageFilter` on a slice from a MRI proton density image of the brain.

```
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();
```

The source code for this section can be found in the file `LaplacianRecursiveGaussianImageFilter2.cxx`.

The previous example 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.

```
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;
}
```

2.6 Neighborhood Filters

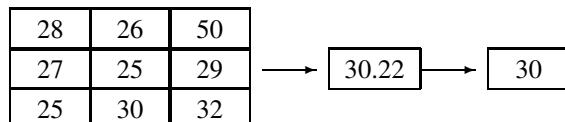
The concept of locality is frequently encountered in image processing in the form of filters that compute every output pixel using information from a small region in the neighborhood of the input pixel. The classical form of these filters are the 3×3 filters in 2D images. Convolution masks based on these 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 the Iterators chapter in Book 1 for more information.)

2.6.1 Mean Filter

The source code for this section can be found in the file `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 2D 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 2.12 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.

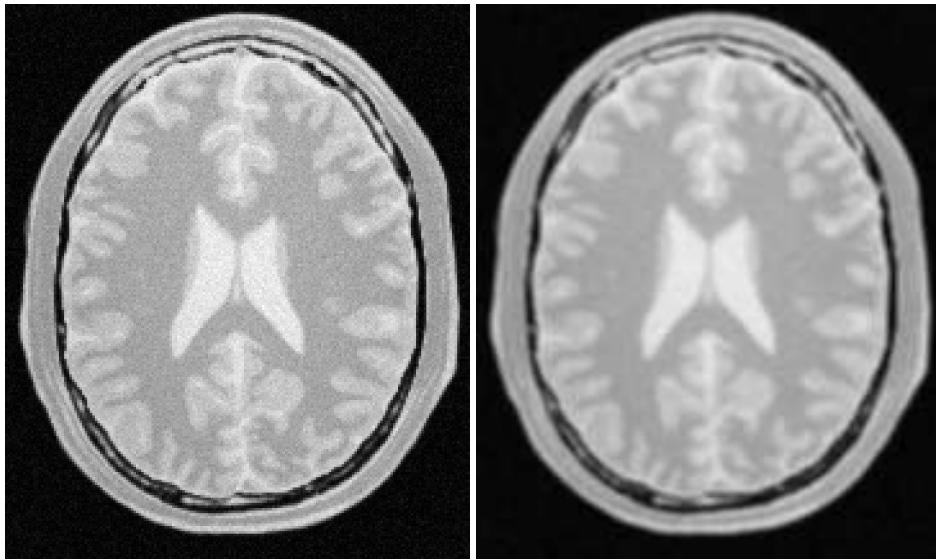


Figure 2.12: Effect of the MeanImageFilter on a slice from a MRI proton density brain image.

2.6.2 Median Filter

The source code for this section can be found in the file `MedianImageFilter.cxx`.

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 2D 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.

28	26	50
27	25	29
25	30	32

→ 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 2.13 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.

2.6.3 Mathematical Morphology

Mathematical morphology has proved to be a powerful resource for image processing and analysis [56]. ITK implements mathematical morphology filters using `NeighborhoodIterators` and `itk::NeighborhoodOperators`s. The toolkit contains two types of image morphology algorithms: filters that operate on binary images and filters that operate on grayscale images.

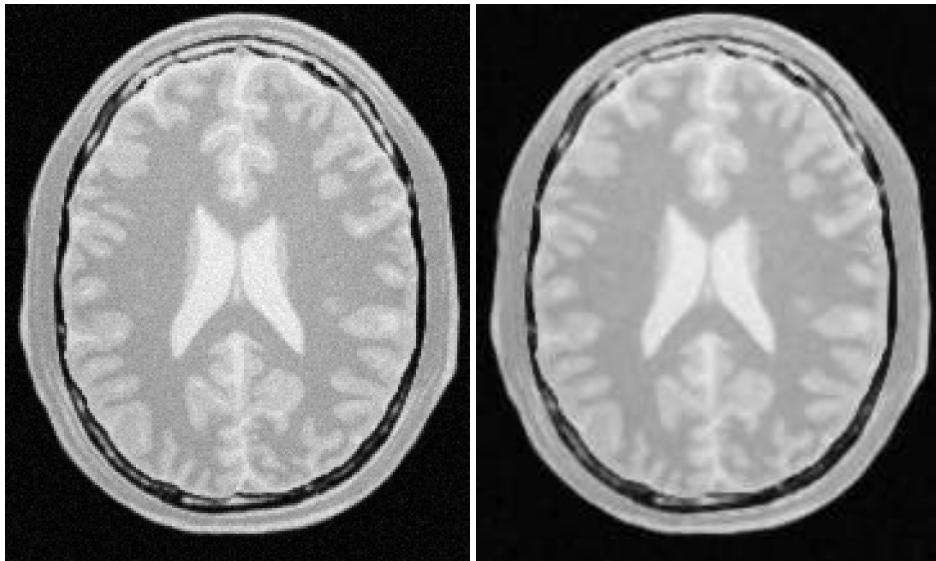


Figure 2.13: Effect of the `MedianImageFilter` on a slice from a MRI proton density brain image.

Binary Filters

The source code for this section can be found in the file `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 neighbor-

hood 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::SmartPointers`.

```
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.

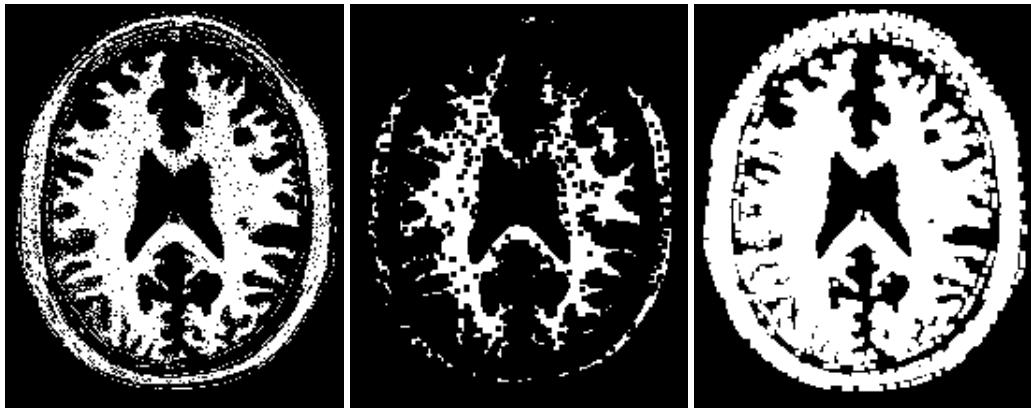


Figure 2.14: Effect of erosion and dilation in a binary image.

```
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, such as an image writer.

```
writerDilation->SetInput( binaryDilate->GetOutput() );
writerDilation->Update();
```

Figure 2.14 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 `MathematicalMorphologyGrayscaleFilters.cxx`.

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() );
```

The filter is executed by invoking its `Update()` method, or by updating any downstream filter, such as an image writer.

```
writerDilation->SetInput( grayscaleDilate->GetOutput() );
writerDilation->Update();
```

Figure 2.15 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.

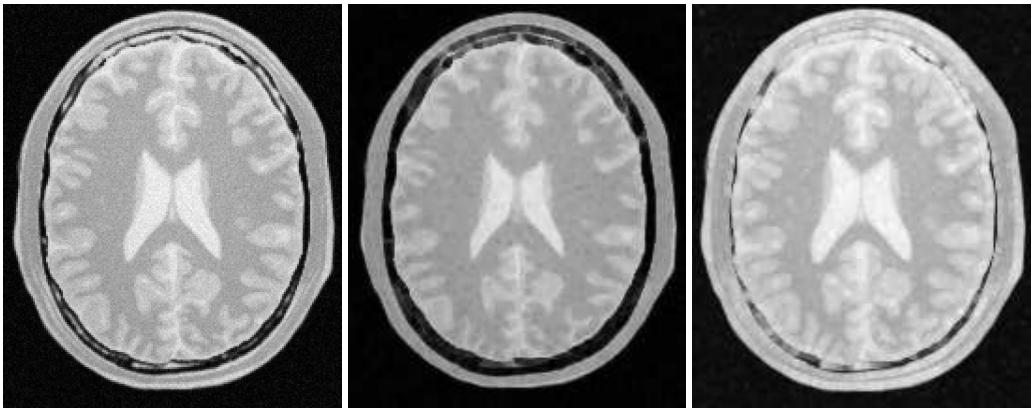


Figure 2.15: Effect of erosion and dilation in a grayscale image.

2.6.4 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 label a foreground pixel as background if the number of background neighbors is greater than or equal to 1. In this context, you could imagine variations of erosion in which the count could be changed to require at least 3 foreground pixels in its neighborhood.

Binary Median Filter

One case of a voting filter is the `BinaryMedianImageFilter`. This filter is equivalent to applying a Median filter over a binary image. Having a binary image as input makes it 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
`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.

```
filter->SetInput( reader->GetOutput() );
writer->SetInput( filter->GetOutput() );
writer->Update();
```

Figure 2.16 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 2.17 below shows the effect of running the median filter with a 3x3 classical window size

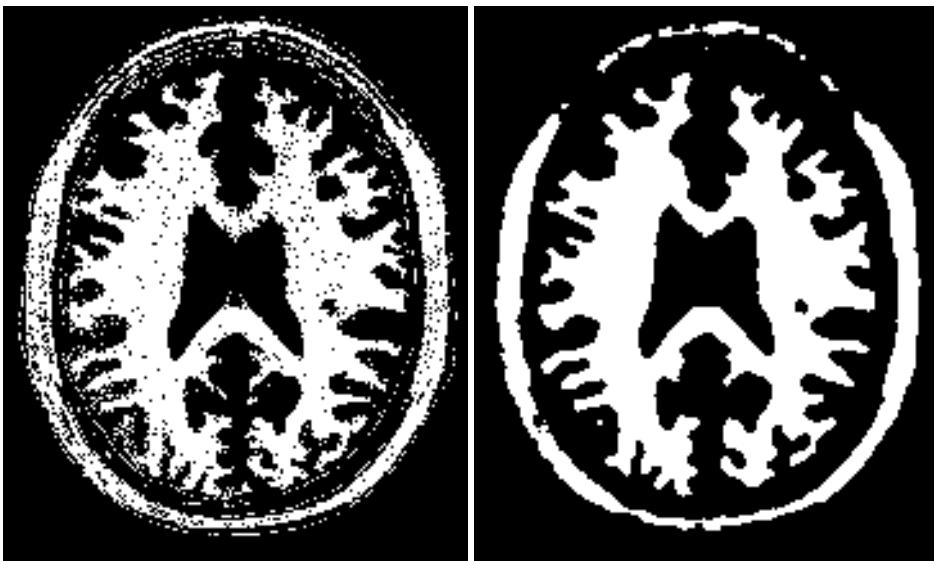


Figure 2.16: Effect of the `BinaryMedianImageFilter` on a slice from a MRI proton density brain image that has been thresholded in order to produce a binary image.

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 sizes. To 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

`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.



Figure 2.17: Effect of 1, 10 and 50 iterations of the `BinaryMedianImageFilter` using a 3×3 window.

```
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 of 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 $(3 \times 3 - 1) / 2 + \text{majority}$. 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();
```

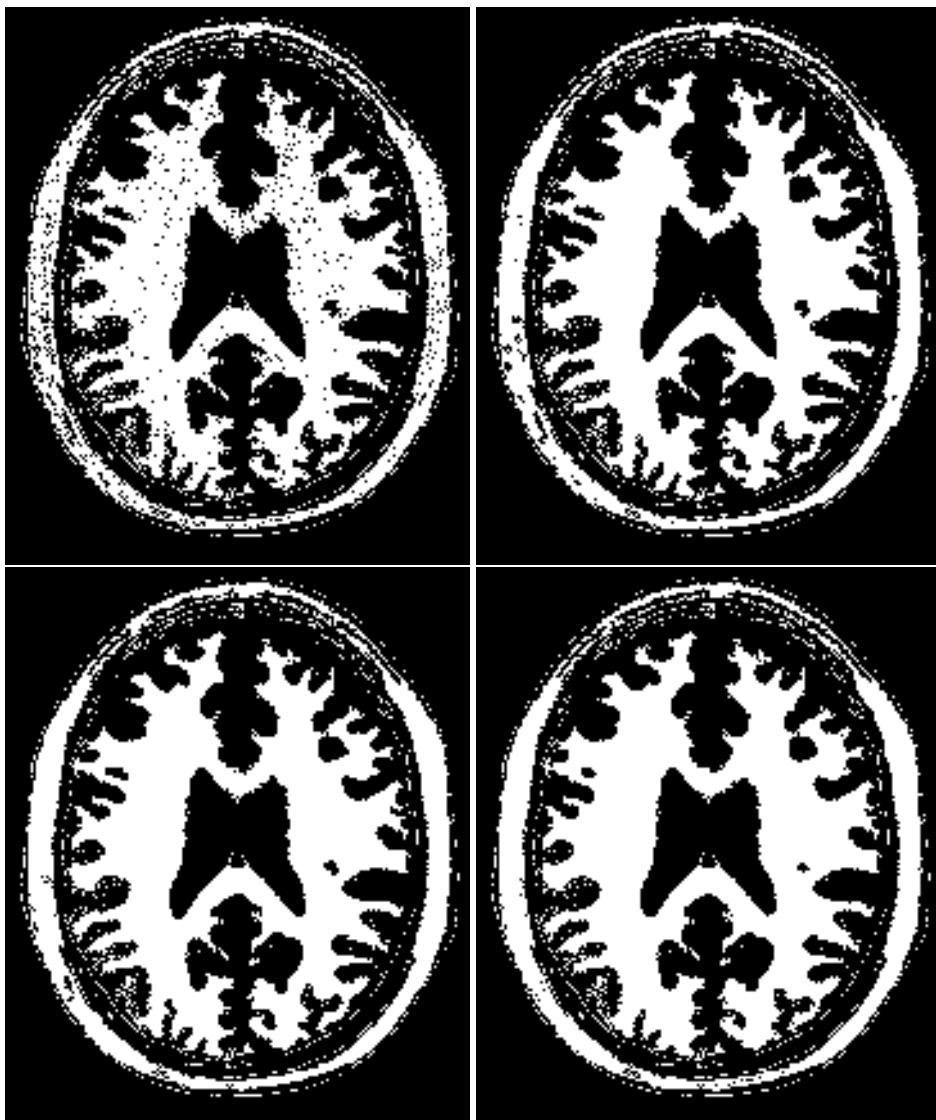


Figure 2.18: Effect of the `VotingBinaryHoleFillingImageFilter` on a slice from a MRI proton density brain image that has been thresholded in order to produce a binary image. The output images have used radius 1,2 and 3 respectively.

Figure 2.18 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 `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 a `itk::VotingBinaryHoleFillingImageFilter` internally 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.

```
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 of 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 $(3 \times 3 - 1)/2 + \text{majority}$. This is done with the `SetMajorityThreshold()` method.

```
filter->SetMajorityThreshold( 2 );
```

Finally we specify the maximum number of iterations for which this filter should run. The number of iterations will determine the maximum size of holes and cavities that this filter will be able to fill. The more iterations you run, 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 2.19 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.

2.7 Smoothing Filters

Real image data has a level of uncertainty which 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.

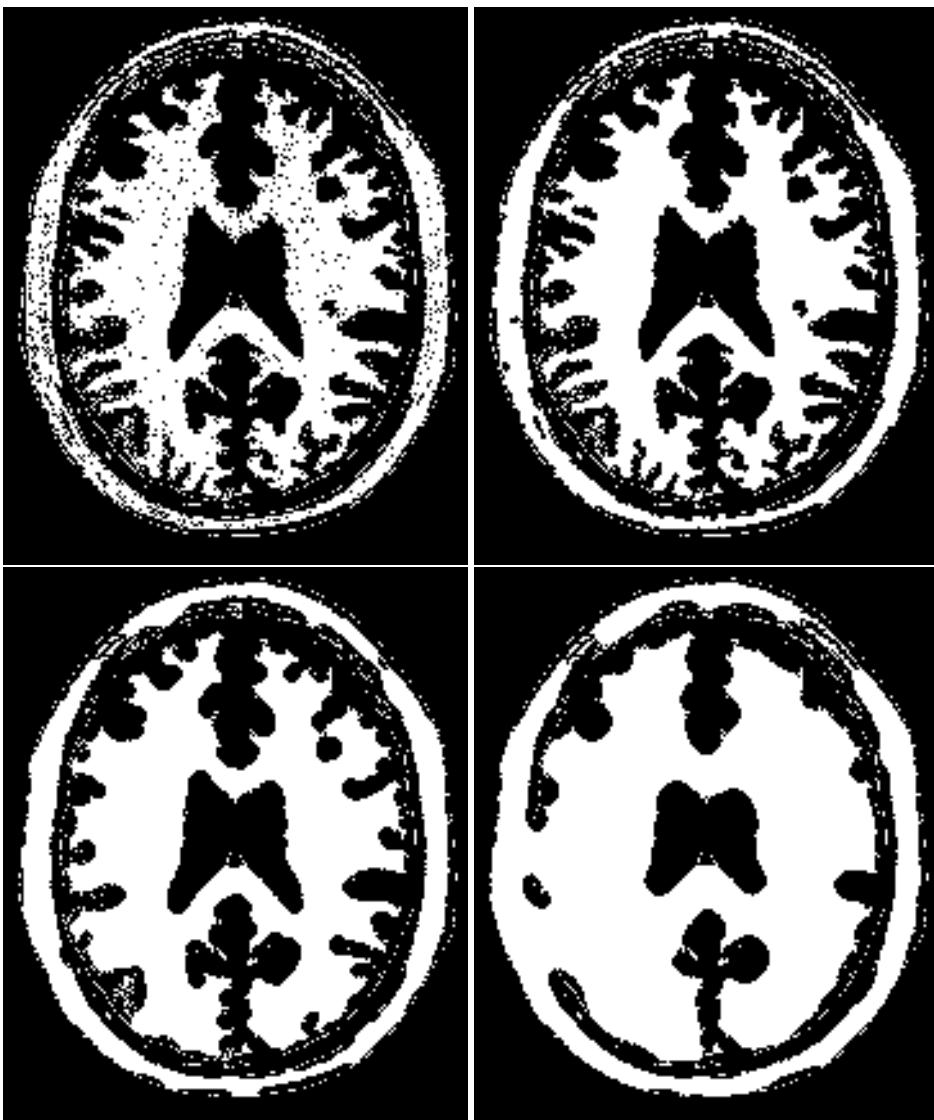


Figure 2.19: Effect of the `VotingBinaryIterativeHoleFillingImageFilter` on a slice from a MRI proton density brain image that has been thresholded in order to produce a binary image. The output images have used radius 1,2 and 3 respectively.

2.7.1 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 [16, 17].

Discrete Gaussian

The source code for this section can be found in the file `DiscreteGaussianImageFilter.cxx`.

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.

Gaussian kernels in ITK are constructed according to the theory of Tony Lindeberg [35] 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. As with other examples, the includes here are truncated to those specific for this example.

```
#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.

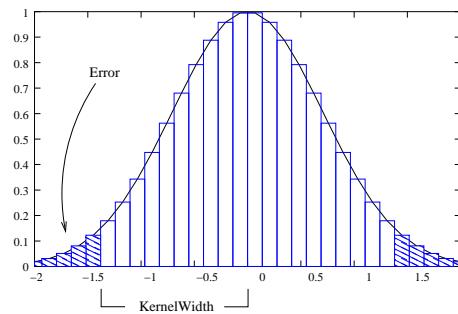


Figure 2.20: Discretized Gaussian.

```
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.

```
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 will trigger the execution of this one. For example, a writer could be used after the filter.

```
rescaler->SetInput( filter->GetOutput() );
writer->SetInput( rescaler->GetOutput() );
writer->Update();
```

Figure 2.21 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 longer computation times. Unless a high degree of accuracy is required, it may be more desirable to use the approximating `itk::RecursiveGaussianImageFilter` with large variances.

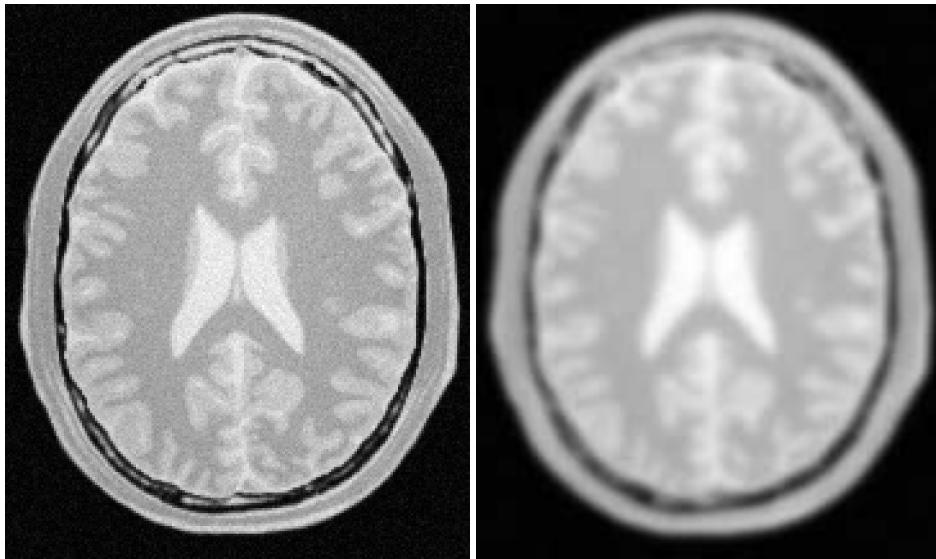


Figure 2.21: Effect of the `DiscreteGaussianImageFilter` on a slice from a MRI proton density image of the brain.

Binomial Blurring

The source code for this section can be found in the file `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.

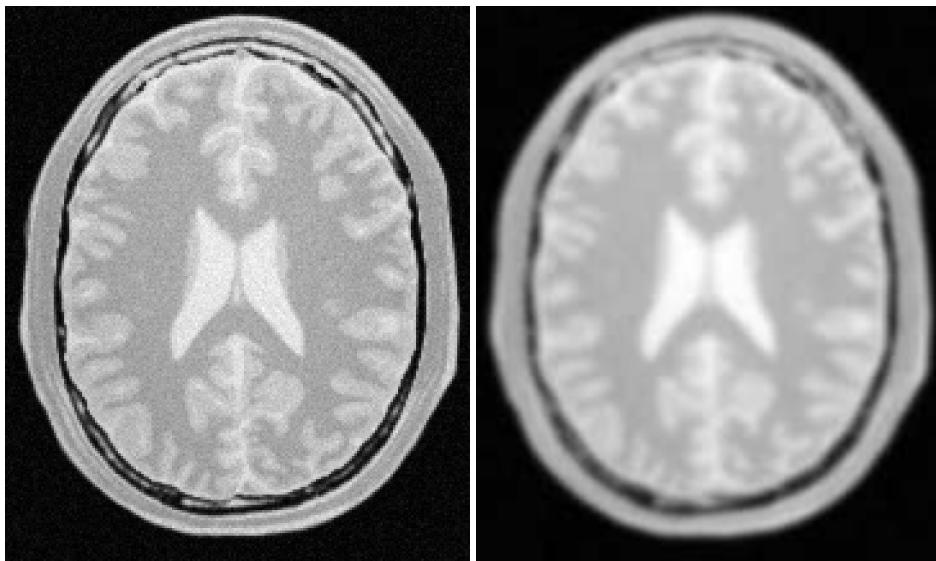


Figure 2.22: Effect of the `BinomialBlurImageFilter` on a slice from a MRI proton density image of the brain.

```
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 2.22 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 `SmoothingRecursiveGaussianImageFilter.cxx`.

The classical method of smoothing an image by convolution with a Gaussian kernel has the draw-

back 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 IIR² filters. In practice this filter requires a constant number of operations for approximating the convolution, regardless of the σ value [16, 17].

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`.

```
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.

²Infinite Impulse Response

```
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.

$$\frac{1}{\sigma\sqrt{2\pi}} \quad (2.4)$$

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.

$$\frac{1}{\sigma^2\sqrt{2\pi}} \quad (2.5)$$

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();
```

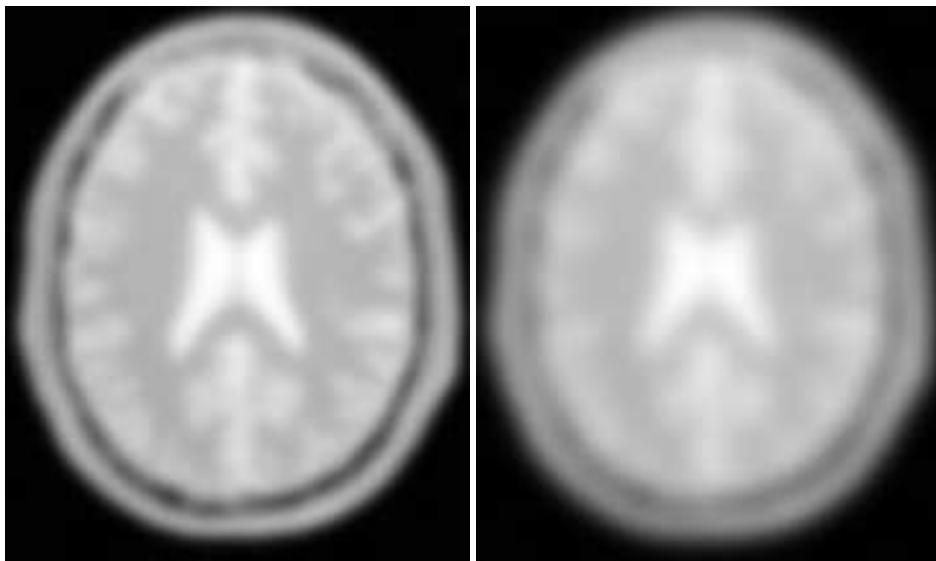


Figure 2.23: Effect of the `SmoothingRecursiveGaussianImageFilter` on a slice from a MRI proton density image of the brain.

Figure 2.23 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 `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.

2.7.2 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 `GaussianBlurImageFunction.cxx`.

2.7.3 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 [46] introduced an alternative to linear-filtering that they called *anisotropic diffusion*. Anisotropic diffusion is closely related to the earlier work of Grossberg [23], 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

$$\frac{\partial g(x,y,t)}{\partial t} = \nabla \cdot \nabla g(x,y,t), \quad (2.6)$$

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.

$$g_t = \nabla \cdot c(|\nabla g|) \nabla g, \quad (2.7)$$

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

$$c(|\nabla g|) = e^{-\frac{|\nabla g|^2}{2k^2}} \quad (2.8)$$

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 2.7 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 [22, 68] demonstrated the effectiveness of anisotropic diffusion on medical images. In a series of papers on the subject [72, 70, 71, 68, 69, 66], 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

$$\vec{F}_t = \nabla \cdot c(\mathcal{D}\vec{F})\vec{F}, \quad (2.9)$$

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 is also useful for processing registered data from different devices or for denoising higher-order geometric or statistical features from scalar-valued images [66, 73].

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 [69] 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 [61]. These include alternatives for constructing dissimilarity measures [54], directional (i.e., tensor-valued) conductance terms [65, 3] and level set interpretations [67].

Gradient Anisotropic Diffusion

The source code for this section can be found in the file
`GradientAnisotropicDiffusionImageFilter.cxx`.

The `itk::GradientAnisotropicDiffusionImageFilter` implements an N -dimensional version of the classic Perona-Malik anisotropic diffusion equation for scalar-valued images [46].

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.

$$C(\mathbf{x}) = e^{-(\frac{\|\nabla U(\mathbf{x})\|}{K})^2} \quad (2.10)$$

The numerical implementation of this equation is similar to that described in the Perona-Malik paper [46], 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();
```

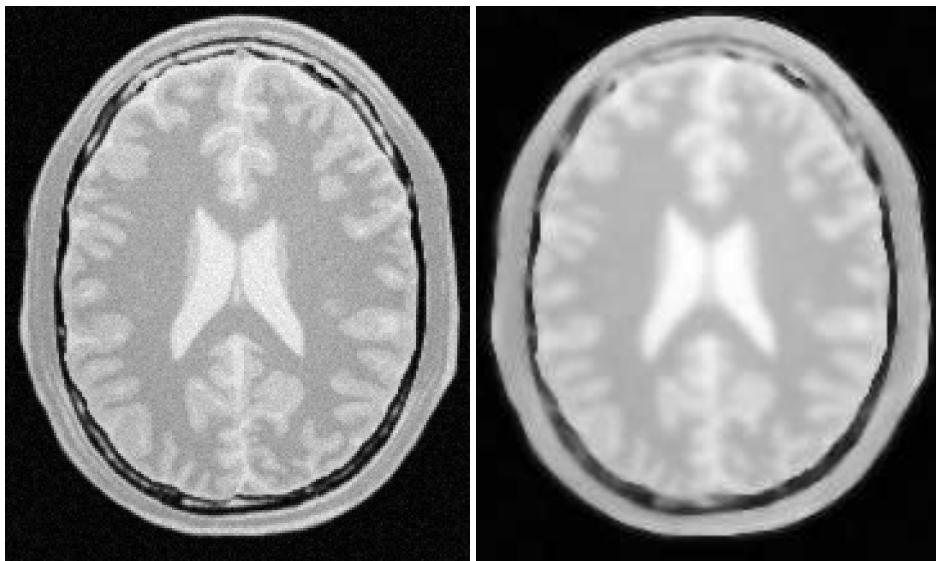


Figure 2.24: Effect of the `GradientAnisotropicDiffusionImageFilter` on a slice from a MRI Proton Density image of the brain.

Typical values for the time step are 0.25 in 2D images and 0.125 in 3D images. The number of iterations is typically set to 5; more iterations result in further smoothing and will increase the computing time linearly.

Figure 2.24 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 `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:

$$f_t = |\nabla f| \nabla \cdot c(|\nabla f|) \frac{\nabla f}{|\nabla f|} \quad (2.11)$$

where the conductance modified curvature term is

$$\nabla \cdot \frac{\nabla f}{|\nabla f|} \quad (2.12)$$

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() );
```

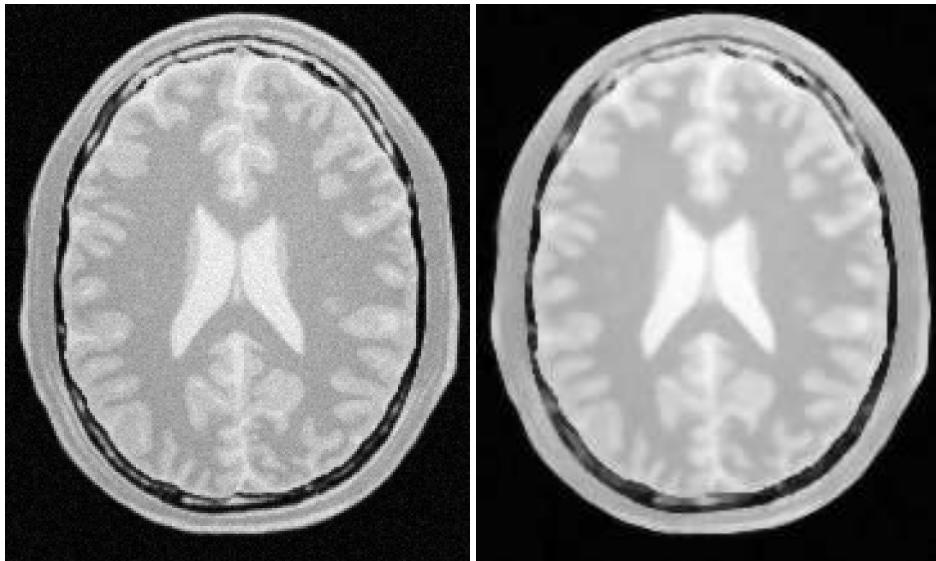


Figure 2.25: Effect of the `CurvatureAnisotropicDiffusionImageFilter` on a slice from a MRI Proton Density image of the brain.

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 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 the computing time linearly. The conductance parameter is usually around 3.0.

Figure 2.25 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 0.125, 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 `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 an 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:

$$I_t = \kappa |\nabla I| \quad (2.13)$$

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 will still occur. Thus, continual application of this curvature flow scheme will eventually result in 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 the computing time linearly. 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 trigger the execution of this one. For example, a writer filter could be used after the curvature flow filter.

```
rescaler->SetInput( filter->GetOutput() );
writer->SetInput( rescaler->GetOutput() );
writer->Update();
```

Figure 2.26 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 0.25 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`

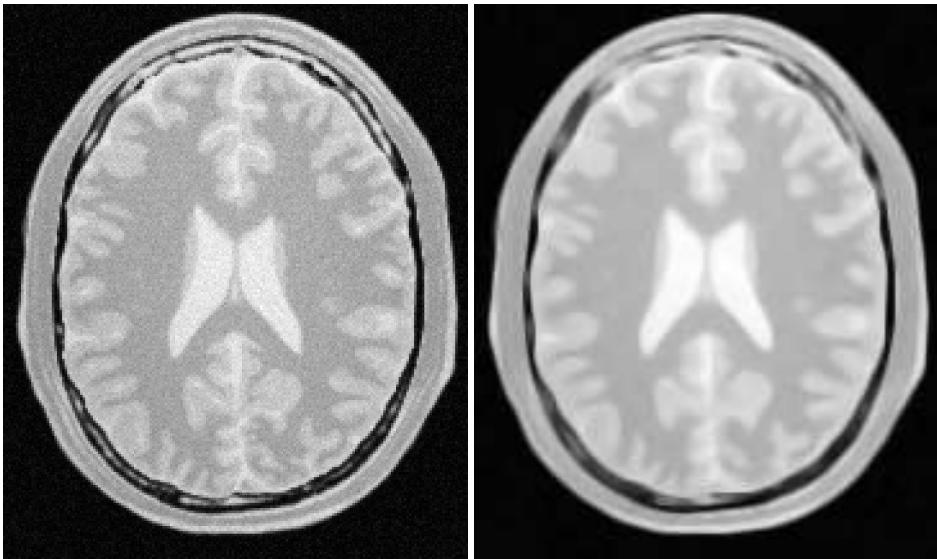


Figure 2.26: Effect of the `CurvatureFlowImageFilter` on a slice from a MRI proton density image of the brain.

MinMaxCurvature Flow

The source code for this section can be found in the file `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:

$$I_t = F |\nabla I| \quad (2.14)$$

where F is defined as

$$F = \begin{cases} \max(\kappa, 0) & : \text{Average} < \text{Threshold} \\ \min(\kappa, 0) & : \text{Average} \geq \text{Threshold} \end{cases} \quad (2.15)$$

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

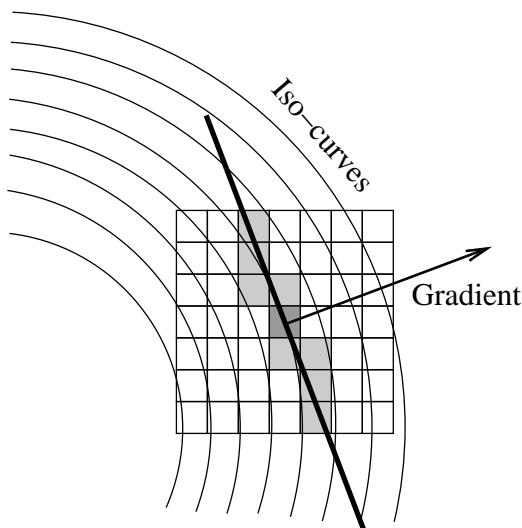


Figure 2.27: Elements involved in the computation of min-max curvature flow.

shrink. Comparison between the neighborhood average and the threshold is used to select the the right speed function to use. This switching prevents the unwanted diffusion of the simple curvature flow method.

Figure 2.27 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, 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 will trigger the execution of this one. For example, a writer filter can be used after the curvature flow filter.

```
rescaler->SetInput( filter->GetOutput() );
writer->SetInput( rescaler->GetOutput() );
writer->Update();
```

Figure 2.28 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 result in the figure has sharper edges than the same example using simple curvature flow in Figure 2.26.

The following classes provide similar functionality:

- `itk::CurvatureFlowImageFilter`

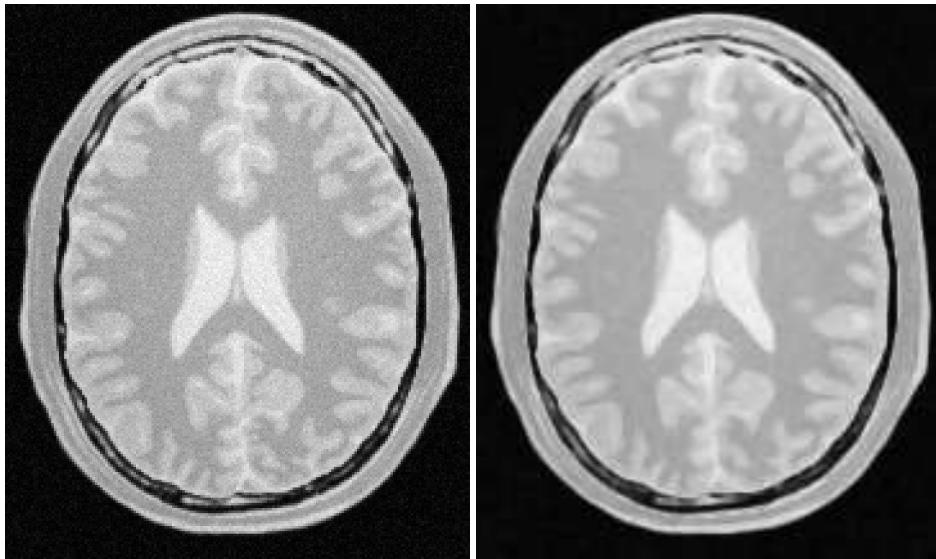


Figure 2.28: Effect of the `MinMaxCurvatureFlowImageFilter` on a slice from a MRI proton density image of the brain.

Bilateral Filter

The source code for this section can be found in the file `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 is 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

$$h(\mathbf{x}) = k(\mathbf{x})^{-1} \int_{\omega} f(\mathbf{w}) c(\mathbf{x}, \mathbf{w}) s(f(\mathbf{x}), f(\mathbf{w})) d\mathbf{w} \quad (2.16)$$

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 respec-

tively. 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

$$k(\mathbf{x}) = \int_{\omega} c(\mathbf{x}, \mathbf{w}) s(f(\mathbf{x}), f(\mathbf{w})) d\mathbf{w} \quad (2.17)$$

The default implementation of this filter uses Gaussian kernels for both $c()$ and $s()$. The c kernel can be described as

$$c(\mathbf{x}, \mathbf{w}) = e^{\left(\frac{\|\mathbf{x}-\mathbf{w}\|^2}{\sigma_c^2}\right)} \quad (2.18)$$

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

$$s(f(\mathbf{x}), f(\mathbf{w})) = e^{\left(\frac{(f(\mathbf{x})-f(\mathbf{w}))^2}{\sigma_s^2}\right)} \quad (2.19)$$

where σ_s is provided by the user and defines how close the neighbor's 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, we must specify the standard deviation σ to be used for the Gaussian kernel on image intensities. Second, the set of σ_s 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 it 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 2.29 illustrates the effect of this filter on a MRI proton density image of the brain. In this example the filter was run with a range σ 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`

2.7.4 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.

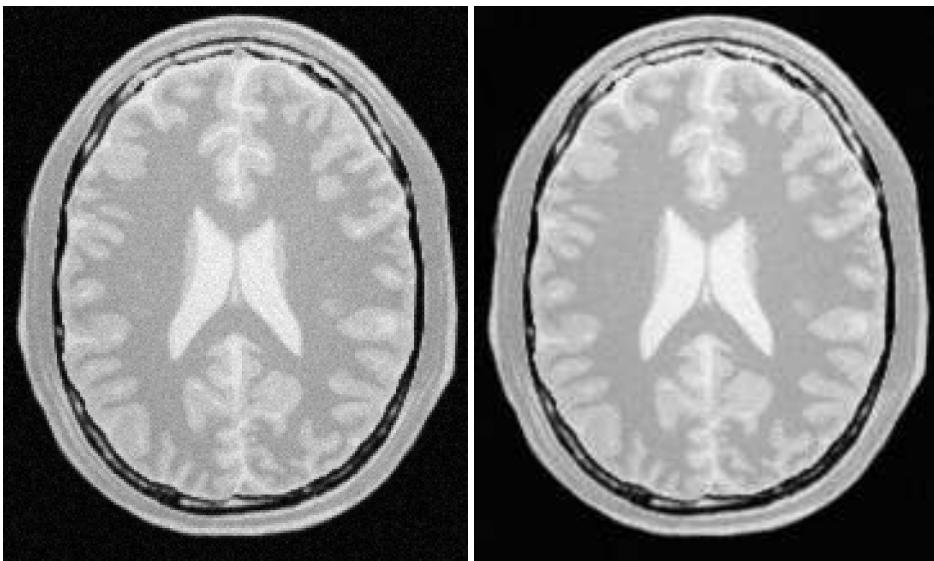


Figure 2.29: Effect of the `BilateralImageFilter` on a slice from a MRI proton density image of the brain.

Vector Gradient Anisotropic Diffusion

The source code for this section can be found in the file `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 [2.7.3](#).

This filter is designed to process images of `itk::Vector` type. The code relies on various `typedefs` and overloaded operators defined in `itk::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 `itk::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, however more iterations will result in further smoothing and will linearly increase the computing time.

Figure 2.30 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 `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 2.7.3.

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`

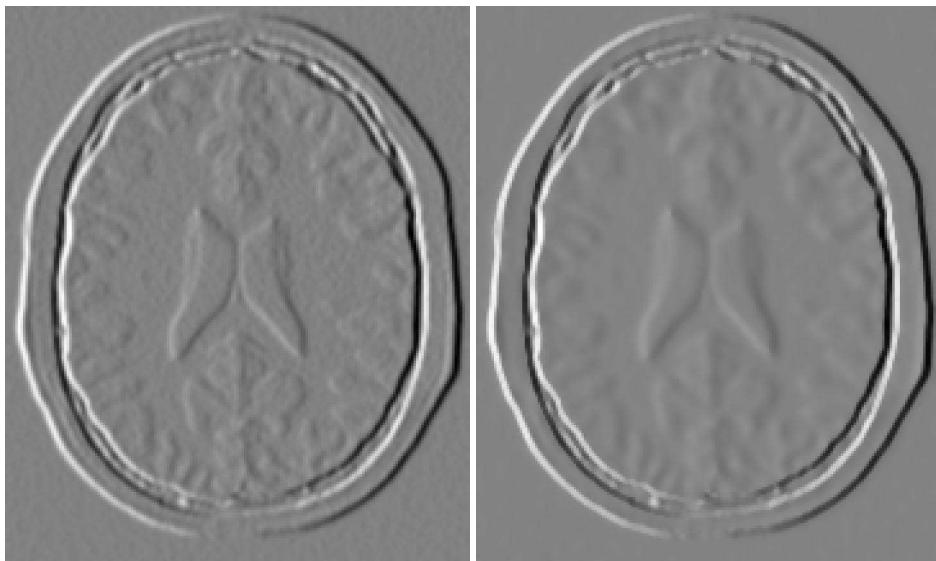


Figure 2.30: Effect of the `VectorGradientAnisotropicDiffusionImageFilter` on the *X* component of the gradient from a MRI proton density brain image.

and overloaded operators defined in `itk::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 `itk::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();
```

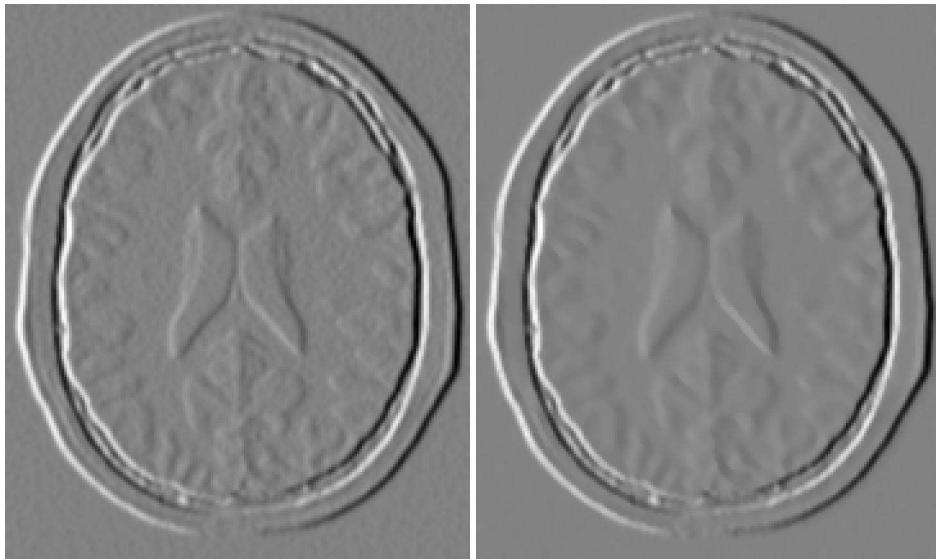


Figure 2.31: Effect of the `VectorCurvatureAnisotropicDiffusionImageFilter` on the *X* component of the gradient from a MRI proton density brain image.

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, however more iterations will result in further smoothing and will increase the computing time linearly.

Figure 2.31 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.

2.7.5 Edge Preserving Smoothing in Color Images

Gradient Anisotropic Diffusion

The source code for this section can be found in the file `RGBGradientAnisotropicDiffusionImageFilter.cxx`.

The vector anisotropic diffusion approach applies to color images equally well. 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.

```
#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 2.32 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.

Curvature Anisotropic Diffusion

The source code for this section can be found in the file `RGBCurvatureAnisotropicDiffusionImageFilter.cxx`.

The vector anisotropic diffusion approach can be applied equally well 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.

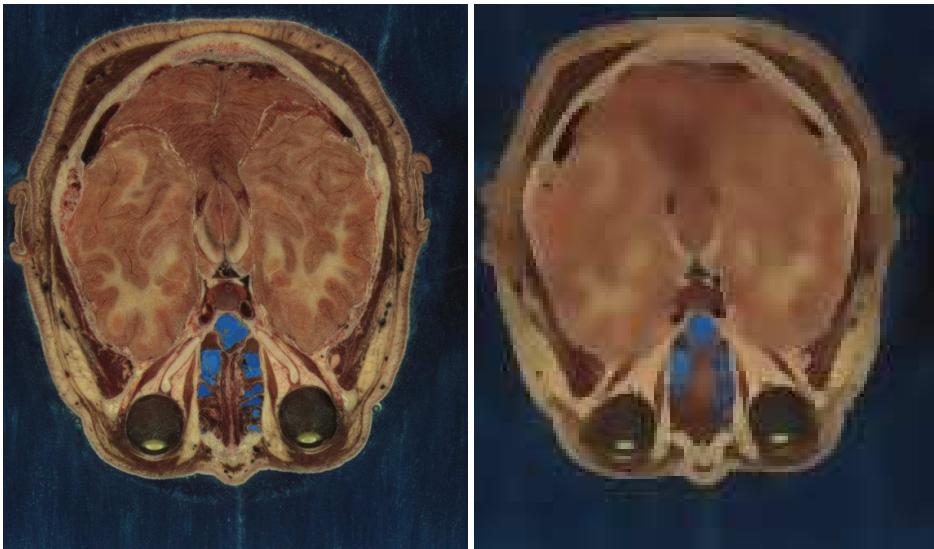


Figure 2.32: Effect of the `VectorGradientAnisotropicDiffusionImageFilter` on a RGB image from a cryogenic section of the Visible Woman data set.

```
#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 a 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 2.33 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 at 2GHz.

Figure 2.34 compares the effect of the gradient and curvature anisotropic diffusion filters on a small region of the same cryogenic slice used in Figure 2.33. The region used in this figure is only 127×162 pixels and took 14 seconds to compute on the same platform.

2.8 Distance Map

The source code for this section can be found in the file `DanielssonDistanceMapImageFilter.cxx`.

This example illustrates the use of the `itk::DanielssonDistanceMapImageFilter`. This filter

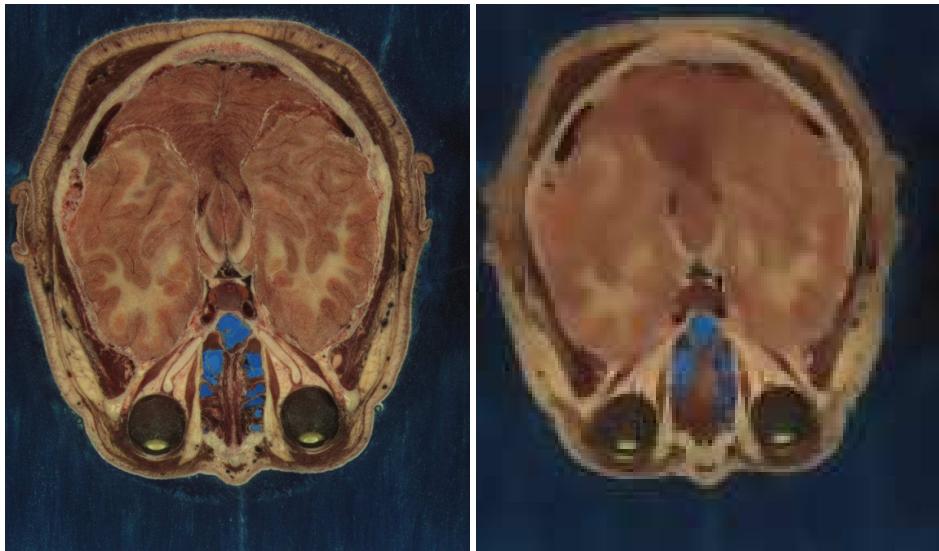


Figure 2.33: Effect of the VectorCurvatureAnisotropicDiffusionImageFilter on a RGB image from a cryogenic section of the Visible Woman data set.

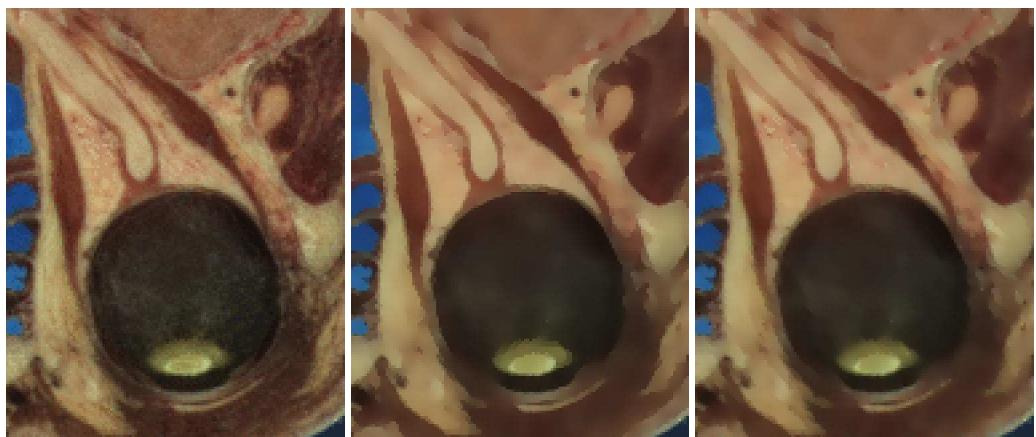


Figure 2.34: Comparison between the gradient (center) and curvature (right) Anisotropic Diffusion filters. Original image at left.

generates a distance map from the input image using the algorithm developed by Danielsson [13]. 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. The label of each group of pixels is assigned by the `itk::ConnectedComponentImageFilter`.

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 -dimensional terms, the maximum extension along all the dimensions. The input, output (distance map), and voronoi partition image types are now defined using their respective pixel type and dimension.

```
typedef unsigned char           InputPixelType;
typedef unsigned short          OutputPixelType;
typedef unsigned char           VoronoiPixelType;
typedef itk::Image< InputPixelType, 2 > InputImageType;
typedef itk::Image< OutputPixelType, 2 > OutputImageType;
typedef itk::Image< VoronoiPixelType, 2 > VoronoiImageType;
```

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, VoronoiImageType > FilterType;
FilterType::Pointer filter = FilterType::New();
```

The input to the filter is taken from a reader and its output is passed to a `itk::RescaleIntensityImageFilter` and then to a writer. The scaler and writer are both templated over the image type, so we instantiate a separate pipeline for the voronoi partition map starting at the scaler.

```
labeler->SetInput(reader->GetOutput() );
filter->SetInput( labeler->GetOutput() );
scaler->SetInput( filter->GetOutput() );
writer->SetInput( scaler->GetOutput() );
```

The Voronoi map is obtained with the `GetVoronoiMap()` method. In the lines below we connect this output to the intensity rescaler.

```
voronoiScaler->SetInput( filter->GetVoronoiMap() );
voronoiWriter->SetInput( voronoiScaler->GetOutput() );
```

Figure 2.35 illustrates the effect of this filter on a binary image with a set of points. The input image is shown at the left, and the distance map at the center and the Voronoi partition at the right. This filter computes distance maps in N -dimensions and is therefore capable of producing N -dimensional

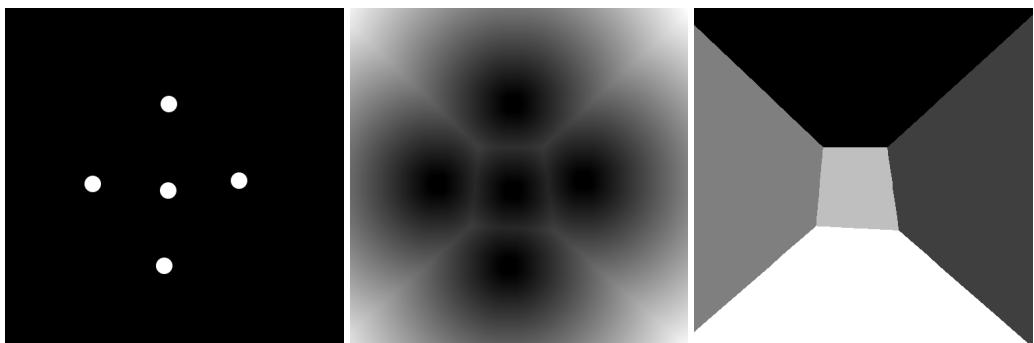


Figure 2.35: `DanielssonDistanceMapImageFilter` output. Set of pixels, distance map and Voronoi partition.

Voronoi partitions.

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 `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 -dimensional 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;
typedef unsigned short VoronoiPixelType;
const unsigned int Dimension = 2;

typedef itk::Image< InputPixelType, Dimension > InputImageType;
typedef itk::Image< OutputPixelType, Dimension > OutputImageType;
typedef itk::Image< VoronoiPixelType, Dimension > VoronoiImageType;
```

The only change with respect to the previous example is to replace the `DanielssonDistanceMapImageFilter` with the `SignedDanielssonDistanceMapImageFilter`.

```
typedef itk::SignedDanielssonDistanceMapImageFilter<
    InputImageType,
    OutputImageType,
    VoronoiImageType > FilterType;

FilterType::Pointer filter = FilterType::New();
```

The distances inside the circle are defined to be negative, while the distances outside the circle are positive. To change the convention, use the `InsideIsPositive(bool)` function.

Figure 2.36 illustrates the effect of this filter. The input image and the distance map are shown.

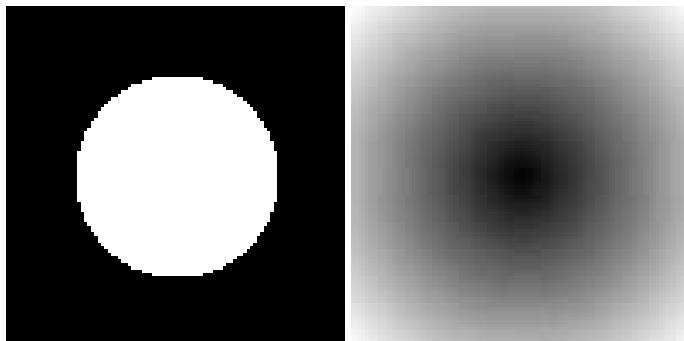


Figure 2.36: `SignedDanielssonDistanceMapImageFilter` applied on a binary circle image. The intensity has been rescaled for purposes of display.

2.9 Geometric Transformations

2.9.1 Filters You Should be Afraid to Use

2.9.2 Change Information Image Filter

This one is the scariest and most 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.

2.9.3 Flip Image Filter

The source code for this section can be found in the file `FlipImageFilter.cxx`.

The `itk::FlipImageFilter` is used for flipping the image content in any of the coordinate axes. This filter must be used with **EXTREME** caution. You probably don't want to appear in the newspapers as responsible for a surgery mistake in which a doctor extirpates the left kidney when he should have extracted the right one³. If that prospect doesn't scare you, maybe it is time for you to reconsider your career in medical image processing. Flipping effects which seem innocuous at first view may still have dangerous consequences. For example, flipping the cranio-caudal axis of a CT scan forces an observer to flip the left-right axis in order to make sense of the image.

³Wrong side surgery accounts for 2% of the reported medical errors in the United States. Trivial... but equally dangerous.

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 axes 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;  
  
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. Invoking `Update()` on any downstream filter will trigger the execution of the `FlipImage` filter.

```
filter->SetInput( reader->GetOutput() );  
writer->SetInput( filter->GetOutput() );  
writer->Update();
```

Figure 2.37 illustrates the effect of this filter on a slice of an MRI brain image using a flip array [0, 1] which means that the *Y* axis was flipped while the *X* axis was conserved.

2.9.4 Resample Image Filter

Introduction

The source code for this section can be found in the file `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

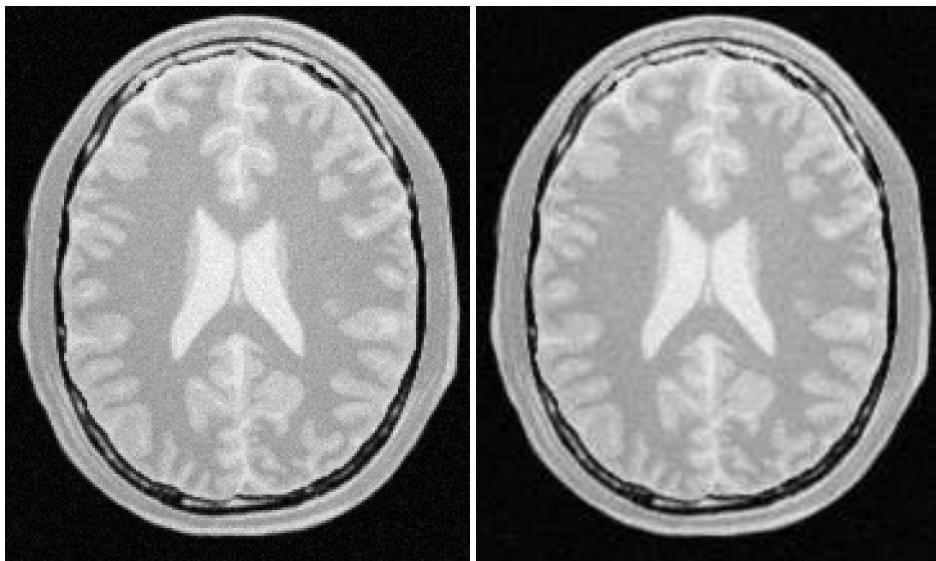


Figure 2.37: Effect of the `FlipImageFilter` on a slice from a MRI proton density brain image.

through the use of `itk::Transforms`. 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 the transform are 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.

```
// pixel spacing in millimeters along X and Y
const double spacing[ Dimension ] = { 1.0, 1.0 };
filter->SetOutputSpacing( spacing );

// Physical space coordinate of origin for X and Y
const double origin[ Dimension ] = { 0.0, 0.0 };
filter->SetOutputOrigin( origin );

InputImageType::DirectionType direction;
direction.SetIdentity();
filter->SetOutputDirection( direction );
```

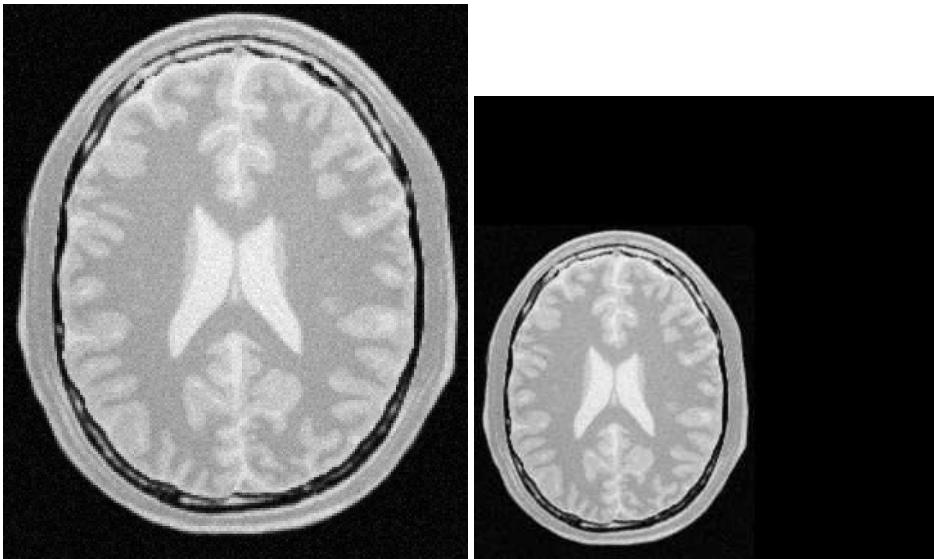


Figure 2.38: Effect of the resample filter.

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.

```
filter->SetInput( reader->GetOutput() );
writer->SetInput( filter->GetOutput() );
writer->Update();
```

Figure 2.38 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 2.38 is particularly misleading because the images are rescaled to fit nicely on the text of this book. Figure 2.39 clarifies the situation. It shows the two same images placed on an equally-scaled

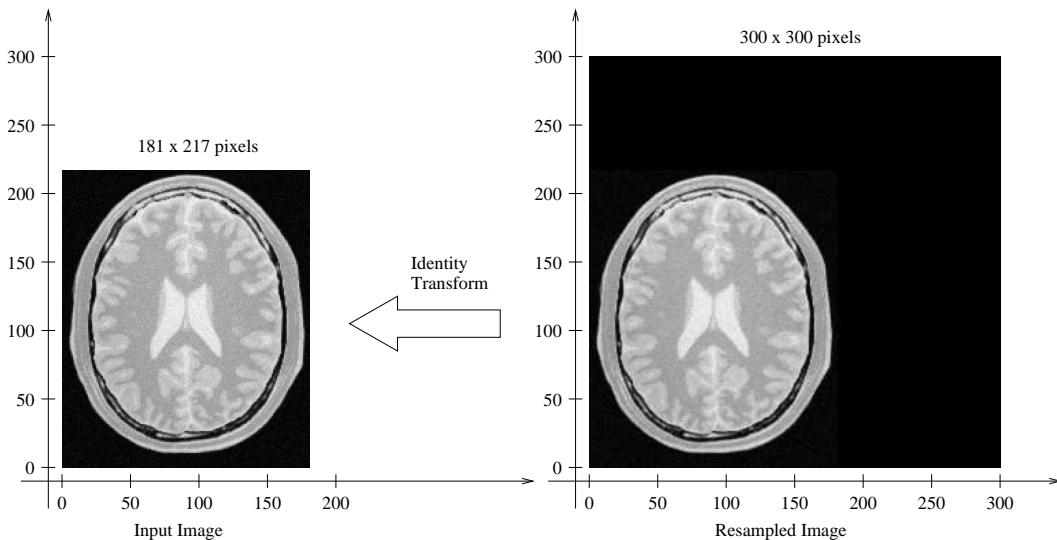


Figure 2.39: Analysis of the resample image done in a common coordinate system.

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 $1\text{mm} \times 1\text{mm}$ 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 );
```

The output image resulting from the translation can be seen in Figure 2.40. Again, it is better to interpret the result in a common coordinate system as illustrated in Figure 2.41.

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 2.41 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 2.41, 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

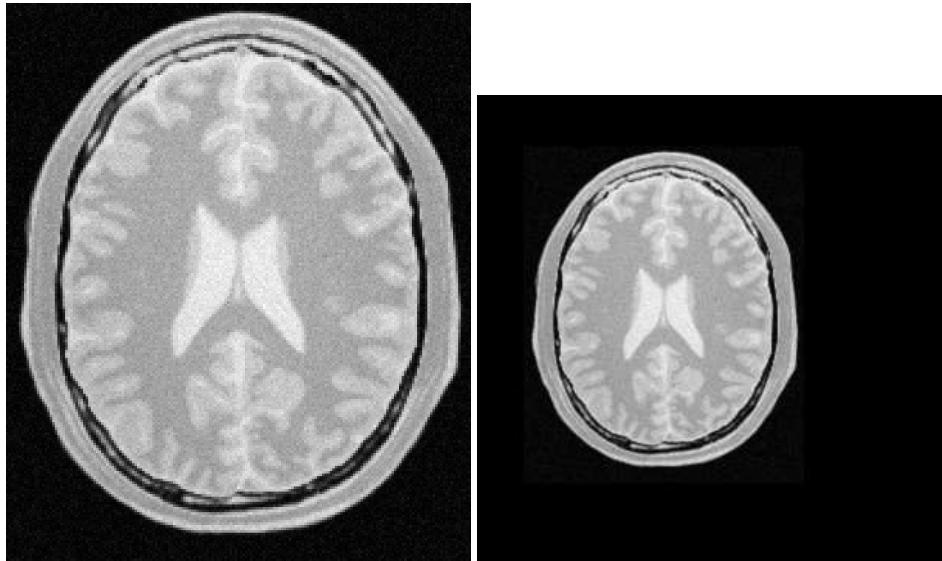


Figure 2.40: ResampleImageFilter with a translation by $(-30, -50)$.

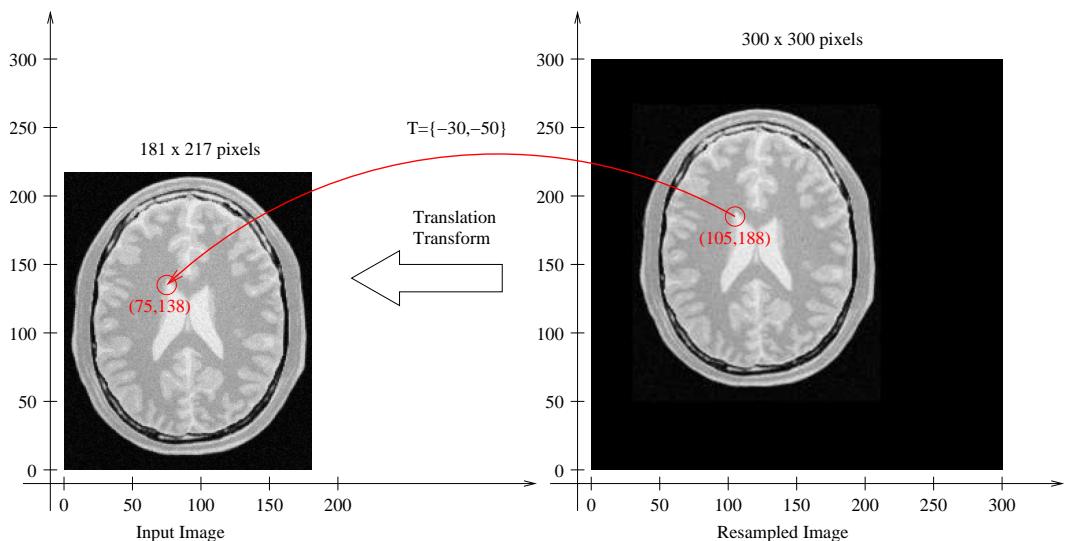


Figure 2.41: ResampleImageFilter. Analysis of a translation by $(-30, -50)$.

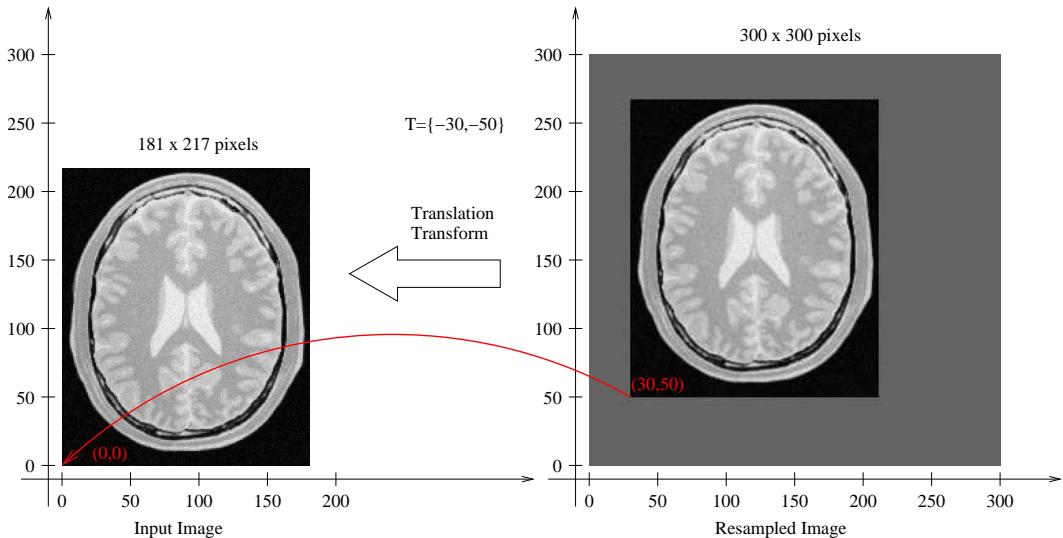


Figure 2.42: `ResampleImageFilter` highlighting image borders with `SetDefaultPixelValue()`.

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 2.42.

```
filter->SetDefaultPixelValue( 100 );
```

With this change we can better appreciate the effect of the previous translation transform on the image resampling. Figure 2.42 illustrates how the point (30,50) of the output image gets its gray value from the point (0,0) of the input image.

Importance of Spacing and Origin

The source code for this section can be found in the file `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

$$P[i] = I[i] \times S[i] + O[i] \quad (2.20)$$

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 2.43. 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 value distinct from the image background.

```
filter->SetDefaultPixelValue( 50 );
```

Let's set up a uniform spacing for the output image.

```
// pixel spacing in millimeters along X & Y
const double spacing[ Dimension ] = { 1.0, 1.0 };
filter->SetOutputSpacing( spacing );
```

We will preserve the orientation of the input image by using the following call.

```
filter->SetOutputDirection( reader->GetOutput()->GetDirection() );
```

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 $(0,0)$.

```
// space coordinate of origin
const double origin[ Dimension ] = { 30.0, 40.0 };
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 2.43.

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 2.44.

```
size[0] = 150; // number of pixels along X
size[1] = 200; // number of pixels along Y
filter->SetSize( size );
```

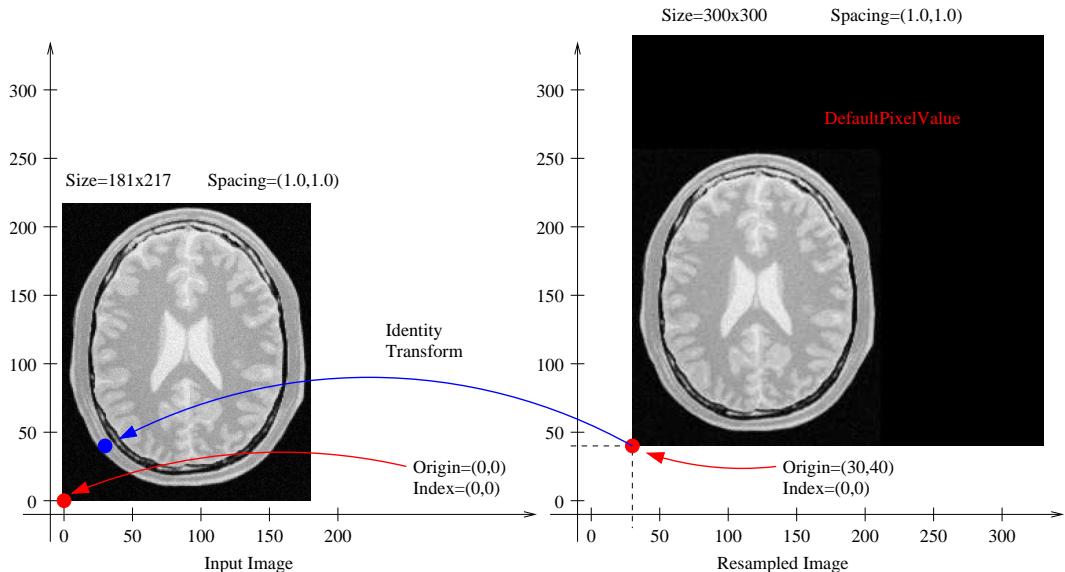


Figure 2.43: `ResampleImageFilter` selecting the origin of the output image.

```
// space coordinate of origin
const double origin[ Dimension ] = { 60.0, 30.0 };
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 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 2.45.

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 set the origin back to zero in both the input and output images.

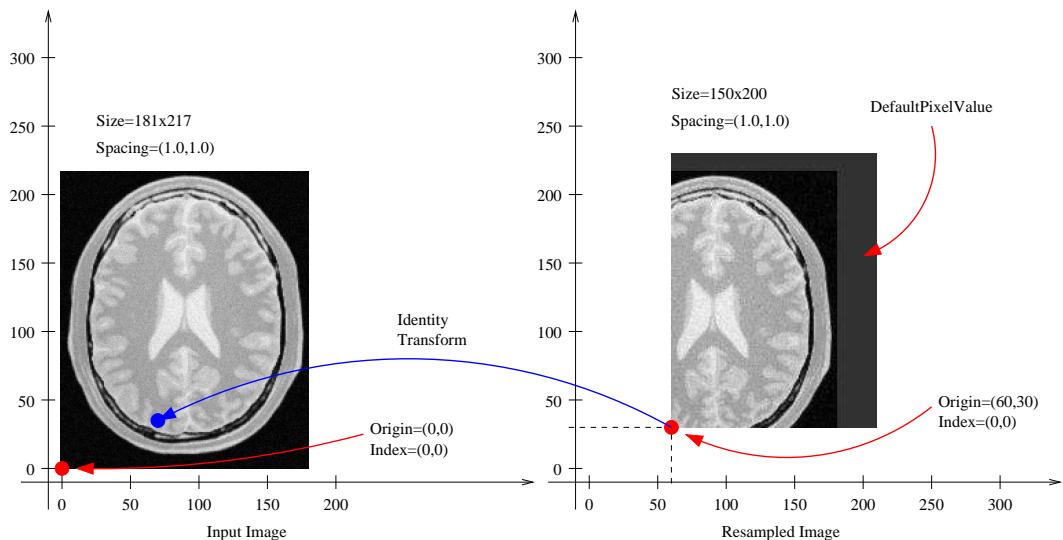


Figure 2.44: `ResampleImageFilter` origin in the output image.

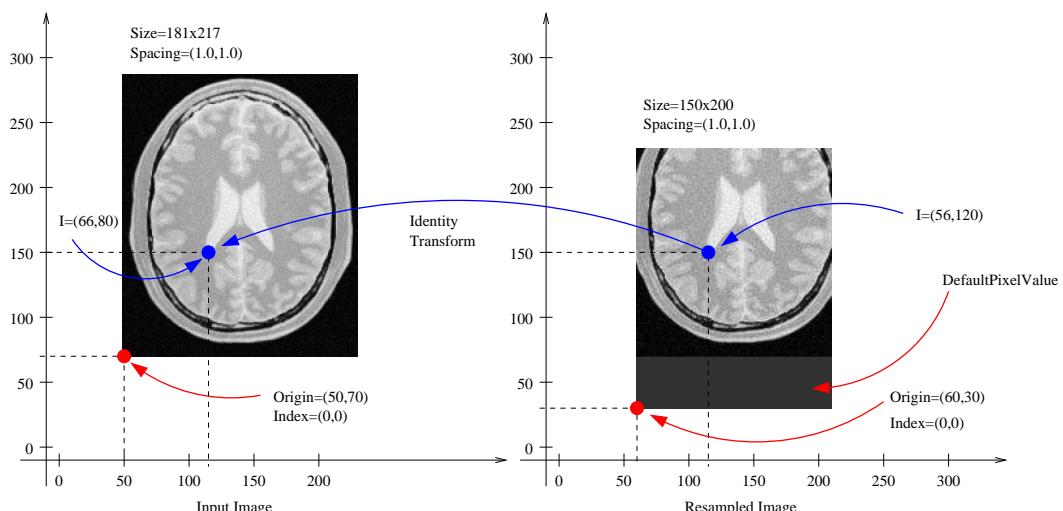


Figure 2.45: Effect of selecting the origin of the input image with `ResampleImageFilter`.

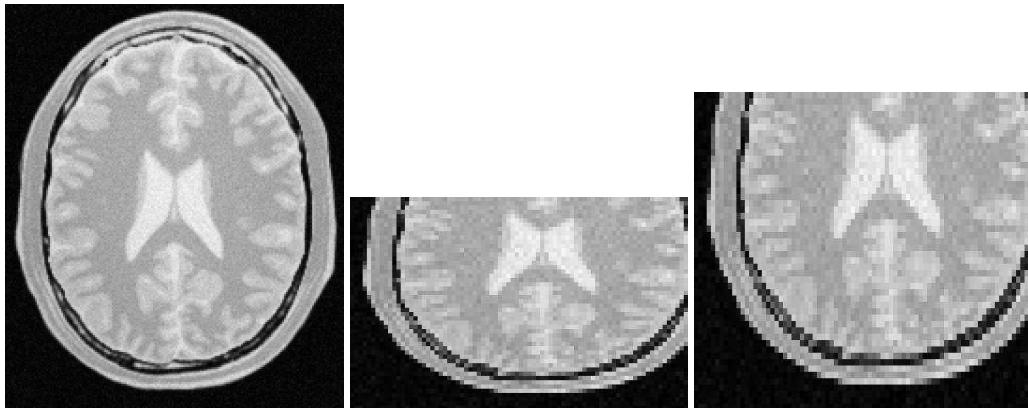


Figure 2.46: Resampling with different spacing seen by a naive viewer (center) and a correct viewer (right), input image (left).

```
// space coordinate of origin
const double origin[ Dimension ] = { 0.0, 0.0 };
filter->SetOutputOrigin( origin );
```

We then specify a non-unit spacing for the output image.

```
// pixel spacing in millimeters
const double spacing[ Dimension ] = { 2.0, 3.0 };
filter->SetOutputSpacing( spacing );
```

Additionally, we reduce the output image extent, since the new pixels are now covering a larger area of 2.0mm \times 3.0mm.

```
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 takes the spacing into account and appropriately scales the images on the screen. Please note that images in formats like PNG are not capable of representing origin and spacing. The toolkit assumes trivial default values for them. Figure 2.46 (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 figure⁴.

The filter output is analyzed in a common coordinate system with the input from Figure 2.47. In this figure, pixel $I = (33, 27)$ of the output image is located at coordinates $P = (66.0, 81.0)$ of the physical

⁴A viewer is provided with ITK under the name of MetaImageViewer. This viewer takes into account pixel spacing.

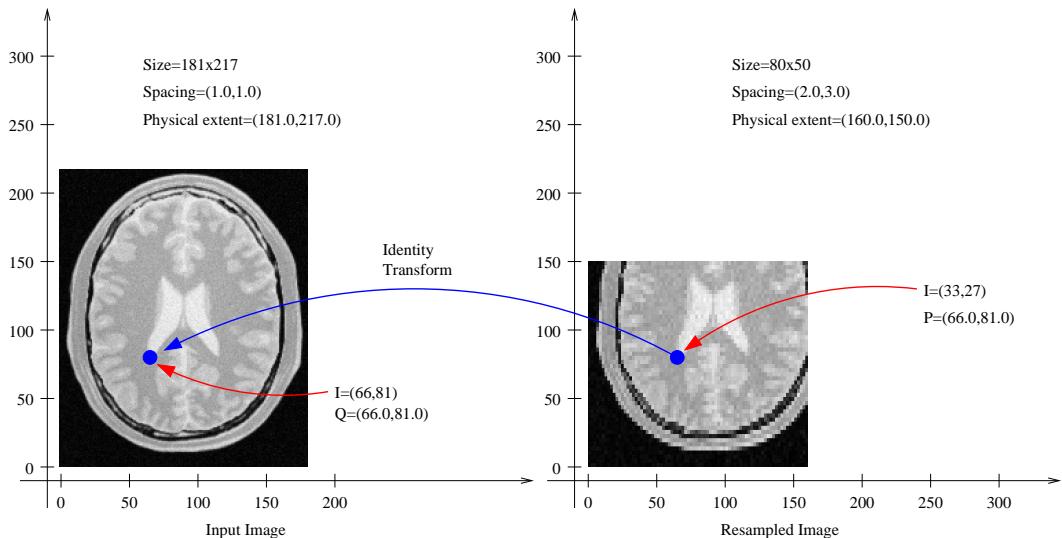


Figure 2.47: Effect of selecting the spacing on the output image.

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 2.43 to 2.47 has been resampled to have pixel spacing of $2\text{mm} \times 3\text{mm}$. The input image is presented in Figure 2.48 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 spacing image located at a non-zero origin. The comparison between input and output in a common reference system is presented in figure 2.49.

Here we start by selecting the origin of the output image.

```
// space coordinate of origin
const double origin[ Dimension ] = { 25.0, 35.0 };
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.

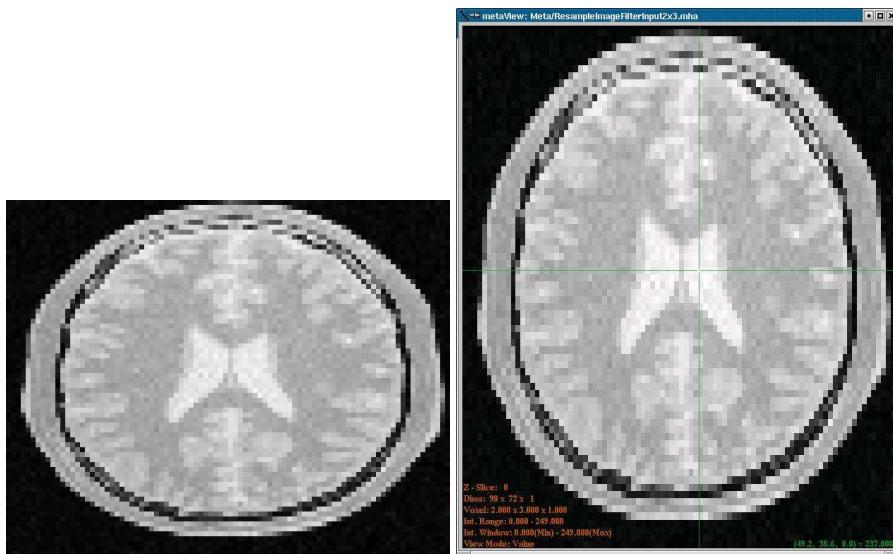


Figure 2.48: Input image with $2 \times 3\text{mm}$ spacing as seen with a naive viewer (left) and a correct viewer (right).

```
const double spacing[ Dimension ] = { 4.0, 4.5 };
filter->SetOutputSpacing( spacing );
```

Figure 2.49 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 a 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 2.49 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 `ResampleImageFilter3.cxx`.

Previous examples have described the basic principles behind the `itk::ResampleImageFilter`. Now it's time to have some fun with it.

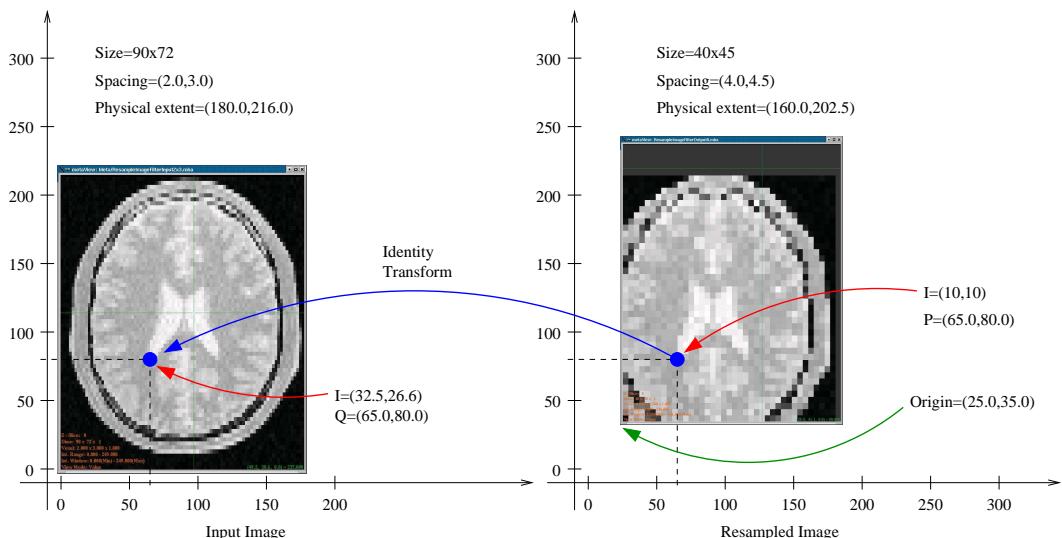


Figure 2.49: Effect of non-unit spacing on the input and output images.

Figure 2.51 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 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 2.51—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 2.50—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 (Colombian) coffee.

The following code implements the conditions illustrated in Figure 2.51 with two differences: the output spacing is 40 times smaller and there are 40 times more pixels in both dimensions. Without these changes, few details will be recognizable in 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 the actual content of the input image in any way.

In order to facilitate the interpretation of the transform we set the default pixel value to value be distinct from the image background.

```
filter->SetDefaultPixelValue( 100 );
```

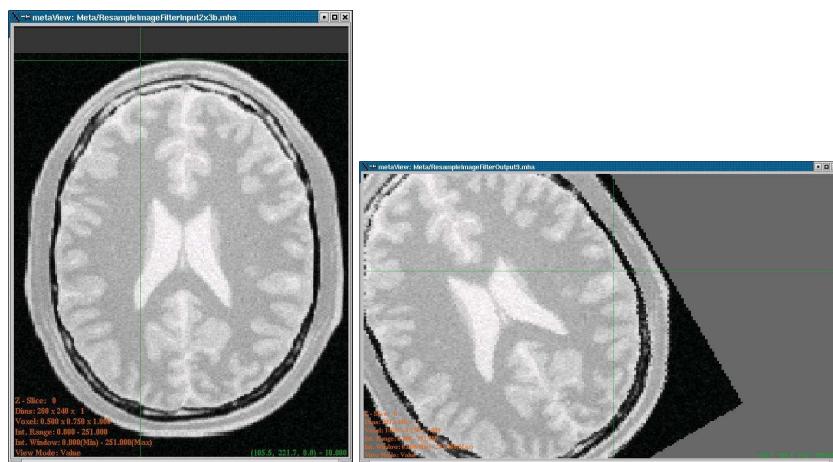


Figure 2.50: Effect of a rotation on the resampling filter. Input image at left, output image at right.

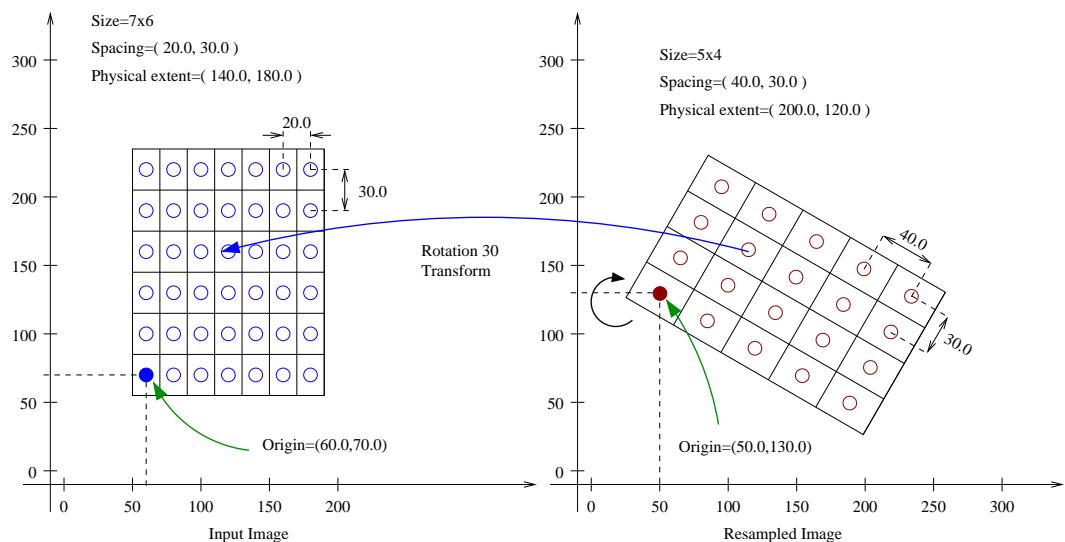


Figure 2.51: Input and output image placed in a common reference system.

The spacing is selected here to be 40 times smaller than the one illustrated in Figure 2.51.

```
double spacing[ Dimension ];
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 );
```

We will preserve the orientation of the input image by using the following call.

```
filter->SetOutputDirection( reader->GetOutput()->GetDirection() );
```

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 2.51.

```
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 2.51 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 = std::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 2.50 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 2.51 and Figure 2.50.

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 subtracts the origin, applies a 30 degrees rotation and adds the origin back. Let's follow those steps. Subtracting the origin from P leads to $P_1 = (40.0, 60.0)$, the rotation maps P_1 to $P_2 = (40.0 \times \cos(30.0) + 60.0 \times \sin(30.0), 40.0 \times \sin(30.0) - 60.0 \times \cos(30.0)) = (64.64, 31.96)$. Finally this point is translated back by the amount of the image origin. This moves P_2 to $P_3 = (114.64, 161.96)$.

The point P_3 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 2.51.

Rotating an Image

The source code for this section can be found in the file `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 `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 * inputImage = reader->GetOutput();

const InputImageType::SpacingType & spacing = inputImage->GetSpacing();
const InputImageType::PointType & origin = inputImage->GetOrigin();
InputImageType::SizeType size =
    inputImage->GetLargestPossibleRegion().GetSize();

filter->SetOutputOrigin( origin );
filter->SetOutputSpacing( spacing );
filter->SetOutputDirection( inputImage->GetDirection() );
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 2.52 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;

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 = std::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 );
```

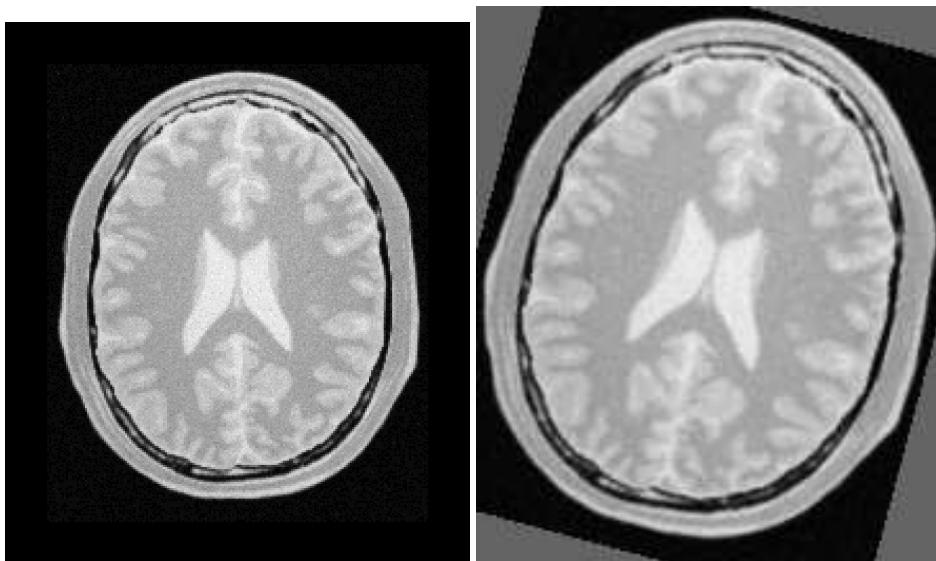


Figure 2.52: Effect of the resample filter rotating an image.

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 `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 $2D$.

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 = std::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 );
```

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 allows different combinations for initialization. In the `Similarity2DTransform` class the rotation and scaling will always be applied before the translation.

Figure 2.53 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°.

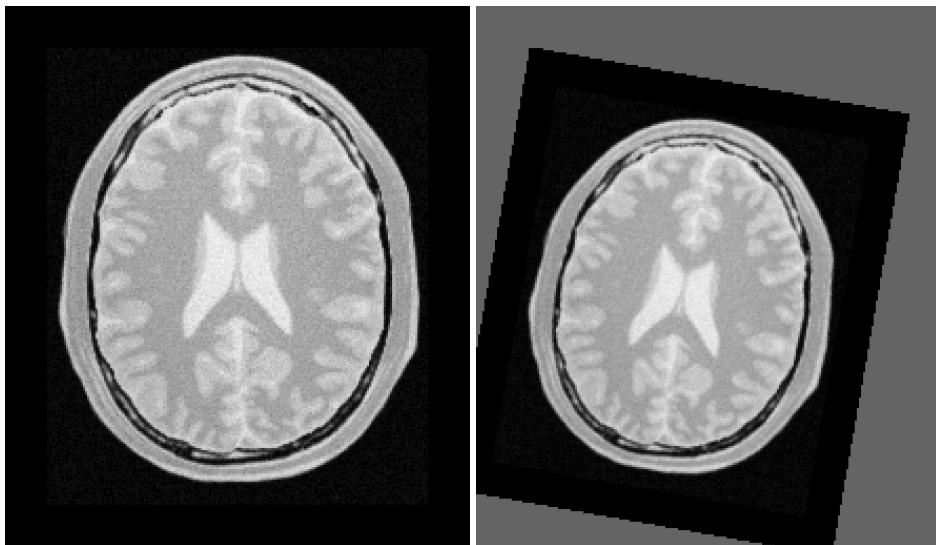


Figure 2.53: Effect of the resample filter rotating and scaling an image.

Resampling using a deformation field

The source code for this section can be found in the file `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"
```

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:

$$p_{in} = p_{out} + \text{distance} \quad (2.21)$$

```
typedef float VectorComponentType;
typedef itk::Vector< VectorComponentType, Dimension > VectorPixelType;
typedef itk::Image< VectorPixelType, Dimension > DisplacementFieldType;

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< DisplacementFieldType > FieldReaderType;
```

```

FieldReaderType::Pointer fieldReader = FieldReaderType::New();
fieldReader->SetFileName( argv[2] );
fieldReader->Update();

DisplacementFieldType::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,
    DisplacementFieldType > 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->SetOutputDirection( deformationField->GetDirection() );

filter->SetDisplacementField( deformationField );

```

Subsampling and image in the same space

The source code for this section can be found in the file `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 a 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 those corresponding to the resampling image filter, the transform, the interpolator and the smoothing filter.

```
#include "itkResampleImageFilter.h"
#include "itkIdentityTransform.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 are computed based on the pixel spacing 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 account 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 because 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 and direction of the input image are both preserved and passed to the output image.

```
resampler->SetOutputOrigin( inputImage->GetOrigin() );  
resampler->SetOutputDirection( inputImage->GetDirection() );
```

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 choose 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 `ResampleVolumesToBeIsotropic.cxx`.

It is unfortunate that it is still very common to find medical image datasets that have been acquired with large inter-slice 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. These datasets are close to **useless** for the purpose of computer-assisted image analysis. The abundance of datasets acquired with anisotropic voxel sizes bespeaks a dearth

of understanding of the third dimension and its importance for medical image analysis in clinical settings and radiology reading rooms. Datasets acquired with large anisotropies bring with them the regressive message: “*I do not think 3D is informative*”. They stubbornly insist: “*all that you need to know, can be known by looking at individual slices, one by one*”. However, the fallacy of this statement is made evident by simply viewing the slices when reconstructed in any of the orthogonal planes. The rectangular pixel shape is ugly and distorted, and cripples any signal processing algorithm not designed specifically for this type of image.

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 [58, 59].

Facing the inertia of many clinical imaging departments and their blithe insistence that these images are “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 an operation can be performed using the filters 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* which will only perpetuate the errant convictions of image acquisition departments. The real solution to the problem of the anisotropic dataset is to educate radiologists regarding the 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 to your radiologist why anisotropic data are problematic for processing, and require crude workarounds which handicap your ability to draw accurate conclusions from the data and preclude his or her ability to provide quality care. Any barbarity such as a [1 : 5] anisotropy ratio should be considered as a mere collection of slices, and not an authentic 3D dataset.

Please, before employing the techniques covered in this section, do kindly invite your fellow radiologist to see the dataset in an orthogonal slice. Magnify 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 it is to process digital data which have been sampled at ratios of [1 : 5] or [1 : 10]. Then, inform them that your only option is to throw away all that high in-plane resolution and to *make up* data between the slices in order to compensate for the low resolution. Only then will you be justified in using the following code.

Let’s now move into the code. It is appropriate for you to experience guilt⁵, because your 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

⁵A feeling of regret or remorse for having committed some improper act; a recognition of one’s own responsibility for doing something wrong.

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 [58, 59]. The smoothing is performed here using the RecursiveGaussian filter, because 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.

```
#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-off⁶.

```
#include "itkIdentityTransform.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. In practice you may want to remove this step of intensity rescaling.

```
#include "itkIntensityWindowingImageFilter.h"
```

We make 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;
```

⁶Although 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 we apply any technical sophistication here, when we are covering up for an improper acquisition of medical data, trying to make it look as if it was correctly acquired.

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 other than intuition and common sense. You can rightfully argue that this choice deserves a more technical consideration, but then, if you are so concerned about the technical integrity of the image sampling process, you should not be using this code, and should discuss these issues with 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 = std::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.

```
smootherX->SetDirection( 0 );
smootherY->SetDirection( 1 );
```

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 used as the output of this filter, then instantiate the image type and the type for the resampling filter. Finally we construct an instantiation of the 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, which 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 and orientation 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() );
resampler->SetOutputDirection( inputImage->GetDirection() );

```

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 computations 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 a moment in silence to reflect on the circumstances that have led us to accept this cover-up for the improper acquisition of medical data.

2.10 Frequency Domain

2.10.1 Computing a Fast Fourier Transform (FFT)

The source code for this section can be found in the file `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 [8, 9].

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 dele-

gate the actual computation of the transform to local available libraries. Particular examples of those libraries are fftw⁷ 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::ForwardFFTImageFilter`, and two of its derived classes are `itk::VnlForwardFFTImageFilter` and `itk::FFTWRealToComplexConjugateImageFilter`.

A typical application that uses FFT will need to include the following header files.

```
#include "itkImage.h"
#include "itkVnlForwardFFTImageFilter.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::VnlForwardFFTImageFilter`. 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::VnlForwardFFTImageFilter< ImageType > 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 an 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;
}
```

⁷<http://www.fftw.org>

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( argv[4] );

complexWriter->SetInput( fftFilter->GetOutput() );
```

Finally we invoke the `Update()` method placed 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 its first template parameter the type of the complex image and as its 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 is convenient to rescale the image in order to visualize it. For this purpose we instantiate 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( argv[4] );
complexReader->Update();
```

2.10.2 Filtering on the Frequency Domain

The source code for this section can be found in the file `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 process 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 input a complex image and produces a real image as output.

```
#include "itkVnlForwardFFTImageFilter.h"
#include "itkVnlInverseFFTImageFilter.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::VnlForwardFFTImageFilter` can be instantiated. Like most ITK filters, the FFT filter is instantiated using the full image type. By not setting the output image type, we decide to use the default one provided by the filter. Using this type we construct one instance of the filter.

```
typedef itk::VnlForwardFFTImageFilter< InputImageType > 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 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::VnlInverseFFTImageFilter<
    SpectralImageType > 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 an 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.

2.11 Extracting Surfaces

2.11.1 Surface extraction

The source code for this section can be found in the file `SurfaceExtraction.cxx`.

Surface extraction has attracted continuous interest since the early days of image analysis, especially in 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 [37]. Although it has been followed by a number of variants [55], Marching Cubes has become an icon in medical image processing. The following example illustrates how to perform surface extraction in ITK using an algorithm similar to Marching Cubes⁸.

The representation of unstructured data in ITK is done with the `itk::Mesh`. This class enables us 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"
```

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.

```
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

⁸Note that the Marching Cubes algorithm is covered by a patent that expired on June 5th 2005.

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 example, the pixel value associated with 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 define 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 placed 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;
}
```

We print out the number of nodes and cells in order to inspect the output mesh.

```
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.

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 3.1. 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.

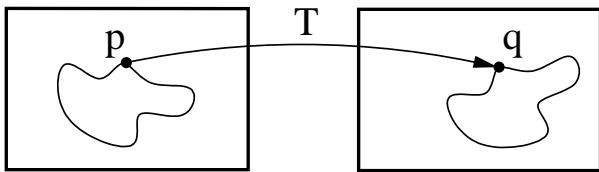


Figure 3.1: Image registration is the task of finding a spatial transform mapping one image into another.

3.1 Registration Framework

Let’s begin with a simplified typical registration framework where its components and their interconnections are shown in Figure 3.2. 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 a quantitative criterion to be optimized by the *optimizer* over the search space defined by the parameters of the *transform*.

ITKv4 registration framework provides more flexibility to the above traditional registration concept. In this new framework, the registration computations can happen on a physical grid completely different than the fixed image domain having different sampling density. This “sampling domain” is

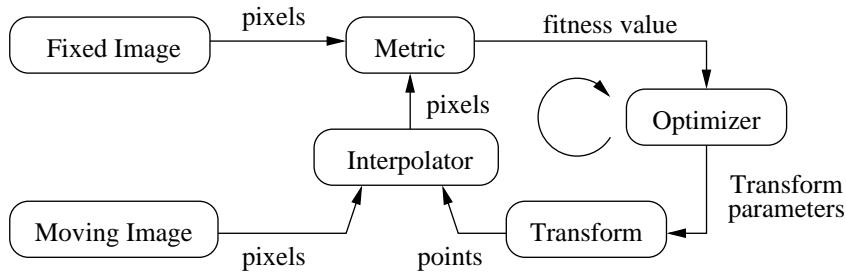


Figure 3.2: The basic components of a typical registration framework are two input images, a transform, a metric, an interpolator and an optimizer.

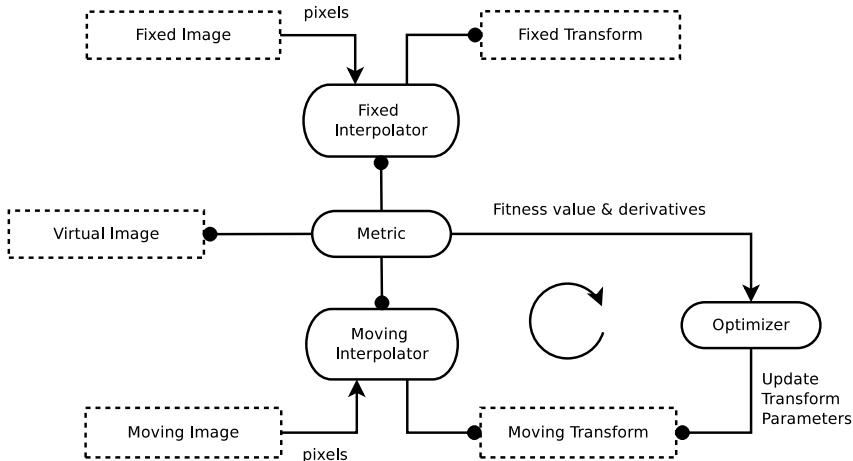


Figure 3.3: The basic components of the ITKv4 registration framework.

considered as a new component in the registration framework known as **virtual image** that can be an arbitrary set of physical points, not necessarily a uniform grid of points.

Various ITKv4 registration components are illustrated in Figure 3.3. Boxes with dashed borders show *data objects*, while those with solid borders show *process objects*.

The matching Metric class is a key component that controls most parts of the registration process since it handles fixed, moving and virtual images as well as fixed and moving transforms and interpolators.

Fixed and moving transforms and interpolators are used by the metric to evaluate the intensity values of the fixed and moving images at each physical point of the virtual space. Those intensity values are then used by the metric cost function to evaluate the fitness value and derivatives, which are passed to the optimizer that asks the moving transform to update its parameters based on the outputs of the cost function. Since the moving transform is shared between metric and optimizer, the above

process will be repeated till the convergence criteria are met.

Later in section 3.3 you will get a better understanding of the behind-the-scenes processes of ITKv4 registration framework. First, we begin with some simple registration examples.

3.2 "Hello World" Registration

The source code for this section can be found in the file `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. Instead of means to an end, this example should be read as a basic introduction to the elements typically involved when solving a problem of image registration.

A registration method requires the following set of components: two input images, a transform, a metric 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 "itkImageRegistrationMethodv4.h"
#include "itkTranslationTransform.h"
#include "itkMeanSquaresImageToImageMetricv4.h"
#include "itkRegularStepGradientDescentOptimizerv4.h"
```

The type of each registration component should be instantiated first. We start by selecting the image dimension and the types to be 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::RegularStepGradientDescentOptimizerv4<double> OptimizerType;
```

The metric will compare how well the two images match each other. Metric types are usually templated over the image types as seen in the following type declaration.

```
typedef itk::MeanSquaresImageToImageMetricv4<
    FixedImageType,
    MovingImageType >    MetricType;
```

The registration method type is instantiated using the types of the fixed and moving images as well as the output transform type. This class is responsible for interconnecting all the components that we have described so far.

```
typedef itk::ImageRegistrationMethodv4<
    FixedImageType,
    MovingImageType,
    TransformType >    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();
OptimizerType::Pointer   optimizer   = OptimizerType::New();
RegistrationType::Pointer registration = RegistrationType::New();
```

Each component is now connected to the instance of the registration method.

```
registration->SetMetric(      metric      );
registration->SetOptimizer(   optimizer   );
```

In this example the transform object does not need to be created and passed to the registration method like above since the registration filter will instantiate an internal transform object using the transform type that is passed to it as a template parameter.

Metric needs an interpolator to evaluate the intensities of the fixed and moving images at non-grid positions. The types of fixed and moving interpolators are declared here.

```
typedef itk::LinearInterpolateImageFunction<
    FixedImageType,
    double > FixedLinearInterpolatorType;

typedef itk::LinearInterpolateImageFunction<
    MovingImageType,
    double > MovingLinearInterpolatorType;
```

Then, fixed and moving interpolators are created and passed to the metric. Since linear interpolators are used as default, we could skip the following step in this example.

```
FixedLinearInterpolatorType::Pointer fixedInterpolator =
    FixedLinearInterpolatorType::New();
MovingLinearInterpolatorType::Pointer movingInterpolator =
    MovingLinearInterpolatorType::New();

metric->SetFixedInterpolator( fixedInterpolator );
metric->SetMovingInterpolator( movingInterpolator );
```

In this example, the fixed and moving images are read from files. This requires the

`itk::ImageRegistrationMethodv4` to acquire its inputs from the output of the readers.

```
registration->SetFixedImage( fixedImageReader->GetOutput() );
registration->SetMovingImage( movingImageReader->GetOutput() );
```

Now the registration process should be initialized. ITKv4 registration framework provides initial transforms for both fixed and moving images. These transforms can be used to setup an initial known correction of the misalignment between the virtual domain and fixed/moving image spaces. In this particular case, a translation transform is being used for initialization of the moving image space. The array of parameters for the initial moving 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.

```
TransformType::Pointer movingInitialTransform = TransformType::New();

TransformType::ParametersType initialParameters(
    movingInitialTransform->GetNumberOfParameters() );
initialParameters[0] = 0.0; // Initial offset in mm along X
initialParameters[1] = 0.0; // Initial offset in mm along Y

movingInitialTransform->SetParameters( initialParameters );

registration->SetMovingInitialTransform( movingInitialTransform );
```

In the registration filter this moving initial transform will be added to a composite transform that already includes an instantiation of the output optimizable transform; then, the resultant composite transform will be used by the optimizer to evaluate the metric values at each iteration.

Despite this, the fixed initial transform does not contribute to the optimization process. It is only used to access the fixed image from the virtual image space where the metric evaluation happens.

Virtual images are a new concept added to the ITKv4 registration framework, which potentially lets us to do the registration process in a physical domain totally different from the fixed and moving image domains. In fact, the region over which metric evaluation is performed is called virtual image domain. This domain defines the resolution at which the evaluation is performed, as well as the physical coordinate system.

The virtual reference domain is taken from the “virtual image” buffered region, and the input images should be accessed from this reference space using the fixed and moving initial transforms.

The legacy intuitive registration framework can be considered as a special case where the virtual domain is the same as the fixed image domain. As this case practically happens in most of the real life applications, the virtual image is set to be the same as the fixed image by default. However, the user can define the virtual domain differently than the fixed image domain by calling either `SetVirtualDomain` or `SetVirtualDomainFromImage`.

In this example, like the most examples of this chapter, the virtual image is considered the same as the fixed image. Since the registration process happens in the fixed image physical domain, the fixed

initial transform maintains its default value of identity and does not need to be set.

However, a “Hello World!” example should show all the basics, so all the registration components are explicitly set here.

In the next section of this chapter, you will get a better understanding from behind the scenes of the registration process when the initial fixed transform is not identity.

```
TransformType::Pointer identityTransform = TransformType::New();
identityTransform->SetIdentity();

registration->SetFixedInitialTransform( identityTransform );
```

Note that the above process shows only one way of initializing the registration configuration. Another option is to initialize the output optimizable transform directly. In this approach, a transform object is created, initialized, and then passed to the registration method via `SetInitialTransform()`. This approach is shown in section 3.6.1.

At this point the registration method is ready for execution. The optimizer is the component that drives the execution of the registration. However, the `ImageRegistrationMethodv4` 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::ImageToImageMetricv4` derivative. 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 relaxation factor. The reducing factor should have a value between 0 and 1. This factor is set to 0.5 by default, and it can be changed to a different value via `SetRelaxationFactor()`. Also, the default value for the initial step length is 1, and this value can be changed manually with the method `SetLearningRate()`.

In addition to manual settings, the initial step size can also be estimated automatically, either at each iteration or only at the first iteration, by assigning a `ScalesEstimator` (as will be seen in later examples).

After several reductions of the step length, the optimizer may be moving in a very restricted area of the transform parameter space. By the method `SetMinimumStepLength()`, 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. User can also set some other stop criteria manually like maximum number of iterations.

In other gradient descent-based optimizers of the ITKv4 framework, such as `itk::GradientDescentLineSearchOptimizerv4` and `itk::ConjugateGradientLineSearchOptimizerv4`, the convergence criteria are set via `SetMinimumConvergenceValue()` which is computed based on the results of the last few iterations. The number of iterations involved in computations are defined by the convergence window

size via `SetConvergenceWindowSize()` which is shown in later examples of this chapter.

Also note that unlike the previous versions, ITKv4 optimizers do not have a “maximize/minimize” option to modify the effect of the metric derivatives. Each assigned metric is assumed to return a parameter derivative result that “improves” the optimization.

```
optimizer->SetLearningRate( 4 );
optimizer->SetMinimumStepLength( 0.001 );
optimizer->SetRelaxationFactor( 0.5 );
```

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 );
```

ITKv4 facilitates a multi-level registration framework whereby each stage is different in the resolution of its virtual space and the smoothness of the fixed and moving images. These criteria need to be defined before registration starts. Otherwise, the default values will be used. In this example, we run a simple registration in one level with no space shrinking or smoothing on the input data.

```
const unsigned int numberOfLevels = 1;

RegistrationType::ShrinkFactorsArrayType shrinkFactorsPerLevel;
shrinkFactorsPerLevel.SetSize( 1 );
shrinkFactorsPerLevel[0] = 1;

RegistrationType::SmoothingSigmasArrayType smoothingSigmasPerLevel;
smoothingSigmasPerLevel.SetSize( 1 );
smoothingSigmasPerLevel[0] = 0;

registration->SetNumberOfLevels ( numberOfLevels );
registration->SetSmoothingSigmasPerLevel( smoothingSigmasPerLevel );
registration->SetShrinkFactorsPerLevel( shrinkFactorsPerLevel );
```

The registration process is triggered by an invocation of 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();
    std::cout << "Optimizer stop condition: "
    << registration->GetOptimizer()->GetStopConditionDescription()
    << std::endl;
}
catch( itk::ExceptionObject & err )
{
    std::cerr << "ExceptionObject caught !" << std::endl;
    std::cerr << err << std::endl;
    return EXIT_FAILURE;
}

```

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 obtained using the `GetTransform()` method that returns a constant pointer to the output transform.

```
TransformType::ConstPointer transform = registration->GetTransform();
```

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.

```

TransformType::ParametersType finalParameters = transform->GetParameters();
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 learning rate has been set too small, which is undesirable since it results in long computational times.

```
const unsigned int numberIterations = 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.

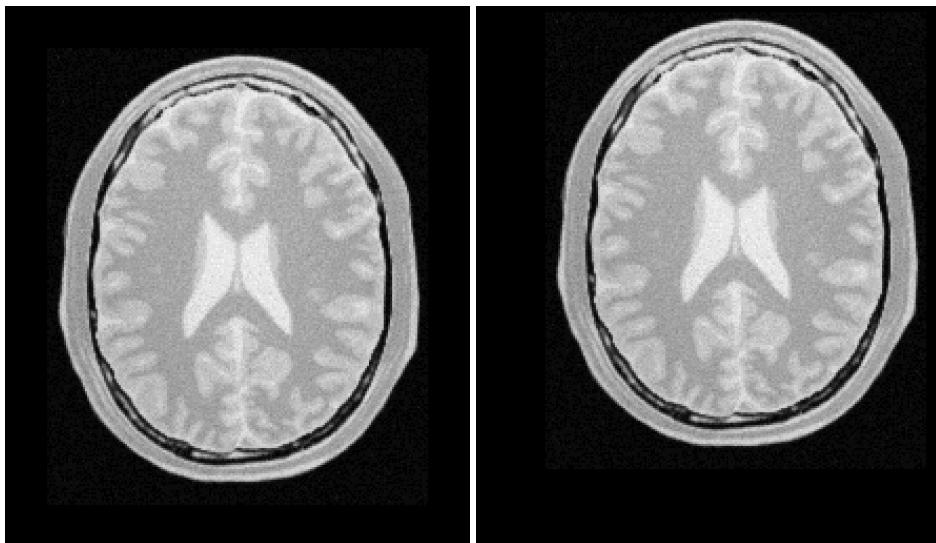


Figure 3.4: Fixed and Moving image provided as input to the registration method.

Both images have unit-spacing and are shown in Figure 3.4. The registration takes 20 iterations and the resulting transform parameters are:

```
Translation X = 13.0012
Translation Y = 16.9999
```

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.

Before the mapping process, notice that we have not used the direct initialization of the output transform in this example, so the parameters of the moving initial transform are not reflected in the output parameters of the registration filter. Hence, a composite transform is needed to concatenate both initial and output transforms together.

```
typedef itk::CompositeTransform<
    double,
    Dimension > CompositeTransformType;
CompositeTransformType::Pointer outputCompositeTransform =
    CompositeTransformType::New();
outputCompositeTransform->AddTransform( movingInitialTransform );
outputCompositeTransform->AddTransform(
    registration->GetModifiableTransform() );
```

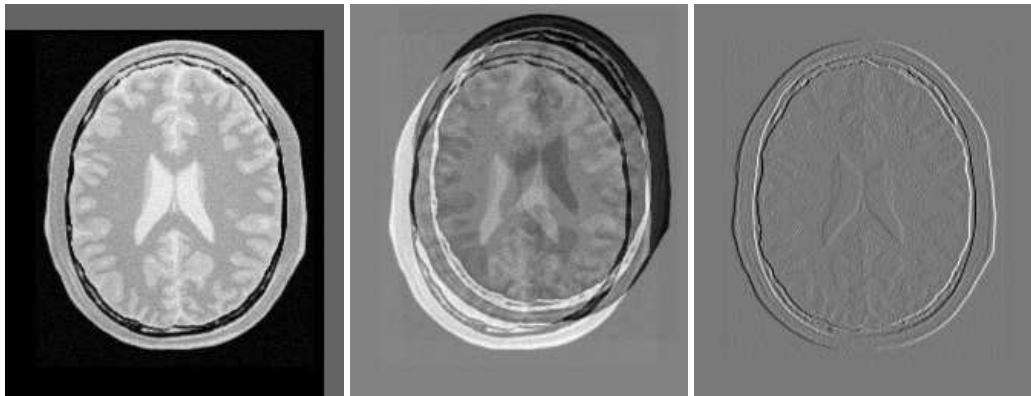


Figure 3.5: Mapped moving image and its difference with the fixed image before and after registration

Now the mapping process is easily done with the `itk::ResampleImageFilter`. Please refer to Section 2.9.4 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.

```
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 created output composite transform is also passed as input to the resampling filter.

```
resampler->SetTransform( outputCompositeTransform );
```

As described in Section 2.9.4, 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() );
resampler->SetOutputOrigin( fixedImage->GetOrigin() );
resampler->SetOutputSpacing( fixedImage->GetSpacing() );
resampler->SetOutputDirection( fixedImage->GetDirection() );
resampler->SetDefaultPixelValue( 100 );
```

The output of the filter is passed to a writer that will store the image in a file. An

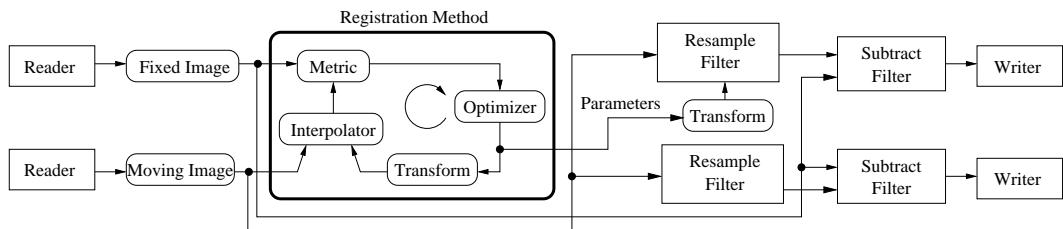


Figure 3.6: Pipeline structure of the registration example.

`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.

```

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 types.

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 it possible to visualize the negative values even if we save the difference image in a file format that only supports unsigned pixel values¹. 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.

```
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.

```
resampler->SetTransform( identityTransform );
```

The complete pipeline structure of the current example is presented in Figure 3.6. The components of the registration method are depicted as well. Figure 3.5 (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 (i.e. the difference before the registration is performed). The right image shows the difference between the fixed image and the transformed moving image (i.e. 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

¹ This is the case of PNG, BMP, JPEG and TIFF among other common file formats.

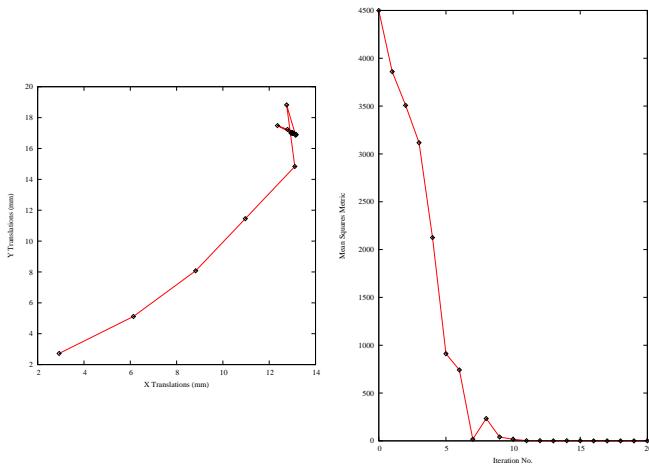


Figure 3.7: The sequence of translations and metric values at each iteration of the optimizer.

causes 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 3.7 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.

In this section, we used a very simple example to introduce the basic components of a registration process in ITKv4. However, studying this example alone is not enough to start using the `itk::ImageRegistrationMethodv4`. In order to choose the best registration practice for a specific application, knowledge of other registration method instantiations and their capabilities are required. For example, direct initialization of the output optimizable transform is shown in section 3.6.1. This method can simplify the registration process in many cases. Also, multi-resolution and multistage registration approaches are illustrated in sections 3.7 and 3.8. These examples illustrate the flexibility in the usage of ITKv4 registration method framework that can help to provide faster and more reliable registration processes.

3.3 Features of the Registration Framework

This section presents internals of the registration process in ITKv4. Understanding what actually happens is necessary to have a correct interpretation of the results of a registration filter. It also helps to understand the most common difficulties that users encounter when they start using the

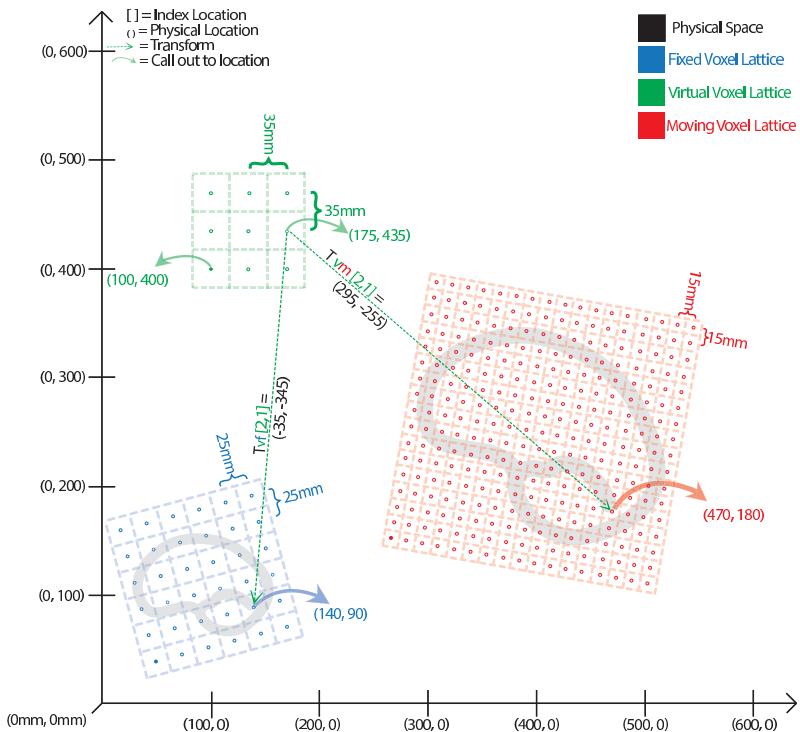


Figure 3.8: Different coordinate systems involved in the image registration process. Note that the transform being optimized is the one mapping from the physical space of the **virtual** image into the physical space of the **moving** image.

ITKv4 registration framework:

- Registration is done in physical coordinates
- The direction of the transform maps from the space of the virtual image to that of the moving image

These two topics tend to create confusion because they are implemented in different ways in other systems, and community members tend to have different expectations regarding how registration should work in ITKv4. The situation is further complicated by the way most people describe image operations, as if they were manually performed on a continuous picture on a piece of paper.

These concepts are discussed in this section through a general example shown in Figure 3.8.

Recall that ITKv4 does the registration in “physical” space where fixed, moving and virtual images are placed. Also, note that the term of virtual image is deceptive here since it does not refer to

any actual image. In fact, the virtual image defines the origin, direction and the spacing of a space lattice that holds the output resampled image of the registration process. The virtual pixel lattice is illustrated in green at the top left side of Figure 3.8.

As shown in this figure, generally there are two transforms involved in the registration process even though only one of them is being optimized. T_{vm} maps points from physical virtual space onto the physical space of the moving image, and in the same way T_{vf} finds homologous points between physical virtual space and the physical space of the fixed image. Note that only T_{vm} is optimized during the registration process. T_{vf} cannot be optimized. The fixed transform usually is an identity transform since the virtual image lattice is commonly defined as the fixed image lattice.

When the registration starts, the algorithm goes through each grid point of the virtual lattice in a raster sweep. At each point the fixed and moving transforms find coordinates of the homologous points in the fixed and moving image physical spaces, and interpolators are used to find the pixel intensities if mapped points are in non-grid positions. These intensity values are passed to a cost function to find the current metric value.

Note the direction of the mapping transforms here. For example, if you consider the T_{vm} transform, confusion often occurs since the transform shifts a virtual lattice 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 T_{vm} transform applies a **clock-wise** rotation to the virtual space 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 same relationships also occur with the T_{vf} transform between the virtual space and the fixed image space.

This mapping direction is chosen because the moving image is resampled on the grid of the virtual image. In the resampling process, an algorithm iterates through every pixel of the output image and computes the intensity assigned to this pixel by mapping to its location in the moving image.

Instead, if we were to use the transform mapping coordinates from the moving image physical space into the virtual image physical space, then the resampling process would not guarantee that every pixel in the grid of the virtual image would receive one and only one value. In other words, the resampling would result in an image with holes and redundant or overlapping pixel values.

As seen in the previous examples, and as corroborated in the remaining examples in this chapter, the transform computed by the registration framework can be used directly in the resampling filter in order to map the moving image onto the discrete grid of the virtual image.

There are exceptional cases in which the transform desired is actually the inverse transform of the one computed by the ITK registration framework. Only those cases may require invoking the `GetInverse()` method that most transforms offer. Before attempting this, read the examples on resampling illustrated in section 2.9 in order to familiarize yourself with the correct interpretation of the transforms.

Now we come back to the situation illustrated in Figure 3.8. This figure shows the flexibility of the ITKv4 registration framework. We can register two images with different scales, sizes and resolutions. Also, we can create the output warped image with any desired size and resolution.

Nevertheless, note that the spatial transform computed during the registration process does not need to be concerned about a different number of pixels and different pixel sizes between fixed, moving and output images because the conversion from index space to the physical space implicitly takes care of the required scaling factor between the involved images.

One important consequence of this fact is that having the correct image origin, image pixel size, and image direction is fundamental for the success of the registration process in ITK, since we need this information to compute the exact location of each pixel lattice in the physical space; we must make sure that the correct values for the origin, spacing, and direction of all fixed, moving and virtual images are provided.

In this example, the spatial transform computed will **physically** map the brain from the moving image onto the virtual space and minimize its difference with the resampled brain from the fixed image into the virtual space. Fortunately in practice there is no need to resample the fixed image since the virtual image physical domain is often assumed to be the same as physical domain of the fixed image.

3.4 Monitoring Registration

The source code for this section can be found in the file
`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 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 `ImageRegistrationMethodv4` class.

`Insight` implements the *Observer/Command* design pattern [21]. 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 facilitates 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::RegularStepGradientDescentOptimizerv4<double> 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) ITK_OVERRIDE
{
    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) ITK_OVERRIDE
{
```

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 =
    static_cast< OptimizerPointer >( object );
```

The next step is to verify that the event invoked is actually the one in which we are interested. This is checked using the RTTI² 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 registra-

²RTTI stands for: Run-Time Type Information

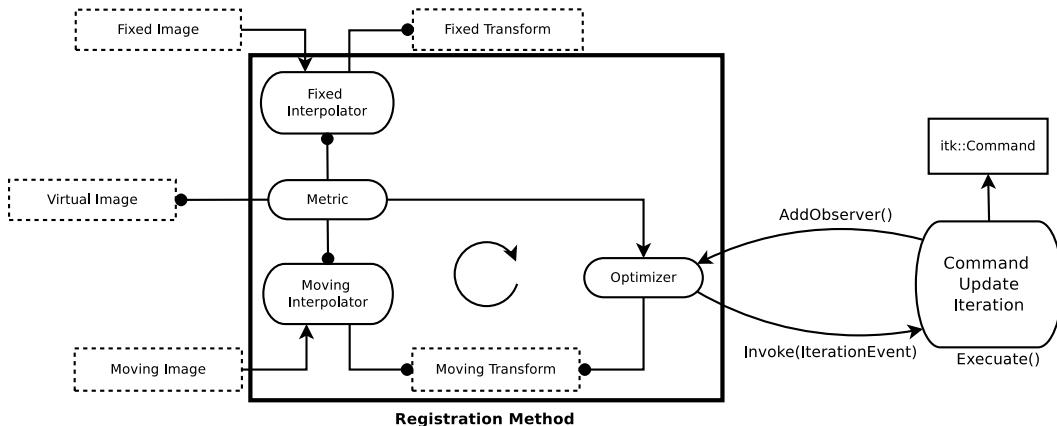


Figure 3.9: Interaction between the Command/Observer and the Registration Method.

tion 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();
```

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 observer. Figure 3.9 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 `Update()` will do it. Note again the use of the `try/catch` block around the `Update()` method in case an exception is thrown.

```
try
{
    registration->Update();
    std::cout << "Optimizer stop condition: "
          << registration->GetOptimizer()->GetStopConditionDescription()
          << std::endl;
}
catch( itk::ExceptionObject & err )
{
    std::cout << "ExceptionObject caught !" << std::endl;
    std::cout << err << std::endl;
    return EXIT_FAILURE;
}
```

The registration process is applied to the following images in Examples/Data:

- BrainProtonDensitySliceBorder20.png
- BrainProtonDensitySliceShifted13x17y.png

It produces the following output.

```
0 = 4499.45 : [2.9286959512455857, 2.7244705953923805]
1 = 3860.84 : [6.135143776902402, 5.115849348610004]
2 = 3508.02 : [8.822660051952475, 8.078492808653918]
3 = 3117.31 : [10.968558473732326, 11.454158663474674]
4 = 2125.43 : [13.105290365964755, 14.835634202454191]
5 = 911.308 : [12.75173580401588, 18.819978461140323]
6 = 741.417 : [13.139053510563274, 16.857840597942413]
7 = 16.8918 : [12.356787624301035, 17.480785285045815]
8 = 233.714 : [12.79212443526829, 17.234854683011704]
9 = 39.8027 : [13.167510875734614, 16.904574468172815]
10 = 16.5731 : [12.938831371165355, 17.005597654570586]
11 = 1.68763 : [13.063495692092735, 16.996443033457986]
12 = 1.79437 : [13.001061362657559, 16.999307384689935]
13 = 0.000762481 : [12.945418587211314, 17.0277701944711]
14 = 1.74802 : [12.974454390534774, 17.01621663980765]
15 = 0.430253 : [13.002439510423766, 17.002309966416835]
16 = 0.00531816 : [12.989877586882951, 16.99301810428082]
17 = 0.0721346 : [12.996759235073881, 16.996716492365685]
18 = 0.00996773 : [13.00288423694971, 17.00156618393022]
19 = 0.00516378 : [12.99928608126834, 17.000045636412015]
20 = 0.000228075 : [13.00123653240422, 16.999943471681494]
```

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.

3.5 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 [12, 38, 64]; a more detailed review can be found in [24, 47]. The Insight Toolkit currently provides two different implementations of Mutual Information metrics (see section 3.11 for details). The following example illustrates the practical use of one of these metrics.

3.5.1 Mattes Mutual Information

The source code for this section can be found in the file `ImageRegistration4.cxx`.

In this example, we will solve a simple multi-modality problem using an implementation of mutual information. This implementation was published by Mattes *et. al* [41].

First, we include the header files of the components used in this example.

```
#include "itkImageRegistrationMethodv4.h"
#include "itkTranslationTransform.h"
#include "itkMattesMutualInformationImageToImageMetricv4.h"
#include "itkRegularStepGradientDescentOptimizerv4.h"
```

In this example the image types and all registration components, except the metric, are declared as in Section 3.2. The Mattes mutual information metric type is instantiated using the image types.

```
typedef itk::MattesMutualInformationImageToImageMetricv4<
    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 the user to specify the number of bins used to compute the entropy. In a typical application, 50 histogram bins are sufficient. Note however, that the number of bins may have dramatic effects on the optimizer's behavior.

```
unsigned int numberOfBins = 24;
metric->SetNumberOfHistogramBins( numberOfBins );
```

To calculate the image gradients, an image gradient calculator based on ImageFunction is used instead of image gradient filters. Image gradient methods are defined in the superclass ImageToImageMetricv4.

```
metric->SetUseMovingImageGradientFilter( false );
metric->SetUseFixedImageGradientFilter( false );
```

Notice that in the ITKv4 registration framework, optimizers always try to minimize the cost function, and the metrics always return a parameter and derivative result that improves the optimization, so this metric computes the negative mutual information. The optimization parameters are tuned for this example, so they are not exactly the same as the parameters used in Section 3.2.

```
optimizer->SetLearningRate( 8.00 );
optimizer->SetMinimumStepLength( 0.001 );
optimizer->SetNumberOfIterations( 200 );
optimizer->ReturnBestParametersAndValueOn();
```

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 {1.0,5.0}. 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.

Whenever the regular step gradient descent optimizer encounters change in the direction of movement 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 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 );
```

Instead of using the whole virtual domain (usually fixed image domain) for the registration, we can use a spatial sampled point set by supplying an arbitrary point list over which to evaluate the metric. The point list is expected to be in the *fixed* image domain, and the points are transformed into the *virtual* domain internally as needed. The user can define the point set via SetFixedSampledPointSet(), and the point set is used by calling SetUsedFixedSampledPointSet().

Also, instead of dealing with the metric directly, the user may define the sampling percentage and sampling strategy for the registration framework at each level. In this case, the registration filter manages the sampling operation over the fixed image space based on the input strategy (REGULAR, RANDOM) and passes the sampled point set to the metric internally.

```
RegistrationType::MetricSamplingStrategyType samplingStrategy =
    RegistrationType::RANDOM;
```

The number of spatial samples to be used depends on the content of the image. If the images are smooth and do not contain many details, the number of spatial samples can usually be as low as 1% of the total number of pixels in the fixed image. On the other hand, if the images are detailed, it may be necessary to use a much higher proportion, such as 20% to 50%. Increasing the number of samples improves the smoothness of the metric, 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 results in longer computation times per every evaluation of the metric.

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 `SetUseFixedSampledPointSet(false)` 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.

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 3.4.

```
double samplingPercentage = 0.20;
```

In ITKv4, a single virtual domain or spatial sample point set is used for the all iterations of the registration process. The use of a single sample set results in a smooth cost function that can improve the functionality of the optimizer.

```
registration->SetMetricSamplingStrategy( samplingStrategy );
registration->SetMetricSamplingPercentage( samplingPercentage );
```

Let's execute this example over two of the images provided in Examples/Data:

- BrainT1SliceBorder20.png
- BrainProtonDensitySliceShifted13x17y.png

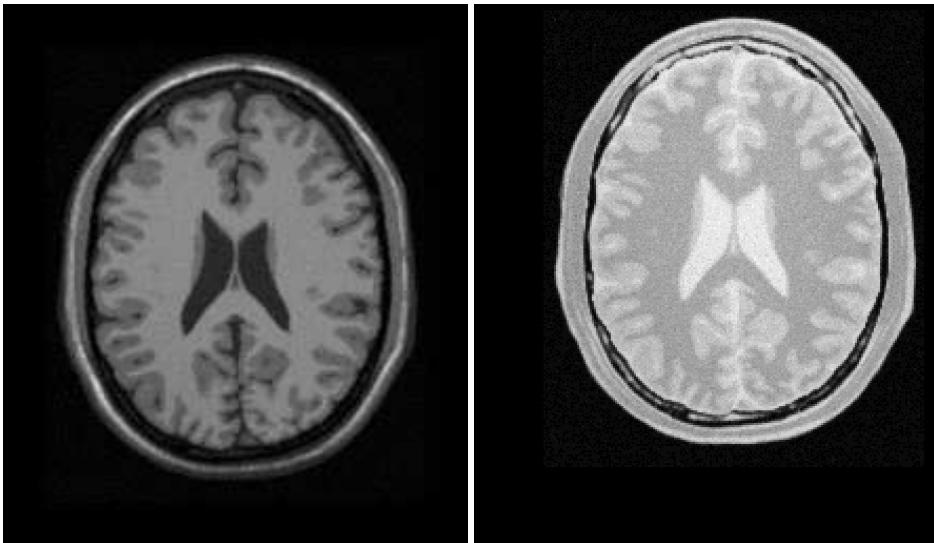


Figure 3.10: A T1 MRI (fixed image) and a proton density MRI (moving image) are provided as input to the registration method.

The second image is the result of intentionally translating the image `BrainProtonDensitySlice-Border20.png` by $(13, 17)$ millimeters. Both images have unit-spacing and are shown in Figure 3.10. The registration process converges after 46 iterations and produces the following results:

```
Translation X = 13.0204  
Translation Y = 17.0006
```

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 3.11. The center and right parts of the figure present a checkerboard composite of the fixed and moving images before and after registration respectively.

Figure 3.12 (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.

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, the 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

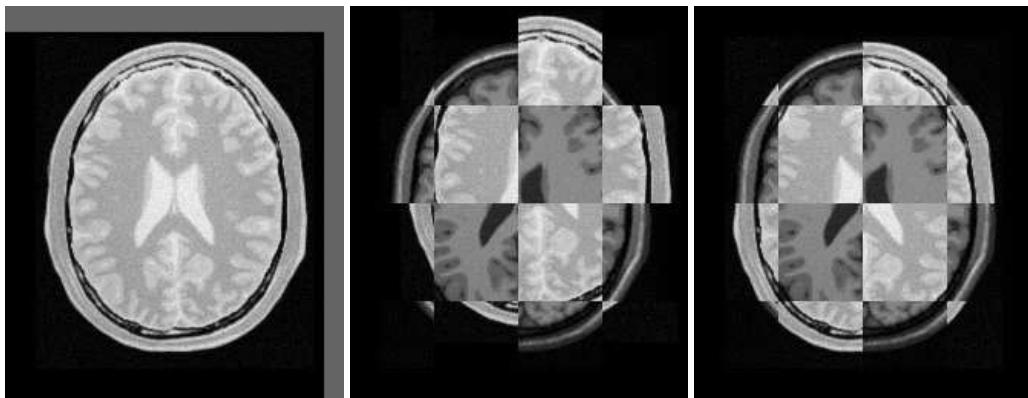


Figure 3.11: The mapped moving image (left) and the composition of fixed and moving images before (center) and after (right) registration with Mattes mutual information.

of bins used, the optimizer's path may get trapped early on in local minima. Figure 3.13 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 are being run with a sub-optimal set of parameters.

The plots in Figures 3.12 and 3.13 were generated using Gnuplot³. The scripts used for this purpose are available in the `ITKSoftwareGuide` Git repository under the directory

`ITKSoftwareGuide/SoftwareGuide/Art`.

Data for the plots were 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`⁴ 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

³<http://www.gnuplot.info/>

⁴<http://www.gnu.org/software/sed/sed.html>

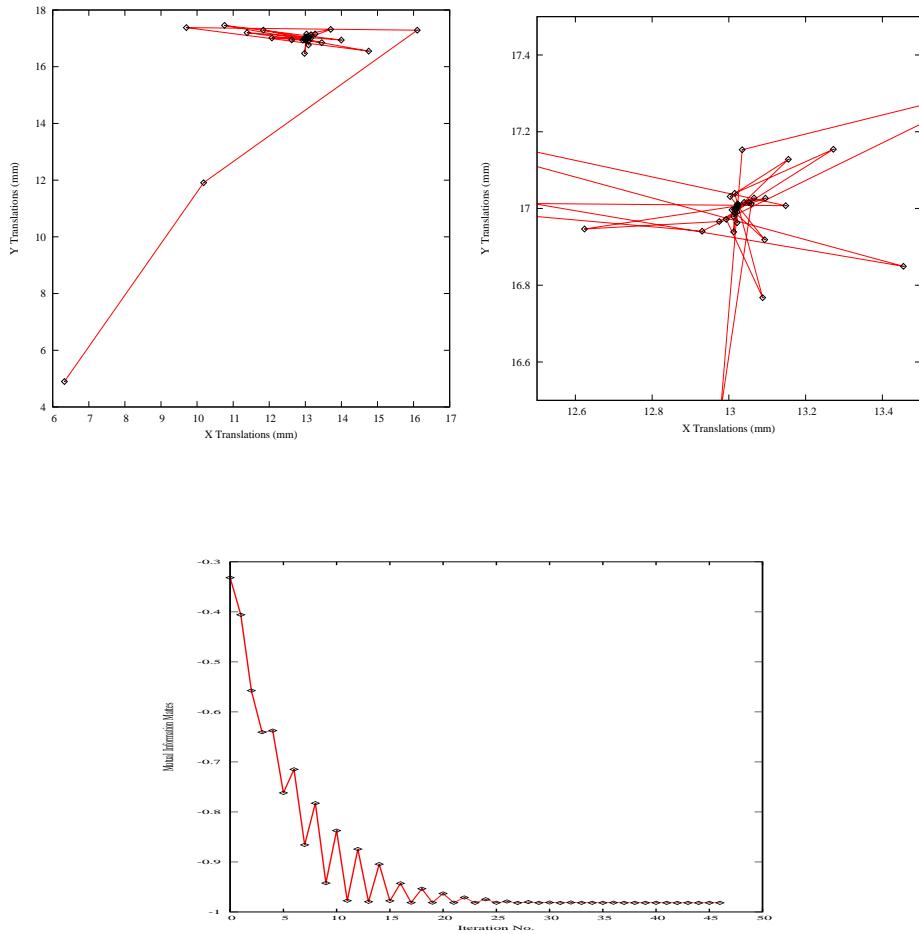


Figure 3.12: Sequence of translations and metric values at each iteration of the optimizer.

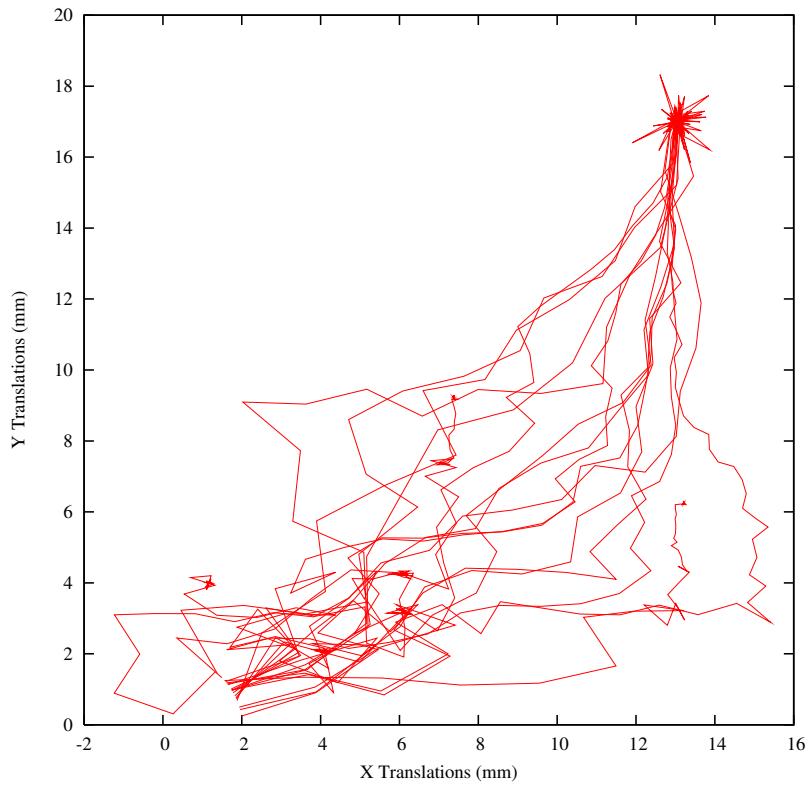


Figure 3.13: Sensitivity of the optimization path to the number of Bins used for estimating the value of Mutual Information with Mattes et al. approach.

those that can not be replicated⁵. Science is dead when people blindly trust authorities⁶ instead of verifying their statements by performing their own experiments [48, 49].

3.6 Centered Transforms

The ITK image coordinate origin is typically located in one of the image corners (see the Defining Origin and Spacing section of Book 1 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 Apache 2.0 license⁷ that does not restrict modification nor distribution of the code, and that does not impose the assimilation demands of viral licenses such as GPL⁸.

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 are, 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 [14].

3.6.1 Rigid Registration in 2D

The source code for this section can be found in the file
ImageRegistration5.cxx.

⁵<http://science.creativecommons.org/>

⁶For example: Reviewers of Scientific Journals.

⁷<http://www.opensource.org/licenses/Apache-2.0>

⁸<http://www.gnu.org/copyleft/gpl.html>

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 3.2. 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 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;
```

In the Hello World! example, we used Fixed/Moving initial transforms to initialize the registration configuration. That approach was good to get an intuition of the registration method, specifically when we aim to run a multistage registration process, from which the output of each stage can be used to initialize the next registration stage.

To get a better understanding of the registration process in such situations, consider an example of 3 stages registration process that is started using an initial moving transform (Γ_{mi}). Multiple stages are handled by linking multiple instantiations of the `itk::ImageRegistrationMethodv4` class. Inside the registration filter of the first stage, the initial moving transform is added to an internal composite transform along with an updatable identity transform (Γ_u). Although the whole composite transform is used for metric evaluation, only the Γ_u is set to be updated by the optimizer at each iteration. The Γ_u will be considered as the output transform of the current stage when the optimization process is converged. This implies that the output of this stage does not include the initialization parameters, so we need to concatenate the output and the initialization transform into a composite transform to be considered as the final transform of the first registration stage.

$$T_1(x) = \Gamma_{mi}(\Gamma_{stage_1}(x))$$

Consider that, as explained in section 3.3, the above transform is a mapping from the virtual domain (i.e. fixed image space, when no fixed initial transform) to the moving image space.

Then, the result transform of the first stage will be used as the initial moving transform for the second stage of the registration process, and this approach goes on until the last stage of the registration process.

At the end of the registration process, the Γ_{mi} and the outputs of each stage can be concatenated into a final composite transform that is considered to be the final output of the whole registration process.

$$I'_m(x) = I_m(\Gamma_{mi}(\Gamma_{stage_1}(\Gamma_{stage_2}(\Gamma_{stage_3}(x)))))$$

The above approach is especially useful if individual stages are characterized by different types of transforms, e.g. when we run a rigid registration process that is proceeded by an affine registration which is completed by a BSpline registration at the end.

In addition to the above method, there is also a direct initialization method in which the initial transform will be optimized directly. In this way the initial transform will be modified during the registration process, so it can be used as the final transform when the registration process is completed. This direct approach is conceptually close to what was happening in ITKv3 registration.

Using this method is very simple and efficient when we have only one level of registration, which is the case in this example. Also, a good application of this initialization method in a multi-stage scenario is when two consequent stages have the same transform types, or at least the initial parameters can easily be inferred from the result of the previous stage, such as when a translation transform is followed by a rigid transform.

The direct initialization approach is shown by the current example in which we try to initialize the parameters of the optimizable transform (Γ_u) directly.

For this purpose, first, the initial transform object is constructed below. This transform will be initialized, and its initial parameters will be used when the registration process starts.

```
TransformType::Pointer initialTransform = TransformType::New();
```

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;
```

Then, we initialize the transform by passing the center of the fixed image as the rotation center with the `SetCenter()` method. Also, 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()`.

```
initialTransform->SetCenter( centerFixed );
initialTransform->SetTranslation( centerMoving - centerFixed );
```

Let's finally initialize the rotation with a zero angle.

```
initialTransform->SetAngle( 0.0 );
```

Now the current parameters of the initial transform will be set to a registration method, so they can be assigned to the Γ_u directly. Note that you should not confuse the following function with the `SetMoving(Fixed) InitialTransform()` methods that were used in Hello World! example.

```
registration->SetInitialTransform( initialTransform );
```

Keep in mind that the scale of units in rotation and translation is quite different. For example, here 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 the translations and the center point coordinates that are measured in millimeters, so a naive application of gradient descent optimizer will not produce a smooth change of parameters, because a similar change of δ to each parameter will produce a different magnitude of impact on the transform. As the result, we need “parameter scales” to customize the learning rate for each parameter. We can take advantage of the scaling functionality provided by the optimizers.

In this example we use small factors in the scales associated with translations and the coordinates of the rotation center. However, for the transforms with larger parameters sets, it is not intuitive for a user to set the scales. Fortunately, a framework for automated estimation of parameter scales is provided by ITKv4 that will be discussed later in the example of section 3.8.

```

typedef OptimizerType::ScalesType          OptimizerScalesType;
OptimizerScalesType optimizerScales( );
  initialTransform->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::RegularStepGradientDescentOptimizerv4`. Below, we define the optimization parameters like the relaxation factor, learning rate (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->SetLearningRate( 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 3.14. The registration takes 20 iterations and produces the results:

`[0.17762, 110.489, 128.487, 0.00925022, 0.00140223]`

These results are interpreted as

- Angle = 0.17762 radians
- Center = (110.489, 128.487) millimeters
- Translation = (0.00925022, 0.00140223) millimeters

As expected, these values match the misalignment intentionally introduced into the moving image quite well, since 10 degrees is about 0.174532 radians.

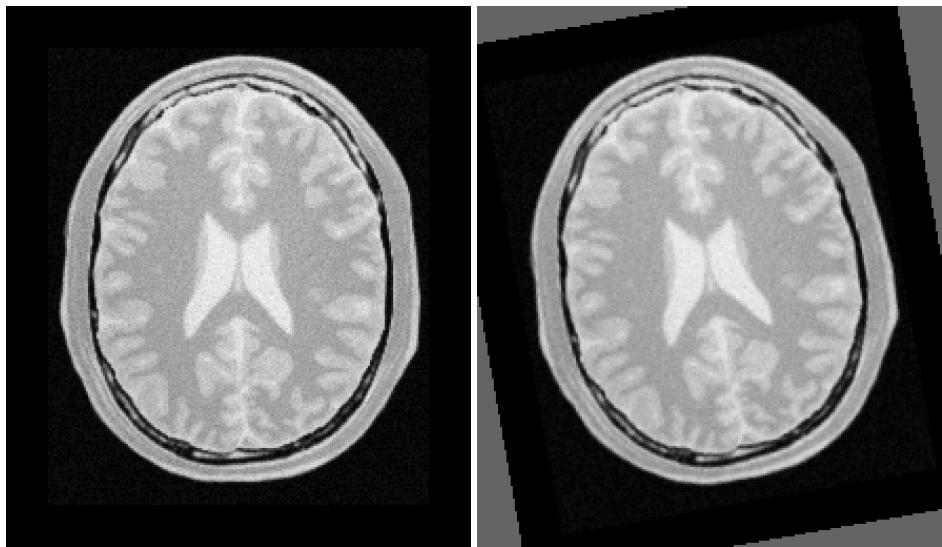


Figure 3.14: Fixed and moving images are provided as input to the registration method using the CenteredRigid2D transform.

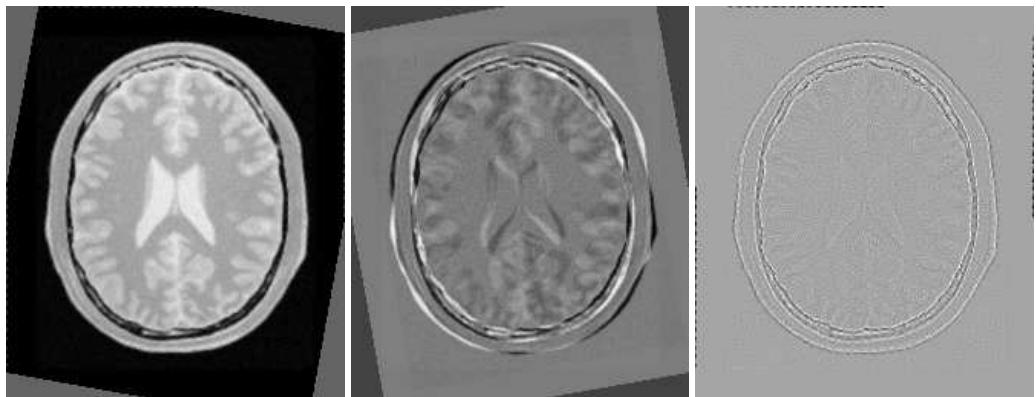


Figure 3.15: Resampled moving image (left). Differences between the fixed and moving images, before (center) and after (right) registration using the CenteredRigid2D transform.

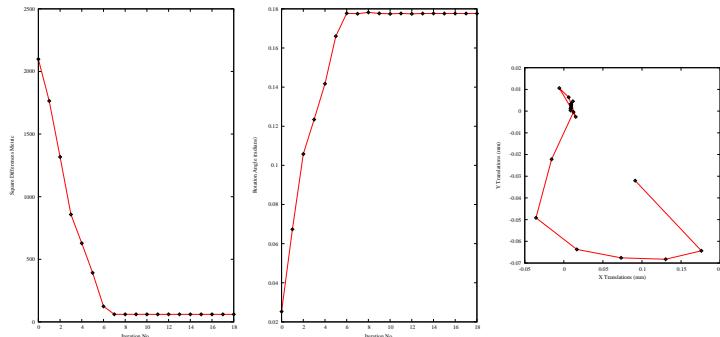


Figure 3.16: Metric values, rotation angle and translations during registration with the CenteredRigid2D transform.

Figure 3.15 shows from left to right the resampled moving image after registration, the difference between the fixed and moving images before registration, and the difference between the 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 3.16 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 progresses.

Let's now consider the case in which rotations and translations are present in the initial registration, as in the following pair of images:

- 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 3.17. In order to accelerate convergence it is convenient to use a larger step length as shown here.

```
optimizer->SetMaximumStepLength( 1.3 );
```

The registration now takes 35 iterations and produces the following results:

```
[0.174552, 110.041, 128.917, 12.9339, 15.9149]
```

These parameters are interpreted as

- Angle = 0.17452 radians

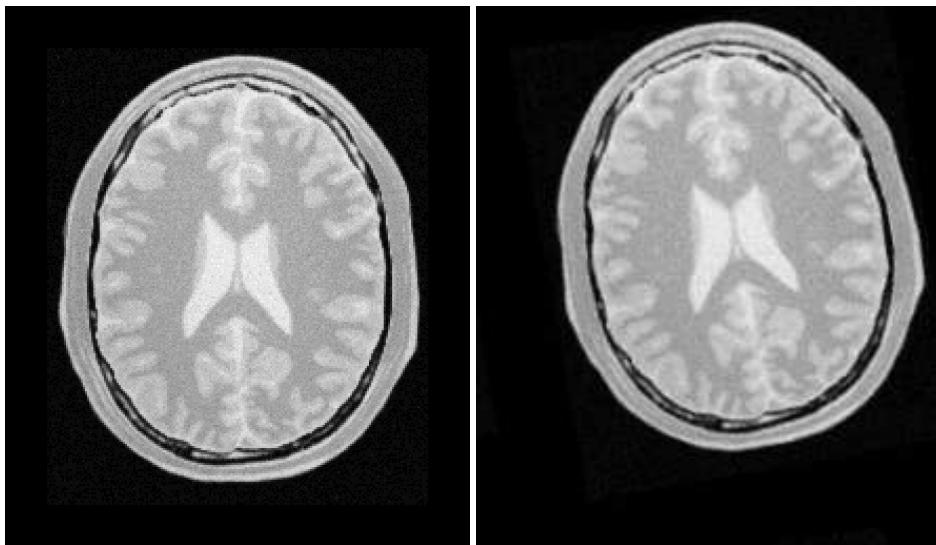


Figure 3.17: Fixed and moving images provided as input to the registration method using the CenteredRigid2D transform.

- Center = (110.041, 128.917) millimeters
- Translation = (12.9339, 15.9149) 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 3.18 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 3.19 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.

3.6.2 Initializing with Image Moments

The source code for this section can be found in the file `ImageRegistration6.cxx`.

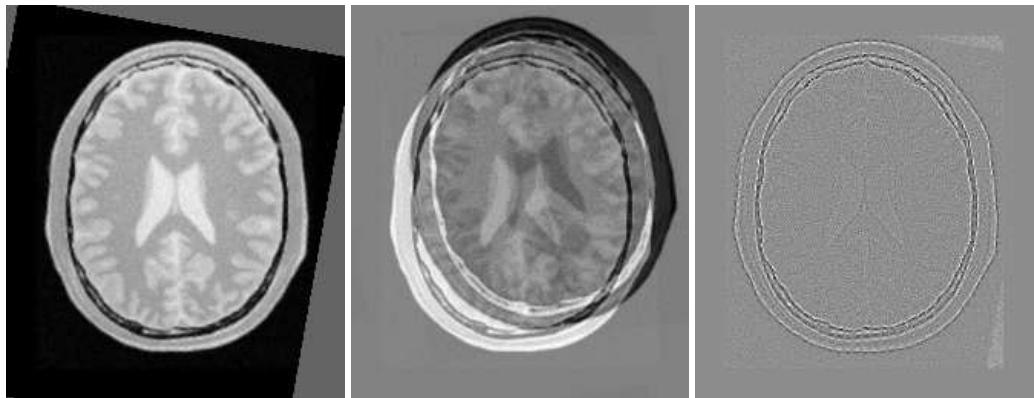


Figure 3.18: Resampled moving image (left). Differences between the fixed and moving images, before (center) and after (right) registration with the CenteredRigid2D transform.

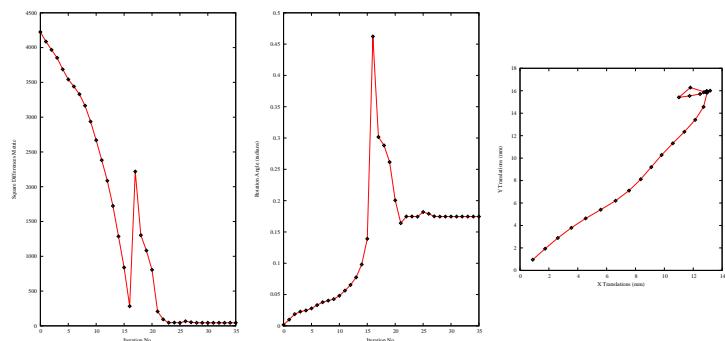


Figure 3.19: Metric values, rotation angle and translations during the registration using the CenteredRigid2D transform on an image with rotation and translation mis-registration.

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 3.6.1. Even though this current example is done in 2D, the class `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 match 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;
```

Like the previous section, a direct initialization method is used here. The transform object is constructed below. This transform will be initialized, and its initial parameters will be considered as the parameters to be used when the registration process begins.

```
TransformType::Pointer transform = TransformType::New();
```

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 images 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 initialized transform object will be set to the registration method, and the starting point of the registration is defined by its initial parameters.

If the `InPlaceOn()` method is called, this initialized transform will be the output transform object or “grafted” to the output. Otherwise, this “InitialTransform” will be deep-copied or “cloned” to the output.

```
registration->SetInitialTransform( transform );
registration->InPlaceOn();
```

Since the registration filter has `InPlace` set, the transform object is grafted to the output and is updated by the registration method.

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 3.14. The registration takes 22 iterations and produces:

```
[0.17429, 111.172, 131.563, 12.4582, 16.0724]
```

These parameters are interpreted as

- Angle = 0.17429 radians
- Center = (111.172, 131.563) millimeters
- Translation = (12.4582, 16.0724) 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 from the five parameters.

```
TransformType::MatrixType matrix = transform->GetMatrix();
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 =
0.98485 -0.173409
0.173409 0.98485
```

```
Offset =
[36.9567, -1.21272]
```

This output illustrates how counter-intuitive the mix of center of rotation and translations can be. Figure 3.20 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). 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

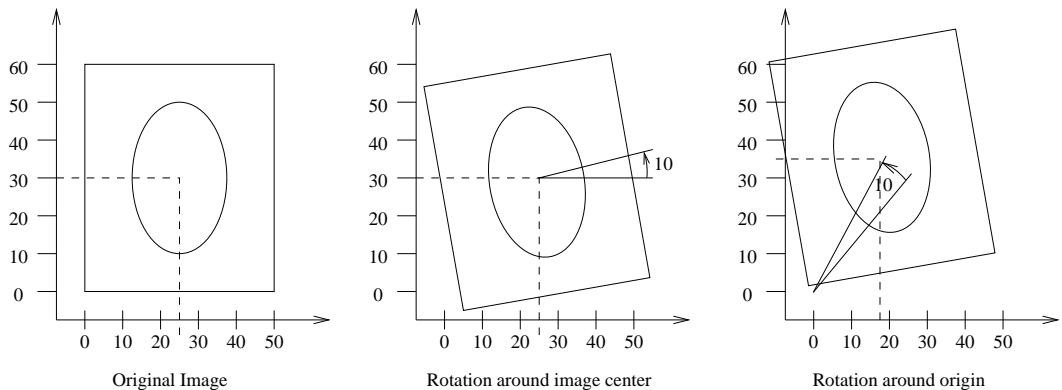


Figure 3.20: Effect of changing the center of rotation.

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 translation of $(36.95, -1.21)$. 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 $(36.95, -1.21)$ maps this point to $(123.47, 146.76)$, which is close to the result of our previous computation.

It is unlikely that we could have chosen these translations as the initial guess, since we tend to think about images 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 3.22 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 3.23 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 progresses. 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!

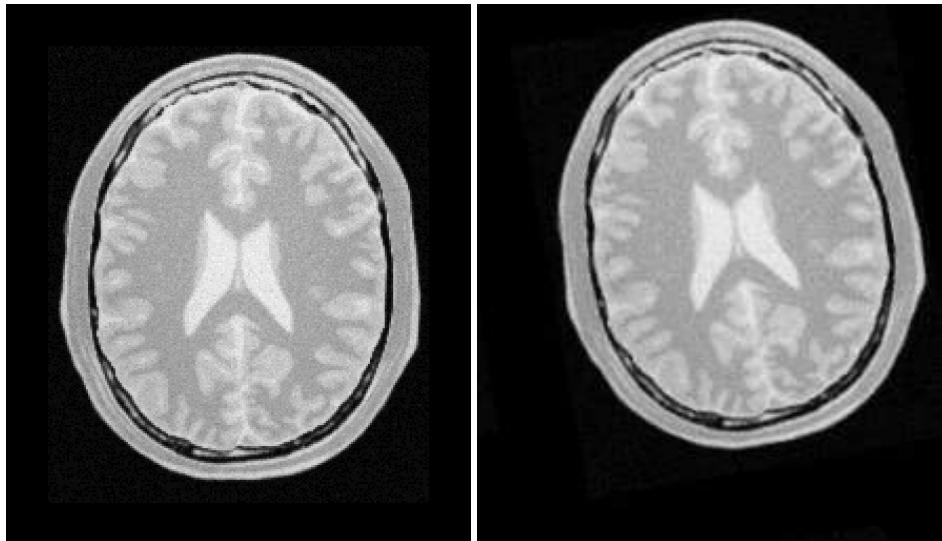


Figure 3.21: Fixed and moving images provided as input to the registration method using `CenteredTransformInitializer`.

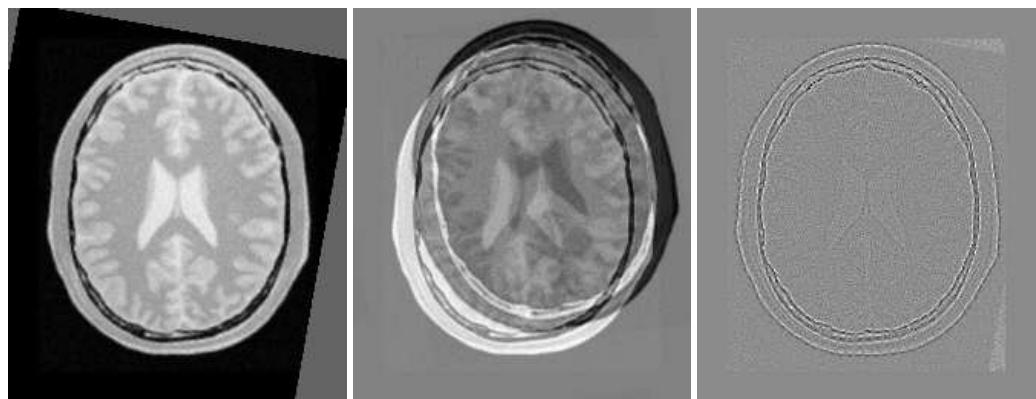


Figure 3.22: Resampled moving image (left). Differences between fixed and moving images, before registration (center) and after registration (right) with the `CenteredTransformInitializer`.

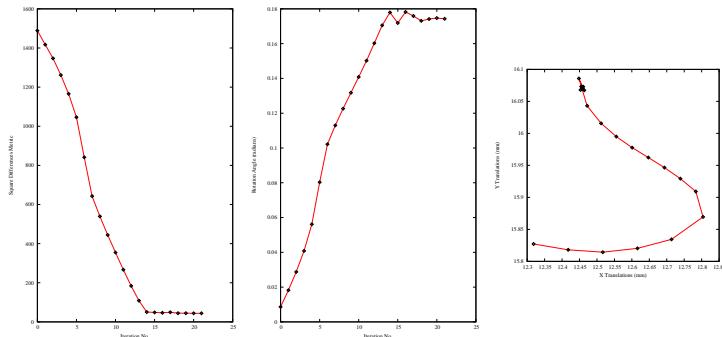


Figure 3.23: Plots of the Metric, rotation angle, center of rotation and translations during the registration using CenteredTransformInitializer.

3.6.3 Similarity Transform in 2D

The source code for this section can be found in the file `ImageRegistration7.cxx`.

This example illustrates the use of the `itk::CenteredSimilarity2DTransform` class for performing registration in 2D. The example code is for the most part identical to the code presented in Section 3.6.2. 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 (isotropic) scaling. It preserves angles and maps lines into lines. This transform is implemented in the toolkit as deriving from a rigid 2D 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 results 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 used 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;
```

As before, the transform object is constructed and initialized before it is passed to the registration filter.

```
TransformType::Pointer transform = TransformType::New();
```

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 );
```

Now the initialized transform object will be set to the registration method, and its initial parameters are used to initialize the registration process.

Also, by calling the `InPlaceOn()` method, this initialized transform will be the output transform object or “grafted” to the output of the registration process.

```
registration->SetInitialTransform( transform );
registration->InPlaceOn();
```

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 fourth 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 also set the ordinary parameters of the optimization method. In this case we are using a `itk::RegularStepGradientDescentOptimizerv4`. Below we define the optimization parameters, i.e. initial learning rate (step length), minimal step length and number of iterations. The last two act as stopping criteria for the optimization.

```

optimizer->SetLearningRate( 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 3.24. The registration takes 60 iterations and produces:

`[0.833193, -0.174514, 111.025, 131.92, -12.7267, -12.757]`

That are interpreted as

- Scale factor = 0.833193
- Angle = -0.174514 radians
- Center = $(111.025, 131.92)$ millimeters
- Translation = $(-12.7267, -12.757)$ millimeters

These values approximate the misalignment intentionally introduced into the moving image. Since 10 degrees is about 0.174532 radians.

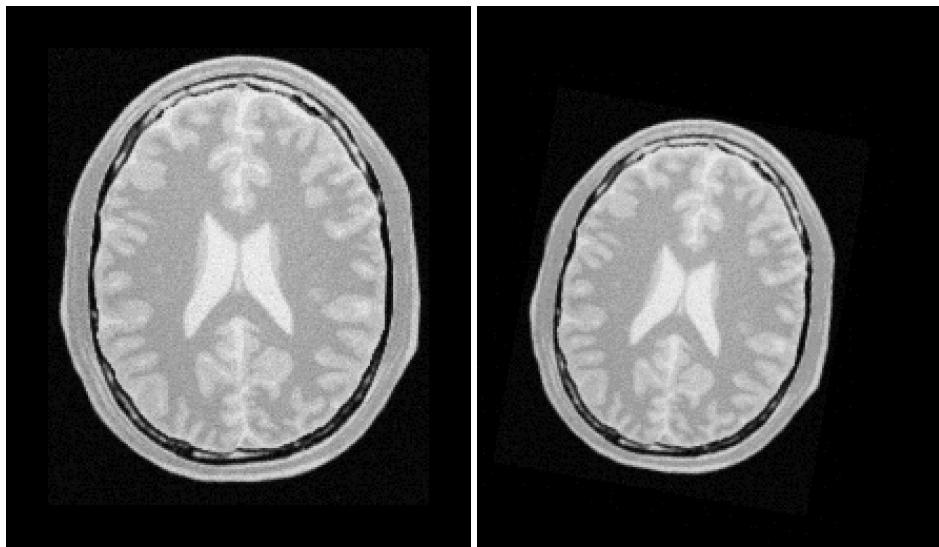


Figure 3.24: Fixed and Moving image provided as input to the registration method using the Similarity2D transform.

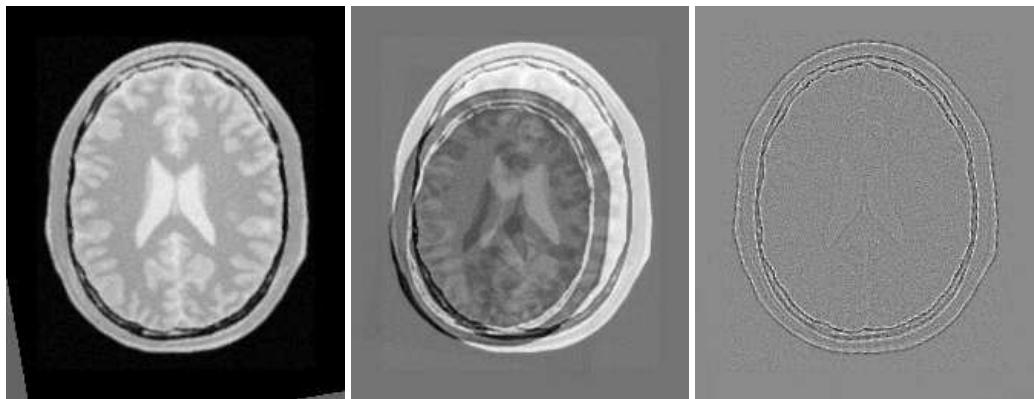


Figure 3.25: Resampled moving image (left). Differences between fixed and moving images, before (center) and after (right) registration with the Similarity2D transform.

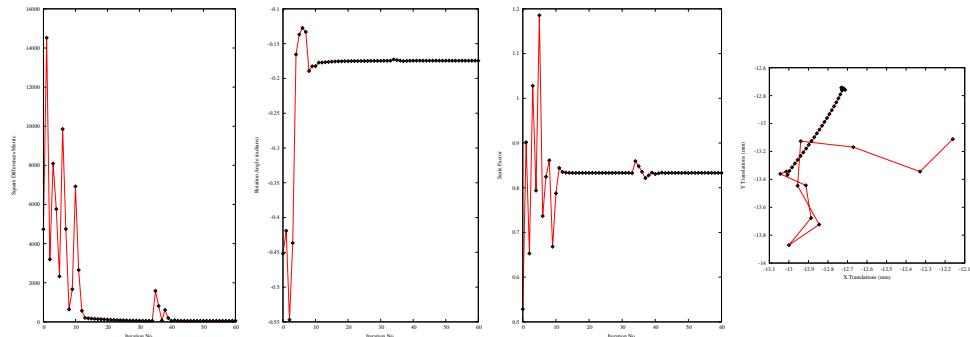


Figure 3.26: Plots of the Metric, rotation angle, scale factor, and translations during the registration using Similarity2D transform.

Figure 3.25 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 3.26 shows the plots of the main output parameters of the registration process. The metric values at every iteration are shown on the left. The rotation angle and scale factor values are shown in the two center plots while the translation components of the registration are presented in the plot on the right.

3.6.4 Rigid Transform in 3D

The source code for this section can be found in the file `ImageRegistration8.cxx`.

This example illustrates the use of the `itk::VersorRigid3DTransform` class for performing registration of two 3D images. The class `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 common uses 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, because addition is not a closed operation in the space of versor components. Hence, we need to use Versor composition operation to update the first three components of the parameter array (rotation parameters), and Vector addition for updating the last three components of the parameters array (translation parameters) [25, 28].

In the previous version of ITK, a special optimizer, `itk::VersorRigid3DTransformOptimizer`

was needed for registration to deal with versor computations. Fortunately in ITKv4, the `itk::RegularStepGradientDescentOptimizerv4` can be used for both vector and versor transform optimizations because, in the new registration framework, the task of updating parameters is delegated to the moving transform itself. The `UpdateTransformParameters` method is implemented in the `itk::Transform` class as a virtual function, and all the derived transform classes can have their own implementations of this function. Due to this fact, the updating function is re-implemented for versor transforms so it can handle versor composition of the rotation parameters.

```
#include "itkRegularStepGradientDescentOptimizerv4.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 initial transform object is constructed below. This transform will be initialized, and its initial parameters will be used when the registration process starts.

```
TransformType::Pointer initialTransform = TransformType::New();
```

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 types 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( initialTransform );
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::Vesor` which is simply a unit quaternion. The `VesorType` can be obtained from the transform traits. The vesor 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 vesor object and initialize its parameters by passing a rotation axis and an angle.

```
typedef TransformType::VesorType VesorType;
typedef VesorType::VectorType VectorType;
VesorType rotation;
VectorType axis;
axis[0] = 0.0;
axis[1] = 0.0;
axis[2] = 1.0;
const double angle = 0;
rotation.Set( axis, angle );
initialTransform->SetRotation( rotation );
```

Now the current initialized transform will be set to the registration method, so its initial parameters can be used to initialize the registration process.

```
registration->SetInitialTransform( initialTransform );
```

Let's execute this example over some of the images available in the following website

<http://public.kitware.com/pub/itk/Data/BrainWeb>.

Note that the images in this website are compressed in `.tgz` files. You should download these files and decompress them in your local system. After decompressing and extracting the files you could take a pair of volumes, for example the pair:

- `brainweb1ela10f20.mha`
- `brainweb1ela10f20Rot10Tx15.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*.

Also, instead of doing the above steps manually, you can turn on the following flag in your build environment:

`ITK_USE_BRAINWEB_DATA`

Then, the above data will be loaded to your local ITK build directory.

The registration takes 21 iterations and produces:

[7.2295e-05, -7.20626e-05, -0.0872168, 2.64765, -17.4626, -0.00147153]

That are interpreted as

- Vensor = $(7.2295e - 05, -7.20626e - 05, -0.0872168)$
- Translation = $(2.64765, -17.4626, -0.00147153)$ millimeters

This Vensor 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 `VensorRigid3DTransform` 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 from the 6 parameters.

```
TransformType::MatrixType matrix = finalTransform->GetMatrix();
TransformType::OffsetType offset = finalTransform->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 =
0.984786 0.173769 -0.000156187
-0.173769 0.984786 -0.000131469
0.000130965 0.000156609 1
```

```
Offset =
[-15, 0.0189186, -0.0305439]
```

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 $\arcsin(0.173769)$ which is very close to 10 degrees, as we expected.

Figure 3.28 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 3.29 shows the plots of the main output parameters of the registration process. The Z component of the vensor is plotted as an indication of how the rotation progresses. The X,Y translation components of the registration are plotted at every iteration too.

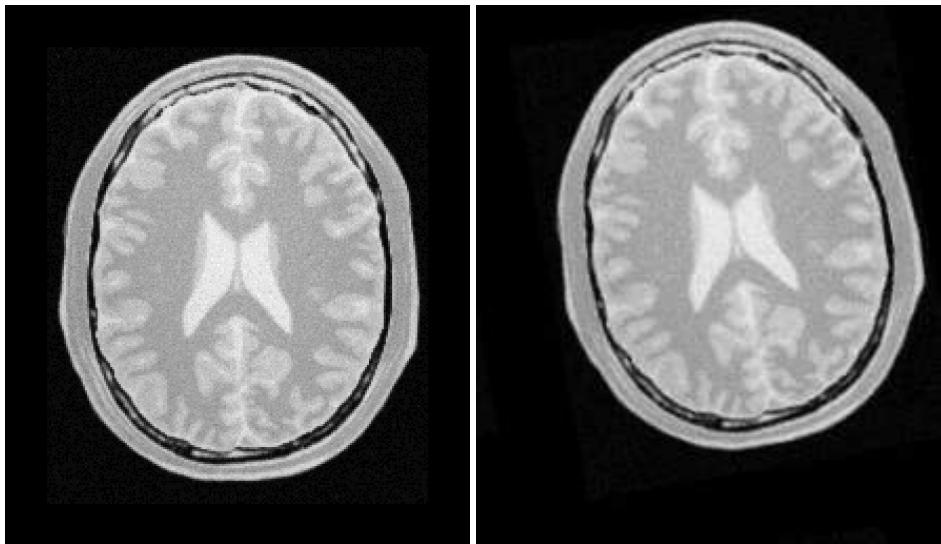


Figure 3.27: Fixed and moving image provided as input to the registration method using `CenteredTransformInitializer`.

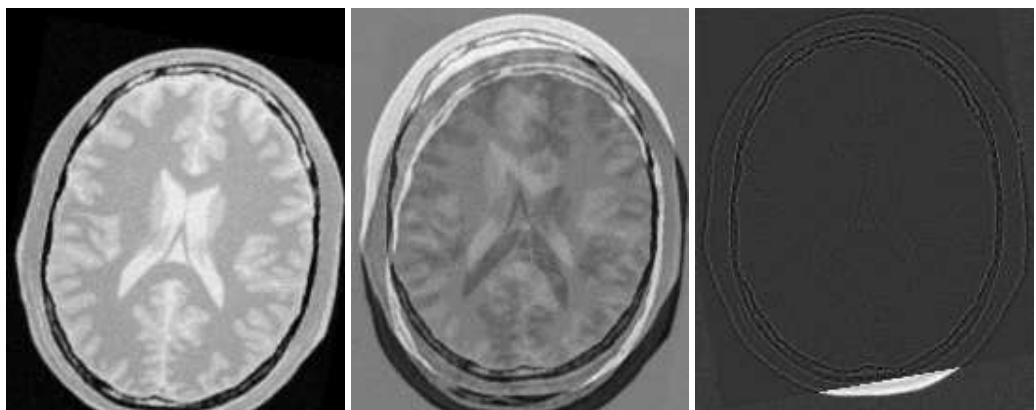


Figure 3.28: Resampled moving image (left). Differences between fixed and moving images, before (center) and after (right) registration with the `CenteredTransformInitializer`.

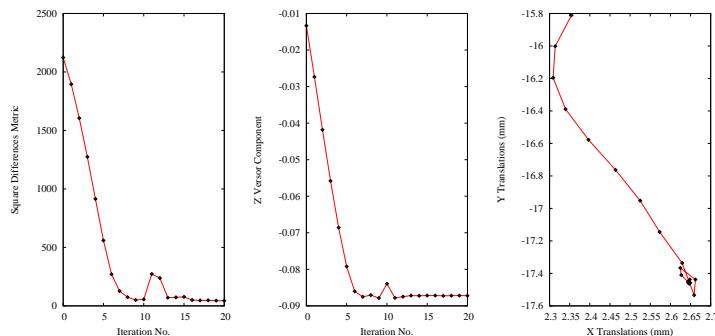


Figure 3.29: Plots of the metric, rotation angle, center of rotation and translations during the registration using `CenteredTransformInitializer`.

Shell and Gnuplot scripts for generating the diagrams in Figure 3.29 are available in the `ITKSoftwareGuide` Git repository under the directory

`ITKSoftwareGuide/SoftwareGuide/Art`.

You are strongly encouraged to run the example code, since only in this way can you gain 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.

3.6.5 Centered Affine Transform

The source code for this section can be found in the file `ImageRegistration9.cxx`.

This example illustrates the use of the `itk::AffineTransform` for performing registration in 2D. The example code is, for the most part, identical to that in 3.6.2. 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 then define 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 is initialized before the registration process starts.

```
TransformType::Pointer transform = TransformType::New();
```

In this example, we again use the `itk::CenteredTransformInitializer` helper class in order to compute reasonable values for the initial center of rotation and the translations. 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 transform object to the registration filter, and it will be grafted to the output transform of the registration filter by updating its parameters during the the registration process.

```
registration->SetInitialTransform( transform );
registration->InPlaceOn();
```

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, 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::RegularStepGradientDescentOptimizerv4` as before. Below, we define the optimization parameters like learning rate (initial step length), minimum step length and number of iterations. These last two act as stopping criteria for the optimization.

```

optimizer->SetLearningRate( steplength );
optimizer->SetMinimumStepLength( 0.0001 );
optimizer->SetNumberOfIterations( maxNumberOfIterations );

```

Finally we trigger the execution of the registration method by calling the `Update()` method. The call is placed in a `try/catch` block in the case any exceptions are thrown.

```

try
{
    registration->Update();
    std::cout << "Optimizer stop condition: "
    << registration->GetOptimizer()->GetStopConditionDescription()
    << std::endl;
}
catch( itk::ExceptionObject & err )
{
    std::cerr << "ExceptionObject caught !" << std::endl;
    std::cerr << err << std::endl;
    return EXIT_FAILURE;
}

```

Once the optimization converges, we recover the parameters from the registration 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.

```

const TransformType::ParametersType finalParameters =
registration->GetOutput()->Get()->GetParameters();

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 3.30. 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 92 iterations and produces

```
90 44.0851 [0.9849, -0.1729, 0.1725, 0.9848, 12.4541, 16.0759] AffineAngle: 9.9494
```

These results are interpreted as

- Iterations = 92
- Final Metric = 44.0386
- Center = (111.204, 131.591) millimeters
- Translation = (12.4542, 16.076) millimeters
- Affine scales = (1.00014, .999732)

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 9.9494 degrees, which is approximately the intentional misalignment of 10.0 degrees.

Figure 3.31 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 3.32 shows the plots of the main output parameters of the registration process. The metric values at every iteration are shown on the left plot. The angle values are shown on the middle plot, while the translation components of the registration are presented on the right plot. Note that the final total offset of the transform is to be computed as a combination of the shift due to rotation plus the explicit translation set on the transform.

3.7 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

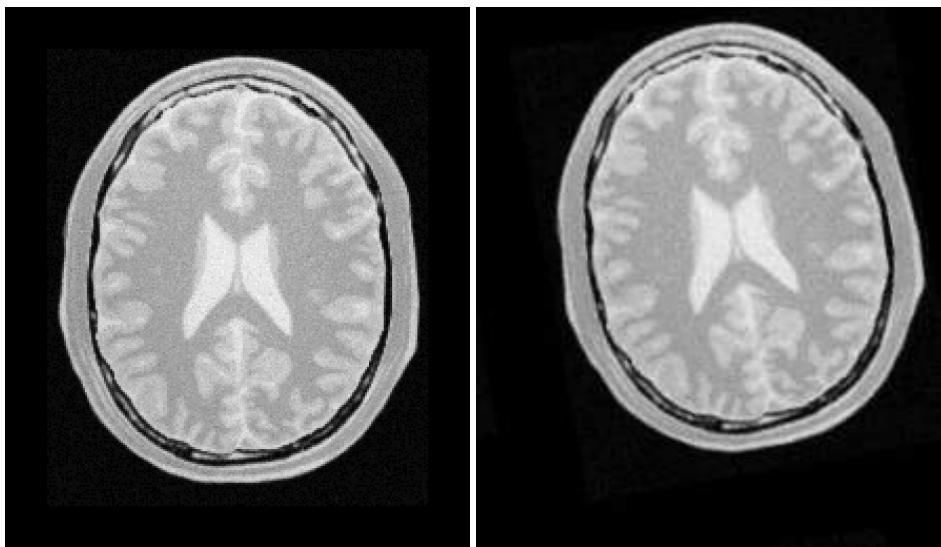


Figure 3.30: Fixed and moving images provided as input to the registration method using the `AffineTransform`.

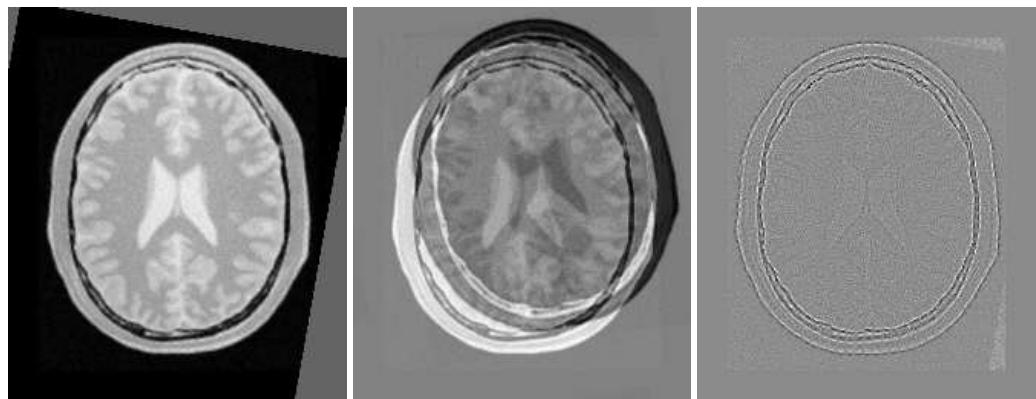


Figure 3.31: The resampled moving image (left), and the difference between the fixed and moving images before (center) and after (right) registration with the `AffineTransform` transform.

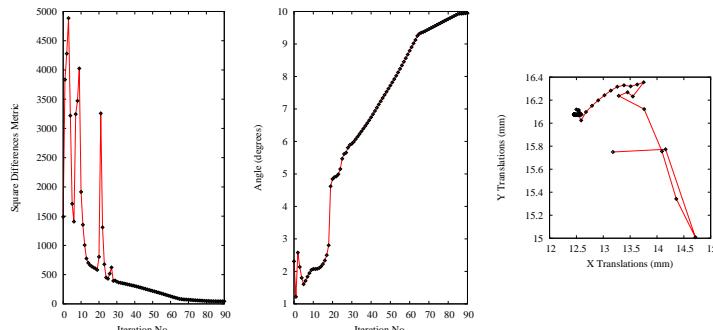


Figure 3.32: Metric values, rotation angle and translations during the registration using the `AffineTransform` transform.

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 improves the registration success rate and also increases robustness by eliminating local optima at coarser scales. Robustness can be improved even more by smoothing the images at coarse scales.

In all previous examples we ran the registration process at a single resolution. However, the ITKv4 registration framework is structured to provide a multi-resolution registration method. For this purpose we only need to define the number of levels as well as the resolution and smoothness of the input images at each level. The registration filter smoothes and subsamples the images according to user-defined *ShrinkFactor* and *SmoothingSigma* vectors.

We now present the multi-resolution capabilities of the framework by way of an example.

3.7.1 Fundamentals

The source code for this section can be found in the file

`MultiResImageRegistration1.cxx`.

This example illustrates the use of the `itk::ImageRegistrationMethodv4` to solve a simple multi-modality registration problem by a multi-resolution approach. Since ITKv4 registration method is designed based on a multi-resolution structure, a separate set of classes are no longer required to run the registration process of this example.

This is a great advantage over the previous versions of ITK, as in ITKv3 we had to use a different filter (`itk::MultiResolutionImageRegistrationMethod`) to run a multi-resolution process. Also, we had to use image pyramids filters (`itk::MultiResolutionPyramidImageFilter`) for creating the sequence of downsampled images. Hence, you can see how ITKv4 framework is more user-friendly in more complex situations.

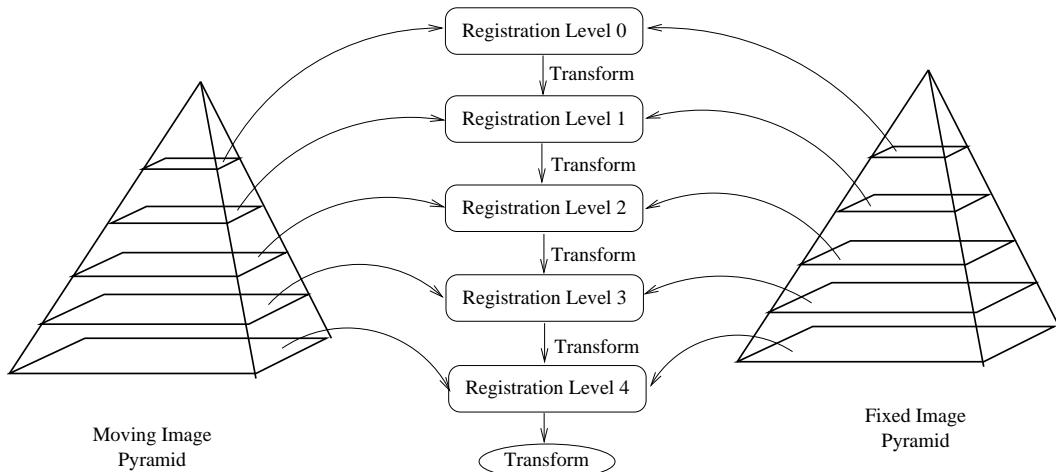


Figure 3.33: Conceptual representation of the multi-resolution registration process.

To begin the example, we include the headers of the registration components we will use.

```
#include "itkImageRegistrationMethodv4.h"
#include "itkTranslationTransform.h"
#include "itkMattesMutualInformationImageToImageMetricv4.h"
#include "itkRegularStepGradientDescentOptimizerv4.h"
```

The `ImageRegistrationMethodv4` solves a registration problem in a coarse-to-fine manner as illustrated in Figure 3.33. 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, it may be possible to take more aggressive step sizes and have a more relaxed convergence criterion.

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, where `ImageRegistrationMethodv4` invokes a `MultiResolutionIterationEvent()`. The registration components can be changed by implementing a `itk::Command` which responds to the event. A brief description of the interaction between events and commands was previously presented in Section 3.4.

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 3.4.

```
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::RegularStepGradientDescentOptimizerv4<double> 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) ITK_OVERRIDE
{
```

First we verify that the event invoked is of the right type, `itk::MultiResolutionIterationEvent()`. If not, we return without any further action.

```
if( !(itk::MultiResolutionIterationEvent().CheckEvent( &event ) ) )
{
    return;
}
```

We then convert the input object pointer to a `RegistrationPointer`. Note that no error checking is done here to verify the `dynamic_cast` was successful since we know the actual object is a registration method. Then we ask for the optimizer object from the registration method.

```
RegistrationPointer registration =
    static_cast<RegistrationPointer>( object );
OptimizerPointer optimizer = static_cast<OptimizerPointer >(
    registration->GetModifiableOptimizer() );
```

If this is the first resolution level we set the learning rate (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 5 in order to allow the optimizer to focus on progressively smaller regions. The learning rate 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.

```
if ( registration->GetCurrentLevel() == 0 )
{
    optimizer->SetLearningRate( 16.00 );
    optimizer->SetMinimumStepLength( 2.5 );
}
else
{
    optimizer->SetLearningRate( optimizer->GetCurrentStepLength() );
    optimizer->SetMinimumStepLength(
        optimizer->GetMinimumStepLength() * 0.2 );
}
```

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 &) ITK_OVERRIDE
{
    return;
}
};
```

The fixed and moving image types are defined as in previous examples. The downsampled images for different resolution levels are created internally by the registration method based on the values provided for *ShrinkFactor* and *SmoothingSigma* vectors.

The types for the registration components are then derived using the fixed and moving image type, as in previous examples.

To set the optimizer parameters, note that *LearningRate* and *MinimumStepLength* are set in the obsever at the begining of each resolution level. The other optimizer parameters are set as follows.

```
optimizer->SetNumberOfIterations( 200 );
optimizer->SetRelaxationFactor( 0.5 );
```

We set the number of multi-resolution levels to three and set the corresponding shrink factor and smoothing sigma values for each resolution level. Using smoothing in the subsampled images in low-resolution levels can avoid large fluctuations in the metric function, which prevents the optimizer from becoming trapped in local minima. In this simple example we have no smoothing, and we have used small shrinkings for the first two resolution levels.

```

const unsigned int numberOfWorks = 3;

RegistrationType::ShrinkFactorsArrayType shrinkFactorsPerLevel;
shrinkFactorsPerLevel.SetSize( 3 );
shrinkFactorsPerLevel[0] = 3;
shrinkFactorsPerLevel[1] = 2;
shrinkFactorsPerLevel[2] = 1;

RegistrationType::SmoothingSigmasArrayType smoothingSigmasPerLevel;
smoothingSigmasPerLevel.SetSize( 3 );
smoothingSigmasPerLevel[0] = 0;
smoothingSigmasPerLevel[1] = 0;
smoothingSigmasPerLevel[2] = 0;

registration->SetNumberOfLevels ( numberOfWorks );
registration->SetShrinkFactorsPerLevel( shrinkFactorsPerLevel );
registration->SetSmoothingSigmasPerLevel( smoothingSigmasPerLevel );

```

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 `AddObserver()` method.

```

typedef RegistrationInterfaceCommand<RegistrationType> CommandType;
CommandType::Pointer command = CommandType::New();

registration->AddObserver( itk::MultiResolutionIterationEvent(), command );

```

Then we trigger the registration process by calling `Update()`.

Let's execute this example using the following images

- BrainT1SliceBorder20.png
- BrainProtonDensitySliceShifted13x17y.png

The output produced by the execution of the method is

```

0 -0.316956 [11.4200, 11.2063]
1 -0.562048 [18.2938, 25.6545]
2 -0.407696 [11.3643, 21.6569]
3 -0.5702 [13.7244, 18.4274]
4 -0.803252 [11.1634, 15.3547]

0 -0.697586 [12.8778, 16.3846]
1 -0.901984 [13.1794, 18.3617]
2 -0.827423 [13.0545, 17.3695]
3 -0.92754 [12.8528, 16.3901]
4 -0.902671 [12.9426, 16.8819]
5 -0.941212 [13.1402, 17.3413]

```

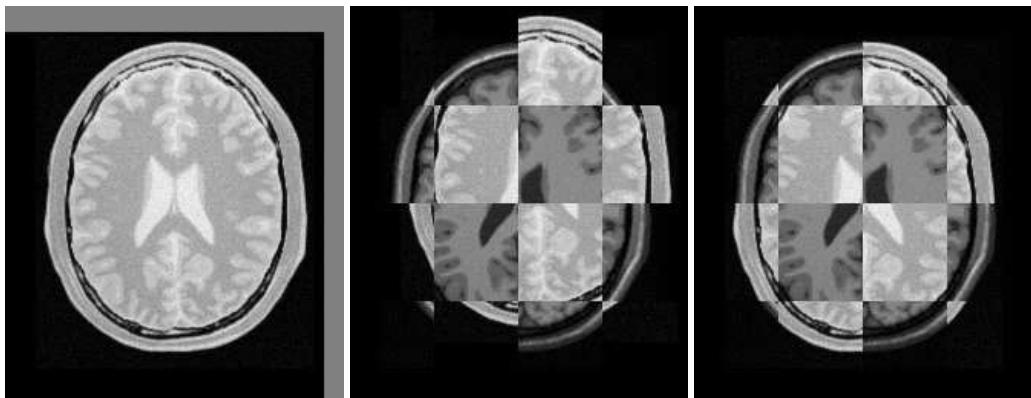


Figure 3.34: Mapped moving image (left) and composition of fixed and moving images before (center) and after (right) registration.

```

0   -0.922239  [13.0364, 17.1138]
1   -0.930203  [12.9463, 16.8806]
2   -0.930959  [13.0191, 16.9822]

```

```

Result =
Translation X = 13.0192
Translation Y = 16.9823
Iterations     = 4
Metric value   = -0.929237

```

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 3.34. The center and right images of the figure depict a checkerboard composite of the fixed and moving images before and after registration.

Figure 3.35 (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 5 iterations with the remaining iterations just doing fine adjustments. It is interesting to compare these results with those of the single resolution example in Section 3.5.1, where 46 iterations were required as more conservative optimization parameters had to be used.

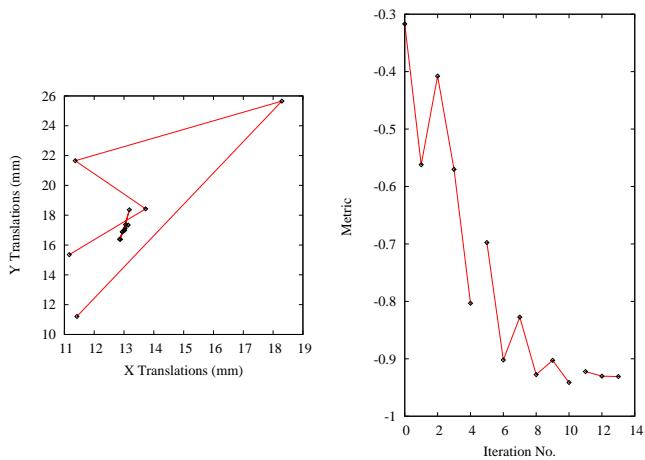


Figure 3.35: Sequence of translations and metric values at each iteration of the optimizer.

3.8 Multi-Stage Registration

In section 3.7 you noticed how to tweak component settings between multi-resolution levels and saw how it can benefit the registration process. That is, the matching metric gets close to the optimal value before final parameter adjustments in full resolution. This approach saves large amounts of time in most practical cases, since fewer iterations are required at the full resolution level. This is helpful in cases like a deformable registration process on a large dataset, e.g. a high-resolution 3D image.

Another possible scheme is to apply a simple rigid transform for the initial coarse registration, then upgrade to an affine transform at the finer level. Finally, proceed to a deformable transform at the last level when we are close enough to the optimal value.

Fortunately, `itk::ImageRegistrationMethodv4` allows for multistage registration whereby each stage is characterized by possibly different transforms and different image metrics. As in the above situation, you may want to perform a linear registration followed by a deformable registration with both stages performed across multiple resolutions.

Multiple stages are handled by linking multiple instantiations of this class. An optional composite transform can be used as a container to concatenate the output transforms of multiple stages.

We now present the multistage capabilities of the framework by way of an example.

3.8.1 Fundamentals

The source code for this section can be found in the file `MultiStageImageRegistration1.cxx`.

This example illustrates the use of more complex components of the registration framework. In particular, it introduces a multistage, multi-resolution approach to run a multi-modal registration process using two linear `itk::TranslationTransform` and `itk::AffineTransform`. Also, it shows the use of *Scale Estimators* for fine-tuning the scale parameters of the optimizer when an Affine transform is used. The `itk::RegistrationParameterScalesFromPhysicalShift` filter is used for automatic estimation of the parameters scales.

To begin the example, we include the headers of the registration components we will use.

```
#include "itkImageRegistrationMethodv4.h"
#include "itkMatteMutualInformationImageToImageMetricv4.h"
#include "itkRegularStepGradientDescentOptimizerv4.h"
#include "itkConjugateGradientLineSearchOptimizerv4.h"

#include "itkTranslationTransform.h"
#include "itkAffineTransform.h"
#include "itkCompositeTransform.h"
```

In a multistage scenario, each stage needs an individual instantiation of the `itk::ImageRegistrationMethodv4`, so each stage can possibly have a different transform, a different optimizer, and a different image metric and can be performed in multiple levels. The configuration of the registration method at each stage closely follows the procedure in the previous section.

In early stages we can use simpler transforms and more aggressive optimization parameters to take big steps toward the optimal value. Then, at the final stage we can have a more complex transform to do fine adjustments of the final parameters.

A possible scheme is to use a simple translation transform for initial coarse registration levels and upgrade to an affine transform at the finer level. Since we have two different types of transforms, we can use a multistage registration approach as shown in the current example.

First we need to configure the registration components of the initial stage. The instantiation of the transform type requires only the dimension of the space and the type used for representing space coordinates.

```
typedef itk::TranslationTransform< double, Dimension > TTransformType;
```

The types of other registration components are defined here.

`itk::RegularStepGradientDescentOptimizerv4` is used as the optimizer of the first stage. Also, we use `itk::MatteMutualInformationImageToImageMetricv4` as the metric since it is fitted for a multi-modal registration.

```
typedef itk::RegularStepGradientDescentOptimizerv4< double > TOptimizerType;
typedef itk::MattesMutualInformationImageToImageMetricv4<
    FixedImageType,
    MovingImageType > MetricType;
typedef itk::ImageRegistrationMethodv4<
    FixedImageType,
    MovingImageType,
    TTransformType > TRegistrationType;
```

Then, all the components are instantiated using their `New()` method and connected to the registration object as in previous examples.

The output transform of the registration process will be constructed internally in the registration filter since the related *TransformType* is already passed to the registration method as a template parameter. However, we should provide an initial moving transform for the registration method if needed.

```
TTransformType::Pointer movingInitTx = TTransformType::New();
```

After setting the initial parameters, the initial transform can be passed to the registration filter by `SetMovingInitialTransform()` method.

```
transRegistration->SetMovingInitialTransform( movingInitTx );
```

We can use a `itk::CompositeTransform` to stack all the output transforms resulted from multiple stages. This composite transform should also hold the moving initial transform (if it exists) because as explained in section 3.6.1, the output of each registration stage does not include the input initial transform to that stage.

```
typedef itk::CompositeTransform< double,
    Dimension > CompositeTransformType;
CompositeTransformType::Pointer compositeTransform =
    CompositeTransformType::New();
compositeTransform->AddTransform( movingInitTx );
```

In the case of this simple example, the first stage is run only in one level of registration at a coarse resolution.

```
const unsigned int numberOfLevels1 = 1;

TRegistrationType::ShrinkFactorsArrayType shrinkFactorsPerLevel1;
shrinkFactorsPerLevel1.SetSize( numberOfLevels1 );
shrinkFactorsPerLevel1[0] = 3;

TRegistrationType::SmoothingSigmasArrayType smoothingSigmasPerLevel1;
smoothingSigmasPerLevel1.SetSize( numberOfLevels1 );
smoothingSigmasPerLevel1[0] = 2;

transRegistration->SetNumberOfLevels( numberOfLevels1 );
transRegistration->SetShrinkFactorsPerLevel( shrinkFactorsPerLevel1 );
transRegistration->SetSmoothingSigmasPerLevel( smoothingSigmasPerLevel1 );
```

Also, for this initial stage we can use a more aggressive parameter set for the optimizer by taking a big step size and relaxing stop criteria.

```
transOptimizer->SetLearningRate( 16 );
transOptimizer->SetMinimumStepLength( 1.5 );
```

Once all the registration components are in place, we trigger the registration process by calling `Update()` and add the result output transform to the final composite transform, so this composite transform can be used to initialize the next registration stage.

```
try
{
    transRegistration->Update();
    std::cout << "Optimizer stop condition: "
    << transRegistration->GetOptimizer()->GetStopConditionDescription()
    << std::endl;
}
catch( itk::ExceptionObject & err )
{
    std::cout << "ExceptionObject caught !" << std::endl;
    std::cout << err << std::endl;
    return EXIT_FAILURE;
}

compositeTransform->AddTransform(
    transRegistration->GetModifiableTransform() );
```

Now we can upgrade to an Affine transform as the second stage of registration process. The Affine-Transform 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 3.9.16. The instantiation of the transform type requires only the dimension of the space and the type used for representing space coordinates.

```
typedef itk::AffineTransform< double, Dimension > ATransformType;
```

We also use a different optimizer in configuration of the second stage while the metric is kept the same as before.

```
typedef itk::ConjugateGradientLineSearchOptimizerv4Template<
    double > AOptimizerType;
typedef itk::ImageRegistrationMethodv4<
    FixedImageType,
    MovingImageType,
    ATransformType > ARegistrationType;
```

Again all the components are instantiated using their `New()` method and connected to the registration object like in previous stages.

The current stage can be initialized using the initial transform of the registration and the result transform of the previous stage, so that both are concatenated into the composite transform.

```
affineRegistration->SetMovingInitialTransform( compositeTransform );
```

In Section 3.6.2 we showed the importance of center of rotation in the registration process. In Affine transforms, the center of rotation is defined by the fixed parameters set, which are set by default to [0, 0]. However, consider a situation where the origin of the virtual space, in which the registration is run, is far away from the zero origin. In such cases, leaving the center of rotation as the default value can make the optimization process unstable. Therefore, we are always interested to set the center of rotation to the center of virtual space which is usually the fixed image space.

Note that either center of gravity or geometrical center can be used as the center of rotation. In this example center of rotation is set to the geometrical center of the fixed image. We could also use `itk::ImageMomentsCalculator` filter to compute the center of mass.

Based on the above discussion, the user must set the fixed parameters of the registration transform outside of the registration method, so first we instantiate an object of the output transform type.

```
ATransformType::Pointer affineTx = ATransformType::New();
```

Then, we compute the physical center of the fixed image and set that as the center of the output Affine transform.

```
typedef FixedImageType::SpacingType SpacingType;
typedef FixedImageType::PointType OriginType;
typedef FixedImageType::RegionType RegionType;
typedef FixedImageType::SizeType SizeType;

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();

ATransformType::InputPointType centerFixed;
centerFixed[0] =
    fixedOrigin[0] + fixedSpacing[0] * fixedSize[0] / 2.0;
centerFixed[1] =
    fixedOrigin[1] + fixedSpacing[1] * fixedSize[1] / 2.0;

const unsigned int numberOfFixedParameters =
    affineTx->GetFixedParameters().Size();
ATransformType::ParametersType fixedParameters( numberOfFixedParameters );
for (unsigned int i = 0; i < numberOfFixedParameters; ++i)
{
    fixedParameters[i] = centerFixed[i];
}
affineTx->SetFixedParameters( fixedParameters );
```

Then, the initialized output transform should be connected to the registration object by using `SetInitialTransform()` method.

It is important to distinguish between the `SetInitialTransform()` and `SetMovingInitialTransform()` that was used to initialize the registration stage based on the results of the previous stages. You can assume that the first one is used for direct manipulation of the optimizable transform in current registration process.

```
affineRegistration->SetInitialTransform( affineTx );
```

The set of optimizable parameters in the Affine transform 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 on the order of 10.0 to 100.0. This difference in dynamic range negatively affects the performance of gradient descent optimizers. ITK provides some mechanisms to compensate for such differences in values among the parameters when they are passed to the optimizer.

The first 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. These scales are estimated by the user intuitively as shown in previous examples of this chapter. In our particular case, a common choice for the scale parameters is to set all those associated with the matrix coefficients to 1.0, that is, the first $N \times N$ factors. Then, we set the remaining scale factors to a small value.

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

$$\begin{bmatrix} P'_x \\ P'_y \end{bmatrix} = \begin{bmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \end{bmatrix} \quad (3.1)$$

Based on the above discussion, we need much smaller scales for translation parameters of vector **T** (T_x, T_y) compared to the parameters of matrix **M** ($M_{11}, M_{12}, M_{21}, M_{22}$). However, it is not easy to have an intuitive estimation of all parameter scales when we have to deal with a large parameter space.

Fortunately, ITKv4 provides a framework for automated parameter scaling. `itk::RegistrationParameterScalesEstimator` vastly reduces the difficulty of tuning parameters for different transform/metric combinations. Parameter scales are estimated by analyzing the result of a small parameter update on the change in the magnitude of physical space deformation induced by the transformation.

The impact from a unit change of a parameter may be defined in multiple ways, such as the maximum shift of voxels in index or physical space, or the average norm of transform Jacobian. Filters `itk::RegistrationParameterScalesFromPhysicalShift` and `itk::RegistrationParameterScalesFromIndexShift` use the first definition to estimate the scales, while the `itk::RegistrationParameterScalesFromJacobian` filter estimates scales based on the later definition. In all methods, the goal is to rescale the transform parameters such that a unit change of each *scaled parameter* will have the same impact on deformation.

In this example the first filter is chosen to estimate the parameter scales. The scales estimator will then be passed to optimizer.

```
typedef itk::RegistrationParameterScalesFromPhysicalShift<
    MetricType> ScalesEstimatorType;
ScalesEstimatorType::Pointer scalesEstimator =
    ScalesEstimatorType::New();
scalesEstimator->SetMetric( affineMetric );
scalesEstimator->SetTransformForward( true );

affineOptimizer->SetScalesEstimator( scalesEstimator );
```

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.

However, we need not worry about the above considerations. Thanks to the *ScalesEstimator*, the initial step size can also be estimated automatically, either at each iteration or only at the first iteration. In this example we choose to estimate learning rate once at the begining of the registration process.

```
affineOptimizer->SetDoEstimateLearningRateOnce( true );
affineOptimizer->SetDoEstimateLearningRateAtEachIteration( false );
```

At the second stage, we run two levels of registration, where the second level is run in full resolution in which we do the final adjustments of the output parameters.

```
const unsigned int numberOfLevels2 = 2;

ARegistrationType::ShrinkFactorsArrayType shrinkFactorsPerLevel2;
shrinkFactorsPerLevel2.SetSize( numberOfLevels2 );
shrinkFactorsPerLevel2[0] = 2;
shrinkFactorsPerLevel2[1] = 1;

ARegistrationType::SmoothingSigmasArrayType smoothingSigmasPerLevel2;
smoothingSigmasPerLevel2.SetSize( numberOfLevels2 );
smoothingSigmasPerLevel2[0] = 1;
smoothingSigmasPerLevel2[1] = 0;

affineRegistration->SetNumberOfLevels( numberOfLevels2 );
affineRegistration->SetShrinkFactorsPerLevel( shrinkFactorsPerLevel2 );
affineRegistration->SetSmoothingSigmasPerLevel( smoothingSigmasPerLevel2 );
```

Finally we trigger the registration process by calling `Update()` and add the output transform of the last stage to the composite transform. This composite transform will be considered as the final transform of this multistage registration process and will be used by the resampler to resample the moving image in to the virtual domain space (fixed image space if there is no fixed initial transform).

```

try
{
    affineRegistration->Update();
    std::cout << "Optimizer stop condition: "
    << affineRegistration->GetOptimizer()->GetStopConditionDescription()
    << std::endl;
}
catch( itk::ExceptionObject & err )
{
    std::cout << "ExceptionObject caught !" << std::endl;
    std::cout << err << std::endl;
    return EXIT_FAILURE;
}

compositeTransform->AddTransform(
    affineRegistration->GetModifiableTransform() );

```

Let's execute this example using the following multi-modality images:

- BrainT1SliceBorder20.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 3.36.

The registration converges after 5 iterations in the translation stage. Also, in the second stage, the registration converges after 46 iterations in the first level, and 6 iterations in the second level. The final results when printed as an array of parameters are:

Initial parameters of the registration process:

[3, 5]

Translation parameters after first registration stage:

[9.0346, 10.8303]

Affine parameters after second registration stage:

[0.9864, -0.1733, 0.1738, 0.9863, 0.9693, 0.1482]

As it can be seen, the translation parameters after the first stage compensate most of the offset between the fixed and moving images. When the images are close to each other, the affine registration is run for the rotation and the final match. By reordering the Affine array of parameters as coefficients of matrix \mathbf{M} and vector \mathbf{T} they can now be seen as

$$\mathbf{M} = \begin{bmatrix} 0.9864 & -0.1733 \\ 0.1738 & 0.9863 \end{bmatrix} \text{ and } \mathbf{T} = \begin{bmatrix} 0.9693 \\ 0.1482 \end{bmatrix} \quad (3.2)$$

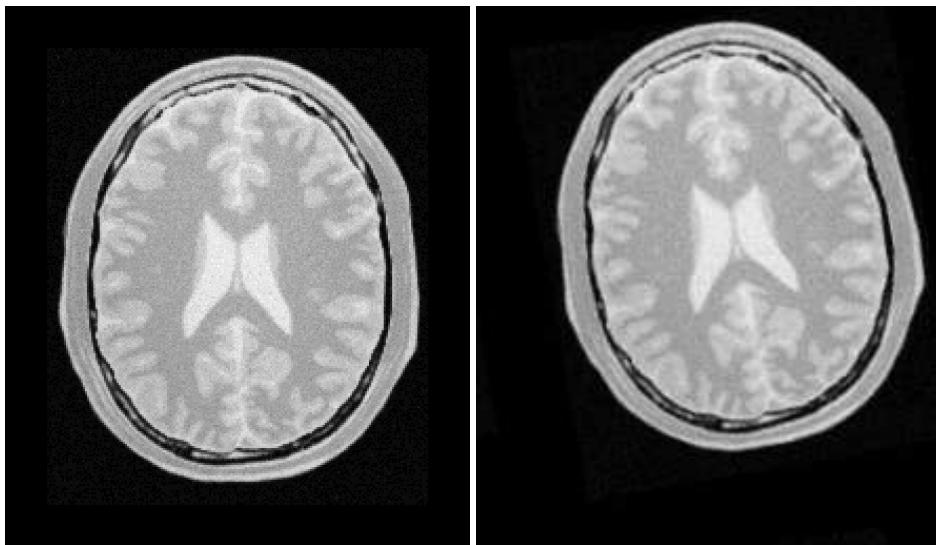


Figure 3.36: Fixed and moving images provided as input to the registration method using the `AffineTransform`.

In this form, it is easier to interpret the effect of the transform. The matrix \mathbf{M} is responsible for scaling, rotation and shearing while \mathbf{T} is responsible for translations.

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 9.975 degrees, which is approximately the intentional misalignment of 10.0 degrees.

Also, let's compute the total translation values resulting from initial transform, translation transform, and the Affine transform together.

In X direction:

$$3 + 9.0346 + 0.9693 = 13.0036 \quad (3.3)$$

In Y direction:

$$5 + 10.8303 + 0.1482 = 15.9785 \quad (3.4)$$

It can be seen that the translation values closely match the true misalignment introduced in the moving image.

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. However, in this example we used a linear interpolator for all stages and different registration levels since it is so fast.

The result of resampling the moving image is presented in the left image of Figure 3.37. The center

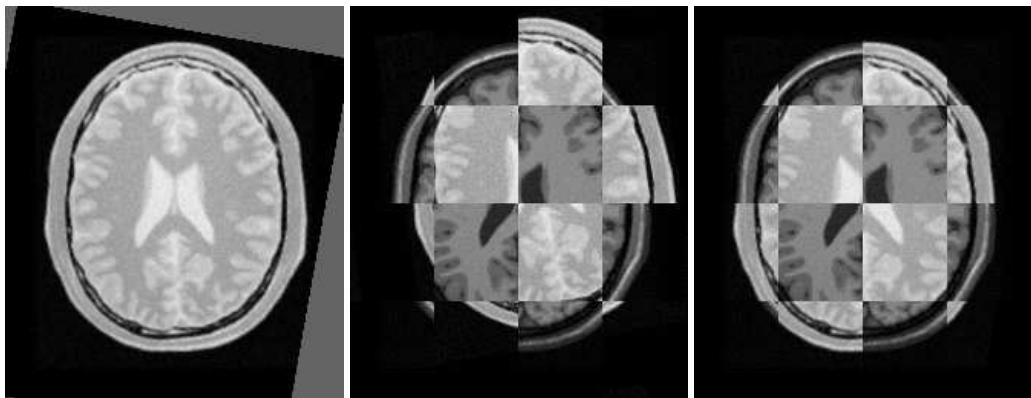


Figure 3.37: Mapped moving image (left) and composition of fixed and moving images before (center) and after (right) registration.

and right images of the figure depict a checkerboard composite of the fixed and moving images before and after registration.

3.8.2 Cascaded Multistage Registration

The source code for this section can be found in the file `MultiStageImageRegistration2.cxx`.

This examples shows how different stages can be cascaded together directly in a multistage registration process. The example code is, for the most part, identical to the previous multistage example. The main difference is that no initial transform is used, and the output of the first stage is directly linked to the second stage, and the whole registration process is triggered only once by calling `Update()` after the last stage stage.

We will focus on the most relevant changes in current code and skip all the similar parts already explained in the previous example.

Let's start by defining different types of the first stage.

```
typedef itk::TranslationTransform< double, Dimension > TTransformType;
typedef itk::RegularStepGradientDescentOptimizerv4<double> TOptimizerType;
typedef itk::MattesMutualInformationImageToImageMetricv4<
    FixedImageType,
    MovingImageType > MetricType;
typedef itk::ImageRegistrationMethodv4<
    FixedImageType,
    MovingImageType > TRegistrationType;
```

Type definitions are the same as previous example with an important subtle change: the transform

type is not passed to the registration method as a template parameter anymore. In this case, the registration filter will consider the transform base class `itk::Transform` as the type of its output transform.

Instead of passing the transform type, we create an explicit instantiation of the transform object outside of the registration filter, and connect that to the registration object using the `SetInitialTransform()` method. Also, by calling `InPlaceOn()` method, this transform object will be the output transform of the registration filter or will be grafted to the output.

```
TTransformType::Pointer translationTx = TTransformType::New();

transRegistration->SetInitialTransform( translationTx );
transRegistration->InPlaceOn();
```

Also, there is no initial transform defined for this example.

As in the previous example, the first stage is run using only one level of registration at a coarse resolution level. However, notice that we do not need to update the translation registration filter at this step since the output of this stage will be directly connected to the initial input of the next stage. Due to ITK's pipeline structure, when we call the `Update()` at the last stage, the first stage will be updated as well.

Now we upgrade to an Affine transform as the second stage of registration process, and as before, we initially define and instantiate different components of the current registration stage. We have used a new optimizer but the same metric in new configurations.

```
typedef itk::AffineTransform< double, Dimension > ATransformType;
typedef itk::ConjugateGradientLineSearchOptimizerv4Template<
    double > AOptimizerType;
typedef itk::ImageRegistrationMethodv4<
    FixedImageType,
    MovingImageType > ARegistrationType;
```

Again notice that `TransformType` is not passed to the type definition of the registration filter. It is important because when the registration filter considers transform base class `itk::Transform` as the type of its output transform, it prevents the type mismatch when the two stages are cascaded to each other.

Then, all components are instantiated using their `New()` method and connected to the registration object among the transform type. Despite the previous example, here we use the fixed image's center of mass to initialize the fixed parameters of the Affine transform. `itk::ImageMomentsCalculator` filter is used for this purpose.

```
typedef itk::ImageMomentsCalculator<
    FixedImageType > FixedImageCalculatorType;

FixedImageCalculatorType::Pointer fixedCalculator =
    FixedImageCalculatorType::New();
fixedCalculator->SetImage( fixedImage );
fixedCalculator->Compute();

FixedImageCalculatorType::VectorType fixedCenter =
    fixedCalculator->GetCenterOfGravity();
```

Then, we initialize the fixed parameters (center of rotation) in the Affine transform and connect that to the registration object.

```
ATransformType::Pointer affineTx = ATransformType::New();

const unsigned int numberOfFixedParameters =
    affineTx->GetFixedParameters().Size();
ATransformType::ParametersType fixedParameters( numberOfFixedParameters );
for (unsigned int i = 0; i < numberOfFixedParameters; ++i)
{
    fixedParameters[i] = fixedCenter[i];
}
affineTx->SetFixedParameters( fixedParameters );

affineRegistration->SetInitialTransform( affineTx );
affineRegistration->InPlaceOn();
```

Now, the output of the first stage is wrapped through a [itk::DataObjectDecorator](#) and is passed to the input of the second stage as the moving initial transform via `SetMovingInitialTransformInput()` method. Note that this API has an “Input” word attached to the name of another initialization method `SetMovingInitialTransform()` that already has been used in previous example. This extension means that the following API expects a data object decorator type.

```
affineRegistration->SetMovingInitialTransformInput(
    transRegistration->GetTransformOutput() );
```

Second stage runs two levels of registration, where the second level is run in full resolution.

Once all the registration components are in place, finally we trigger the whole registration process, including two cascaded registration stages, by calling `Update()` on the registration filter of the last stage, which causes both stages be updated.

```

try
{
    affineRegistration->Update();
    std::cout << "Optimizer stop condition: "
        << affineRegistration->
            GetOptimizer()->GetStopConditionDescription()
        << std::endl;
}
catch( itk::ExceptionObject & err )
{
    std::cout << "ExceptionObject caught !" << std::endl;
    std::cout << err << std::endl;
    return EXIT_FAILURE;
}

```

Finally, a composite transform is used to concatenate the results of all stages together, which will be considered as the final output of this multistage process and will be passed to the resampler to resample the moving image into the virtual domain space (fixed image space if there is no fixed initial transform).

```

typedef itk::CompositeTransform< double,
    Dimension > CompositeTransformType;
CompositeTransformType::Pointer compositeTransform =
    CompositeTransformType::New();
compositeTransform->AddTransform( translationTx );
compositeTransform->AddTransform( affineTx );

```

Let's execute this example using the same multi-modality images as before. The registration converges after 6 iterations in the first stage, also in 45 and 11 iterations corresponding to the first level and second level of the Affine stage. The final results when printed as an array of parameters are:

Translation parameters after first registration stage:
[11.600, 15.1814]

Affine parameters after second registration stage:
[0.9860, -0.1742, 0.1751, 0.9862, 0.9219, 0.8023]

Let's reorder the Affine array of parameters again as coefficients of matrix **M** and vector **T**. They can now be seen as

$$M = \begin{bmatrix} 0.9860 & -0.1742 \\ 0.1751 & 0.9862 \end{bmatrix} \text{ and } T = \begin{bmatrix} 0.9219 \\ 0.8023 \end{bmatrix} \quad (3.5)$$

10.02 degrees is the rotation value computed from the affine matrix parameters, which approximately equals the intentional misalignment.

Also for the total translation value resulted from both transforms, we have:

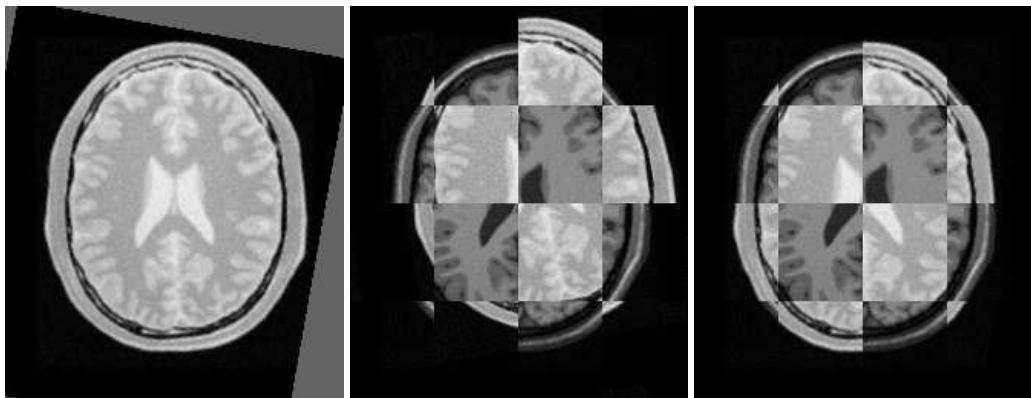


Figure 3.38: Mapped moving image (left) and composition of fixed and moving images before (center) and after (right) registration.

In X direction:

$$11.6004 + 0.9219 = 12.5223 \quad (3.6)$$

In Y direction:

$$15.1814 + 0.8023 = 15.9837 \quad (3.7)$$

These results closely match the true misalignment introduced in the moving image.

The result of resampling the moving image is presented in the left image of Figure 3.38. The center and right images of the figure depict a checkerboard composite of the fixed and moving images before and after registration.

With the completion of these examples, we will now review the main features of the components forming the registration framework.

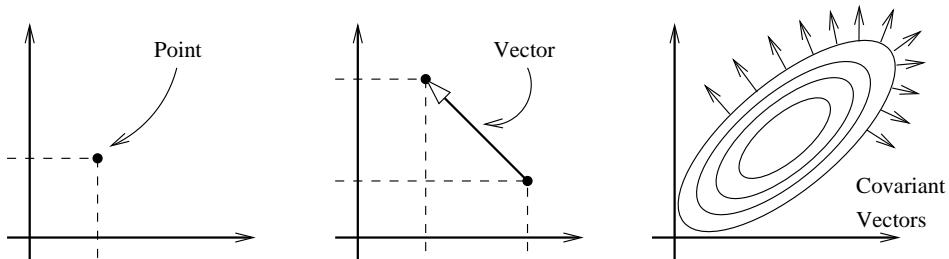


Figure 3.39: Geometric representation objects in ITK.

3.9 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.

3.9.1 Geometrical Representation

ITK implements a consistent geometric representation of space. The characteristics of classes involved in this representation are summarized in Table 3.1. 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 the Data Representation chapter of Book 1. Each one of these classes behaves differently under spatial transformations. It is therefore quite important to keep their distinction clear. Figure 3.39 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 deserves further comments is the `itk::Vector`. This ITK class tends to be misinterpreted as a container of elements instead of a geometrical object. This is a common misconception originating from the colloquial use by computer scientists and software engineers of

Class	Geometrical concept
<code>itk::Point</code>	Position in space. In N -dimensional space it is represented by an array of N numbers associated with space coordinates.
<code>itk::Vector</code>	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.
<code>itk::CovariantVector</code>	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 3.1: Summary of objects representing geometrical concepts in ITK.

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)[25]. 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 [18].

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” [5]. 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 [36].

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 continues using the word “Vector” in this manner [5, 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 which require insertion and deletion of elements, and those which 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 compartmentalizes the behavior of the individual classes. The use of the `itk::FixedArray` as the base class of the `itk::Point`, the `itk::Vector` and the `itk::CovariantVector` was a design decision based on the decision to use the correct nomenclature.

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 that it will have 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 can perform ITK operations such as:

- `Vector = Point - Point`
- `Point += Vector`
- `Point -= Vector`
- `Point = BarycentricCombination(Point, Point)`

and you cannot (because you **should not**) perform operations such as

- `Point = Point * Scalar`

- $\text{Point} = \text{Point} + \text{Point}$
- $\text{Point} = \text{Point} / \text{Scalar}$

The `itk::Vector` is, by Hamilton's definition, the subtraction between two points. Therefore a Vector must satisfy the following basic operations:

- $\text{Vector} = \text{Point} - \text{Point}$
- $\text{Point} = \text{Point} + \text{Vector}$
- $\text{Point} = \text{Point} - \text{Vector}$
- $\text{Vector} = \text{Vector} + \text{Vector}$
- $\text{Vector} = \text{Vector} - \text{Vector}$

An `itk::Vector` object is intended to be instantiated over elements that support mathematical operation such as addition, subtraction and multiplication by scalars.

3.9.2 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 elements are the partial derivatives of the output point with respect to the array of parameters that defines the transform:⁹

⁹Note 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. [18]

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 3.2: Characteristics of the identity transform.

$$J = \begin{bmatrix} \frac{\partial x_1}{\partial p_1} & \frac{\partial x_1}{\partial p_2} & \dots & \frac{\partial x_1}{\partial p_m} \\ \frac{\partial x_2}{\partial p_1} & \frac{\partial x_2}{\partial p_2} & \dots & \frac{\partial x_2}{\partial p_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial x_n}{\partial p_1} & \frac{\partial x_n}{\partial p_2} & \dots & \frac{\partial x_n}{\partial p_m} \end{bmatrix} \quad (3.8)$$

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 differences 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.

3.9.3 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 3.2

3.9.4 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

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 have the same number of dimensions.

Table 3.3: Characteristics of the TranslationTransform class.

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 3.3.

3.9.5 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 on the basic geometric objects.

$$\begin{array}{lll} \text{Point} & \mathbf{P}' = T(\mathbf{P}) : \mathbf{P}'_i = \mathbf{P}_i \cdot \mathbf{S}_i \\ \text{Vector} & \mathbf{V}' = T(\mathbf{V}) : \mathbf{V}'_i = \mathbf{V}_i \cdot \mathbf{S}_i \\ \text{CovariantVector} & \mathbf{C}' = T(\mathbf{C}) : \mathbf{C}'_i = \mathbf{C}_i / \mathbf{S}_i \end{array} \quad (3.9)$$

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.

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} S_1 & 0 & 0 \\ 0 & S_2 & 0 \\ 0 & 0 & S_3 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad (3.10)$$

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 <i>dividing</i> 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 have the same number of dimensions.

Table 3.4: Characteristics of the ScaleTransform class.

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 $(1.0 + \epsilon)$ is quite different from a scale variation of $(5.0 + \epsilon)$.

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 3.4.

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 <i>dividing</i> 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 have 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 3.5: Characteristics of the ScaleLogarithmicTransform class.

3.9.6 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.

3.9.7 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 equation illustrates the effect of this transform on a 2D point,

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \end{bmatrix} \quad (3.11)$$

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 linearly independent parameters. Typical optimizers make

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 3.6: Characteristics of the Euler2DTransform class.

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 3.6.

3.9.8 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 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.

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 3.7: Characteristics of the CenteredRigid2DTransform class.

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') ,

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \cdot \begin{bmatrix} x - C_x \\ y - C_y \end{bmatrix} + \begin{bmatrix} T_x + C_x \\ T_y + C_y \end{bmatrix} \quad (3.12)$$

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 3.7.

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 3.8: Characteristics of the `Similarity2DTransform` class.

3.9.9 `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 `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') ,

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \lambda & 0 \\ 0 & \lambda \end{bmatrix} \cdot \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} x - C_x \\ y - C_y \end{bmatrix} + \begin{bmatrix} T_x + C_x \\ T_y + C_y \end{bmatrix} \quad (3.13)$$

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 3.8.

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.

Behavior	Number of Parameters	Parameter Ordering	Restrictions
<p>Represents a 3D rotation and a 3D translation. The rotation is specified as a quaternion, defined by a set of four numbers \mathbf{q}. The relationship between quaternion and rotation about vector \mathbf{n} by angle θ is as follows:</p> $\mathbf{q} = (\mathbf{n} \sin(\theta/2), \cos(\theta/2))$ <p>Note that if the quaternion is not of unit length, scaling will also result.</p>	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 3.9: Characteristics of the QuaternionRigidTransform class.

3.9.10 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 adapted for a quaternion space. This class ensures 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 3.9.

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 of 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.

Behavior	Number of Parameters	Parameter Ordering	Restrictions
Represents a 3D 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 3.10: Characteristics of the Versor Transform

3.9.11 VersorTransform

By definition, a *Versor* is the rotational part of a Quaternion. It can also be defined as a *unit-quaternion* [25, 28]. 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 3.10.

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.

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 [25].

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.

3.9.12 VersorRigid3DTransform

The `itk::VersorRigid3DTransform` implements a rigid transformation in 3D 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

Behavior	Number of Parameters	Parameter Ordering	Restrictions
Represents a 3D rotation and a 3D 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 3.11: Characteristics of the `VersorRigid3DTransform` class.

respect 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 3.11. This transform is probably the best option to use when dealing with rigid transformations in 3D.

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::VersorRigid3DTransformOptimizer` 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.

3.9.13 Euler3DTransform

The `itk::Euler3DTransform` implements a rigid transformation in 3D 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 3.12.

Three rotational parameters are non-linear and do not behave like Vector spaces. This must be taken into account when selecting an optimizer to work with this transform and when fine tuning the parameters of the optimizer. It is strongly recommended to use this transform by introducing very

Behavior	Number of Parameters	Parameter Ordering	Restrictions
Represents a rigid rotation in 3D space. That is, a rotation followed by a 3D translation. The rotation is specified by three angles representing rotations to be applied around the X, Y and Z axes 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 axes, and the last three parameters are the translations along each dimension.	Only defined for three-dimensional input and output spaces.

Table 3.12: Characteristics of the Euler3DTransform class.

small variations on the rotational components. A small rotation will be in the range of 1 degree, which in radians is approximately 0.01745.

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.

3.9.14 Similarity3DTransform

The `itk::Similarity3DTransform` implements a similarity transformation in 3D 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 user 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 3.13.

The scaling and rotational spaces are non-linear and do not behave like Vector spaces. This must be taken into account when selecting an optimizer to work with this transform and when fine tuning the parameters of the optimizer.

Behavior	Number of Parameters	Parameter Ordering	Restrictions
Represents a 3D rotation, a 3D 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, which is the same center used for scaling.	7	The first three parameters define the Versor, the next three parameters the translation in each dimension, and the last parameter is the isotropic scaling factor.	Only defined for three-dimensional input and output spaces.

Table 3.13: Characteristics of the Similarity3DTransform class.

3.9.15 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 in 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 3.14. 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 VersorRigid3DTransformOptimizer in order to optimize the parameters of this class.

3.9.16 AffineTransform

The `itk::AffineTransform` is one of the most popular transformations used for image registration. Its main advantage comes from its representation as a linear transformation. The main features

Behavior	Number of Parameters	Parameter Ordering	Restrictions
Represents a rigid <i>3D</i> 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 specify 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 3.14: 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 3.15: Characteristics of the AffineTransform class.

of this transform are presented in Table 3.15.

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 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 3.14 illustrates the effect of applying the `AffineTransform` to a point in 3D space.

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} M_{00} & M_{01} & M_{02} \\ M_{10} & M_{11} & M_{12} \\ M_{20} & M_{21} & M_{22} \end{bmatrix} \cdot \begin{bmatrix} x - C_x \\ y - C_y \\ z - C_z \end{bmatrix} + \begin{bmatrix} T_x + C_x \\ T_y + C_y \\ T_z + C_z \end{bmatrix} \quad (3.14)$$

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 a wire-frame cube which is shown in a 3D visualization display.

3.9.17 BSplineDeformableTransform

The `itk::BSplineDeformableTransform` is designed to be used for solving deformable registration problems. This transform is equivalent to generating 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, which is usually referred to as the BSpline grid.

Behavior	Number of Parameters	Parameter Ordering	Restrictions
Represents a free-form 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 being able to compute deformable registration. It also has the disadvantage of a very high-dimensional parametric space, and therefore requiring long computation times.

Table 3.16: Characteristics of the BSplineDeformableTransform class.

The BSplineDeformableTransform is not flexible enough to account 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 applied to points before they are mapped with the displacement field.

This transform does not provide functionality for mapping Vectors nor CovariantVectors—only Points can be mapped. This is because the variations of a vector under a deformable transform actually depend on the location of the vector in space. In other words, Vectors 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 was proposed in the following papers [53, 40, 41].

3.9.18 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 in 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* [15] and the papers by Rohr *et. al* [51, 52].

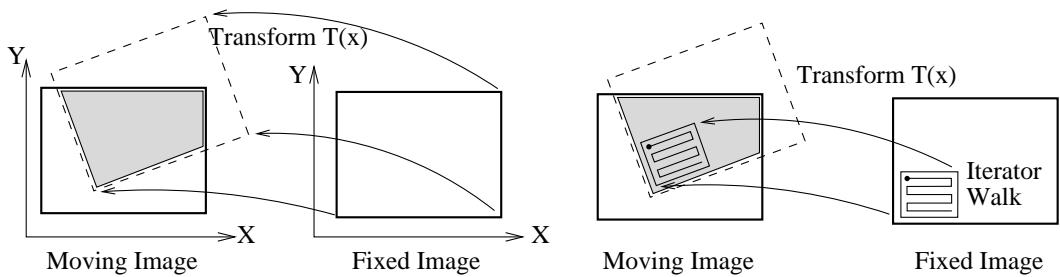


Figure 3.40: The moving image is mapped into the fixed image space under some spatial transformation. An iterator walks through the fixed image and its coordinates are mapped onto the moving image.

3.10 Interpolators

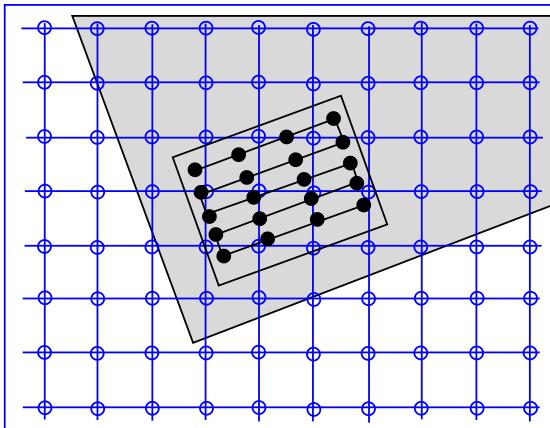


Figure 3.41: Grid positions of the fixed image map to non-grid positions of the moving image.

the moving image space in order to find the homologous pixel.

Figure 3.41 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:

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 3.40 (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

- `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.

3.10.1 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.

3.10.2 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.

3.10.3 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 3.42 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.

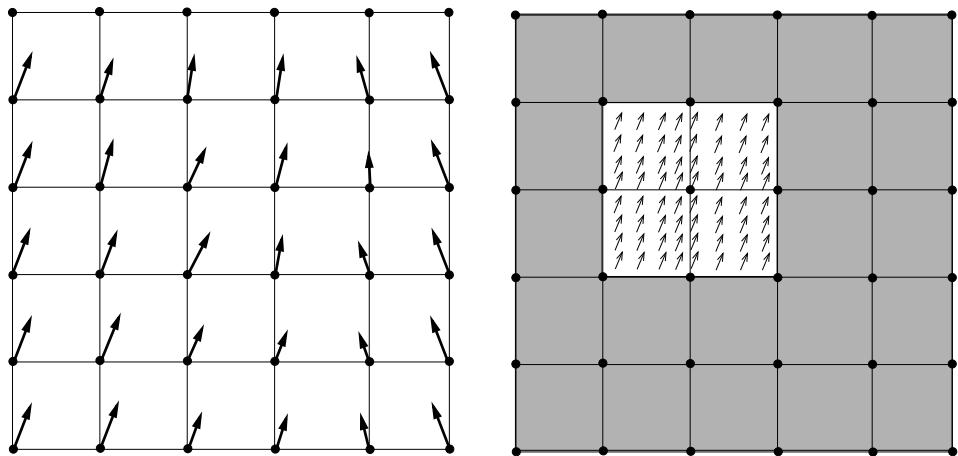


Figure 3.42: The left side illustrates the B-Spline grid and the deformations that are known on those nodes. The right side illustrates the region where interpolation is possible when the B-Spline is of cubic order. The small arrows represent deformation values that were interpolated from the grid deformations shown on the left side of the diagram.

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.

3.10.4 Windowed Sinc Interpolation

The `itk::WindowedSincInterpolateImageFunction` is the best possible interpolator for data that have 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

$$\text{sinc}(x) = \sin(x)/x \quad (3.15)$$

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 an 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* [42].

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.

$$I(x, y) = \sum_{i=\lfloor x \rfloor + 1 - m}^{\lfloor x \rfloor + m} \sum_{j=\lfloor y \rfloor + 1 - m}^{\lfloor y \rfloor + m} I_{i,j} K(x - i) K(y - j) \quad (3.16)$$

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.

$$K(t) = w(t)\text{sinc}(t) = w(t) \frac{\sin(\pi t)}{\pi t} \quad (3.17)$$

Some of the windows that can be used with this interpolator are

Cosinus window

$$w(x) = \cos\left(\frac{\pi x}{2m}\right) \quad (3.18)$$

Hamming window

$$w(x) = 0.54 + 0.46\cos\left(\frac{\pi x}{m}\right) \quad (3.19)$$

Welch window

$$w(x) = 1 - \left(\frac{x^2}{m^2}\right) \quad (3.20)$$

Lancos window

$$w(x) = \text{sinc}\left(\frac{x}{m}\right) \quad (3.21)$$

Blackman window

$$w(x) = 0.42 + 0.5\cos\left(\frac{\pi x}{m}\right) + 0.08\cos\left(\frac{2\pi x}{m}\right) \quad (3.22)$$

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 `ImageInterpolateFunction`. 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 the transform has been found using another less expensive interpolator in the registration process.

3.11 Metrics

In ITK, `itk::ImageToImageMetricv4` 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 matching Metric class controls most parts of the registration process since it handles fixed, moving and virtual images as well as fixed and moving transforms and interpolators. 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 virtual lattice. For each point, the corresponding fixed and moving image positions are computed using the fixed initial transform and the moving transform with the specified parameters. Then, the fixed and moving interpolators are used to compute the fixed and moving image's intensities at the mapped positions. Details on this mapping are illustrated in Figures 3.40 and 3.41 assuming that virtual lattice is the same as the fixed image lattice, which is usually the case in practice.

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 ITKv4 registration framework:

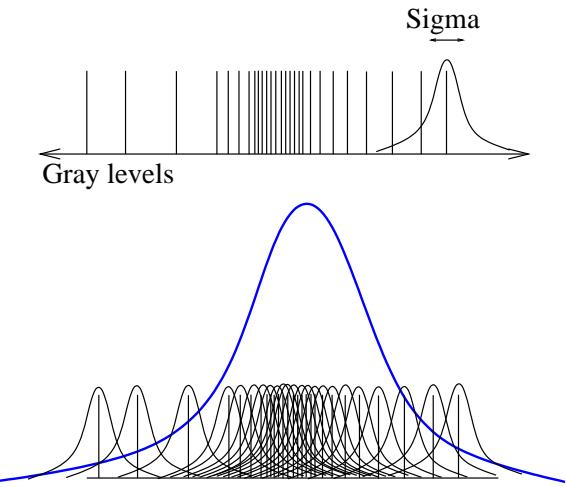


Figure 3.43: In Parzen windowing, a continuous density function is constructed by superimposing kernel functions (Gaussian function in this case) centered on the intensity samples obtained from the image.

- Mean squares

```
itk::MeanSquaresImageToImageMetricv4
```

- Correlation

```
itk::CorrelationImageToImageMetricv4
```

- Mutual information by Mattes

```
itk::MattesMutualInformationImageToImageMetricv4
```

- Joint histogram mutual information

```
itk::JointHistogramMutualInformationHistogramImageToImageMetricv4
```

- Demons metric

```
itk::DemonsImageToImageMetricv4
```

- ANTS neighborhood correlation metric

```
itk::ANTSNeighborhoodCorrelationImageToImageMetricv4
```

Also, in case you are interested in using the legacy ITK registration framework, the following is the list of metrics currently available in ITKv3:

- 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 the ITKv4 metric types in detail. You can check ITK descriptions in doxygen for details about ITKv3 metric classes.

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 .

3.11.1 Mean Squares Metric

The `itk::MeanSquaresImageToImageMetricv4` computes the mean squared pixel-wise difference in intensity between image A and B over a user defined region:

$$MS(A, B) = \frac{1}{N} \sum_{i=1}^N (A_i - B_i)^2 \quad (3.23)$$

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
`MeanSquaresImageMetric1.cxx`.

This example illustrates how to explore the domain of an image metric. This is a useful exercise before starting a registration process, since familiarity with the characteristics of the metric is fundamental for appropriate selection of the optimizer and its parameters used to drive the registration process. This process helps 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 "itkMeanSquaresImageToImageMetricv4.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 float 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::MeanSquaresImageToImageMetricv4<
    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->SetMovingInterpolator( interpolator );

metric->SetFixedImage( fixedImage );
metric->SetMovingImage( movingImage );
```

Note that the `SetTransform()` method is equivalent to the `SetMovingTransform()` function. In this example there is no need to use the `SetFixedTransform()`, since the virtual domain is assumed

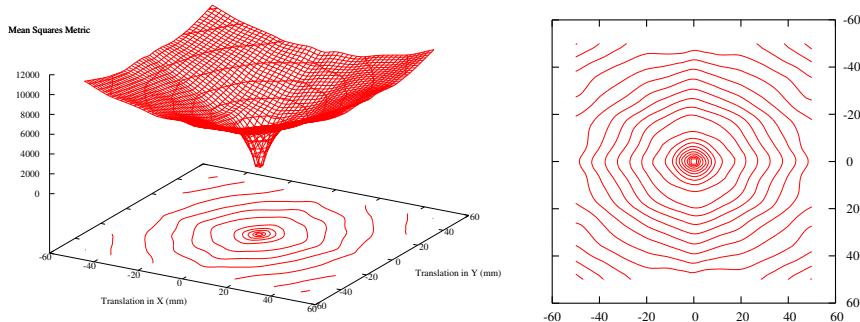


Figure 3.44: Plots of the Mean Squares Metric for an image compared to itself under multiple translations.

to be the same as the fixed image domain set as following.

```
metric->SetVirtualDomainFromImage( fixedImage );
```

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.

```
MetricType::MovingTransformParametersType 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;
        metric->SetParameters( displacement );
        const double value = metric->GetValue();
        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 3.44. 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 Gnuplot¹⁰ scripts used for generating the graphics in Figure 3.44 are available in the directory

`ITKSoftwareGuide/SoftwareGuide/Art`

Of course, this plotting exercise becomes more challenging when the transform has more than three parameters, and when those parameters have very different value ranges. In those cases it 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.

3.11.2 Normalized Correlation Metric

The `itk::CorrelationImageToImageMetricv4` computes pixel-wise cross-correlation and normalizes it by the square root of the autocorrelation of the images:

$$NC(A, B) = -1 \times \frac{\sum_{i=1}^N (A_i \cdot B_i)}{\sqrt{\sum_{i=1}^N A_i^2 \cdot \sum_{i=1}^N B_i^2}} \quad (3.24)$$

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.

3.11.3 Mutual Information Metric

The `itk::MattesMutualInformationImageToImageMetricv4` 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 [47].

¹⁰<http://www.gnuplot.info>

Mutual information is defined in terms of entropy. Let

$$H(A) = - \int p_A(a) \log p_A(a) da \quad (3.25)$$

be the entropy of random variable A , $H(B)$ the entropy of random variable B and

$$H(A, B) = \int p_{AB}(a, b) \log p_{AB}(a, b) da db \quad (3.26)$$

be the joint entropy of A and B . If A and B are independent, then

$$p_{AB}(a, b) = p_A(a)p_B(b) \quad (3.27)$$

and

$$H(A, B) = H(A) + H(B). \quad (3.28)$$

However, if there is any dependency, then

$$H(A, B) < H(A) + H(B). \quad (3.29)$$

The difference is called Mutual Information : $I(A, B)$

$$I(A, B) = H(A) + H(B) - H(A, B) \quad (3.30)$$

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 3.43:

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.

$$p(a) \approx P^*(a) = \frac{1}{N} \sum_{s_j \in S} K(a - s_j) \quad (3.31)$$

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.

Mattes et al. Implementation

The implementation of mutual information metric available in ITKv4 follows the method specified by Mattes et al. in [40] and is implemented by the `itk::MattesMutualInformationImageToImageMetricv4` 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 a ratio of the total number of samples and is set using the `SetMetricSamplingPercentage()` method directly from the registration framework `itk::ImageRegistrationMethodv4`. Also, The number of bins used to compute the entropy values is set in the metric class via the `SetNumberOfHistogramBins()` method.

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.

3.11.4 Normalized Mutual Information Metric

Given two images, A and B , the normalized mutual information may be computed as

$$NMI(A, B) = 1 + \frac{I(A, B)}{H(A, B)} = \frac{H(A) + H(B)}{H(A, B)} \quad (3.32)$$

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 3.11.3. Details of the implementation may be found in [24].

3.11.5 Demons metric

The implementation of the `itk::DemonsImageToImageMetricv4` metric is taken from `itk::DemonsRegistrationFunction`.

The metric derivative can be calculated using image derivatives either from the fixed or moving images. The default is to use fixed-image gradients. See `ObjectToObjectMetric::SetGradientSource` to change this behavior.

An intensity threshold is used, below which image pixels are considered equal for the purpose of derivative calculation. The threshold can be changed by calling `SetIntensityDifferenceThreshold`.

Note that this metric supports only moving transforms with local support and with a number of local parameters that match the moving image dimension. In particular, it's meant to be used with `itk::DisplacementFieldTransform` and derived classes.

3.11.6 ANTS neighborhood correlation metric

The `itk::ANTSNeighborhoodCorrelationImageToImageMetricv4` metric computes normalized cross correlation using a small neighborhood for each voxel between two images, with speed optimizations for dense registration.

Around each voxel, the neighborhood is defined as a N-Dimensional rectangle centered at the voxel. The size of the rectangle is $2*\text{radius}+1$. Normalized correlation between neighborhoods of the fixed image and the moving image are averaged over the whole image as the final metric. A radius less than 2 can be unstable. 2 is the default.

3.12 Optimizers

Optimization algorithms are encapsulated as `itk::ObjectToObjectOptimizer` objects within ITKv4. Optimizers are generic and can be used for applications other than registration. Within the registration framework, subclasses of `itk::SingleValuedNonLinearVnlOptimizerv4` are implemented as a wrap around already implemented vnl classes.

The basic input to an optimizer is a cost function or metric object. In the context of registration, `itk::ImageToImageMetricv4` classes provide this functionality. The metric is 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 parameter 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 using `SetScales()`.

Estimating the scales parameters can also be done automatically using the `itk::OptimizerParameterScalesEstimatorTemplate` and its subclasses. The scales estimator object is then set to the optimizer via `SetScalesEstimator()`.

Despite the old version of ITK, there are only *Single Valued* types of optimizers available in ITKv4, which are suitable for dealing with cost functions that return a single value. These are indeed the most common type of cost functions, and are also known as *Single Valued* functions.

The types of single valued optimizers currently available in ITKv4 are:

- **Amoeba:** Nelder-Mead downhill simplex. This optimizer is actually implemented in the vxl/vnl numerics toolkit. The ITK class `itk::AmoebaOptimizerv4` is merely an adaptor class.
- **Gradient Descent:** Advances parameters in the direction of the gradient where the step size is governed by a learning rate (`itk::GradientDescentOptimizerv4`).
- **Gradient Descent Line Search:** Gradient descent with a golden section line search. `itk::GradientDescentLineSearchOptimizerv4` implements a simple gradient descent optimizer that is followed by a line search to find the best value for the learning rate.
- **Conjugate Gradient Descent Line Search:** Advances parameters in the direction of the Polak-Ribiere conjugate gradient where a line search is used to find the best value for the learning rate (`itk::ConjugateGradientLineSearchOptimizerv4`).
- **Quasi Newton:** Implements a Quasi-Newton optimizer with BFGS Hessian estimation. Second order approximation of the cost function is usually more efficient since it estimates the

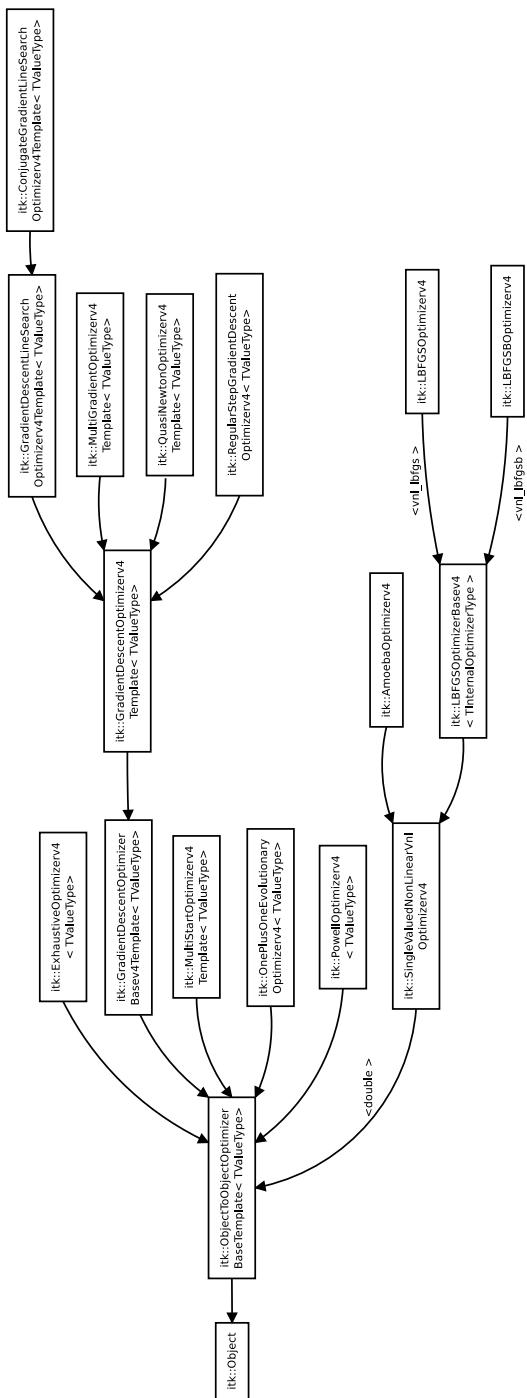


Figure 3.45: Class diagram of the optimizersv4 hierarchy.

descent or ascent direction more precisely. However, computation of Hessian is usually expensive or unavailable. Alternatively Quasi-Newton methods can estimate a Hessian from the gradients in previous steps. Here a specific Quasi-Newton method, BFGS, is used to compute the Quasi-Newton steps (`itk::QuasiNewtonOptimizerv4`).

- **LBFGS:** Limited memory Broyden, Fletcher, Goldfarb and Shannon minimization. It is an adaptor to an optimizer in vnl (`itk::LBFGSOptimizerv4`).
- **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 [10, 76] (`itk::LBFGSBOptimizerv4`).
- **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::OnePlusOneEvolutionaryOptimizerv4`). Details on this optimizer can be found in [60].
- **Regular Step Gradient Descent:** Advances parameters in the direction of the gradient where a bipartition scheme is used to compute the step size (`itk::RegularStepGradientDescentOptimizerv4`). This optimizer is also used for Vensor transforms parameters, where the current rotation is composed with the gradient rotation to produce the new rotation vessor. The translational part of the transform parameters are updated as usually done in a vector space. It follows the definition of vessor gradients defined by Hamilton [25]
- **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 [50]. (`itk::PowellOptimizerv4`).
- **Exhaustive Optimizer:** Fully samples a grid on the parameteric space. This optimizer is equivalent to an exahausive search in a discrete grid defined over the parametric space. The grid is centered on the initial position. The subdivisions of the grid along each one of the dimensions of the parametric space is defined by an array of number of steps (`itk::ExhaustiveOptimizerv4`).

Figure 3.45 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`.

3.12.1 Registration using the One plus One Evolutionary Optimizer

The source code for this section can be found in the file `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 to the one illustrated in `ImageRegistration4`. Therefore we focus here on the setup that is specifically required for the evolutionary optimizer.

```
#include "itkImageRegistrationMethodv4.h"
#include "itkTranslationTransform.h"
#include "itkMattesMutualInformationImageToImageMetricv4.h"
#include "itkOnePlusOneEvolutionaryOptimizerv4.h"
#include "itkNormalVariateGenerator.h"
```

In this example the image types and all registration components, except the metric, are declared as in Section 3.2. The Mattes mutual information metric type is instantiated using the image types.

```
typedef itk::MattesMutualInformationImageToImageMetricv4<
    FixedImageType,
    MovingImageType > MetricType;
```

The histogram bins metric parameter is set as follows.

```
metric->SetNumberOfHistogramBins( 20 );
```

As our previous discussion in section 3.5.1, only a subsample of the virtual domain is needed to evaluate the metric. The number of spatial samples to be used depends on the content of the image, and the user can define the sampling percentage and the way that sampling operation is managed by the registration framework as follows. Sampling startegy can can be defined as REGULAR or RANDOM, while the default value is NONE.

```
registration->SetMetricSamplingPercentage( samplingPercentage );

RegistrationType::MetricSamplingStrategyType samplingStrategy =
    RegistrationType::RANDOM;
registration->SetMetricSamplingStrategy( samplingStrategy );
```

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);
```

Now we set the optimizer parameters.

```
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.

3.12.2 Registration using masks constructed with Spatial objects

The source code for this section can be found in the file `ImageRegistration12.cxx`.

This example illustrates the use of `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 EXIT_FAILURE;  
}
```

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 3.14.

The registration converges after 23 iterations and produces the following results:

```
Angle (radians) 0.174407  
Angle (degrees) 9.99281  
Center X      = 111.172  
Center Y      = 131.563  
Translation X = 12.4584  
Translation Y = 16.0726
```

These values are a very close match to the true misalignments introduced in the moving image.

Now we resample the moving image using the transform resulting from the registration process.

```
TransformType::MatrixType matrix = transform->GetMatrix();
TransformType::OffsetType offset = transform->GetOffset();

std::cout << "Matrix = " << std::endl << matrix << std::endl;
std::cout << "Offset = " << std::endl << offset << std::endl;
```

3.12.3 Rigid registrations incorporating prior knowledge

The source code for this section can be found in the file `ImageRegistration13.cxx`.

This example illustrates how to do registration with a 2D Rigid Transform and with MutualInformation metric.

```
#include "itkMatteMutualInformationImageToImageMetricv4.h"
```

The CenteredRigid2DTransform applies a rigid transform in 2D space.

```
typedef itk::CenteredRigid2DTransform< double > TransformType;
typedef itk::MatteMutualInformationImageToImageMetricv4<
    FixedImageType,
    MovingImageType > MetricType;

metric->SetNumberOfHistogramBins( 20 );

double samplingPercentage = 0.20;
registration->SetMetricSamplingPercentage( samplingPercentage );

RegistrationType::MetricSamplingStrategyType samplingStrategy =
    RegistrationType::RANDOM;
registration->SetMetricSamplingStrategy( samplingStrategy );
```

The `itk::CenteredRigid2DTransform` is initialized with 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, the first of which 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. Therefore, 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->SetLearningRate( 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
- 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 3.14. The example yielded the following results.

```

Angle (radians) 0.174585
Angle (degrees) 10.003
Center X        = 110
Center Y        = 128
Translation X   = 13.09

```

Translation Y = 15.91

These values match the true misalignment introduced in the moving image.

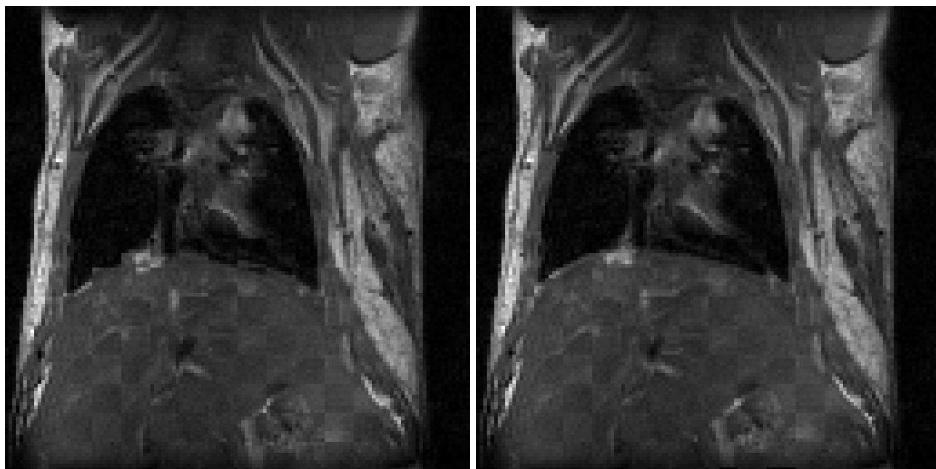


Figure 3.46: Checkerboard comparisons before and after FEM-based deformable registration.

3.13 Deformable Registration

3.13.1 FEM-Based Image Registration

The source code for this section can be found in the file `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 "itkFEMRegistrationFilter.h"
```

Next, we use `typedefs` 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> DiskImageType;
typedef itk::Image<float, 2> ImageType;
typedef itk::fem::Element2DC0LinearQuadrilateralMembrane ElementType;
typedef itk::fem::Element2DC0LinearTriangularMembrane ElementType2;
typedef itk::fem::FEMObjectType<2> FEMObjectType;
```

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;
typedef itk::fem::FEMObject<3> FEMObject3DType;
```

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, FEMObjectType>
RegistrationType;
```

In order to begin the registration, we declare an instance of the `FEMRegistrationFilter` and set its parameters. For simplicity, we will call it `registrationFilter`.

```
RegistrationType::Pointer registrationFilter = RegistrationType::New();
registrationFilter->SetMaxLevel(1);
registrationFilter->SetUseNormalizedGradient( true );
registrationFilter->ChooseMetric( 0 );

unsigned int maxiters = 20;
float E = 100;
float p = 1;
registrationFilter->SetElasticity(E, 0);
registrationFilter->SetRho(p, 0);
registrationFilter->SetGamma(1., 0);
registrationFilter->SetAlpha(1.);
registrationFilter->SetMaximumIterations( maxiters, 0 );
registrationFilter->SetMeshPixelsPerElementAtEachResolution(4, 0);
registrationFilter->SetWidthOfMetricRegion(1, 0);
registrationFilter->SetNumberOfIntegrationPoints(2, 0);
registrationFilter->SetDoLineSearchOnImageEnergy( 0 );
registrationFilter->SetTimeStep(1.);
registrationFilter->SetEmployRegridding(false);
registrationFilter->SetUseLandmarks(false);
```

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 earlier. 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->SetGlobalNumber(0);
// Young's modulus of the membrane
m->SetYoungsModulus(registrationFilter->GetElasticity());
m->SetCrossSectionalArea(1.0); // Cross-sectional area
m->SetThickness(1.0); // Thickness
m->SetMomentOfInertia(1.0); // Moment of inertia
m->SetPoissonsRatio(0.); // Poisson's ratio -- DONT CHOOSE 1.0!!
m->SetDensityHeatProduct(1.0); // Density-Heat capacity product

// Create the element type
ElementType::Pointer el=ElementType::New();
el->SetMaterial(m.GetPointer());
registrationFilter->SetElement(el.GetPointer());
registrationFilter->SetMaterial(m);
```

Now we are ready to run the registration:

```
registrationFilter->RunRegistration();
```

To output the image resulting from the registration, we can call `GetWarpedImage()`. The image is written in floating point format.

```
itk::ImageFileWriter<ImageType>::Pointer warpedImageWriter;
warpedImageWriter = itk::ImageFileWriter<ImageType>::New();
warpedImageWriter->SetInput( registrationFilter->GetWarpedImage() );
warpedImageWriter->SetFileName("warpedMovingImage.mha");
try
{
    warpedImageWriter->Update();
}
catch( itk::ExceptionObject & excp )
{
    std::cerr << excp << std::endl;
    return EXIT_FAILURE;
}
```

We can also output the displacement field resulting from the registration; we can call `GetDisplacementField()` to get the multi-component image.

```

typedef itk::ImageFileWriter<RegistrationType::FieldType> DispWriterType;
DispWriterType::Pointer dispWriter = DispWriterType::New();
dispWriter->SetInput( registrationFilter->GetDisplacementField() );
dispWriter->SetFileName("displacement.mha");
try
{
    dispWriter->Update();
}
catch( itk::ExceptionObject & excp )
{
    std::cerr << excp << std::endl;
return EXIT_FAILURE;
}

```

Figure 3.46 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.

```

% 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/Examples/RegistrationITKv4/DeformableRegistration1.cxx
% ITK/Examples/Data/RatLungSlice*
% ITK_Build_Dir/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
```

3.13.2 BSplines Image Registration

The source code for this section can be found in the file
DeformableRegistration4.cxx.

This example illustrates the use of the `itk::BSplineTransform` class for performing registration

of two 2D images in an ITKv4 registration framework. Due to the large number of parameters of the BSpline transform, we will use a `itk::LBFGSOptimizerv4` instead of a simple steepest descent or a conjugate gradient descent optimizer.

The following are the most relevant headers to this example.

```
#include "itkB_SplineTransform.h"
#include "itkLBFGSOptimizerv4.h"
```

The parameter space of the `BSplineTransform` is composed by the set of all the deformations associated with the nodes of the BSpline grid. This large number of parameters makes it possible to represent a wide variety of deformations, at the cost of requiring a significant amount of computation time.

We instantiate now the type of the `BSplineTransform` 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::BSplineTransform<
    CoordinateRepType,
    SpaceDimension,
    SplineOrder >           TransformType;
```

The transform object is constructed below.

```
TransformType::Pointer transform = TransformType::New();
```

Fixed parameters of the BSpline transform should be defined before the registration. These parameters define origin, dimension, direction and mesh size of the transform grid and are set based on specifications of the fixed image space lattice. We can use `itk::BSplineTransformInitializer` to initialize fixed parameters of a BSpline transform.

```
typedef itk::BSplineTransformInitializer<
    TransformType,
    FixedImageType> InitializerType;

InitializerType::Pointer transformInitializer = InitializerType::New();

unsigned int numberOfGridNodesInOneDimension = 8;

TransformType::MeshSizeType meshSize;
meshSize.Fill( numberOfGridNodesInOneDimension - SplineOrder );

transformInitializer->SetTransform( transform );
transformInitializer->SetImage( fixedImage );
transformInitializer->SetTransformDomainMeshSize( meshSize );
transformInitializer->InitializeTransform();
```

After setting the fixed parameters of the transform, we set the initial transform to be an identity transform. It is like setting all the transform parameters to zero in created parameter space.

```
transform->SetIdentity();
```

Then, the initialized transform is connected to the registration object and is set to be optimized directly during the registration process.

Calling `InPlaceOn()` means that the current initialized transform will be optimized directly and is grafted to the output, so it can be considered as the output transform object. Otherwise, the initial transform will be copied or “cloned” to the output transform object, and the copied object will be optimized during the registration process.

```
registration->SetInitialTransform( transform );
registration->InPlaceOn();
```

The `itk::RegistrationParameterScalesFromPhysicalShift` class is used to estimate the parameters scales before we set the optimizer.

```
typedef itk::RegistrationParameterScalesFromPhysicalShift<MetricType>
    ScalesEstimatorType;
ScalesEstimatorType::Pointer scalesEstimator = ScalesEstimatorType::New();
scalesEstimator->SetMetric( metric );
scalesEstimator->SetTransformForward( true );
scalesEstimator->SetSmallParameterVariation( 1.0 );
```

Now the scale estimator is passed to the `itk::LBFGSOptimizerv4`, and we set other parameters of the optimizer as well.

```
optimizer->SetGradientConvergenceTolerance( 5e-2 );
optimizer->SetLineSearchAccuracy( 1.2 );
optimizer->SetDefaultStepLength( 1.5 );
optimizer->TraceOn();
optimizer->SetMaximumNumberOfFunctionEvaluations( 1000 );
optimizer->SetScalesEstimator( scalesEstimator );
```

Let's execute this example using the rat lung images from the previous examples.

- RatLungSlice1.mha
- RatLungSlice2.mha

The `transform` object is updated during the registration process and is passed to the resampler to map the moving image space onto the fixed image space.

```
OptimizerType::ParametersType finalParameters = transform->GetParameters();
```

3.13.3 Level Set Motion for Deformable Registration

The source code for this section can be found in the file `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"
```

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 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 > DisplacementFieldType;
typedef itk::LevelSetMotionRegistrationFilter<
    InternalImageType,
    InternalImageType,
    DisplacementFieldType> 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,
    DisplacementFieldType >    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() );
warper->SetOutputDirection( fixedImage->GetDirection() );
```

Unlike the `ResampleImageFilter`, the `WarpImageFilter` warps or transforms 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->SetDisplacementField( 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 3.47. 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< DisplacementFieldType > 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.

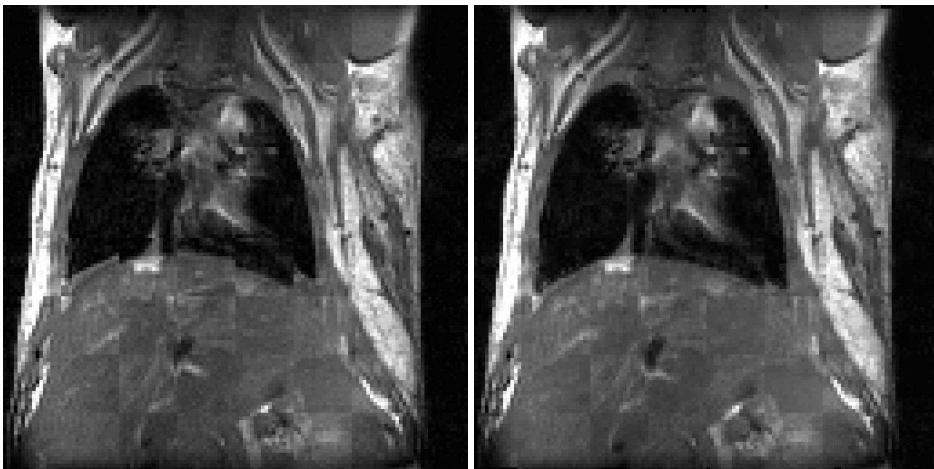


Figure 3.47: Checkerboard comparisons before and after demons-based deformable registration.

3.13.4 BSplines Multi-Grid Image Registration

The source code for this section can be found in the file `DeformableRegistration6.cxx`.

This example illustrates the use of the `itk::BSplineTransform` class in a multi-resolution scheme. Here we run 3 levels of resolutions. The first level of registration is performed with the spline grid of low resolution. Then, a common practice is to increase the resolution of the B-spline mesh (or, analogously, the control point grid size) at each level.

For this purpose, we introduce the concept of transform adaptors. Each level of each stage is defined by a transform adaptor which describes how to adapt the transform to the current level by increasing the resolution from the previous level. Here, we used `itk::BSplineTransformParametersAdaptor` class to adapt the BSpline transform parameters at each resolution level. Note that for many transforms, such as affine, the concept of an adaptor may be nonsensical since the number of transform parameters does not change between resolution levels.

Since this example is quite similar to the previous example on the use of the `BSplineTransform` we omit 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, optimizer and adaptor.

```
#include "itkBSplineTransform.h"
#include "itkLBFGSOptimizerv4.h"
#include "itkBsplineTransformParametersAdaptor.h"
```

We instantiate the type of the BSplineTransform 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::BSplineTransform<
    CoordinateRepType,
    SpaceDimension,
    SplineOrder >      TransformType;
```

We construct the transform object, initialize its parameters and connect that to the registration object.

```
TransformType::Pointer outputBSplineTransform = TransformType::New();

// Initialize the fixed parameters of transform (grid size, etc).
//
typedef itk::BSplineTransformInitializer<
    TransformType,
    FixedImageType> InitializerType;

InitializerType::Pointer transformInitializer = InitializerType::New();

unsigned int numberGridNodesInOneDimension = 8;

TransformType::MeshSizeType meshSize;
meshSize.Fill( numberGridNodesInOneDimension - SplineOrder );

transformInitializer->SetTransform( outputBSplineTransform );
transformInitializer->SetImage( fixedImage );
transformInitializer->SetTransformDomainMeshSize( meshSize );
transformInitializer->InitializeTransform();

// Set transform to identity
//
typedef TransformType::ParametersType ParametersType;
const unsigned int numberParameters =
    outputBSplineTransform->GetNumberOfParameters();
ParametersType parameters( numberParameters );
parameters.Fill( 0.0 );
outputBSplineTransform->SetParameters( parameters );

registration->SetInitialTransform( outputBSplineTransform );
registration->InPlaceOn();
```

The registration process is run in three levels. The shrink factors and smoothing sigmas are set for each level.

```
const unsigned int numberOfLevels = 3;

RegistrationType::ShrinkFactorsArrayType shrinkFactorsPerLevel;
shrinkFactorsPerLevel.SetSize( numberOfLevels );
shrinkFactorsPerLevel[0] = 3;
shrinkFactorsPerLevel[1] = 2;
shrinkFactorsPerLevel[2] = 1;

RegistrationType::SmoothingSigmasArrayType smoothingSigmasPerLevel;
smoothingSigmasPerLevel.SetSize( numberOfLevels );
smoothingSigmasPerLevel[0] = 2;
smoothingSigmasPerLevel[1] = 1;
smoothingSigmasPerLevel[2] = 0;

registration->SetNumberOfLevels( numberOfLevels );
registration->SetSmoothingSigmasPerLevel( smoothingSigmasPerLevel );
registration->SetShrinkFactorsPerLevel( shrinkFactorsPerLevel );
```

Create the transform adaptors to modify the flexibility of the deformable transform for each level of this multi-resolution scheme.

```

RegistrationType::TransformParametersAdaptorsContainerType adaptors;

// First, get fixed image physical dimensions
TransformType::PhysicalDimensionsType fixedPhysicalDimensions;
for( unsigned int i=0; i< SpaceDimension; i++ )
{
    fixedPhysicalDimensions[i] = fixedImage->GetSpacing()[i] *
static_cast<double>(
    fixedImage->GetLargestPossibleRegion().GetSize()[i] - 1 );
}

// Create the transform adaptors specific to B-splines
for( unsigned int level = 0; level < numberOfLevels; level++ )
{
typedef itk::ShrinkImageFilter<
    FixedImageType,
    FixedImageType> ShrinkFilterType;
ShrinkFilterType::Pointer shrinkFilter = ShrinkFilterType::New();
shrinkFilter->SetShrinkFactors( shrinkFactorsPerLevel[level] );
shrinkFilter->SetInput( fixedImage );
shrinkFilter->Update();

// A good heuristic is to double the b-spline mesh resolution at each level
//
TransformType::MeshSizeType requiredMeshSize;
for( unsigned int d = 0; d < ImageDimension; d++ )
{
    requiredMeshSize[d] = meshSize[d] << level;
}

typedef itk::BSplineTransformParametersAdaptor<TransformType>
    BSplineAdaptorType;
BSplineAdaptorType::Pointer bsplineAdaptor = BSplineAdaptorType::New();
bsplineAdaptor->SetTransform( outputBSplineTransform );
bsplineAdaptor->SetRequiredTransformDomainMeshSize( requiredMeshSize );
bsplineAdaptor->SetRequiredTransformDomainOrigin(
    shrinkFilter->GetOutput()->GetOrigin() );
bsplineAdaptor->SetRequiredTransformDomainDirection(
    shrinkFilter->GetOutput()->GetDirection() );
bsplineAdaptor->SetRequiredTransformDomainPhysicalDimensions(
    fixedPhysicalDimensions );

adaptors.push_back( bsplineAdaptor.GetPointer() );
}

registration->SetTransformParametersAdaptorsPerLevel( adaptors );

```

3.13.5 BSplines Multi-Grid Image Registration in 3D

The source code for this section can be found in the file
DeformableRegistration7.cxx.

This example illustrates the use of the `itk::BSplineTransform` class for performing registration of two 3D images. The example code is for the most part identical to the code presented in Section 3.13.4. The major difference is that in this example we set the image dimension to 3 and replace the `itk::LBFGSOptimizerv4` optimizer with the `itk::LBFGSBOptimizerv4`. We made the modification because we found that LBFGS does not behave well when the starting position is at or close to optimal; instead we used LBFGSB in unconstrained mode.

The following are the most relevant headers to this example.

```
#include "itkBBSplineTransform.h"
#include "itkLBFGSBOptimizerv4.h"
```

The parameter space of the `BSplineTransform` is composed by the set of all the deformations associated with the nodes of the BSpline grid. This large number of parameters enables it to represent a wide variety of deformations, at the cost of requiring a significant amount of computation time.

We instantiate now the type of the `BSplineTransform` 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::BSplineTransform<
    CoordinateRepType,
    SpaceDimension,
    SplineOrder >      TransformType;
```

The transform object is constructed, initialized like previous examples and passed to the registration method.

```
TransformType::Pointer outputBSplineTransform = TransformType::New();
registration->SetInitialTransform( outputBSplineTransform );
registration->InPlaceOn();
```

Next we set the parameters of the LBFGSB Optimizer. Note that this optimizer does not support scales estimator and sets all the parameters scales to one. Also, we should set the boundary condition for each variable, where `boundSelect[i]` can be set as: UNBOUNDED, LOWERBOUNDED, BOTHBOUNDED, UPPERBOUNDED.

```

const unsigned int numParameters =
    outputBSplineTransform->GetNumberOfParameters();
OptimizerType::BoundSelectionType boundSelect( numParameters );
OptimizerType::BoundValueType upperBound( numParameters );
OptimizerType::BoundValueType lowerBound( numParameters );

boundSelect.Fill( OptimizerType::UNBOUNDED );
upperBound.Fill( 0.0 );
lowerBound.Fill( 0.0 );

optimizer->SetBoundSelection( boundSelect );
optimizer->SetUpperBound( upperBound );
optimizer->SetLowerBound( lowerBound );

optimizer->SetCostFunctionConvergenceFactor( 1e+12 );
optimizer->SetGradientConvergenceTolerance( 1.0e-35 );
optimizer->SetNumberOfIterations( 500 );
optimizer->SetMaximumNumberOfFunctionEvaluations( 500 );
optimizer->SetMaximumNumberOfCorrections( 5 );

```

3.13.6 Image Warping with Kernel Splines

The source code for this section can be found in the file `LandmarkWarping2.cxx`.

This example illustrates how to deform an image using a KernelBase spline and two sets of landmarks.

In addition to standard headers included in previous examples, this example requires the following includes:

```

#include "itkVector.h"
#include "itkLandmarkDisplacementFieldSource.h"
#include <iostream>

```

After reading in the fixed and moving images, the deformer object is instantiated from the `itk::LandmarkDisplacementFieldSource` class, and parameters of the image space and orientation are set.

```

typedef itk::LandmarkDisplacementFieldSource<
    DisplacementFieldType
    > DisplacementSourceType;

DisplacementSourceType::Pointer deformer = DisplacementSourceType::New();

deformer->SetOutputSpacing( fixedImage->GetSpacing() );
deformer->SetOutputOrigin( fixedImage->GetOrigin() );
deformer->SetOutputRegion( fixedImage->GetLargestPossibleRegion() );
deformer->SetOutputDirection( fixedImage->GetDirection() );

```

Source and target landmarks are then created, and the points themselves are read in from a file

stream.

```
typedef DisplacementSourceType::LandmarkContainer LandmarkContainerType;
typedef DisplacementSourceType::LandmarkPointType LandmarkPointType;

LandmarkContainerType::Pointer sourceLandmarks =
    LandmarkContainerType::New();
LandmarkContainerType::Pointer targetLandmarks =
    LandmarkContainerType::New();

LandmarkPointType sourcePoint;
LandmarkPointType targetPoint;

std::ifstream pointsFile;
pointsFile.open( argv[1] );

unsigned int pointId = 0;

pointsFile >> sourcePoint;
pointsFile >> targetPoint;

while( !pointsFile.fail() )
{
    sourceLandmarks->InsertElement( pointId, sourcePoint );
    targetLandmarks->InsertElement( pointId, targetPoint );
    ++pointId;

    pointsFile >> sourcePoint;
    pointsFile >> targetPoint;

}

pointsFile.close();
```

The source and target landmark objects are then assigned to deformer.

```
deformer->SetSourceLandmarks( sourceLandmarks.GetPointer() );
deformer->SetTargetLandmarks( targetLandmarks.GetPointer() );
```

After calling `UpdateLargestPossibleRegion()` on the deformer, the displacement field may be obtained via the `GetOutput()` method.

3.13.7 Image Warping with BSplines

The source code for this section can be found in the file `BSplineWarping1.cxx`.

This example illustrates how to deform a 2D image using a BSplineTransform.

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

#include "itkResampleImageFilter.h"

#include "itkBBSplineTransform.h"
#include "itkTransformFileWriter.h"
```

First, we define the necessary types for the fixed and moving images and image readers.

```
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;
```

Use the values from the fixed image to set the corresponding values in the resampler.

```
FixedImageType::SpacingType fixedSpacing = fixedImage->GetSpacing();
FixedImageType::PointType fixedOrigin = fixedImage->GetOrigin();
FixedImageType::DirectionType fixedDirection = fixedImage->GetDirection();

resampler->SetOutputSpacing( fixedSpacing );
resampler->SetOutputOrigin( fixedOrigin );
resampler->SetOutputDirection( fixedDirection );

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 BSplineTransform 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::BSplineTransform<
    CoordinateRepType,
    SpaceDimension,
    SplineOrder >      TransformType;

TransformType::Pointer bsplineTransform = TransformType::New();

```

Next, fill the parameters of the B-spline transform using values from the fixed image and mesh.

```

const unsigned int numberGridNodes = 7;

TransformType::PhysicalDimensionsType   fixedPhysicalDimensions;
TransformType::MeshSizeType            meshSize;

for( unsigned int i=0; i< SpaceDimension; i++ )
{
    fixedPhysicalDimensions[i] = fixedSpacing[i] * static_cast<double>(
        fixedSize[i] - 1 );
}
meshSize.Fill( numberGridNodes - SplineOrder );

bsplineTransform->SetTransformDomainOrigin( fixedOrigin );
bsplineTransform->SetTransformDomainPhysicalDimensions(
    fixedPhysicalDimensions );
bsplineTransform->SetTransformDomainMeshSize( meshSize );
bsplineTransform->SetTransformDomainDirection( fixedDirection );

typedef TransformType::ParametersType      ParametersType;
const unsigned int numberParameters =
    bsplineTransform->GetNumberOfParameters();

const unsigned int numberNodes = numberParameters / SpaceDimension;

ParametersType parameters( numberParameters );

```

The B-spline grid should now be fed with coefficients 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 elements 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 convenience 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()` method.

```
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;
}
```

3.14 Demons Deformable Registration

For the problem of intra-modality deformable registration, the Insight Toolkit provides an implementation of Thirion’s “demons” algorithm [62, 63]. 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:

$$\mathbf{D}(\mathbf{X}) \cdot \nabla \mathbf{f}(\mathbf{X}) = -(\mathbf{m}(\mathbf{X}) - \mathbf{f}(\mathbf{X})) \quad (3.33)$$

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 3.33 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:

$$\mathbf{D}(\mathbf{X}) = -\frac{(\mathbf{m}(\mathbf{X}) - \mathbf{f}(\mathbf{X})) \nabla \mathbf{f}(\mathbf{X})}{\|\nabla \mathbf{f}\|^2} \quad (3.34)$$

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:

$$\mathbf{D}(\mathbf{X}) = -\frac{(\mathbf{m}(\mathbf{X}) - \mathbf{f}(\mathbf{X})) \nabla \mathbf{f}(\mathbf{X})}{\|\nabla \mathbf{f}\|^2 + (\mathbf{m}(\mathbf{X}) - \mathbf{f}(\mathbf{X}))^2 / K} \quad (3.35)$$

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 ensures the force computation is 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 3.35 such that the field at the N -th iteration is given by:

$$\mathbf{D}^N(\mathbf{X}) = \mathbf{D}^{N-1}(\mathbf{X}) - \frac{(\mathbf{m}(\mathbf{X} + \mathbf{D}^{N-1}(\mathbf{X})) - \mathbf{f}(\mathbf{X})) \nabla \mathbf{f}(\mathbf{X})}{\|\nabla \mathbf{f}\|^2 + (\mathbf{m}(\mathbf{X} + \mathbf{D}^{N-1}(\mathbf{X})) - \mathbf{f}(\mathbf{X}))^2} \quad (3.36)$$

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.

3.14.1 Asymmetrical Demons Deformable Registration

The source code for this section can be found in the file `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"
```

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 > DisplacementFieldType;
typedef itk::DemonsRegistrationFilter<
    InternalImageType,
    InternalImageType,
    DisplacementFieldType> 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,
    DisplacementFieldType >    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() );
warper->SetOutputDirection( fixedImage->GetDirection() );
```

Unlike `ResampleImageFilter`, `WarpImageFilter` warps or transforms 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->SetDisplacementField( 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 3.48. 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< DisplacementFieldType > 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.

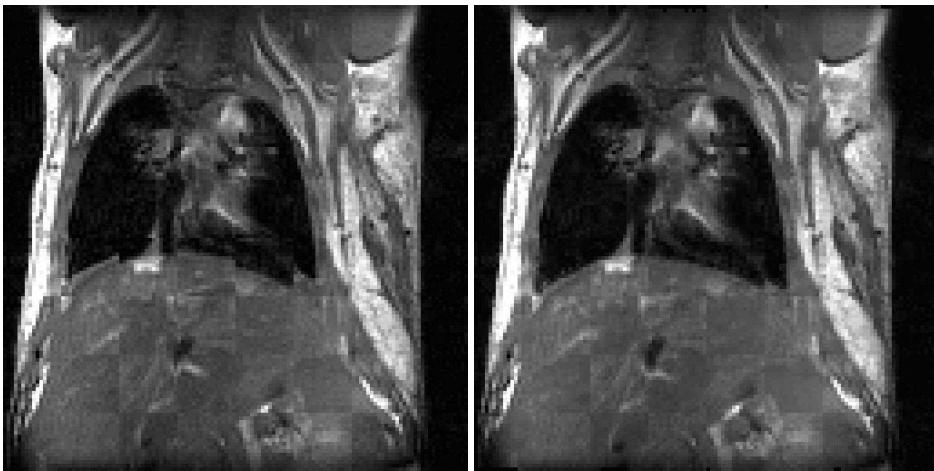


Figure 3.48: Checkerboard comparisons before and after demons-based deformable registration.

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

$$\mathbf{D}(\mathbf{X}) = -\frac{2(\mathbf{m}(\mathbf{X}) - \mathbf{f}(\mathbf{X}))(\nabla \mathbf{f}(\mathbf{X}) + \nabla \mathbf{g}(\mathbf{X}))}{\|\nabla \mathbf{f} + \nabla \mathbf{g}\|^2 + (\mathbf{m}(\mathbf{X}) - \mathbf{f}(\mathbf{X}))^2 / \mathbf{K}} \quad (3.37)$$

The following example illustrates the use of this deformable registration method.

3.14.2 Symmetrical Demons Deformable Registration

The source code for this section can be found in the file `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"
```

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 > DisplacementFieldType;
typedef itk::SymmetricForcesDemonsRegistrationFilter<
    InternalImageType,
    InternalImageType,
    DisplacementFieldType> 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,
    DisplacementFieldType > 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() );
warper->SetOutputDirection( fixedImage->GetDirection() );

```

Unlike the ResampleImageFilter, the WarpImageFilter warps or transforms 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->SetDisplacementField( 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 3.49. 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< DisplacementFieldType > 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.

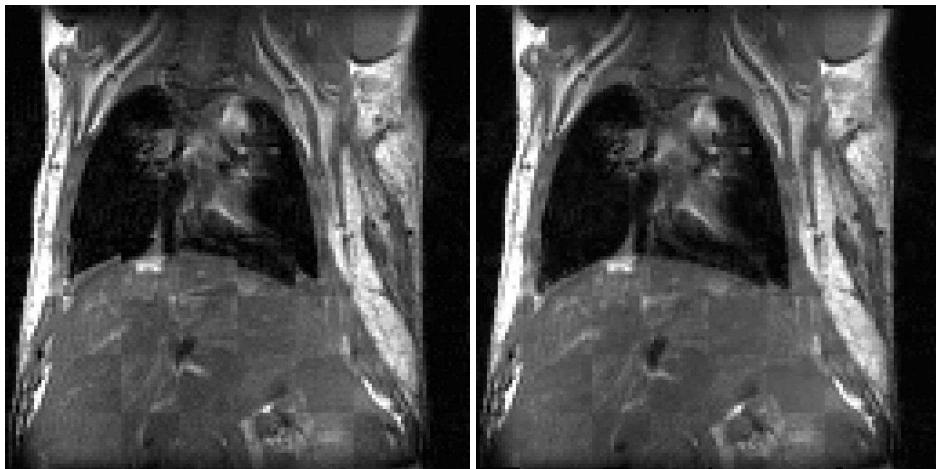


Figure 3.49: Checkerboard comparisons before and after demons-based deformable registration.

3.15 Visualizing Deformation fields

Vector deformation fields may be visualized using ParaView. ParaView [26] 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>.

3.15.1 Visualizing 2D deformation fields

Let us visualize the deformation field obtained from Demons Registration algorithm generated from ITK/Examples/RegistrationITKv4/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 3.50.

Convert 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 3.51. 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 a reasonable amount. This uniformly downsamples the number of glyphs. Alternatively, you may apply a

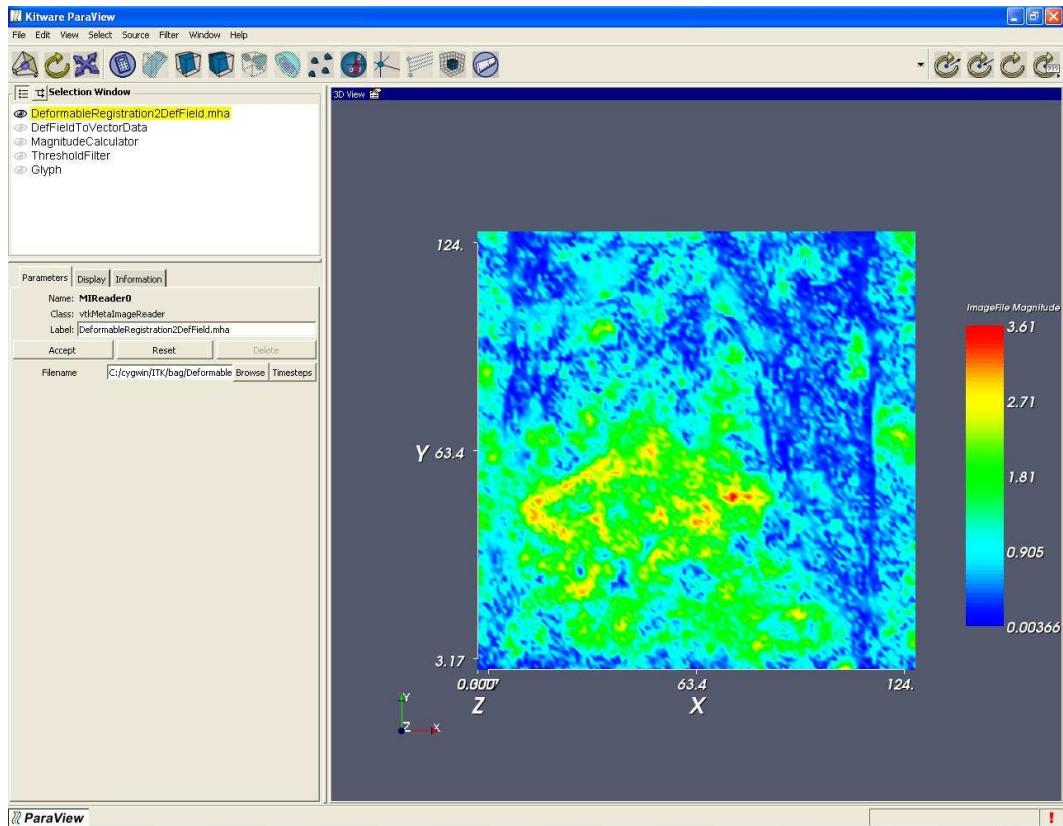


Figure 3.50: Deformation field magnitudes displayed using Paraview

Threshold filter to the *Magnitude* of the vector dataset and then *glyph* the vector data that lie 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 3.52 shows the vector field visualized using Paraview by thresholding the vector magnitudes by 2.1 and restricting the number of glyphs to 100.

3.15.2 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

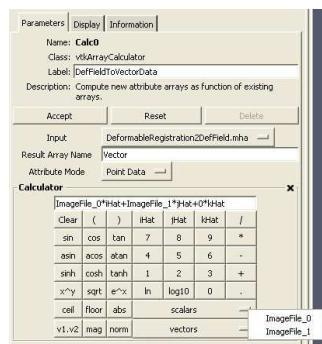


Figure 3.51: Calculators and filters may be used to compute the vector magnitude, compose vectors etc.

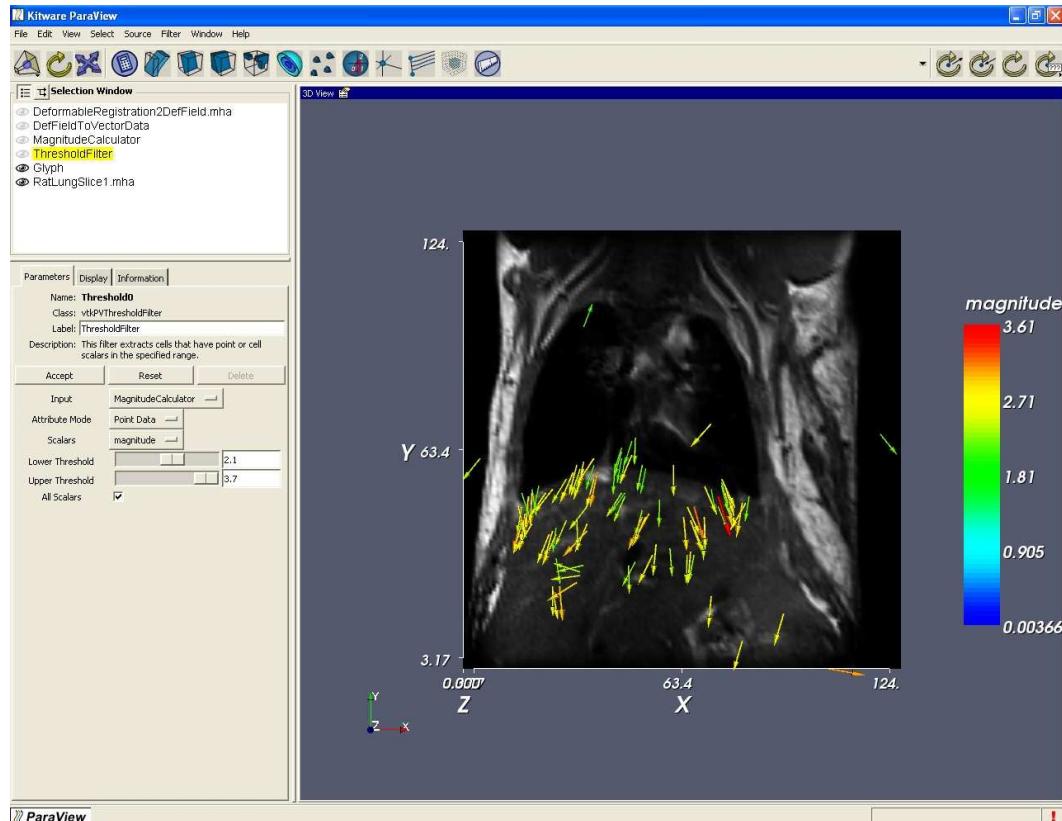


Figure 3.52: Deformation field visualized using Paraview after thresholding and subsampling.

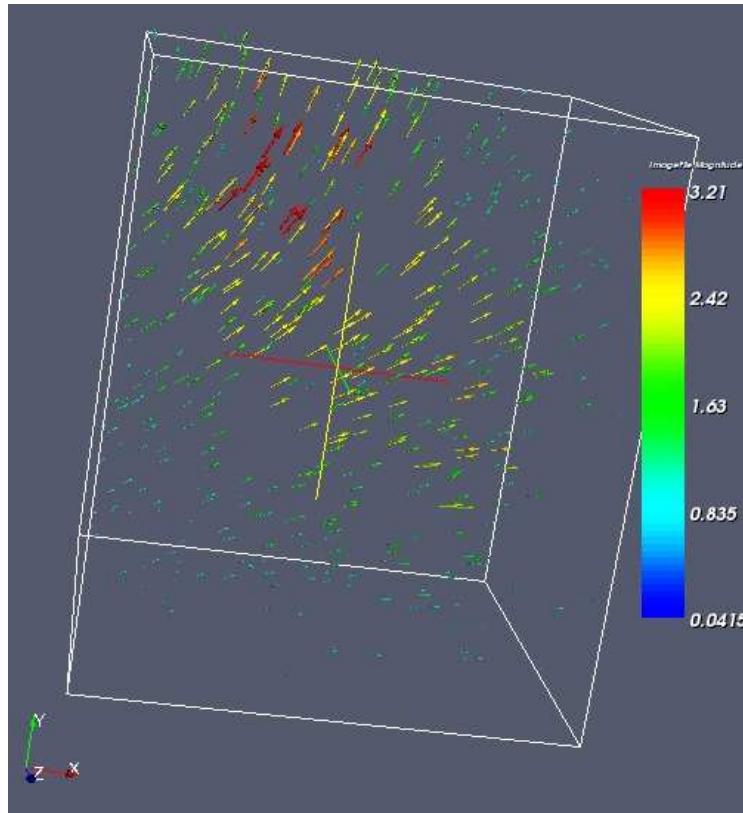


Figure 3.53: 3D Deformation field visualized using Paraview.

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

$$I[f(x,y)] = \iint (f_{xx}^2 + 2f_{xy}^2 + f_{yy}^2) dx dy \quad (3.38)$$

where the subscripts denote partial derivatives of f .

We may now proceed as before to visualize the deformation field using Paraview as shown in Figure 3.53.

Let us register the deformed volumes generated by Thin plate warping in the previous example 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. 3.17 prints the

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 3.17: LBFGS Optimizer trace.

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 3.54. A difference image of two slices before and after registration is shown in 3.55. As can be seen from the figures, the deformation field is in close agreement to the one generated from the Thin plate spline warping.

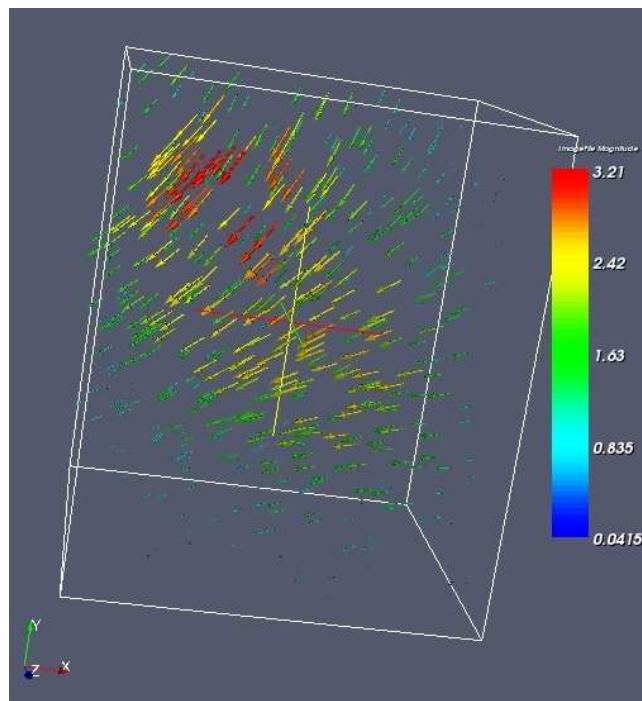


Figure 3.54: Resulting deformation field that maps the moving image to the fixed image.

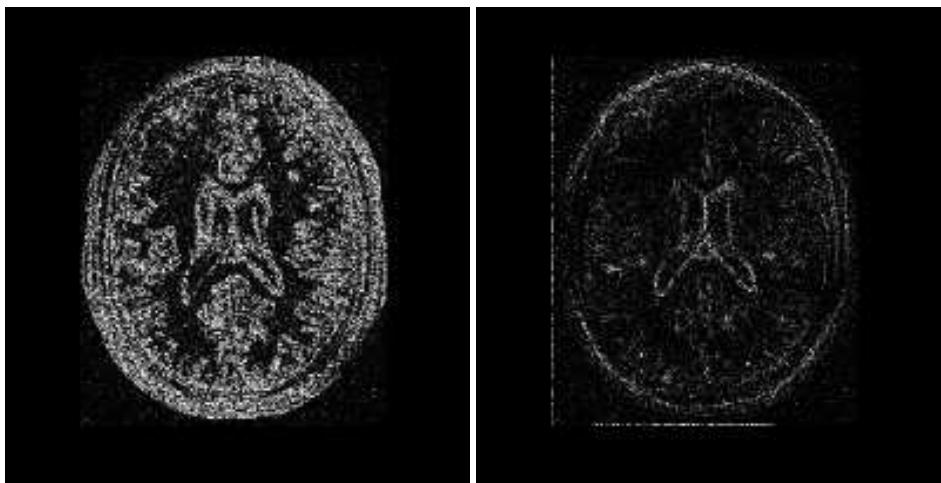


Figure 3.55: Difference image from a slice before and after registration.

3.16 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.

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 3.57. 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 3.56 illustrates the major components of the registration framework in ITK when a model-based 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.

The following sections present examples of the typical uses of these powerful elements of the toolkit.

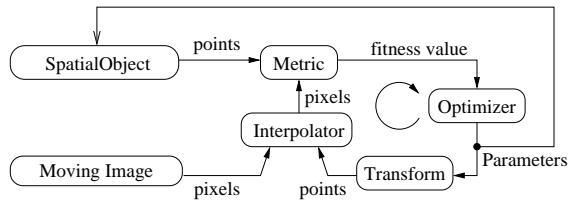


Figure 3.56: The basic components of model based registration are an image, a spatial object, a transform, a metric, an interpolator and an optimizer.

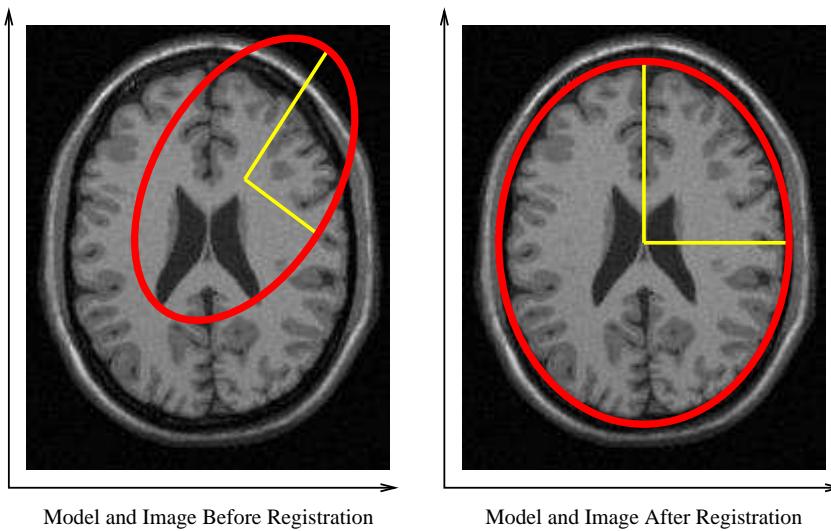


Figure 3.57: Basic concept of Model-to-Image registration. A simplified geometrical model (ellipse) is registered against an anatomical structure (skull) by applying a spatial transform and modifying the model internal parameters. This image is not the result of an actual registration, it is shown here only with the purpose of illustrating the concept of model to image registration.

The source code for this section can be found in the file
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) ITK_OVERRIDE
    {
        Execute( (const itk::Object *)caller, event );
    }

    void Execute(const itk::Object *,
                const itk::EventObject & event) ITK_OVERRIDE
    {
        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 ImageToSpatialObjectMetric 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 ITK_OVERRIDE
{
    double value;
    this->m_Transform->SetParameters( parameters );

    value = 0;
    for( PointListType::const_iterator it = m_PointList.begin();
          it != m_PointList.end(); ++it)
    {
        PointType transformedPoint = this->m_Transform->TransformPoint(*it);
        if( this->m_Interpolator->IsInsideBuffer( transformedPoint ) )
        {
            value += this->m_Interpolator->Evaluate( transformedPoint );
        }
    }
    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 2D 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();
```

Below we 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 2D 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

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 `Update()` method. We place this call in a `try/catch` block in case any exception is thrown during the process.

```

try
{
    registration->Update();
    std::cout << "Optimizer stop condition: "
        << registration->GetOptimizer()->GetStopConditionDescription()
        << std::endl;
}
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 3.58. 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.

3.17 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 [27]. 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 [52]. In both circumstances, the availability of 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.

3.17.1 Point Set Registration in 2D

The source code for this section can be found in the file `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`.

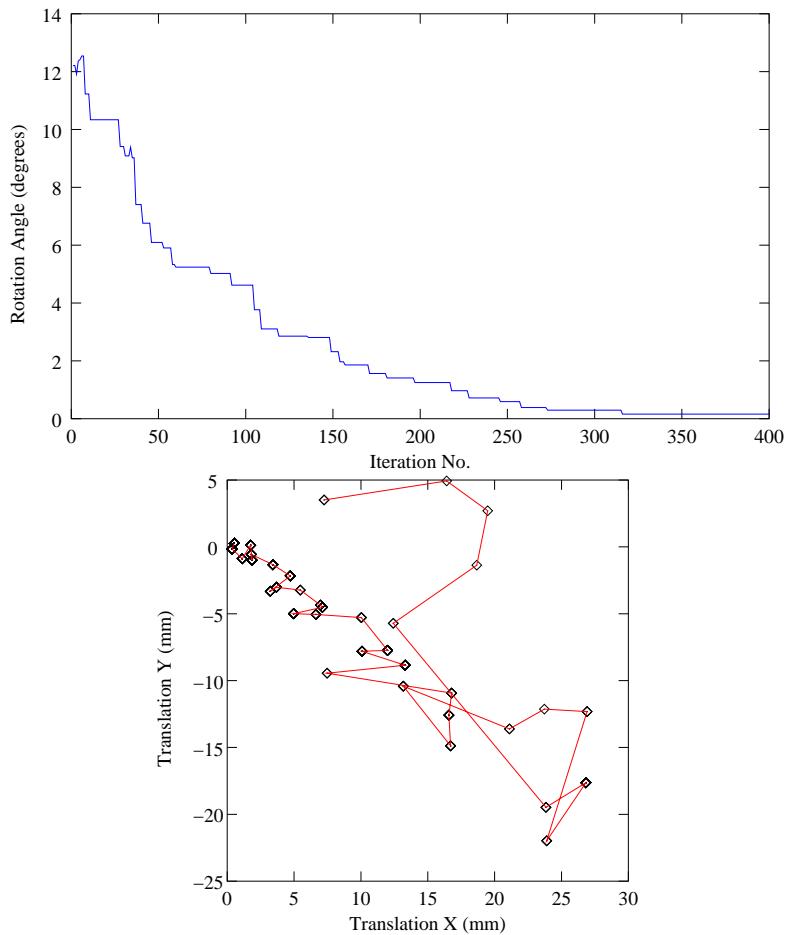


Figure 3.58: Plots of the angle and translation parameters for a registration process between a spatial object and an image.

The first step is to include the relevant headers.

```
#include "itkTranslationTransform.h"
#include "itkEuclideanDistancePointMetric.h"
#include "itkLevenbergMarquardtOptimizer.h"
#include "itkPointSetToPointSetRegistrationMethod.h"
```

Next, define the necessary types for the fixed and moving pointsets and point containers.

```
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;
```

After the points are read in from files, set up the metric type.

```
typedef itk::EuclideanDistancePointMetric<
    PointSetType,
    PointSetType>
    MetricType;

MetricType::Pointer metric = MetricType::New();
```

Now, setup the transform, optimizers, and registration method using the point set types defined earlier.

```

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 );

```

Next we setup the convergence criteria, and other properties required by the optimizer.

```

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 );

```

In this case we start from an identity transform, but in reality the user will usually be able to provide a better guess than this.

```
transform->SetIdentity();
```

Finally, connect all the components required for the registration, and an observer.

```

registration->SetMetric(      metric      );
registration->SetOptimizer(   optimizer    );
registration->SetTransform(   transform    );
registration->SetFixedPointSet( fixedPointSet );
registration->SetMovingPointSet( movingPointSet );

// Connect an observer
CommandIterationUpdate::Pointer observer = CommandIterationUpdate::New();
optimizer->AddObserver( itk::IterationEvent(), observer );

```

3.17.2 Point Set Registration in 3D

The source code for this section can be found in the file `IterativeClosestPoint2.cxx`.

This example illustrates how to perform Iterative Closest Point (ICP) registration in ITK using sets of 3D points.

The first step is to include the relevant headers.

```

#include "itkEuler3DTransform.h"
#include "itkEuclideanDistancePointMetric.h"
#include "itkLevenbergMarquardtOptimizer.h"
#include "itkPointSetToPointSetRegistrationMethod.h"
#include <iostream>
#include <fstream>

```

First, define the necessary types for the moving and fixed point sets.

```

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;

```

After the points are read in from files, setup the metric to be used later by the registration.

```

typedef itk::EuclideanDistancePointMetric<
    PointSetType,
    PointSetType>
    MetricType;

MetricType::Pointer metric = MetricType::New();

```

Next, setup the transform, optimizers, and registration.

```
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() );
```

Next, set the scales and ranges for translations and rotations in the transform. Also, set the convergence criteria and number of iterations to be used by the optimizer.

```
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 );
```

Here we start with an identity transform, although the user will usually be able to provide a better guess than this.

```
transform->SetIdentity();
```

Connect all the components required for the registration.

```
registration->SetMetric(      metric      );
registration->SetOptimizer(   optimizer    );
registration->SetTransform(   transform    );
registration->SetFixedPointSet( fixedPointSet );
registration->SetMovingPointSet( movingPointSet );
```

3.17.3 Point Set to Distance Map Metric

The source code for this section can be found in the file `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 saved 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`.

The first step is to include the relevant headers.

```
#include "itkTranslationTransform.h"
#include "itkEuclideanDistancePointMetric.h"
#include "itkLevenbergMarquardtOptimizer.h"
#include "itkPointSetToPointSetRegistrationMethod.h"
#include "itkDanielssonDistanceMapImageFilter.h"
#include "itkPointSetToImageFilter.h"
#include <iostream>
#include <fstream>
```

Next, define the necessary types for the fixed and moving point sets.

```

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;

```

Setup the metric, transform, optimizers and registration in a manner similar to the previous two examples.

In the preparation of the distance map, we first need to map the fixed points into a binary image.

```

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 );

```

Continue to prepare the distance map, in order to accelerate the distance computations.

```

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() );

```

3.18 Registration Troubleshooting

So you read the previous sections, you wrote the code, it compiles and links fine, but when you run it the registration results are not what you were expecting. In that case, this section is for you. This is a compilation of the most common problems that users face when performing image registration. It provides explanations on the potential sources of the problems, and advice on how to deal with those problems.

Most of the material in this section has been taken from frequently asked questions of the ITK users list.

3.18.1 Too many samples outside moving image buffer

<http://public.kitware.com/pipermail/insight-users/2007-March/021442.html>

This is a common error message in image registration.

It means that at the current iteration of the optimization, the two images are so off-registration that their spatial overlap is not large enough for bringing them back into registration.

The common causes of this problem are:

- Poor initialization: You must initialize the transform properly. Please familiarize yourself with the `itk::CenteredTransformInitializer` class.
- Optimizer steps too large. If your optimizer takes steps that are too large, it risks becoming unstable and sending the images too far apart. You may want to start the optimizer with a maximum step length of 1.0, and only increase it once you have managed to fine tune all other registration parameters.

Increasing the step length makes your program faster, but it also makes it more unstable.

- Poor set up of the transform parameters scaling. This is extremely critical in registration. You must make sure that you balance the relative difference of scale between the rotation parameters and the translation parameters.

In typical medical datasets such as CT and MR, translations are measured in millimeters, and therefore are in the range of -100:100, while rotations are measured in radians, and therefore they tend to be in the range of -1:1.

A rotation of 3 radians is catastrophic, while a translation of 3 millimeters is rather inoffensive. That difference in scale is the one that must be accounted for.

3.18.2 General heuristics for parameter fine-tunning

<http://public.kitware.com/pipermail/insight-users/2007-March/021435.html>

Here is some advice on how to fine tune the parameters of the registration process.

1) Set Maximum step length to 0.1 and do not change it until all other parameters are stable.

2) Set Minimum step length to 0.001 and do not change it.

You could interpret these two parameters as if their units were radians. So, 0.1 radian = 5.7 degrees.

3) Number of histogram bins:

First plot the histogram of your image using the example program in

`Insight/Examples/Statistics/ImageHistogram2.cxx`

In that program use first a large number of bins (for example 2000) and identify the different populations of intensity level and to what anatomical structures they correspond.

Once you identify the anatomical structures in the histogram, then rerun that same program with less and less number of bins, until you reach the minimum number of bins for which all the tissues that are important for your application, are still distinctly differentiated in the histogram. At that point, take that number of bins and use it for your Mutual Information metric.

4) Number of Samples: The trade-off with the number of samples is the following:

a) computation time of registration is linearly proportional to the number of samples b) the samples must be enough to significantly populate the joint histogram. c) Once the histogram is populated, there is not much use in adding more samples. Therefore do the following:

Plot the joint histogram of both images, using the number of bins that you selected in item (3). You can do this by modifying the code of the example:

`Insight/Examples/Statistics/ ImageMutualInformation1.cxx` you have to change the code to print out the values of the bins. Then use a plotting program such as gnuplot, or Matlab, or even Excel and look at the distribution. The number of samples to take must be enough for producing the same "appearance" of the joint histogram. As an arbitrary rule of thumb you may want to start using a high number of samples (80% - 100%). And do not change it until you have mastered the other parameters of the registration. Once you get your registration to converge you can revisit the number of samples and reduce it in order to make the registration run faster. You can simply reduce it until you find that the registration becomes unstable. That's your critical bound for the minimum number of samples. Take that number and multiply it by the magic number 1.5, to send it back to a stable region, or if your application is really critical, then use an even higher magic number x2.0.

This is just engineering: you figure out what is the minimal size of a piece of steel that will support a bridge, and then you enlarge it to keep it away from the critical value.

5) The MOST critical values of the registration process are the scaling parameters that define the proportions between the parameters of the transform. In your case, for an Affine Transform in 2D, you have 6 parameters. The first four are the ones of the Matrix, and the last two are the translation. The rotation matrix value must be in the ranges of radians which is typically [-1 to 1], while the translation values are in the ranges of millimeters (your image size units). You want to start by setting the scaling of the matrix parameters to 1.0, and the scaling of the Translation parameters to

the holy esoteric values:

```
1.0 / ( 10.0 * pixelspacing[0] * imagesize[0] ) 1.0 / ( 10.0 * pixelspacing[1] * imagesize[1] )
```

This is telling the optimizer that you consider that rotating the image by 57 degrees is as "significant" as translating the image by half its physical extent.

Note that esoteric value has included the arbitrary number 10.0 in the denominator, for no other reason that we have been lucky when using that factor. This of course is just a superstition, so you should feel free to experiment with different values of this number.

Just keep in mind that what the optimizer will do is to "jump" in a parametric space of 6 dimensions, and that the component of the jump on every dimension will be proportional to 1/scaling factor * OptimizerStepLength. Since you set the optimizer Step Length to 0.1, the optimizer will start by exploring the rotations at jumps of about 5 degrees, which is a conservative rotation for most medical applications.

If you have reasons to think that your rotations are larger or smaller, then you should modify the scaling factor of the matrix parameters accordingly.

In the same way, if you think that 1/10 of the image size is too large as the first step for exploring the translations, then you should modify the scaling of translation parameters accordingly.

In order to drive all these you need to analyze the feedback that the observer is providing you. For example, plot the metric values, and plot the translation coordinates so that you can get a feeling of how the registration is behaving.

Note also that image registration is not a science. It is a pure engineering practice, and therefore, there are no correct answers, nor "truths" to be found. It is all about how much quality you want, and how much computation time, and development time you are willing to pay for that quality. The "satisfying" answer for your specific application must be found by exploring the trade-offs between the different parameters that regulate the image registration process.

If you are proficient in VTK you may want to consider attaching some visualization to the Event observer, so that you can have a visual feedback on the progress of the registration. This is a lot more productive than trying to interpret the values printed out on the console by the observer.

CHAPTER
FOUR

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.

4.1 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.

4.1.1 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 `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. Lower and upper threshold values should be provided. The region-growing algorithm includes those pixels whose intensities are inside the interval.

$$I(\mathbf{X}) \in [\text{lower}, \text{upper}] \quad (4.1)$$

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 2.7.3 could be used to this end. In this particular example we use the `itk::CurvatureFlowImageFilter`, so 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() );
```

`CurvatureFlowImageFilter` requires a couple of parameters. The following are typical values for *2D* images. However, these values may have to be adjusted depending on the amount of noise present in the input image.

```
smoothing->SetNumberOfIterations( 5 );  
smoothing->SetTimeStep( 0.125 );
```

We now set the lower and upper threshold values. Any pixel whose value is between `lowerThreshold` and `upperThreshold` will be included in the region, and any pixel whose value is outside will be excluded. Setting these values too close together will be too restrictive for the region to grow; setting them too far apart will cause the region to engulf 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()`.

```
connectedThreshold->SetReplaceValue( 255 );
```

The algorithm must be initialized by setting a seed point (i.e., the `itk::Index` of the pixel from which the region will grow) using the `SetSeed()` method. It is convenient to initialize with a point in a *typical* region of the anatomical structure to be segmented.

```
connectedThreshold->SetSeed( index );
```

Structure	Seed Index	Lower	Upper	Output Image
White matter	(60, 116)	150	180	Second from left in Figure 4.1
Ventricle	(81, 112)	210	250	Third from left in Figure 4.1
Gray matter	(107, 69)	180	210	Fourth from left in Figure 4.1

Table 4.1: Parameters used for segmenting some brain structures shown in Figure 4.1 with the filter `itk::ConnectedThresholdImageFilter`.

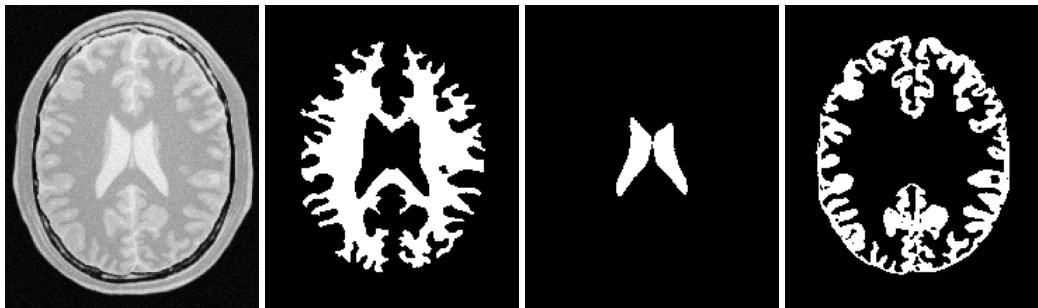


Figure 4.1: Segmentation results for the `ConnectedThreshold` filter for various seed points.

Invocation of the `Update()` method on the writer triggers 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 4.1 illustrates several examples of segmentation. The parameters used are presented in Table 4.1.

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 an operator clicks on multiple points of the object to be segmented and each selected point is passed as a seed to this filter.

4.1.2 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 `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, and to define the image dimension.

```
typedef unsigned char InputPixelType;
typedef unsigned char OutputPixelType;
const unsigned int Dimension = 2;
```

The input and output image types are now defined using their respective pixel types and dimensions.

```
typedef itk::Image< InputPixelType, Dimension > InputImageType;
typedef itk::Image< OutputPixelType, Dimension > 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 1 on page 1 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< OutputImageType > 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 );
```

Execution of the filter is triggered by invoking the `Update()` method, which we wrap in a `try/catch` block. If the filter's output has been passed as input to subsequent filters, the `Update()` call on any downstream filters in the pipeline will indirectly trigger the update of this filter.

```
try
{
    filter->Update();
}
catch( itk::ExceptionObject & excp )
{
    std::cerr << "Exception thrown " << excp << std::endl;
}
```

We can now retrieve the internally-computed threshold value with the `GetThreshold()` method and print it to the console.

```
int threshold = filter->GetThreshold();
std::cout << "Threshold = " << threshold << std::endl;
```

Figure 4.2 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 `OtsuMultipleThresholdImageFilter.cxx`.

This example illustrates how to use the `itk::OtsuMultipleThresholdsCalculator`.

```
#include "itkOtsuMultipleThresholdsCalculator.h"
```

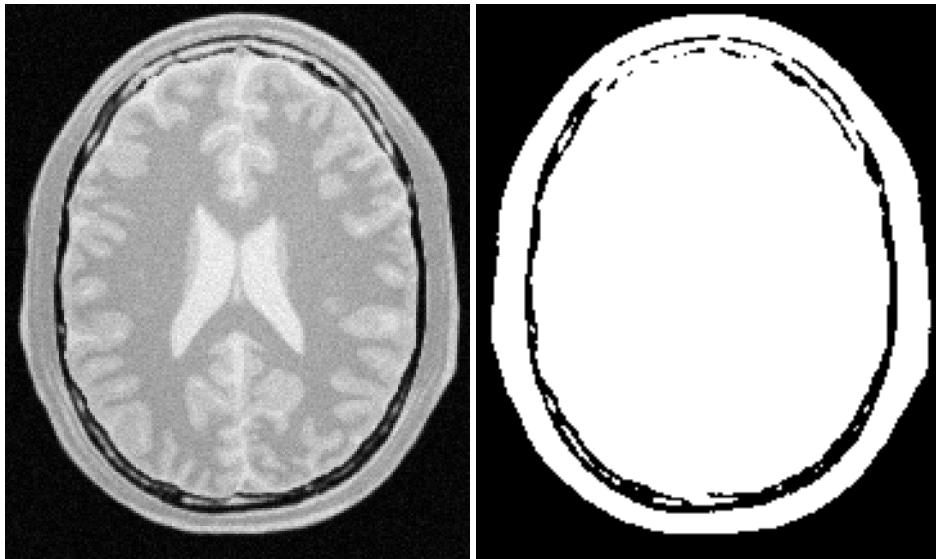


Figure 4.2: Effect of the `OtsuThresholdImageFilter` on a slice from a MRI proton density image of the brain.

`OtsuMultipleThresholdsCalculator` calculates thresholds for a given histogram so as to maximize the between-class variance. We use `ScalarImageToHistogramGenerator` to generate histograms. The histogram type defined by the generator is then used to instantiate the type of the `Otsu` threshold calculator.

```
typedef itk::Statistics::ScalarImageToHistogramGenerator<
    InputImageType > ScalarImageToHistogramGeneratorType;

typedef ScalarImageToHistogramGeneratorType::HistogramType HistogramType;

typedef itk::OtsuMultipleThresholdsCalculator< HistogramType >
    CalculatorType;
```

Once thresholds are computed we will use `BinaryThresholdImageFilter` to segment the input image.

```
typedef itk::BinaryThresholdImageFilter<
    InputImageType, OutputImageType > FilterType;
```

Create a histogram generator and calculator using the standard `New()` method.

```
ScalarImageToHistogramGeneratorType::Pointer scalarImageToHistogramGenerator
= ScalarImageToHistogramGeneratorType::New();

CalculatorType::Pointer calculator = CalculatorType::New();
FilterType::Pointer filter = FilterType::New();
```

Set the following properties for the histogram generator and the calculators, in this case grabbing the number of thresholds from the command line.

```
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() );
```

Here we obtain a const reference to the thresholds by calling the `GetOutput()` method.

```
const CalculatorType::OutputType &thresholdVector = calculator->GetOutput();
```

We now iterate through `thresholdVector`, printing each value to the console and writing an image thresholded with adjacent values from the container. (In the edge cases, the minimum and maximum values of the `InternalPixelType` are used).

```
typedef CalculatorType::OutputType::const_iterator ThresholdItType;

for( ThresholdItType itNum = thresholdVector.begin();
      itNum != thresholdVector.end();
      ++itNum )
{
    std::cout << "OtsuThreshold["
        << (int) (itNum - thresholdVector.begin())
        << "] = "
        << static_cast<itk::NumericTraits<
            CalculatorType::MeasurementType>::PrintType>(*itNum)
        << std::endl;
```

Also write out the image thresholded between the upper threshold and the max intensity.

```
upperThreshold = itk::NumericTraits<InputPixelType>::max();
filter->SetLowerThreshold( lowerThreshold );
filter->SetUpperThreshold( upperThreshold );
```

4.1.3 Neighborhood Connected

The source code for this section can be found in the file `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` considers only the value of the pixel itself when determining whether it belongs to the region: if its value is within the interval [lowerThreshold,upperThreshold] it is included, otherwise it is excluded. `NeighborhoodConnectedImageFilter`, on the other hand, considers a user-defined neighborhood surrounding the pixel, requiring that the intensity of **each** neighbor be within the interval for it to be included.

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 `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() );
```

`CurvatureFlowImageFilter` requires a couple of 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 );
```

`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` example, we must provide the intensity value to be used for the output pixels accepted in the region and at least one seed point to define the starting point.

```
neighborhoodConnected->SetSeed( index );
neighborhoodConnected->SetReplaceValue( 255 );
```

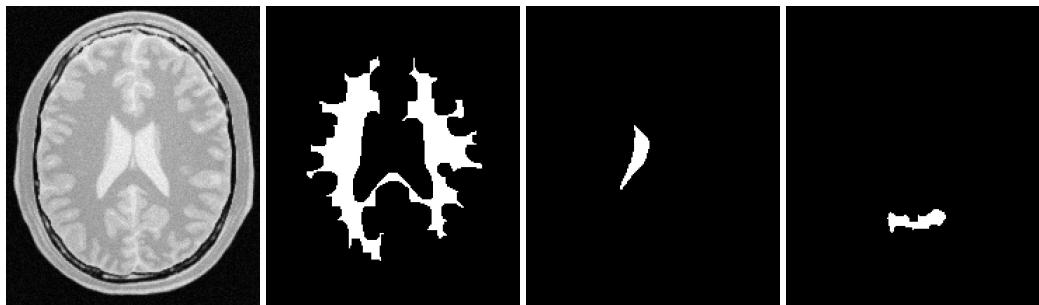


Figure 4.3: Segmentation results of the `NeighborhoodConnectedImageFilter` for various seed points.

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 4.3
Ventricle	(81, 112)	210	250	Third from left in Figure 4.3
Gray matter	(107, 69)	180	210	Fourth from left in Figure 4.3

As with the `ConnectedThresholdImageFilter` example, several seeds could be provided to the filter by repeatedly calling the `AddSeed()` method with different indices. Compare Figures 4.3 and 4.1, demonstrating the outputs of `NeighborhoodConnectedThresholdImageFilter` and `ConnectedThresholdImageFilter`, respectively. It is instructive to adjust the neighborhood radii and observe its effect on the smoothness of segmented object borders, size of the segmented region, and computing time.

4.1.4 Confidence Connected

The source code for this section can be found in the file `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,

$$I(\mathbf{X}) \in [m - f\sigma, m + f\sigma] \quad (4.2)$$

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 2.7.3 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() );
```

`CurvatureFlowImageFilter` requires 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 );
```

`ConfidenceConnectedImageFilter` also requires two parameters. First, the 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 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 which may belong to separate anatomical structures. This is not desirable behavior.

```
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 guarantee that this algorithm will converge on 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 an `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 )
{
    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

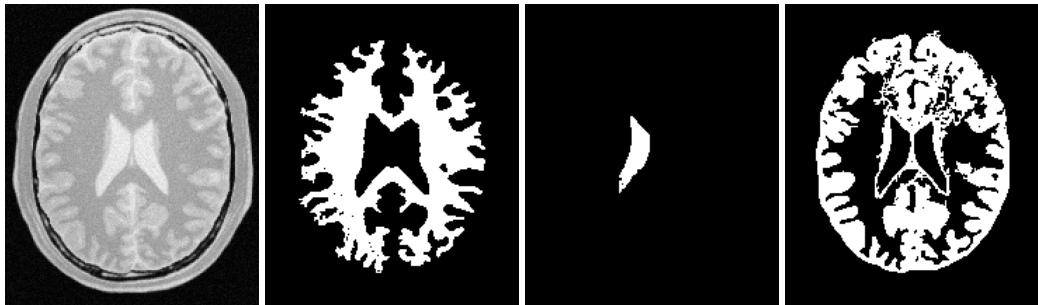


Figure 4.4: Segmentation results for the ConfidenceConnected filter for various seed points.

Structure	Seed Index	Output Image
White matter	(60, 116)	Second from left in Figure 4.4
Ventricle	(81, 112)	Third from left in Figure 4.4
Gray matter	(107, 69)	Fourth from left in Figure 4.4

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 Brain-Web 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 Anisotropic 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), (63,87,94), (63,157,90), (111,188,90), (111,50,88). The ConfidenceConnected 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 4.5 shows the rendered volume. Figure 4.6 shows an axial, saggital and a coronal slice of the volume.

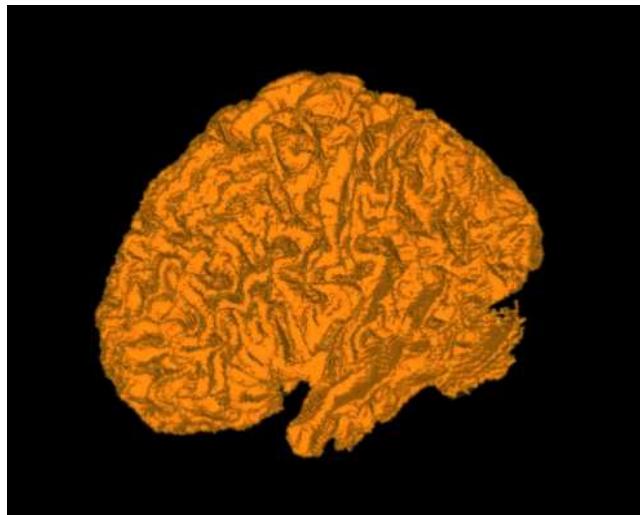


Figure 4.5: White matter segmented using Confidence Connected region growing.



Figure 4.6: Axial, sagittal and coronal slice segmented using Confidence Connected region growing.

4.1.5 Isolated Connected

The source code for this section can be found in the file `IsolatedConnectedImageFilter.cxx`.

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->AddSeed1( indexSeed1 );
isolatedConnected->AddSeed2( indexSeed2 );
```

As in the `itk::ConnectedThresholdImageFilter` we must now specify the intensity value to be

Adjacent Structures	Seed1	Seed2	Lower	Isolated value found
Gray matter vs White matter	(61, 140)	(63, 43)	150	183.31

Table 4.2: Parameters used for separating white matter from gray matter in Figure 4.7 using the `IsolatedConnectedImageFilter`.

set on the output pixels and at least one seed point to define the initial region.

```
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 using 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 4.7 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 4.2 presents the parameters used to obtain the images shown in Figure 4.7.

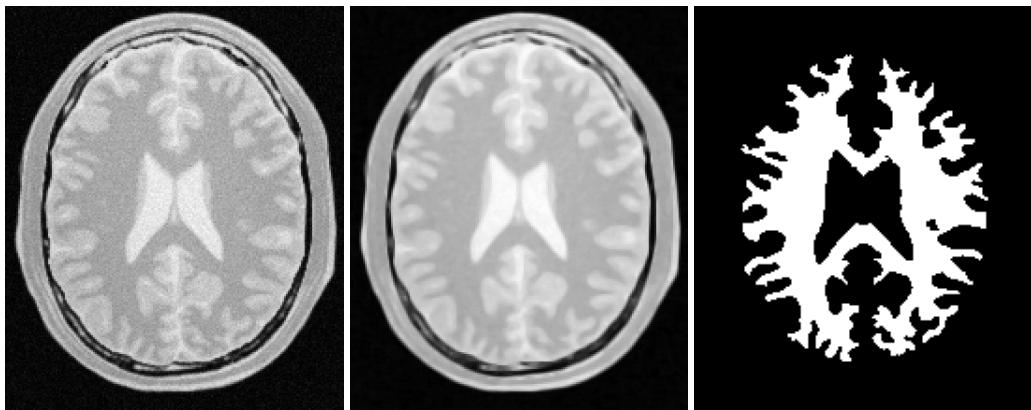


Figure 4.7: Segmentation results of the `IsolatedConnectedImageFilter`.

4.1.6 Confidence Connected in Vector Images

The source code for this section can be found in the file `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.

```
const unsigned int Dimension = 2;

typedef unsigned char PixelComponentType;
typedef itk::RGBPixel< PixelComponentType > InputPixelType;
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() );
```

`VectorConfidenceConnectedImageFilter` requires 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 which 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 an `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



Figure 4.8: Segmentation results of the `VectorConfidenceConnected` filter for various seed points.

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 4.8
Rectum	(23, 93)	7	1	Third from left in Figure 4.8
Vitreo	(66, 66)	3	1	Fourth from left in Figure 4.8

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 facilitate 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;
typedef ConnectedFilterType::CovarianceMatrixType CovarianceMatrixType;

const MeanVectorType & mean = confidenceConnected->GetMean();
const CovarianceMatrixType & covariance
    = confidenceConnected->GetCovariance();

std::cout << "Mean vector = "      << mean      << std::endl;
std::cout << "Covariance matrix = " << covariance << std::endl;

```

4.2 Segmentation Based on Watersheds

4.2.1 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 [74, 75].

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.

Gradient descent associates regions with local minima of f (clearly interior points) using the watersheds of the graph of f , as in Figure 4.9. 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” [30, 31, 20] in the graph of f . In the 1D case ($U \subset \Reals$), 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

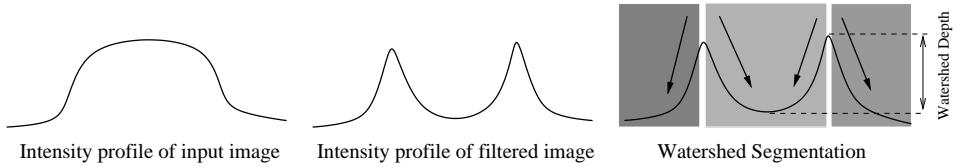


Figure 4.9: A fuzzy-valued boundary map, from an image or set of images, is segmented using local minima and catchment basins.

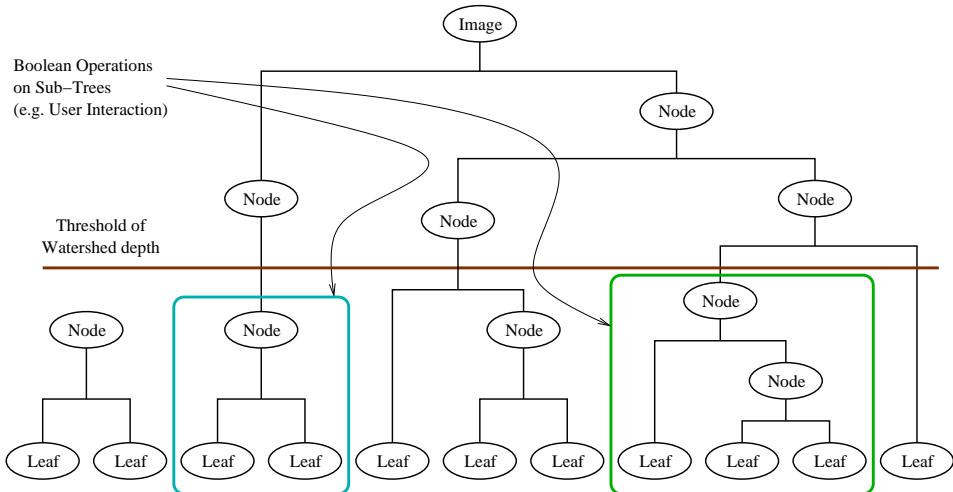


Figure 4.10: A watershed segmentation combined with a saliency measure (watershed depth) produces a hierarchy of regions. Structures can be derived from images by either thresholding the saliency measure or combining subtrees within the hierarchy.

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 4.10. 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 [75].

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

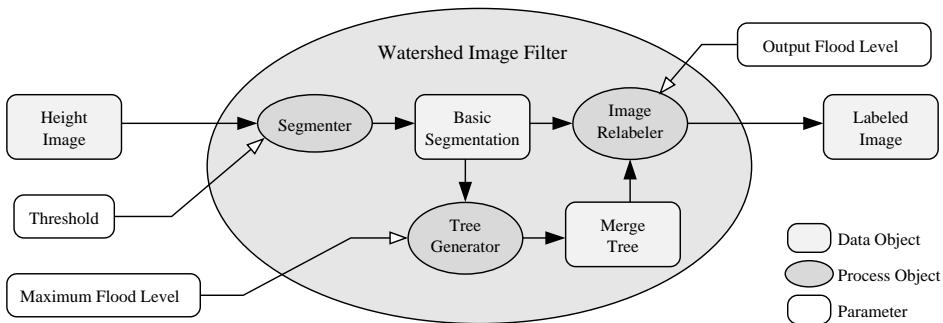


Figure 4.11: The construction of the Insight watersheds filter.

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* [56].) This limits the accuracy by enforcing a set of discrete gray levels on the image.

Figure 4.11 shows how the ITK image-to-image watershed 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 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 4.11 connected to their relevant processing stages.

4.2.2 Using the ITK Watershed Filter

The source code for this section can be found in the file `WatershedSegmentation1.cxx`.

The following example illustrates how to preprocess and segment images using the `itk::WatershedImageFilter`. Note that the care with which the data are 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 the bilateral image filter. As noted in Section 4.2.1, 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 dataset 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 applied directly to the vector-valued data and the segmentation uses 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< itk::IdentifierType, 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 2.7.3 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 special function object designed to hash a scalar value into an `itk::RGBPixel`. Plugging this functor into the `itk::UnaryFunctorImageFilter` creates an image filter which 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 4.12 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.

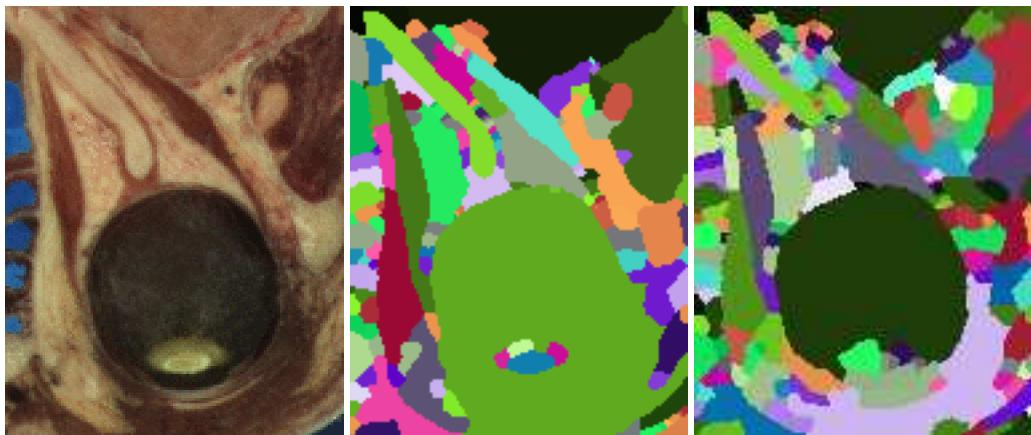


Figure 4.12: Segmented section of Visible Human female head and neck cryosection data. At left is the original image. The image in the middle was generated with parameters: conductance = 2.0, iterations = 10, threshold = 0.0, level = 0.05, principal components = on. The image on the right was generated with parameters: conductance = 2.0, iterations = 10, threshold = 0.001, level = 0.15, principal components = off.

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.

4.3 Level Set Segmentation

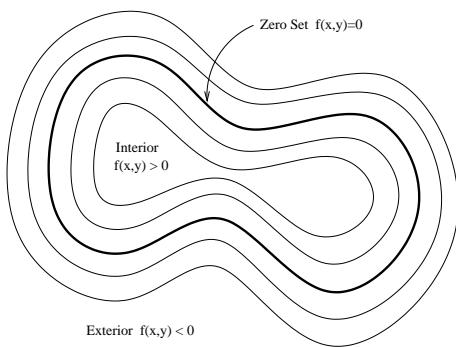


Figure 4.13: Concept of zero set in a level set.

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 $\Gamma(\mathbf{X}, \mathbf{t}) = \{\psi(\mathbf{X}, \mathbf{t}) = 0\}$ 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 [57].

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.

$$\frac{d}{dt}\psi = -\alpha \mathbf{A}(\mathbf{x}) \cdot \nabla \psi - \beta P(\mathbf{x}) |\nabla \psi| + \gamma Z(\mathbf{x}) \kappa |\nabla \psi| \quad (4.3)$$

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 produce 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

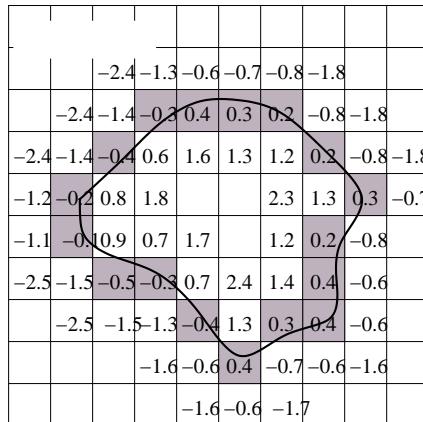


Figure 4.14: The implicit level set surface Γ is the black line superimposed over the image grid. The location of the surface is interpolated by the image pixel values. The grid pixels closest to the implicit surface are shown in gray.

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 4.14. 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 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

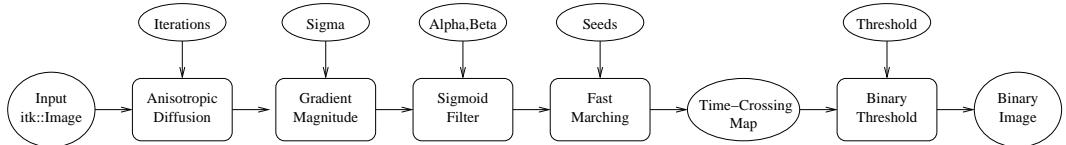


Figure 4.15: Collaboration diagram of the `FastMarchingImageFilter` applied to a segmentation task.

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.

4.3.1 Fast Marching Segmentation

The source code for this section can be found in the file `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 number 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 4.15 shows the major components involved in the application of the `FastMarchingIm-`

ageFilter 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 Curvature-AnisotropicDiffusionImageFilter 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 "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 2.3.2.

```
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 4.15 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 2.7.3.

```
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 2.4.2.

```
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()` method. 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()` method of this filter has been called directly or indirectly.

```
fastMarching->SetOutputSize( reader->GetOutput()->GetBufferedRegion().GetSize() );
```

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;
    return EXIT_FAILURE;
}
```

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 4.16 presents the intermediate outputs of the pipeline illustrated in Figure 4.15. 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

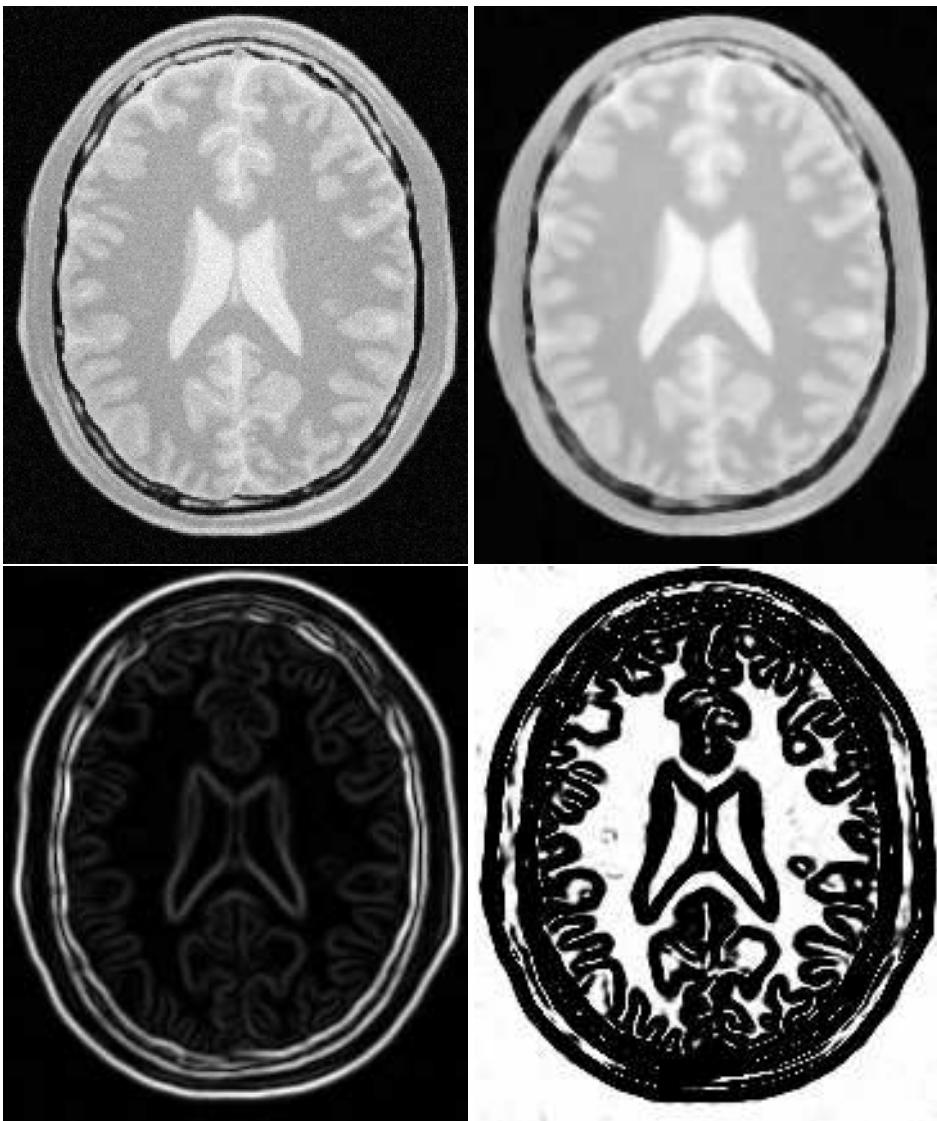


Figure 4.16: Images generated by the segmentation process based on the `FastMarchingImageFilter`. From left to right and top to bottom: input image to be segmented, image smoothed with an edge-preserving smoothing filter, gradient magnitude of the smoothed image, sigmoid of the gradient magnitude. This last image, the sigmoid, is used to compute the speed term for the front propagation.

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 4.3: Parameters used for segmenting some brain structures shown in Figure 4.17 using the filter `FastMarchingImageFilter`. All of them used a stopping value of 100.

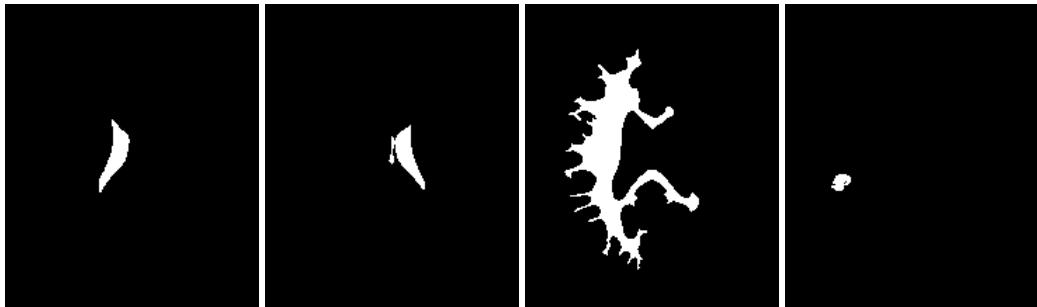


Figure 4.17: Images generated by the segmentation process based on the `FastMarchingImageFilter`. From left to right: segmentation of the left ventricle, segmentation of the right ventricle, segmentation of the white matter, attempt of segmentation of the gray matter.

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 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.

4.3.2 Shape Detection Segmentation

The source code for this section can be found in the file `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 [39]. 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

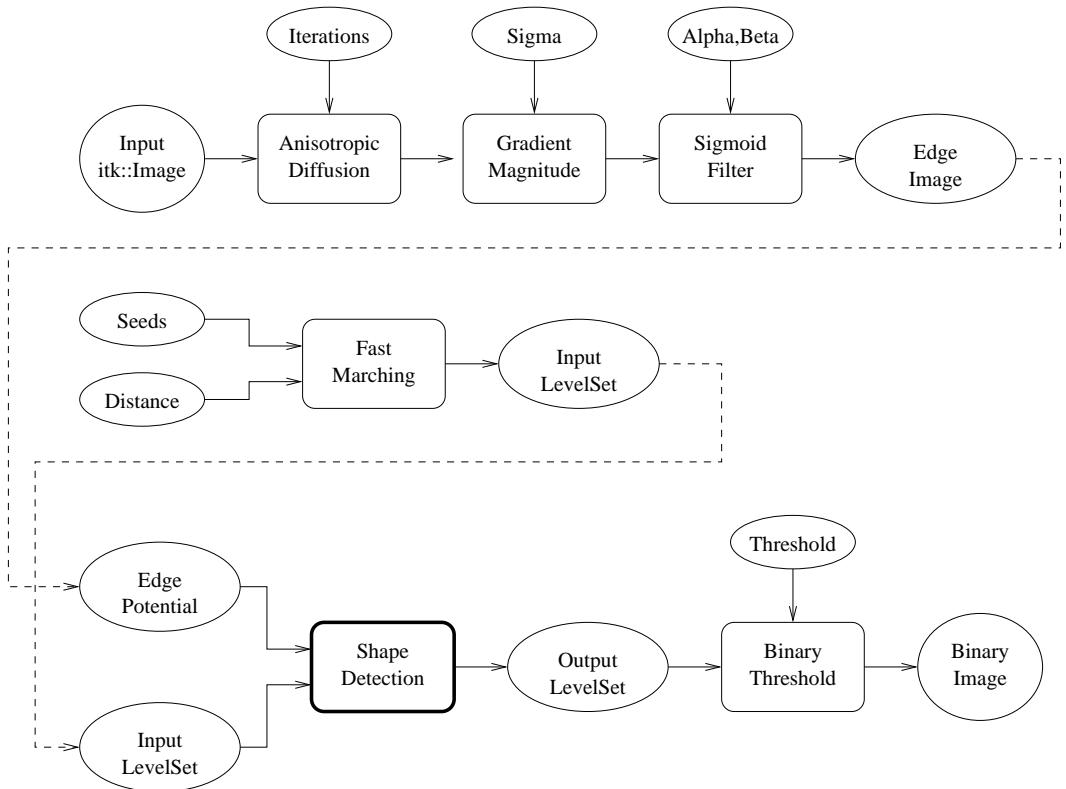


Figure 4.18: Collaboration diagram for the `ShapeDetectionLevelSetImageFilter` applied to a segmentation task.

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 4.3.1.

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 4.18 shows the major components involved in the application of the `ShapeDetectionLevelSetImageFilter` 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 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 `ShapeDetectionLevelSetImageFilter`.

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 `ShapeDetectionLevelSetImageFilter` class. Hence we include their headers here.

```
#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 2.3.2.

```

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 4.18 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 2.7.3.

```
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 2.4.2.

```
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 2.3.2 and 4.3.1.

```
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 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.

```
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;
return EXIT_FAILURE;
}
```

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 4.4 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 4.19 presents the intermediate outputs of the pipeline illustrated in Figure 4.18. 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 4.20 the segmented shapes are rounder than in Figure 4.17 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 required for segmenting large structures since it takes longer for the

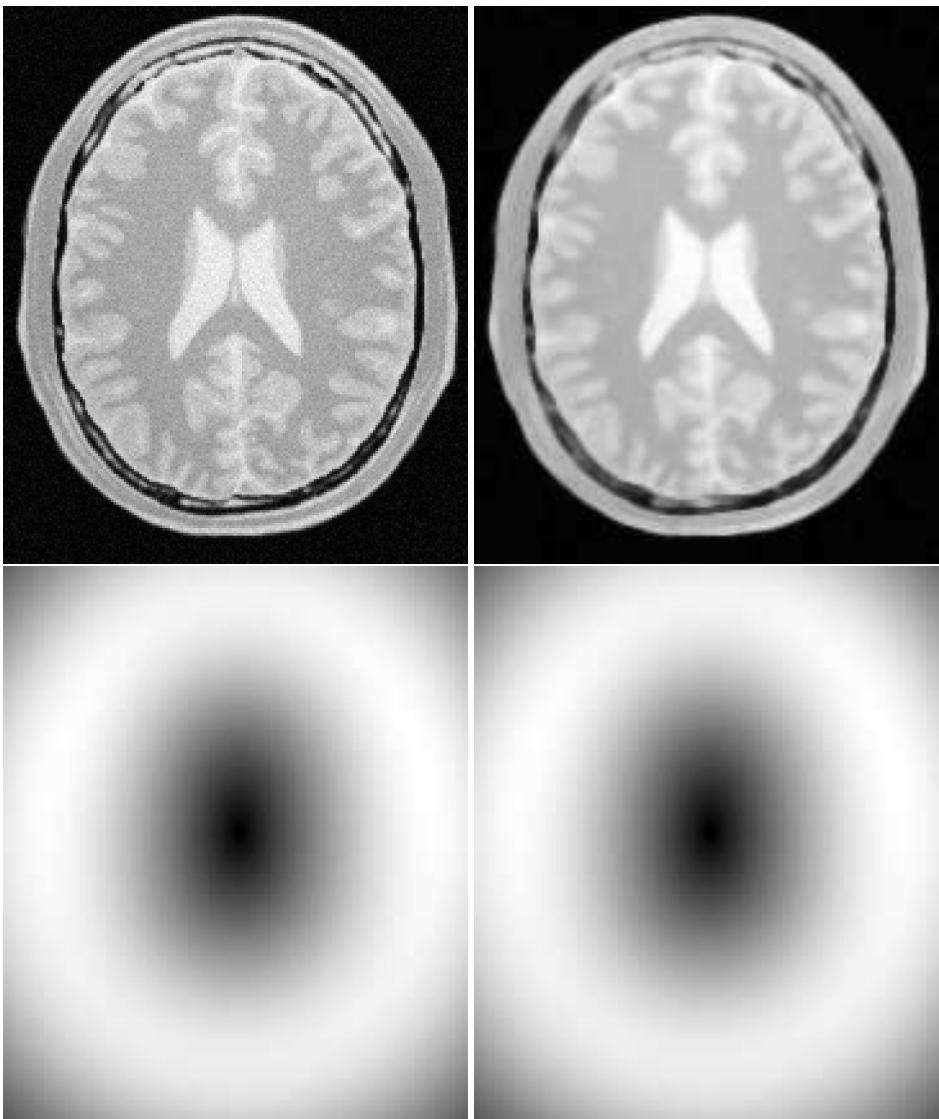


Figure 4.19: Images generated by the segmentation process based on the ShapeDetectionLevelSetImageFilter. From left to right and top to bottom: input image to be segmented, image smoothed with an edge-preserving smoothing filter, gradient magnitude of the smoothed image, sigmoid of the gradient magnitude. This last image, the sigmoid, is used to compute the speed term for the front propagation.

Structure	Seed Index	Distance	σ	α	β	Output Image
Left Ventricle	(81, 114)	5.0	1.0	-0.5	3.0	First in Figure 4.20
Right Ventricle	(99, 114)	5.0	1.0	-0.5	3.0	Second in Figure 4.20
White matter	(56, 92)	5.0	1.0	-0.3	2.0	Third in Figure 4.20
Gray matter	(40, 90)	5.0	0.5	-0.3	2.0	Fourth in Figure 4.20

Table 4.4: Parameters used for segmenting some brain structures shown in Figure 4.19 using the filter `ShapeDetectionLevelSetFilter`. All of them used a propagation scaling of 1.0 and curvature scaling of 0.05.

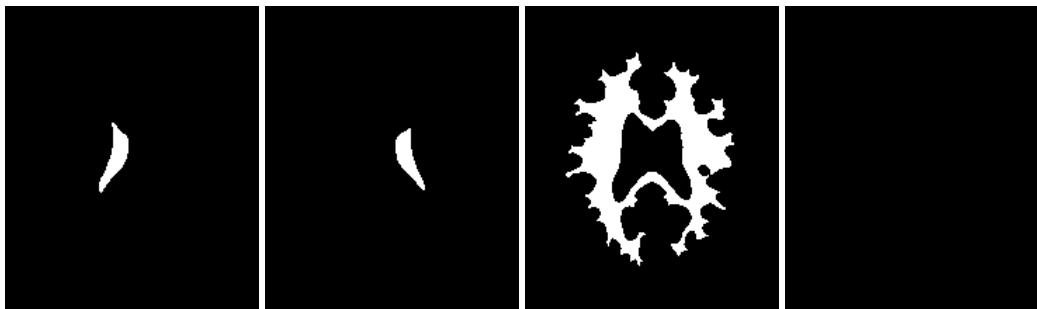


Figure 4.20: Images generated by the segmentation process based on the `ShapeDetectionLevelSetImageFilter`. From left to right: segmentation of the left ventricle, segmentation of the right ventricle, segmentation of the white matter, attempt of segmentation of the gray matter.

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.

4.3.3 Geodesic Active Contours Segmentation

The source code for this section can be found in the file `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 [11]. 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 4.3.2. The configuration of this example is quite similar to

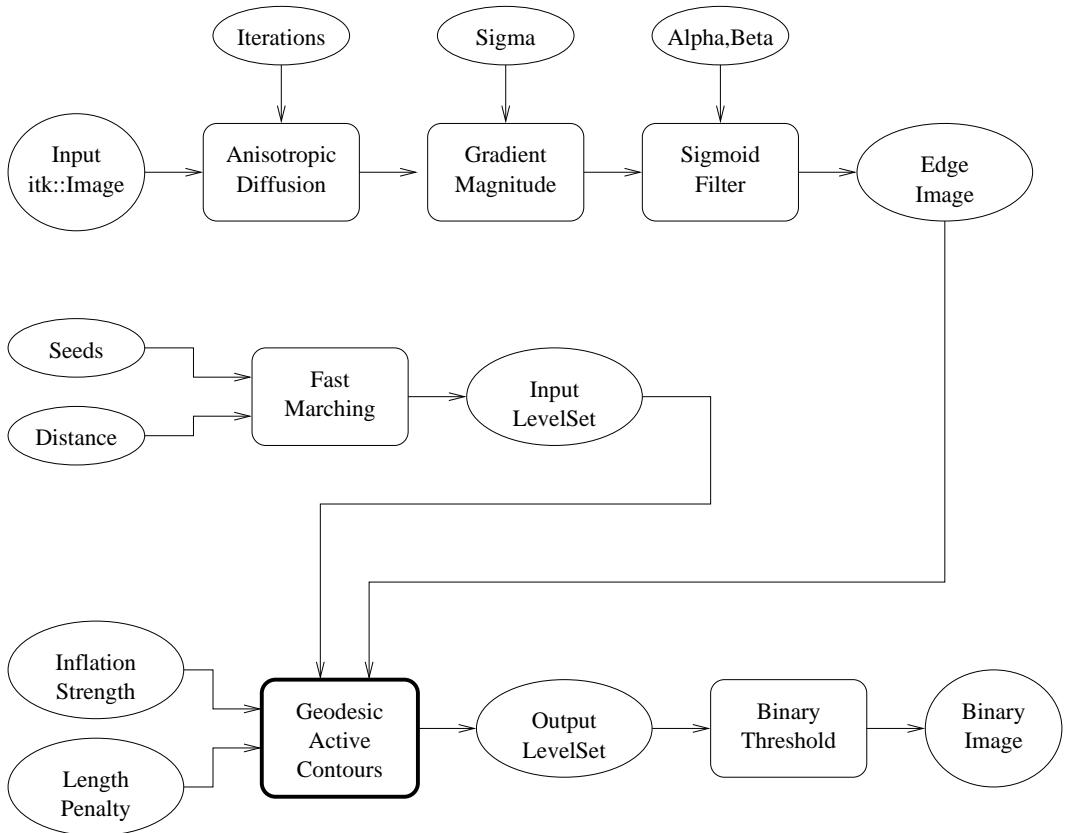


Figure 4.21: Collaboration diagram for the `GeodesicActiveContourLevelSetImageFilter` applied to a segmentation task.

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 4.21 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 4.3.2.

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`.

tImageFilter.

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 "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 4.21 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() );
```

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.

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 4.5: Parameters used for segmenting some brain structures shown in Figure 4.23 using the filter GeodesicActiveContourLevelSetImageFilter.

```

try
{
    writer->Update();
}
catch( itk::ExceptionObject & excep )
{
    std::cerr << "Exception caught !" << std::endl;
    std::cerr << excep << std::endl;
    return EXIT_FAILURE;
}

```

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 4.5 presents the parameters used for some structures.

Figure 4.22 presents the intermediate outputs of the pipeline illustrated in Figure 4.21. 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 4.23. The results are quite similar to those obtained with the ShapeDetectionLevelSetImageFilter in Section 4.3.2.

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.

4.3.4 Threshold Level Set Segmentation

The source code for this section can be found in the file `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 in-

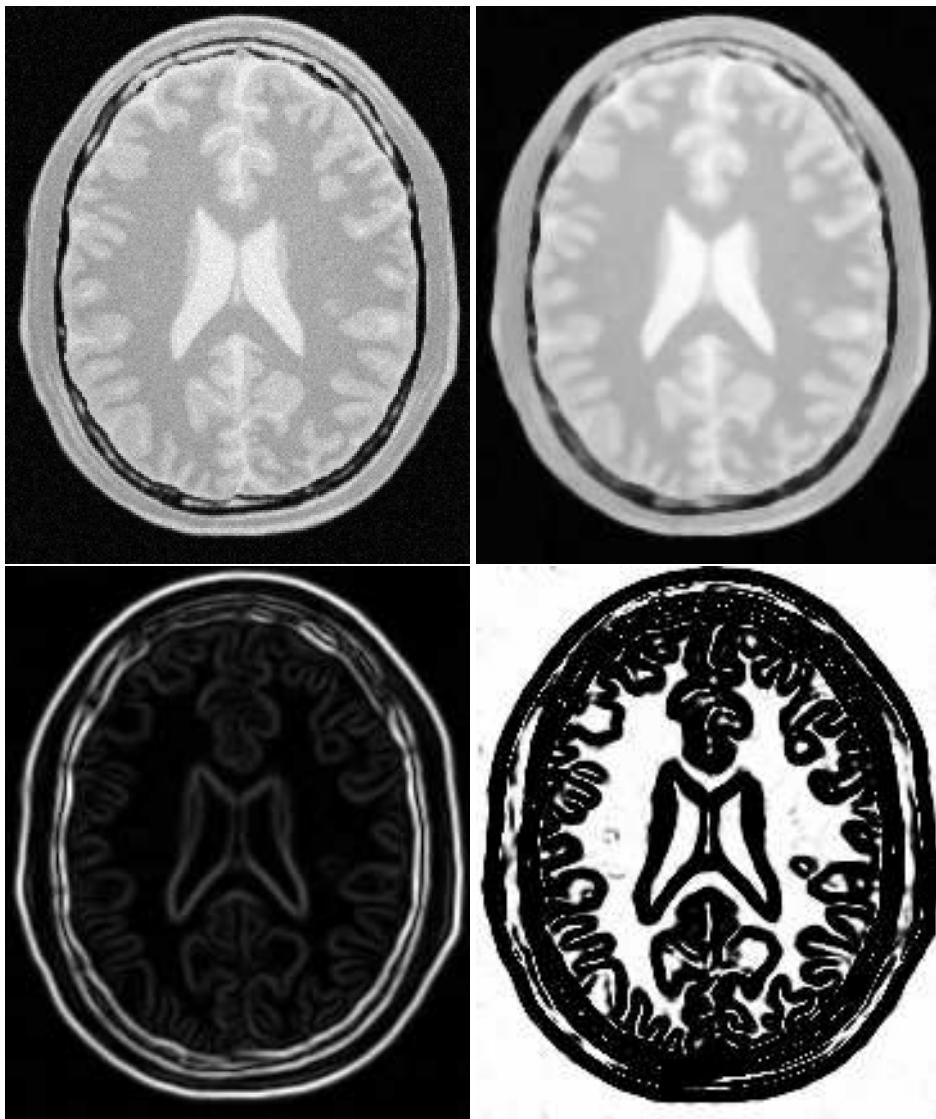


Figure 4.22: Images generated by the segmentation process based on the GeodesicActiveContourLevelSetImageFilter. From left to right and top to bottom: input image to be segmented, image smoothed with an edge-preserving smoothing filter, gradient magnitude of the smoothed image, sigmoid of the gradient magnitude. This last image, the sigmoid, is used to compute the speed term for the front propagation.

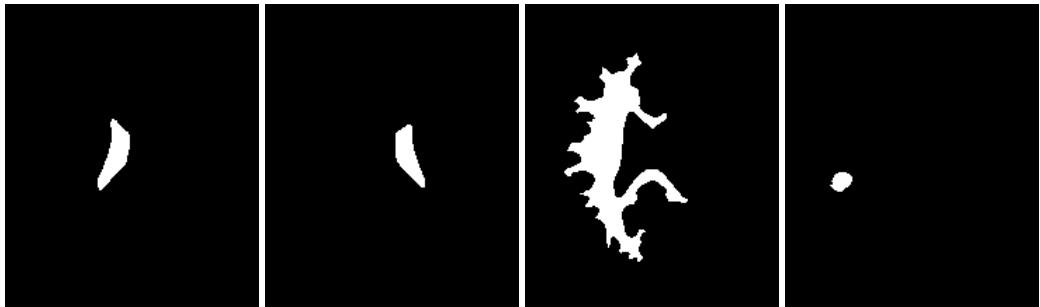


Figure 4.23: Images generated by the segmentation process based on the `GeodesicActiveContourImageFilter`. From left to right: segmentation of the left ventricle, segmentation of the right ventricle, segmentation of the white matter, attempt of segmentation of the gray matter.

tensity 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 4.3 is calculated from the `FeatureImage` input g with `UpperThreshold U` and `LowerThreshold L` according to the following formula.

$$P(\mathbf{x}) = \begin{cases} g(\mathbf{x}) - L & \text{if } g(\mathbf{x}) < (U - L)/2 + L \\ U - g(\mathbf{x}) & \text{otherwise} \end{cases} \quad (4.4)$$

Figure 4.25 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 .

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 4.24 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.

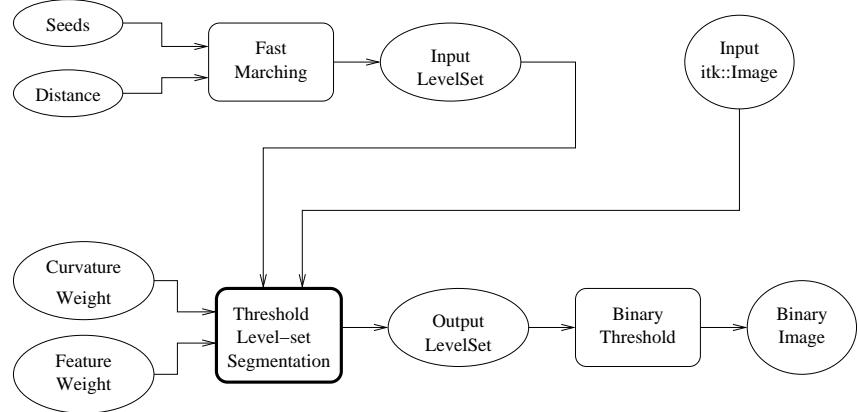


Figure 4.24: Collaboration diagram for the `ThresholdSegmentationLevelSetImageFilter` applied to a segmentation task.

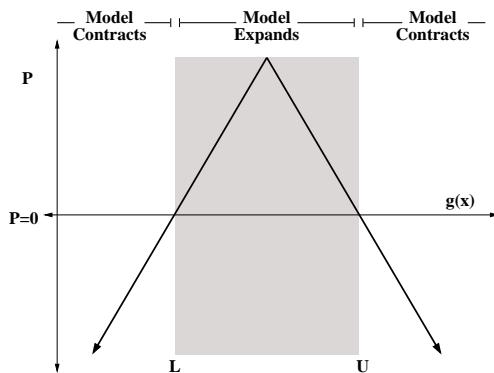


Figure 4.25: Propagation term for threshold-based level set segmentation. From Equation 4.4.

```
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 4.3. 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 4.24. 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.

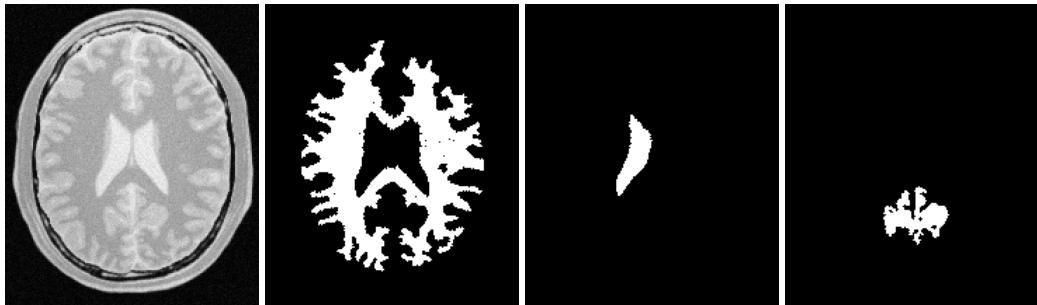


Figure 4.26: Images generated by the segmentation process based on the `ThresholdSegmentationLevelSetImageFilter`. From left to right: segmentation of the left ventricle, segmentation of the right ventricle, segmentation of the white matter, attempt of segmentation of the gray matter. The parameters used in this segmentations are presented in Table 4.6.

```

try
{
  reader->Update();
  const InternalImageType * inputImage = reader->GetOutput();
  fastMarching->SetOutputRegion( inputImage->GetBufferedRegion() );
  fastMarching->SetOutputSpacing( inputImage->GetSpacing() );
  fastMarching->SetOutputOrigin( inputImage->GetOrigin() );
  fastMarching->SetOutputDirection( inputImage->GetDirection() );
  writer->Update();
}
catch( itk::ExceptionObject & excep )
{
  std::cerr << "Exception caught !" << std::endl;
  std::cerr << excep << std::endl;
  return EXIT_FAILURE;
}

```

Let's run this application with the same data and parameters as the example given for `itk::ConnectedThresholdImageFilter` in Section 4.1.1. 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 4.26 with those in Figure 4.1. 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.

4.3.5 Canny-Edge Level Set Segmentation

The source code for this section can be found in the file `CannySegmentationLevelSetImageFilter.cxx`.

The `itk::CannySegmentationLevelSetImageFilter` defines a speed term that minimizes dis-

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 4.6: Segmentation results using the `ThresholdSegmentationLevelSetImageFilter` for various seed points. The resulting images are shown in Figure 4.26 .

tance 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 4.3. The advection term is constructed by minimizing the squared distance transform from the Canny edges.

$$\min \int D^2 \Rightarrow D \nabla D \quad (4.5)$$

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 4.27 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;
```

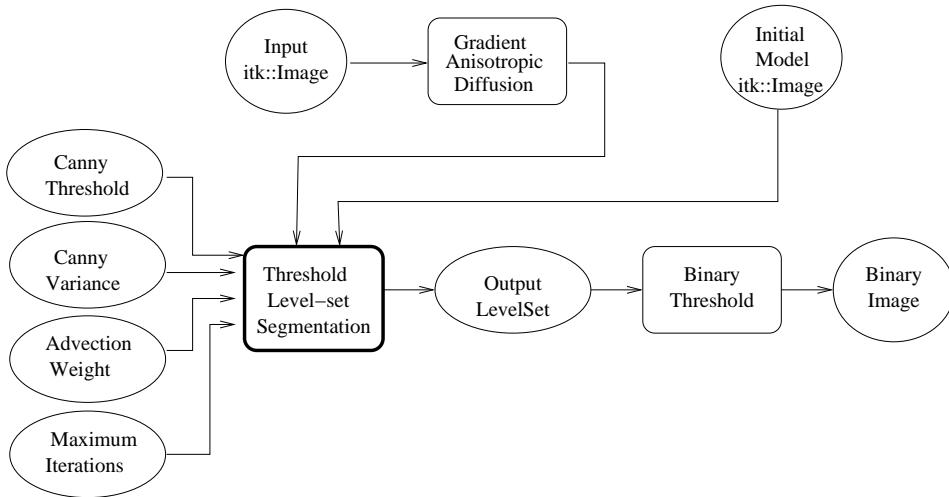


Figure 4.27: Collaboration diagram for the CannySegmentationLevelSetImageFilter applied to a segmentation task.

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 CannySegmentationLevelSetImageFilter 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 4.27.

```
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;
return EXIT_FAILURE;
}
```

We can use this filter to make some subtle refinements to the ventricle segmentation from the previous example that used the `itk::ThresholdSegmentationLevelSetImageFilter`. 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



Figure 4.28: 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 4.26, 15 iterations of the `CannySegmentationLevelSetImageFilter`, and the `CannySegmentationLevelSetImageFilter` run to convergence.

of Figure 4.28 with the ventricle segmentation from Figure 4.26 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() );
```

4.3.6 Laplacian Level Set Segmentation

The source code for this section can be found in the file `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.

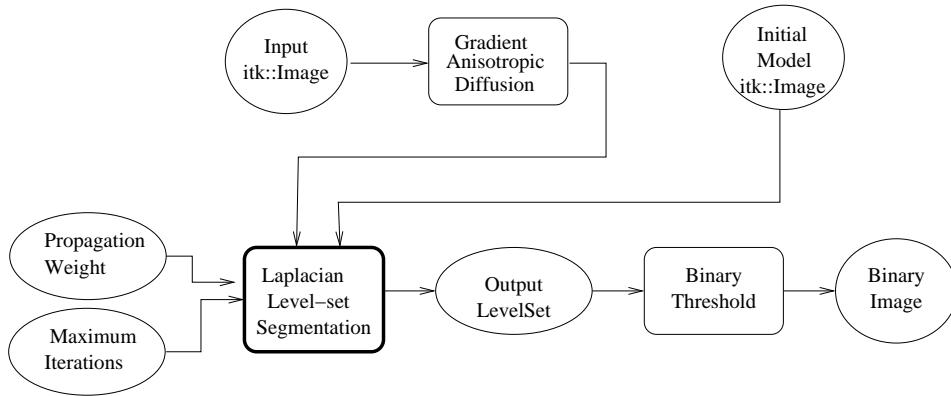


Figure 4.29: An image processing pipeline using `LaplacianSegmentationLevelSetImageFilter` for segmentation.

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 4.3). 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 4.29 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 LaplacianSegmentationLevelSetImageFilter 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 4.29.

```
diffusion->SetInput( reader1->GetOutput() );
laplacianSegmentation->SetInput( reader2->GetOutput() );
laplacianSegmentation->SetFeatureImage( diffusion->GetOutput() );
thresholder->SetInput( laplacianSegmentation->GetOutput() );
writer->SetInput( thresholder->GetOutput() );
```

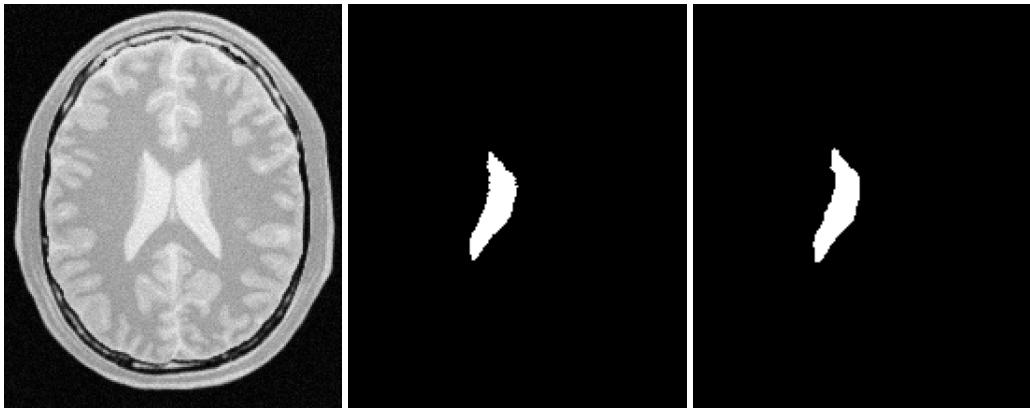


Figure 4.30: Results of applying `LaplacianSegmentationLevelSetImageFilter` to a prior ventricle segmentation. Shown from left to right are the original image, the prior segmentation of the ventricle from Figure 4.26, and the refinement of the prior using `LaplacianSegmentationLevelSetImageFilter`.

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;
    return EXIT_FAILURE;
}
```

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 4.30 with the ventricle segmentation from Figure 4.26 shown in the middle. Jagged edges are straightened and the small spur at the upper right-hand side of the mask has been removed.

4.3.7 Geodesic Active Contours Segmentation With Shape Guidance

The source code for this section can be found in the file `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 [34], Leventon *et al.* extended the geodesic active contours method with an additional shape-influenced term in the driving PDE. The `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(4.3)) is extended to incorporate a shape guidance term:

$$\xi(\psi^*(\mathbf{x}) - \psi(\mathbf{x})) \quad (4.6)$$

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 `GeodesicActiveContour-ShapePriorLevelSetImageFilter` 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 4.3.3 and hence the description will focus on the new objects involved in the segmentation process as shown in Figure 4.31.

The process pipeline begins with centering the input image using 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 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 4.31 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

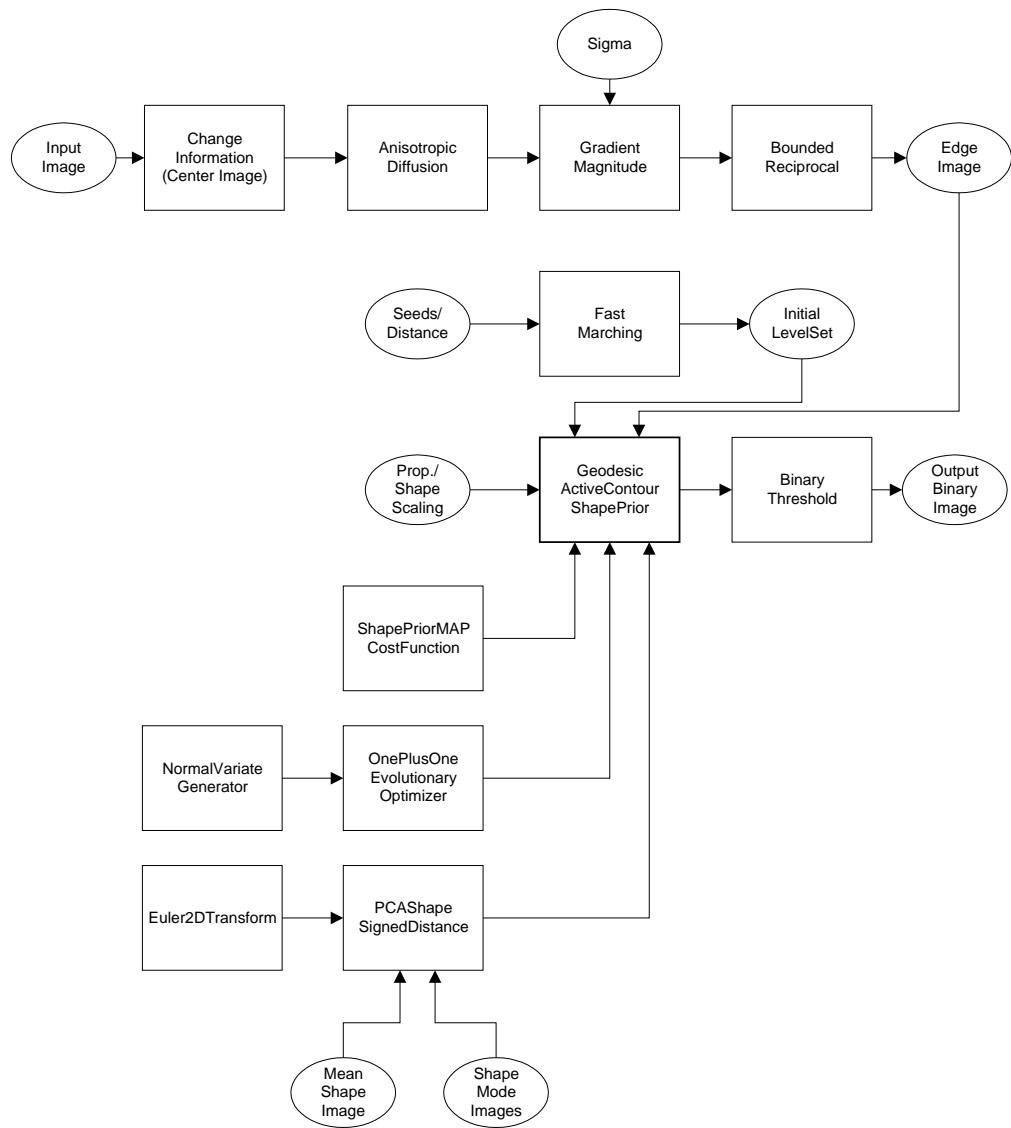


Figure 4.31: Collaboration diagram for the `GeodesicActiveContourShapePriorLevelSetImageFilter` applied to a segmentation task.

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 "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) ITK_OVERRIDE
    {
        Execute( (const itk::Object *) caller, event);
    }

    void Execute(const itk::Object * object,
                const itk::EventObject & event) ITK_OVERRIDE
    {
        const TFilter * filter = static_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 4.31.

```
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:

$$\psi^*(\mathbf{x}) = \mu(\mathbf{x}) + \sum_k \alpha_k u_k(\mathbf{x}) \quad (4.7)$$

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 correspond-

ing 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 $(1 - \text{edge potential})$ 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:

$$\sum (G(\psi(\mathbf{x})) - |1 - g(\mathbf{x})|)^2 \quad (4.8)$$

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[numberPCAModes + 1] = atof( argv[16] ); // startX
parameters[numberPCAModes + 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 );

```

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;
return EXIT_FAILURE;
}

```

Deviating from previous examples, we will demonstrate this example using `BrainMidSagittalSlice.png` (Figure 4.32, 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 4.33, 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.

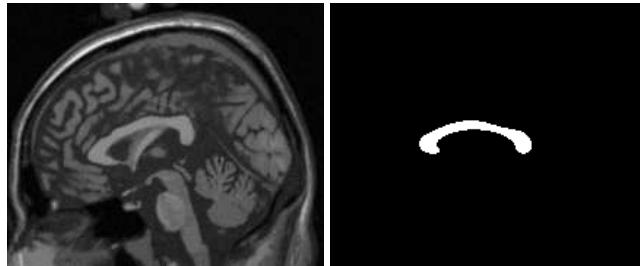


Figure 4.32: The input image to the GeodesicActiveContourShapePriorLevelSetImageFilter is a synthesized MR-T1 mid-sagittal slice (217×180 pixels, 1×1 mm spacing) of the brain (left) and the initial best-fit shape (right) chosen to roughly overlap the corpus callosum in the image to be segmented.

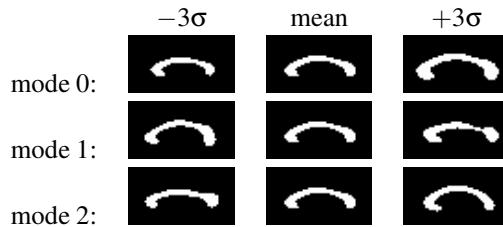


Figure 4.33: First three PCA modes of a low-resolution (58×31 pixels, 2×2 mm spacing) corpus callosum model used in the shape guided geodesic active contours example.

Segmentation results with and without shape guidance are shown in Figure 4.34.

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 4.32 (right).

From Figure 4.34 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 surrounding 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 4.34 (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



Figure 4.34: 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.

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.

4.4 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.

4.4.1 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 `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;
    return EXIT_FAILURE;
}
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(localImage->GetLargestPossibleRegion());
localOutputImage->SetRegions(region);
localOutputImage->CopyInformation(localImage);
localOutputImage->Allocate(true); // initialize buffer to zero

```

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 = std::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 = std::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 outputRegion =
        localOutputImage->GetLargestPossibleRegion();

    if( outputRegion.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;
    return EXIT_FAILURE;
}

```

Circle Extraction

The source code for this section can be found in the file `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;
    return EXIT_FAILURE;
}
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(true); // initializes buffer to zero

```

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* itk::Math::pi; angle += itk::Math::pi/60.0 )
{
    localIndex[0] =
        (long int)((*itCircles)->GetObjectToParentTransform()->GetOffset()[0]
            + (*itCircles)->GetRadius()[0]*std::cos(angle));
    localIndex[1] =
        (long int)((*itCircles)->GetObjectToParentTransform()->GetOffset()[1]
            + (*itCircles)->GetRadius()[0]*std::sin(angle));
    OutputImageType::RegionType outputRegion =
        localOutputImage->GetLargestPossibleRegion();

    if( outputRegion.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;
    return EXIT_FAILURE;
}
```

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 preprocess 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.

5.1 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.

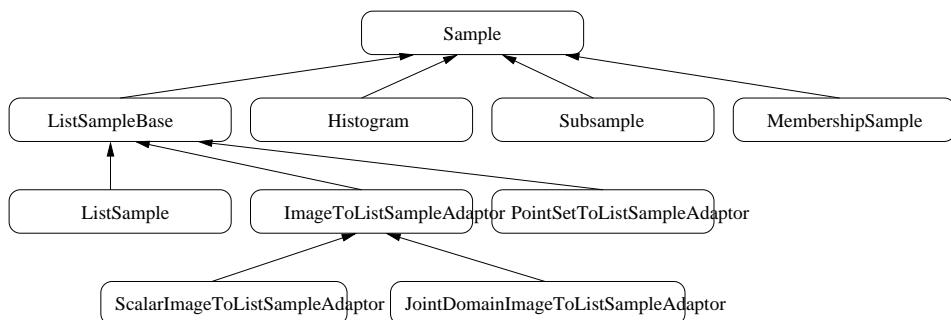


Figure 5.1: Sample class inheritance diagram.

5.1.1 Sample Interface

The source code for this section can be found in the file `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 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 a 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);
```

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 5.1.2), 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;

```

5.1.2 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 come from the source data container plugged into them. First, we will describe how to create an `itk::Statistics::ImageToListSampleAdaptor` and then an `itk::Statistics::PointSetToListSampleAdaptor` object.

ImageToListSampleAdaptor

The source code for this section can be found in the file `ImageToListSampleAdaptor.cxx`.

This example shows how to instantiate an `itk::Statistics::ImageToListSampleAdaptor` object and plug-in an `itk::Image` object as the data source for the adaptor.

In this example, we use the `ImageToListSampleAdaptor` class that requires the input type of `Image` as the template argument. To users of the `ImageToListSampleAdaptor`, the pixels of the input image are treated as measurement vectors. The `ImageToListSampleAdaptor` is one of two adaptor classes among the subclasses of the `itk::Statistics::Sample`. That means an `ImageToList-`

SampleAdaptor 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 ImageToListSampleAdaptor 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::ComposeImageFilter` which creates an image with pixels of array type from one or more input images composed of 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 of scalar pixels without the help from the ComposeImageFilter, we can use the `itk::Statistics::ScalarImageToListSampleAdaptor` class that is derived from the `itk::Statistics::ImageToListSampleAdaptor`. The usage of the ScalarImageToListSampleAdaptor is identical to that of the ImageToListSampleAdaptor.

```
#include "itkImageToListSampleAdaptor.h"
#include "itkImage.h"
#include "itkRandomImageSource.h"
#include "itkComposeImageFilter.h"
```

We assume you already know how to create an image. 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 );

typedef FloatImage2DType::SpacingValueType SpacingValueType;
typedef FloatImage2DType::SizeValueType    SizeValueType;
typedef FloatImage2DType::PointValueType   PointValueType;

SizeValueType size[2] = {20, 20};
random->SetSize( size );

SpacingValueType spacing[2] = {0.7, 2.1};
random->SetSpacing( spacing );

PointValueType 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`, or `itk::CovariantVector`).

Since the image pixel type is `float` in this example, we will use a single element `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::ComposeImageFilter< 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::ImageToListSampleAdaptor<
    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 5.1.1.

```
sample->SetImage( caster->GetOutput() );
```

If we are interested only in pixel values, the `ScalarImageToListSampleAdaptor` (scalar pixels) or the `ImageToListSampleAdaptor` (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::JointDomainImageToListSampleAdaptor` class. With this 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 `ImageToListSampleAdaptor`. One important difference between `JointDomainImageToListSampleAdaptor` and the other two image adaptors is that the `JointDomainImageToListSampleAdaptor` has the `SetNormalizationFactors()` method. Each component of a measurement vector from the `JointDomainImageToListSampleAdaptor` is divided by the corresponding component value from the supplied normalization factors.

PointSetToListSampleAdaptor

The source code for this section can be found in the file `PointSetToListSampleAdaptor.cxx`.

We will describe how to use `itk::PointSet` as a `itk::Statistics::Sample` using an adaptor in this example.

The `itk::Statistics::PointSetToListSampleAdaptor` 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 `PointSetToListSam-`

pleAdaptor takes only the point part into consideration. In other words, the measurement vectors from a PointSetToListSampleAdaptor object are points from the PointSet object that is plugged into the adaptor object.

To use an PointSetToListSampleAdaptor class, we include the header file for the class.

```
#include "itkPointSetToListSampleAdaptor.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. 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 the 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 creating 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 `PointSetToListSampleAdaptor` object cares only about the points, and the type of measurement vectors is the type of points.

To make the example a little bit realistic, we add two points 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 with two points in it. The `PointSet` is ready to be plugged into the adaptor. First, we create an instance of the `PointSetToListSampleAdaptor` class with the type of the input `PointSet` object.

```
typedef itk::Statistics::PointSetToListSampleAdaptor<
    PointSetType > SampleType;
SampleType::Pointer sample = SampleType::New();
```

Second, all we have to do is plug in the `PointSet` object to the adaptor. After that, we can use the common methods and iterator interfaces shown in Section [5.1.1](#).

```
sample->SetPointSet( pointSet );

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 source code for this section can be found in the file `PointSetToAdaptor.cxx`.

We will describe how to use `itk::PointSet` as a `Sample` using an adaptor in this example.

`itk::Statistics::PointSetToListSampleAdaptor` 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 `PointSetToListSampleAdaptor` takes only the point part into consideration. In other words, the measurement vectors from a `PointSetToListSampleAdaptor` object are points from the `PointSet` object that is plugged-into the adaptor object.

To use, an `itk::PointSetToListSampleAdaptor` object, we include the header file for the class.

```
#include "itkPointSetToListSampleAdaptor.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::PointSetToListSampleAdaptor<
                                         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 [5.1.1](#).

```
sample->SetPointSet( caster->GetOutput() );
```

5.1.3 Histogram

The source code for this section can be found in the file `Histogram.cxx`.

This example shows how to create an `itk::Statistics::Histogram` object and use it.

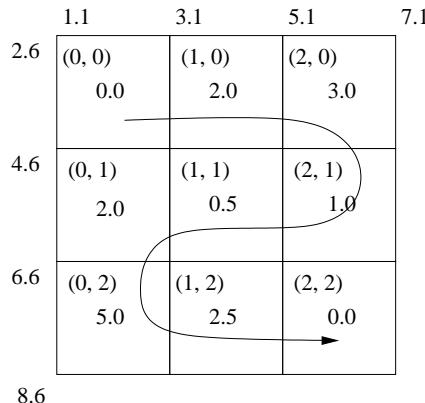


Figure 5.2: Conceptual histogram data structure.

We call an instance in a `Histogram` object a *bin*. The `Histogram` differs from the `itk::Statistics::ListSample`, `itk::Statistics::ImageToListSampleAdaptor`, or `itk::Statistics::PointSetToListSampleAdaptor` in significant ways. Histograms 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) with identical measurement vector values. However, in a `Histogram` object, there is one unique instance for any given measurement vector.

```
#include "itkHistogram.h"
#include "itkDenseFrequencyContainer2.h"
```

Here we create a histogram with dense frequency containers. In this example we will not have any zero-frequency measurements, so the dense frequency container is the appropriate choice. If the histogram is expected to have many empty (zero) bins, a sparse frequency container would be the better option. Here we also set the size of the measurement vectors to be 2 components.

```
typedef float MeasurementType;
typedef itk::Statistics::DenseFrequencyContainer2 FrequencyContainerType;
typedef FrequencyContainerType::AbsoluteFrequencyType FrequencyType;

const unsigned int numberOfComponents = 2;
typedef itk::Statistics::Histogram< MeasurementType,
    FrequencyContainerType > HistogramType;

HistogramType::Pointer histogram = HistogramType::New();
histogram->SetMeasurementVectorSize( numberOfComponents );
```

We initialize it as a 3×3 histogram with equal size intervals.

```
HistogramType::SizeType size( numberOfComponents );
size.Fill(3);
HistogramType::MeasurementVectorType lowerBound( numberOfComponents );
HistogramType::MeasurementVectorType upperBound( numberOfComponents );
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 each bin's frequency according to the Figure 5.2. 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, static_cast<FrequencyType>(0.0));
histogram->SetFrequency(1UL, static_cast<FrequencyType>(2.0));
histogram->SetFrequency(2UL, static_cast<FrequencyType>(3.0));
histogram->SetFrequency(3UL, static_cast<FrequencyType>(2.0f));
histogram->SetFrequency(4UL, static_cast<FrequencyType>(0.5f));
histogram->SetFrequency(5UL, static_cast<FrequencyType>(1.0f));
histogram->SetFrequency(6UL, static_cast<FrequencyType>(5.0f));
histogram->SetFrequency(7UL, static_cast<FrequencyType>(2.5f));
histogram->SetFrequency(8UL, static_cast<FrequencyType>(0.0f));
```

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( numberOfComponents );
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( numberOfComponents );
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 valid, 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;
```

5.1.4 Subsample

The source code for this section can be found in the file `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 elements 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;

```

5.1.5 MembershipSample

The source code for this section can be found in the file `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 << "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 `Size()` method of the `ClassSampleType` instance returned by the `GetClassSample(index)` method.

```
std::cout << "class label = 0 sample size = "
    << membershipSample->GetClassSample(0)->Size() << std::endl;
std::cout << "class label = 1 sample size = "
    << membershipSample->GetClassSample(1)->Size() << 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;
}
```

5.1.6 MembershipSampleGenerator

The source code for this section can be found in the file `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 to **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()` methods return the same values as the sample.

```
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 belong to the class **0**.

```
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;
}
```

5.1.7 K-d Tree

The source code for this section can be found in the file `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 "itkMath.h"
#include "itkListSample.h"
#include "itkWeightedCentroidKdTreeGenerator.h"
#include "itkEuclideanDistanceMetric.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 [29].

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::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::EuclideanDistanceMetric< 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 <<
"\n*** kd-tree knn search result using an Euclidean distance metric:"
<< std::endl
<< "query point = [" << queryPoint << "]"
<< "k = " << numberOfNeighbors << std::endl;
std::cout << "measurement vector : distance from query point " << std::endl;
std::vector<double> distances1( numberOfNeighbors );
for ( unsigned int i = 0; i < numberOfNeighbors; ++i )
{
    distances1[i] = distanceMetric->Evaluate(
        tree->GetMeasurementVector( neighbors[i] ) );
    std::cout << "[" << tree->GetMeasurementVector( neighbors[i] )
        << "] : "
        << distances1[i]
        << std::endl;
}
```

Instead of using an Euclidean distance metric, Tree itself can also return the distance vector. Here we get the distance values from tree and compare them with previous values.

```

std::vector<double> distances2;
tree->Search( queryPoint, numberOfNeighbors, neighbors, distances2 );

std::cout << "\n*** kd-tree knn search result directly from tree:"
    << std::endl
    << "query point = [" << queryPoint << "]"
    << "k = " << numberOfNeighbors << std::endl;
std::cout << "measurement vector : distance from query point " << std::endl;
for ( unsigned int i = 0; i < numberOfNeighbors; ++i )
{
    std::cout << "[" << tree->GetMeasurementVector( neighbors[i] )
        << "] : "
        << distances2[i]
        << std::endl;
    if ( itk::Math::NotAlmostEquals( distances2[i], distances1[i] ) )
    {
        std::cerr << "Mismatched distance values by tree." << std::endl;
        return EXIT_FAILURE;
    }
}

```

As previously indicated, the interface for finding nearest neighbors in the centroid tree is very similar.

```

std::vector<double> distances3;
centroidTree->Search(
    queryPoint, numberOfNeighbors, neighbors, distances3 );

centroidTree->Search( queryPoint, numberOfNeighbors, neighbors );
std::cout << "\n*** Weighted centroid kd-tree knn search result:"
    << std::endl
    << "query point = [" << queryPoint << "]"
    << "k = " << numberOfNeighbors << std::endl;
std::cout << "measurement vector : distance_by_distMetric : distance_by_tree"
    << std::endl;
std::vector<double> distances4 (numberOfNeighbors);
for ( unsigned int i = 0; i < numberOfNeighbors; ++i )
{
    distances4[i] = distanceMetric->Evaluate(
        centroidTree->GetMeasurementVector( neighbors[i] ));
    std::cout << "[" << centroidTree->GetMeasurementVector( neighbors[i] )
        << "] : "
        << distances4[i]
        << " : "
        << distances3[i]
        << std::endl;
    if ( itk::Math::NotAlmostEquals( distances2[i], distances1[i] ) )
    {
        std::cerr << "Mismatched distance values by centroid tree." << std::endl;
        return EXIT_FAILURE;
    }
}

```

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 << "\nSearching points within a hyper-spherical kernel:"
    << std::endl;
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 << "\n*** 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;
}
```

5.2 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.

5.2.1 Sample Statistics

We will show how to get sample statistics such as means and covariance from the (`itk::Statistics::Sample`) classes. Statistics can tell 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 5.2.1). 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 `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 "itkMeanSampleFilter.h"
#include "itkCovarianceSampleFilter.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::MeanSampleFilter` 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 `GetMean()` method. The output from the `GetOutput()` method is the pointer to the mean vector.

```

typedef itk::Statistics::MeanSampleFilter< SampleType > MeanAlgorithmType;

MeanAlgorithmType::Pointer meanAlgorithm = MeanAlgorithmType::New();

meanAlgorithm->SetInput( sample );
meanAlgorithm->Update();

std::cout << "Sample mean = " << meanAlgorithm->GetMean() << std::endl;

```

The covariance calculation algorithm will also calculate the mean while performing the covariance matrix calculation. The mean can be accessed using the `GetMean()` method while the covariance can be accessed using the `GetCovarianceMatrix()` method.

```
typedef itk::Statistics::CovarianceSampleFilter< SampleType >
    CovarianceAlgorithmType;
CovarianceAlgorithmType::Pointer covarianceAlgorithm =
    CovarianceAlgorithmType::New();

covarianceAlgorithm->SetInput( sample );
covarianceAlgorithm->Update();

std::cout << "Mean = " << std::endl;
std::cout << covarianceAlgorithm->GetMean() << std::endl;

std::cout << "Covariance = " << std::endl;
std::cout << covarianceAlgorithm->GetCovarianceMatrix() << std::endl;
```

Weighted Mean and Covariance

The source code for this section can be found in the file `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 "itkWeightedMeanSampleFilter.h"
#include "itkWeightedCovarianceSampleFilter.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::WeightedMeanSampleFilter< 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->SetInput( sample );
weightedMeanAlgorithm->SetWeights( weightArray );
weightedMeanAlgorithm->Update();

std::cout << "Sample weighted mean = "
    << weightedMeanAlgorithm->GetMean() << std::endl;

typedef itk::Statistics::WeightedCovarianceSampleFilter< SampleType >
WeightedCovarianceAlgorithmType;

WeightedCovarianceAlgorithmType::Pointer weightedCovarianceAlgorithm =
    WeightedCovarianceAlgorithmType::New();

weightedCovarianceAlgorithm->SetInput( sample );
weightedCovarianceAlgorithm->SetWeights( weightArray );
weightedCovarianceAlgorithm->Update();

std::cout << "Sample weighted covariance = " << std::endl;
std::cout << weightedCovarianceAlgorithm->GetCovarianceMatrix() << 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 `SetWeightingFunction(weights)` method.

```

ExampleWeightFunction::Pointer weightFunction = ExampleWeightFunction::New();

weightedMeanAlgorithm->SetWeightingFunction( weightFunction );
weightedMeanAlgorithm->Update();

std::cout << "Sample weighted mean = "
    << weightedMeanAlgorithm->GetMean() << std::endl;

weightedCovarianceAlgorithm->SetWeightingFunction( weightFunction );
weightedCovarianceAlgorithm->Update();

std::cout << "Sample weighted covariance = " << std::endl;
std::cout << weightedCovarianceAlgorithm->GetCovarianceMatrix();

std::cout << "Sample weighted mean (from WeightedCovarianceSampleFilter) = "
    << std::endl << weightedCovarianceAlgorithm->GetMean()
    << std::endl;

```

5.2.2 Sample Generation

SampleToHistogramFilter

The source code for this section can be found in the file `SampleToHistogramFilter.cxx`.

Sometimes we want to work with a histogram instead of a list of measurement vectors (e.g. `itk::Statistics::ListSample`, `itk::Statistics::ImageToListSampleAdaptor`, or `itk::Statistics::PointSetToListSample`) to use less memory or to perform a particular type of analysis. In such cases, we can import data from a sample type to a `itk::Statistics::Histogram` object using the `itk::Statistics::SampleToHistogramFilter`.

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 "itkSampleToHistogramFilter.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 set up the size and bound of the output histogram.

```

typedef float HistogramMeasurementType;
const unsigned int numberOfComponents = 2;
typedef itk::Statistics::Histogram< HistogramMeasurementType >
HistogramType;

HistogramType::SizeType size( numberOfComponents );
size.Fill(5);

HistogramType::MeasurementVectorType lowerBound( numberOfComponents );
HistogramType::MeasurementVectorType upperBound( numberOfComponents );

lowerBound[0] = 0.5;
lowerBound[1] = 0.5;

upperBound[0] = 5.5;
upperBound[1] = 5.5;

```

Now, we set up the `SampleToHistogramFilter` object by passing `listSample` as the input and initializing the histogram size and bounds with the `SetHistogramSize()`, `SetHistogramBinMinimum()`, and `SetHistogramBinMaximum()` methods. We execute the filter by calling the `Update()` method.

```
typedef itk::Statistics::SampleToHistogramFilter< ListSampleType,
                                                HistogramType > FilterType;
FilterType::Pointer filter = FilterType::New();

filter->SetInput( listSample );
filter->SetHistogramSize( size );
filter->SetHistogramBinMinimum( lowerBound );
filter->SetHistogramBinMaximum( upperBound );
filter->Update();
```

The `Size()` and `GetTotalFrequency()` methods return the same values as the sample does.

```
const HistogramType* histogram = filter->GetOutput();

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 `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;
}

```

5.2.3 Sample Sorting

The source code for this section can be found in the file `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 [43] for samples are implemented in ITK. To learn pros and cons of each algorithm, please refer to [19]. 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"
```

The sorting and selecting related functions are in the include file `itkStatisticsAlgorithm.h`. Note that all functions in this file are in the `itk::Statistics::Algorithm` namespace.

```
#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::Algorithm::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::Algorithm::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::Algorithm::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::Algorithm::QuickSelect< SubsampleType >( subsample,
        activeDimension,
        0, subsample->Size(),
        subsample->Size()/2 );

std::cout << std::endl;
std::cout << "Quick Select: median = " << median << std::endl;
```

5.2.4 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 `GaussianMembershipFunction.cxx`.

The Gaussian probability density function `itk::Statistics::GaussianMembershipFunction` 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 "itkGaussianMembershipFunction.h"
```

We define the type of the measurement vector that will be input to the Gaussian membership 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::GaussianMembershipFunction< MeasurementVectorType >
DensityFunctionType;
DensityFunctionType::Pointer densityFunction = DensityFunctionType::New();
```

The length of the measurement vectors in the membership 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::MeanVectorType mean( 2 );
mean.Fill( 0.0 );

DensityFunctionType::CovarianceMatrixType 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;

```

5.2.5 Distance Metric

Euclidean Distance

The source code for this section can be found in the file `EuclideanDistanceMetric.cxx`.

The Euclidean distance function (`itk::Statistics::EuclideanDistanceMetric` 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, `EuclideanDistanceMetric` 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 "itkEuclideanDistanceMetric.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::EuclideanDistanceMetric< 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 `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;
```

5.2.6 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 `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::Statistics::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.

```
DecisionRuleType::MembershipVectorType 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 `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::Statistics::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.

```
DecisionRuleType::MembershipVectorType 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
`MaximumRatioDecisionRule.cxx`.

`MaximumRatioDecisionRule` returns the class label using a Bayesian style decision rule. The discriminant scores are evaluated in the context of class priors. If the discriminant scores are actual conditional probabilities (likelihoods) and the class priors are actual a priori class probabilities, then this decision rule operates as Bayes rule, returning the class i if

$$p(x|i)p(i) > p(x|j)p(j) \quad (5.1)$$

for all class j . The discriminant scores and priors are not required to be true probabilities.

This class is named the `MaximumRatioDecisionRule` as it can be implemented as returning the class i if

$$\frac{p(x|i)}{p(x|j)} > \frac{p(j)}{p(i)} \quad (5.2)$$

for all class j .

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::Statistics::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.

```
DecisionRuleType::MembershipVectorType discriminantScores;
discriminantScores.push_back( 0.1 );
discriminantScores.push_back( 0.3 );
discriminantScores.push_back( 0.6 );

DecisionRuleType::PriorProbabilityVectorType aPrioris;
aPrioris.push_back( 0.1 );
aPrioris.push_back( 0.8 );
aPrioris.push_back( 0.1 );

decisionRule->SetPriorProbabilities( aPrioris );
std::cout << "MaximumRatioDecisionRule: The index of the chosen = "
      << decisionRule->Evaluate( discriminantScores )
      << std::endl;
```

5.2.7 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 `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;
}
```

5.3 Statistics applied to Images

5.3.1 Image Histograms

Scalar Image Histogram with Adaptor

The source code for this section can be found in the file `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::ImageToListSampleAdaptor`. Once we have connected this adaptor to an image, we can proceed to use the `itk::Statistics::SampleToHistogramFilter` in order to compute the histogram of the image.

First, we need to include the headers for the `itk::Statistics::ImageToListSampleAdaptor` and the `itk::Image` classes.

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

Now we include the headers for the Histogram, the `SampleToHistogramFilter`, and the reader that we will use for reading the image from a file.

```
#include "itkImageFileReader.h"
#include "itkHistogram.h"
#include "itkSampleToHistogramFilter.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::ImageToListSampleAdaptor< 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 EXIT_FAILURE;  
}
```

At this point, we are ready for instantiating the type of the histogram filter. We must first declare the type of histogram we wish to use. The adaptor type is also used as template parameter of the filter. Having instantiated this type, we proceed to create one filter by invoking its `New()` method.

```
typedef PixelType HistogramMeasurementType;  
typedef itk::Statistics::Histogram< HistogramMeasurementType >  
    HistogramType;  
typedef itk::Statistics::SampleToHistogramFilter<  
    AdaptorType,  
    HistogramType>  
    FilterType;  
  
FilterType::Pointer filter = FilterType::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 filter.

```
const unsigned int numberOfComponents = 1;
HistogramType::SizeType size( numberOfComponents );
size.Fill( 255 );

filter->SetInput( adaptor );
filter->SetHistogramSize( size );
filter->SetMarginalScale( 10 );

HistogramType::MeasurementVectorType min( numberOfComponents );
HistogramType::MeasurementVectorType max( numberOfComponents );

min.Fill( 0 );
max.Fill( 255 );

filter->SetHistogramBinMinimum( min );
filter->SetHistogramBinMaximum( max );

filter->Update();
```

Now we are ready for using the image histogram for any further processing. The histogram is obtained from the filter by invoking the `GetOutput()` method.

```
HistogramType::ConstPointer histogram = filter->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 `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( 256 );
histogramGenerator->SetMarginalScale( 10.0 );

histogramGenerator->SetHistogramMin( -0.5 );
histogramGenerator->SetHistogramMax( 255.5 );
```

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 `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 `ImageToHistogramFilter`. In this first example we compute the histogram of each channel independently.

We start by including the header of the `itk::Statistics::ImageToHistogramFilter`, as well as the headers for the image class and the `RGBPixel` class.

```
#include "itkImageToHistogramFilter.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 filter and construct one filter by invoking its `New()` method.

```
typedef itk::Statistics::ImageToHistogramFilter<
    RGBImageType >   HistogramFilterType;
HistogramFilterType::Pointer histogramFilter =
    HistogramFilterType::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 `HistogramFilterType` 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 HistogramFilterType::HistogramSizeType   SizeType;
SizeType size( 3 );
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
histogramFilter->SetHistogramSize( size );
```

The marginal scale must be defined in the filter. This will determine the precision in the assignment of values to the histogram bins.

```
histogramFilter->SetMarginalScale( 10.0 );
```

Finally, we must specify the upper and lower bounds for the histogram. This can either be done manually using the `SetHistogramBinMinimum()` and `SetHistogramBinMaximum()` methods or it can be done automatically by calling `SetHistogramAutoMinimumMaximum(true)`. Here we use the manual method.

```
HistogramFilterType::HistogramMeasurementVectorType lowerBound( 3 );
HistogramFilterType::HistogramMeasurementVectorType upperBound( 3 );

lowerBound[0] = 0;
lowerBound[1] = 0;
lowerBound[2] = 0;
upperBound[0] = 256;
upperBound[1] = 256;
upperBound[2] = 256;

histogramFilter->SetHistogramBinMinimum( lowerBound );
histogramFilter->SetHistogramBinMaximum( upperBound );
```

The input of the filter is taken from an image reader, and the computation of the histogram is triggered by invoking the `Update()` method of the filter.

```
histogramFilter->SetInput( reader->GetOutput() );
histogramFilter->Update();
```

We can now access the results of the histogram computation by declaring a pointer to histogram and getting its value from the filter 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 filter 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 HistogramFilterType::HistogramType HistogramType;

const HistogramType * histogram = histogramFilter->GetOutput();
```

Just for the sake of exercising the experimental method [49], we verify that the resulting histogram actually have the size that we requested when we configured the filter. 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 always 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 filter 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 filter again by calling the `Update()` 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

histogramFilter->SetHistogramSize( size );

histogramFilter->Update();
```

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

histogramFilter->SetHistogramSize( size );

histogramFilter->Update();
```

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 `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::ImageToHistogramFilter`. This class will take care of adapting the `itk::Image` to a list of samples and then to a histogram filter. 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 filter, the RGB pixel type and the Image.

```

#include "itkImageToHistogramFilter.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 filter class. We then use that type for constructing an instance of the filter by invoking its `New()` method and assigning the result to a smart pointer.

```
typedef itk::Statistics::ImageToHistogramFilter<
    RGBImageType > HistogramFilterType;

HistogramFilterType::Pointer histogramFilter =
    HistogramFilterType::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 `HistogramSizeType` trait from the filter 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 HistogramFilterType::HistogramSizeType SizeType;

SizeType size(3);

size[0] = 256; // number of bins for the Red channel
size[1] = 256; // number of bins for the Green channel
size[2] = 256; // number of bins for the Blue channel

histogramFilter->SetHistogramSize( size );
```

Finally, we must specify the upper and lower bounds for the histogram using the `SetHistogramBinMinimum()` and `SetHistogramBinMaximum()` methods.

```
typedef HistogramFilterType::HistogramMeasurementVectorType
    HistogramMeasurementVectorType;

HistogramMeasurementVectorType binMinimum( 3 );
HistogramMeasurementVectorType binMaximum( 3 );

binMinimum[0] = -0.5;
binMinimum[1] = -0.5;
binMinimum[2] = -0.5;

binMaximum[0] = 255.5;
binMaximum[1] = 255.5;
binMaximum[2] = 255.5;

histogramFilter->SetHistogramBinMinimum( binMinimum );
histogramFilter->SetHistogramBinMaximum( binMaximum );
```

The input to the histogram filter 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.

```
histogramFilter->SetInput( reader->GetOutput() );
```

The marginal scale is defined in the histogram filter. This value will define the precision in the

assignment of values to the histogram bins.

```
histogramFilter->SetMarginalScale( 10.0 );
```

Finally, the computation of the histogram is triggered by invoking the `Update()` method of the filter.

```
histogramFilter->Update();
```

At this point, we can recover the histogram by calling the `GetOutput()` method of the filter. The result is assigned to a variable that is instantiated using the `HistogramType` trait of the filter type.

```
typedef HistogramFilterType::HistogramType HistogramType;  
  
const HistogramType * histogram = histogramFilter->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::AbsoluteFrequencyType AbsoluteFrequencyType;  
  
while( itr != end )  
{  
    const AbsoluteFrequencyType frequency = itr.GetFrequency();  
    histogramFile.write( (const char *)&frequency, sizeof(frequency) );  
  
    if (frequency != 0)  
    {  
        HistogramType::IndexType index;  
        index = histogram->GetIndex(itr.GetInstanceIdentifier());  
        std::cout << "Index = " << index << ", Frequency = " << frequency  
              << std::endl;  
    }  
    ++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.

5.3.2 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 [58, 59, 33].

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 map 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 information¹.

Since we could have picked any pixel at random, our chances of picking the ones that are associated to the i -th histogram bin are given by p_i . Therefore, the expected reduction in uncertainty that we can get from measuring the value of one pixel is given by

¹Note 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 have 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.

$$H = - \sum_i p_i \cdot \log_2(p_i) \quad (5.3)$$

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 H* actually represents. 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 convention 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 and 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
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 forced 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 probability 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 a 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 = 3;

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, including 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 to 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 makes it 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 * std::log( probability ) / std::log( 2.0 );
    }
    ++itr;
}
```

The result of this sum is considered to be our estimation of the image entropy. Note that the Entropy 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 `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 ImageMetrics 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 measure the amount of information that one image provides on the other image; this would require us 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, among the most interesting of which is the semantic description of image in 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 filter, 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 "itkImageToHistogramFilter.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 EXIT_FAILURE;
}
```

We now prepare 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::ImageToHistogramFilter`. We then construct one by invoking the `New()` method.

```
typedef JoinFilterType::OutputImageType VectorImageType;

typedef itk::Statistics::ImageToHistogramFilter<
    VectorImageType > HistogramFilterType;

HistogramFilterType::Pointer histogramFilter = HistogramFilterType::New();
```

We pass the multiple-component image as input to the histogram filter, and setup the marginal scale value that will define the precision to be used for classifying values into the histogram bins.

```
histogramFilter->SetInput( joinFilter->GetOutput() );
histogramFilter->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 `HistogramSizeType` from the traits of the histogram filter type.

```
typedef HistogramFilterType::HistogramSizeType HistogramSizeType;

HistogramSizeType size( 2 );

size[0] = 255; // number of bins for the first channel
size[1] = 255; // number of bins for the second channel

histogramFilter->SetHistogramSize( size );
```

Finally, we must specify the upper and lower bounds for the histogram using the `SetHistogramBinMinimum()` and `SetHistogramBinMaximum()` methods. The `Update()` method is then called in order to trigger the computation of the histogram.

```
typedef HistogramFilterType::HistogramMeasurementVectorType
HistogramMeasurementVectorType;

HistogramMeasurementVectorType binMinimum( 3 );
HistogramMeasurementVectorType binMaximum( 3 );

binMinimum[0] = -0.5;
binMinimum[1] = -0.5;
binMinimum[2] = -0.5;

binMaximum[0] = 255.5;
binMaximum[1] = 255.5;
binMaximum[2] = 255.5;

histogramFilter->SetHistogramBinMinimum( binMinimum );
histogramFilter->SetHistogramBinMaximum( binMaximum );

histogramFilter->Update();
```

The histogram can be recovered from the filter by creating a variable with the histogram type taken from the filter traits.

```
typedef HistogramFilterType::HistogramType HistogramType;

const HistogramType * histogram = histogramFilter->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 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 * std::log( probability ) / std::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

histogramFilter->SetHistogramSize( size );
histogramFilter->Update();
```

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 * std::log( probability ) / std::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

histogramFilter->SetHistogramSize( size );
histogramFilter->Update();

```

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 * std::log( probability ) / std::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 is 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.

5.4 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[19]) 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 5.3 illustrates this general framework.

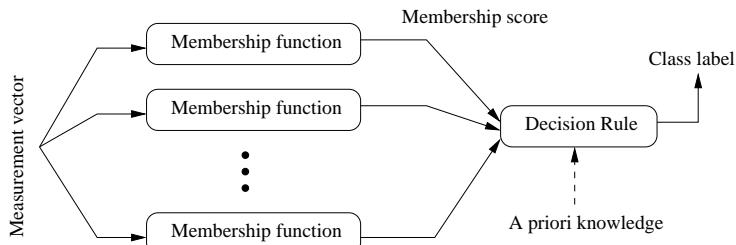


Figure 5.3: Simple conceptual classifier.

This framework closely follows that of Duda and Hart[19]. The classification process can be described as follows:

1. A measurement vector is input to each membership function.
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 formulate 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

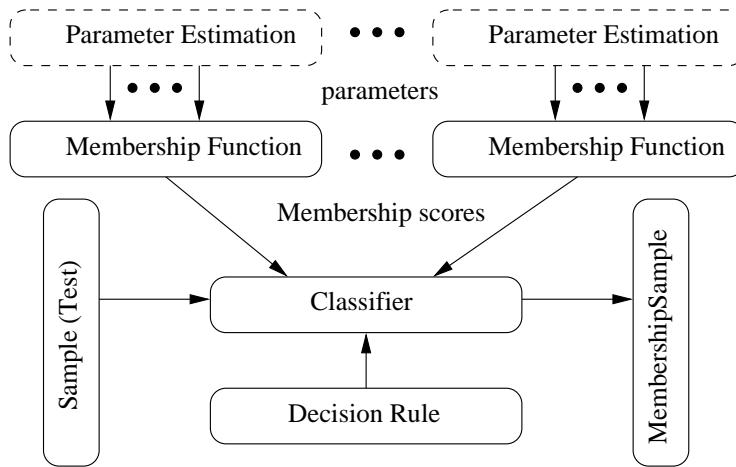


Figure 5.4: Statistical classification framework.

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.

5.4.1 k-d Tree Based k-Means Clustering

The source code for this section can be found in the file `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 criterion 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 vector 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] [45] [29]. Our implementation of this scheme follows the article by the Kanungo et al [29].

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::DistanceToCentroidMembershipFunction` function as the membership functions for each cluster and plug that—along with a sample—into an `itk::Statistics::SampleClassifierFilter` object to get a `itk::Statistics::MembershipSample` that stores pairs of measurement vectors and their associated class labels (k labels).

```
#include "itkMinimumDecisionRule.h"
#include "itkSampleClassifierFilter.h"
```

We will fill the sample with random variables from two normal distribution using the `itk::Statistics::NormalVariateGenerator`.

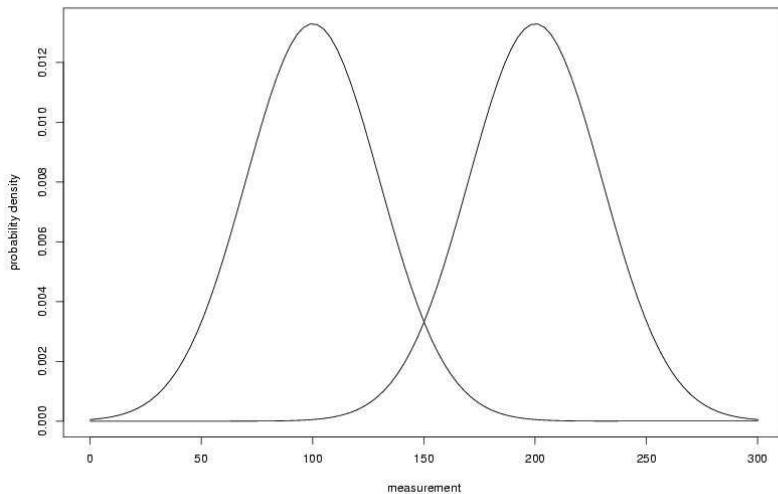


Figure 5.5: Two normal distributions' probability density plot (The means are 100 and 200, and the standard deviation is 30)

```
#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 5.5.

```

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 );
}

```

We create a k-d tree. To see the details on the k-d tree generation, see the Section [5.1.7](#).

```

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 DistanceToCentroidMembershipFunction 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` filter object and a `MinimumDecisionRule` object, we plug-in the `decisionRule` and the `sample` to the classifier filter. 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::DistanceToCentroidMembershipFunction<
    MeasurementVectorType > MembershipFunctionType;
typedef itk::Statistics::MinimumDecisionRule           DecisionRuleType;
DecisionRuleType::Pointer decisionRule = DecisionRuleType::New();

typedef itk::Statistics::SampleClassifierFilter< SampleType > ClassifierType;
ClassifierType::Pointer classifier = ClassifierType::New();

classifier->SetDecisionRule( decisionRule );
classifier->SetInput( sample );
classifier->SetNumberOfClasses( 2 );

typedef ClassifierType::ClassLabelVectorObjectType
    ClassLabelVectorObjectType;
typedef ClassifierType::ClassLabelVectorType ClassLabelVectorType;
typedef ClassifierType::ClassLabelText      ClassLabelText;

ClassLabelVectorObjectType::Pointer classLabelsObject =
    ClassLabelVectorObjectType::New();
ClassLabelVectorType& classLabelsVector = classLabelsObject->Get();

ClassLabelText class1 = 200;
classLabelsVector.push_back( class1 );
ClassLabelText class2 = 100;
classLabelsVector.push_back( class2 );

classifier->SetClassLabels( classLabelsObject );
```

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 `SetCentroid()` method. To plug-in two distance functions, we create a `MembershipFunctionVectorObject` that contains a `MembershipFunctionVector` with two components and add it using the `SetMembershipFunctions` method. Then invocation of the `Update()` method will perform the classification.

```

typedef ClassifierType::MembershipFunctionVectorObjectType
    MembershipFunctionVectorObjectType;
typedef ClassifierType::MembershipFunctionVectorType
    MembershipFunctionVectorType;

MembershipFunctionVectorObjectType::Pointer membershipFunctionVectorObject =
    MembershipFunctionVectorObjectType::New();
MembershipFunctionVectorType& membershipFunctionVector =
    membershipFunctionVectorObject->Get();

int index = 0;
for (unsigned int i = 0; i < 2; i++)
{
    MembershipFunctionType::Pointer membershipFunction
        = MembershipFunctionType::New();
    MembershipFunctionType::CentroidType centroid(
        sample->GetMeasurementVectorSize() );
    for (unsigned int j = 0; j < sample->GetMeasurementVectorSize(); j++)
    {
        centroid[j] = estimatedMeans[index++];
    }
    membershipFunction->SetCentroid( centroid );
    membershipFunctionVector.push_back( membershipFunction.GetPointer() );
}
classifier->SetMembershipFunctions( membershipFunctionVectorObject );

classifier->Update();

```

The following code snippet prints out the measurement vectors and their class labels in the sample.

```

const ClassifierType::MembershipSampleType* membershipSample =
    classifier->GetOutput();
ClassifierType::MembershipSampleType::ConstIterator iter
    = membershipSample->Begin();

while ( iter != membershipSample->End() )
{
    std::cout << "measurement vector = " << iter.GetMeasurementVector()
        << " class label = " << iter.GetClassLabel()
        << std::endl;
    ++iter;
}

```

5.4.2 K-Means Classification

The source code for this section can be found in the file `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 divided 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 `SetUseNonContiguousLabels()`.

```
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 ";
  std::cerr << " 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 5.6 illustrates the effect of this filter with three classes. The means were estimated by



Figure 5.6: Effect of the KMeans classifier on a T1 slice of the brain.

ScalarImageKmeansModelEstimator.cxx.

5.4.3 Bayesian Plug-In Classifier

The source code for this section can be found in the file
`BayesianPluginClassifier.cxx`.

In this example, we present a system that places measurement vectors into two Gaussian classes. The Figure 5.7 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 is that this classifier uses the `itk::Statistics::GaussianDensityFunctions` as membership functions instead of the `itk::Statistics::DistanceToCentroidMembershipFunction`. 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::CovarianceSampleFilter` (sample covariance) for the estimation algorithms of the two parameters. If we want a more robust estimation algorithm, we can replace this estimation algorithm 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,

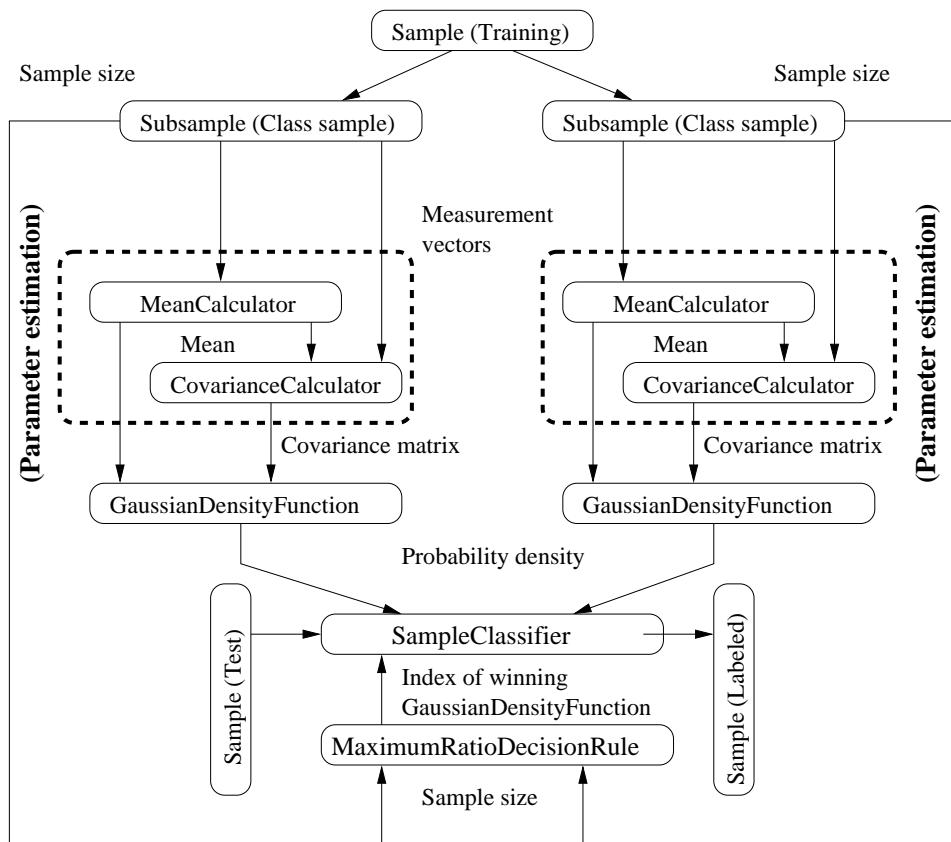


Figure 5.7: Bayesian plug-in classifier for two Gaussian classes.

we use the `itk::Statistics::Subsample` objects.

```
#include "itkVector.h"
#include "itkListSample.h"
#include "itkSubsample.h"
```

The following file provides us the parameter estimation algorithm.

```
#include "itkCovarianceSampleFilter.h"
```

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 "itkGaussianMembershipFunction.h"
#include "itkSampleClassifierFilter.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.

```
const unsigned int measurementVectorLength = 1;
typedef itk::Vector< double, measurementVectorLength > MeasurementVectorType;
typedef itk::Statistics::ListSample< MeasurementVectorType > SampleType;
SampleType::Pointer sample = SampleType::New();
// length of measurement vectors in the sample.
sample->SetMeasurementVectorSize( measurementVectorLength );

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 5.5.

```
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 `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::CovarianceSampleFilter< ClassSampleType >
CovarianceEstimatorType;

std::vector< CovarianceEstimatorType::Pointer > covarianceEstimators;

for ( unsigned int i = 0; i < 2; ++i )
{
    covarianceEstimators.push_back( CovarianceEstimatorType::New() );
    covarianceEstimators[i]->SetInput( classSamples[i] );
    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 : "
        << covarianceEstimators[i]->GetMean()
        << "    covariance matrix : "
        << covarianceEstimators[i]->GetCovarianceMatrix() << 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:

$$\text{Decide } \omega_i \text{ if } \frac{p(\vec{x}|\omega_i)}{p(\vec{x}|\omega_j)} > \frac{P(\omega_j)}{P(\omega_i)} \text{ for all } j \neq i \quad (5.4)$$

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::GaussianMembershipFunction< MeasurementVectorType >
    MembershipFunctionType;
typedef itk::Statistics::MaximumRatioDecisionRule DecisionRuleType;
DecisionRuleType::Pointer decisionRule = DecisionRuleType::New();

DecisionRuleType::PriorProbabilityVectorType aPrioris;
aPrioris.push_back( (double)classSamples[0]->GetTotalFrequency()
    / (double)sample->GetTotalFrequency() );
aPrioris.push_back( (double)classSamples[1]->GetTotalFrequency()
    / (double)sample->GetTotalFrequency() );
decisionRule->SetPriorProbabilities( aPrioris );

typedef itk::Statistics::SampleClassifierFilter< SampleType > ClassifierType;
ClassifierType::Pointer classifier = ClassifierType::New();

classifier->SetDecisionRule( decisionRule );
classifier->SetInput( sample );
classifier->SetNumberOfClasses( 2 );

typedef ClassifierType::ClassLabelVectorObjectType
    ClassLabelVectorObjectType;
typedef ClassifierType::ClassLabelVectorType ClassLabelVectorType;

ClassLabelVectorObjectType::Pointer classLabelVectorObject =
    ClassLabelVectorObjectType::New();
ClassLabelVectorType classLabelVector = classLabelVectorObject->Get();

ClassifierType::ClassLabelType class1 = 100;
classLabelVector.push_back( class1 );
ClassifierType::ClassLabelType class2 = 200;
classLabelVector.push_back( class2 );

classLabelVectorObject->Set( classLabelVector );
classifier->SetClassLabels( classLabelVectorObject );
```

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 Gaussian distribution functions. The distribution functions have two parameters, the mean, set by the `SetMean()` method, and the covariance, set by the `SetCovariance()` method. To plug-in two distribution functions, we create a new instance of `MembershipFunctionVectorObjectType` and populate its internal vector with new instances of `MembershipFunction` (i.e. `GaussianMembershipFunction`). This is done by calling the `Get()` method of `membershipFunctionVectorObject` to get the internal vector, populating this vector with two new membership functions and then calling `membershipFunctionVectorObject->Set(membershipFunctionVector)`. Finally, the invocation of the `Update()` method will perform the classification.

```

typedef ClassifierType::MembershipFunctionVectorObjectType
    MembershipFunctionVectorObjectType;
typedef ClassifierType::MembershipFunctionVectorType
    MembershipFunctionVectorType;

MembershipFunctionVectorObjectType::Pointer membershipFunctionVectorObject =
    MembershipFunctionVectorObjectType::New();
MembershipFunctionVectorType membershipFunctionVector =
    membershipFunctionVectorObject->Get();

for (unsigned int i = 0; i < 2; ++i)
{
    MembershipFunctionType::Pointer membershipFunction =
        MembershipFunctionType::New();
    membershipFunction->SetMean( covarianceEstimators[i]->GetMean() );
    membershipFunction->SetCovariance(
        covarianceEstimators[i]->GetCovarianceMatrix() );
    membershipFunctionVector.push_back( membershipFunction.GetPointer() );
}
membershipFunctionVectorObject->Set( membershipFunctionVector );
classifier->SetMembershipFunctions( membershipFunctionVectorObject );

classifier->Update();

```

The following code snippet prints out pairs of a measurement vector and its class label in the sample.

```

const ClassifierType::MembershipSampleType* membershipSample
    = classifier->GetOutput();
ClassifierType::MembershipSampleType::ConstIterator iter
    = membershipSample->Begin();

while ( iter != membershipSample->End() )
{
    std::cout << "measurement vector = " << iter.GetMeasurementVector()
        << " class label = " << iter.GetClassLabel() << std::endl;
    ++iter;
}

```

5.4.4 Expectation Maximization Mixture Model Estimation

The source code for this section can be found in the file
 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 5.4.3) 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 :

$$p(x) = \sum_{i=0}^c \alpha_i f_i(x) \quad (5.5)$$

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::is` 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 5.5.

```
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::cout << "    Parameters:" << std::endl;
    std::cout << "        " << (components[i])->GetFullParameters()
    << std::endl;
    std::cout << "    Proportion: ";
    std::cout << "        " << estimator->GetProportions() [i] << std::endl;
}

```

5.4.5 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 get 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 [6].

The source code for this section can be found in the file

`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 "itkImageFileReader.h"
#include "itkImageFileWriter.h"
#include "itkComposeImageFilter.h"

```

The following headers are related to the statistical classification classes.

```

#include "itkMRFImageFilter.h"
#include "itkDistanceToCentroidMembershipFunction.h"
#include "itkMinimumDecisionRule.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 [5.4.2](#).

```
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::ComposeImageFilter`. 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::ComposeImageFilter<
    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::Statistics::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;
MembershipFunctionType::CentroidType 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]);
}
if (numberOfClasses > 0)
{
    meanDistance /= numberOfClasses;
}
else
{
    std::cerr << "ERROR: numberOfClasses is 0" << std::endl;
    return EXIT_FAILURE;
}

```

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 );
```



Figure 5.8: Effect of the MRF filter on a T1 slice of the brain.

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 ";
std::cerr << " image file : " << argv[2] << std::endl;
std::cerr << excp << std::endl;
return EXIT_FAILURE;
}
```

Figure 5.8 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`.

BIBLIOGRAPHY

- [1] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Professional Computing Series. Addison-Wesley, 2001. [1.8.3](#), [1.10](#), [3.9.1](#)
- [2] K. Alsabti, S. Ranka, and V. Singh. An efficient k-means clustering algorithm. In *First Workshop on High-Performance Data Mining*, 1998. [5.4.1](#)
- [3] L. Alvarez and J.-M. Morel. *A Morphological Approach To Multiscale Analysis: From Principles to Equations*, pages 229–254. Kluwer Academic Publishers, 1994. [2.7.3](#)
- [4] ANSI-ISO. *Programming Languages - C++*. American National Standards Institutue, 1998. [1.9](#)
- [5] M. H. Austern. *Generic Programming and the STL*. Professional Computing Series. Addison-Wesley, 1999. [1.8.3](#), [1.10](#), [3.9.1](#)
- [6] J. Besag. On the statistical analysis of dirty pictures. *J. Royal Statist. Soc. B.*, 48:259–302, 1986. [5.4.5](#)
- [7] Eric Boix, Mathieu Malaterre, Benoit Reignan, and Jean-Pierre Roux. *The GDCM Library*. CNRS, INSERM, INSA Lyon, UCB Lyon, <http://www.creatis.insa-lyon.fr/Public/Gdcm/>. [1.12.1](#)
- [8] R. N. Bracewell. *The Fourier Transform and its Applications*. McGraw-Hill, 1999. [2.10.1](#)
- [9] R. N. Bracewell. *Fourier Analysis and Imaging*. Plenum US, 2004. [2.10.1](#)
- [10] 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. [3.12](#)
- [11] V. Caselles, R. Kimmel, and G. Sapiro. Geodesic active contours. *International Journal on Computer Vision*, 22(1):61–97, 1997. [4.3.3](#)

- [12] 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. [3.5](#)
- [13] P. E. Danielsson. Euclidean distance mapping. *Computer Graphics and Image Processing*, 14:227–248, 1980. [2.8](#)
- [14] C. Darwin. *On the Origin of Species*. <http://www.gutenberg.org>, sixth edition, 1999. [3.6](#)
- [15] M. H. Davis, A. Khotanzad, D. P. Flammig, and S. E. Harms. A physics-based coordinate transformation for 3-d image matching. *IEEE Transactions on Medical Imaging*, 16(3), June 1997. [3.9.18](#)
- [16] R. Deriche. Fast algorithms for low level vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(1):78–87, 1990. [2.4.2](#), [2.7.1](#), [2.7.1](#)
- [17] R. Deriche. Recursively implementing the gaussian and its derivatives. Technical Report 1893, Unite de recherche INRIA Sophia-Antipolis, avril 1993. Research Report. [2.4.2](#), [2.7.1](#), [2.7.1](#)
- [18] C. Dodson and T. Poston. *Tensor Geometry: The Geometric Viewpoint and its Uses*. Springer, 1997. [3.9.1](#), [9](#)
- [19] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern classification*. A Wiley-Interscience Publication, second edition, 2000. [5.2.3](#), [5.4](#), [5.4](#)
- [20] David Eberly. *Ridges in Image and Data Analysis*. Kluwer Academic Publishers, Dordrecht, 1996. [4.2.1](#)
- [21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995. [1.2](#), [3.4](#)
- [22] 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. [2.7.3](#)
- [23] Stephen Grossberg. Neural dynamics of brightness perception: Features, boundaries, diffusion, and resonance. *Perception and Psychophysics*, 36(5):428–456, 1984. [2.7.3](#)
- [24] J. Hajnal, D. J. Hawkes, and D. Hill. *Medical Image Registration*. CRC Press, 2001. [3.5](#), [3.11.4](#)
- [25] W. R. Hamilton. *Elements of Quaternions*. Chelsea Publishing Company, 1969. [3.6.4](#), [3.9.1](#), [3.9.11](#), [3.12](#)
- [26] A. Hendersen. *The Paraview Guide*. Kitware, Inc, 2004. [3.15](#)
- [27] B. K. Horn. Closed-form solution of absolute orientation using unit quaternions. *Journal of the Optical Society of America*, 4:629–642, April 1987. [3.17](#)

- [28] C. J. Joly. *A Manual of Quaternions*. MacMillan and Co. Limited, 1905. [3.6.4](#), [3.9.11](#)
- [29] 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. [5.1.7](#), [5.4.1](#)
- [30] J. Koënderink and A. van Doorn. The Structure of Two-Dimensional Scalar Fields with Applications to Vision. *Biol. Cybernetics*, 33:151–158, 1979. [4.2.1](#)
- [31] 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. [4.2.1](#)
- [32] L. Kohn, J. Corrigan, and M. Donaldson, editors. *To Err is Human: Building a safer health system*. National Academy Press, 2001. [1.12.4](#)
- [33] S. Kullback. *Information Theory and Statistics*. Dover Publications, 1997. [5.3.2](#)
- [34] 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. [4.3.7](#)
- [35] T. Lindeberg. *Scale-Space Theory in Computer Science*. Kluwer Academic Publishers, 1994. [2.7.1](#)
- [36] H. Lodish, A. Berk, S. Zipursky, P. Matsudaira, D. Baltimore, and J. Darnell. *Molecular Cell Biology*. W. H. Freeman and Company, 2000. [3.9.1](#)
- [37] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics*, 21(4):163–169, July 1987. [2.11.1](#)
- [38] 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. [3.5](#)
- [39] 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. [4.3.2](#)
- [40] 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. [3.9.17](#), [3.11.3](#)
- [41] 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. [3.5.1](#), [3.9.17](#)

- [42] E. H. Meijering, W. J. Niessen, J. P. Pluim, and M. A. Viergever. Quantitative comparison of 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. [3.10.4](#)
- [43] David R. Musser. Introspective sorting and selection algorithms. *Software—Practice and Experience*, 8:983–993, 1997. [5.2.3](#)
- [44] NEMA. The dicom standard. Technical report, NEMA, <http://medical.nema.org/>, 2013. [1.12.1](#)
- [45] 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. [5.4.1](#)
- [46] P. Perona and J. Malik. Scale-space and edge detection using anisotropic diffusion. *IEEE Transactions on Pattern Analysis Machine Intelligence*, 12:629–639, 1990. [2.7.3](#), [2.7.3](#), [2.7.3](#)
- [47] 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. [3.5](#), [3.11.3](#)
- [48] K. Popper. *Open Society and Its Enemies*. Princeton University Press, 1971. [3.5.1](#)
- [49] K. Popper. *The Logic of Scientific Discovery*. Routledge, 2002. [3.5.1](#), [5.3.1](#)
- [50] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C*. Cambridge University Press, second edition, 1992. [3.12](#)
- [51] 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-Pkrope, editors, *Information Processing in Medical Imaging 1999 (IPMI'99)*, pages 252–265. Springer, 1999. [3.9.18](#)
- [52] 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. [3.9.18](#), [3.17](#)
- [53] D. Rueckert, L. I. Sonoda, C. Hayes, D. L. G. Hill, M. O. Leach, and D. J. Hawkes. Nonrigid registration using free-form deformations: Application to breast mr images. *IEEE Transaction on Medical Imaging*, 18(8):712–721, 1999. [3.9.17](#)
- [54] G. Sapiro and D. Ringach. Anisotropic diffusion of multivalued images with applications to color filtering. *IEEE Trans. on Image Processing*, 5:1582–1586, 1996. [2.7.3](#)
- [55] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit, An Object Oriented Approach to 3D Graphics*. Kitware Inc, 1998. [2.11.1](#)

- [56] J. P. Serra. *Image Analysis and Mathematical Morphology*. Academic Press Inc., 1982. [2.6.3](#), [4.2.1](#)
- [57] J.A. Sethian. *Level Set Methods and Fast Marching Methods*. Cambridge University Press, 1996. [4.3](#)
- [58] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, July 1948. [2.9.4](#), [5.3.2](#)
- [59] C. E. Shannon and W. Weaver. *The Mathematical Theory of Communication*. University of Illinois Press, 1948. [2.9.4](#), [5.3.2](#)
- [60] 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. [3.12](#)
- [61] Baart M. ter Haar Romeny, editor. *Geometry-Driven Diffusion in Computer Vision*. Kluwer Academic Publishers, 1994. [2.7.3](#)
- [62] J. P. Thirion. Fast non-rigid matching of 3D medical image. Technical report, Research Report RR-2547, Epidure Project, INRIA Sophia, May 1995. [3.14](#)
- [63] J.-P. Thirion. Image matching as a diffusion process: an analogy with maxwell’s demons. *Medical Image Analysis*, 2(3):243–260, 1998. [3.14](#)
- [64] P. Viola and W. M. Wells III. Alignment by maximization of mutual information. *IJCV*, 24(2):137–154, 1997. [3.5](#)
- [65] 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. [2.7.3](#)
- [66] R. T. Whitaker and G. Gerig. *Vector-Valued Diffusion*, pages 93–134. Kluwer Academic Publishers, 1994. [2.7.3](#), [2.7.3](#)
- [67] 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. [2.7.3](#)
- [68] 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. [2.7.3](#)
- [69] Ross T. Whitaker. *Geometry-Limited Diffusion*. PhD thesis, The University of North Carolina, Chapel Hill, North Carolina 27599-3175, 1993. [2.7.3](#), [2.7.3](#)
- [70] 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. [2.7.3](#)

- [71] 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. [2.7.3](#)
- [72] 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. [2.7.3](#)
- [73] 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. [2.7.3](#)
- [74] 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. [4.2.1](#)
- [75] 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. [4.2.1](#), [4.2.1](#)
- [76] C. Zhu, R. H. Byrd, 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. [3.12](#)

INDEX

- Amount of information
 - Image, [476](#)
- Anisotropic data sets, [156](#)
- BinaryMask3DMeshSource
 - Header, [166](#)
 - Instantiation, [167](#)
 - SetInput, [167](#)
- BSplineInterpolateImageFunction, [264](#)
- BSplineTransform, [303](#)
 - Instantiation, [292, 298, 301, 304](#)
 - New, [292, 298, 301, 304](#)
- Casting Images, [60](#)
- CenteredTransformInitializer
 - GeometryOn(), [215](#)
 - MomentsOn(), [215](#)
- CenteredTransformInitializer
 - GeometryOn(), [206](#)
 - MomentsOn(), [206](#)
- Complex images
 - Instantiation, [20](#)
 - Reading, [20](#)
 - Writing, [20](#)
- CreateStructuringElement()
 - itk::BinaryBallStructuringElement, [84, 87](#)
- DICOM, [30](#)
- Changing Headers, [48](#)
- Dictionary, [41](#)
- GDCM, [41](#)
- Header, [41, 44](#)
- Introduction, [30](#)
- Printing Tags, [41, 44](#)
- Series, [30](#)
- Standard, [30](#)
- Tags, [41, 44](#)
- Dicom
 - HIPPA, [38](#)
- Distance Map
 - itk::SignedDanielssonDistanceMap-
 - ImageFilter,
 - [129](#)
- EllipseSpatialObject
 - Instantiation, [325](#)
- Entropy
 - Images, [476](#)
 - What's wrong in images, [475](#)
- GDCM
 - Dictionary, [41](#)
- GDCMImageIO
 - header, [41](#)
- GDCMSeriesFileNames
 - GetOutputFileNames(), [40](#)
 - SetOutputDirectory(), [40](#)

GetMetaDataDictionary()
 ImageIOBase, 42

GroupSpatialObject
 Instantiation, 325

HIPAA
 Dicom, 38
 Privacy, 38

Image
 Amount of information, 476
 Entropy, 476

Image Series
 Reading, 23
 Writing, 23

ImageToSpatialObjectMetric
 GetValue(), 325

ImageFileRead
 Vector images, 18

ImageFileWriter
 Vector images, 16

ImageIO
 GetMetaDataDictionary(), 46

ImageIOBase
 GetMetaDataDictionary(), 42

ImageSeriesWriter
 SetFileNames(), 40

Isosurface extraction
 Mesh, 166

itk::AddImageFilter
 Instantiation, 76

itk::AffineTransform, 219, 259
 Composition, 146
 header, 132, 219
 Image Registration, 231
 Instantiation, 220, 233
 instantiation, 133, 148
 New(), 148, 220
 Pointer, 148, 220
 resampling, 147
 Rotate2D(), 146, 148
 SetIdentity(), 138
 Translate(), 135, 146, 148, 150

itk::AmoebaOptimizer, 278

itk::ANTSNeighborhoodCorrelationImageToImageMetricv4,
 277

itk::BilateralImageFilter, 116
 header, 116
 instantiation, 116
 New(), 116
 Pointer, 116
 SetDomainSigma(), 117
 SetRangeSigma(), 117

itk::BinaryThresholdImageFilter
 Header, 51
 Instantiation, 51
 SetInput(), 52
 SetInsideValue(), 53
 SetOutsideValue(), 53

itk::BinaryBallStructuringElement
 CreateStructuringElement(), 84, 87
 SetRadius(), 84, 87

itk::BinaryDilateImageFilter
 header, 83
 New(), 84
 Pointer, 84
 SetDilateValue(), 85
 SetKernel(), 84
 Update(), 85

itk::BinaryErodeImageFilter
 header, 83
 New(), 84
 Pointer, 84
 SetErodeValue(), 85
 SetKernel(), 84
 Update(), 85

itk::BinaryMedianImageFilter, 89
 GetOutput(), 89
 header, 89
 instantiation, 89
 Neighborhood, 89
 New(), 89
 Pointer, 89
 Radius, 89
 SetInput(), 89

itk::BinomialBlurImageFilter, 99

- itk::BinomialBlurImageFilter
header, 99
instantiation, 99
New(), 99
Pointer, 99
SetInput(), 100
SetRepetitions(), 100
Update(), 100
- itk::BSplineDeformableTransform, 261
- itk::BSplineInterpolateImageFunction, 265
- itk::BSplineTransform, 292, 297, 301
DeformableRegistration, 292, 297, 301
header, 292, 297, 301
- itk::BSplineTransformParametersAdaptor, 297
- itk::CannySegmentationLevelSetImageFilter, 395
GenerateSpeedImage(), 399
GetSpeedImage(), 399
SetAdvectionScaling(), 397
- itk::CannyEdgeDetectionImageFilter, 57
header, 57
- itk::CastImageFilter, 60
header, 60
New(), 60
Pointer, 60
SetInput(), 60
Update(), 61
- itk::CenteredRigid2DTransform, 252
- itk::CenteredTransformInitializer
header, 214
In 3D, 214
Instantiation, 215
New(), 215
SmartPointer, 215
- itk::CenteredRigid2DTransform, 197, 205
header, 197, 205
Instantiation, 197, 205, 206
New(), 198, 206
Pointer, 198, 205
SmartPointer, 206
- itk::CenteredSimilarity2DTransform, 210
header, 210
- Instantiation, 210
Pointer, 211
SetAngle(), 211
SetScale(), 211
- itk::ChangeInformationImageFilter
CenterImageOn(), 407
- itk::ComplexToRealImageFilter, 162
- itk::ConfidenceConnectedImageFilter, 354
header, 354
SetInitialNeighborhoodRadius(), 356
SetMultiplier(), 355
SetNumberOfIterations(), 356
SetReplaceValue(), 356
SetSeed(), 356
- itk::ConjugateGradientLineSearch-
Optimizerv4, 278
- itk::ConnectedThresholdImageFilter, 344
header, 344
SetLower(), 345
SetReplaceValue(), 345
SetSeed(), 345
SetUpper(), 345
- itk::CorrelationImageToImageMetricv4, 274
- itk::CovariantVector
Concept, 244
- itk::CurvatureAnisotropicDiffusionImage-
Filter, 108
header, 108
instantiation, 108
New(), 108
Pointer, 108
SetConductanceParameter(), 109
SetNumberOfIterations(), 109
SetTimeStep(), 109
Update(), 109
- itk::CurvatureFlowImageFilter, 110
header, 110
instantiation, 110
New(), 111
Pointer, 111
SetNumberOfIterations(), 111

- SetTimeStep(), 111
- Update(), 111
- itk::DanielssonDistanceMapImageFilter
 - GetOutput(), 127
 - GetVoronoiMap(), 127
 - Header, 127
 - Instantiation, 127
 - instantiation, 127
 - New(), 127
 - Pointer, 127
 - SetInput(), 127
- itk::DataObjectDecorator
 - Get(), 178
 - Use in Registration, 178
- itk::DemonsImageToImageMetricv4, 276
- itk::DemonsRegistrationFilter, 309
 - SetFixedImage(), 309
 - SetMovingImage(), 309
 - SetNumberOfIterations(), 309
 - SetStandardDeviations(), 309
- itk::DerivativeImageFilter, 68
 - GetOutput(), 69
 - header, 68
 - instantiation, 69
 - New(), 69
 - Pointer, 69
 - SetDirection(), 69
 - SetInput(), 69
 - SetOrder(), 69
- itk::DiscreteGaussianImageFilter, 97
 - header, 97
 - instantiation, 98
 - New(), 98
 - Pointer, 98
 - SetMaximumKernelWidth(), 98
 - SetVariance(), 98
 - Update(), 98
- itk::ElasticBodyReciprocalSplineKernelTransform, 262
- itk::ElasticBodySplineKernelTransform, 262
- itk::EllipseSpatialObject
 - header, 322
- SetRadius(), 326
- itk::Euler2DTransform, 251
- itk::Euler3DTransform, 257
- itk::EventObject
 - CheckEvent, 186
- itk::ExhaustiveOptimizerv4, 278
- itk::ExtractImageFilter
 - header, 12
 - SetExtractionRegion(), 14
- itk::FastMarchingImageFilter
 - Multiple seeds, 376, 383
 - NodeContainer, 376, 384
 - Nodes, 376, 384
 - NodeType, 376, 384
 - Seed initialization, 376, 384
 - SetStoppingValue(), 377
 - SetTrialPoints(), 377, 384
- itk::FFTWForwardFFTImageFilter, 161, 164
- itk::FileImageReader
 - GetOutput(), 52, 56, 348
- itk::FlipImageFilter, 130
 - GetOutput(), 131
 - header, 131
 - instantiation, 131
 - Neighborhood, 131
 - New(), 131
 - Pointer, 131
 - Radius, 131
 - SetInput(), 131
- itk::FloodFillIterator
 - In Region Growing, 344, 354
- itk::ForwardFFTImageFilter, 161, 164
- itk::GDCMImageIO
 - header, 34
- itk::GDCMSeriesFileNames
 - GetFileNames(), 36
 - header, 34
 - SetDirectory(), 36
- itk::GeodesicActiveContourLevelSetImageFilter
 - SetAdvectionScaling(), 389
 - SetCurvatureScaling(), 389
 - SetPropagationScaling(), 389

itk::GeodesicActiveContourShapePrior-
LevelSetImageFilter
Monitoring, 405
SetAdvectionScaling(), 407
SetCurvatureScaling(), 407
SetPropagationScaling(), 407
itk::GradientAnisotropicDiffusionImage-
Filter,
106
header, 106
instantiation, 106
New(), 106
Pointer, 106
SetConductanceParameter(), 106
SetNumberOfIterations(), 106
SetTimeStep(), 106
Update(), 106
itk::GradientDescentLineSearch-
Optimizerv4,
278
itk::GradientDescentOptimizerv4, 278
itk::GradientDescentOptimizerv4Template
GetCurrentIteration(), 176
SetLearningRate(), 175
SetMinimumStepLength(), 175
SetNumberOfIterations(), 175
SetRelaxationFactor(), 175
itk::GradientMagnitudeRecursiveGaussian-
ImageFilter,
66
header, 67
Instantiation, 67
New(), 67
Pointer, 67
SetSigma(), 67, 375, 383
Update(), 67
itk::GradientRecursiveGaussianImageFilter
header, 16
itk::GradientMagnitudeImageFilter, 64
header, 64
instantiation, 65
New(), 65
Pointer, 65
Update(), 65
itk::GrayscaleDilateImageFilter
header, 86
New(), 87
Pointer, 87
SetKernel(), 87
Update(), 87
itk::GrayscaleErodeImageFilter
header, 86
New(), 87
Pointer, 87
SetKernel(), 87
Update(), 87
itk::GroupSpatialObject
header, 322
New(), 327
Pointer, 327
itk::HistogramMatchingImageFilter, 312
SetNumberOfHistogramLevels(), 313
SetNumberOfMatchPoints(), 313
SetInput(), 312
SetReferenceImage(), 312
SetSourceImage(), 312
ThresholdAtMeanIntensityOn(), 313
itk::HistogramMatchingImageFilter, 294,
308
SetInput(), 295, 309
SetNumberOfHistogramLevels(), 295,
309
SetNumberOfMatchPoints(), 295, 309
SetReferenceImage(), 295, 309
SetSourceImage(), 295, 309
ThresholdAtMeanIntensityOn(), 295,
309
itk::IdentityTransform, 248
itk::Image
Header, 171
Instantiation, 171
itk::ImageRegistrationMethod
Multi-Modality, 189
itk::ImageRegistrationMethodv4
SetMovingInitialTransform(), 232
SetNumberOfLevels(), 227

SetShrinkFactorsPerLevel(), 227
 SetSmoothingSigmasPerLevel(), 227
 itk::ImageToImageMetricv4, 269
 GetDerivatives(), 269
 GetValue(), 269
 GetValueAndDerivatives(), 269
 itk::ImageToSpatialObjectMetric
 header, 323
 Instantiation, 328
 itk::ImageToSpatialObjectRegistration-
 Method
 Instantiation, 328
 New(), 328
 Pointer, 328
 SetFixedImage(), 329
 SetInterpolator(), 329
 SetMetric(), 329
 SetMovingSpatialObject(), 329
 SetOptimizer(), 329
 SetTransform(), 329
 Update(), 330
 itk::ImageFileRead
 Complex images, 20
 Vector images, 14, 22
 itk::ImageFileReader, 1
 header, 1, 6, 8, 25
 Instantiation, 2, 6, 9
 New(), 2, 6, 9, 11, 13, 17, 19, 23
 RGB Image, 8
 SetFileName(), 2, 6, 9, 11, 13, 17, 19,
 23
 SmartPointer, 2, 6, 9, 11, 13, 17, 19, 23
 itk::ImageFileWrite
 Complex images, 20
 Vector images, 14
 itk::ImageFileWriter, 1
 header, 1, 6, 8, 34
 Instantiation, 2, 6, 9, 24
 New(), 2, 6, 9, 11, 13, 17, 19, 23
 RGB Image, 8, 28
 SetFileName(), 2, 6, 9, 11, 13, 17, 19,
 23
 SetImageIO(), 7
 SmartPointer, 2, 6, 9, 11, 13, 17, 19, 23
 UseInputMetaDataDictionaryOff(), 34
 itk::ImageMaskSpatialObject
 header, 282
 Instantiation, 282
 New, 282
 Pointer, 282
 SetImage(), 283
 itk::ImageMomentsCalculator, 205
 itk::ImageRegistrationMethod
 DataObjectDecorator, 178
 GetOutput(), 178
 Monitoring, 185
 Pipeline, 178
 Resampling image, 178
 itk::ImageRegistrationMethodv4
 AffineTransform, 231
 GetTransform(), 176
 InPlaceOn(), 301
 Multi-Modality, 189, 224, 231, 281
 Multi-Resolution, 224, 231
 Multi-Stage, 231, 239
 Scaling parameter space, 231
 SetInitialTransform(), 301
 itk::ImageSeriesReader
 GetMetaDataDictionaryArray(), 40
 header, 24, 34
 Instantiation, 24
 RGB Image, 28
 SetFileNames(), 37
 itk::ImageSeriesWriter
 header, 25
 SetMetaDataDictionaryArray(), 40
 itk::ImageToImageMetricv4
 SetFixedImageMask(), 283
 itk::InterpolateImageFunction, 265
 Evaluate(), 265
 EvaluateAtContinuousIndex(), 265
 IsInsideBuffer(), 265
 SetInputImage(), 265
 itk::IsolatedConnectedImageFilter
 AddSeed1(), 359
 AddSeed2(), 359

- GetIsolatedValue(), 360
- header, 359
- SetLower(), 359
- SetReplaceValue(), 360
- itk::KernelTransforms, 262
- itk::LaplacianSegmentationLevelSetImage-
 - Filter,
399
 - SetPropagationScaling(), 401
- itk::LaplacianRecursiveGaussianImageFilter,
 - 77
 - header, 77
 - New(), 78
 - Pointer, 78
 - SetSigma(), 78
 - Update(), 78
- itk::LBFGSOptimizerv4, 278
- itk::LBFGSBOptimizerv4, 278
- itk::LBFGSBOptimizerv4, 301
 - header, 301
- itk::LBFGSOptimizer
 - header, 297
- itk::LBFGSOptimizerv4, 292, 297
 - header, 292
- itk::LevelSetMotionRegistrationFilter, 295
 - SetFixedImage(), 295
 - SetMovingImage(), 295
 - SetNumberOfIterations(), 295
 - SetStandardDeviations(), 295
- itk::LinearInterpolateImageFunction, 265
 - header, 323
- itk::MaskImageFilter, 164
- itk::MattesMutualInformationImageTo-
 - ImageMetricv4,
276
 - SetNumberOfHistogramBins(), 189,
276
- itk::MeanSquaresImageToImageMetricv4,
 - 271
- itk::MeanImageFilter, 80
 - GetOutput(), 80
 - header, 80
 - instantiation, 80
- Neighborhood, 80
- New(), 80
- Pointer, 80
- Radius, 80
- SetInput(), 80
- itk::MeanSquaresImageToImageMetricv4
 - SetFixedInterpolator(), 172
 - SetMovingInterpolator(), 172
- itk::MedianImageFilter, 81
 - GetOutput(), 82
 - header, 81
 - instantiation, 82
 - Neighborhood, 82
 - New(), 82
 - Pointer, 82
 - Radius, 82
 - SetInput(), 82
- itk::MinMaxCurvatureFlowImageFilter, 113
 - header, 113
 - instantiation, 113
 - New(), 113
 - Pointer, 113
 - SetNumberOfIterations(), 114
 - SetTimeStep(), 114
 - Update(), 114
- itk::MultiGradientOptimizerv4, 278
- itk::MutualInformationImageToImage-
 - Metricv4
 - Trade-offs, 191
- itk::NearestNeighborInterpolateImage-
 - Function,
265
 - header, 132
 - instantiation, 133
- itk::NeighborhoodConnectedImageFilter
 - SetLower(), 352
 - SetReplaceValue(), 352
 - SetSeed(), 352
 - SetUpper(), 352
- itk::NormalizeImageFilter, 60
 - header, 60
 - New(), 60
 - Pointer, 60

SetInput(), 60
 Update(), 61
itk::NormalVariateGenerator
 Initialize(), 281, 329
 New(), 281, 329
 Pointer, 281, 329
itk::NumericSeriesFileNames
 header, 24
itk::ObjectToObjectOptimizer, 278
 GetCurrentPosition(), 278
 SetMetric(), 278
 SetScales(), 278
 SetScalesEstimator(), 278
 StartOptimization(), 278
itk::OnePlusOneEvolutionaryOptimizer
 Instantiation, 328
itk::OnePlusOneEvolutionaryOptimizerv4,
 278
itk::OnePlusOneEvolutionaryOptimizer
 Initialize(), 410
 SetEpsilon(), 410
 SetMaximumIteration(), 410
 SetNormalVariateGenerator(), 410
 SetScales(), 410
itk::OnePlusOneEvolutionaryOptimizerv4
 Multi-Modality, 281
itk::Optimizer
 MaximizeOff(), 330
 MaximizeOn(), 330
itk::OtsuThresholdImageFilter
 SetInput(), 348
 SetInsideValue(), 348
 SetOutsideValue(), 348
itk::OtsuMultipleThresholdsCalculator
 GetOutput(), 350
itk::PCAShapeSignedDistanceFunction
 New(), 408
 SetPrincipalComponentStandard-
 Deviations(),
 409
 SetMeanImage(), 408
 SetNumberOfPrincipalComponents(),
 408
 SetPrincipalComponentsImages(), 408
 SetTransform(), 409
itk::Point
 Concept, 244
itk::PowellOptimizerv4, 278
itk::QuasiNewtonOptimizerv4, 278
itk::QuaternionRigidTransform, 255
itk::RecursiveGaussianImageFilter, 74, 101
 header, 74, 101
 Instantiation, 74, 78, 101
 New(), 74, 101
 Pointer, 74, 101
 SetSigma(), 76, 102
 Update(), 102
itk::RegionOfInterestImageFilter
 header, 10
 SetRegionOfInterest(), 11
itk::RegistrationMethod
 SetTransform(), 298
itk::RegistrationMethodv4
 GetCurrentIteration(), 221
 GetValue(), 221
 SetFixedImage(), 172
 SetMetric(), 172
 SetMovingImage(), 172
 SetMovingInitialTransform(), 173
 SetOptimizer(), 172
 SetTransform(), 220
itk::RegularStepGradientDescentOptimizer
 SetRelaxationFactor(), 190
itk::RegularStepGradientDescent-
 Optimizerv4,
 278
itk::ResampleImageFilter, 132
 GetOutput(), 134
 header, 132
 Image internal transform, 138
 instantiation, 133
 New(), 133
 Pointer, 133
 SetDefaultPixelValue(), 133, 137, 138,
 144
 SetInput(), 134

- SetInterpolator(), 133
- SetOutputOrigin(), 133, 138, 139, 142, 146
- SetOutputSpacing(), 133, 138, 141, 142, 146
- SetSize(), 134, 138, 141, 142, 146
- SetTransform(), 133, 138
- itk::RescaleIntensityImageFilter, 60
 - header, 8, 60
 - New(), 60
 - Pointer, 60
 - SetInput(), 60
 - SetOutputMaximum(), 9, 61
 - SetOutputMinimum(), 9, 61
 - Update(), 61
- itk::RGBPixel
 - header, 472
 - Image, 7, 28
 - Instantiation, 8, 28
 - Statistics, 472
- itk::Rigid3DPerspectiveTransform, 259
- itk::Sample
 - Histogram, 429
 - Interfaces, 422
 - PointSetToListSampleAdaptor, 428
- itk::ScaleLogarithmicTransform, 251
- itk::ScaleTransform, 249
- itk::SegmentationLevelSetImageFilter
 - SetAdvectionScaling(), 389
 - SetCurvatureScaling(), 385, 389, 394
 - SetMaximumRMSError(), 385
 - SetNumberOfIterations(), 385
 - SetPropagationScaling(), 385, 389, 394, 401
- itk::SegmentationLevelSetImageFilter
 - GenerateSpeedImage(), 399
 - GetSpeedImage(), 399
 - SetAdvectionScaling(), 397
- itk::ShapeDetectionLevelSetImageFilter
 - SetCurvatureScaling(), 385
 - SetMaximumRMSError(), 385
 - SetNumberOfIterations(), 385
- itk::ShapeDetectionLevelSetImageFilter
 - SetPropagationScaling(), 385
- itk::ShapePriorSegmentationLevelSetImageFilter
 - SetPropagationScaling(), 385
- itk::ShapePriorSegmentationLevelSetImageFilter
 - Filter
 - Monitoring, 405
 - SetAdvectionScaling(), 407
 - SetCurvatureScaling(), 407
 - SetPropagationScaling(), 407
- itk::ShapePriorMAPCostFunction
 - SetShapeParameterMeans(), 410
 - SetShapeParameterStandardDeviations(), 410
 - SetWeights(), 409
- itk::ShapeSignedDistanceFunction
 - SetTransform(), 409
- itk::ShiftScaleImageFilter, 60
 - header, 60
 - New(), 60
 - Pointer, 60
 - SetInput(), 60
 - SetScale(), 61
 - SetShift(), 61
 - Update(), 61
- itk::SigmoidImageFilter
 - GetOutput(), 63
 - header, 62
 - instantiation, 62
 - New(), 62
 - Pointer, 62
 - SetAlpha(), 62
 - SetBeta(), 62
 - SetInput(), 63
 - SetOutputMaximum(), 62
 - SetOutputMinimum(), 62
- itk::SigmoidImageFilter, 62
- itk::SignedDanielssonDistanceMapImageFilter
 - Header, 129
 - Instantiation, 129
- itk::Similarity2DTransform, 254
 - header, 149
 - instantiation, 150
 - New(), 150

Pointer, 150
 SetAngle(), 150
 SetRotationCenter(), 150
 SetScale(), 150
 itk::Similarity3DTransform, 258
 itk::SingleValuedNonLinearVnlOptimizerv4,
 278
 itk::SpatialObjectToImageFilter
 header, 322
 Instantiation, 327
 itk::SpatialObjectToImageFilter
 New(), 327
 Pointer, 327
 SetInput(), 327
 SetSize(), 327
 Update(), 327
 itk::Statistics
 Color Images, 468
 itk::Statistics::CovarianceSampleFilter, 446
 itk::Statistics::EuclideanDistanceMetric, 459
 itk::Statistics::ExpectationMaximization-
 MixtureModelEstimator,
 500
 itk::Statistics::GaussianMixtureModel-
 Component,
 500
 itk::Statistics::GaussianMembershipFunction,
 458, 494
 itk::Statistics::HeapSort, 455
 itk::Statistics::Histogram
 GetFrequency(), 470
 Iterators, 468
 Size(), 470
 itk::Statistics::ImageToListAdaptor, 424
 itk::Statistics::ImageToHistogramFilter
 header, 468, 472
 Update(), 470
 itk::Statistics::ImageToHistogramFilter
 GetOutput(), 470
 itk::Statistics::InsertSort, 455
 itk::Statistics::IntrospectiveSort, 455
 itk::Statistics::JointDomainImageToList-
 Adaptor,

 424
 itk::Statistics::KdTree, 440
 itk::Statistics::KdTreeBasedKmeans-
 Estimator,
 485
 itk::Statistics::KdTreeGenerator, 440
 itk::Statistics::ListSampleToHistogramFilter,
 438, 451
 itk::Statistics::ListSampleToHistogram-
 Generator,
 438
 header, 464
 itk::Statistics::ListSample, 422
 itk::Statistics::MaximumDecisionRule, 461
 itk::Statistics::MaximumRatioDecisionRule,
 462
 itk::Statistics::MeanCalculator, 446
 itk::Statistics::MembershipSampleGenerator,
 438
 itk::Statistics::MembershipSample, 435
 itk::Statistics::MinimumDecisionRule, 461
 itk::Statistics::NeighborhoodSampler, 438
 itk::Statistics::NeighborhoodSampler, 453
 itk::Statistics::NormalVariateGenerator, 463,
 494
 Initialize(), 410
 itk::Statistics::PointSetToListSampleAdaptor,
 426
 itk::Statistics::QuickSelect, 455
 itk::Statistics::SampleToHistogramFilter
 instantiation, 465
 itk::Statistics::SampleToHistogram-
 ProjectionFilter,
 438
 itk::Statistics::SampleClassifier, 494
 itk::Statistics::ScalarImageToHistogram-
 Generator
 Compute(), 467
 header, 467
 itk::Statistics::ScalarImageToListAdaptor,
 424
 header, 464
 instantiation, 465

- itk::Statistics::SelectiveSubsampleGenerator,
438
- itk::Statistics::Subsample, 432, 455
- itk::Statistics::WeightedCentroidKdTree-
Generator,
440
- itk::Statistics::WeightedCovariance-
Calculator,
448
- itk::Statistics::WeightedMeanCalculator, 448
- itk::SymmetricForcesDemonsRegistration-
Filter,
313
- SetFixedImage(), 313
 SetMovingImage(), 313
 SetNumberOfIterations(), 313
 SetStandardDeviations(), 313
- itk::ThinPlateR2LogRSplineKernel-
Transform,
262
- itk::ThinPlateSplineKernelTransform, 262
- itk::ThresholdSegmentationLevelSetImage-
Filter,
390
- SetCurvatureScaling(), 394
 SetPropagationScaling(), 394
- itk::ThresholdImageFilter
- Header, 55
 Instantiation, 55
 SetInput(), 56
 SetOutsideValue(), 56
 ThresholdAbove(), 55
 ThresholdBelow(), 55, 56
 ThresholdOutside(), 55
- itk::Transform, 244
- GetJacobian(), 247
 SetParameters(), 247
 TransformCovariantVector(), 244
 TransformPoint(), 244
 TransformVector(), 244
- itk::TranslationTransform, 248
- GetNumberOfParameters(), 173
 Instantiation, 231
- New(), 232
 Pointer, 232
- itk::Vector
- Concept, 244
- itk::VectorConfidenceConnectedImageFilter
- SetInitialNeighborhoodRadius(), 362
 SetMultiplier(), 362
 SetNumberOfIterations(), 362
 SetReplaceValue(), 362
 SetSeed(), 362
- itk::VectorCurvatureAnisotropicDiffusion-
ImageFilter,
120
- header, 120, 123
 instantiation, 120, 124
 New(), 120, 124
 Pointer, 120, 124
 RGB Images, 123
 SetNumberOfIterations(), 121, 125
 SetTimeStep(), 121, 125
 Update(), 121, 125
- itk::VectorGradientAnisotropicDiffusion-
ImageFilter,
118
- header, 118, 122
 instantiation, 119, 122
 New(), 119, 122
 Pointer, 119, 122
 RGB Images, 122
 SetNumberOfIterations(), 119, 123
 SetTimeStep(), 119, 123
 Update(), 119, 123
- itk::VectorIndexSelectionCastImageFilter
- header, 22
 Instantiation, 22
 New(), 22
 Pointer, 22
 SetIndex(), 22
- itk::VectorCastImageFilter
- instantiation, 123, 125
 New(), 123, 125
 Pointer, 123, 125
- itk::Versor

Definition, 256
itk::VesorRigid3DTransform, 214
 header, 214
 Instantiation, 215
 Pointer, 215
itk::VesorRigid3DTransform, 256
itk::VesorTransform, 256
itk::VesorTransformOptimizer, 256
itk::VnlForwardFFTImageFilter, 161, 164
itk::VolumeSplineKernelTransform, 262
itk::VotingBinaryHoleFillingImageFilter, 90
 GetOutput(), 92
 header, 90
 instantiation, 92
 Neighborhood, 92
 New(), 92
 Pointer, 92
 Radius, 92
 SetBackgroundValue(), 92
 SetForegroundValue(), 92
 SetInput(), 92
 SetMajorityThreshold(), 92
itk::VotingBinaryIterativeHoleFillingImage-
 Filter,
 94
 GetOutput(), 95
 header, 94
 instantiation, 94
 Neighborhood, 94
 New(), 94
 Pointer, 94
 Radius, 94
 SetBackgroundValue(), 95
 SetForegroundValue(), 95
 SetInput(), 95
 SetMajorityThreshold(), 95
 SetMaximumNumberOfIterations(), 95
itk::VTKImageIO
 header, 6
 Instantiation, 6
 New(), 6
 SetFileTypeToASCII(), 7
 SmartPointer, 6
itk::WarpImageFilter
 SetInterpolator(), 152
itk::WarpImageFilter, 151, 296, 310, 313
 SetDisplacementField(), 296, 310, 314
 SetInput(), 296, 310, 313
 SetInterpolator(), 296, 310, 313
 SetOutputOrigin(), 296, 310, 313
 SetOutputSpacing(), 296, 310, 313
itk::WindowedSincInterpolateImage-
 Function,
 266
itksys
 MakeDirectory, 40
 SystemTools, 40
Joint Entropy
 Statistics, 479
Joint Histogram
 Statistics, 479
LandmarkDisplacementFieldSource, 302
LaplacianRecursiveGaussianImageFilter
 SetNormalizeAcrossScale(), 78
LinearInterpolateImageFunction, 264
MakeDirectory
 itksys, 40
 SystemTools, 40
Marching Cubes, 166
Medical Errors, 38
Mesh
 Isosurface extraction, 166
MetaDataDictionary, 42, 46
 Begin(), 46
 ConstIterator, 46
 End(), 46
 header, 41
 Iterator, 46
 MetaDataObject, 46
 String entries, 46
MetaDataObject
 GetMetaDataObjectValue(), 42
 header, 41
 Strings, 46

- Model to Image Registration
Observer, 323
- Mutual Information
Statistics, 479
- NearestNeighborInterpolateImageFunction,
264
- Open Science, 193
- RecursiveGaussianImageFilter
SetDirection(), 74, 101
SetNormalizeAcrossScale(), 75, 102
SetOrder(), 75, 101
- Registration
Finite Element-Based, 287
- RegularStepGradientDescentOptimizer
SetLearningRate(), 200
SetMinimumStepLength(), 200
SetNumberOfIterations(), 200
SetRelaxationFactor(), 200
- Resampling, 156
- RescaleIntensityImageFilter
Instantiation, 19, 22
New(), 19, 22
Pointer, 19, 22
SetOutputMaximum(), 19, 23
SetOutputMinimum(), 19, 23
- RGB
reading Image, 7, 28
writing Image, 7, 28
- Series
Reading, 23
Writing, 23
- SetConductanceParameter()
itk::CurvatureAnisotropicDiffusion-
ImageFilter, 109
- SetDilateValue()
itk::BinaryDilateImageFilter, 85
- SetDomainSigma()
itk::BilateralImageFilter, 117
- SetErodeValue()
- itk::BinaryErodeImageFilter, 85
- SetFileName()
itk::ImageFileReader, 2, 9, 11, 13, 17,
19, 23
- itk::ImageFileWriter, 2, 9, 11, 13, 17,
19, 23
- SetInsideValue()
itk::BinaryThresholdImageFilter, 53
itk::OtsuThresholdImageFilter, 348
- SetKernel()
itk::BinaryDilateImageFilter, 84
itk::BinaryErodeImageFilter, 84
itk::GrayscaleDilateImageFilter, 87
itk::GrayscaleErodeImageFilter, 87
- SetNumberOfIterations()
itk::CurvatureAnisotropicDiffusion-
ImageFilter,
109
- itk::CurvatureFlowImageFilter, 111
- itk::GradientAnisotropicDiffusion-
ImageFilter,
106
- itk::MinMaxCurvatureFlowImageFilter,
114
- itk::VectorCurvatureAnisotropic-
DiffusionImageFilter, 121,
125
- itk::VectorGradientAnisotropic-
DiffusionImageFilter, 119,
123
- SetOutputMaximum()
itk::RescaleIntensityImageFilter, 61
- SetOutputMinimum()
itk::RescaleIntensityImageFilter, 61
- SetOutsideValue()
itk::BinaryThresholdImageFilter, 53
itk::OtsuThresholdImageFilter, 348
itk::ThresholdImageFilter, 56
- SetRadius()
itk::BinaryBallStructuringElement, 84,
87
- SetRangeSigma()
itk::BilateralImageFilter, 117

- SetScale()
 - itk::ShiftScaleImageFilter, 61
- SetShift()
 - itk::ShiftScaleImageFilter, 61
- SetSigma()
 - itk::GradientMagnitudeRecursiveGaussianImageFilter, 67
 - itk::LaplacianRecursiveGaussianImageFilter, 78
 - itk::RecursiveGaussianImageFilter, 76, 102
- SetTimeStep()
 - itk::CurvatureAnisotropicDiffusionImageFilter, 109
 - itk::CurvatureFlowImageFilter, 111
 - itk::GradientAnisotropicDiffusionImageFilter, 106
 - itk::MinMaxCurvatureFlowImageFilter, 114
 - itk::VectorCurvatureAnisotropicDiffusionImageFilter, 121, 125
 - itk::VectorGradientAnisotropicDiffusionImageFilter, 119, 123
- Statistics
 - Bayesian plugin classifier, 494
 - Covariance, 446
 - Expectation maximization, 500
 - Gaussian (normal) probability density function, 458
 - Heap sort, 455
 - Images, 464
 - Importing ListSample to Histogram, 451
 - Insert sort, 455
 - Introspective sort, 455
 - Joint Entropy, 479
 - Joint Histogram, 479
 - k-means clustering (using k-d tree), 485
 - Mean, 446
 - Mixture model estimation, 500
 - Mutual Information, 479
 - Order statistics, 455
 - Quick select, 455
 - Random number generation
 - Normal (Gaussian) distribution, 463
 - Sampling measurement vectors using radius, 453
 - Sorting, 455
 - Weighted covariance, 448
 - Weighted mean, 448
 - Subsampling, 156
 - Supersampling, 156
 - Surface Extraction, 166
 - SystemTools, 40
 - MakeDirectory, 40
- Vector
 - Geometrical Concept, 244
- Vector images
 - Reading, 14
 - Writing, 14
- VectorMagnitudeImageFilter
 - header, 18
 - Instantiation, 19
 - New(), 19
 - Pointer, 19
- Voronoi partitions, 128
 - itk::DanielssonDistanceMapImageFilter, 128
- WarpImageFilter, 302
- Watersheds, 364
 - ImageFilter, 366
 - Overview, 364
- WindowedSincInterpolateImageFunction, 264