The ORFEO Tool Box Software Guide
Sixth Edition
Updated for OTB-2.2

OTB Development Team

May 30, 2008

http://otb.cnes.fr
e-mail: otb@cnes.fr

The ORFEO Toolbox is not a black box.

Ch.D.

Foreword

Beside the Pleiades (PHR) and Cosmo-Skymed (CSK) systems developments forming ORFEO, the dual and bilateral system (France - Italy) for Earth Observation, the ORFEO Accompaniment Program was set up, to prepare, accompany and promote the use and the exploitation of the images derived from these sensors.

The creation of a preparatory program[1] is needed because of :

- the new capabilities and performances of the ORFEO systems (optical and radar high resolution, access capability, data quality, possibility to acquire simultaneously in optic and radar),
- the implied need of new methodological developments : new processing methods, or adaptation of existing methods,
- the need to realise those new developments in very close cooperation with the final users for better integration of new products in their systems.

This program was initiated by CNES mid-2003 and will last until 2009. It consists in two parts, between which it is necessary to keep a strong interaction :

- A Thematic part,
- A Methodological part.

The Thematic part covers a large range of applications (civil and defence), and aims at specifying and validating value added products and services required by end users. This part includes consideration about products integration in the operational systems or processing chains. It also includes a careful thought on intermediary structures to be developed to help non-autonomous

[1] http://smsc.cnes.fr/PLEIADES/A_prog_accomp.htm

users. Lastly, this part aims at raising future users awareness, through practical demonstrations and validations.

The Methodological part objective is the definition and the development of tools for the operational exploitation of the future submetric optic and radar images (tridimensional aspects, changes detection, texture analysis, pattern matching, optic radar complementarities). It is mainly based on R&D studies and doctorate and post-doctorate researches.

In this context, CNES[2] decided to develop the *ORFEO ToolBox* (OTB), a set of algorithms encapsulated in a software library. The goals of the OTB is to capitalise a methological *savoir faire* in order to adopt an incremental development approach aiming to efficiently exploit the results obtained in the frame of methodological R&D studies.

All the developments are based on FLOSS (Free/Libre Open Source Software) or existing CNES developments. OTB is distributed under the CéCILL licence, `http://www.cecill.info/licences/Licence_CeCILL_V2-en.html`.

OTB is implemented in C++ and is mainly based on ITK[3] (Insight Toolkit).

[2] http://www.cnes.fr
[3] http://www.itk.org

Contributors

The ORFEO Toolbox is a project conducted by CNES and developed in cooperation with CS (Communication & Systèmes), `http://www.c-s.fr`.

This Software Guide is based on the ITK Software Guide: the build process for the documentation, many examples and even the LaTeX sources were taken from ITK. We are very grateful to the ITK developpers and contributors and especially to Luis Ibáñez.

The OTB specifics were implemented and documented by the OTB Development Team:

- Jordi Inglada did most of the editing work for this guide and is guilty for the choice of data and examples. He also implemented the SVM classification approach and the change detection framework.
- Thomas Feuvrier is the OTB system guru: he implemented the build procedures for the code and the documentation; he implemented the IO functionalities and the streaming IO capabilities; he also developped the visualization tools.
- Julien Michel implemented the morphological pyramid functionnalities, the spatial reasoning tools, Kohonen's SOM, disparity map estimation, as well as some applications and filters.
- Romain Garrigues is responsible for some applications and coded most of the orthorectification routines (ported from code developped by Miarintsoa Ramanantsimiavona); he also worked on the multi-platform installation procedures.
- Emmanuel Christophe developped a road extraction algorithm and is responsible for the tutorials.
- Cyrille Valladeau coded some vegetation indices and ported the Bayesian fusion algorithm kindly provided by Julien Radoux and Dominique Fasbender (UCL).

- Grégoire Mercier contributed the Kullback-Leibler change detectors, several SVM kernels and the SEM algorithm.
- Vincent Poulain contributed a DXF reader.
- Patrick Imbo developped some filters and feature extraction algorithms.
- Caroline Ruffel coded the edge and line detectors among other fuctionnalities.

Contributions from users are expected and encouraged for the comming versions of OTB.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Part I

Introduction

CHAPTER

ONE

Welcome

Welcome to the *ORFEO ToolBox (OTB) Software Guide*.

This document presents the essential concepts used in OTB. It will guide you through the road of learning and using OTB. The Doxygen documentation for the OTB application programming interface is available on line at `http://orfeo-toolbox.sourceforge.net/Doxygen/html`.

1.1 Organization

This software guide is divided into three parts, each of which is further divided into several chapters. Part I is a general introduction to OTB, with—in the next chapter—a description of how to install the ORFEO Toolbox on your computer. Part I also introduces basic system concepts such as an overview of the system architecture, and how to build applications in the C++ programming language. Part II describes the system from the user point of view. Dozens of examples are used to illustrate important system features. Part III is for the OTB developer. It explains how to create your own classes and extend the system.

1.2 How to Learn OTB

There are two broad categories of users of OTB. First are class developers, those who create classes in C++. The second, users, employ existing C++ classes to build applications. Class developers must be proficient in C++, and if they are extending or modifying OTB, they must also be familiar with OTB's internal structures and design (material covered in Part III).

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

1.3 Software Organization

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

1.3.1 Obtaining the Software

Periodic releases of the software are available on the OTB Web site. These official releases are available a few times a year and announced on the ORFEO Web pages and mailing lists.

This software guide assumes that you are working with the official OTB version 1.0 release (available on the OTB Web site).

1.4 Downloading OTB

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

`http://otb.cnes.fr/`

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

Once you fill out this form you will have access to the download page. This page can be book marked to facilitate subsequent visits to the download site without having to complete any form again.

Then choose the tarball that better fits your system. The options are `.zip` and `.tgz` files. The first type is better suited for MS-Windows while the second one is the preferred format for UNIX systems.

Once you unzip or untar the file, a directory called `OTB` will be created in your disk and you will be ready for starting the configuration process described in Section 2.2.1 on page 12.

1.4.1 Join the Mailing List

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

`http://groups.google.com/group/otb-users`

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

1.4.2 Directory Structure

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

OTB is organized into several different modules. There are three: the `OTB`, `OTB-Documents` and `OTB-Applications` modules. The source code, examples and applications are found in the `OTB` module; documentation, tutorials, and material related to the design and marketing of OTB are found in `OTB-Documents`; and fairly complex applications using OTB (and other systems such as VTK and FLTK) are available from `OTB-Applications`. Usually you will work with the `OTB` module unless you are a developer, are teaching a course, or are looking at the details of various design documents. The `OTB-Applications` module should only be downloaded and compiled once the `OTB` module is functioning properly.

The `OTB` module contains the following subdirectories:

- `OTB/Code`—the heart of the software; the location of the majority of the source code.
- `OTB/Examples`—a suite of simple, well-documented examples used by this guide and to illustrate important OTB concepts.
- `OTB/Testing`—a large number of small programs used to test OTB. These examples tend to be minimally documented but may be useful to demonstrate various system concepts.
- `OTB/Utilities`—supporting software for the OTB source code. For example, libraries such as `ITK`.

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

- `OTB/Code/Common`—core classes, macro definitions, typedefs, and other software constructs central to OTB.
- `OTB/Code/BasicFilters`—basic image processing filters.
- `OTB/Code/IO`—classes that support the reading and writing of data.
- `OTB/Code/Projections`—classes allowing to deal with sensor models and cartographic projections.
- `OTB/Code/Radiometry`—classes allowing to compute vegetation indices and radiometric corrections.
- `OTB/Code/Fusion`—image fusion algorithms, as for instance, pansharpening.
- `OTB/Code/FeatureExtraction`—the location of many feature extraction algorithms.
- `OTB/Code/ChangeDetection`—a set of remote sensing image change detection algorithms.
- `OTB/Code/MultiScale`—a set of functionalities for multiscale image analysis and synthesis.
- `OTB/Code/Learning`—several functionnalities for supervised learning and classification.
- `OTB/Code/SpatialReasoning`—several functionnalities high level image analysis using spatial reasoning techniques.
- `OTB/Code/Visu`—utilities for simple image visualization.
- `OTB/Code/Gui`—very basic widgets for building graphical user interfaces, such as progress bars for filters, etc.

The `OTB-Documents` module contains the following subdirectories:

- `OTB-Documents/CourseWare`—material related to teaching OTB.
- `OTB-Documents/Latex`—LaTeX styles to produce this work as well as other documents.
- `OTB-Documents/SoftwareGuide`—LaTeX files used to create this guide. (Note that the code found in `OTB/Examples` is used in conjunction with these LaTeX files.)

The `OTB-Applications` module contains large, relatively complex examples of OTB usage.

1.4.3 Documentation

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

Doxygen Documentation. The Doxygen documentation is an essential resource when working with OTB. These extensive Web pages describe in detail every class and method in the system. The documentation also contains inheritance and collaboration diagrams, listing of event invocations, and data members. The documentation is heavily hyper-linked to other classes and to the source code. The Doxygen documentation is available on-line at `http://orfeo-toolbox.sourceforge.net/Doxygen/html`.

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

1.4.4 Data

The OTB Toolkit was designed to support the ORFEO Acompaniment Program and its associated data. This data is available `http://smsc.cnes.fr/PLEIADES/index.htm`.

1.5 The OTB Community and Support

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

- Users may actively report bugs, defects in the system API, and/or submit feature requests. Currently the best way to do this is through the OTB users mailing list.
- Developers may contribute classes or improve existing classes. If you are a developer, you may request permission to join the OTB developers mailing list. Please do so by sending email to otb “at” cnes.fr. To become a developer you need to demonstrate both a level of competence as well as trustworthiness. You may wish to begin by submitting fixes to the OTB users mailing list.
- Research partnerships with members of the ORFEO Acompaniment Program are encouraged. CNES will encourage the use of OTB in proposed work and research projects.
- Educators may wish to use OTB in courses. Materials are being developed for this purpose, e.g., a one-day, conference course and semester-long graduate courses. Watch the OTB web pages or check in the `OTB-Documents/CourseWare` directory for more information.

1.6 A Brief History of OTB

Beside the Pleiades (PHR) and Cosmo-Skymed (CSK) systems developments forming ORFEO, the dual and bilateral system (France - Italy) for Earth Observation, the ORFEO Accompaniment Program was set up, to prepare, accompany and promote the use and the exploitation of the images derived from these sensors.

The creation of a preparatory program[1] is needed because of :

- the new capabilities and performances of the ORFEO systems (optical and radar high resolution, access capability, data quality, possibility to acquire simultaneously in optic and radar),
- the implied need of new methodological developments : new processing methods, or adaptation of existing methods,
- the need to realise those new developments in very close cooperation with the final users for better integration of new products in their systems.

This program was initiated by CNES mid-2003 and will last until 2009. It consists in two parts, between which it is necessary to keep a strong interaction :

- A Thematic part
- A Methodological part.

The Thematic part covers a large range of applications (civil and defence ones), and aims at specifying and validating value added products and services required by end users. This part includes consideration about products integration in the operational systems or processing lines. It also includes a careful thought on intermediary structures to be developed to help non-autonomous users. Lastly, this part aims at raising future users awareness, through practical demonstrations and validations.

The Methodological part objective is the definition and the development of tools for the operational exploitation of the future submetric optic and radar images (tridimensional aspects, change detection, texture analysis, pattern matching, optic radar complementarities). It is mainly based on R&D studies and doctorate and post-doctorate research.

In this context, CNES[2] decided to develop the *ORFEO ToolBox* (OTB), a set of algorithms encapsulated in a software library. The goals of the OTB is to capitalise a methological *savoir faire* in order to adopt an incremental development approach aimin to efficiently exploit the results obtained in the frame of methodological R&D studies.

All the developments are based on FLOSS (Free/Libre Open Source Software) or existing CNES developments.

[1] http://smsc.cnes.fr/PLEIADES/A_prog_accomp.htm
[2] http://www.cnes.fr

OTB is implemented in C++ and is mainly based on ITK[3] (Insight Toolkit):

- ITK is used as the core element of OTB
- OTB classes inherit from ITK classes
- The software development procedure of OTB is strongly inspired from ITK's (Extreme Programming, test-based coding, Generic Programming, etc.)
- The documentation production procedure is the same as for ITK
- Several chapters of the Software Guide are litterally copied from ITK's Software Guide (with permission).
- Many examples are taken from ITK.

1.6.1 ITK's history

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 Ibáñez at UNC (Luis is now at Kitware), Ross Whitaker with Josh Cates at UT (both now at Utah), and Dimitri Metaxas at UPenn (now at Rutgers). In addition, several subcontractors rounded out the consortium including Peter Raitu at Brigham & Women's Hospital, Celina Imielinska and Pat Molholt at Columbia University, Jim Gee at UPenn's Grasp Lab, and George Stetten at the University of Pittsburgh.

In 2002 the first official public release of ITK was made available.

[3] http://www.itk.org

CHAPTER

TWO

Installation

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

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

- Cygwin, MinGW, Visual Studio 7 and 8 on MS-Windows
- GCC on Unix/Linux systems

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

2.1 External Libraries

The OTB depends on 3 libraries:

- ITK: you have the choice between using OTB's internal version of ITK or building your own ITK outside the OTB source tree. The recommended choice is the first one. See next section for details. If you choose to use an external version of ITK, go to `http://www.itk.org` and follow the guidelines to download and install ITK.
- GDAL: The support of remote sensing imagery formats is ensured through the use of the GDAL library. Please see `http://www.remotesensing.org/gdal/` for informations on how to download and install this library on your system.

- Fltk: this library is used for the visualization functionnalities. See `http://www.fltk.org/` for details about dowloading and installing Fltk. OTB has been tested with version 1.1.7.

See section 23.4 for quick installation guidelines.

2.2 Configuring OTB

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

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

2.2.1 Preparing CMake

CMake can be downloaded at no cost from

`http://www.cmake.org`

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

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

```
mkdir OTB-binary
cd OTB-binary
```

```
ccmake ../OTB
```

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

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

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

2.2.2 Configuring OTB

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

An additional resource is available in the `OTB-Applications` module, which contains applications incorporating GUIs and different levels of visualization. However, building this module should be postponed until you are familiar with the basic structure of the toolbox and the building process.

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

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

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

Building ITK

The OTB installation procedure allows you to choose between building the OTB with an external version of ITK already present in your system. The choice is made by using the `OTB_USE_EXTERNAL_ITK` CMake variable.

2.3 Getting Started With OTB

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

Once both files are in your directory you can run CMake in order to configure your project. Under UNIX, you can cd to your newly created directory and type "`ccmake .`". Note the "." in the command line for indicating that the `CMakeLists.txt` file is in the current directory. The curses interface will require you to provide the directory where OTB was built. This is the same path that you indicated for the `OTB_BINARY_DIR` variable at the time of configuring OTB. Under Windows you can run CMakeSetup and provide your newly created directory as being both the source directory and the binary directory for your new project (i.e., an in-source build). Then CMake will require you to provide the path to the binary directory where OTB was built. The OTB binary directory will contain a file named `OTBConfig.cmake` generated during the configuration process at the time OTB was built. From this file, CMake will recover all the information required to configure your new OTB project.

2.3.1 Hello World !

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

```
PROJECT(HelloWorld)

FIND_PACKAGE(OTB)
IF(OTB_FOUND)
```

```
  INCLUDE(${OTB_USE_FILE})
ELSE(OTB_FOUND)
  MESSAGE(FATAL_ERROR
      "Cannot build OTB project without OTB.  Please set OTB_DIR.")
ENDIF(OTB_FOUND)


ADD_EXECUTABLE(HelloWorld HelloWorld.cxx )

TARGET_LINK_LIBRARIES(HelloWorld OTBCommon OTBIO ITKCommon ITKIO)
```

The first line defines the name of your project as it appears in Visual Studio (it will have no effect under UNIX). The second line loads a CMake file with a predefined strategy for finding OTB [1]. If the strategy for finding OTB fails, CMake will prompt you for the directory where OTB is installed in your system. In that case you will write this information in the `OTB_DIR` variable. The line `INCLUDE(${USE_OTB_FILE})` loads the `UseOTB.cmake` file to set all the configuration information from OTB.

The next block of lines is needed in order for CMake to know whether you are using the OTB's internal version of ITK or an external one. In the second case, CMake will try to find ITK in your system. As for OTB, if it fails in finding ITK, it will ask you to manually set the ITK location.

The line `ADD_EXECUTABLE` defines as its first argument the name of the executable that will be produced as result of this project. The remaining arguments of `ADD_EXECUTABLE` are the names of the source files to be compiled and linked. Finally, the `TARGET_LINK_LIBRARIES` line specifies which OTB libraries will be linked against this project.

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

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

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

int main()
{
  typedef otb::Image< unsigned short, 2 > ImageType;

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

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

  return 0;
}
```

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

This code instantiates an image whose pixels are represented with type `unsigned short`. The image is then constructed and assigned to a `itk::SmartPointer`. Although later in the text we will discuss `SmartPointer`'s in detail, for now think of it as a handle on an instance of an object (see section 3.2.4 for more information). The `itk::Image` class will be described in Section 5.1.

At this point you have successfully installed and compiled OTB, and created your first simple program. If you have difficulties, please join the otb-users mailing list (Section 1.4.1 on page 5) and post questions there.

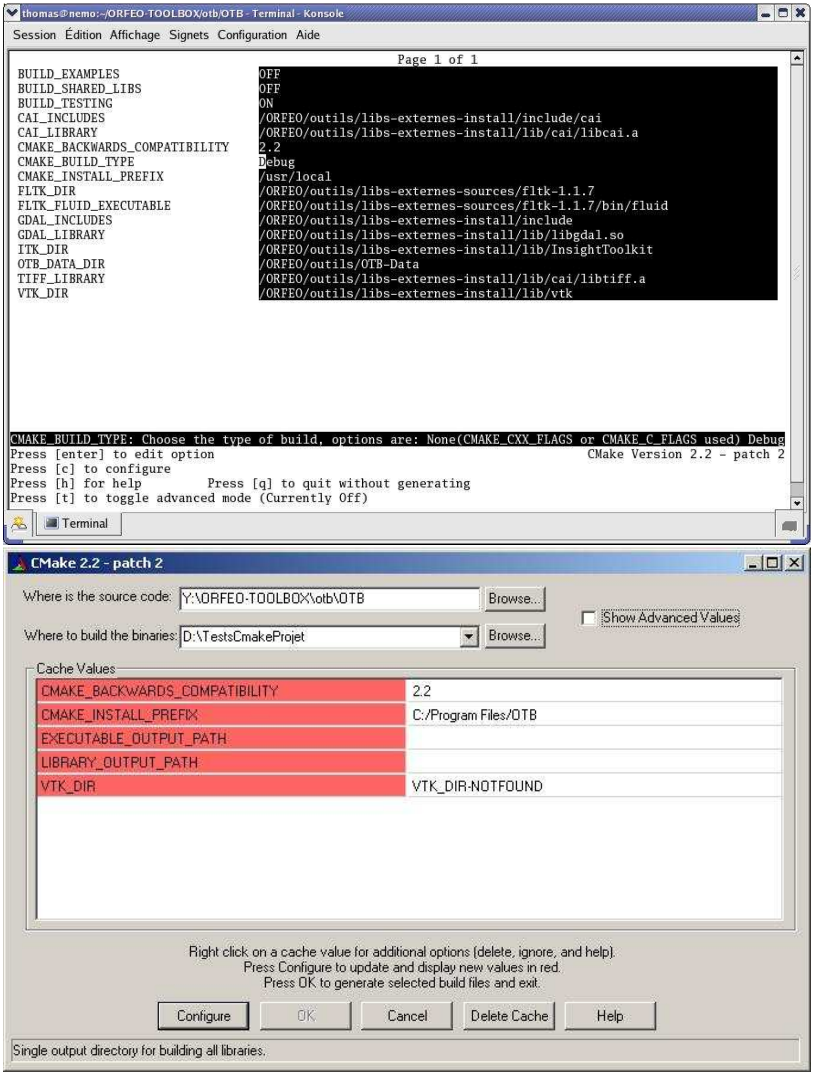


Figure 2.1: CMake interface. Top) `ccmake`, the UNIX version based on `curses`. Bottom) `CMakeSetup`, the MS-Windows version based on MFC.

CHAPTER

THREE

System Overview

The purpose of this chapter is to provide you with an overview of the *ORFEO Toolbox* system. We recommend that you read this chapter to gain an appreciation for the breadth and area of application of OTB. In this chapter, we will make reference either to *OTB features* or *ITK features* without distinction. Bear in mind that OTB uses ITK as its core element, so all the fundamental elements of OTB come from ITK. OTB extends the functionalities of ITK for the remote sensing image processing comunity. We benefit from the Open Source development approach chosen for ITK, which allows us to provide an impressive set of functionalities with much lesser effort than it would have been the case in a closed source universe!

3.1 System Organization

The ORFEO Toolbox consists of several subsystems. A brief description of these subsystems follows. Later sections in this chapter—and in some cases additional chapters—cover these concepts in more detail. (Note: in the previous chapter two other modules—`OTB-Documents` and `OTB-Applications` were briefly described.)

Essential System Concepts. Like any software system, OTB is built around some core design concepts. OTB uses those of ITK. 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 OTB, as ITK uses VXL's VNL numerics libraries. These are easy-to-use C++ wrappers around the Netlib Fortran numerical analysis routines (`http://www.netlib.org`).

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

ITK's 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 multi-threading, and are streaming capable (i.e., can operate on pieces of data to minimize the memory footprint).

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

Spatial Objects. Geometric shapes are represented in OTB using the ITK spatial object hierarchy. These classes are intended to support modeling of anatomical structures in ITK. OTB uses them in order to model cartographic elements. 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 geometrical priors in both segmentation and registration methods.

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

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

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

Wrapping. ITK uses a unique, powerful system for producing interfaces (i.e., "wrappers") to interpreted languages such as Tcl and Python. The GCC_XML tool is used to produce an XML description of arbitrarily complex C++ code; CSWIG is then used to transform the XML description into wrappers using the SWIG package. OTB does not use this system at present.

3.2 Essential System Concepts

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

3.2.1 Generic Programming

Generic programming is a method of organizing libraries consisting of generic—or reusable—software components [65]. The idea is to make software that is capable of "plugging together" in an efficient, adaptable manner. The essential ideas of generic programming are *containers* to hold data, *iterators* to access the data, and *generic algorithms* that use containers and iterators to create efficient, fundamental algorithms such as sorting. Generic programming is implemented in C++ with the *template* programming mechanism and the use of the STL Standard Template Library [5].

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

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

3.2.2 Include Files and Class Definitions

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

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

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

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

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

3.2.3 Object Factories

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

The object factory enables users to control run-time instantiation of classes by registering one or more factories with `itk::ObjectFactoryBase`. These registered factories support the method `CreateInstance(classname)` which takes as input the name of a class to create. The factory can choose to create the class based on a number of factors including the computer system configuration and environment variables. For example, in a particular application an OTB user may wish to deploy their own class implemented using specialized image processing hardware (i.e., to realize a performance gain). By using the object factory mechanism, it is possible at run-time to replace the creation of a particular OTB filter with such a custom class. (Of course, the class must provide the exact same API as the one it is replacing.) To do this, the user compiles her class (using the same compiler, build options, etc.) and inserts the object code into a shared library or DLL. The library is then placed in a directory referred to by the `OTB_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 OTB 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 `Utilities/ITK/Common/itkMacro.h`.

3.2.4 Smart Pointers and Memory Management

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

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

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

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

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

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

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

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

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

3.2.5 Error Handling and Exceptions

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

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

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

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

Note that `itk::ByteSwapperError` is a subclass of `itk::ExceptionObject`. (In fact in OTB all exceptions should be derived from `itk::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 OTB error, or you can trap *any* OTB exception by catching ExceptionObject.

3.2.6 Event Handling

Event handling in OTB is implemented using the Subject/Observer design pattern [33] (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 OTB 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 OTB are subclasses of EventObject; look in `itkEventObject.h` to determine which events are available.)

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

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

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

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

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

3.2.7 Multi-Threading

Multithreading is handled in OTB through ITK's high-level design abstraction. This approach provides portable multithreading and hides the complexity of differing thread implementations on the many systems supported by OTB. 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

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

The VNL numerics library includes classes for

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

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

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

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

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

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

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

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

3.4 Data Representation

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

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

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

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

The `otb::Image` class extends the functionalities of the `itk::Image` in order to take into account particular remote sensing features as geographical projections, etc.

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

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

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

3.5 Data Processing Pipeline

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

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

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

```
typedef otb::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);

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

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

3.6 Spatial Objects

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

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

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

1. Specify a spatial object's parent and children objects. In this way, a city may contain roads and those roads 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., pixel spacing) of the image used to generate a particular instance of a `itk::LineSpatialObject`.

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

There are a limited number of spatial objects and their methods in ITK, but their number is growing and their potential is huge. Using the nominal spatial object capabilities, methods such as 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 building with an image or to register two different segmentations of a particular object in object-based change detection.

Part II

Tutorials

CHAPTER

FOUR

Building Simple Applications with OTB

Well, that's it, you've just downloaded and installed OTB, lured by the promise that you will be able to do everything with it. That's true, you will be able to do everything but - there is always a *but* - some effort is required.

OTB uses the very powerful systems of generic programing, many classes are already available, some powerful tools are defined to help you with recurrent tasks, but it is not an easy world to enter.

These tutorials are designed to help you enter this world and grasp the logic behind OTB. Each of these tutorials should not take more than 10 minutes (typing included) and each is designed to highlight a specific point. You may not be concerned by the latest tutorials but it is strongly advised to go through the first few which cover the basics you'll use almost everywhere.

4.1 Hello world

Let's start by the typical *Hello world* program. We are going to compile this C++ program linking to your new OTB.

First, create a new folder to put your new programs (all the examples from this tutorial) in and go into this folder.

Since all programs using OTB are handled using the CMake system, we need to create a `CMakeLists.txt` that will be used by CMake to compile our program. An example of this file can be found in the `OTB/Examples/Tutorials` directory. The `CMakeLists.txt` will be very similar between your projects.

Open the `CMakeLists.txt` file and write in the few lines:

```
PROJECT(Tutorials)

FIND_PACKAGE(OTB)
IF(OTB_FOUND)
```

```
  INCLUDE(${OTB_USE_FILE})
ELSE(OTB_FOUND)
  MESSAGE(FATAL_ERROR
      "Cannot build OTB project without OTB.  Please set OTB_DIR.")
ENDIF(OTB_FOUND)

ADD_EXECUTABLE(HelloWorldOTB HelloWorldOTB.cxx )
TARGET_LINK_LIBRARIES(HelloWorldOTB OTBCommon OTBIO)
```

The first line defines the name of your project as it appears in Visual Studio (it will have no effect under UNIX or Linux). The second line loads a CMake file with a predefined strategy for finding OTB [1]. If the strategy for finding OTB fails, CMake will prompt you for the directory where OTB is installed in your system. In that case you will write this information in the `OTB_DIR` variable. The line `INCLUDE(${USE_OTB_FILE})` loads the `UseOTB.cmake` file to set all the configuration information from OTB.

The line `ADD_EXECUTABLE` defines as its first argument the name of the executable that will be produced as result of this project. The remaining arguments of `ADD_EXECUTABLE` are the names of the source files to be compiled and linked. Finally, the `TARGET_LINK_LIBRARIES` line specifies which OTB libraries will be linked against this project.

The source code for this example can be found in the file
`Examples/Tutorials/HelloWorldOTB.cxx`.

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

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

int main( int argc, char * argv[] )
{
  typedef otb::Image< unsigned short, 2 > ImageType;

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

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

  return 0;
}
```

This code instantiates an image whose pixels are represented with type `unsigned short`. The image is then created and assigned to a `itk::SmartPointer`. 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).

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

Once the file is written, run `ccmake` on the current directory (that is `ccmake ./` under Linux/Unix). If OTB is on a non standard place, you will have to tell CMake where it is. Once your done with CMake (you shouldn't have to do it anymore) run `make`.

You finally have your program. When you run it, you will have the *OTB Hello World !* printed.

Ok, well done! You've just compiled and executed your first OTB program. Actually, using OTB for that is not very useful, and we doubt that you downloaded OTB only to do that. It's time to move on to a more advanced level.

4.2 Pipeline basics: read and write

OTB is designed to read images, process them and write them to disk or view the result. In this tutorial, we are going to see how to read and write images and the basics of the pipeline system.

First, let's add the following lines at the end of the `CMakeLists.txt` file:

```
ADD_EXECUTABLE(Pipeline Pipeline.cxx )
TARGET_LINK_LIBRARIES(Pipeline OTBCommon OTBIO)
```

Now, create a `Pipeline.cxx` file.

The source code for this example can be found in the file
`Examples/Tutorials/Pipeline.cxx`.

Start by including some necessary headers and with the usual `main` declaration:

```
#include "otbImage.h"
#include "otbImageFileReader.h"
#include "otbStreamingImageFileWriter.h"

int main( int argc, char * argv[] )
{
```

Declare the image as an `otb::Image`, the pixel type is declared as an unsigned char (one byte) and the image is specified as having two dimensions.

```
  typedef otb::Image<unsigned char, 2> ImageType;
```

To read the image, we need an `otb::ImageFileReader` which is templated with the image type.

```
  typedef otb::ImageFileReader<ImageType> ReaderType;
  ReaderType::Pointer reader = ReaderType::New();
```

Then, we need an `otb::StreamingImageFileWriter` also templated with the image type.

```
  typedef otb::StreamingImageFileWriter<ImageType> WriterType;
  WriterType::Pointer writer = WriterType::New();
```

The filenames are passed as arguments to the program. We keep it simple for now and we don't check their validity.

```
  reader->SetFileName(argv[1]);
  writer->SetFileName(argv[2]);
```

Now that we have all the elements, we connect the pipeline, pluging the output of the reader to the input of the writer.

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

And finally, we trigger the pipeline execution calling the Update() method on the last element of the pipeline. The last element will make sure to update all previous elements in the pipeline.

```
  writer->Update();

  return 0;
}
```

Once this file is written you just have to run `make`. The `ccmake` call is not required anymore.

Get one image from the `OTB/Examples/Data` directory in the OTB sources. For example get `QB_Suburb.png`.

Now, run your new program as `Pipeline QB_Suburb.png output.png`. You obtain the file `output.png` which is the same image as `QB_Suburb.png`. When you triggered the `Update()` method, OTB opened the original image and wrote it back under another name.

Well. . . that's nice but a bit complicated for a copy program!

Wait a minute! We didn't specify the file format anywhere! Let's try `Pipeline QB_Suburb.png output.jpg`. And voila! The output image is a jpeg file.

That's starting to be a bit more interesting: this is not just a program to copy image files, but also to convert between image formats.

You have just experienced the pipeline structure which executes the filters only when needed and the automatic image format detection.

Now it's time to do some processing in between.

4.3 Filtering pipeline

We are now going to insert a simple filter to do some processing between the reader and the writer.

Let's first add the 2 following lines to the `CMakeLists.txt` file:

```
ADD_EXECUTABLE(FilteringPipeline FilteringPipeline.cxx )
TARGET_LINK_LIBRARIES(FilteringPipeline OTBCommon OTBIO)
```

The source code for this example can be found in the file
`Examples/Tutorials/FilteringPipeline.cxx`.

We are going to use the `itk::GradientMagnitudeImageFilter` to compute the gradient of the image. The begining of the file is similar to the Pipeline.cxx.

We include the required headers, without forgetting to add the header for the `itk::GradientMagnitudeImageFilter`.

```
#include "otbImage.h"
#include "otbImageFileReader.h"
#include "otbStreamingImageFileWriter.h"
#include "itkGradientMagnitudeImageFilter.h"

int main( int argc, char * argv[] )
{
```

We declare the image type, the reader and the writer as before:

```
  typedef otb::Image<unsigned char, 2> ImageType;

  typedef otb::ImageFileReader<ImageType> ReaderType;
  ReaderType::Pointer reader = ReaderType::New();

  typedef otb::StreamingImageFileWriter<ImageType> WriterType;
  WriterType::Pointer writer = WriterType::New();

  reader->SetFileName(argv[1]);
  writer->SetFileName(argv[2]);
```

Now we have to declare the filter. It is templated with the input image type and the output image type like many filters in OTB. Here we are using the same type for the input and the output images:

```
  typedef itk::GradientMagnitudeImageFilter
      <ImageType,ImageType> FilterType;
  FilterType::Pointer filter = FilterType::New();
```

Let's plug the pipeline:

```
  filter->SetInput(reader->GetOutput());
  writer->SetInput(filter->GetOutput());
```

And finally, we trigger the pipeline execution calling the `Update()` method on the writer

```
  writer->Update();

  return 0;
}
```

Compile with `make` and execute as `FilteringPipeline QB_Suburb.png output.png`.

You have the filtered version of your image in the `output.png` file.

Now, you can practice a bit and try to replace the filter by one of the 150+ filters which inherit from the `otb::ImageToImageFilter` class. You will definitely find some useful filters here!

4.4 Handling types: scaling output

If you tried some other filter in the previous example, you may have noticed that in some cases, it does not make sense to save the output directly as an integer. This is the case if you tried the `itk::CannyEdgeDetectionImageFilter`. If you tried to use it directly in the previous example, you will have some warning about converting to unsigned char from double.

The output of the Canny edge detection is a floating point number. A simple solution would be to used double as the pixel type. Unfortunately, most image formats use integer typed and you should convert the result to an integer image if you still want to visualize your images with your usual viewer (we will see in a tutorial later how you can avoid that using the built-in viewer).

To realize this conversion, we will use the `itk::RescaleIntensityImageFilter`.

Add the two lines to the `CMakeLists.txt` file:

```
ADD_EXECUTABLE(ScalingPipeline ScalingPipeline.cxx )
TARGET_LINK_LIBRARIES(ScalingPipeline OTBCommon OTBIO)
```

The source code for this example can be found in the file
`Examples/Tutorials/ScalingPipeline.cxx`.

This example illustrates the use of the `itk::RescaleIntensityImageFilter` to convert the result for proper display.

We include the required header including the header for the `itk::CannyEdgeImageFilter` and the `itk::RescaleIntensityImageFilter`.

```
#include "otbImage.h"
#include "otbImageFileReader.h"
#include "otbStreamingImageFileWriter.h"
#include "itkCannyEdgeDetectionImageFilter.h"
#include "itkRescaleIntensityImageFilter.h"

int main( int argc, char * argv[] )
{
```

We need to declare two different image types, one for the internal processing and one to output the results:

```
  typedef double PixelType;
  typedef otb::Image<PixelType, 2> ImageType;

  typedef unsigned char OutputPixelType;
  typedef otb::Image<OutputPixelType, 2> OutputImageType;
```

We declare the reader with the image template using the pixel type double. It is worth noticing that this instanciation does not imply anything about the type of the input image. The original image can be anything, the reader will just convert the result to double.

The writer is templated with the unsigned char image to be able to save the result on one byte images (like png for example).

```
  typedef otb::ImageFileReader<ImageType> ReaderType;
  ReaderType::Pointer reader=ReaderType::New();

  typedef otb::StreamingImageFileWriter<OutputImageType> WriterType;
  WriterType::Pointer writer=WriterType::New();

  reader->SetFileName(argv[1]);
  writer->SetFileName(argv[2]);
```

Now we are declaring the edge detection filter which is going to work with double input and output.

```
  typedef itk::CannyEdgeDetectionImageFilter
      <ImageType,ImageType> FilterType;
  FilterType::Pointer filter = FilterType::New();
```

Here comes the interesting part: we declare the `itk::RescaleIntensityImageFilter`. The input image type is the output type of the edge detection filter. The output type is the same as the input type of the writer.

Desired minimum and maximum values for the output are specified by the methods `SetOutputMinimum()` and `SetOutputMaximum()`.

This filter will actually rescale all the pixels of the image but also cast the type of these pixels.

```
  typedef itk::RescaleIntensityImageFilter
      <ImageType,OutputImageType> RescalerType;
  RescalerType::Pointer rescaler = RescalerType::New();

  rescaler->SetOutputMinimum(0);
  rescaler->SetOutputMaximum(255);
```

Let's plug the pipeline:

```
  filter->SetInput(reader->GetOutput());
  rescaler->SetInput(filter->GetOutput());
  writer->SetInput(rescaler->GetOutput());
```

And finally, we trigger the pipeline execution calling the Update() method on the writer

```
  writer->Update();

  return 0;
}
```

As you should be getting used to it by now, compile with `make` and execute as `ScalingPipeline QB_Suburb.png output.png`.

You have the filtered version of your image in the `output.png` file.

4.5 Working with multispectral or color images

So far, as you may have noticed, we have been working with grey level images, i.e. with only one spectral band. If you tried to process a color image with some of the previous examples you have probably obtained a deceiving grey result.

Often, satellite images combine several spectral band to help the identification of materials: this is called multispectral imagery. In this tutorial, we are going to explore some of the mechanisms used by OTB to process multispectral images.

Add the following lines in the `CMakeLists.txt` file:

```
ADD_EXECUTABLE(Multispectral Multispectral.cxx )
TARGET_LINK_LIBRARIES(Multispectral OTBCommon OTBIO)
```

The source code for this example can be found in the file
`Examples/Tutorials/Multispectral.cxx`.

First, we are going to use `otb::VectorImage` instead of the now traditionnal `otb::Image`. So we include the required header:

```
#include "otbVectorImage.h"
```

We also include some other header which will be useful later. Note that we are still using the `otb::Image` in this example for some of the output.

```
#include "otbImage.h"
#include "otbImageFileReader.h"
#include "otbStreamingImageFileWriter.h"
#include "otbMultiToMonoChannelExtractROI.h"
#include "itkShiftScaleImageFilter.h"
#include "otbPerBandVectorImageFilter.h"


int main( int argc, char * argv[] )
{
```

We want to read a multispectral image so we declare the image type and the reader. As we have done in the previous example we get the filename from the command line.

```
  typedef unsigned short int PixelType;
  typedef otb::VectorImage<PixelType, 2> VectorImageType;

  typedef otb::ImageFileReader<VectorImageType> ReaderType;
  ReaderType::Pointer reader = ReaderType::New();

  reader->SetFileName(argv[1]);
```

Sometime, you need to process only one spectral band of the image. To get only one of the spectral band we use the /doxygenotbMultiToMonoChannelExtractROI. The declaration is as usual:

```
  typedef otb::MultiToMonoChannelExtractROI<PixelType, PixelType> ExtractChannelType;
  ExtractChannelType::Pointer extractChannel = ExtractChannelType::New();
```

We need to pass the parameters to the filter for the extraction. This filter also allow to extract only a spatial subset of the image. However, we will extract the whole channel in this case.

To do that, we need to pass the desired region using the `SetExtractionRegion()` (method such as `SetStartX`, `SetSizeX` are also available). We get the region from the reader with the

`GetLargestPossibleRegion()` method. Before doing that we need to read the metadata from the file: this is done by calling the `UpdateOutputInformation()` on the reader's output. The difference with the `Update()` is that the pixel array is not allocated (yet !) and reduce the memory usage.

```
reader->UpdateOutputInformation();
extractChannel->SetExtractionRegion(reader->GetOutput()->GetLargestPossibleRegion());
```

We chose the channel number to extract (starting from 1) and we plug the pipeline.

```
extractChannel->SetChannel(3);
extractChannel->SetInput(reader->GetOutput());
```

To output this image, we need a writer. As the output of the `otb::MultiToMonoChannelExtractROI` is a `otb::Image`, we need to template the writer with this type.

```
typedef otb::Image<PixelType, 2> ImageType;
typedef otb::StreamingImageFileWriter<ImageType> WriterType;
WriterType::Pointer writer = WriterType::New();

writer->SetFileName(argv[2]);
writer->SetInput(extractChannel->GetOutput());

writer->Update();
```

After this, we have a one band image that we can process with most OTB filters.

In some situation, you may want to apply the same process to all bands of the image. You don't have to extract each band and process them separately. There is several situations:

- the filter (or the combination of filters) you want to use are doing operations that are well defined for `itk::VariableLengthVector` (which is the pixel type), then you don't have to do anything special.
- if this is not working, you can look for the equivalent filter specially designed for vector images.
- some of the filter you need to use applies operations undefined for `itk::VariableLengthVector`, then you can use the `otb::PerBandVectorImageFilter` specially designed for this purpose.

Let's see how this filter is working. We chose to apply the `itk::ShiftScaleImageFilter` to each of the spectral band. We start by declaring the filter on a normal `otb::Image`. Note that we don't need to specify any input for this filter.

```
  typedef itk::ShiftScaleImageFilter<ImageType, ImageType> ShiftScaleType;
  ShiftScaleType::Pointer shiftScale = ShiftScaleType::New();
  shiftScale->SetScale(0.5);
  shiftScale->SetShift(10);
```

We declare the `otb::PerBandVectorImageFilter` which has three template: the input image type, the output image type and the filter type to apply to each band.

The filter is selected using the `SetFilter()` method and the input by the usual `SetInput()` method.

```
  typedef otb::PerBandVectorImageFilter
      <VectorImageType, VectorImageType, ShiftScaleType> VectorFilterType;
  VectorFilterType::Pointer vectorFilter = VectorFilterType::New();
  vectorFilter->SetFilter(shiftScale);

  vectorFilter->SetInput(reader->GetOutput());
```

Now, we just have to save the image using a writer templated over an `otb::VectorImage`:

```
  typedef otb::StreamingImageFileWriter<VectorImageType> VectorWriterType;
  VectorWriterType::Pointer writerVector = VectorWriterType::New();

  writerVector->SetFileName(argv[3]);
  writerVector->SetInput(vectorFilter->GetOutput());

  writerVector->Update();

  return 0;
}
```

Compile with `make` and execute as `./Multispectral qb_RoadExtract.tif qb_blue.tif qb_shiftscale.tif`.

4.6 Parsing command line arguments

Well, if you play with some other filters in the previous example, you probably noticed that in many cases, you need to set some parameters to the filters. Ideally, you want to set some of these parameters from the command line.

In OTB, there is a mechanism to help you parse the command line parameters. Let try it!

Add the following lines in the `CMakeLists.txt` file:

```
ADD_EXECUTABLE(SmarterFilteringPipeline SmarterFilteringPipeline.cxx )
TARGET_LINK_LIBRARIES(SmarterFilteringPipeline OTBCommon OTBIO)
```

The source code for this example can be found in the file
`Examples/Tutorials/SmarterFilteringPipeline.cxx`.

We are going to use the `otb::HarrisImageFilter` to find the points of interest in one image.

The derivative computation is performed by a convolution with the derivative of a Gaussian kernel of variance σ_D (derivation scale) and the smoothing of the image is performed by convolving with a Gaussian kernel of variance σ_I (integration scale). This allows the computation of the following matrix:

$$\mu(\mathbf{x},\sigma_I,\sigma_D) = \sigma_D^2 g(\sigma_I) \star \begin{bmatrix} L_x^2(\mathbf{x},\sigma_D) & L_x L_y^2(\mathbf{x},\sigma_D) \\ L_x L_y^2(\mathbf{x},\sigma_D) & L_y^2(\mathbf{x},\sigma_D) \end{bmatrix}$$

The output of the detector is $\det(\mu) - \alpha \mathrm{trace}^2(\mu)$.

We want to set 3 parameters of this filter through the command line: σ_D (SigmaD), σ_I (SigmaI) and α (Alpha).

We are also going to do the things properly and catch the exceptions.

Let first add the two following headers:

```
#include "itkExceptionObject.h"
#include "otbCommandLineArgumentParser.h"
```

The first one is to handle the exceptions, the second one to help us parse the command line.

We include the other required headers, without forgetting to add the header for the `otb::HarrisImageFilter`. Then we start the usual main function.

```
#include "otbImage.h"
#include "otbImageFileReader.h"
#include "otbStreamingImageFileWriter.h"
#include "itkRescaleIntensityImageFilter.h"
#include "otbHarrisImageFilter.h"

int main( int argc, char * argv[] )
{
```

To handle the exceptions properly, we need to put all the instructions inside a `try`.

```
  try
  {
```

Now, we can declare the `otb::CommandLineArgumentParser` which is going to parse the command line, select the proper variables, handle the missing compulsory arguments and print an error message if necessary.

Let's declare the parser:

```
typedef otb::CommandLineArgumentParser ParserType;
ParserType::Pointer parser = ParserType::New();
```

It's now time to tell the parser what are the options we want. Special options are available for input and output images with the `AddInputImage()` and `AddOutputImage()` methods.

For the other options, we need to use the `AddOption()` method. This method allows us to specify

- the name of the option
- a message to explain the meaning of this option
- a shortcut for this option
- the number of expected parameters for this option
- whether or not this option is compulsory

```
parser->AddInputImage();
parser->AddOutputImage();
parser->AddOption("--SigmaD",
                  "Set the sigmaD parameter. Default is 1.0.","-d",1,false);
parser->AddOption("--SigmaI",
                  "Set the sigmaI parameter. Default is 1.0.","-i",1,false);
parser->AddOption("--Alpha",
                  "Set the alpha parameter. Default is 1.0.","-a",1,false);
```

Now that the parser has all this information, it can actually look at the command line to parse it. We have to do this within a `try` - `catch` loop to handle exceptions nicely.

```
typedef otb::CommandLineArgumentParseResult ParserResultType;
ParserResultType::Pointer  parseResult = ParserResultType::New();

try
{
  parser->ParseCommandLine(argc,argv,parseResult);
}

catch( itk::ExceptionObject & err )
{
  std::string descriptionException = err.GetDescription();
  if(descriptionException.find("ParseCommandLine(): Help Parser")
     != std::string::npos)
    {
      return EXIT_SUCCESS;
    }
```

```
    if(descriptionException.find("ParseCommandLine(): Version Parser")
       != std::string::npos)
      {
        return EXIT_SUCCESS;
      }
    return EXIT_FAILURE;
  }
```

Now, we can declare the image type, the reader and the writer as before:

```
typedef double PixelType;
typedef otb::Image<PixelType, 2> ImageType;

typedef unsigned char OutputPixelType;
typedef otb::Image<OutputPixelType, 2> OutputImageType;

typedef otb::ImageFileReader<ImageType> ReaderType;
ReaderType::Pointer reader=ReaderType::New();

typedef otb::StreamingImageFileWriter<OutputImageType> WriterType;
WriterType::Pointer writer=WriterType::New();
```

We are getting the filenames for the input and the output images directly from the parser:

```
reader->SetFileName(parseResult->GetInputImage().c_str());
writer->SetFileName(parseResult->GetOutputImage().c_str());
```

Now we have to declare the filter. It is templated with the input image type and the output image type like many filters in OTB. Here we are using the same type for the input and the output images:

```
typedef otb::HarrisImageFilter
    <ImageType,ImageType> FilterType;
FilterType::Pointer filter = FilterType::New();
```

We set the filter parameters from the parser. The method `IsOptionPresent()` let us know if an optional option was provided in the command line.

```
if(parseResult->IsOptionPresent("--SigmaD"))
  filter->SetSigmaD(parseResult->GetParameterDouble("--SigmaD"));

if(parseResult->IsOptionPresent("--SigmaI"))
  filter->SetSigmaI(parseResult->GetParameterDouble("--SigmaI"));

if(parseResult->IsOptionPresent("--Alpha"))
  filter->SetAlpha(parseResult->GetParameterDouble("--Alpha"));
```

We add the rescaler filter as before

```
    typedef itk::RescaleIntensityImageFilter
      <ImageType,OutputImageType> RescalerType;
    RescalerType::Pointer rescaler = RescalerType::New();

    rescaler->SetOutputMinimum(0);
    rescaler->SetOutputMaximum(255);
```

Let's plug the pipeline:

```
    filter->SetInput(reader->GetOutput());
    rescaler->SetInput(filter->GetOutput());
    writer->SetInput(rescaler->GetOutput());
```

We trigger the pipeline execution calling the `Update()` method on the writer

```
    writer->Update();

  }
```

Finally, we have to handle exceptions we may have raised before

```
  catch( itk::ExceptionObject & err )
    {
    std::cout << "Following otbException catch :" << std::endl;
    std::cout << err << std::endl;
    return EXIT_FAILURE;
    }
  catch( std::bad_alloc & err )
    {
    std::cout << "Exception bad_alloc : "<<(char*)err.what()<< std::endl;
    return EXIT_FAILURE;
    }
  catch( ... )
    {
    std::cout << "Unknown Exception found !" << std::endl;
    return EXIT_FAILURE;
    }
  return EXIT_SUCCESS;
}
```

Compile with `make` as usual. The execution is a bit different now as we have an automatic parsing of the command line. First, try to execute as `SmarterFilteringPipeline` without any argument.

The usage message (automatically generated) appears:

```
'--InputImage' option is obligatory !!!

 Usage : ./SmarterFilteringPipeline
      [--help|-h]           :  Help
      [--version|-v]        :  Version
       --InputImage|-in     :  input image file name   (1 parameter)
       --OutputImage|-out   :  output image file name   (1 parameter)
      [--SigmaD|-d]         :  Set the sigmaD parameter of the Harris points of
interest  algorithm. Default is 1.0.  (1 parameter)
      [--SigmaI|-i]         :  Set the SigmaI parameter of the Harris points of
interest  algorithm. Default is 1.0.  (1 parameter)
      [--Alpha|-a]          :  Set the alpha parameter of the Harris points of
interest  algorithm. Default is 1.0.  (1 parameter)
```

That looks a bit more professional: another user should be able to play with your program. As this is automatic, that's a good way not to forget to document your programs.

So now you have a better idea of the command line options that are possible. Try `SmarterFilteringPipeline -in QB_Suburb.png -out output.png` for a basic version with the default values.

If you want a result that looks a bit better, you have to adjust the parameter with `SmarterFilteringPipeline -in QB_Suburb.png -out output.png -d 1.5 -i 2 -a 0.1` for example.

4.7 Viewer

So far, we had to save the image and use an external viewer every time we wanted to see the result of our processing. That is not very convenient, especially for some *exotic* formats (16 bits, floating point ...). Thankfully, OTB comes with it's own visualization tool.

This tool is accessible by the class `otb::ImageViewer`. We will now design a simple, minimalistic example to illustrate the use for this viewer.

First you need to add the following lines in the `CMakeLists.txt` file:

```
ADD_EXECUTABLE(SimpleViewer SimpleViewer.cxx )
TARGET_LINK_LIBRARIES(SimpleViewer OTBCommon OTBIO OTBGui OTBVisu)
```

Notice that you have to link to 2 other OTB libraries: OTBGui and OTBVisu.

The source code for this example can be found in the file
`Examples/Tutorials/SimpleViewer.cxx`.

Now, we are going to illustrate the use of the `otb::ImageViewer` to display an image or the result of an algorithm without saving the image.

We include the required header including the header for the `itk::GradientMagnitudeImageFilter` and the `otb::ImageViewer`.

```
#include "otbImage.h"
#include "otbImageFileReader.h"
#include "itkGradientMagnitudeImageFilter.h"
#include "otbImageViewer.h"

int main( int argc, char * argv[] )
{
```

We need to declare two different image types, one for the internal processing and one to output the results:

```
  typedef double PixelType;
  typedef otb::Image<PixelType, 2> ImageType;

  typedef otb::ImageFileReader<ImageType> ReaderType;
  ReaderType::Pointer reader=ReaderType::New();

  reader->SetFileName(argv[1]);
```

Now we are declaring the edge detection filter which is going to work with double input and output.

```
  typedef itk::GradientMagnitudeImageFilter
      <ImageType,ImageType> FilterType;
  FilterType::Pointer filter = FilterType::New();
```

Unlike most OTB filters, the `otb::ImageViewer` is templated over the input pixel type instead of the image type. This will allow to use it with scalar and vector images.

```
  typedef otb::ImageViewer<PixelType> ViewerType;
  ViewerType::Pointer viewer = ViewerType::New();
```

Let's plug the pipeline: for the viewer the method is `SetImage()`.

```
  filter->SetInput(reader->GetOutput());
  viewer->SetImage(filter->GetOutput());
```

We trigger the pipeline execution and the image display with the `Show()` method of the viewer.

```
  viewer->Show();
```

A call to `Fl::run()` is mandatory to ask the program to listen to mouse and keyword events until the viewer is closed.

```
  Fl::run();

  return 0;
}
```

After compiling you can execute the program with `SimpleViewer QB_Suburb.png`. The result of the edge detection is displayed. Notice that you can call this simple program with a big image (let's say 30000×30000 pixels). For all multithreaded filters (filters which implement a `ThreadedGenerateData()` method), the image is splitted into piece and only the piece on display is processed.

4.8 Going from raw satellite images to useful products

Quite often, when you buy satellite images, you end up with several images. In the case of optical satellite, you often have a panchromatic spectral band with the highest spatial resolution and a multispectral product of the same area with a lower resolution. The resolution ratio is likely to be around 4.

To get the best of the image processing algorithms, you want to combine these data to produce a new image with the highest spatial resolution and several spectral band. This step is called fusion and you can find more details about it in 12. However, the fusion suppose that your two images represents exactly the same area. There are different solutions to process your data to reach this situation. Here we are going to use the metadata available with the images to produce an orthorectification as detailled in 10.

First you need to add the following lines in the `CMakeLists.txt` file:

```
ADD_EXECUTABLE(OrthoFusion  OrthoFusion.cxx)
TARGET_LINK_LIBRARIES(OrthoFusion OTBProjections OTBCommon OTBIO)
```

The source code for this example can be found in the file
`Examples/Tutorials/OrthoFusion.cxx`.

Start by including some necessary headers and with the usual `main` declaration. Apart from the classical header related to image input and output. We need the headers related to the fusion and the orthorectification. One header is also required to be able to process vector images (the XS one) with the orthorectification.

```
#include "otbImage.h"
```

```
#include "otbVectorImage.h"
#include "otbImageFileReader.h"
#include "otbStreamingImageFileWriter.h"

#include "otbOrthoRectificationFilter.h"
#include "otbMapProjections.h"
#include "otbPerBandVectorImageFilter.h"

#include "otbSimpleRcsPanSharpeningFusionImageFilter.h"
#include "otbStandardFilterWatcher.h"

int main( int argc, char* argv[] )
{
```

We initialize ossim which is required for the orthorectification and we check that all parameters are provided. Basically, we need:

- the name of the input PAN image;
- the name of the input XS image;
- the desired name for the output;
- as the coordinates are given in UTM, we need the UTM zone number;
- of course, we need the UTM coordinates of the final image;
- the size in pixels of the final image;
- and the sampling of the final image.

We check that all those parameters are provided.

```
  if(argc!=12)
  {
  std::cout << argv[0] <<" <input_pan_filename> <input_xs_filename> ";
  std::cout << "<output_filename> <utm zone> <hemisphere N/S> <x_ground_upper_left_corner> ";
  std::cout << "<y_ground_upper_left_corner> <x_Size> <y_Size> ";
  std::cout << "<x_groundSamplingDistance> ";
  std::cout << "<y_groundSamplingDistance (should be negative since origin is upper left)>"
        << std::endl;

    return EXIT_FAILURE;
  }
```

We declare the different images, readers and writer:

```
typedef otb::Image<unsigned int, 2>     ImageType;
typedef otb::VectorImage<unsigned int, 2>     VectorImageType;
typedef otb::Image<double, 2>     DoubleImageType;
typedef otb::VectorImage<double, 2>     DoubleVectorImageType;
typedef otb::ImageFileReader<ImageType>  ReaderType;
typedef otb::ImageFileReader<VectorImageType>  VectorReaderType;
typedef otb::StreamingImageFileWriter<VectorImageType>  WriterType;


ReaderType::Pointer      readerPAN=ReaderType::New();
VectorReaderType::Pointer     readerXS=VectorReaderType::New();
WriterType::Pointer      writer=WriterType::New();

readerPAN->SetFileName(argv[1]);
readerXS->SetFileName(argv[2]);
writer->SetFileName(argv[3]);
```

We declare the projection (here we chose the UTM projection, other choices are possible) and retrieve the paremeters from the command line:

- the UTM zone
- the hemisphere

```
typedef otb::UtmInverseProjection utmMapProjectionType ;
utmMapProjectionType::Pointer utmMapProjection =
    utmMapProjectionType::New();
utmMapProjection->SetZone(atoi(argv[4]));
utmMapProjection->SetHemisphere(*(argv[5]));
```

We will need to pass several parameters to the orthorectification concerning the desired output region:

```
ImageType::IndexType start;
start[0]=0;
start[1]=0;

ImageType::SizeType size;
size[0]=atoi(argv[8]);
size[1]=atoi(argv[9]);

ImageType::SpacingType spacing;
spacing[0]=atof(argv[10]);
```

```
spacing[1]=atof(argv[11]);

ImageType::PointType origin;
origin[0]=strtod(argv[6], NULL);
origin[1]=strtod(argv[7], NULL);
```

We declare the orthorectification filter. And provide the different parameters:

```
typedef otb::OrthoRectificationFilter<ImageType, DoubleImageType,
utmMapProjectionType> OrthoRectifFilterType ;

OrthoRectifFilterType::Pointer  orthoRectifPAN =
    OrthoRectifFilterType::New();
orthoRectifPAN->SetMapProjection(utmMapProjection);

orthoRectifPAN->SetInput(readerPAN->GetOutput());

orthoRectifPAN->SetOutputStartIndex(start);
orthoRectifPAN->SetSize(size);
orthoRectifPAN->SetOutputSpacing(spacing);
orthoRectifPAN->SetOutputOrigin(origin);
```

Now we are able to have the orthorectified area from the PAN image. We just have to follow a similar process for the XS image. However, the `otb::OrthoRectificationFilter` is designed to work with one band images. To be able to process the XS image (which is a `otb::VectorImage`), we need to use the `otb::PerBandVectorImageFilter` which is going to apply the filter set via the method `SetFilter()` to all spectral bands.

```
typedef otb::PerBandVectorImageFilter<VectorImageType,
  DoubleVectorImageType, OrthoRectifFilterType> VectorOrthoRectifFilterType;


OrthoRectifFilterType::Pointer  orthoRectifXS =
    OrthoRectifFilterType::New();
VectorOrthoRectifFilterType::Pointer  orthoRectifXSVector =
    VectorOrthoRectifFilterType::New();
orthoRectifXSVector->SetFilter(orthoRectifXS);
```

This is the only difference, the rest of the parameters are provided as before:

```
orthoRectifXS->SetMapProjection(utmMapProjection);

orthoRectifXSVector->SetInput(readerXS->GetOutput());

orthoRectifXS->SetOutputStartIndex(start);
orthoRectifXS->SetSize(size);
```

```
  orthoRectifXS->SetOutputSpacing(spacing);
  orthoRectifXS->SetOutputOrigin(origin);
```

It's time to declare the fusion filter and set its inputs:

```
  typedef otb::SimpleRcsPanSharpeningFusionImageFilter
      <DoubleImageType,DoubleVectorImageType,VectorImageType> FusionFilterType;
  FusionFilterType::Pointer fusion = FusionFilterType::New();
  fusion->SetPanInput(orthoRectifPAN->GetOutput());
  fusion->SetXsInput(orthoRectifXSVector->GetOutput());
```

And we can plug it to the writer. To be able to process the images by tiles, we use the `SetTilingStreamDivisions()` method of the writer. We trigger the pipeline execution with the `Update()` method.

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

  writer->SetTilingStreamDivisions();

  otb::StandardFilterWatcher watcher(writer, "OrthoFusion");

  writer->Update();

  return EXIT_SUCCESS;

}
```

Part III

User’s guide

CHAPTER

FIVE

Data Representation

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

5.1 Image

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

5.1.1 Creating an Image

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

This example illustrates how to manually construct an `otb::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 "otbImage.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 2D image, which is what we often use in remote sensing applications, anyway, with `unsigned short` pixel data.

```
  typedef otb::Image< unsigned short, 2 > 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 OTB, 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 OTB 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 OTB, manually creating an image requires that the image is instantiated as previously shown, and that regions describing the image are then associated with it.

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

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

```
ImageType::IndexType start;

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

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

```
ImageType::SizeType  size;

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

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

```
ImageType::RegionType region;

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

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

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

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

5.1.2 Reading an Image from a File

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

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

```
#include "otbImageFileReader.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 = 2;

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

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

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

The minimum information required by the reader is the filename of the image to be loaded in memory. This is provided through the `SetFileName()` method. The file format here is inferred from the filename extension. The user may also explicitly specify the data format explicitly using the `itk::ImageIO` (See Chapter 6.1 95 for more information):

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

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

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

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

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

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

5.1.3 Accessing Pixel Data

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

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

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

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

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

```
ImageType::IndexType pixelIndex;

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

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.

5.1.4 Defining Origin and Spacing

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

Even though OTB can be used to perform general image processing tasks, the primary purpose of the toolkit is the processing of remote sensing image data. In that respect, additional information about the images is considered mandatory. In particular the information associated with

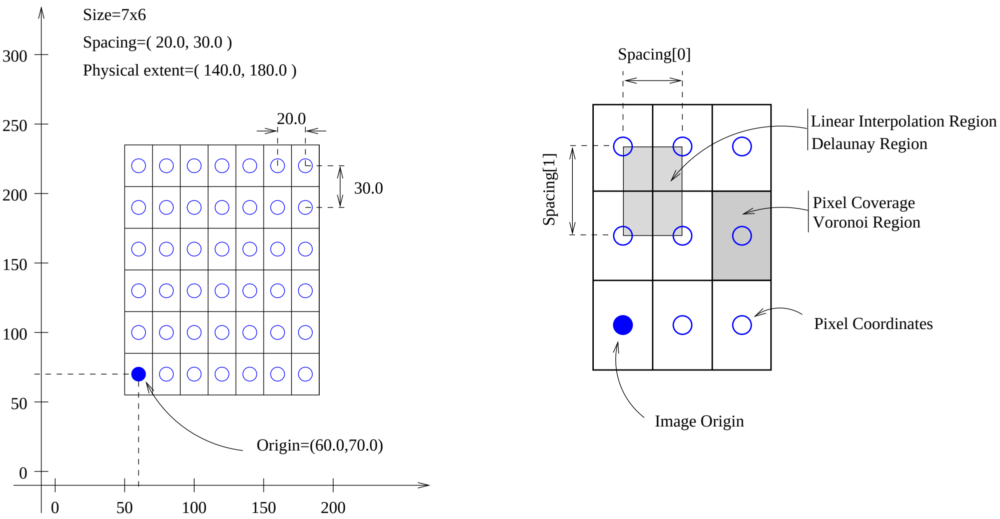


Figure 5.1: Geometrical concepts associated with the OTB image.

the physical spacing between pixels and the position of the image in space with respect to some world coordinate system are extremely important.

Image origin and spacing are fundamental to many applications. Registration, for example, is performed in physical coordinates. Improperly defined spacing and origins will result in inconsistent results in such processes. Remote sensing images with no spatial information should not be used for image analysis, feature extraction, GIS input, etc. In other words, remote sensing images lacking spatial information are not only useless but also hazardous.

Figure 5.1 illustrates the main geometrical concepts associated with the `otb::Image`. In this figure, circles are used to represent the center of pixels. The value of the pixel is assumed to exist as a Dirac Delta Function located at the pixel center. Pixel spacing is measured between the pixel centers and can be different along each dimension. The image origin is associated with the coordinates of the first pixel in the image. A *pixel* is considered to be the rectangular region surrounding the pixel center holding the data value. This can be viewed as the Voronoi region of the image grid, as illustrated in the right side of the figure. Linear interpolation of image values is performed inside the Delaunay region whose corners are pixel centers.

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

```
ImageType::SpacingType spacing;
```

```
// Note: measurement units (e.g., meters, feet, etc.) are defined by the application.
spacing[0] = 0.70; // spacing along X
spacing[1] = 0.70; // spacing along Y
```

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] << std::endl;
```

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

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

```
ImageType::PointType origin;

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


image->SetOrigin( origin );
```

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

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

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

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

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

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

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

```
PointType point;

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

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

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

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

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

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

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

```
pixelValue += 5;

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

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

5.1.5 Accessing Image Metadata

The source code for this example can be found in the file
`Examples/IO/MetadataExample.cxx`.

This example illustrates the access to metadata image information with OTB. By metadata, we mean data which is typically stored with remote sensing images, like geographical coordinates of pixels, pixel spacing or resolution, etc. Of course, the availability of these data depends on the image format used and on the fact that the image producer must fill the available metadata fields. The image formats which typically support metadata are for example CEOS and GeoTiff.

The metadata support is embedded in OTB's IO functionnalities and is accessible through the `otb::Image` and `otb::VectorImage` classes. You should avoid using the `itk::Image` class if you want to have metadata support.

This simple example will consist on reading an image from a file and writing the metadata to an output ASCII file. As usual we start by defining the types needed for the image to be read.

```
typedef unsigned char                           InputPixelType;
const   unsigned int                                Dimension = 2;


typedef otb::Image< InputPixelType,  Dimension >      InputImageType;

typedef otb::ImageFileReader< InputImageType >        ReaderType;
```

We can now instantiate the reader and get a pointer to the input image.

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

reader->SetFileName( inputFilename  );
reader->Update();

image = reader->GetOutput();
```

Once the image has been read, we can access the metadata information. We will copy this information to an ASCII file, so we create an output file stream for this purpose.

```
std::ofstream file;

file.open(outputAsciiFilename);
```

We can now call the different available methods for accessing the metadata. Useful methods are :

- `GetSpacing`: the sampling step;
- `GetOrigin`: the coordinates of the origin of the image;
- `GetProjectionRef`: the image projection reference;
- `GetGCPProjection`: the projection for the eventual ground control points;
- `GetGCPCount`: the number of GCPs available;

```
file << "Spacing " << image->GetSpacing() << std::endl;
file << "Origin " << image->GetOrigin() << std::endl;

file << "Projection REF " << image->GetProjectionRef() << std::endl;

file << "GCP Projection " << image->GetGCPProjection() << std::endl;

unsigned int GCPCount = image->GetGCPCount();
file << "GCP Count " << image->GetGCPCount() << std::endl;
```

One can also get the GCPs by number, as well as their coordinates in image and geographical space.

```
for(unsigned int GCPnum = 0 ; GCPnum < GCPCount ; GCPnum++ )
  {
  file << "GCP[" << GCPnum << "] Id " << image->GetGCPId(GCPnum) << std::endl;
  file << "GCP[" << GCPnum << "] Info " << image->GetGCPInfo(GCPnum) << std::endl;
  file << "GCP[" << GCPnum << "] Row " << image->GetGCPRow(GCPnum) << std::endl;
  file << "GCP[" << GCPnum << "] Col " << image->GetGCPCol(GCPnum) << std::endl;
  file << "GCP[" << GCPnum << "] X " << image->GetGCPX(GCPnum) << std::endl;
  file << "GCP[" << GCPnum << "] Y " << image->GetGCPY(GCPnum) << std::endl;
  file << "GCP[" << GCPnum << "] Z " << image->GetGCPZ(GCPnum) << std::endl;
  file << "----------------" << std::endl;
  }
```

If a geographical transformation is available, it can be recovered as follows.

```
InputImageType::VectorType tab = image->GetGeoTransform();

file << "Geo Transform " << std::endl;
for(unsigned int i = 0 ; i < tab.size() ; i++ )
  {
  file << " " <<i<<" -> "<<tab[i]<< std::endl;
  }
tab.clear();

tab = image->GetUpperLeftCorner();
file << "Corners " << std::endl;
for(unsigned int i = 0 ; i < tab.size() ; i++ )
  {
  file << " UL[" <<i<<"] -> "<<tab[i]<< std::endl;
  }
tab.clear();

tab = image->GetUpperRightCorner();
for(unsigned int i = 0 ; i < tab.size() ; i++ )
  {
  file << " UR[" <<i<<"] -> "<<tab[i]<< std::endl;
  }
tab.clear();

tab = image->GetLowerLeftCorner();
for(unsigned int i = 0 ; i < tab.size() ; i++ )
  {
  file << " LL[" <<i<<"] -> "<<tab[i]<< std::endl;
  }
tab.clear();

tab = image->GetLowerRightCorner();
for(unsigned int i = 0 ; i < tab.size() ; i++ )
  {
  file << " LR[" <<i<<"] -> "<<tab[i]<< std::endl;
  }
tab.clear();

file.close();
```

5.1.6 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 [62, 99]. The human retina possess different types of light sensitive cells. Three of them, known as *cones*, are sensitive to color [36] and their regions of sensitivity loosely match regions of the spectrum that will be perceived as red, green and blue respectively. The *rods* on the other hand provide no color discrimination and favor high resolution and high sensitivity[1]. A fifth type of receptors, the *ganglion cells*, also known as circadian[2] receptors are sensitive to the lighting conditions that differentiate day from night. These receptors evolved as a mechanism for synchronizing the physiology with the time of the day. Cellular controls for circadian rythms are present in every cell of an organism and are known to be exquisitively precise [58].

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

$$ColorA = ColorB - ColorC \tag{5.1}$$

just as a way of saying that we can produce *ColorB* by combining *ColorA* and *ColorC*. However, we must be aware that (at least in emitted light) it is not possible to *substract light*. So when we mention Equation 5.1 we actually mean

$$ColorB = ColorA + ColorC \tag{5.2}$$

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 [99].

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 [99].

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

When you anticipate to perform the operation of "Mean" on a RGB type you are assuming that in the color space provides the action of finding a color in the middle of two colors, can be found

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

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

by using a linear operation between their numerical representation. This is unfortunately not the case in color spaces due to the fact that they are based on a human physiological response [62].

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 OTB.

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

Thanks to the flexibility offered by the Generic Programming style on which OTB 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 otb::Image< PixelType, 2 >   ImageType;
```

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

```
  typedef otb::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;
```

5.1.7 Vector Images

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

Many image processing tasks require images of non-scalar pixel type. A typical example is a multispectral image. The following code illustrates how to instantiate and use an image whose pixels are of vector type.

We could 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.

However, the `itk::Vector` is a fixed size array and it assumes that the number of channels of the image is known at compile time. Therefore, we prefer to use the `otb::VectorImage` class which allows to choose the number of channels of the image at runtime. The pixels will be of type `itk::VariableLengthVector`.

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

```
#include "otbVectorImage.h"
```

The VectorImage 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 to represent Pléiades images which have 4 bands.

```
typedef unsigned char                 PixelType;
```

```
typedef otb::VectorImage< PixelType, 2 >    ImageType;
```

Since the pixel dimensionality is choosen at runtime, one has to pass this parameter to the image before memory allocation.

```
image->SetNumberOfComponentsPerPixel(4);
image->Allocate();
```

The VariableLengthVector class overloads the operator `[]`. This makes it possible to access the Vector's components using index notation. The user must not forget to allocate the memory for each individual pixel by using the `Reserve` method.

```
ImageType::PixelType   pixelValue;
pixelValue.Reserve( 4 );

pixelValue[0] =  1;   // Blue component
pixelValue[1] =  6;   // Red component
pixelValue[2] =  100; // Green component
pixelValue[3] =  100; // NIR 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  );
```

The GetPixel method can also be used to read Vectors pixels from the image

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

Lets repeat that both `SetPixel()` and `GetPixel()` are inefficient and should only be used for debugging purposes or for implementing interactions with a graphical user interface such as querying pixel value by clicking with the mouse.

5.1.8 Importing Image Data from a Buffer

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

This example illustrates how to import data into the `otb::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 `otb::ImportImageFilter`, thereby producing an Image as output.

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

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

```
#include "otbImage.h"
#include "otbImportImageFilter.h"
```

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

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

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

```
  typedef otb::ImportImageFilter< ImageType >   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

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

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

  importFilter->SetRegion( region );
```

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

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

  importFilter->SetOrigin( origin );
```

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

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

  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.

```
// MODIFIED
  const unsigned int numberOfPixels =  size[0] * size[1];
  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 otb does not use `for()` loops in its internal code to access pixels. All pixel access tasks are instead performed using `otb::ImageIterator`s that support the management of n-dimensional images.

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

  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 ;
      *it++ = ( d2 < radius2 ) ? 255 : 0;
      }
    }
```

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

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

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

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

```
writer->SetInput(  dynamic_cast<ImageType*>(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.

5.1.9 Image Lists

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

This example illustrates the use of the `otb::ImageList::`class. This class provides the functionnalities needed in order to integrate image lists as data objects into the OTB pipeline. Indeed, if a `std::list< ImageType >` was used, the update operations on the pipeline might not have the desired effects.

In this example, we will only present the basic operations which can be applied on an `otb::ImageList::`object.

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

```
#include "otbImageList.h"
```

As usual, we start by defining the types for the pixel and image types, as well as those for the readers and writers.

```
    const unsigned int Dimension = 2;
```

```
typedef unsigned char InputPixelType;
typedef otb::Image< InputPixelType, Dimension > InputImageType;
typedef otb::ImageFileReader< InputImageType > ReaderType;
typedef otb::ImageFileWriter< InputImageType > WriterType;
```

We can now define the type for the image list. The `otb::ImageList::`class is templated over the type of image contained in it. This means that all images in a list must have the same type.

```
typedef otb::ImageList< InputImageType > ImageListType;
```

Let us assume now that we want to read an image from a file and store it in a list. The first thing to do is to instantiate the reader and set the image file name. We effectively read the image by calling the `Update()`.

```
ReaderType::Pointer reader = ReaderType::New();
reader->SetFileName(inputFilename);
reader->Update();
```

We create an image list by using the `New()` method.

```
ImageListType::Pointer imageList = ImageListType::New();
```

In order to store the image in the list, the `PushBack()` method is used.

```
imageList->PushBack(reader->GetOutput());
```

We could repeat this operation for other readers or the outputs of filters. We will now write an image of the list to a file. We therefore instantiate a writer, set the image file name and set the input image for it. This is done by calling the `Back()` method of the list, which allows us to get the last element.

```
// Getting the image from the list and writing it to file
WriterType::Pointer writer = WriterType::New();
writer->SetFileName(outputFilename);
```

```
writer->SetInput(imageList->Back());
writer->Update();
```

Other useful methods are:

- `SetNthElement()` and `GetNthElement()` allow to randomly access any element of the list.
- `Front()` to access to the first element of the list.
- `Erase()` to remove an element.

Also, iterator classes are defined in order to have an efficient mean of moving through the list. Finally, the `otb::ImageListToImageListFilter::`is provided in order to implement filter which operate on image lists and produce image lists.

5.2 PointSet

5.2.1 Creating a PointSet

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

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

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

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

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

```
typedef itk::PointSet< unsigned short, 2 > PointSetType;
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.2.2 Getting Access to Points

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
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.

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

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

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

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

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

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

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

5.2.3 Getting Access to Data in Points

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

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

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

```
typedef unsigned short  PixelType;
typedef itk::PointSet< PixelType, 2 > 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 `PointDataContainer`s. 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.

5.2.4 Vectors as Pixel Type

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

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

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

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

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

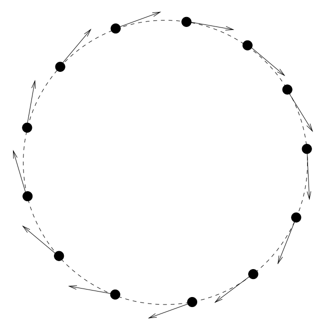


Figure 5.2: Vectors as PixelType.

```
  const unsigned int Dimension = 2;
  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 circle 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 5.2.

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

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

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

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

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

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

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

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

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

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

```
pointIterator = pointSet->GetPoints()->Begin();
```

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

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

5.3 Mesh

5.3.1 Creating a Mesh

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

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

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

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

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

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

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

```
  typedef   float   PixelType;
```

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

```
const unsigned int Dimension = 2;
typedef itk::Mesh< PixelType, Dimension >   MeshType;
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.3.2 Inserting Cells

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

A `itk::Mesh` can contain a variety of cell types. Typical cells are the `itk::LineCell`, `itk::TriangleCell`, `itk::QuadrilateralCell` and `itk::TetrahedronCell`. The latter will not be used very often in the remote sensing context. Additional flexibility is provided for managing cells at the price of a bit more of complexity than in the case of point management.

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

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

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

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

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

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

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

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

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

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

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

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

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

MeshType::PointType p0;
MeshType::PointType p1;
MeshType::PointType p2;

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

mesh->SetPoint( 0, p0 );
```

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

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

```
CellAutoPointer line0;
CellAutoPointer line1;

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

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

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

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

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

```
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.

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

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

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

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

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

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

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

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

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

5.3.3 Managing Data in Cells

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

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

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

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

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

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

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

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

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

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

  typedef MeshType::PointType PointType;
  PointType point;

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

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

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

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

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

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

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

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

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

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

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

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

```
while( cellDataIterator != end )
  {
```

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

More details about the use of itk::Mesh can be found in the ITK Software Guide.

5.4 Path

5.4.1 Creating a PolyLineParametricPath

The source code for this example can be found in the file
Examples/DataRepresentation/Path/PolyLineParametricPath1.cxx.

This example illustrates how to use the itk::PolyLineParametricPath. This class will typically be used for representing in a concise way the output of an image segmentation algorithm in 2D. See section 13.3 for an example in the context of alignment detection. 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.

```
const unsigned int Dimension = 2;

typedef otb::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 );
```

CHAPTER

SIX

Reading and Writing Images

This chapter describes the toolkit architecture supporting reading and writing of images to files. OTB does not enforce any particular file format, instead, it provides a structure inherited from ITK, 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.

6.1 Basic Example

The source code for this example can be found in the file
`Examples/IO/ImageReadWrite.cxx`.

The classes responsible for reading and writing images are located at the beginning and end of the data processing pipeline. These classes are known as data sources (readers) and data sinks (writers). Generally speaking they are referred to as filters, although readers have no pipeline input and writers have no pipeline output.

The reading of images is managed by the class `otb::ImageFileReader` while writing is performed by the class `otb::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`. Actually, the OTB image Readers and Writers are very similar to those of ITK, but provide new functionnalities which are specific to remote sensing images.

The first step for performing reading and writing is to include the following headers.

```
#include "otbImageFileReader.h"
#include "otbImageFileWriter.h"
```

Then, as usual, a decision must be made about the type of pixel used to represent the image processed by the pipeline. Note that when reading and writing images, the pixel type of the image **is not necessarily** the same as the pixel type stored in the file. Your choice of the pixel type (and hence template parameter) should be driven mainly by two considerations:

- It should be possible to cast the file pixel type in the file to the pixel type you select. This casting will be performed using the standard C-language rules, so you will have to make sure that the conversion does not result in information being lost.
- The pixel type in memory should be appropriate to the type of processing you intended to apply on the images.

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

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

Note that the dimension of the image in memory should match the one of the image in file. There are a couple of special cases in which this condition may be relaxed, but in general it is better to ensure that both dimensions match. This is not a real issue in remote sensing, unless you want to consider multi-band images as volumes (3D) of data.

We can now instantiate the types of the reader and writer. These two classes are parameterized over the image type.

```
typedef otb::ImageFileReader< ImageType >  ReaderType;
typedef otb::ImageFileWriter< ImageType >  WriterType;
```

Then, we create one object of each type using the New() method and assigning the result to a `itk::SmartPointer`.

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

The name of the file to be read or written is passed with the SetFileName() method.

```
reader->SetFileName( inputFilename  );
writer->SetFileName( outputFilename );
```

We can now connect these readers and writers to filters to create a pipeline. For example, we can create a short pipeline by passing the output of the reader directly to the input of the writer.

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

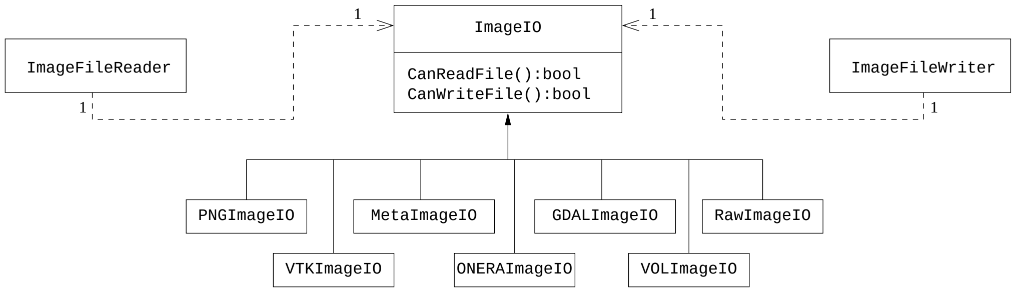


Figure 6.1: Collaboration diagram of the ImageIO classes.

At first view, this may seem as a quite useless program, but it is actually implementing a powerful file format conversion tool! The execution of the pipeline is triggered by the invocation of the `Update()` methods in one of the final objects. In this case, the final data pipeline object is the writer. It is a wise practice of defensive programming to insert any `Update()` call inside a `try/catch` block in case exceptions are thrown during the execution of the pipeline.

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

Note that exceptions should only be caught by pieces of code that know what to do with them. In a typical application this `catch` block should probably reside on the GUI code. The action on the `catch` block could inform the user about the failure of the IO operation.

The IO architecture of the toolkit makes it possible to avoid explicit specification of the file format used to read or write images.[1] The object factory mechanism enables the ImageFileReader and ImageFileWriter to determine (at run-time) with which file format it is working with. Typically, file formats are chosen based on the filename extension, but the architecture supports arbitrarily complex processes to determine whether a file can be read or written. Alternatively, the user can specify the data file format by explicit instantiation and assignment the appropriate `itk::ImageIO` subclass.

To better understand the IO architecture, please refer to Figures 6.1, 6.2, and 6.3.

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

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

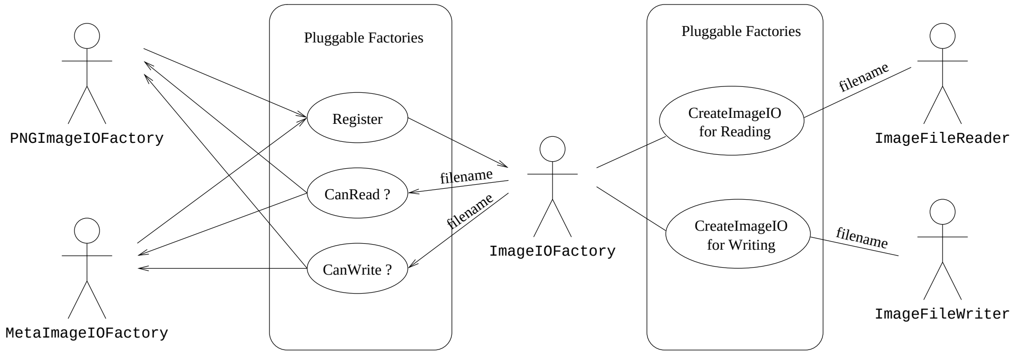


Figure 6.2: Use cases of ImageIO factories.

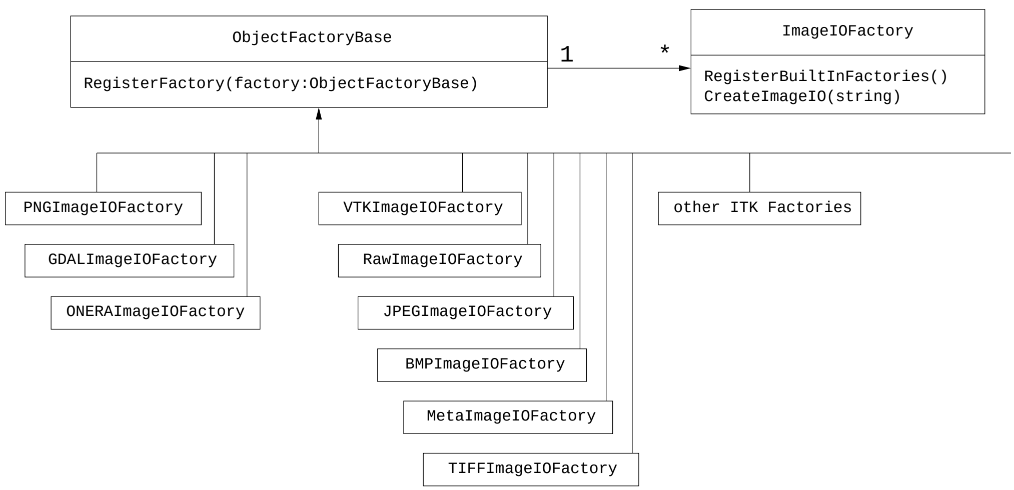


Figure 6.3: Class diagram of the ImageIO factories.

6.2 Pluggable Factories

The principle behind the input/output mechanism used in ITK and therefore OTB is known as *pluggable-factories* [33]. This concept is illustrated in the UML diagram in Figure 6.1. From the user's point of view the objects responsible for reading and writing files are the `otb::ImageFileReader` and `otb::ImageFileWriter` classes. These two classes, however, are not aware of the details involved in reading or writing particular file formats like PNG or GeoTIFF. What they do is to dispatch the user's requests to a set of specific classes that are aware of the details of image file formats. These classes are the `itk::ImageIO` classes. The ITK delegation mechanism enables users to extend the number of supported file formats by just adding new classes to the ImageIO hierarchy.

Each instance of ImageFileReader and ImageFileWriter has a pointer to an ImageIO object. If this pointer is empty, it will be impossible to read or write an image and the image file reader/writer must determine which ImageIO class to use to perform IO operations. This is done basically by passing the filename to a centralized class, the `itk::ImageIOFactory` and asking it to identify any subclass of ImageIO capable of reading or writing the user-specified file. This is illustrated by the use cases on the right side of Figure 6.2. The ImageIOFactory acts here as a dispatcher that help to 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 6.3.

For example, in order to read PNG files, a PNGImageIOFactory is created and registered with the central ImageIOFactory singleton[2] class as illustrated in the left side of Figure 6.2. When the ImageFileReader asks the ImageIOFactory for an ImageIO capable of reading the file identified with *filename* the ImageIOFactory will iterate over the list of registered factories and will ask each one of them is they know how to read the file. The factory that responds affirmatively will be used to create the specific ImageIO instance that will be returned to the ImageFileReader and used to perform the read operations.

With respect to the ITK formats, OTB adds most of the remote sensing image formats. In order to do so, the Geospatial Data Abstraction Library, GDAL `http://www.gdal.org/`, is encapsultated in a ImageIO factory. GDAL is a translator library for raster geospatial data formats that is released under an X/MIT style Open Source license. As a library, it presents a single abstract data model to the calling application for all supported formats, which include CEOS, GeoTIFF, ENVI, and much more. See `http://www.gdal.org/formats_list.html` for the full format list.

Since GDAL is itself a multi-format library, the GDAL IO factory is able to choose the appro-

[2] *Singleton* means that there is only one instance of this class in a particular application

priate ressource for reading and writing images.

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. Please see the ITK Software for more details about this.

6.3 IO Streaming

6.3.1 Implicit Streaming

The source code for this example can be found in the file
`Examples/IO/StreamingImageReadWrite.cxx`.

As we have seen, the reading of images is managed by the class `otb::ImageFileReader` while writing is performed by the class `otb::ImageFileWriter`. ITK's pipeline implements streaming. That means that a filter for which the `ThreadedGenerateData` method is implemented, will only produce the data for the region requested by the following filter in the pipeline. Therefore, in order to use the streaming functionnality one needs to use a filter at the end of the pipeline which requests for adjacent regions of the image to be processed. In ITK, the `itk::StreamingImageFilter` class is used for this purpose. However, ITK does not implement streaming from/to files. This means that even if the pipeline has a small memory footprint, the images have to be stored in memory at least after the read operation and before the write operation.

OTB implements read/write streaming. For the image file reading, this is transparent for the programmer, and if a streaming loop is used at the end of the pipeline, the read operation will be streamed. For the file writing, the `otb::StreamingImageFileWriter` has to be used.

The first step for performing streamed reading and writing is to include the following headers.

```
#include "otbImageFileReader.h"
#include "otbStreamingImageFileWriter.h"
```

Then, as usual, a decision must be made about the type of pixel used to represent the image processed by the pipeline.

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

We can now instantiate the types of the reader and writer. These two classes are parameterized over the image type. We will rescale the intensities of the as an example of intermediate processing step.

```
typedef otb::ImageFileReader< ImageType >  ReaderType;
typedef itk::RescaleIntensityImageFilter< ImageType, ImageType> RescalerType;
typedef otb::StreamingImageFileWriter< ImageType >  WriterType;
```

Then, we create one object of each type using the New() method and assigning the result to a `itk::SmartPointer`.

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

The name of the file to be read or written is passed with the SetFileName() method. We also choose the range of intensities for the rescaler.

```
reader->SetFileName( inputFilename  );
rescaler->SetOutputMinimum(0);
rescaler->SetOutputMaximum(255);
writer->SetFileName( outputFilename );
```

We can now connect these readers and writers to filters to create a pipeline.

```
rescaler->SetInput( reader->GetOutput() );
writer->SetInput( rescaler->GetOutput() );
```

We can now trigger the pipeline execution by calling the `Update` method on the writer.

```
  writer->Update();
```

The writer will ask its preceding filter to provide different portions of the image. Each filter in the pipeline will do the same until the request arrives to the reader. In this way, the pipeline will be executed for each requested region and the whole input image will be read, processed and written without being fully loaded in memory.

6.3.2 Explicit Streaming

The source code for this example can be found in the file
`Examples/IO/ExplicitStreamingExample.cxx`.

Usually, the streaming process is hidden within the pipeline. This allows the user to get rid of the annoying task of splitting the images into tiles, and so on. However, for some kinds of processing, we do not really need a pipeline: no writer is needed, only read access to pixel values is wanted. In these cases, one has to explicitly set up the streaming procedure. Fortunately, OTB offers a high level of abstraction for this task. We will need to include the following header files:

```
#include "itkImageRegionSplitter.h"
#include "otbStreamingTraits.h"
```

The `otb::StreamingTraits` class manages the streaming approaches which are possible with the image type over which it is templated. The class `itk::ImageRegionSplitter` is templated over the number of dimensions of the image and will perform the actual image splitting. More information on splitter can be found in section 21.3

```
  typedef otb::StreamingTraits<ImageType> StreamingTraitsType;
  typedef itk::ImageRegionSplitter<2>  SplitterType;
```

Once a region of the image is available, we will use classical region iterators to get the pixels.

```
  typedef ImageType::RegionType RegionType;

  typedef itk::ImageRegionConstIterator<ImageType> IteratorType;
```

We instantiate the image file reader, but in order to avoid reading the whole image, we call the `GenerateOutputInformation()` method instead of the `Update()` one. `GenerateOutputInformation()` will make available the information about sizes, band, resolutions, etc. After that, we can access the largest possible region of the input image.

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

  reader->SetFileName(infname);


  reader->GenerateOutputInformation();

  RegionType largestRegion = reader->GetOutput()->GetLargestPossibleRegion();
```

We set up now the local streaming capabilities by asking the streaming traits to compute the number of regions to split the image into given the splitter, the user defined number of lines, and the input image information.

```
  SplitterType::Pointer splitter = SplitterType::New();
  unsigned int numberOfStreamDivisions =
    StreamingTraitsType::CalculateNumberOfStreamDivisions(
      reader->GetOutput(),
      largestRegion,
      splitter,
      otb::SET_BUFFER_NUMBER_OF_LINES,
      0,0,nbLinesForStreaming);
```

We can now get the split regions and iterate through them.

```
unsigned int piece = 0;
RegionType streamingRegion;


for (piece = 0;
     piece < numberOfStreamDivisions;
     piece++)
{
```

We get the region

```
  streamingRegion =
    splitter->GetSplit(piece,numberOfStreamDivisions,largestRegion);

  std::cout<<"Processing region: "<<streamingRegion<<std::endl;
```

We ask the reader to provide the region.

```
  reader->GetOutput()->SetRequestedRegion(streamingRegion);
  reader->GetOutput()->PropagateRequestedRegion();
  reader->GetOutput()->UpdateOutputData();
```

We declare an iterator and walk through the region.

```
  IteratorType it(reader->GetOutput(),streamingRegion);
  it.GoToBegin();

  while(!it.IsAtEnd())
  {
    std::cout << it.Get() << std::endl;
    ++it;
  }
```

6.4 Reading and Writing RGB Images

The source code for this example can be found in the file
`Examples/IO/RGBImageReadWrite.cxx`.

RGB images are commonly used for representing data acquired from multispectral sensors. 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 "otbImage.h"
#include "otbImageFileReader.h"
#include "otbImageFileWriter.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 otb::Image< PixelType, 2 >       ImageType;
```

The image type is used as a template parameter to instantiate the reader and writer.

```
  typedef otb::ImageFileReader< ImageType >  ReaderType;
  typedef otb::ImageFileWriter< ImageType >  WriterType;

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

The filenames of the input and output files must be provided to the reader and writer respectively.

```
  reader->SetFileName( inputFilename  );
  writer->SetFileName( outputFilename );
```

Finally, execution of the pipeline can be triggered by invoking the `Update()` method in the writer.

```
  writer->Update();
```

You may have noticed that apart from the declaration of the `PixelType` there is nothing in this code that is specific for RGB images. All the actions required to support color images are implemented internally in the itk::ImageIO objects.

6.5 Reading, Casting and Writing Images

The source code for this example can be found in the file
`Examples/IO/ImageReadCastWrite.cxx`.

Given that ITK and OTB are based on the Generic Programming paradigm, most of the types are defined at compilation time. It is sometimes important to anticipate conversion between different types of images. The following example illustrates the common case of reading an image of one pixel type and writing it on a different pixel type. This process not only involves casting but also rescaling the image intensity since the dynamic range of the input and output pixel types can be quite different. The `itk::RescaleIntensityImageFilter` is used here to linearly rescale the image values.

The first step in this example is to include the appropriate headers.

```
#include "otbImageFileReader.h"
#include "otbImageFileWriter.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 otb::Image< InputPixelType,  Dimension >    InputImageType;
  typedef otb::Image< OutputPixelType, Dimension >    OutputImageType;
```

We can now instantiate the types of the reader and writer. These two classes are parameterized over the image type.

```
  typedef otb::ImageFileReader< InputImageType  >  ReaderType;
  typedef otb::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;
  }
```

6.6 Extracting Regions

The source code for this example can be found in the file
`Examples/IO/ImageReadRegionOfInterestWrite.cxx`.

This example should arguably be placed in the filtering chapter. However its usefulness for typical IO operations makes it interesting to mention here. The purpose of this example is to read and image, extract a subregion and write this subregion to a file. This is a common task when we want to apply a computationally intensive method to the region of interest of an image.

As usual with OTB IO, we begin by including the appropriate header files.

```
#include "otbImageFileReader.h"
#include "otbImageFileWriter.h"
```

The `otb::ExtractROI` is the filter used to extract a region from an image. Its header is included below.

```
#include "otbExtractROI.h"
```

Image types are defined below.

```
typedef unsigned char        InputPixelType;
typedef unsigned char        OutputPixelType;
const   unsigned int        Dimension = 2;
typedef otb::Image< InputPixelType,  Dimension >    InputImageType;
typedef otb::Image< OutputPixelType, Dimension >    OutputImageType;
```

The types for the `otb::ImageFileReader` and `otb::ImageFileWriter` are instantiated using the image types.

```
typedef otb::ImageFileReader< InputImageType  >  ReaderType;
typedef otb::ImageFileWriter< OutputImageType >  WriterType;
```

The ExtractROI type is instantiated using the input and output pixel types. Using the pixel types as template parameters instead of the image types allows to restrict the use of this class to `otb::Images` which are used with scalar pixel types. See section 6.8.1 for the extraction of ROIs on `otb::VectorImages`. A filter object is created with the New() method and assigned to a `itk::SmartPointer`.

```
typedef otb::ExtractROI< InputImageType::PixelType,
                         OutputImageType::PixelType > FilterType;

FilterType::Pointer filter = FilterType::New();
```

The ExtractROI requires a region to be defined by the user. This is done by defining a rectangle with the following methods (the filter assumes that a 2D image is being processed, for N-D region extraction, you can use the `itk::RegionOfInterestImageFilter` class).

```
filter->SetStartX( atoi( argv[3] ) );
filter->SetStartY( atoi( argv[4] ) );
filter->SetSizeX( atoi( argv[5] ) );
filter->SetSizeY( atoi( argv[6] ) );
```

Below, we create the reader and writer using the New() method and assigning the result to a SmartPointer.

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

The name of the file to be read or written is passed with the SetFileName() method.

```
reader->SetFileName( inputFilename  );
writer->SetFileName( outputFilename );
```

Below we connect the reader, filter and writer to form the data processing pipeline.

```
filter->SetInput( reader->GetOutput() );
writer->SetInput( filter->GetOutput() );
```

Finally we execute the pipeline by invoking Update() on the writer. The call is placed in a `try/catch` block in case exceptions are thrown.

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

6.7 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. One of the uses of these tye of images is the processing of SLC SAR images, which are complex.

6.7.1 Reading and Writing Complex Images

The source code for this example can be found in the file
`Examples/IO/ComplexImageReadWrite.cxx`.

This example illustrates how to read and write an image of pixel type `std::complex`. The complex type is defined as an integral part of the C++ language.

We start by including the headers of the complex class, the image, and the reader and writer classes.

```
#include <complex>
#include "otbImage.h"
#include "otbImageFileReader.h"
#include "otbImageFileWriter.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 otb::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 otb::ImageFileReader< ImageType > ReaderType;
typedef otb::ImageFileWriter< ImageType > WriterType;

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

Filenames should be provided for both the reader and the writer. In this particular example we take those filenames from the command line arguments.

```
reader->SetFileName( argv[1] );
writer->SetFileName( argv[2] );
```

Here we simply connect the output of the reader as input to the writer. This simple program could be used for converting complex images from one fileformat to another.

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

The execution of this short pipeline is triggered by invoking the Update() method of the writer. This invocation must be placed inside a try/catch block since its execution may result in exceptions being thrown.

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

For a more interesting use of this code, you may want to add a filter in between the reader and the writer and perform any complex image to complex image operation.

6.8 Reading and Writing Multiband Images

The source code for this example can be found in the file
`Examples/IO/MultibandImageReadWrite.cxx`.

The `otb::Image` class with a vector pixel type could be used for representing multispectral images, with one band per vector component, however, this is not a practical way, since the dimensionality of the vector must be known at compile time. OTB offers the `otb::VectorImage` where the dimensionality of the vector stored for each pixel can be chosen at runtime. This is needed for the image file readers in order to dynamically set the number of bands of an image read from a file.

The OTB Readers and Writers are able to deal with `otb::VectorImage`s transparently for the user.

The first step for performing reading and writing is to include the following headers.

```
#include "otbImageFileReader.h"
#include "otbImageFileWriter.h"
```

Then, as usual, a decision must be made about the type of pixel used to represent the image processed by the pipeline. The pixel type corresponds to the scalar type stored in the vector components. Therefore, for a multiband Pléiades image we will do:

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

We can now instantiate the types of the reader and writer. These two classes are parameterized over the image type.

```
  typedef otb::ImageFileReader< ImageType >  ReaderType;
  typedef otb::ImageFileWriter< ImageType >  WriterType;
```

Then, we create one object of each type using the New() method and assigning the result to a `itk::SmartPointer`.

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

The name of the file to be read or written is passed with the SetFileName() method.

```
  reader->SetFileName( inputFilename  );
  writer->SetFileName( outputFilename );
```

We can now connect these readers and writers to filters to create a pipeline. The only thig to take care of is, when executing the program, choosing an output image file format which supports multiband images.

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


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

6.8.1 Extracting ROIs

The source code for this example can be found in the file
`Examples/IO/ExtractROI.cxx`.

This example shows the use of the `otb::MultiChannelExtractROI` and `otb::MultiToMonoChannelExtractROI` which allow the extraction of ROIs from multiband images stored into `otb::VectorImage`s. The first one povides a Vector Image as output, while the second one provides a classical `otb::Image` with a scalar pixel type. The present example shows how to extract a ROI from a 4-band SPOT 5 image and to produce a first multi-band 3-channel image and a second mono-channel one for the SWIR band.

We start by including the needed header files.

```
#include "otbImageFileReader.h"
#include "otbImageFileWriter.h"
#include "otbMultiChannelExtractROI.h"
#include "otbMultiToMonoChannelExtractROI.h"
```

The program arguments define the image file names as well as the rectangular area to be extracted.

```
  const char * inputFilename  = argv[1];
  const char * outputFilenameRGB = argv[2];
  const char * outputFilenameMIR = argv[3];

  unsigned int  startX((unsigned int)::atoi(argv[4]));
  unsigned int  startY((unsigned int)::atoi(argv[5]));
```

```
unsigned int  sizeX((unsigned int)::atoi(argv[6]));
unsigned int  sizeY((unsigned int)::atoi(argv[7]));
```

As usual, we define the input and output pixel types.

```
typedef   unsigned char  InputPixelType;
typedef   unsigned char  OutputPixelType;
```

First of all, we extract the multiband part by using the `otb::MultiChannelExtractROI` class, which is templated over the input and output pixel types. This class in not templated over the images types in order to force these images to be of `otb::VectorImage` type.

```
typedef otb::MultiChannelExtractROI< InputPixelType,
                                             OutputPixelType >  ExtractROIFilterType;
```

We create the extractor filter by using the `New` method of the class and we set its parameters.

```
ExtractROIFilterType::Pointer extractROIFilter = ExtractROIFilterType::New();

extractROIFilter->SetStartX( startX );
extractROIFilter->SetStartY( startY );
extractROIFilter->SetSizeX( sizeX );
extractROIFilter->SetSizeY( sizeY );
```

We must tell the filter which are the channels to be used. When selecting contiguous bands, we can use the `SetFirstChannel` and the `SetLastChannel`. Otherwise, we select individual channels by using the `SetChannel` method.

```
extractROIFilter->SetFirstChannel( 1 );
extractROIFilter->SetLastChannel( 3 );
```

We will use the OTB readers and writers for file access.

```
typedef otb::ImageFileReader< ExtractROIFilterType::InputImageType >       ReaderType;
typedef otb::ImageFileWriter< ExtractROIFilterType::InputImageType >      WriterType;

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

Since the number of bands of the input image is dynamically set at runtime, the `Update` method of the reader must be called before using the extractor filter.

```
reader->SetFileName( inputFilename  );
reader->Update();
writer->SetFileName( outputFilenameRGB );
```

We can then build the pipeline as usual.

```
extractROIFilter->SetInput( reader->GetOutput() );

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

And execute the pipeline by calling the `Update` method of the writer.

```
writer->Update();
```

The usage of the `otb::MultiToMonoChannelExtractROI` is similar to the one of the `otb::MultiChannelExtractROI` described above.

The goal now is to extract an ROI from a multi-band image and generate a mono-channel image as output.

We could use the `otb::MultiChannelExtractROI` and select a single channel, but using the `otb::MultiToMonoChannelExtractROI` we generate a `otb::Image` instead of an `otb::VectorImage`. This is useful from a computing and memory usage point of view. This class is also templated over the pixel types.

```
typedef otb::MultiToMonoChannelExtractROI< InputPixelType,
                                           OutputPixelType >  ExtractROIMonoFilterType;
```

For this filter, only one output channel has to be selected.

```
extractROIMonoFilter->SetChannel( 4 );
```

Figure 6.5 illustrates the result of the application of both extraction filters on the image presented in figure 6.4.

6.9 Reading Image Series

The source code for this example can be found in the file
`Examples/IO/ImageSeriesIOExample.cxx`.

This example shows how to read a list of images and concatenate them into a vector image. We will write a program which is able to perform this operation taking advantage of the streaming functionnalities of the processing pipeline. We will assume that all the input images have the same size and a single band.

The following header files will be needed:

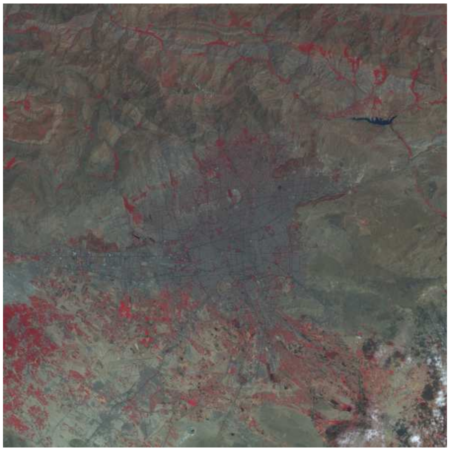


Figure 6.4: Quicklook of the original SPOT 5 image.

Figure 6.5: Result of the extraction. Left: 3-channel image. Right: mono-band image.

```
#include "otbImage.h"
#include "otbVectorImage.h"
#include "otbImageFileReader.h"
#include "otbObjectList.h"
#include "otbImageList.h"
#include "otbImageListToVectorImageFilter.h"
#include "otbStreamingImageFileWriter.h"
```

We will start by defining the types for the input images and the associated readers.

```
  typedef unsigned short int PixelType;
  const unsigned int Dimension = 2;

  typedef otb::Image< PixelType, Dimension > InputImageType;

  typedef otb::ImageFileReader< InputImageType > ImageReaderType;
```

We will use a list of image file readers in order to open all the input images at once. For this, we use the `otb::ObjectList` object and we template it over the type of the readers.

```
  typedef otb::ObjectList< ImageReaderType > ReaderListType;

  ReaderListType::Pointer readerList = ReaderListType::New();
```

We will also build a list of input images in order to store the smart pointers obtained at the output of each reader. This allows us to build a pipeline without really reading the images and using lots of RAM. The `otb::ImageList` object will be used.

```
  typedef otb::ImageList< InputImageType > ImageListType;

  ImageListType::Pointer imageList = ImageListType::New();
```

We can now loop over the input image list in order to populate the reader list and the input image list.

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

    ImageReaderType::Pointer imageReader = ImageReaderType::New();

    imageReader->SetFileName( argv[i+2]  );

    std::cout << "Adding image " << argv[i+2] << std::endl;
```

```
  imageReader->UpdateOutputInformation();

  imageList->PushBack( imageReader->GetOutput() );

  readerList->PushBack( imageReader );

  }
```

All the input images will be concatenated into a single output vector image. For this matter, we will use the `otb::ImageListToVectorImageFilter` which is templated over the input image list type and the output vector image type.

```
typedef otb::VectorImage< PixelType, Dimension > VectorImageType;

typedef otb::ImageListToVectorImageFilter< ImageListType, VectorImageType >
  ImageListToVectorImageFilterType;

ImageListToVectorImageFilterType::Pointer iL2VI =
  ImageListToVectorImageFilterType::New();
```

We plug the image list as input of the filter and use a `otb::StreamingImageFileWriter` to write the result image to a file, so that the streaming capabilities of all the readers and the filter are used.

```
iL2VI->SetInput( imageList );

typedef otb::StreamingImageFileWriter< VectorImageType > ImageWriterType;

ImageWriterType::Pointer imageWriter = ImageWriterType::New();

imageWriter->SetFileName( argv[1] );
```

We can tune the size of the image tiles as a function of the number of input images, so that the total memory footprint of the pipeline is constant for any execution of the program.

```
unsigned long size = (10000 * 10000 * sizeof(PixelType)) / NbImages;

std::cout<<"Streaming size: "<<size<<std::endl;

imageWriter->SetBufferMemorySize(size);
```

```
imageWriter->SetInput( iL2VI->GetOutput() );

imageWriter->Update();
```

6.10 Reading and Writing Vector Data

In Remote Sensing the use of vector data is common. Vector data is used to represent cartographic objects, segmentation results, etc. OTB provides functionnalities for accessing this kind of data.

6.10.1 Reading DXF Files

The source code for this example can be found in the file
`Examples/IO/DXFReaderExample.cxx`.

This example illustrates how to read a DXF file and how to draw objects on a $2D$ binary image. The graphical DXF objects which can be read are the following : Point, Line Polyline, Circle and 3DFace. The example begins by including the appropriate headers.

```
#include "itkExceptionObject.h"
#include "otbImage.h"
#include "otbImageFileWriter.h"
#include "otbSpatialObjectDXFReader.h"
#include "otbSpatialObjectToImageDrawingFilter.h"
#include "itkRescaleIntensityImageFilter.h"
```

Then, as usual, we select the pixel types and the image dimension.

```
const unsigned int Dimension = 2;
typedef double PixelType;
typedef unsigned char OutputPixelType;
```

The DXF file reader and the image file writer types are instantiated. We can then create objects for both of them. Graphical DXF objects will be represented in a GroupSpatialObject.

```
typedef itk::GroupSpatialObject<Dimension> GroupType;
typedef otb::Image<PixelType,Dimension> ImageType;
typedef otb::Image<OutputPixelType,Dimension> OutputImageType;
typedef otb::ImageFileWriter<OutputImageType> WriterType;
typedef otb::SpatialObjectDXFReader<GroupType>
                                SpatialObjectDXFReaderType;
typedef otb::SpatialObjectToImageDrawingFilter<GroupType,ImageType>
```

```
                    SpatialObjectToImageDrawingFilterType;
  typedef itk::RescaleIntensityImageFilter< ImageType,
                                  OutputImageType >     CastFilterType;
  typedef itk::SpatialObject<Dimension> SpatialObjectType;


  // Instantiating object
  SpatialObjectDXFReaderType::Pointer reader =
                              SpatialObjectDXFReaderType::New();
  SpatialObjectToImageDrawingFilterType::Pointer imageGenerator =
                   SpatialObjectToImageDrawingFilterType::New();
  WriterType::Pointer writer = WriterType::New();
  CastFilterType::Pointer castFilter = CastFilterType::New();
```

Filenames should be provided for both the reader and the writer. In this particular example we take those filenames from the command line arguments. The size of the output image is also specified. Thanks to the SetLayerName() method, a particular layer can be specified and other layers will not be read. If no layer name is specified, all layers are read.

```
  reader->SetFileName(inputFilename);
  reader->SetLayerName(argv[2]);
  writer->SetFileName(outputFilename);
  const unsigned int outputSize = atoi(argv[3]);
```

The reading of the DXF file is performed with the Update() method. Consequently the group of Spatial Objects is created.

```
  reader->Update();
  GroupType::Pointer group = reader->GetOutput();
```

We check if the group is empty. If it is not the case we will represent the Spatial Object group on the output image. To determine the minimum and maximum coordinates of the group we compute the bounding box of each element of the group.

```
  if(group->GetNumberOfChildren() != 0)
{
 /** Writing image  **/
SpatialObjectType::ChildrenListType* children=group->GetChildren(0);
SpatialObjectType::ChildrenListType::iterator it = children->begin();
SpatialObjectType::ChildrenListType::iterator end = children->end();
double maximum[Dimension],minimum[Dimension];
(*it)->ComputeBoundingBox();
minimum[0]=(*it)->GetBoundingBox()->GetMinimum()[0];
minimum[1]=(*it)->GetBoundingBox()->GetMinimum()[1];

while(it != end)
{
```

```
(*it)->ComputeBoundingBox();

if ((*it)->GetBoundingBox()->GetMinimum()[0] < minimum[0])
{
minimum[0]=(*it)->GetBoundingBox()->GetMinimum()[0];
}
if ((*it)->GetBoundingBox()->GetMinimum()[1] < minimum[1])
{
minimum[1]=(*it)->GetBoundingBox()->GetMinimum()[1];
}
it++;
}
```

Origin can be set at the minimum coordinate of the group and the spacing be adapted to the specified output image size in order to represent all Spatial Objects in the output image.

```
ImageType::SpacingType spacing;
spacing[0]=(maximum[0]-origin[0])/size[0];
spacing[1]=(maximum[1]-origin[1])/size[1];
imageGenerator->SetSpacing(spacing);
```

The output image is created with previously specified origin, spacing and size.

```
imageGenerator->SetInput(group);
imageGenerator->Update();
```

The output image is written by calling the Update() method.

```
writer->Update();
```

Figure 6.6 represents Spatial Objects extracted from a DXF file.

6.10.2 Reading and Writing Vector Data Files

The source code for this example can be found in the file
`Examples/IO/VectorDataIOExample.cxx`.

Although specific vector data import approaches, as the one presented in 6.10.1, can be useful, it is even more interesting to have available approaches which are independent of the input format. Unfortunately, many vector data formats do not share the models for the data they represent. However, in some cases, when simple data is stored, it can be decomposed in simple objects as for instance polylines, polygons and points. This is the case for the Shapefile and the KML (Keyhole Markup Language), for instance.

Even though specific reader/writer for Shapefile (and soon KML) are available in OTB, we designed a generic approach for the IO of this kind of data.

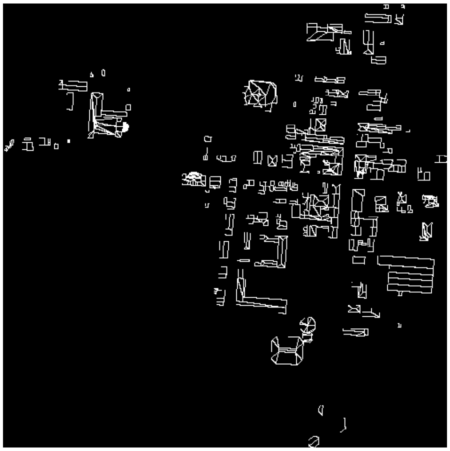


Figure 6.6: Representation of a DXF file on an image.

This example illustrates the use of OTB's vector data IO framework.

We will start by including the header files for the classes describing the vector data and the corresponding reader and writer.

```
#include "otbVectorData.h"
#include "otbVectorDataFileReader.h"
#include "otbVectorDataFileWriter.h"
```

We will also need to include the header files for the classes which model the individual objects that we get from the vector data structure.

```
#include "itkPreOrderTreeIterator.h"
#include "otbObjectList.h"
#include "otbPolygon.h"
```

We define the types for the vector data structure and the corresponding file reader.

```
  typedef otb::VectorData<PixelType,2>            VectorDataType;

  typedef otb::VectorDataFileReader<VectorDataType>
      VectorDataFileReaderType;
```

We can now instantiate the reader and read the data.

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

```
reader->SetFileName(argv[1]);
reader->Update();
```

The vector data obtained from the reader wil provide a tree of nodes containing the actual objects of the scene. This tree will be accessed using an `itk::PreOrderTreeIterator`.

```
typedef VectorDataType::DataTreeType      DataTreeType;
typedef itk::PreOrderTreeIterator<DataTreeType>    TreeIteratorType;
```

In this example we will only read polygon objects from the input file before writing them to the output file. We define the type for the polygon object as well as an iterator to the vertices. The polygons obtained will be stored in an `otb::ObjectList`.

```
typedef otb::Polygon<PixelType> PolygonType;
typedef PolygonType::VertexListIteratorType PolygonIteratorType;
typedef otb::ObjectList<PolygonType> PolygonListType;

typedef PolygonListType::Iterator PolygonListIteratorType;

PolygonListType::Pointer polygonList = PolygonListType::New();
```

We get the data tree and instantiate an iterator to walk through it.

```
TreeIteratorType it(reader->GetOutput()->GetDataTree());

it.GoToBegin();
```

We check that the current object is a polygon using the `IsPolygonFeature()` method and get its exterior ring in order to sore it into the list.

```
while(!it.IsAtEnd())
{
  if(it.Get()->IsPolygonFeature())
  {
    polygonList->PushBack(it.Get()->GetPolygonExteriorRing());
  }
  ++it;
}
```

Before writing the polygons to the output file, we have to build the vector data structure. This structure will be build up of nodes. We define the types needed for that.

```
VectorDataType::Pointer outVectorData = VectorDataType::New();

typedef VectorDataType::DataNodeType              DataNodeType;
```

We fill the data structure with the nodes. The root node is a document which is composed of folders. A list of polygons can be seen as a multi polygon object.

```
  DataNodeType::Pointer document = DataNodeType::New();
  document->SetNodeType(otb::DOCUMENT);
  document->SetNodeId("polygon");
  DataNodeType::Pointer folder = DataNodeType::New();
  folder->SetNodeType(otb::FOLDER);
  DataNodeType::Pointer multiPolygon = DataNodeType::New();
  multiPolygon->SetNodeType(otb::FEATURE_MULTIPOLYGON);
```

We assign these objects to the data tree stored by the vector data object.

```
  DataTreeType::Pointer tree = outVectorData->GetDataTree();
  DataNodeType::Pointer root = tree->GetRoot()->Get();

  tree->Add(document,root);
  tree->Add(folder,document);
  tree->Add(multiPolygon,folder);
```

We can now iterate through the polygon list and fill the vector data structure.

```
  for(PolygonListType::Iterator it = polygonList->Begin();
      it != polygonList->End(); ++it)
  {
    DataNodeType::Pointer newPolygon = DataNodeType::New();
    newPolygon->SetPolygonExteriorRing(it.Get());
    tree->Add(newPolygon,multiPolygon);
  }
```

An finally we write the vector data to a file using a generic `otb::VectorDataFileWriter`.

```
typedef otb::VectorDataFileWriter<VectorDataType> WriterType;

WriterType::Pointer writer = WriterType::New();
writer->SetFileName(argv[2]);
writer->Update();
```

6.11 Reading DEM Files

The source code for this example can be found in the file
`Examples/IO/DEMToImageGenerator.cxx`.

The following example illustrates the use of the otb::DEMToImageGenerator class. The aim of this class is to generate an image from the srtm data (precising the start extraction latitude and longitude point). Each pixel is a geographic point and its intensity is the altitude of the point. If srtm doesn't have altitude information for a point, the altitude value is set at -32768 (value of the srtm norm).

Let's look at the minimal code required to use this algorithm. First, the following header defining the `otb::DEMToImageGenerator` class must be included.

```
#include "otbDEMToImageGenerator.h"
```

The image type is now defined using pixel type and dimension. The output image is defined as an `otb::Image`.

```
const unsigned int Dimension = 2;
typedef otb::Image<double , Dimension>           ImageType;
```

The DEMToImageGenerator is defined using the image pixel type as a template parameter. After that, the object can be instancied.

```
typedef otb::DEMToImageGenerator<ImageType>      DEMToImageGeneratorType;

DEMToImageGeneratorType::Pointer object = DEMToImageGeneratorType::New();
```

Input parameter types are defined to set the value in the DEMToImageGenerator.

```
typedef DEMToImageGeneratorType::SizeType        SizeType;
typedef DEMToImageGeneratorType::SpacingType     SpacingType;
typedef DEMToImageGeneratorType::DEMHandlerType  DEMHandlerType;
typedef DEMHandlerType::PointType                PointType;
```

The path to the DEM folder is given to the filter.

```
object->SetDEMDirectoryPath(folderPath);
```

The origin (Longitude/Latitude) of the output image in the DEM is given to the filter.

```
PointType origin;
origin[0] = ::atof(argv[3]);
origin[1] = ::atof(argv[4]);

object->SetOutputOrigin(origin);
```

The size (in Pixel) of the output image is given to the filter.

```
SizeType size;
size[0] = ::atoi(argv[5]);
size[1] = ::atoi(argv[6]);

object->SetOutputSize(size);
```

The spacing (step between to consecutive pixel) is given to the filter. By default, this spacing is set at 0.001.

```
SpacingType spacing;
spacing[0] = ::atof(argv[7]);
spacing[1] = ::atof(argv[8]);

object->SetOutputSpacing(spacing);
```

The output image name is given to the writer and the filter output is linked to the writer input.

```
writer->SetFileName( outputName );

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

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();
 }
```

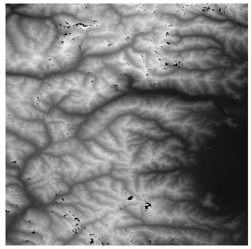


Figure 6.7: DEMToImageGenerator image.

```
catch( itk::ExceptionObject & err )
  {
    std::cout << "Exception itk::ExceptionObject thrown !" << std::endl;
    std::cout << err << std::endl;
  return EXIT_FAILURE;
  }
```

Let's now run this example using as input the SRTM data contained in `DEM_srtm` folder. Figure 6.7 shows the obtained DEM. Invalid data values – hidden areas due to SAR shadowing – are set to zero.

CHAPTER

SEVEN

Basic Filtering

This chapter introduces the most commonly used filters found in OTB. 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. OTB is based ITK's data pipeline architecture in which the output of one filter is passed as input to another filter. (See Section 3.5 on page 28 for more information.)

7.1 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 OTB.

7.1.1 Binary Thresholding

The source code for this example can be found in the file
`Examples/Filtering/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 7.1. 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.

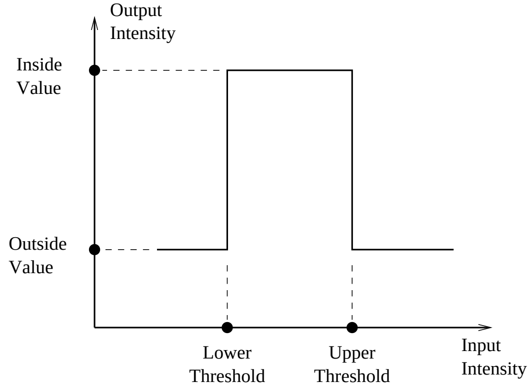


Figure 7.1: Transfer function of the BinaryThresholdImageFilter.

The first step required to use the `itk::BinaryThresholdImageFilter` is to include its header file.

```
#include "itkBinaryThresholdImageFilter.h"
```

The next step is to decide which pixel types to use for the input and output images.

```
  typedef  unsigned char  InputPixelType;
  typedef  unsigned char  OutputPixelType;
```

The input and output image types are now defined using their respective pixel types and dimensions.

```
  typedef otb::Image< InputPixelType,  2 >   InputImageType;
  typedef otb::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 `otb::ImageFileReader` class is also instantiated in order to read image data from a file. (See Section 6 on page 95 for more information about reading and writing data.)

```
  typedef otb::ImageFileReader< InputImageType >  ReaderType;
```

An `otb::ImageFileWriter` is instantiated in order to write the output image to a file.

```
typedef otb::ImageFileWriter< InputImageType >  WriterType;
```

Both the filter and the reader are created by invoking their `New()` methods and assigning the result to `itk::SmartPointers`.

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

The image obtained with the reader is passed as input to the BinaryThresholdImageFilter.

```
filter->SetInput( reader->GetOutput() );
```

The method `SetOutsideValue()` defines the intensity value to be assigned to those pixels whose intensities are outside the range defined by the lower and upper thresholds. The method `SetInsideValue()` defines the intensity value to be assigned to pixels with intensities falling inside the threshold range.

```
filter->SetOutsideValue( outsideValue );
filter->SetInsideValue(  insideValue  );
```

The methods `SetLowerThreshold()` and `SetUpperThreshold()` define the range of the input image intensities that will be transformed into the `InsideValue`. Note that the lower and upper thresholds are values of the type of the input image pixels, while the inside and outside values are of the type of the output image pixels.

```
filter->SetLowerThreshold( lowerThreshold );
filter->SetUpperThreshold( upperThreshold );
```

The execution of the filter is triggered by invoking the `Update()` method. If the filter's output has been passed as input to subsequent filters, the `Update()` call on any posterior filters in the pipeline will indirectly trigger the update of this filter.

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

Figure 7.2 illustrates the effect of this filter on a ROI of a Spot 5 image of an agricultural area. 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.

The following classes provide similar functionality:

- `itk::ThresholdImageFilter`

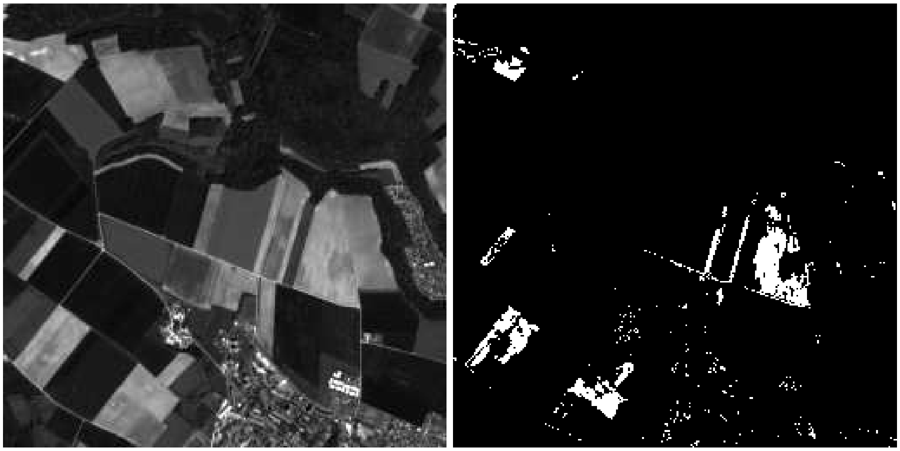


Figure 7.2: Effect of the BinaryThresholdImageFilter on a ROI of a Spot 5 image.

7.1.2 General Thresholding

The source code for this example can be found in the file
`Examples/Filtering/ThresholdImageFilter.cxx`.

This example illustrates the use of the `itk::ThresholdImageFilter`. This filter can be used to transform the intensity levels of an image in three different ways.

- First, the user can define a single threshold. Any pixels with values below this threshold will be replaced by a user defined value, called here the `OutsideValue`. Pixels with values above the threshold remain unchanged. This type of thresholding is illustrated in Figure 7.3.
- 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 7.4.
- 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 7.5.

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

- `ThresholdBelow()`
- `ThresholdAbove()`
- `ThresholdOutside()`

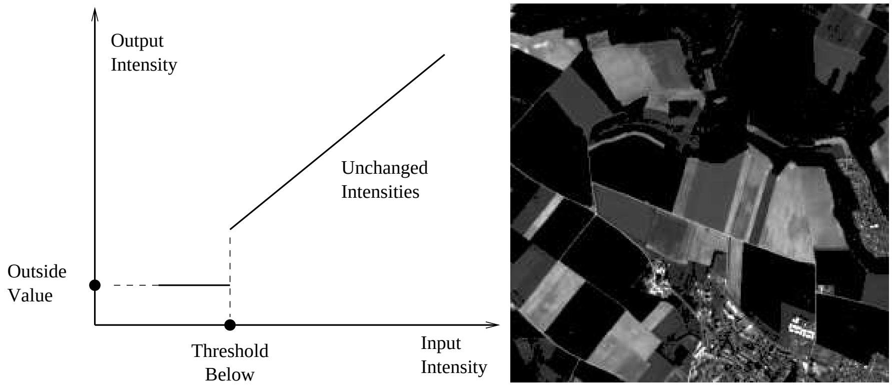


Figure 7.3: ThresholdImageFilter using the threshold-below mode.

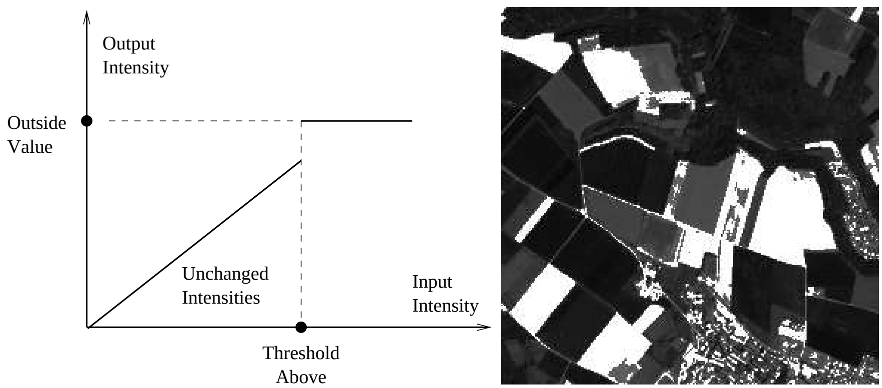


Figure 7.4: ThresholdImageFilter using the threshold-above mode.

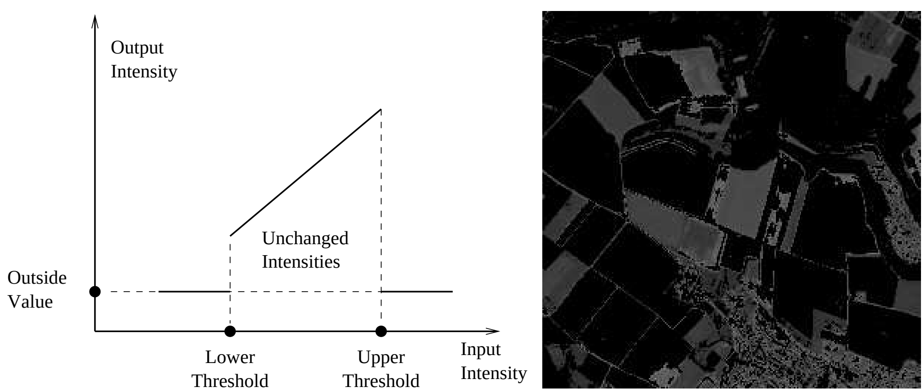


Figure 7.5: ThresholdImageFilter using the threshold-outside mode.

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 otb::Image< PixelType,  2 >   ImageType;
```

The filter can be instantiated using the image type defined above.

```
  typedef itk::ThresholdImageFilter< ImageType >  FilterType;
```

An `otb::ImageFileReader` class is also instantiated in order to read image data from a file.

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

An `otb::ImageFileWriter` is instantiated in order to write the output image to a file.

```
  typedef otb::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( 40 );
```

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 7.3. The second operating mode of the filter is now enabled by calling the method ThresholdAbove().

```
filter->ThresholdAbove( 100 );
filter->SetOutsideValue( 255 );
filter->Update();
```

Updating the filter with this new setting produces the output shown in Figure 7.4. The third operating mode of the filter is enabled by calling ThresholdOutside().

```
filter->ThresholdOutside( 40,100 );
filter->SetOutsideValue( 0 );
filter->Update();
```

The output of this third, "band-pass" thresholding mode is shown in Figure 7.5.

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.

The following classes provide similar functionality:

- itk::BinaryThresholdImageFilter

7.1.3 Threshold to Point Set

The source code for this example can be found in the file
Examples/FeatureExtraction/ThresholdToPointSetExample.cxx.

Sometimes, it may be more valuable not to get an image from the threshold step but rather a list of coordinates. This can be done with the otb::ThresholdImageToPointSetFilter.

The following example illustrates the use of the `otb::ThresholdImageToPointSetFilter` which provide a list of points within given thresholds. Points set are described in section 5.2 on page 76.

The first step required to use this filter is to include the header

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

The next step is to decide which pixel types to use for the input image and the Point Set as well as their dimension.

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

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

A reader is instanciated to read the input image

```
  typedef otb::ImageFileReader< ImageType > ReaderType;
  ReaderType::Pointer reader = ReaderType::New();

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

We get the parameters from the command line for the threshold filter. The lower and upper thresholds parameters are similar to those of the `itk::BinaryThresholdImageFilter` (see Section 7.1.1 on page 127 for more information).

```
  int lowerThreshold = atoi(argv[2]);
  int upperThreshold = atoi(argv[3]);
```

Then we create the ThresholdImageToPointSetFilter and we pass the parameters.

```
  typedef otb::ThresholdImageToPointSetFilter
      < ImageType, PointSetType > FilterThresholdType;
  FilterThresholdType::Pointer filterThreshold = FilterThresholdType::New();
  filterThreshold->SetLowerThreshold( lowerThreshold );
  filterThreshold->SetUpperThreshold( upperThreshold );
  filterThreshold->SetInput(0, reader->GetOutput() );
```

To manipulate and display the result of this filter, we manually instanciate a point set and we call the `Update()` method on the threshold filter to trigger the pipeline execution.

After this step, the `pointSet` variable contains the point set.

```
PointSetType::Pointer pointSet     = PointSetType::New();
pointSet = filterThreshold->GetOutput();

filterThreshold->Update();
```

To display each point, we create an iterator on the list of points, which is accessible through the method `GetPoints()` of the PointSet.

```
typedef PointSetType::PointsContainer ContainerType;
ContainerType* pointsContainer = pointSet->GetPoints();
typedef ContainerType::Iterator IteratorType;
IteratorType itList = pointsContainer->Begin();
```

A while loop enable us to through the list a display the coordinate of each point.

```
while( itList != pointsContainer->End() )
{
  std::cout << itList.Value() << std::endl;
  ++itList;
}
```

7.2 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.

7.2.1 Gradient Magnitude

The source code for this example can be found in the file
`Examples/Filtering/GradientMagnitudeImageFilter.cxx`.

The magnitude of the image gradient is extensively used in image analysis, mainly to help in the determination of object contours and the separation of homogeneous regions. The `itk::GradientMagnitudeImageFilter` computes the magnitude of the image gradient at each pixel location using a simple finite differences approach. For example, in the case of $2D$ the computation is equivalent to convolving the image with masks of type

-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 otb::Image< InputPixelType,  2 >   InputImageType;
  typedef otb::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();
```

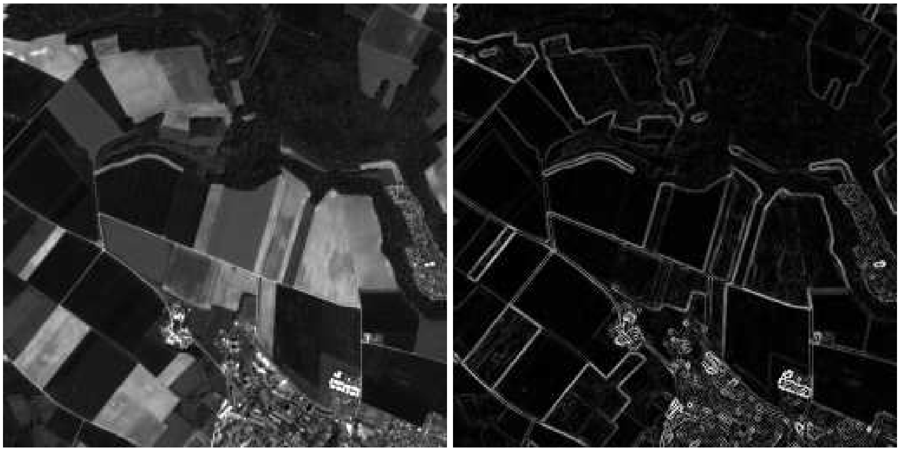


Figure 7.6: Effect of the GradientMagnitudeImageFilter.

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 7.6 illustrates the effect of the gradient magnitude. 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, images of man-made objects that contain high contrast objects.

This filter does not apply any smoothing to the image before computing the gradients. The results can therefore be very sensitive to noise and may not be best choice for scale space analysis.

7.2.2 Gradient Magnitude With Smoothing

The source code for this example can be found in the file
`Examples/Filtering/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 IIR [1] filter that approximates a convolution with the derivative of the Gaussian kernel. Traditional convolution will produce a more accurate result, but the IIR approach is much faster, especially using large σs [22, 23].

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

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

```
#include "itkGradientMagnitudeRecursiveGaussianImageFilter.h"
```

Types should be instantiated based on the pixels of the input and output images.

```
  typedef    float    InputPixelType;
  typedef    float    OutputPixelType;
```

With them, the input and output image types can be instantiated.

```
  typedef otb::Image< InputPixelType,  2 >   InputImageType;
  typedef otb::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.

[1] Infinite Impulse Response

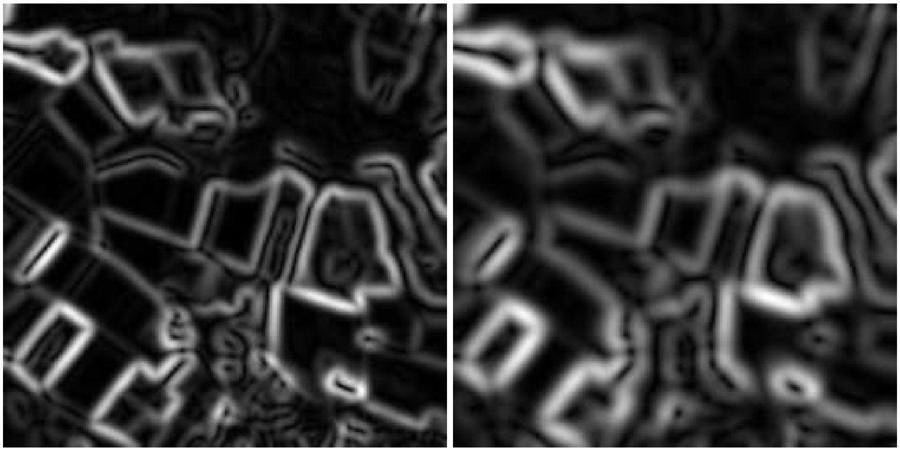


Figure 7.7: Effect of the GradientMagnitudeRecursiveGaussianImageFilter.

```
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 7.7 illustrates the effect of this filter using σ values of 3 (left) and 5 (right). The figure shows how the sensitivity to noise can be regulated by selecting an appropriate σ. This type of scale-tunable filter is suitable for performing scale-space analysis.

Attention should be paid to the image type chosen to represent the output image since the dynamic range of the gradient magnitude image is usually smaller than the dynamic range of the input image.

7.2.3 Derivative Without Smoothing

The source code for this example can be found in the file
`Examples/Filtering/DerivativeImageFilter.cxx`.

The `itk::DerivativeImageFilter` is used for computing the partial derivative of an image, the derivative of an image along a particular axial direction.

The header file corresponding to this filter should be included first.

```
#include "itkDerivativeImageFilter.h"
```

Next, the pixel types for the input and output images must be defined and, with them, the image types can be instantiated. Note that it is important to select a signed type for the image, since the values of the derivatives will be positive as well as negative.

```
  typedef   float  InputPixelType;
  typedef   float  OutputPixelType;

  const unsigned int Dimension = 2;

  typedef otb::Image< InputPixelType,  Dimension >   InputImageType;
  typedef otb::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 7.8 illustrates the effect of the DerivativeImageFilter. The derivative is taken along the *x* direction. The sensitivity to noise in the image is evident from this result.

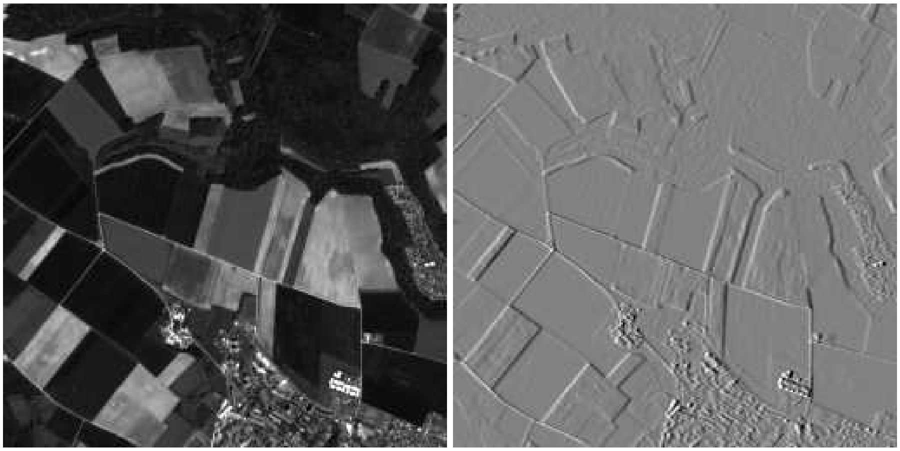


Figure 7.8: Effect of the Derivative filter.

7.3 Second Order Derivatives

7.3.1 Laplacian Filters

Laplacian Filter Recursive Gaussian

The source code for this example can be found in the file
`Examples/Filtering/LaplacianRecursiveGaussianImageFilter1.cxx`.

This example illustrates how to use the `itk::RecursiveGaussianImageFilter` for computing the Laplacian of an 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 otb::Image< InputPixelType,  2 >   InputImageType;
  typedef otb::Image< OutputPixelType, 2 >   OutputImageType;
```

The filter type is now instantiated using both the input image and the output image types.

```
typedef itk::RecursiveGaussianImageFilter<
                      InputImageType, OutputImageType >  FilterType;
```

This filter applies the approximation of the convolution along a single dimension. It is therefore necessary to concatenate several of these filters to produce smoothing in all directions. In this example, we create a pair of filters since we are processing a $2D$ image. The filters are created by invoking the `New()` method and assigning the result to a `itk::SmartPointer`.

We need two filters for computing the X component of the Laplacian and two other filters for computing the Y component.

```
FilterType::Pointer filterX1 = FilterType::New();
FilterType::Pointer filterY1 = FilterType::New();

FilterType::Pointer filterX2 = FilterType::New();
FilterType::Pointer filterY2 = FilterType::New();
```

Since each one of the newly created filters has the potential to perform filtering along any dimension, we have to restrict each one to a particular direction. This is done with the `SetDirection()` method.

```
filterX1->SetDirection( 0 );   // 0 --> X direction
filterY1->SetDirection( 1 );   // 1 --> Y direction

filterX2->SetDirection( 0 );   // 0 --> X direction
filterY2->SetDirection( 1 );   // 1 --> Y direction
```

The `itk::RecursiveGaussianImageFilter` can approximate the convolution with the Gaussian or with its first and second derivatives. We select one of these options by using the `SetOrder()` method. Note that the argument is an `enum` whose values can be `ZeroOrder`, `FirstOrder` and `SecondOrder`. For example, to compute the x partial derivative we should select `FirstOrder` for x and `ZeroOrder` for y. Here we want only to smooth in x and y, so we select `ZeroOrder` in both directions.

```
filterX1->SetOrder( FilterType::ZeroOrder );
filterY1->SetOrder( FilterType::SecondOrder );

filterX2->SetOrder( FilterType::SecondOrder );
filterY2->SetOrder( FilterType::ZeroOrder );
```

There are two typical ways of normalizing Gaussians depending on their application. For scale-space analysis it is desirable to use a normalization that will preserve the maximum value of the input. This normalization is represented by the following equation.

$$\frac{1}{\sigma\sqrt{2\pi}} \tag{7.1}$$

In applications that use the Gaussian as a solution of the diffusion equation it is desirable to use a normalization that preserve the integral of the signal. This last approach can be seen as a conservation of mass principle. This is represented by the following equation.

$$\frac{1}{\sigma^2\sqrt{2\pi}} \tag{7.2}$$

The `itk::RecursiveGaussianImageFilter` has a boolean flag that allows users to select between these two normalization options. Selection is done with the method `SetNormalizeAcrossScale()`. Enable this flag to analyzing an image across scale-space. In the current example, this setting has no impact because we are actually renormalizing the output to the dynamic range of the reader, so we simply disable the flag.

```
const bool normalizeAcrossScale = false;
filterX1->SetNormalizeAcrossScale( normalizeAcrossScale );
filterY1->SetNormalizeAcrossScale( normalizeAcrossScale );
filterX2->SetNormalizeAcrossScale( normalizeAcrossScale );
filterY2->SetNormalizeAcrossScale( normalizeAcrossScale );
```

The input image can be obtained from the output of another filter. Here, an image reader is used as the source. The image is passed to the x filter and then to the y filter. The reason for keeping these two filters separate is that it is usual in scale-space applications to compute not only the smoothing but also combinations of derivatives at different orders and smoothing. Some factorization is possible when separate filters are used to generate the intermediate results. Here this capability is less interesting, though, since we only want to smooth the image in all directions.

```
filterX1->SetInput( reader->GetOutput() );
filterY1->SetInput( filterX1->GetOutput() );

filterY2->SetInput( reader->GetOutput() );
filterX2->SetInput( filterY2->GetOutput() );
```

It is now time to select the σ of the Gaussian used to smooth the data. Note that σ must be passed to both filters and that sigma is considered to be in the units of the image spacing. 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 `otb::ImageFileWriter` class.

```
typedef  float WritePixelType;

typedef otb::Image< WritePixelType, 2 >    WriteImageType;

typedef otb::ImageFileWriter< WriteImageType >  WriterType;

WriterType::Pointer writer = WriterType::New();

writer->SetInput( addFilter->GetOutput() );

writer->SetFileName( argv[2] );

writer->Update();
```

Figure 7.9 illustrates the effect of this filter using σ values of 3 (left) and 5 (right). The figure shows how the attenuation of noise can be regulated by selecting the appropriate standard deviation. This type of scale-tunable filter is suitable for performing scale-space analysis.

The source code for this example can be found in the file
`Examples/Filtering/LaplacianRecursiveGaussianImageFilter2.cxx`.

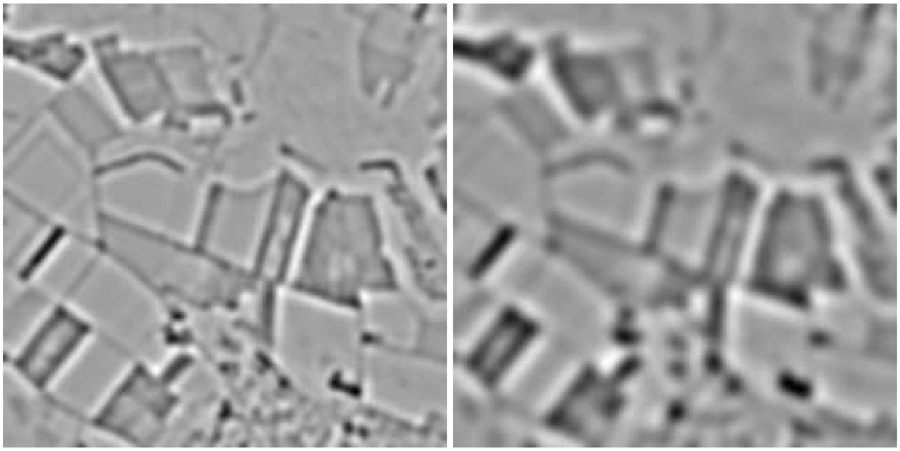


Figure 7.9: Effect of the LaplacianRecursiveGaussianImageFilter.

The previous exampled showed how to use the `itk::RecursiveGaussianImageFilter` for computing the equivalent of a Laplacian of an image after smoothing with a Gaussian. The elements used in this previous example have been packaged together in the `itk::LaplacianRecursiveGaussianImageFilter` in order to simplify its usage. This current example shows how to use this convenience filter for achieving the same results as the previous example.

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

```
#include "itkLaplacianRecursiveGaussianImageFilter.h"
```

Types should be selected on the desired input and output pixel types.

```
  typedef    float    InputPixelType;
  typedef    float    OutputPixelType;
```

The input and output image types are instantiated using the pixel types.

```
  typedef otb::Image< InputPixelType,  2 >   InputImageType;
  typedef otb::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 the units of the image spacing. That is, at the moment of applying the smoothing process, the filter will take into account the spacing values defined in the image.

```
laplacian->SetSigma( sigma );
```

Finally the pipeline is executed by invoking the `Update()` method.

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

Figure 7.10 illustrates the effect of this filter 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.

7.4 Edge Detection

7.4.1 Canny Edge Detection

The source code for this example can be found in the file
`Examples/Filtering/CannyEdgeDetectionImageFilter.cxx`.

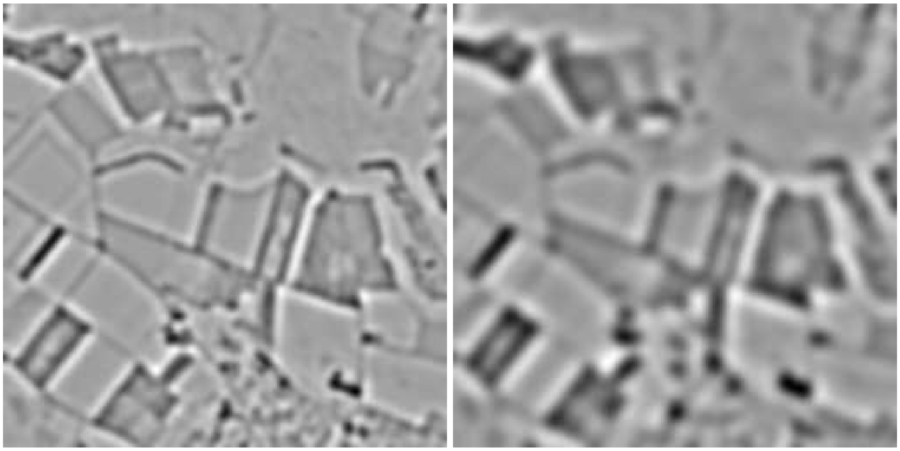


Figure 7.10: Effect of the LaplacianRecursiveGaussianImageFilter.

This example introduces the use of the `itk::CannyEdgeDetectionImageFilter`. This filter is widely used for edge detection since it is the optimal solution satisfying the constraints of good sensitivity, localization and noise robustness.

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

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

This filter operates on image of pixel type float. It is then necessary to cast the type of the input images that are usually of integer type. The `itk::CastImageFilter` is used here for that purpose. Its image template parameters are defined for casting from the input type to the float type using for processing.

```
  typedef itk::CastImageFilter< CharImageType, RealImageType> CastToRealFilterType;
```

The `itk::CannyEdgeDetectionImageFilter` is instantiated using the float image type.

Figure 7.11 illustrates the effect of this filter on a ROI of a Spot 5 image of an agricultural area.

7.4.2 Ratio of Means Detector

The source code for this example can be found in the file
`Examples/FeatureExtraction/TouziEdgeDetectorExample.cxx`.

This example illustrates the use of the `otb::TouziEdgeDetectorImageFilter`. This filter belongs to the family of the fixed false alarm rate edge detectors but it is apropriate for SAR images, where the speckle noise is considered as multiplicative. By analogy with the classical

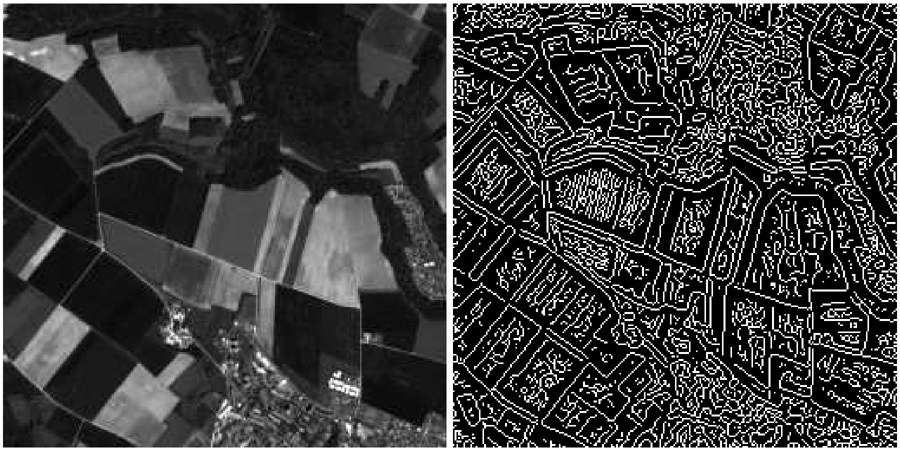


Figure 7.11: Effect of the CannyEdgeDetectorImageFilter on a ROI of a Spot 5 image.

gradient-based edge detectors which are suited to the additive noise case, this filter computes a ratio of local means in both sides of the edge [85]. In order to have a normalized response, the following computation is performed :

$$r = 1 - min\{\frac{\mu_A}{\mu_B}, \frac{\mu_B}{\mu_A}\}, \tag{7.3}$$

where μ_A and μ_B are the local means computed at both sides of the edge. In order to detect edges with any orientation, r is computed for the 4 principal directions and the maximum response is kept.

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

```
#include "otbTouziEdgeDetectorImageFilter.h"
```

Then we must decide what pixel type to use for the image. We choose to make all computations with floating point precision and rescale the results between 0 and 255 in order to export PNG images.

```
  typedef  float  InternalPixelType;
  typedef  unsigned char  OutputPixelType;
```

The images are defined using the pixel type and the dimension.

```
  typedef otb::Image< InternalPixelType,  2 >   InternalImageType;
  typedef otb::Image< OutputPixelType,  2 >   OutputImageType;
```

The filter can be instantiated using the image types defined above.

```
typedef otb::TouziEdgeDetectorImageFilter< InternalImageType, InternalImageType >  FilterTyp
```

An `ImageFileReader::` class is also instantiated in order to read image data from a file.

```
typedef otb::ImageFileReader< InternalImageType >  ReaderType;
```

An `ImageFileWriter::` is instantiated in order to write the output image to a file.

```
typedef otb::ImageFileWriter< OutputImageType >  WriterType;
```

The intensity rescaling of the results will be carried out by the `itk::RescaleIntensityImageFilter` which is templated by the input and output image types.

```
typedef itk::RescaleIntensityImageFilter< InternalImageType,
                                          OutputImageType > RescalerType;
```

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 same is done for the rescaler and the writer.

```
RescalerType::Pointer rescaler = RescalerType::New();
WriterType::Pointer writer = WriterType::New();
```

The `itk::RescaleIntensityImageFilter` needs to know which is the minimu and maximum values of the output generated image. Those can be chosen in a generic way by using the `NumericTraits` functions, since they are templated over the pixel type.

```
rescaler->SetOutputMinimum( itk::NumericTraits< OutputPixelType >::min());
rescaler->SetOutputMaximum( itk::NumericTraits< OutputPixelType >::max());
```

The image obtained with the reader is passed as input to the `otb::TouziEdgeDetectorImageFilter`. The pipeline is built as follows.

```
filter->SetInput( reader->GetOutput() );
rescaler->SetInput( filter->GetOutput() );
writer->SetInput( rescaler->GetOutput() );
```

Figure 7.12: Result of applying the otb::TouziEdgeDetectorImageFilter to a SAR image. From left to right : original image, edge intensity and edge orientation.

The method SetRadius() defines the size of the window to be used for the computation of the local means.

```
FilterType::SizeType Radius;
Radius[0]= atoi(argv[4]);
Radius[1]= atoi(argv[4]);

filter->SetRadius( Radius );
```

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();
```

We can also obtain the direction of the edges by invoking the GetOutputDirection() method.

```
rescaler->SetInput( filter->GetOutputDirection() );
writer->SetInput( rescaler->GetOutput() );
writer->Update();
```

Figure 7.12 shows the result of applying the Touzi edge detector filter to a SAR image.

7.5 Neighborhood Filters

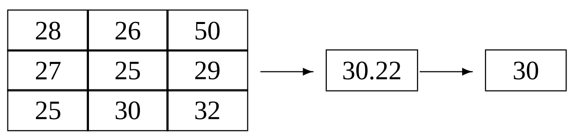
The concept of locality is frequently encountered in image processing in the form of filters that compute every output pixel using information from a small region in the neighborhood of the input pixel. The classical form of these filters are the 3×3 filters in 2D images. Convolution masks based on these neighborhoods can perform diverse tasks ranging from noise reduction, to differential operations, to mathematical morphology.

The Insight toolkit implements an elegant approach to neighborhood-based image filtering. The input image is processed using a special iterator called the `itk::NeighborhoodIterator`. This iterator is capable of moving over all the pixels in an image and, for each position, it can address the pixels in a local neighborhood. Operators are defined that apply an algorithmic operation in the neighborhood of the input pixel to produce a value for the output pixel. The following section describes some of the more commonly used filters that take advantage of this construction. (See Chapter 19 on page 471 for more information about iterators.)

7.5.1 Mean Filter

The source code for this example can be found in the file
`Examples/Filtering/MeanImageFilter.cxx`.

The `itk::MeanImageFilter` is commonly used for noise reduction. The filter computes the value of each output pixel by finding the statistical mean of the neighborhood of the corresponding input pixel. The following figure illustrates the local effect of the MeanImageFilter. 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 otb::Image< InputPixelType,  2 >   InputImageType;
  typedef otb::Image< OutputPixelType, 2 >   OutputImageType;
```

Using the image types it is now possible to instantiate the filter type and create the filter object.

Figure 7.13: Effect of the MeanImageFilter.

```
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 7.13 illustrates the effect of this filter 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.

7.5.2 Median Filter

The source code for this example can be found in the file
`Examples/Filtering/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. 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 otb::Image< InputPixelType,  2 >   InputImageType;
  typedef otb::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.

Figure 7.14: Effect of the MedianImageFilter.

```
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 7.14 illustrates the effect of the MedianImageFilter filter 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.

7.5.3 Mathematical Morphology

Mathematical morphology has proved to be a powerful resource for image processing and analysis [79]. ITK implements mathematical morphology filters using NeighborhoodIterators and `itk::NeighborhoodOperators`. The toolkit contains two types of image morphology algorithms, filters that operate on binary images and filters that operate on grayscale images.

Binary Filters

The source code for this example can be found in the file
`Examples/Filtering/MathematicalMorphologyBinaryFilters.cxx`.

The following section illustrates the use of filters that perform basic mathematical morphology operations on binary images. The `itk::BinaryErodeImageFilter` and `itk::BinaryDilateImageFilter` are described here. The filter names clearly specify the type of image on which they operate. The header files required to construct a simple example of the use of the mathematical morphology filters are included below.

```
#include "itkBinaryErodeImageFilter.h"
#include "itkBinaryDilateImageFilter.h"
#include "itkBinaryBallStructuringElement.h"
```

The following code defines the input and output pixel types and their associated image types.

```
  const unsigned int Dimension = 2;

  typedef unsigned char   InputPixelType;
  typedef unsigned char   OutputPixelType;

  typedef otb::Image< InputPixelType,  Dimension >   InputImageType;
  typedef otb::Image< OutputPixelType, Dimension >   OutputImageType;
```

Mathematical morphology operations are implemented by applying an operator over the neighborhood of each input pixel. The combination of the rule and the neighborhood is known as *structuring element*. Although some rules have become de facto standards for image processing, there is a good deal of freedom as to what kind of algorithmic rule should be applied to the neighborhood. The implementation in ITK follows the typical rule of minimum for erosion and maximum for dilation.

The structuring element is implemented as a NeighborhoodOperator. In particular, the default structuring element is the `itk::BinaryBallStructuringElement` class. This class is instantiated using the pixel type and dimension of the input image.

```
  typedef itk::BinaryBallStructuringElement<
                      InputPixelType,
                      Dimension  >             StructuringElementType;
```

The structuring element type is then used along with the input and output image types for instantiating the type of the filters.

```
  typedef itk::BinaryErodeImageFilter<
                            InputImageType,
```

```
                          OutputImageType,
                          StructuringElementType >  ErodeFilterType;

typedef itk::BinaryDilateImageFilter<
                          InputImageType,
                          OutputImageType,
                          StructuringElementType >  DilateFilterType;
```

The filters can now be created by invoking the `New()` method and assigning the result to `itk::SmartPointer`s.

```
ErodeFilterType::Pointer  binaryErode  = ErodeFilterType::New();
DilateFilterType::Pointer binaryDilate = DilateFilterType::New();
```

The structuring element is not a reference counted class. Thus it is created as a C++ stack object instead of using `New()` and SmartPointers. The radius of the neighborhood associated with the structuring element is defined with the `SetRadius()` method and the `CreateStructuringElement()` method is invoked in order to initialize the operator. The resulting structuring element is passed to the mathematical morphology filter through the `SetKernel()` method, as illustrated below.

```
StructuringElementType  structuringElement;

structuringElement.SetRadius( 1 );  // 3x3 structuring element

structuringElement.CreateStructuringElement();

binaryErode->SetKernel(  structuringElement );
binaryDilate->SetKernel( structuringElement );
```

A binary image is provided as input to the filters. This image might be, for example, the output of a binary threshold image filter.

```
thresholder->SetInput( reader->GetOutput() );

InputPixelType background =   0;
InputPixelType foreground = 255;

thresholder->SetOutsideValue( background );
thresholder->SetInsideValue(  foreground );

thresholder->SetLowerThreshold( lowerThreshold );
thresholder->SetUpperThreshold( upperThreshold );

binaryErode->SetInput( thresholder->GetOutput() );
binaryDilate->SetInput( thresholder->GetOutput() );
```

Figure 7.15: Effect of erosion and dilation in a binary image.

The values that correspond to “objects” in the binary image are specified with the methods SetErodeValue() and SetDilateValue(). The value passed to these methods will be considered the value over which the dilation and erosion rules will apply.

```
binaryErode->SetErodeValue( foreground );
binaryDilate->SetDilateValue( foreground );
```

The filter is executed by invoking its Update() method, or by updating any downstream filter, like, for example, an image writer.

```
writerDilation->SetInput( binaryDilate->GetOutput() );
writerDilation->Update();
```

Figure 7.15 illustrates the effect of the erosion and dilation filters. The figure shows how these operations can be used to remove spurious details from segmented images.

Grayscale Filters

The source code for this example can be found in the file
Examples/Filtering/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 otb::Image< InputPixelType,  Dimension >   InputImageType;
typedef otb::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

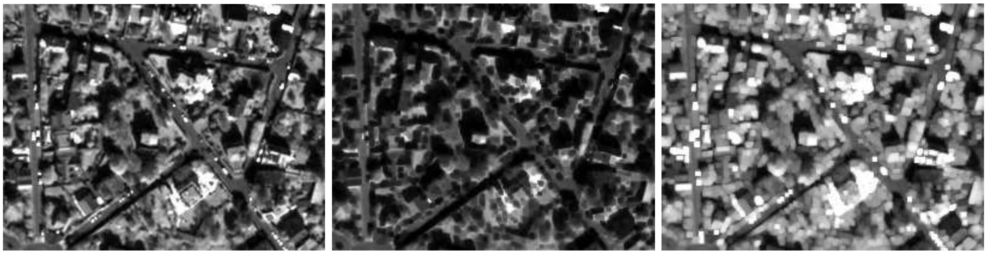


Figure 7.16: Effect of erosion and dilation in a grayscale image.

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, like, for example, an image writer.

```
writerDilation->SetInput( grayscaleDilate->GetOutput() );
writerDilation->Update();
```

Figure 7.16 illustrates the effect of the erosion and dilation filters. The figure shows how these operations can be used to remove spurious details from segmented images.

7.6 Smoothing Filters

Real image data has a level of uncertainty that is manifested in the variability of measures assigned to pixels. This uncertainty is usually interpreted as noise and considered an undesirable component of the image data. This section describes several methods that can be applied to reduce noise on images.

7.6.1 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 [22, 23].

Discrete Gaussian

The source code for this example can be found in the file `Examples/Filtering/DiscreteGaussianImageFilter.cxx`.

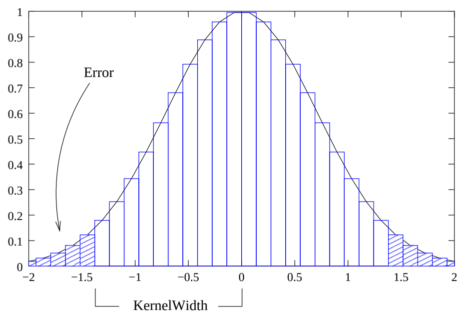


Figure 7.17: Discretized Gaussian.

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 [57] so that smoothing and derivative operations commute before and after discretization. In other words, finite difference derivatives on an image I that has been smoothed by convolution with the Gaussian are equivalent to finite differences computed on I by convolving with a derivative of the Gaussian.

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

```
#include "itkDiscreteGaussianImageFilter.h"
```

Types should be chosen for the pixels of the input and output images. Image types can be instantiated using the pixel type and dimension.

```
typedef    float    InputPixelType;
typedef    float    OutputPixelType;

typedef otb::Image< InputPixelType,  2 >   InputImageType;
typedef otb::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 would have triggered the execution of this one. For example, a writer could have been used after the filter.

```
rescaler->SetInput( filter->GetOutput() );
writer->SetInput( rescaler->GetOutput() );
writer->Update();
```

Figure 7.18 illustrates the effect of this filter.

Note that large Gaussian variances will produce large convolution kernels and correspondingly slower computation times. Unless a high degree of accuracy is required, it may be more desirable to use the approximating `itk::RecursiveGaussianImageFilter` with large variances.

Figure 7.18: Effect of the DiscreteGaussianImageFilter.

7.6.2 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 [70] introduced an alternative to linear-filtering that they called *anisotropic diffusion*. Anisotropic diffusion is closely related to the earlier work of Grossberg [37], who used similar nonlinear diffusion processes to model human vision. The motivation for anisotropic diffusion (also called *nonuniform* or *variable conductance* diffusion) is that a Gaussian smoothed image is a single time slice of the solution to the heat equation, that has the original image as its initial conditions. Thus, the solution to

$$\frac{\partial g(x,y,t)}{\partial t} = \nabla \cdot \nabla g(x,y,t), \tag{7.4}$$

where $g(x,y,0) = f(x,y)$ is the input image, is $g(x,y,t) = G(\sqrt{2t}) \otimes f(x,y)$, where $G(\sigma)$ is a Gaussian with standard deviation σ.

Anisotropic diffusion includes a variable conductance term that, in turn, depends on the differential structure of the image. Thus, the variable conductance can be formulated to limit the smoothing at "edges" in images, as measured by high gradient magnitude, for example.

$$g_t = \nabla \cdot c(|\nabla g|)\nabla g, \tag{7.5}$$

where, for notational convenience, we leave off the independent parameters of g and use the subscripts with respect to those parameters to indicate partial derivatives. The function $c(|\nabla g|)$ is a fuzzy cutoff that reduces the conductance at areas of large $|\nabla g|$, and can be any one of a number of functions. The literature has shown

$$c(|\nabla g|) = e^{-\frac{|\nabla g|^2}{2k^2}} \tag{7.6}$$

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 7.5 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 [34, 92] demonstrated the effectiveness of anisotropic diffusion on medical images. In a series of papers on the subject [97, 94, 96, 92, 93, 95], Whitaker described a detailed analytical and empirical analysis, introduced a smoothing term in the conductance that made the process more robust, invented a numerical scheme that virtually eliminated directional artifacts in the original algorithm, and generalized anisotropic diffusion to vector-valued images, an image processing technique that can be used on vector-valued medical data (such as the color cryosection data of the Visible Human Project).

For a vector-valued input $\vec{F} : U \mapsto \Re^m$ the process takes the form

$$\vec{F}_t = \nabla \cdot c(\mathcal{D}\vec{F})\vec{F}, \tag{7.7}$$

where $\mathcal{D}\vec{F}$ is a *dissimilarity* measure of $\vec{F}$, a generalization of the gradient magnitude to vector-valued images, that can incorporate linear and nonlinear coordinate transformations on the range of $\vec{F}$. In this way, the smoothing of the multiple images associated with vector-valued data is coupled through the conductance term, that fuses the information in the different images. Thus vector-valued, nonlinear diffusion can combine low-level image features (e.g. edges) across all "channels" of a vector-valued image in order to preserve or enhance those features in all of image "channels".

Vector-valued anisotropic diffusion is useful for denoising data from devices that produce multiple values such as MRI or color photography. When performing nonlinear diffusion on a color image, the color channels are diffused separately, but linked through the conductance term. Vector-valued diffusion it is also useful for processing registered data from different devices or for denoising higher-order geometric or statistical features from scalar-valued images [95, 102].

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 [93] 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 [84]. These include alternatives for constructing dissimilarity measures [78], directional (i.e., tensor-valued) conductance terms [90, 4] and level set interpretations [98].

Gradient Anisotropic Diffusion

The source code for this example can be found in the file
`Examples/Filtering/GradientAnisotropicDiffusionImageFilter.cxx`.

The `itk::GradientAnisotropicDiffusionImageFilter` implements an N-dimensional version of the classic Perona-Malik anisotropic diffusion equation for scalar-valued images [70].

The conductance term for this implementation is chosen as a function of the gradient magnitude of the image at each point, reducing the strength of diffusion at edge pixels.

$$C(\mathbf{x}) = e^{-\left(\frac{\|\nabla U(\mathbf{x})\|}{K}\right)^2} \tag{7.8}$$

The numerical implementation of this equation is similar to that described in the Perona-Malik paper [70], 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 otb::Image< InputPixelType,  2 >   InputImageType;
  typedef otb::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();
```

Figure 7.19: Effect of the GradientAnisotropicDiffusionImageFilter.

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();
```

A typical value for the time step is 0.125. The number of iterations is typically set to 5; more iterations result in further smoothing and will increase the computing time linearly.

Figure 7.19 illustrates the effect of this filter. In this example the filter was run with a time step of 0.125, 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`

7.6.3 Edge Preserving Speckle Reduction Filters

The source code for this example can be found in the file
`Examples/BasicFilters/LeeImageFilter.cxx`.

This example illustrates the use of the `otb::LeeImageFilter`. This filter belongs to the family of the edge-preserving smoothing filters which are usually used for speckle reduction in radar images. The Lee filter [56] aplies a linear regression which minimizes the mean-square error in the frame of a multiplicative speckle model.

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

```
#include "otbLeeImageFilter.h"
```

Then we must decide what pixel type to use for the image.

```
  typedef  unsigned char  PixelType;
```

The images are defined using the pixel type and the dimension.

```
  typedef otb::Image< PixelType,  2 >   InputImageType;
  typedef otb::Image< PixelType,  2 >   OutputImageType;
```

The filter can be instantiated using the image types defined above.

```
  typedef otb::LeeImageFilter< InputImageType, OutputImageType >  FilterType;
```

An `otb::ImageFileReader` class is also instantiated in order to read image data from a file.

```
  typedef otb::ImageFileReader< InputImageType >  ReaderType;
```

An `otb::ImageFileWriter` is instantiated in order to write the output image to a file.

```
  typedef otb::ImageFileWriter< OutputImageType >  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 `otb::LeeImageFilter`.

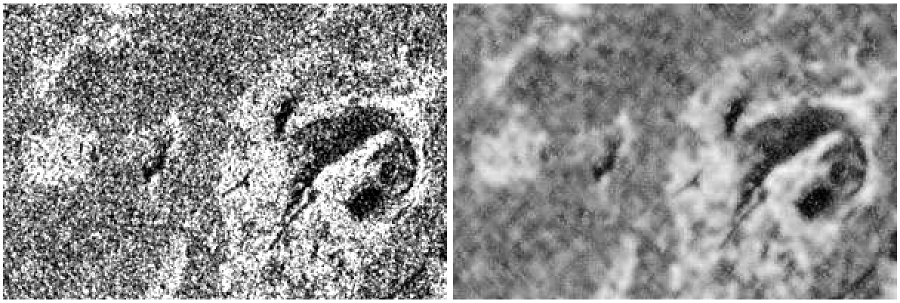


Figure 7.20: Result of applying the `otb::LeeImageFilter` to a SAR image.

```
filter->SetInput( reader->GetOutput() );
```

The method `SetRadius()` defines the size of the window to be used for the computation of the local statistics. The method `SetNbLooks()` sets the number of looks of the input image.

```
FilterType::SizeType Radius;
Radius[0]= atoi(argv[3]);
Radius[1]= atoi(argv[3]);

filter->SetRadius( Radius );
filter->SetNbLooks( atoi(argv[4]) );
```

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();
```

Figure 7.20 shows the result of applying the Lee filter to a SAR image.

The following classes provide similar functionality:

- `otb::FrostImageFilter`

7.6.4 Edge preserving Markov Random Field

The Markov Random Field framework for OTB is more detailled in 17.1.5 (p. 419).

The source code for this example can be found in the file
`Examples/Markov/MarkovRestaurationExample.cxx`.

The Markov Random Field framework can be used to apply an edge preserving filtering, thus playing a role of restauration.

This example applies the `otb::MarkovRandomFieldFilter` for image restauration. The structure of the example is similar to the other MRF example. The original image is assumed to be coded in one byte, thus 256 states are possible for each pixel. The only other modifications reside in the energy function chosen for the fidelity and for the regularization.

For the regularization energy function, we choose an edge preserving function:

$$\Phi(u) = \frac{u^2}{1+u^2} \tag{7.9}$$

and for the fidelity function, we choose a gaussian model.

The starting state of the Markov Random Field is given by the image itself as the final state should not be too far from it.

The first step toward the use of this filter is the inclusion of the proper header files:

```
#include "otbMRFEnergyEdgeFidelity.h"
#include "otbMRFEnergyGaussian.h"
#include "otbMRFOptimizerMetropolis.h"
#include "otbMRFSamplerRandom.h"
```

We declare the usual types:

```
  const unsigned int Dimension = 2;

  typedef double InternalPixelType;
  typedef unsigned char LabelledPixelType;
  typedef otb::Image<InternalPixelType, Dimension>  InputImageType;
  typedef otb::Image<LabelledPixelType, Dimension>    LabelledImageType;
```

We need to declare an additional reader for the initial state of the MRF. This reader has to be instantiated on the LabelledImageType.

```
  typedef otb::ImageFileReader< InputImageType >  ReaderType;
  typedef otb::ImageFileReader< LabelledImageType >  ReaderLabelledType;
  typedef otb::ImageFileWriter< LabelledImageType >  WriterType;

  ReaderType::Pointer reader = ReaderType::New();
  ReaderLabelledType::Pointer reader2 = ReaderLabelledType::New();
  WriterType::Pointer writer = WriterType::New();

  const char * inputFilename  = argv[1];
  const char * labelledFilename  = argv[2];
```

```
const char * outputFilename = argv[3];

reader->SetFileName( inputFilename );
reader2->SetFileName( labelledFilename );
writer->SetFileName( outputFilename );
```

We declare all the necessary types for the MRF:

```
typedef otb::MarkovRandomFieldFilter
<InputImageType,LabelledImageType> MarkovRandomFieldFilterType;

typedef otb::MRFSamplerRandom< InputImageType, LabelledImageType> SamplerType;

typedef otb::MRFOptimizerMetropolis OptimizerType;
```

The regularization and the fidelity energy are declared and instanciated:

```
typedef otb::MRFEnergyEdgeFidelity
<LabelledImageType, LabelledImageType>  EnergyRegularizationType;
typedef otb::MRFEnergyGaussian
<InputImageType, LabelledImageType>  EnergyFidelityType;


MarkovRandomFieldFilterType::Pointer markovFilter = MarkovRandomFieldFilterType::New();

EnergyRegularizationType::Pointer energyRegularization = EnergyRegularizationType::New();
EnergyFidelityType::Pointer energyFidelity = EnergyFidelityType::New();

OptimizerType::Pointer optimizer = OptimizerType::New();
SamplerType::Pointer sampler = SamplerType::New();
```

The number of possible states for each pixel is 256 as the image is assumed to be coded on one byte and we pass the parameters to the markovFilter.

```
unsigned int nClass = 256;


optimizer->SetSingleParameter(atof(argv[6]));
markovFilter->SetNumberOfClasses(nClass);
markovFilter->SetMaximumNumberOfIterations(atoi(argv[5]));
markovFilter->SetErrorTolerance(0.0);
markovFilter->SetLambda(atof(argv[4]));
markovFilter->SetNeighborhoodRadius(1);
```

Figure 7.21: Result of applying the `otb::MarkovRandomFieldFilter` to an extract from a PAN Quickbird image for restauration. From left to right : original image, restaured image with edge preservation.

```
markovFilter->SetEnergyRegularization(energyRegularization);
markovFilter->SetEnergyFidelity(energyFidelity);
markovFilter->SetOptimizer(optimizer);
markovFilter->SetSampler(sampler);
```

The original state of the MRF filter is passed through the `SetTrainingInput()` method:

```
markovFilter->SetTrainingInput(reader2->GetOutput());
```

And we plug the pipeline:

```
markovFilter->SetInput(reader->GetOutput());

typedef itk::RescaleIntensityImageFilter
    < LabelledImageType, LabelledImageType > RescaleType;
RescaleType::Pointer rescaleFilter = RescaleType::New();
rescaleFilter->SetOutputMinimum(0);
rescaleFilter->SetOutputMaximum(255);

rescaleFilter->SetInput( markovFilter->GetOutput() );

writer->SetInput( rescaleFilter->GetOutput() );

writer->Update();
```

Figure 7.21 shows the output of the Markov Random Field restauration.

7.7 Distance Map

The source code for this example can be found in the file
`Examples/Filtering/DanielssonDistanceMapImageFilter.cxx`.

This example illustrates the use of the `itk::DanielssonDistanceMapImageFilter`. This filter generates a distance map from the input image using the algorithm developed by Danielsson [19]. As secondary outputs, a Voronoi partition of the input elements is produced, as well as a vector image with the components of the distance vector to the closest point. The input to the map is assumed to be a set of points on the input image. Each point/pixel is considered to be a separate entity even if they share the same gray level value.

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

```
#include "itkDanielssonDistanceMapImageFilter.h"
```

Then we must decide what pixel types to use for the input and output images. Since the output will contain distances measured in pixels, the pixel type should be able to represent at least the width of the image, or said in $N-D$ terms, the maximum extension along all the dimensions. The input and output image types are now defined using their respective pixel type and dimension.

```
  typedef  unsigned char   InputPixelType;
  typedef  unsigned short  OutputPixelType;
  typedef otb::Image< InputPixelType,  2 >   InputImageType;
  typedef otb::Image< OutputPixelType, 2 >   OutputImageType;
```

The filter type can be instantiated using the input and output image types defined above. A filter object is created with the `New()` method.

```
  typedef itk::DanielssonDistanceMapImageFilter<
               InputImageType, OutputImageType >  FilterType;
  FilterType::Pointer filter = FilterType::New();
```

The input to the filter is taken from a reader and its output is passed to a `itk::RescaleIntensityImageFilter` and then to a writer.

```
  filter->SetInput( reader->GetOutput() );
  scaler->SetInput( filter->GetOutput() );
  writer->SetInput( scaler->GetOutput() );
```

The type of input image has to be specified. In this case, a binary image is selected.

```
  filter->InputIsBinaryOn();
```

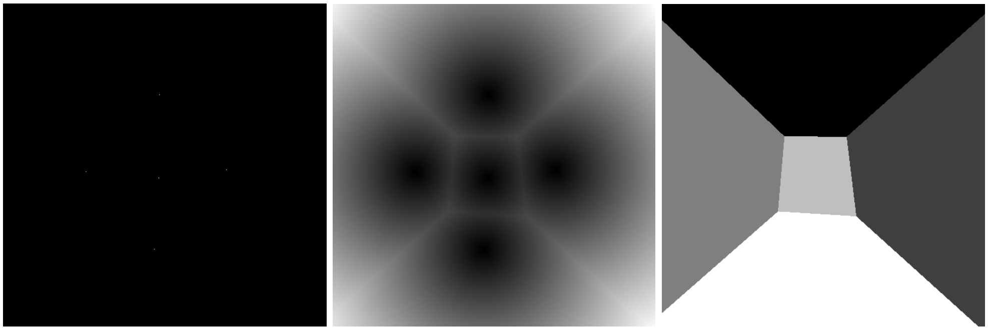


Figure 7.22: DanielssonDistanceMapImageFilter output. Set of pixels, distance map and Voronoi partition.

Figure 7.22 illustrates the effect of this filter on a binary image with a set of points. The input image is shown at left, the distance map at the center and the Voronoi partition at right. This filter computes distance maps in N-dimensions and is therefore capable of producing $N-D$ Voronoi partitions.

The Voronoi map is obtained with the `GetVoronoiMap()` method. In the lines below we connect this output to the intensity rescaler and save the result in a file.

```
scaler->SetInput( filter->GetVoronoiMap() );
writer->SetFileName( voronoiMapFileName );
writer->Update();
```

The distance filter also produces an image of `itk::Offset` pixels representing the vectorial distance to the closest object in the scene. The type of this output image is defined by the VectorImageType trait of the filter type.

```
typedef FilterType::VectorImageType   OffsetImageType;
```

We can use this type for instantiating an `otb::ImageFileWriter` type and creating an object of this class in the following lines.

```
typedef otb::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`.

CHAPTER

EIGHT

Image Registration

This chapter introduces OTB's (actually mainly ITK's) capabilities for performing image registration. Please note that the disparity map estimation approach presented in chapter 9 are very closely related to image registration. Image registration is the process of determining the spatial transform that maps points from one image to homologous points on a object in the second image. This concept is schematically represented in Figure 8.1. In OTB, registration is performed within a framework of pluggable components that can easily be interchanged. This flexibility means that a combinatorial variety of registration methods can be created, allowing users to pick and choose the right tools for their specific application.

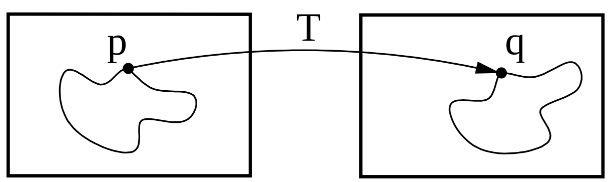


Figure 8.1: Image registration is the task of finding a spatial transform mapping on image into another.

8.1 Registration Framework

The components of the registration framework and their interconnections are shown in Figure 8.2. The basic input data to the registration process are two images: one is defined as the *fixed* image $f(\mathbf{X})$ and the other as the *moving* image $m(\mathbf{X})$. Where $\mathbf{X}$ represents a position in N-dimensional space. Registration is treated as an optimization problem with the goal of finding the spatial mapping that will bring the moving image into alignment with the fixed image.

The *transform* component $T(\mathbf{X})$ represents the spatial mapping of points from the fixed image space to points in the moving image space. The *interpolator* is used to evaluate moving image intensities at non-grid positions. The *metric* component $S(f, m \circ T)$ provides a measure of how well the fixed image is matched by the transformed moving image. This measure forms the quantitative criterion to be optimized by the *optimizer* over the search space defined by the

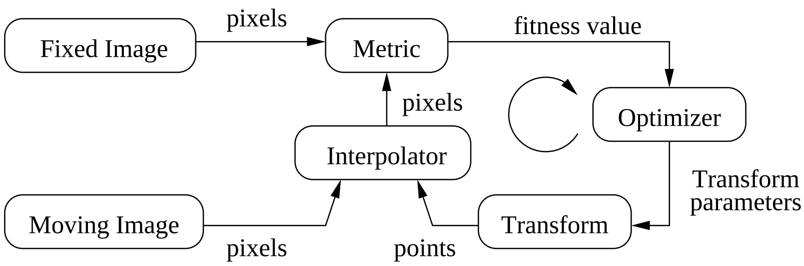


Figure 8.2: The basic components of the registration framework are two input images, a transform, a metric, an interpolator and an optimizer.

parameters of the *transform*.

These various OTB/ITK registration components will be described in later sections. First, we begin with some simple registration examples.

8.2 "Hello World" Registration

The source code for this example can be found in the file
`Examples/Registration/ImageRegistration1.cxx`.

This example illustrates the use of the image registration framework in ITK/OTB. It should be read as a "Hello World" for registration. Which means that for now, you don't ask "why?". Instead, use the example as an introduction to the elements that are typically involved in solving an image registration problem.

A registration method requires the following set of components: two input images, a transform, a metric, an interpolator and an optimizer. Some of these components are parameterized by the image type for which the registration is intended. The following header files provide declarations of common types used for these components.

```
#include "itkImageRegistrationMethod.h"
#include "itkTranslationTransform.h"
#include "itkMeanSquaresImageToImageMetric.h"
#include "itkLinearInterpolateImageFunction.h"
#include "itkRegularStepGradientDescentOptimizer.h"
#include "otbImage.h"
```

The types of each one of the components in the registration methods should be instantiated first. With that purpose, we start by selecting the image dimension and the type used for representing image pixels.

```
  const    unsigned int    Dimension = 2;
```

```
typedef  float           PixelType;
```

The types of the input images are instantiated by the following lines.

```
typedef otb::Image< PixelType, Dimension >  FixedImageType;
typedef otb::Image< PixelType, Dimension >  MovingImageType;
```

The transform that will map the fixed image space into the moving image space is defined below.

```
typedef itk::TranslationTransform< double, Dimension > TransformType;
```

An optimizer is required to explore the parameter space of the transform in search of optimal values of the metric.

```
typedef itk::RegularStepGradientDescentOptimizer       OptimizerType;
```

The metric will compare how well the two images match each other. Metric types are usually parameterized by the image types as it can be seen in the following type declaration.

```
typedef itk::MeanSquaresImageToImageMetric<
                                  FixedImageType,
                                  MovingImageType >    MetricType;
```

Finally, the type of the interpolator is declared. The interpolator will evaluate the intensities of the moving image at non-grid positions.

```
typedef itk:: LinearInterpolateImageFunction<
                                  MovingImageType,
                                  double          >    InterpolatorType;
```

The registration method type is instantiated using the types of the fixed and moving images. This class is responsible for interconnecting all the components that we have described so far.

```
typedef itk::ImageRegistrationMethod<
                                  FixedImageType,
                                  MovingImageType >    RegistrationType;
```

Each one of the registration components is created using its `New()` method and is assigned to its respective `itk::SmartPointer`.

```
MetricType::Pointer         metric        = MetricType::New();
TransformType::Pointer      transform     = TransformType::New();
OptimizerType::Pointer      optimizer     = OptimizerType::New();
InterpolatorType::Pointer   interpolator  = InterpolatorType::New();
RegistrationType::Pointer   registration  = RegistrationType::New();
```

Each component is now connected to the instance of the registration method.

```
registration->SetMetric(        metric        );
registration->SetOptimizer(     optimizer     );
registration->SetTransform(     transform     );
registration->SetInterpolator(  interpolator  );
```

Since we are working with high resolution images and expected shifts are larger than the resolution, we will need to smooth the images in order to avoid the optimizer to get stucked on local minima. In order to do this, we will use a simple mean filter.

```
typedef itk::MeanImageFilter<
             FixedImageType, FixedImageType >  FixedFilterType;

typedef itk::MeanImageFilter<
             MovingImageType, MovingImageType >  MovingFilterType;
```

```
FixedFilterType::Pointer  fixedFilter  = FixedFilterType::New();
MovingFilterType::Pointer movingFilter = MovingFilterType::New();

FixedImageType::SizeType indexFRadius;

indexFRadius[0] = 4; // radius along x
indexFRadius[1] = 4; // radius along y

fixedFilter->SetRadius( indexFRadius );

MovingImageType::SizeType indexMRadius;

indexMRadius[0] = 4; // radius along x
indexMRadius[1] = 4; // radius along y

movingFilter->SetRadius( indexMRadius );

fixedFilter->SetInput( fixedImageReader->GetOutput() );
movingFilter->SetInput( movingImageReader->GetOutput() );
```

Now we can plug the output of the smoothing filters at the input of the registration method.

```
registration->SetFixedImage(    fixedFilter->GetOutput()    );
registration->SetMovingImage(   movingFilter->GetOutput()   );
```

The registration can be restricted to consider only a particular region of the fixed image as input to the metric computation. This region is defined with the `SetFixedImageRegion()` method. You could use this feature to reduce the computational time of the registration or to avoid unwanted objects present in the image from affecting the registration outcome. In this example we use the full available content of the image. This region is identified by the `BufferedRegion` of the fixed image. Note that for this region to be valid the reader must first invoke its `Update()` method.

```
fixedFilter->Update();
registration->SetFixedImageRegion(
                 fixedFilter->GetOutput()->GetBufferedRegion() );
```

The parameters of the transform are initialized by passing them in an array. This can be used to setup an initial known correction of the misalignment. In this particular case, a translation transform is being used for the registration. The array of parameters for this transform is simply composed of the translation values along each dimension. Setting the values of the parameters to zero initializes the transform to an *Identity* transform. Note that the array constructor requires the number of elements to be passed as an argument.

```
typedef RegistrationType::ParametersType ParametersType;
ParametersType initialParameters( transform->GetNumberOfParameters() );

initialParameters[0] = 0.0;  // Initial offset in mm along X
initialParameters[1] = 0.0;  // Initial offset in mm along Y

registration->SetInitialTransformParameters( initialParameters );
```

At this point the registration method is ready for execution. The optimizer is the component that drives the execution of the registration. However, the ImageRegistrationMethod class orchestrates the ensemble to make sure that everything is in place before control is passed to the optimizer.

It is usually desirable to fine tune the parameters of the optimizer. Each optimizer has particular parameters that must be interpreted in the context of the optimization strategy it implements. The optimizer used in this example is a variant of gradient descent that attempts to prevent it from taking steps that are too large. At each iteration, this optimizer will take a step along the direction of the `itk::ImageToImageMetric` derivative. The initial length of the step is defined by the user. Each time the direction of the derivative abruptly changes, the optimizer assumes that a local extrema has been passed and reacts by reducing the step length by a half. After several reductions of the step length, the optimizer may be moving in a very restricted area of the transform parameter space. The user can define how small the step length should be to consider convergence to have been reached. This is equivalent to defining the precision with which the final transform should be known.

The initial step length is defined with the method `SetMaximumStepLength()`, while the tolerance for convergence is defined with the method `SetMinimumStepLength()`.

```
optimizer->SetMaximumStepLength( 3 );
optimizer->SetMinimumStepLength( 0.01 );
```

In case the optimizer never succeeds reaching the desired precision tolerance, it is prudent to establish a limit on the number of iterations to be performed. This maximum number is defined with the method `SetNumberOfIterations()`.

```
optimizer->SetNumberOfIterations( 200 );
```

The registration process is triggered by an invocation to the `Update()` method. If something goes wrong during the initialization or execution of the registration an exception will be thrown. We should therefore place the `Update()` method inside a `try/catch` block as illustrated in the following lines.

```
try
  {
  registration->Update();
  }
catch( itk::ExceptionObject & err )
  {
  std::cerr << "ExceptionObject caught !" << std::endl;
  std::cerr << err << std::endl;
  return -1;
  }
```

In a real life application, you may attempt to recover from the error by taking more effective actions in the catch block. Here we are simply printing out a message and then terminating the execution of the program.

The result of the registration process is an array of parameters that defines the spatial transformation in an unique way. This final result is obtained using the `GetLastTransformParameters()` method.

```
ParametersType finalParameters = registration->GetLastTransformParameters();
```

In the case of the `itk::TranslationTransform`, there is a straightforward interpretation of the parameters. Each element of the array corresponds to a translation along one spatial dimension.

```
const double TranslationAlongX = finalParameters[0];
const double TranslationAlongY = finalParameters[1];
```

The optimizer can be queried for the actual number of iterations performed to reach convergence. The `GetCurrentIteration()` method returns this value. A large number of iterations may be an indication that the maximum step length has been set too small, which is undesirable since it results in long computational times.

Figure 8.3: Fixed and Moving image provided as input to the registration method.

```
const unsigned int numberOfIterations = optimizer->GetCurrentIteration();
```

The value of the image metric corresponding to the last set of parameters can be obtained with the `GetValue()` method of the optimizer.

```
const double bestValue = optimizer->GetValue();
```

Let's execute this example over two of the images provided in `Examples/Data`:

- `QB_Suburb.png`
- `QB_Suburb13x17y.png`

The second image is the result of intentionally translating the first image by $(13, 17)$ pixels. Both images have unit-spacing and are shown in Figure 8.3. The registration takes 18 iterations and the resulting transform parameters are:

```
Translation X = 12.0192
Translation Y = 16.0231
```

As expected, these values match quite well the misalignment that we intentionally introduced in the moving image.

It is common, as the last step of a registration task, to use the resulting transform to map the moving image into the fixed image space. This is easily done with the `itk::ResampleImageFilter`. First, a ResampleImageFilter type is instantiated using the image types. It is convenient to use the fixed image type as the output type since it is likely that the transformed moving image will be compared with the fixed image.

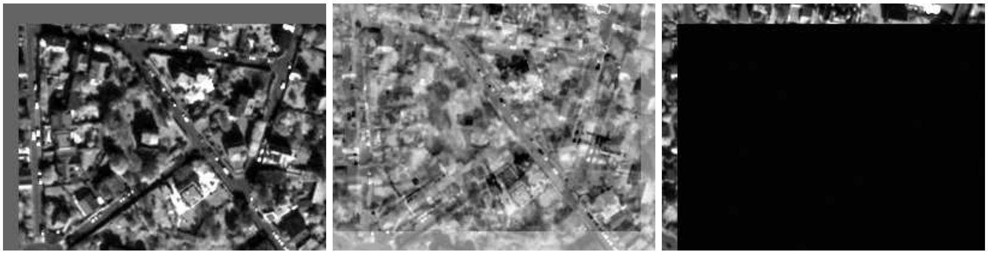


Figure 8.4: Mapped moving image and its difference with the fixed image before and after registration

```
typedef itk::ResampleImageFilter<
                            MovingImageType,
                            FixedImageType >    ResampleFilterType;
```

A resampling filter is created and the moving image is connected as its input.

```
ResampleFilterType::Pointer resampler = ResampleFilterType::New();
resampler->SetInput( movingImageReader->GetOutput() );
```

The Transform that is produced as output of the Registration method is also passed as input to the resampling filter. Note the use of the methods `GetOutput()` and `Get()`. This combination is needed here because the registration method acts as a filter whose output is a transform decorated in the form of a `itk::DataObject`. For details in this construction you may want to read the documentation of the `itk::DataObjectDecorator`.

```
resampler->SetTransform( registration->GetOutput()->Get() );
```

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->SetDefaultPixelValue( 100 );
```

The output of the filter is passed to a writer that will store the image in a file. An `itk::CastImageFilter` is used to convert the pixel type of the resampled image to the final type used by the writer. The cast and writer filters are instantiated below.

```
typedef unsigned char OutputPixelType;
```

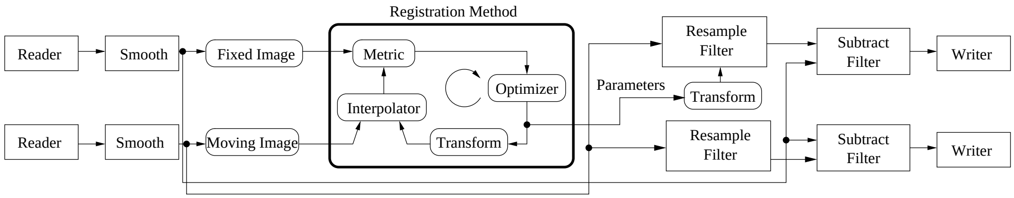


Figure 8.5: Pipeline structure of the registration example.

```
typedef otb::Image< OutputPixelType, Dimension > OutputImageType;
typedef itk::CastImageFilter<
                      FixedImageType,
                      OutputImageType > CastFilterType;
typedef otb::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 type.

Since the differences between the two images may correspond to very low values of intensity, we rescale those intensities with a itk::RescaleIntensityImageFilter in order to make them more visible. This rescaling will also make possible to visualize the negative values even if we save the difference image in a file format that only support unsigned pixel values[1]. We also reduce the DefaultPixelValue to "1" in order to prevent that value from absorbing the dynamic range of the differences between the two images.

```
typedef itk::RescaleIntensityImageFilter<
                                FixedImageType,
                                OutputImageType >   RescalerType;

RescalerType::Pointer intensityRescaler = RescalerType::New();

intensityRescaler->SetInput( difference->GetOutput() );
intensityRescaler->SetOutputMinimum(   0 );
intensityRescaler->SetOutputMaximum( 255 );

resampler->SetDefaultPixelValue( 1 );
```

Its output can be passed to another writer.

```
WriterType::Pointer writer2 = WriterType::New();
writer2->SetInput( intensityRescaler->GetOutput() );
```

For the purpose of comparison, the difference between the fixed image and the moving image before registration can also be computed by simply setting the transform to an identity transform. Note that the resampling is still necessary because the moving image does not necessarily have the same spacing, origin and number of pixels as the fixed image. Therefore a pixel-by-pixel operation cannot in general be performed. The resampling process with an identity transform will ensure that we have a representation of the moving image in the grid of the fixed image.

```
TransformType::Pointer identityTransform = TransformType::New();
identityTransform->SetIdentity();
resampler->SetTransform( identityTransform );
```

The complete pipeline structure of the current example is presented in Figure 8.5. The components of the registration method are depicted as well. Figure 8.4 (left) shows the result of resampling the moving image in order to map it onto the fixed image space. The top and right borders of the image appear in the gray level selected with the SetDefaultPixelValue() in the ResampleImageFilter. The center image shows the difference between the fixed image and the original moving image. That is, the difference before the registration is performed. The right image shows the difference between the fixed image and the transformed moving image.

[1]This is the case of PNG, BMP, JPEG and TIFF among other common file formats.

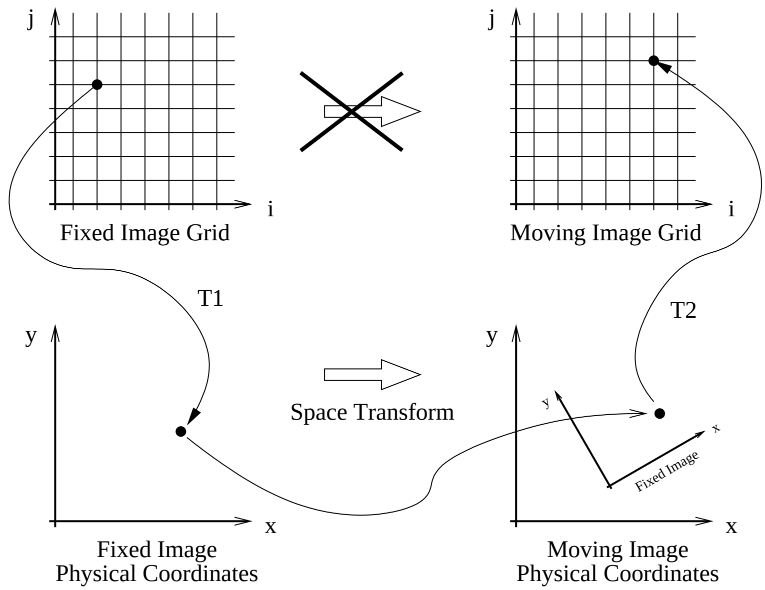


Figure 8.6: Different coordinate systems involved in the image registration process. Note that the transform being optimized is the one mapping from the physical space of the fixed image into the physical space of the moving image.

That is, after the registration has been performed. Both difference images have been rescaled in intensity in order to highlight those pixels where differences exist. Note that the final registration is still off by a fraction of a pixel, which results in bands around edges of anatomical structures to appear in the difference image. A perfect registration would have produced a null difference image.

8.3 Features of the Registration Framework

This section presents a discussion on the two most common difficulties that users encounter when they start using the ITK registration framework. They are, in order of difficulty

- The direction of the Transform mapping
- The fact that registration is done in physical coordinates

Probably the reason why these two topics tend to create confusion is that they are implemented in different ways in other systems and therefore users tend to have different expectations regarding how things should work in OTB. The situation is further complicated by the fact that most people describe image operations as if they were manually performed in a picture in paper.

8.3.1 Direction of the Transform Mapping

The Transform that is optimized in the ITK registration framework is the one that maps points from the physical space of the fixed image into the physical space of the moving image. This is illustrated in Figure 8.6. This implies that the Transform will accept as input points from the fixed image and it will compute the coordinates of the analogous points in the moving image. What tends to create confusion is the fact that when the Transform shifts a point on the **positive** X direction, the visual effect of this mapping, once the moving image is resampled, is equivalent to *manually shifting* the moving image along the **negative** X direction. In the same way, when the Transform applies a **clock-wise** rotation to the fixed image points, the visual effect of this mapping once the moving image has been resampled is equivalent to *manually rotating* the moving image **counter-clock-wise**.

The reason why this direction of mapping has been chosen for the ITK implementation of the registration framework is that this is the direction that better fits the fact that the moving image is expected to be resampled using the grid of the fixed image. The nature of the resampling process is such that an algorithm must go through every pixel of the *fixed* image and compute the intensity that should be assigned to this pixel from the mapping of the *moving* image. This computation involves taking the integral coordinates of the pixel in the image grid, usually called the "(i,j)" coordinates, mapping them into the physical space of the fixed image (transform **T1** in Figure 8.6), mapping those physical coordinates into the physical space of the moving image (Transform to be optimized), then mapping the physical coordinates of the moving image in to the integral coordinates of the discrete grid of the moving image (transform **T2** in the figure), where the value of the pixel intensity will be computed by interpolation.

If we have used the Transform that maps coordinates from the moving image physical space into the fixed image physical space, then the resampling process could not guarantee that every pixel in the grid of the fixed image was going to receive one and only one value. In other words, the resampling will have resulted in an image with holes and with redundant or overlapped pixel values.

As you have seen in the previous examples, and you will corroborate in the remaining examples in this chapter, the Transform computed by the registration framework is the Transform that can be used directly in the resampling filter in order to map the moving image into the discrete grid of the fixed image.

There are exceptional cases in which the transform that you want is actually the inverse transform of the one computed by the ITK registration framework. Only in those cases you may have to recur to invoking the `GetInverse()` method that most transforms offer. Make sure that before you consider following that dark path, you interact with the examples of resampling illustrated in section **??** in order to get familiar with the correct interpretation of the transforms.

8.3.2 Registration is done in physical space

The second common difficulty that users encounter with the ITK registration framework is related to the fact that ITK performs registration in the context of physical space and not in the discrete space of the image grid. Figure 8.6 show this concept by crossing the transform that goes between the two image grids. One important consequence of this fact is that having the correct image origin and image pixel size is fundamental for the success of the registration process in ITK. Users must make sure that they provide correct values for the origin and spacing of both the fixed and moving images.

A typical case that helps to understand this issue, is to consider the registration of two images where one has a pixel size different from the other. For example, a SPOt 5 image and a Quick-Bird image. Typically a Quickbird image will have a pixel size in the order of 0.6 m, while a SPOT 5 image will have a pixel size of 2.5 m.

A user performing registration between a SPOT 5 image and a Quickbird image may be naively expecting that because the SPOT 5 image has less pixels, a *scaling* factor is required in the Transform in order to map this image into the Quickbird image. At that point, this person is attempting to interpret the registration process directly between the two image grids, or in *pixel space*. What ITK will do in this case is to take into account the pixel size that the user has provided and it will use that pixel size in order to compute a scaling factor for Transforms *T1* and *T2* in Figure 8.6. Since these two transforms take care of the required scaling factor, the spatial Transform to be computed during the registration process does not need to be concerned about such scaling. The transform that ITK is computing is the one that will physically map the landscape the moving image into the landscape of the fixed image.

In order to better understand this concepts, it is very useful to draw sketches of the fixed and moving image *at scale* in the same physical coordinate system. That is the geometrical configuration that the ITK registration framework uses as context. Keeping this in mind helps a lot for interpreting correctly the results of a registration process performed with ITK.

8.4 Multi-Modality Registration

Some of the most challenging cases of image registration arise when images of different modalities are involved. In such cases, metrics based on direct comparison of gray levels are not applicable. It has been extensively shown that metrics based on the evaluation of mutual information are well suited for overcoming the difficulties of multi-modality registration.

The concept of Mutual Information is derived from Information Theory and its application to image registration has been proposed in different forms by different groups [17, 61, 89], a more detailed review can be found in [45]. The OTB, through ITK, currently provides five different implementations of Mutual Information metrics (see section 8.7 for details). The following example illustrates the practical use of some of these metrics.

8.4.1 Viola-Wells Mutual Information

The source code for this example can be found in the file
`Examples/Registration/ImageRegistration2.cxx`.

The following simple example illustrates how multiple imaging modalities can be registered using the ITK registration framework. The first difference between this and previous examples is the use of the `itk::MutualInformationImageToImageMetric` as the cost-function to be optimized. The second difference is the use of the `itk::GradientDescentOptimizer`. Due to the stochastic nature of the metric computation, the values are too noisy to work successfully with the `itk::RegularStepGradientDescentOptimizer`. Therefore, we will use the simpler GradientDescentOptimizer with a user defined learning rate. The following headers declare the basic components of this registration method.

```
#include "itkImageRegistrationMethod.h"
#include "itkTranslationTransform.h"
#include "itkMutualInformationImageToImageMetric.h"
#include "itkLinearInterpolateImageFunction.h"
#include "itkGradientDescentOptimizer.h"
#include "otbImage.h"
```

One way to simplify the computation of the mutual information is to normalize the statistical distribution of the two input images. The `itk::NormalizeImageFilter` is the perfect tool for this task. It rescales the intensities of the input images in order to produce an output image with zero mean and unit variance.

```
#include "itkNormalizeImageFilter.h"
```

Additionally, low-pass filtering of the images to be registered will also increase robustness against noise. In this example, we will use the `itk::DiscreteGaussianImageFilter` for that purpose. The characteristics of this filter have been discussed in Section 7.6.1.

```
#include "itkDiscreteGaussianImageFilter.h"
```

The moving and fixed images types should be instantiated first.

```
  const    unsigned int    Dimension = 2;
  typedef  unsigned short  PixelType;

  typedef otb::Image< PixelType, Dimension >  FixedImageType;
  typedef otb::Image< PixelType, Dimension >  MovingImageType;
```

It is convenient to work with an internal image type because mutual information will perform better on images with a normalized statistical distribution. The fixed and moving images will be normalized and converted to this internal type.

```
typedef   float     InternalPixelType;
typedef otb::Image< InternalPixelType, Dimension > InternalImageType;
```

The rest of the image registration components are instantiated as illustrated in Section 8.2 with the use of the `InternalImageType`.

```
typedef itk::TranslationTransform< double, Dimension > TransformType;
typedef itk::GradientDescentOptimizer                  OptimizerType;
typedef itk::LinearInterpolateImageFunction<
                                  InternalImageType,
                                  double             > InterpolatorType;
typedef itk::ImageRegistrationMethod<
                                  InternalImageType,
                                  InternalImageType >  RegistrationType;
```

The mutual information metric type is instantiated using the image types.

```
typedef itk::MutualInformationImageToImageMetric<
                                        InternalImageType,
                                        InternalImageType >    MetricType;
```

The metric is created using the `New()` method and then connected to the registration object.

```
MetricType::Pointer         metric        = MetricType::New();
registration->SetMetric( metric  );
```

The metric requires a number of parameters to be selected, including the standard deviation of the Gaussian kernel for the fixed image density estimate, the standard deviation of the kernel for the moving image density and the number of samples use to compute the densities and entropy values. Details on the concepts behind the computation of the metric can be found in Section 8.7.4. Experience has shown that a kernel standard deviation of 0.4 works well for images which have been normalized to a mean of zero and unit variance. We will follow this empirical rule in this example.

```
metric->SetFixedImageStandardDeviation(  0.4 );
metric->SetMovingImageStandardDeviation( 0.4 );
```

The normalization filters are instantiated using the fixed and moving image types as input and the internal image type as output.

```
typedef itk::NormalizeImageFilter<
                              FixedImageType,
                              InternalImageType
```

```
                                        > FixedNormalizeFilterType;

  typedef itk::NormalizeImageFilter<
                                MovingImageType,
                                InternalImageType
                                              > MovingNormalizeFilterType;

  FixedNormalizeFilterType::Pointer fixedNormalizer =
                                            FixedNormalizeFilterType::New();

  MovingNormalizeFilterType::Pointer movingNormalizer =
                                            MovingNormalizeFilterType::New();
```

The blurring filters are declared using the internal image type as both the input and output types. In this example, we will set the variance for both blurring filters to 2.0.

```
  typedef itk::DiscreteGaussianImageFilter<
                                      InternalImageType,
                                      InternalImageType
                                                    > GaussianFilterType;

  GaussianFilterType::Pointer fixedSmoother  = GaussianFilterType::New();
  GaussianFilterType::Pointer movingSmoother = GaussianFilterType::New();

  fixedSmoother->SetVariance( 4.0 );
  movingSmoother->SetVariance( 4.0 );
```

The output of the readers becomes the input to the normalization filters. The output of the normalization filters is connected as input to the blurring filters. The input to the registration method is taken from the blurring filters.

```
  fixedNormalizer->SetInput(  fixedImageReader->GetOutput() );
  movingNormalizer->SetInput( movingImageReader->GetOutput() );

  fixedSmoother->SetInput( fixedNormalizer->GetOutput() );
  movingSmoother->SetInput( movingNormalizer->GetOutput() );

  registration->SetFixedImage(    fixedSmoother->GetOutput()    );
  registration->SetMovingImage(   movingSmoother->GetOutput()   );
```

We should now define the number of spatial samples to be considered in the metric computation. Note that we were forced to postpone this setting until we had done the preprocessing of the images because the number of samples is usually defined as a fraction of the total number of pixels in the fixed image.

The number of spatial samples can usually be as low as 1% of the total number of pixels in the fixed image. Increasing the number of samples improves the smoothness of the metric from one iteration to another and therefore helps when this metric is used in conjunction with optimizers that rely of the continuity of the metric values. The trade-off, of course, is that a larger number of samples result in longer computation times per every evaluation of the metric.

It has been demonstrated empirically that the number of samples is not a critical parameter for the registration process. When you start fine tuning your own registration process, you should start using high values of number of samples, for example in the range of 20% to 50% of the number of pixels in the fixed image. Once you have succeeded to register your images you can then reduce the number of samples progressively until you find a good compromise on the time it takes to compute one evaluation of the Metric. Note that it is not useful to have very fast evaluations of the Metric if the noise in their values results in more iterations being required by the optimizer to converge.

```
const unsigned int numberOfPixels = fixedImageRegion.GetNumberOfPixels();

const unsigned int numberOfSamples =
                      static_cast< unsigned int >( numberOfPixels * 0.01 );

metric->SetNumberOfSpatialSamples( numberOfSamples );
```

Since larger values of mutual information indicate better matches than smaller values, we need to maximize the cost function in this example. By default the GradientDescentOptimizer class is set to minimize the value of the cost-function. It is therefore necessary to modify its default behavior by invoking the `MaximizeOn()` method. Additionally, we need to define the optimizer's step size using the `SetLearningRate()` method.

```
optimizer->SetLearningRate( 150.0 );
optimizer->SetNumberOfIterations( 300 );
optimizer->MaximizeOn();
```

Note that large values of the learning rate will make the optimizer unstable. Small values, on the other hand, may result in the optimizer needing too many iterations in order to walk to the extrema of the cost function. The easy way of fine tuning this parameter is to start with small values, probably in the range of $\{5.0, 10.0\}$. Once the other registration parameters have been tuned for producing convergence, you may want to revisit the learning rate and start increasing its value until you observe that the optimization becomes unstable. The ideal value for this parameter is the one that results in a minimum number of iterations while still keeping a stable path on the parametric space of the optimization. Keep in mind that this parameter is a multiplicative factor applied on the gradient of the Metric. Therefore, its effect on the optimizer step length is proportional to the Metric values themselves. Metrics with large values will require you to use smaller values for the learning rate in order to maintain a similar optimizer behavior.

Let's execute this example over two of the images provided in `Examples/Data`:

- `RamsesROISmall.png`

Figure 8.7: A SAR image (fixed image) and an aerial photograph (moving image) are provided as input to the registration method.

- `ADS40RoiSmall.png`

The moving image after resampling is presented on the left side of Figure 8.8. The center and right figures present a checkerboard composite of the fixed and moving images before and after registration. Since the real deformation between the 2 images is not simply a shift, some registration errors remain, but the left part of the images is correctly registered.

Figure 8.8: Mapped moving image (left) and composition of fixed and moving images before (center) and after (right) registration.

8.5 Centered Transforms

The OTB/ITK image coordinate origin is typically located in one of the image corners (see section 5.1.4 for details). This results in counter-intuitive transform behavior when rotations and scaling are involved. Users tend to assume that rotations and scaling are performed around a fixed point at the center of the image. In order to compensate for this difference in natural interpretation, the concept of *centered* transforms have been introduced into the toolkit. The following sections describe the main characteristics of such transforms.

8.5.1 Rigid Registration in 2D

The source code for this example can be found in the file
`Examples/Registration/ImageRegistration5.cxx`.

This example illustrates the use of the `itk::CenteredRigid2DTransform` for performing rigid registration in 2*D*. The example code is for the most part identical to that presented in Section 8.2. The main difference is the use of the CenteredRigid2DTransform here instead of the `itk::TranslationTransform`.

In addition to the headers included in previous examples, the following header must also be included.

```
#include "itkCenteredRigid2DTransform.h"
```

The transform type is instantiated using the code below. The only template parameter for this class is the representation type of the space coordinates.

```
  typedef itk::CenteredRigid2DTransform< double > TransformType;
```

The transform object is constructed below and passed to the registration method.

```
  TransformType::Pointer  transform = TransformType::New();
  registration->SetTransform( transform );
```

Since we are working with high resolution images and expected shifts are larger than the resolution, we will need to smooth the images in order to avoid the optimizer to get stucked on local minima. In order to do this, we will use a simple mean filter.

```
  typedef itk::MeanImageFilter<
              FixedImageType, FixedImageType >  FixedFilterType;

  typedef itk::MeanImageFilter<
              MovingImageType, MovingImageType >  MovingFilterType;
```

```
FixedFilterType::Pointer  fixedFilter  = FixedFilterType::New();
MovingFilterType::Pointer movingFilter = MovingFilterType::New();

FixedImageType::SizeType indexFRadius;

indexFRadius[0] = 4; // radius along x
indexFRadius[1] = 4; // radius along y

fixedFilter->SetRadius( indexFRadius );

MovingImageType::SizeType indexMRadius;

indexMRadius[0] = 4; // radius along x
indexMRadius[1] = 4; // radius along y

movingFilter->SetRadius( indexMRadius );

fixedFilter->SetInput( fixedImageReader->GetOutput() );
movingFilter->SetInput( movingImageReader->GetOutput() );
```

Now we can plug the output of the smoothing filters at the input of the registration method.

```
registration->SetFixedImage(    fixedFilter->GetOutput()    );
registration->SetMovingImage(   movingFilter->GetOutput()   );
```

In this example, the input images are taken from readers. The code below updates the readers in order to ensure that the image parameters (size, origin and spacing) are valid when used to initialize the transform. We intend to use the center of the fixed image as the rotation center and then use the vector between the fixed image center and the moving image center as the initial translation to be applied after the rotation.

```
fixedImageReader->Update();
movingImageReader->Update();
```

The center of rotation is computed using the origin, size and spacing of the fixed image.

```
FixedImageType::Pointer fixedImage = fixedImageReader->GetOutput();

const SpacingType fixedSpacing = fixedImage->GetSpacing();
const OriginType  fixedOrigin  = fixedImage->GetOrigin();
const RegionType  fixedRegion  = fixedImage->GetLargestPossibleRegion();
const SizeType    fixedSize    = fixedRegion.GetSize();
```

```
TransformType::InputPointType centerFixed;

centerFixed[0] = fixedOrigin[0] + fixedSpacing[0] * fixedSize[0] / 2.0;
centerFixed[1] = fixedOrigin[1] + fixedSpacing[1] * fixedSize[1] / 2.0;
```

The center of the moving image is computed in a similar way.

```
MovingImageType::Pointer movingImage = movingImageReader->GetOutput();

const SpacingType movingSpacing = movingImage->GetSpacing();
const OriginType  movingOrigin  = movingImage->GetOrigin();
const RegionType  movingRegion  = movingImage->GetLargestPossibleRegion();
const SizeType    movingSize    = movingRegion.GetSize();

TransformType::InputPointType centerMoving;

centerMoving[0] = movingOrigin[0] + movingSpacing[0] * movingSize[0] / 2.0;
centerMoving[1] = movingOrigin[1] + movingSpacing[1] * movingSize[1] / 2.0;
```

The most straightforward method of initializing the transform parameters is to configure the transform and then get its parameters with the method `GetParameters()`. Here we initialize the transform by passing the center of the fixed image as the rotation center with the `SetCenter()` method. Then the translation is set as the vector relating the center of the moving image to the center of the fixed image. This last vector is passed with the method `SetTranslation()`.

```
transform->SetCenter( centerFixed );
transform->SetTranslation( centerMoving - centerFixed );
```

Let's finally initialize the rotation with a zero angle.

```
transform->SetAngle( 0.0 );
```

Now we pass the current transform's parameters as the initial parameters to be used when the registration process starts.

```
registration->SetInitialTransformParameters( transform->GetParameters() );
```

Keeping in mind that the scale of units in rotation and translation is quite different, we take advantage of the scaling functionality provided by the optimizers. We know that the first element of the parameters array corresponds to the angle that is measured in radians, while the other parameters correspond to translations that are measured in the units of the spacin (pixels in our case). For this reason we use small factors in the scales associated with translations and the coordinates of the rotation center .

```
typedef OptimizerType::ScalesType       OptimizerScalesType;
OptimizerScalesType optimizerScales( transform->GetNumberOfParameters() );
const double translationScale = 1.0 / 1000.0;

optimizerScales[0] = 1.0;
optimizerScales[1] = translationScale;
optimizerScales[2] = translationScale;
optimizerScales[3] = translationScale;
optimizerScales[4] = translationScale;

optimizer->SetScales( optimizerScales );
```

Next we set the normal parameters of the optimization method. In this case we are using an `itk::RegularStepGradientDescentOptimizer`. Below, we define the optimization parameters like the relaxation factor, initial step length, minimal step length and number of iterations. These last two act as stopping criteria for the optimization.

```
double initialStepLength = 0.1;


optimizer->SetRelaxationFactor( 0.6 );
optimizer->SetMaximumStepLength( initialStepLength );
optimizer->SetMinimumStepLength( 0.001 );
optimizer->SetNumberOfIterations( 200 );
```

Let's execute this example over two of the images provided in `Examples/Data`:

- `QB_Suburb.png`
- `QB_SuburbRotated10.png`

The second image is the result of intentionally rotating the first image by 10 degrees around the geometrical center of the image. Both images have unit-spacing and are shown in Figure 8.9. The registration takes 21 iterations and produces the results:

```
[0.176168, 134.515, 103.011, -0.00182313, 0.0717891]
```

These results are interpreted as

- Angle = 0.176168 radians
- Center = $(134.515, 103.011)$ pixels
- Translation = $(-0.00182313, 0.0717891)$ pixels

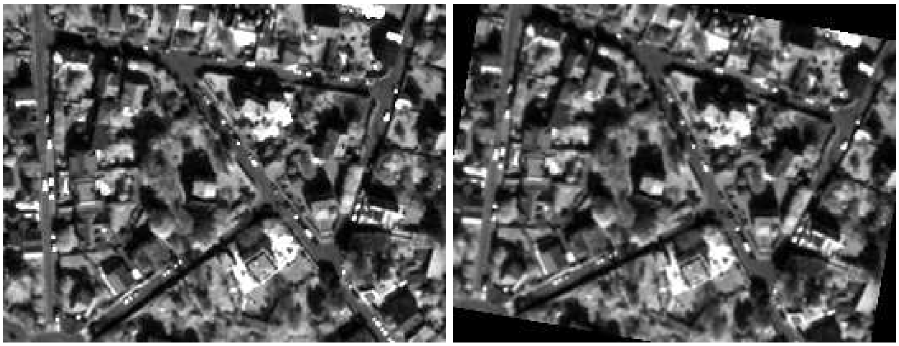


Figure 8.9: Fixed and moving images are provided as input to the registration method using the CenteredRigid2D transform.

Figure 8.10: Resampled moving image (left). Differences between the fixed and moving images, before (center) and after (right) registration using the CenteredRigid2D transform.

As expected, these values match the misalignment intentionally introduced into the moving image quite well, since 10 degrees is about 0.174532 radians.

Figure 8.10 shows from left to right the resampled moving image after registration, the difference between fixed and moving images before registration, and the difference between fixed and resampled moving image after registration. It can be seen from the last difference image that the rotational component has been solved but that a small centering misalignment persists.

Let's now consider the case in which rotations and translations are present in the initial registration, as in the following pair of images:

- `QB_Suburb.png`
- `QB_SuburbR10X13Y17.png`

The second image is the result of intentionally rotating the first image by 10 degrees and then translating it 13 pixels in X and 17 pixels in Y. Both images have unit-spacing and are shown in Figure 8.11. In order to accelerate convergence it is convenient to use a larger step length as shown here.

```
optimizer->SetMaximumStepLength( 1.0 );
```

The registration now takes 34 iterations and produces the following results:

```
[0.176125, 135.553, 102.159, -11.9102, -15.8045]
```

These parameters are interpreted as

- Angle = 0.176125 radians
- Center = $(135.553, 102.159)$ millimeters
- Translation = $(-11.9102, -15.8045)$ millimeters

These values approximately match the initial misalignment intentionally introduced into the moving image, since 10 degrees is about 0.174532 radians. The horizontal translation is well resolved while the vertical translation ends up being off by about one millimeter.

Figure 8.12 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.

8.5.2 Centered Affine Transform

The source code for this example can be found in the file `Examples/Registration/ImageRegistration9.cxx`.

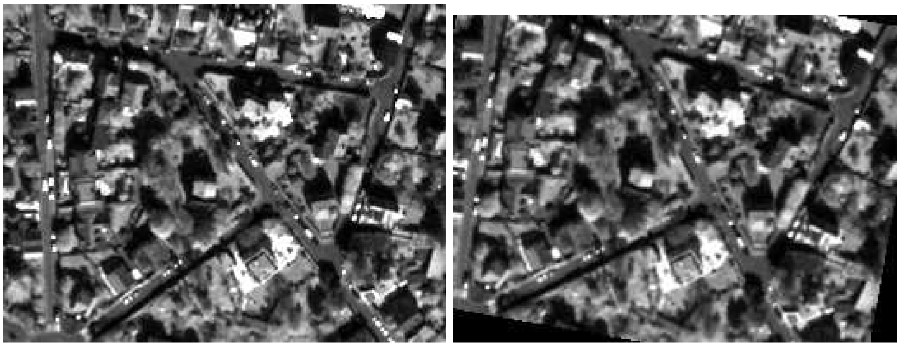


Figure 8.11: Fixed and moving images provided as input to the registration method using the CenteredRigid2D transform.

Figure 8.12: Resampled moving image (left). Differences between the fixed and moving images, before (center) and after (right) registration with the CenteredRigid2D transform.

This example illustrates the use of the `itk::AffineTransform` for performing registration. The example code is, for the most part, identical to previous ones. The main difference is the use of the AffineTransform here instead of the `itk::CenteredRigid2DTransform`. We will focus on the most relevant changes in the current code and skip the basic elements already explained in previous examples.

Let's start by including the header file of the AffineTransform.

```
#include "itkAffineTransform.h"
```

We define then the types of the images to be registered.

```
  const    unsigned int    Dimension = 2;
  typedef  float           PixelType;

  typedef otb::Image< PixelType, Dimension >  FixedImageType;
  typedef otb::Image< PixelType, Dimension >  MovingImageType;
```

The transform type is instantiated using the code below. The template parameters of this class are the representation type of the space coordinates and the space dimension.

```
  typedef itk::AffineTransform<
                                double,
                                Dimension  >     TransformType;
```

The transform object is constructed below and passed to the registration method.

```
  TransformType::Pointer  transform = TransformType::New();
  registration->SetTransform( transform );
```

Since we are working with high resolution images and expected shifts are larger than the resolution, we will need to smooth the images in order to avoid the optimizer to get stucked on local minima. In order to do this, we will use a simple mean filter.

```
  typedef itk::MeanImageFilter<
             FixedImageType, FixedImageType >  FixedFilterType;

  typedef itk::MeanImageFilter<
             MovingImageType, MovingImageType >  MovingFilterType;



  FixedFilterType::Pointer  fixedFilter  = FixedFilterType::New();
  MovingFilterType::Pointer movingFilter = MovingFilterType::New();
```

```
FixedImageType::SizeType indexFRadius;

indexFRadius[0] = 4; // radius along x
indexFRadius[1] = 4; // radius along y

fixedFilter->SetRadius( indexFRadius );

MovingImageType::SizeType indexMRadius;

indexMRadius[0] = 4; // radius along x
indexMRadius[1] = 4; // radius along y

movingFilter->SetRadius( indexMRadius );

fixedFilter->SetInput( fixedImageReader->GetOutput() );
movingFilter->SetInput( movingImageReader->GetOutput() );
```

Now we can plug the output of the smoothing filters at the input of the registration method.

```
registration->SetFixedImage(    fixedFilter->GetOutput()    );
registration->SetMovingImage(   movingFilter->GetOutput()   );
```

In this example, we use the `itk::CenteredTransformInitializer` helper class in order to compute a reasonable value for the initial center of rotation and the translation. The initializer is set to use the center of mass of each image as the initial correspondence correction.

```
typedef itk::CenteredTransformInitializer<
                                  TransformType,
                                  FixedImageType,
                                  MovingImageType >  TransformInitializerType;
TransformInitializerType::Pointer initializer = TransformInitializerType::New();
initializer->SetTransform(   transform );
initializer->SetFixedImage(  fixedImageReader->GetOutput() );
initializer->SetMovingImage( movingImageReader->GetOutput() );
initializer->MomentsOn();
initializer->InitializeTransform();
```

Now we pass the parameters of the current transform as the initial parameters to be used when the registration process starts.

```
registration->SetInitialTransformParameters(
                               transform->GetParameters() );
```

Keeping in mind that the scale of units in scaling, rotation and translation are quite different, we take advantage of the scaling functionality provided by the optimizers. We know that the first $N \times N$ elements of the parameters array correspond to the rotation matrix factor, the next N correspond to the rotation center, and the last N are the components of the translation to be applied after multiplication with the matrix is performed.

```
typedef OptimizerType::ScalesType       OptimizerScalesType;
OptimizerScalesType optimizerScales( transform->GetNumberOfParameters() );

optimizerScales[0] =  1.0;
optimizerScales[1] =  1.0;
optimizerScales[2] =  1.0;
optimizerScales[3] =  1.0;
optimizerScales[4] =  translationScale;
optimizerScales[5] =  translationScale;

optimizer->SetScales( optimizerScales );
```

We also set the usual parameters of the optimization method. In this case we are using an `itk::RegularStepGradientDescentOptimizer`. Below, we define the optimization parameters like initial step length, minimal step length and number of iterations. These last two act as stopping criteria for the optimization.

```
optimizer->SetMaximumStepLength( steplength );
optimizer->SetMinimumStepLength( 0.0001 );
optimizer->SetNumberOfIterations( maxNumberOfIterations );
```

We also set the optimizer to do minimization by calling the `MinimizeOn()` method.

```
optimizer->MinimizeOn();
```

Finally we trigger the execution of the registration method by calling the `Update()` method. The call is placed in a `try/catch` block in case any exceptions are thrown.

```
try
  {
  registration->StartRegistration();
  }
catch( itk::ExceptionObject & err )
  {
  std::cerr << "ExceptionObject caught !" << std::endl;
  std::cerr << err << std::endl;
  return -1;
  }
```

Once the optimization converges, we recover the parameters from the registration method. This is done with the `GetLastTransformParameters()` method. We can also recover the final value of the metric with the `GetValue()` method and the final number of iterations with the `GetCurrentIteration()` method.

```
OptimizerType::ParametersType finalParameters =
                  registration->GetLastTransformParameters();

const double finalRotationCenterX = transform->GetCenter()[0];
const double finalRotationCenterY = transform->GetCenter()[1];
const double finalTranslationX    = finalParameters[4];
const double finalTranslationY    = finalParameters[5];

const unsigned int numberOfIterations = optimizer->GetCurrentIteration();
const double bestValue = optimizer->GetValue();
```

Let's execute this example over two of the images provided in `Examples/Data`:

- `QB_Suburb.png`
- `QB_SuburbR10X13Y17.png`

The second image is the result of intentionally rotating the first image by 10 degrees and then translating by $(13,17)$. Both images have unit-spacing and are shown in Figure 8.13. 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 83 iterations and produces

```
20.2134   [0.983291, -0.173507, 0.174626, 0.983028, -12.1899, -16.0882]
```

These results are interpreted as

- Iterations = 83
- Final Metric = 20.2134
- Center = $(134.152, 104.067)$ pixels
- Translation = $(-12.1899, -16.0882)$ pixels
- Affine scales = $(0.999024, 0.997875)$

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 10.0401 degrees, which is approximately the intentional misalignment of 10.0 degrees.

Figure 8.14 shows the output of the registration. The right most image of this figure shows the squared magnitude difference between the fixed image and the resampled moving image.

Figure 8.13: Fixed and moving images provided as input to the registration method using the AffineTransform.

Figure 8.14: The resampled moving image (left), and the difference between the fixed and moving images before (center) and after (right) registration with the AffineTransform transform.

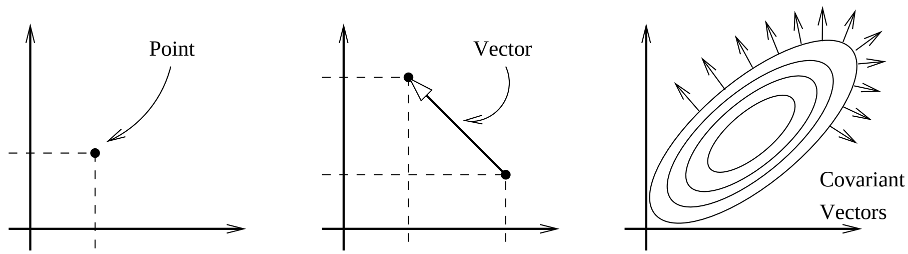


Figure 8.15: Geometric representation objects in ITK.

8.6 Transforms

In OTB, we use the Insight Toolkit `itk::Transform` objects encapsulate the mapping of points and vectors from an input space to an output space. If a transform is invertible, back transform methods are also provided. Currently, ITK provides a variety of transforms from simple translation, rotation and scaling to general affine and kernel transforms. Note that, while in this section we discuss transforms in the context of registration, transforms are general and can be used for other applications. Some of the most commonly used transforms will be discussed in detail later. Let's begin by introducing the objects used in ITK for representing basic spatial concepts.

8.6.1 Geometrical Representation

ITK implements a consistent geometric representation of the space. The characteristics of classes involved in this representation are summarized in Table 8.1. In this regard, ITK takes full advantage of the capabilities of Object Oriented programming and resists the temptation of using simple arrays of `float` or `double` in order to represent geometrical objects. The use of basic arrays would have blurred the important distinction between the different geometrical concepts and would have allowed for the innumerable conceptual and programming errors that result from using a vector where a point is needed or vice versa.

Additional uses of the `itk::Point`, `itk::Vector` and `itk::CovariantVector` classes have been discussed in Chapter 5. Each one of these classes behaves differently under spatial transformations. It is therefore quite important to keep their distinction clear. Figure 8.15 illustrates the differences between these concepts.

Transform classes provide different methods for mapping each one of the basic space-representation objects. Points, vectors and covariant vectors are transformed using the methods `TransformPoint()`, `TransformVector()` and `TransformCovariantVector()` respectively.

One of the classes that deserve further comments is the `itk::Vector`. This ITK class tend to be misinterpreted as a container of elements instead of a geometrical object. This is a common misconception originated by the fact that Computer Scientist and Software Engineers misuse the term "Vector". The actual word "Vector" is relatively young. It was coined by William

Class	**Geometrical concept**
`itk::Point`	Position in space. In N-dimensional space it is represented by an array of N numbers associated with space coordinates.
`itk::Vector`	Relative position between two points. In N-dimensional space it is represented by an array of N numbers, each one associated with the distance along a coordinate axis. Vectors do not have a position in space. A vector is defined as the subtraction of two points.
`itk::CovariantVector`	Orthogonal direction to a $(N-1)$-dimensional manifold in space. For example, in $3D$ it corresponds to the vector orthogonal to a surface. This is the appropriate class for representing Gradients of functions. Covariant vectors do not have a position in space. Covariant vector should not be added to Points, nor to Vectors.

Table 8.1: Summary of objects representing geometrical concepts in ITK.

Hamilton in his book "*Elements of Quaternions*" published in 1886 (post-mortem)[39]. 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 [26].

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 [58].

Textbooks in programming do not help to clarify those concepts and loosely use the term "*Vector*" for the purpose of representing an "enumerated set of common elements". STL follows this trend and continue using the word "*Vector*" for what it was not supposed to be used [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 that require to insert and delete elements, and that may have large numbers of elements.

The Insight Toolkit deals with real objects that inhabit the physical space. This is particularly true in the context of the image registration framework. We chose to give the appropriate name to the mathematical objects that describe geometrical relationships in N-Dimensional space. It is for this reason that we explicitly make clear the distinction between Point, Vector and CovariantVector, despite the fact that most people would be happy with a simple use of `double[3]` for the three concepts and then will proceed to perform all sort of conceptually flawed operations such as

- Adding two Points
- Dividing a Point by a Scalar
- Adding a Covariant Vector to a Point
- Adding a Covariant Vector to a Vector

In order to enforce the correct use of the Geometrical concepts in ITK we organized these classes in a hierarchy that supports reuse of code and yet compartmentalize the behavior of the individual classes. The use of the `itk::FixedArray` as base class of the `itk::Point`, the `itk::Vector` and the `itk::CovariantVector` was a design decision based on calling things by their correct name.

An `itk::FixedArray` is an enumerated collection with a fixed number of elements. You can instantiate a fixed array of letters, or a fixed array of images, or a fixed array of transforms, or a fixed array of geometrical shapes. Therefore, the FixedArray only implements the functionality that is necessary to access those enumerated elements. No assumptions can be made at this point on any other operations required by the elements of the FixedArray, except the fact of having a default constructor.

The `itk::Point` is a type that represents the spatial coordinates of a spatial location. Based on geometrical concepts we defined the valid operations of the Point class. In particular we made sure that no `operator+()` was defined between Points, and that no `operator*( scalar )` nor `operator/( scalar )` were defined for Points.

In other words, you could do in ITK operations such as:

- Vector = Point - Point
- Point += Vector
- Point -= Vector
- Point = BarycentricCombination(Point, Point)

and you cannot (because you **should not**) do operation such as

- Point = Point * Scalar

- Point = Point + Point
- Point = Point / Scalar

The `itk::Vector` is, by Hamilton's definition, the subtraction between two points. Therefore a Vector must satisfy the following basic operations:

- Vector = Point - Point
- Point = Point + Vector
- Point = Point - Vector
- Vector = Vector + Vector
- Vector = Vector - Vector

An `itk::Vector` object is intended to be instantiated over elements that support mathematical operation such as addition, subtraction and multiplication by scalars.

8.6.2 Transform General Properties

Each transform class typically has several methods for setting its parameters. For example, `itk::Euler2DTransform` provides methods for specifying the offset, angle, and the entire rotation matrix. However, for use in the registration framework, the parameters are represented by a flat Array of doubles to facilitate communication with generic optimizers. In the case of the Euler2DTransform, the transform is also defined by three doubles: the first representing the angle, and the last two the offset. The flat array of parameters is defined using `SetParameters()`. A description of the parameters and their ordering is documented in the sections that follow.

In the context of registration, the transform parameters define the search space for optimizers. That is, the goal of the optimization is to find the set of parameters defining a transform that results in the best possible value of an image metric. The more parameters a transform has, the longer its computational time will be when used in a registration method since the dimension of the search space will be equal to the number of transform parameters.

Another requirement that the registration framework imposes on the transform classes is the computation of their Jacobians. In general, metrics require the knowledge of the Jacobian in order to compute Metric derivatives. The Jacobian is a matrix whose element are the partial derivatives of the output point with respect to the array of parameters that defines the transform:[2]

[2]Note that the term *Jacobian* is also commonly used for the matrix representing the derivatives of output point coordinates with respect to input point coordinates. Sometimes the term is loosely used to refer to the determinant of such a matrix. [26]

Behavior	Number of Parameters	Parameter Ordering	Restrictions
Maps every point to itself, every vector to itself and every co-variant vector to itself.	0	NA	Only defined when the input and output space has the same number of dimensions.

Table 8.2: Characteristics of the identity transform.

$$J = \begin{bmatrix} \frac{\partial x_1}{\partial p_1} & \frac{\partial x_1}{\partial p_2} & \cdots & \frac{\partial x_1}{\partial p_m} \\ \frac{\partial x_2}{\partial p_1} & \frac{\partial x_2}{\partial p_2} & \cdots & \frac{\partial x_2}{\partial p_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial x_n}{\partial p_1} & \frac{\partial x_n}{\partial p_2} & \cdots & \frac{\partial x_n}{\partial p_m} \end{bmatrix} \tag{8.1}$$

where $\{p_i\}$ are the transform parameters and $\{x_i\}$ are the coordinates of the output point. Within this framework, the Jacobian is represented by an `itk::Array2D` of doubles and is obtained from the transform by method `GetJacobian()`. The Jacobian can be interpreted as a matrix that indicates for a point in the input space how much its mapping on the output space will change as a response to a small variation in one of the transform parameters. Note that the values of the Jacobian matrix depend on the point in the input space. So actually the Jacobian can be noted as $J(\mathbf{X})$, where $\mathbf{X} = \{x_i\}$. The use of transform Jacobians enables the efficient computation of metric derivatives. When Jacobians are not available, metrics derivatives have to be computed using finite difference at a price of $2M$ evaluations of the metric value, where M is the number of transform parameters.

The following sections describe the main characteristics of the transform classes available in ITK.

8.6.3 Identity Transform

The identity transform `itk::IdentityTransform` is mainly used for debugging purposes. It is provided to methods that require a transform and in cases where we want to have the certainty that the transform will have no effect whatsoever in the outcome of the process. It is just a `NULL` operation. The main characteristics of the identity transform are summarized in Table 8.2

8.6.4 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

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Represents a simple translation of points in the input space and has no effect on vectors or covariant vectors.	Same as the input space dimension.	The *i*-th parameter represents the translation in the *i*-th dimension.	Only defined when the input and output space has the same number of dimensions.

Table 8.3: Characteristics of the TranslationTransform class.

unchanged under this transformation since they are not associated with a particular position in space. Translation is the best transform to use when starting a registration method. Before attempting to solve for rotations or scaling it is important to overlap the anatomical objects in both images as much as possible. This is done by resolving the translational misalignment between the images. Translations also have the advantage of being fast to compute and having parameters that are easy to interpret. The main characteristics of the translation transform are presented in Table 8.3.

8.6.5 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.

$$\begin{array}{lccccccc} \text{Point} & \mathbf{P}' & = & T(\mathbf{P}) & : & \mathbf{P}'_\mathbf{i} & = & \mathbf{P_i} \cdot \mathbf{S_i} \\ \text{Vector} & \mathbf{V}' & = & T(\mathbf{V}) & : & \mathbf{V}'_\mathbf{i} & = & \mathbf{V_i} \cdot \mathbf{S_i} \\ \text{CovariantVector} & \mathbf{C}' & = & T(\mathbf{C}) & : & \mathbf{C}'_\mathbf{i} & = & \mathbf{C_i} / \mathbf{S_i} \end{array} \tag{8.2}$$

where $\mathbf{P_i}$, $\mathbf{V_i}$ and $\mathbf{C_i}$ are the point, vector and covariant vector i-th components while $\mathbf{S_i}$ is the scaling factor along dimension $i-th$. The following equation illustrates the effect of the scaling transform on a $3D$ point.

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} S_1 & 0 & 0 \\ 0 & S_2 & 0 \\ 0 & 0 & S_3 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{8.3}$$

Behavior	Number of Parameters	Parameter Ordering	Restrictions
Points are transformed by multiplying each one of their coordinates by the corresponding scale factor for the dimension. Vectors are transformed as points. Covariant vectors are transformed by *dividing* their components by the scale factor in the corresponding dimension.	Same as the input space dimension.	The *i*-th parameter represents the scaling in the *i*-th dimension.	Only defined when the input and output space has the same number of dimensions.

Table 8.4: Characteristics of the ScaleTransform class.

Scaling appears to be a simple transformation but there are actually a number of issues to keep in mind when using different scale factors along every dimension. There are subtle effects—for example, when computing image derivatives. Since derivatives are represented by covariant vectors, their values are not intuitively modified by scaling transforms.

One of the difficulties with managing scaling transforms in a registration process is that typical optimizers manage the parameter space as a vector space where addition is the basic operation. Scaling is better treated in the frame of a logarithmic space where additions result in regular multiplicative increments of the scale. Gradient descent optimizers have trouble updating step length, since the effect of an additive increment on a scale factor diminishes as the factor grows. In other words, a scale factor variation of $(1.0+\varepsilon)$ is quite different from a scale variation of $(5.0+\varepsilon)$.

Registrations involving scale transforms require careful monitoring of the optimizer parameters in order to keep it progressing at a stable pace. Note that some of the transforms discussed in following sections, for example, the AffineTransform, have hidden scaling parameters and are therefore subject to the same vulnerabilities of the ScaleTransform.

In cases involving misalignments with simultaneous translation, rotation and scaling components it may be desirable to solve for these components independently. The main characteristics of the scale transform are presented in Table 8.4.

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Points are transformed by multiplying each one of their coordinates by the corresponding scale factor for the dimension. Vectors are transformed as points. Covariant vectors are transformed by *dividing* their components by the scale factor in the corresponding dimension.	Same as the input space dimension.	The *i*-th parameter represents the scaling in the *i*-th dimension.	Only defined when the input and output space has the same number of dimensions. The difference between this transform and the ScaleTransform is that here the scaling factors are passed as logarithms, in this way their behavior is closer to the one of a Vector space.

Table 8.5: Characteristics of the ScaleLogarithmicTransform class.

8.6.6 Scale Logarithmic Transform

The `itk::ScaleLogarithmicTransform` is a simple variation of the `itk::ScaleTransform`. It is intended to improve the behavior of the scaling parameters when they are modified by optimizers. The difference between this transform and the ScaleTransform is that the parameter factors are passed here as logarithms. In this way, multiplicative variations in the scale become additive variations in the logarithm of the scaling factors.

8.6.7 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,

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \end{bmatrix} \tag{8.4}$$

where θ is the rotation angle and (T_x, T_y) are the components of the translation.

A challenging aspect of this transformation is the fact that translations and rotations do not form a vector space and cannot be managed as linear independent parameters. Typical optimizers

Behavior	Number of Parameters	Parameter Ordering	Restrictions
Represents a $2D$ rotation and a $2D$ translation. Note that the translation component has no effect on the transformation of vectors and covariant vectors.	3	The first parameter is the angle in radians and the last two parameters are the translation in each dimension.	Only defined for two-dimensional input and output spaces.

Table 8.6: Characteristics of the Euler2DTransform class.

make the loose assumption that parameters exist in a vector space and rely on the step length to be small enough for this assumption to hold approximately.

In addition to the non-linearity of the parameter space, the most common difficulty found when using this transform is the difference in units used for rotations and translations. Rotations are measured in radians; hence, their values are in the range $[-\pi,\pi]$. Translations are measured in millimeters and their actual values vary depending on the image modality being considered. In practice, translations have values on the order of 10 to 100. This scale difference between the rotation and translation parameters is undesirable for gradient descent optimizers because they deviate from the trajectories of descent and make optimization slower and more unstable. In order to compensate for these differences, ITK optimizers accept an array of scale values that are used to normalize the parameter space.

Registrations involving angles and translations should take advantage of the scale normalization functionality in order to obtain the best performance out of the optimizers. The main characteristics of the Euler2DTransform class are presented in Table 8.6.

8.6.8 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.

In practice, the user must supply the center of rotation in the input space, the angle of rotation

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Represents a $2D$ rotation around a user-provided center followed by a $2D$ translation.	5	The first parameter is the angle in radians. Second and third are the center of rotation coordinates and the last two parameters are the translation in each dimension.	Only defined for two-dimensional input and output spaces.

Table 8.7: Characteristics of the CenteredRigid2DTransform class.

and a translation to be applied after the rotation. With these parameters, the transform initializes a rotation matrix and a translation vector that together perform the equivalent of translating the center of rotation to the origin of coordinates, rotating by the specified angle, translating back to the center of rotation and finally translating by the user-specified vector.

As with the Euler2DTransform, this transform suffers from the difference in units used for rotations and translations. Rotations are measured in radians; hence, their values are in the range $[-\pi, \pi]$. The center of rotation and the translations are measured in millimeters, and their actual values vary depending on the image modality being considered. Registrations involving angles and translations should take advantage of the scale normalization functionality of the optimizers in order to get the best performance out of them.

The following equation illustrates the effect of the transform on an input point (x, y) that maps to the output point (x', y'),

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} x - C_x \\ y - C_y \end{bmatrix} + \begin{bmatrix} T_x + C_x \\ T_y + C_y \end{bmatrix} \tag{8.5}$$

where θ is the rotation angle, (C_x, C_y) are the coordinates of the rotation center and (T_x, T_y) are the components of the translation. Note that the center coordinates are subtracted before the rotation and added back after the rotation. The main features of the CenteredRigid2DTransform are presented in Table 8.7.

8.6.9 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

Behavior	Number of Parameters	Parameter Ordering	Restrictions
Represents a 2*D* rotation, homogeneous scaling and a 2*D* translation. Note that the translation component has no effect on the transformation of vectors and covariant vectors.	4	The first parameter is the scaling factor for all dimensions, the second is the angle in radians, and the last two parameters are the translations in (x,y) respectively.	Only defined for two-dimensional input and output spaces.

Table 8.8: Characteristics of the Similarity2DTransform class.

`itk::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'),

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \lambda & 0 \\ 0 & \lambda \end{bmatrix} \cdot \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} x - C_x \\ y - C_y \end{bmatrix} + \begin{bmatrix} T_x + C_x \\ T_y + C_y \end{bmatrix} \tag{8.6}$$

where λ is the scale factor, θ is the rotation angle, (C_x, C_y) are the coordinates of the rotation center and (T_x, T_y) are the components of the translation. Note that the center coordinates are subtracted before the rotation and scaling, and they are added back afterwards. The main features of the Similarity2DTransform are presented in Table 8.8.

A possible approach for controlling optimization in the parameter space of this transform is to dynamically modify the array of scales passed to the optimizer. The effect produced by the parameter scaling can be used to steer the walk in the parameter space (by giving preference to some of the parameters over others). For example, perform some iterations updating only the rotation angle, then balance the array of scale factors in the optimizer and perform another set of iterations updating only the translations.

8.6.10 QuaternionRigidTransform

The `itk::QuaternionRigidTransform` class implements a rigid transformation in 3*D* space. The rotational part of the transform is represented using a quaternion while the translation is

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Represents a $3D$ rotation and a $3D$ translation. The rotation is specified as a quaternion, defined by a set of four numbers $\mathbf{q}$. The relationship between quaternion and rotation about vector $\mathbf{n}$ by angle θ is as follows: $$\mathbf{q} = (\mathbf{n}\sin(\theta/\mathbf{2}), \cos(\theta/\mathbf{2}))$$ Note that if the quaternion is not of unit length, scaling will also result.	7	The first four parameters defines the quaternion and the last three parameters the translation in each dimension.	Only defined for three-dimensional input and output spaces.

Table 8.9: Characteristics of the QuaternionRigidTransform class.

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 insures that after advancing in any direction on the parameter space, the resulting set of transform parameters is mapped back into the permissible set of parameters. In practice, this comes down to normalizing the newly-computed quaternion to make sure that the transformation remains rigid and no scaling is applied. The main characteristics of the QuaternionRigidTransform are presented in Table 8.9.

The Quaternion rigid transform also accepts a user-defined center of rotation. In this way, the transform can easily be used for registering images where the rotation is mostly relative to the center of the image instead one of the corners. The coordinates of this rotation center are not subject to optimization. They only participate in the computation of the mappings for Points and in the computation of the Jacobian. The transformations for Vectors and CovariantVector are not affected by the selection of the rotation center.

8.6.11 VersorTransform

By definition, a *Versor* is the rotational part of a Quaternion. It can also be defined as a *unit-quaternion* [39, 49]. Versors only have three independent components, since they are restricted

Behavior	Number of Parameters	Parameter Ordering	Restrictions
Represents a 3*D* rotation. The rotation is specified by a versor or unit quaternion. The rotation is performed around a user-specified center of rotation.	3	The three parameters define the versor.	Only defined for three-dimensional input and output spaces.

Table 8.10: Characteristics of the Versor Transform

to reside in the space of unit-quaternions. The implementation of versors in the toolkit uses a set of three numbers. These three numbers correspond to the first three components of a quaternion. The fourth component of the quaternion is computed internally such that the quaternion is of unit length. The main characteristics of the `itk::VersorTransform` are presented in Table 8.10.

This transform exclusively represents rotations in 3*D*. 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 3*D* 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 [39].

The center of rotation can be specified by the user with the `SetCenter()` method. The center is not part of the parameters to be optimized, therefore it remains the same during an optimization process. Its value is used during the computations for transforming Points and when computing the Jacobian.

8.6.12 VersorRigid3DTransform

The `itk::VersorRigid3DTransform` implements a rigid transformation in 3*D* space. It is a variant of the `itk::QuaternionRigidTransform` and the `itk::VersorTransform`. It can be seen as a `itk::VersorTransform` plus a translation defined by a vector. The advantage of this class with respect 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

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Represents a 3*D* rotation and a 3*D* translation. The rotation is specified by a versor or unit quaternion, while the translation is represented by a vector. Users can specify the coordinates of the center of rotation.	6	The first three parameters define the versor and the last three parameters the translation in each dimension.	Only defined for three-dimensional input and output spaces.

Table 8.11: Characteristics of the VersorRigid3DTransform class.

of rotation. The center coordinates are not modified during the optimization performed in a registration process. The main features of this transform are summarized in Table 8.11. This transform is probably the best option to use when dealing with rigid transformations in 3*D*.

Given that the space of Versors is not a Vector space, typical gradient descent optimizers are not well suited for exploring the parametric space of this transform. The `itk::VersorRigid3DTranformOptimizer` has been introduced in the ITK toolkit with the purpose of providing an optimizer that is aware of the Versor space properties on the rotational part of this transform, as well as the Vector space properties on the translational part of the transform.

8.6.13 Euler3DTransform

The `itk::Euler3DTransform` implements a rigid transformation in 3*D* space. It can be seen as a rotation followed by a translation. This class exposes six parameters, three for the Euler angles that represent the rotation and three for the translational components. This transform also allows the users to set a specific center of rotation. The center coordinates are not modified during the optimization performed in a registration process. The main features of this transform are summarized in Table 8.12.

The fact that the three rotational parameters are non-linear and do not behave like Vector spaces must be taken into account when selecting an optimizer to work with this transform and when fine tuning the parameters of such optimizer. It is strongly recommended to use this transform by introducing very small variations on the rotational components. A small rotation will be in the range of 1 degree, which in radians is approximately 0.0.1745.

You should not expect this transform to be able to compensate for large rotations just by be-

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Represents a rigid rotation in 3*D* space. That is, a rotation followed by a 3*D* translation. The rotation is specified by three angles representing rotations to be applied around the X, Y and Z axis one after another. The translation part is represented by a Vector. Users can also specify the coordinates of the center of rotation.	6	The first three parameters are the rotation angles around X, Y and Z axis, and the last three parameters are the translations along each dimension.	Only defined for three-dimensional input and output spaces.

Table 8.12: Characteristics of the Euler3DTransform class.

ing driven with the optimizer. In practice you must provide a reasonable initialization of the transform angles and only need to correct for residual rotations in the order of 10 or 20 degrees.

8.6.14 Similarity3DTransform

The `itk::Similarity3DTransform` implements a similarity transformation in 3*D* space. It can be seen as an homogeneous scaling followed by a `itk::VersorRigid3DTransform`. This class exposes seven parameters, one for the scaling factor, three for the versor components and three for the translational components. This transform also allows the users to set a specific center of rotation. The center coordinates are not modified during the optimization performed in a registration process. Both the rotation and scaling operations are performed with respect to the center of rotation. The main features of this transform are summarized in Table 8.13.

The fact that the scaling and rotational spaces are non-linear and do not behave like Vector spaces must be taken into account when selecting an optimizer to work with this transform and when fine tuning the parameters of such optimizer.

8.6.15 Rigid3DPerspectiveTransform

The `itk::Rigid3DPerspectiveTransform` implements a rigid transformation in 3*D* space followed by a perspective projection. This transform is intended to be used in $3D/2D$ registra-

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, that is the same center used for scaling.	7	The first parameter is the scaling factor, the next three parameters define the versor and the last three parameters the translation in each dimension.	Only defined for three-dimensional input and output spaces.

Table 8.13: Characteristics of the Similarity3DTransform class.

tion problems where a 3D object is projected onto a 2D plane. This is the case of Fluoroscopic images used for image guided intervention, and it is also the case for classical radiography. Users must provide a value for the focal distance to be used during the computation of the perspective transform. This transform also allows users to set a specific center of rotation. The center coordinates are not modified during the optimization performed in a registration process. The main features of this transform are summarized in Table 8.14. This transform is also used when creating Digitally Reconstructed Radiographs (DRRs).

The strategies for optimizing the parameters of this transform are the same ones used for optimizing the VersorRigid3DTransform. In particular, you can use the same VersorRigid3DTranformOptimizer in order to optimize the parameters of this class.

8.6.16 AffineTransform

The `itk::AffineTransform` is one of the most popular transformations used for image registration. Its main advantage comes from the fact that it is represented as a linear transformation. The main features of this transform are presented in Table 8.15.

The set of AffineTransform coefficients can actually be represented in a vector space of dimension $(N+1)\times N$. This makes it possible for optimizers to be used appropriately on this search space. However, the high dimensionality of the search space also implies a high computational complexity of cost-function derivatives. The best compromise in the reduction of this computational time is to use the transform's Jacobian in combination with the image gradient for

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Represents a rigid $3D$ transformation followed by a perspective projection. The rotation is specified by a Versor, while the translation is represented by a Vector. Users can specify the coordinates of the center of rotation. They must specifically a focal distance to be used for the perspective projection. The rotation center and the focal distance parameters are not modified during the optimization process.	6	The first three parameters define the Versor and the last three parameters the Translation in each dimension.	Only defined for three-dimensional input and two-dimensional output spaces. This is one of the few transforms where the input space has a different dimension from the output space.

Table 8.14: Characteristics of the Rigid3DPerspectiveTransform class.

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Represents an affine transform composed of rotation, scaling, shearing and translation. The transform is specified by a $N \times N$ matrix and a $N \times 1$ vector where N is the space dimension.	$(N+1) \times N$	The first $N \times N$ parameters define the matrix in column-major order (where the column index varies the fastest). The last N parameters define the translations for each dimension.	Only defined when the input and output space have the same dimension.

Table 8.15: Characteristics of the AffineTransform class.

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 8.7 illustrates the effect of applying the AffineTransform in a point in 3D space.

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} M_{00} & M_{01} & M_{02} \\ M_{10} & M_{11} & M_{12} \\ M_{20} & M_{21} & M_{22} \end{bmatrix} \cdot \begin{bmatrix} x - C_x \\ y - C_y \\ z - C_z \end{bmatrix} + \begin{bmatrix} T_x + C_x \\ T_y + C_y \\ T_z + C_z \end{bmatrix} \tag{8.7}$$

A registration based on the affine transform may be more effective when applied after simpler transformations have been used to remove the major components of misalignment. Otherwise it will incur an overwhelming computational cost. For example, using an affine transform, the first set of optimization iterations would typically focus on removing large translations. This task could instead be accomplished by a translation transform in a parameter space of size N instead of the $(N+1) \times N$ associated with the affine transform.

Tracking the evolution of a registration process that uses AffineTransforms can be challenging, since it is difficult to represent the coefficients in a meaningful way. A simple printout of the transform coefficients generally does not offer a clear picture of the current behavior and trend of the optimization. A better implementation uses the affine transform to deform wire-frame cube which is shown in a 3D visualization display.

8.6.17 BSplineDeformableTransform

The `itk::BSplineDeformableTransform` is designed to be used for solving deformable registration problems. This transform is equivalent to generation a deformation field where a deformation vector is assigned to every point in space. The deformation vectors are computed using BSpline interpolation from the deformation values of points located in a coarse grid, that is usually referred to as the BSpline grid.

The BSplineDeformableTransform is not flexible enough for accounting for large rotations or shearing, or scaling differences. In order to compensate for this limitation, it provides the functionality of being composed with an arbitrary transform. This transform is known as the *Bulk* transform and it is applied to points before they are mapped with the displacement field.

This transform do not provide functionalities for mapping Vectors nor CovariantVectors, only

Behavior	Number of Parameters	Parameter Ordering	Restrictions
Represents a free from deformation by providing a deformation field from the interpolation of deformations in a coarse grid.	$M \times N$	Where M is the number of nodes in the BSpline grid and N is the dimension of the space.	Only defined when the input and output space have the same dimension. This transform has the advantage of allowing to compute deformable registration. It also has the disadvantage of having a very high dimensional parametric space, and therefore requiring long computation times.

Table 8.16: Characteristics of the BSplineDeformableTransform class.

Points can be mapped. The reason is that the variations of a vector under a deformable transform actually depend on the location of the vector in space. In other words, Vector only make sense as the relative position between two points.

The BSplineDeformableTransform has a very large number of parameters and therefore is well suited for the `itk::LBFGSOptimizer` and `itk::LBFGSBOptimizer`. The use of this transform for was proposed in the following papers [77, 63, 64].

8.6.18 KernelTransforms

Kernel Transforms are a set of Transforms that are also suitable for performing deformable registration. These transforms compute on the fly the displacements corresponding to a deformation field. The displacement values corresponding to every point in space are computed by interpolation from the vectors defined by a set of *Source Landmarks* and a set of *Target Landmarks*.

Several variations of these transforms are available in the toolkit. They differ on the type of interpolation kernel that is used when computing the deformation in a particular point of space. Note that these transforms are computationally expensive and that their numerical complexity is proportional to the number of landmarks and the space dimension.

The following is the list of Transforms based on the KernelTransform.

- `itk::ElasticBodySplineKernelTransform`
- `itk::ElasticBodyReciprocalSplineKernelTransform`
- `itk::ThinPlateSplineKernelTransform`

- `itk::ThinPlateR2LogRSplineKernelTransform`
- `itk::VolumeSplineKernelTransform`

Details about the mathematical background of these transform can be found in the paper by Davis *et. al* [20] and the papers by Rohr *et. al* [75, 76].

8.7 Metrics

In OTB, `itk::ImageToImageMetric` objects quantitatively measure how well the transformed moving image fits the fixed image by comparing the gray-scale intensity of the images. These metrics are very flexible and can work with any transform or interpolation method and do not require reduction of the gray-scale images to sparse extracted information such as edges.

The metric component is perhaps the most critical element of the registration framework. The selection of which metric to use is highly dependent on the registration problem to be solved. For example, some metrics have a large capture range while others require initialization close to the optimal position. In addition, some metrics are only suitable for comparing images obtained from the same type of sensor, while others can handle multi-sensor comparisons. Unfortunately, there are no clear-cut rules as to how to choose a metric.

The basic inputs to a metric are: the fixed and moving images, a transform and an interpolator. The method `GetValue()` can be used to evaluate the quantitative criterion at the transform parameters specified in the argument. Typically, the metric samples points within a defined region of the fixed image. For each point, the corresponding moving image position is computed using the transform with the specified parameters, then the interpolator is used to compute the moving image intensity at the mapped position.

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 OTB:

- Mean squares
  `itk::MeanSquaresImageToImageMetric`
- Normalized correlation
  `itk::NormalizedCorrelationImageToImageMetric`
- Mean reciprocal squared difference
  `itk::MeanReciprocalSquareDifferenceImageToImageMetric`
- Mutual information by Viola and Wells
  `itk::MutualInformationImageToImageMetric`
- Mutual information by Mattes
  `itk::MattesMutualInformationImageToImageMetric`
- Kullback Liebler distance metric by Kullback and Liebler
  `itk::KullbackLeiblerCompareHistogramImageToImageMetric`

- Normalized mutual information
  `itk::NormalizedMutualInformationHistogramImageToImageMetric`
- Mean squares histogram
  `itk::MeanSquaresHistogramImageToImageMetric`
- Correlation coefficient histogram
  `itk::CorrelationCoefficientHistogramImageToImageMetric`
- Cardinality Match metric
  `itk::MatchCardinalityImageToImageMetric`
- Kappa Statistics metric
  `itk::KappaStatisticImageToImageMetric`
- Gradient Difference metric
  `itk::GradientDifferenceImageToImageMetric`

In the following sections, we describe each metric type in detail. For ease of notation, we will refer to the fixed image $f(\mathbf{X})$ and transformed moving image $(m \circ T(\mathbf{X}))$ as images A and B.

8.7.1 Mean Squares Metric

The `itk::MeanSquaresImageToImageMetric` computes the mean squared pixel-wise difference in intensity between image A and B over a user defined region:

$$MS(A,B) = \frac{1}{N}\sum_{i=1}^{N}(A_i - B_i)^2 \tag{8.8}$$

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.

8.7.2 Normalized Correlation Metric

The `itk::NormalizedCorrelationImageToImageMetric` computes pixel-wise cross-correlation and normalizes it by the square root of the autocorrelation of the images:

$$NC(A,B) = -1 \times \frac{\sum_{i=1}^{N} (A_i \cdot B_i)}{\sqrt{\sum_{i=1}^{N} A_i^2 \cdot \sum_{i=1}^{N} B_i^2}} \tag{8.9}$$

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 – illumination changes – between the two images. This metric produces a cost function with sharp peaks and well defined minima. On the other hand, it has a relatively small capture radius.

8.7.3 Mean Reciprocal Square Differences

The `itk::MeanReciprocalSquareDifferenceImageToImageMetric` computes pixel-wise differences and adds them after passing them through a bell-shaped function $\frac{1}{1+x^2}$:

$$PI(A,B) = \sum_{i=1}^{N} \frac{1}{1 + \frac{(A_i - B_i)^2}{\lambda^2}} \tag{8.10}$$

A_i is the i-th pixel of Image A
B_i is the i-th pixel of Image B
N is the number of pixels considered
λ controls the capture radius

The optimal value is N and poor matches results in small measure values. The characteristics of this metric have been studied by Penney and Holden [40][69].

This image metric has the advantage of producing poor values when few pixels are considered. This makes it consistent when its computation is subject to the size of the overlap region between the images. The capture radius of the metric can be regulated with the parameter λ. The profile of this metric is very peaky. The sharp peaks of the metric help to measure spatial misalignment with high precision. Note that the notion of capture radius is used here in terms of the intensity domain, not the spatial domain. In that regard, λ should be given in intensity units and be associated with the differences in intensity that will make drop the metric by 50%.

The metric is limited to images of the same image modality. The fact that its derivative is large at the central peak is a problem for some optimizers that rely on the derivative to decrease as the extrema are reached. This metric is also sensitive to linear changes in intensity.

8.7.4 Mutual Information Metric

The `itk::MutualInformationImageToImageMetric` computes the mutual information between image A and image B. Mutual information (MI) measures how much information one random variable (image intensity in one image) tells about another random variable (image intensity in the other image). The major advantage of using MI is that the actual form of the dependency does not have to be specified. Therefore, complex mapping between two images can be modeled. This flexibility makes MI well suited as a criterion of multi-modality registration [71].

Mutual information is defined in terms of entropy. Let

$$H(A) = -\int p_A(a) \log p_A(a)\, da \tag{8.11}$$

be the entropy of random variable A, $H(B)$ the entropy of random variable B and

$$H(A,B) = \int p_{AB}(a,b) \log p_{AB}(a,b)\, da\, db \tag{8.12}$$

be the joint entropy of A and B. If A and B are independent, then

$$p_{AB}(a,b) = p_A(a) p_B(b) \tag{8.13}$$

and

$$H(A,B) = H(A) + H(B). \tag{8.14}$$

However, if there is any dependency, then

$$H(A,B) < H(A) + H(B). \tag{8.15}$$

The difference is called Mutual Information : $I(A,B)$

$$I(A,B) = H(A) + H(B) - H(A,B) \tag{8.16}$$

Parzen Windowing

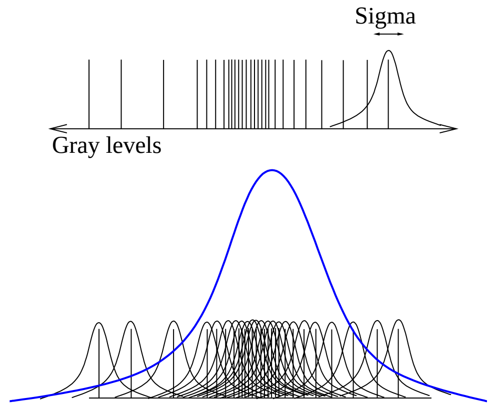


Figure 8.16: In Parzen windowing, a continuous density function is constructed by superimposing kernel functions (Gaussian function in this case) centered on the intensity samples obtained from the image.

In a typical registration problem, direct access to the marginal and joint probability densities is not available and hence the densities must be estimated from the image data. Parzen windows (also known as kernel density estimators) can be used for this purpose. In this scheme, the densities are constructed by taking intensity samples S from the image and super-positioning kernel functions $K(\cdot)$ centered on the elements of S as illustrated in Figure 8.16:

A variety of functions can be used as the smoothing kernel with the requirement that they are smooth, symmetric, have zero mean and integrate to one. For example, boxcar, Gaussian and B-spline functions are suitable candidates. A smoothing parameter is used to scale the kernel function. The larger the smoothing parameter, the wider the kernel function used and hence the smoother the density estimate. If the parameter is too large, features such as modes in the density will get smoothed out. On the other hand, if the smoothing parameter is too small, the resulting density may be too noisy. The estimation is given by the following equation.

$$p(a) \approx P^*(a) = \frac{1}{N} \sum_{s_j \in S} K\left(a - s_j\right) \tag{8.17}$$

Choosing the optimal smoothing parameter is a difficult research problem and beyond the scope of this software guide. Typically, the optimal value of the smoothing parameter will depend on the data and the number of samples used.

Viola and Wells Implementation

OTB, through ITK, has multiple implementations of the mutual information metric. One of the most commonly used is `itk::MutualInformationImageToImageMetric` and follows the method specified by Viola and Wells in [89].

In this implementation, two separate intensity samples S and R are drawn from the image: the first to compute the density, and the second to approximate the entropy as a sample mean:

$$H(A) = \frac{1}{N} \sum_{r_j \in R} \log P^*(r_j). \tag{8.18}$$

Gaussian density is used as a smoothing kernel, where the standard deviation σ acts as the smoothing parameter.

The number of spatial samples used for computation is defined using the `SetNumberOfSpatialSamples()` method. Typical values range from 50 to 100. Note that computation involves an $N \times N$ loop and hence, the computation burden becomes very expensive when a large number of samples is used.

The quality of the density estimates depends on the choice of the standard deviation of the Gaussian kernel. The optimal choice will depend on the content of the images. In our experience with the toolkit, we have found that a standard deviation of 0.4 works well for images that have been normalized to have a mean of zero and standard deviation of 1.0. The standard deviation of the fixed image and moving image kernel can be set separately using methods `SetFixedImageStandardDeviation()` and `SetMovingImageStandardDeviation()`.

Mattes et al. Implementation

Another form of mutual information metric available in ITK follows the method specified by Mattes et al. in [63] and is implemented by the `itk::MattesMutualInformationImageToImageMetric` class.

In this implementation, only one set of intensity samples is drawn from the image. Using this set, the marginal and joint probability density function (PDF) is evaluated at discrete positions or bins uniformly spread within the dynamic range of the images. Entropy values are then computed by summing over the bins.

The number of spatial samples used is set using method `SetNumberOfSpatialSamples()`. The number of bins used to compute the entropy values is set via `SetNumberOfHistogramBins()`.

Since the fixed image PDF does not contribute to the metric derivatives, it does not need to be smooth. Hence, a zero order (boxcar) B-spline kernel is used for computing the PDF. On the other hand, to ensure smoothness, a third order B-spline kernel is used to compute the moving image intensity PDF. The advantage of using a B-spline kernel over a Gaussian kernel is that the B-spline kernel has a finite support region. This is computationally attractive, as each intensity sample only affects a small number of bins and hence does not require a $N \times N$ loop to compute the metric value.

During the PDF calculations, the image intensity values are linearly scaled to have a minimum of zero and maximum of one. This rescaling means that a fixed B-spline kernel bandwidth of one can be used to handle image data with arbitrary magnitude and dynamic range.

8.7.5 Kullback-Leibler distance metric

The `itk::KullbackLeiblerCompareHistogramImageToImageMetric` is yet another information based metric. Kullback-Leibler distance measures the relative entropy between two discrete probability distributions. The distributions are obtained from the histograms of the two input images, A and B.

The Kullback-Liebler distance between two histograms is given by

$$KL(A,B) = \sum_{i}^{N} p_A(i) \times \log \frac{p_A(i)}{p_B(i)} \tag{8.19}$$

The distance is always non-negative and is zero only if the two distributions are the same. Note that the distance is not symmetric. In other words, $KL(A,B) \neq KL(B,A)$. Nevertheless, if the distributions are not too dissimilar, the difference between $KL(A,B)$ and $KL(B,A)$ is small.

The implementation in ITK is based on [16].

8.7.6 Normalized Mutual Information Metric

Given two images, A and B, the normalized mutual information may be computed as

$$NMI(A,B) = 1 + \frac{I(A,B)}{H(A,B)} = \frac{H(A)+H(B)}{H(A,B)} \tag{8.20}$$

where the entropy of the images, $H(A)$, $H(B)$, the mutual information, $I(A,B)$ and the joint entropy $H(A,B)$ are computed as mentioned in 8.7.4. Details of the implementation may be found in the [38].

8.7.7 Mean Squares Histogram

The `itk::MeanSquaresHistogramImageToImageMetric` is an alternative implementation of the Mean Squares Metric. In this implementation the joint histogram of the fixed and the mapped moving image is built first. The user selects the number of bins to use in this joint histogram. Once the joint histogram is computed, the bins are visited with an iterator. Given that each bin is associated to a pair of intensities of the form: {fixed intensity, moving intensity}, along with the number of pixels pairs in the images that fell in this bin, it is then possible to compute the sum of square distances between the intensities of both images at the quantization levels defined by the joint histogram bins.

This metric can be represented with Equation 8.21

$$MSH = \sum_{f} \sum_{m} H(f,m)(f-m)^2 \tag{8.21}$$

where $H(f,m)$ is the count on the joint histogram bin identified with fixed image intensity f and moving image intensity m.

8.7.8 Correlation Coefficient Histogram

The `itk::CorrelationCoefficientHistogramImageToImageMetric` computes the cross correlation coefficient between the intensities in the fixed image and the intensities on the mapped moving image. This metric is intended to be used in images of the same modality where the relationship between the intensities of the fixed image and the intensities on the moving images is given by a linear equation.

The correlation coefficient is computed from the Joint histogram as

$$CC = \frac{\sum_f \sum_m H(f,m)\left(f \cdot m - \overline{f} \cdot \overline{m}\right)}{\sum_f H(f)\left((f-\overline{f})^2\right) \cdot \sum_m H(m)\left((m-\overline{m})^2\right)} \tag{8.22}$$

Where $H(f,m)$ is the joint histogram count for the bin identified with the fixed image intensity f and the moving image intensity m. The values $\overline{f}$ and $\overline{m}$ are the mean values of the fixed and moving images respectively. $H(f)$ and $H(m)$ are the histogram counts of the fixed and moving images respectively. The optimal value of the correlation coefficient is 1, which would indicate a perfect straight line in the histogram.

8.7.9 Cardinality Match Metric

The `itk::MatchCardinalityImageToImageMetric` computes cardinality of the set of pixels that match exactly between the moving and fixed images. In other words, it computes the number of pixel matches and mismatches between the two images. The match is designed for label maps. All pixel mismatches are considered equal whether they are between label 1 and label 2 or between label 1 and label 500. In other words, the magnitude of an individual label mismatch is not relevant, or the occurrence of a label mismatch is important.

The spatial correspondence between the fixed and moving images is established using a `itk::Transform` using the `SetTransform()` method and an interpolator using `SetInterpolator()`. Given that we are matching pixels with labels, it is advisable to use Nearest Neighbor interpolation.

8.7.10 Kappa Statistics Metric

The `itk::KappaStatisticImageToImageMetric` computes spatial intersection of two binary images. The metric here is designed for matching pixels in two images with the same exact value, which may be set using `SetForegroundValue()`. Given two images A and B, the κ coefficient is computed as

$$\kappa = \frac{|A| \cap |B|}{|A| + |B|} \tag{8.23}$$

where $|A|$ is the number of foreground pixels in image A. This computes the fraction of area in the two images that is common to both the images. In the computation of the metric, only foreground pixels are considered.

8.7.11 Gradient Difference Metric

This `itk::GradientDifferenceImageToImageMetric` metric evaluates the difference in the derivatives of the moving and fixed images. The derivatives are passed through a function $\frac{1}{1+x}$ and then they are added. The purpose of this metric is to focus the registration on the edges of structures in the images. In this way the borders exert larger influence on the result of the registration than do the inside of the homogeneous regions on the image.

8.8 Optimizers

Optimization algorithms are encapsulated as `itk::Optimizer` objects within OTB. Optimizers are generic and can be used for applications other than registration. Within the registration framework, subclasses of `itk::SingleValuedNonLinearOptimizer` are used to optimize the metric criterion with respect to the transform parameters.

The basic input to an optimizer is a cost function object. In the context of registration, `itk::ImageToImageMetric` classes provides this functionality. The initial parameters are set using `SetInitialPosition()` and the optimization algorithm is invoked by `StartOptimization()`. Once the optimization has finished, the final parameters can be obtained using `GetCurrentPosition()`.

Some optimizers also allow rescaling of their individual parameters. This is convenient for normalizing parameters spaces where some parameters have different dynamic ranges. For example, the first parameter of `itk::Euler2DTransform` represents an angle while the last two parameters represent translations. A unit change in angle has a much greater impact on an image than a unit change in translation. This difference in scale appears as long narrow valleys in the search space making the optimization problem more difficult. Rescaling the translation parameters can help to fix this problem. Scales are represented as an `itk::Array` of doubles and set defined using `SetScales()`.

There are two main types of optimizers in OTB. In the first type we find optimizers that are suitable for dealing with cost functions that return a single value. These are indeed the most common type of cost functions, and are known as *Single Valued* functions, therefore the corresponding optimizers are known as *Single Valued* optimizers. The second type of optimizers are those suitable for managing cost functions that return multiple values at each evaluation. These cost functions are common in model-fitting problems and are known as *Multi Valued* or *Multivariate* functions. The corresponding optimizers are therefore called *MultipleValued* optimizers in OTB.

The `itk::SingleValuedNonLinearOptimizer` is the base class for the first type of optimiz-

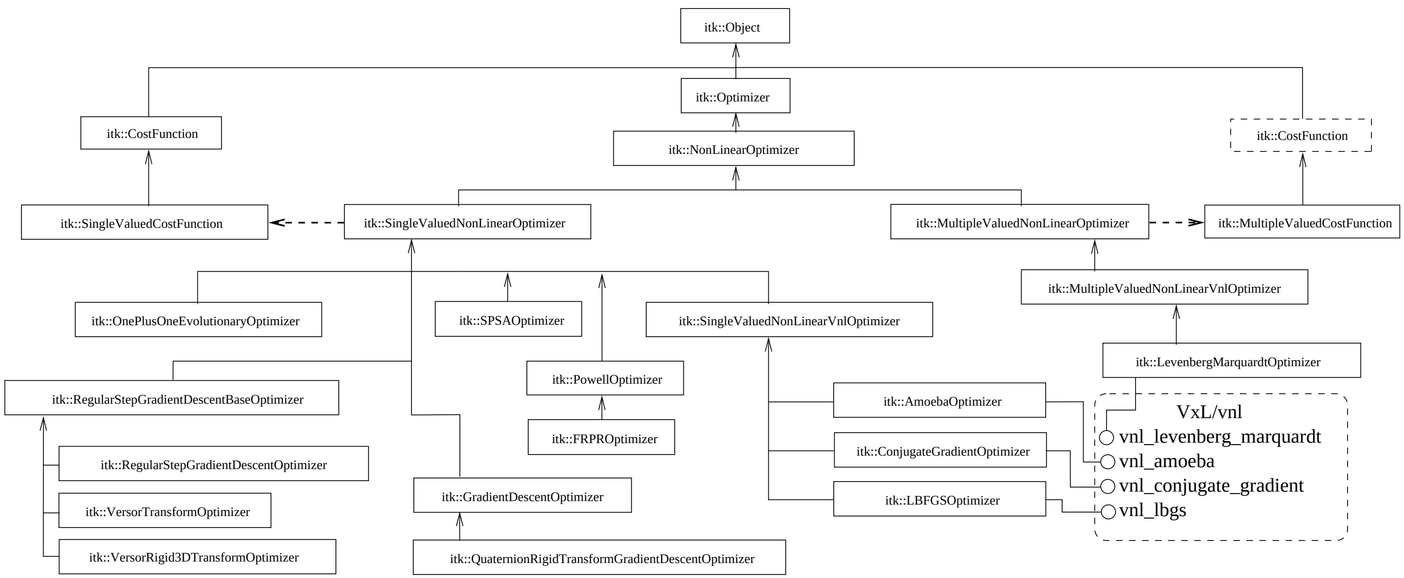


Figure 8.17: Class diagram of the optimizers hierarchy.

ers while the `itk::MultipleValuedNonLinearOptimizer` is the base class for the second type of optimizers.

The types of single valued optimizer currently available in OTB are:

- **Amoeba**: Nelder-Meade downhill simplex. This optimizer is actually implemented in the `vxl/vnl` numerics toolkit. The ITK class `itk::AmoebaOptimizer` is merely an adaptor class.
- **Conjugate Gradient**: Fletcher-Reeves form of the conjugate gradient with or without preconditioning (`itk::ConjugateGradientOptimizer`). It is also an adaptor to an optimizer in `vnl`.
- **Gradient Descent**: Advances parameters in the direction of the gradient where the step size is governed by a learning rate (`itk::GradientDescentOptimizer`).
- **Quaternion Rigid Transform Gradient Descent**: A specialized version of GradientDescentOptimizer for QuaternionRigidTransform parameters, where the parameters representing the quaternion are normalized to a magnitude of one at each iteration to represent a pure rotation (`itk::QuaternionRigidTransformGradientDescent`).
- **LBFGS**: Limited memory Broyden, Fletcher, Goldfarb and Shannon minimization. It is an adaptor to an optimizer in `vnl` (`itk::LBFGSOptimizer`).
- **LBFGSB**: A modified version of the LBFGS optimizer that allows to specify bounds for the parameters in the search space. It is an adaptor to an optimizer in `netlib`. Details on this optimizer can be found in [11, 12] (`itk::LBFGSBOptimizer`).
- **One Plus One Evolutionary**: Strategy that simulates the biological evolution of a set of samples in the search space (`itk::OnePlusOneEvolutionaryOptimizer.`). Details on this optimizer can be found in [83].
- **Regular Step Gradient Descent**: Advances parameters in the direction of the gradient where a bipartition scheme is used to compute the step size (`itk::RegularStepGradientDescentOptimizer`).
- **Powell Optimizer**: Powell optimization method. For an N-dimensional parameter space, each iteration minimizes(maximizes) the function in N (initially orthogonal) directions. This optimizer is described in [72]. (`itk::PowellOptimizer`).
- **SPSA Optimizer**: Simultaneous Perturbation Stochastic Approximation Method. This optimizer is described in `http://www.jhuapl.edu/SPSA` and in [81]. (`itk::SPSAOptimizer`).
- **Versor Transform Optimizer**: A specialized version of the RegularStepGradientDescentOptimizer for VersorTransform parameters, where the current rotation is composed with the gradient rotation to produce the new rotation versor. It follows the definition of versor gradients defined by Hamilton [39] (`itk::VersorTransformOptimizer`).

- **Versor Rigid3D Transform Optimizer**: A specialized version of the RegularStepGradientDescentOptimizer for VersorRigid3DTransform parameters, where the current rotation is composed with the gradient rotation to produce the new rotation versor. The translational part of the transform parameters are updated as usually done in a vector space. (`itk::VersorRigid3DTransformOptimizer`).

A parallel hierarchy exists for optimizing multiple-valued cost functions. The base optimizer in this branch of the hierarchy is the `itk::MultipleValuedNonLinearOptimizer` whose only current derived class is:

- **Levenberg Marquardt**: Non-linear least squares minimization. Adapted to an optimizer in `vnl` (`itk::LevenbergMarquardtOptimizer`). This optimizer is described in [72].

Figure 8.17 illustrates the full class hierarchy of optimizers in OTB. 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.

CHAPTER

NINE

Disparity Map Estimation

This chapter introduces the tools available in OTB for the estimation of geometric disparities between images.

9.1 Disparity Maps

The problem we want to deal with is the one of the automatic disparity map estimation of images acquired with different sensors. By different sensors, we mean sensors which produce images with different radiometric properties, that is, sensors which measure different physical magnitudes: optical sensors operating in different spectral bands, radar and optical sensors, etc.

For this kind of image pairs, the classical approach of fine correlation [54, 30], can not always be used to provide the required accuracy, since this similarity measure (the correlation coefficient) can only measure similarities up to an affine transformation of the radiometries.

There are two main questions which can be asked about what we want to do:

1. Can we define what the similarity is between, for instance, a radar and an optical image?
2. What does *fine registration* mean in the case where the geometric distortions are so big and the source of information can be located in different places (for instance, the same edge can be produced by the edge of the roof of a building in an optical image and by the wall-ground bounce in a radar image)?

We can answer by saying that the images of the same object obtained by different sensors are two different representations of the same reality. For the same spatial location, we have two different measures. Both informations come from the same source and thus they have a lot of common information. This relationship may not be perfect, but it can be evaluated in a relative way: different geometrical distortions are compared and the one leading to the strongest link between the two measures is kept.

When working with images acquired with the same (type of) sensor one can use a very effective approach. Since a correlation coefficient measure is robust and fast for similar images, one can afford to apply it in every pixel of one image in order to search for the corresponding HP in the other image. One can thus build a deformation grid (a sampling of the deformation map). If the sampling step of this grid is short enough, the interpolation using an analytical model is not needed and high frequency deformations can be estimated. The obtained grid can be used as a re-sampling grid and thus obtain the registered images.

No doubt, this approach, combined with image interpolation techniques (in order to estimate sub-pixel deformations) and multi-resolution strategies allows for obtaining the best performances in terms of deformation estimation, and hence for the automatic image registration.

Unfortunately, in the multi-sensor case, the correlation coefficient can not be used. We will thus try to find similarity measures which can be applied in the multi-sensor case with the same approach as the correlation coefficient.

We start by giving several definitions which allow for the formalization of the image registration problem. First of all, we define the master image and the slave image:

Definition 1 *Master image: image to which other images will be registered; its geometry is considered as the reference.*

Definition 2 *Slave image: image to be geometrically transformed in order to be registered to the master image.*

Two main concepts are the one of *similarity measure* and the one of *geometric transformation*:

Definition 3 *Let I and J be two images and let c a similarity criterion, we call similarity measure any scalar, strictly positive function*

$$S_c(I,J) = f(I,J,c). \tag{9.1}$$

S_c has an absolute maximum when the two images I and J are identical in the sense of the criterion c.

Definition 4 *A geometric transformation T is an operator which, applied to the coordinates (x,y) of a point in the slave image, gives the coordinates (u,v) of its HP in the master image:*

$$\begin{pmatrix} u \\ v \end{pmatrix} = T \begin{pmatrix} x \\ y \end{pmatrix} \tag{9.2}$$

Finally we introduce a definition for the image registration problem:

Definition 5 *Registration problem:*

1. *determine a geometric transformation T which maximizes the similarity between a master image I and the result of the transformation $T \circ J$:*

$$Arg \max_{T}(S_c(I, T \circ J)); \tag{9.3}$$

2. *re-sampling of J by applying T.*

9.1.1 Geometric deformation modeling

The geometric transformation of definition 4 is used for the correction of the existing deformation between the two images to be registered. This deformation contains informations which are linked to the observed scene and the acquisition conditions. They can be classified into 3 classes depending on their physical source:

1. deformations linked to the mean attitude of the sensor (incidence angle, presence or absence of yaw steering, etc.);
2. deformations linked to a stereo vision (mainly due to the topography);
3. deformations linked to attitude evolution during the acquisition (vibrations which are mainly present in push-broom sensors).

These deformations are characterized by their spatial frequencies and intensities which are summarized in table 9.1.

Depending on the type of deformation to be corrected, its model will be different. For example, if the only deformation to be corrected is the one introduced by the mean attitude, a physical model for the acquisition geometry (independent of the image contents) will be enough. If the sensor is not well known, this deformation can be approximated by a simple analytical model. When the deformations to be modeled are high frequency, analytical (parametric) models are not suitable for a fine registration. In this case, one has to use a fine sampling of the

	Intensity	Spatial Frequency
Mean Attitude	Strong	Low
Stereo	Medium	High and Medium
Attitude evolution	Low	Low to Medium

Table 9.1: Characterization of the geometric deformation sources

deformation, that means the use of deformation grids. These grids give, for a set of pixels of the master image, their location in the slave image.

The following points summarize the problem of the deformation modeling:

1. An analytical model is just an approximation of the deformation. It is often obtained as follows:
   (a) Directly from a physical model without using any image content information.
   (b) By estimation of the parameters of an a priori model (polynomial, affine, etc.). These parameters can be estimated:
      i. Either by solving the equations obtained by taking HP. The HP can be manually or automatically extracted.
      ii. Or by maximization of a global similarity measure.
2. A deformation grid is a sampling of the deformation map.

The last point implies that the sampling period of the grid must be short enough in order to account for high frequency deformations (Shannon theorem). Of course, if the deformations are non stationary (it is usually the case of topographic deformations), the sampling can be irregular.

As a conclusion, we can say that definition 5 poses the registration problem as an optimization problem. This optimization can be either global or local with a similarity measure which can also be either local or global. All this is synthesized in table 9.2.

The ideal approach would consist in a registration which is locally optimized, both in similarity and deformation, in order to have the best registration quality. This is the case when deformation grids with dense sampling are used. Unfortunately, this case is the most computationally heavy and one often uses either a low sampling rate of the grid, or the evaluation of the similarity in a small set of pixels for the estimation of an analytical model. Both of these

Geometric model	Similarity measure	Optimization of the deformation
Physical model	None	Global
Analytical model with a priori HP	Local	Global
Analytical model without a priori HP	Global	Global
Grid	Local	Local

Table 9.2: Approaches to image registration

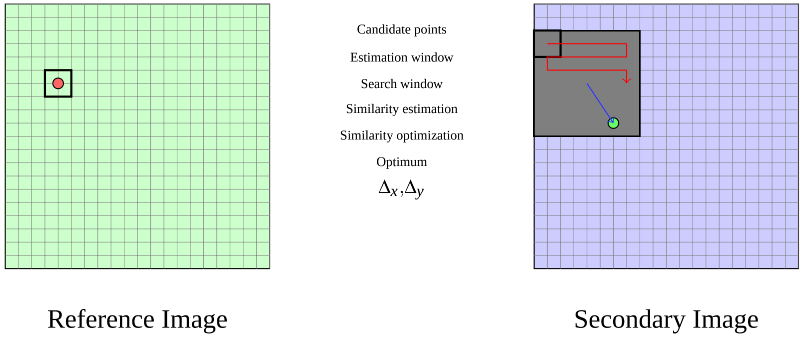


Figure 9.1: Estimation of the correlation surface.

choices lead to local registration errors which, depending on the topography, can amount several pixels.

Even if this registration accuracy can be enough in many applications, (ortho-registration, import into a GIS, etc.), it is not acceptable in the case of data fusion, multi-channel segmentation or change detection [86]. This is why we will focus on the problem of deformation estimation using dense grids.

9.1.2 Similarity measures

The fine modeling of the geometric deformation we are looking for needs for the estimation of the coordinates of nearly every pixel in the master image inside the slave image. In the classical mono-sensor case where we use the correlation coefficient we proceed as follows.

The geometric deformation is modeled by local rigid displacements. One wants to estimate the coordinates of each pixel of the master image inside the slave image. This can be represented by a displacement vector associated to every pixel of the master image. Each of the two components (lines and columns) of this vector field will be called deformation grid.

We use a small window taken in the master image and we test the similarity for every possible shift within an exploration area inside the slave image (figure 9.1).

That means that for each position we compute the correlation coefficient. The result is a correlation surface whose maximum gives the most likely local shift between both images:

$$\rho_{I,J}(\Delta x,\Delta y) = \frac{1}{N}\frac{\sum_{x,y}(I(x,y)-m_I)(J(x+\Delta x,y+\Delta y)-m_J)}{\sigma_I\sigma_J}. \tag{9.4}$$

In this expression, N is the number of pixels of the analysis window, m_I and m_J are the estimated mean values inside the analysis window of respectively image I and image J and σ_I and σ_J are their standard deviations.

Quality criteria can be applied to the estimated maximum in order to give a confidence factor to the estimated shift: width of the peak, maximum value, etc. Sub-pixel shifts can be measured by applying fractional shifts to the sliding window. This can be done by image interpolation.

The interesting parameters of the procedure are:

- The size of the exploration area: it determines the computational load of the algorithm (we want to reduce it), but it has to be large enough in order to cope with large deformations.
- The size of the sliding window: the robustness of the correlation coefficient estimation increases with the window size, but the hypothesis of local rigid shifts may not be valid for large windows.

The correlation coefficient cannot be used with original grey-level images in the multi-sensor case. It could be used on extracted features (edges, etc.), but the feature extraction can introduce localization errors. Also, when the images come from sensors using very different modalities, it can be difficult to find similar features in both images. In this case, one can try to find the similarity at the pixel level, but with other similarity measures and apply the same approach as we have just described.

The concept of similarity measure has been presented in definition 3. The difficulty of the procedure lies in finding the function f which properly represents the criterion c. We also need that f be easily and robustly estimated with small windows. We extend here what we proposed in [43].

9.1.3 The correlation coefficient

We remind here the computation of the correlation coefficient between two image windows I and J. The coordinates of the pixels inside the windows are represented by (x,y):

$$\rho(I,J) = \frac{1}{N}\frac{\sum_{x,y}(I(x,y)-m_I)(J(x,y)-m_J)}{\sigma_I\sigma_J}. \tag{9.5}$$

In order to qualitatively characterize the different similarity measures we propose the following experiment. We take two images which are perfectly registered and we extract a small window of size $N \times M$ from each of the images (this size is set to 101×101 for this experiment). For the master image, the window will be centered on coordinates (x_0, y_0) (the center of the image) and for the slave image, it will be centered on coordinates $(x_0 + \Delta x, y_0)$. With different values of Δx (from -10 pixels to 10 pixels in our experiments), we obtain an estimate of $\rho(I,J)$ as a function of Δx, which we write as $\rho(\Delta x)$ for short. The obtained curve should have a maximum for $\Delta x = 0$, since the images are perfectly registered. We would also like to have an absolute maximum with a high value and with a sharp peak, in order to have a good precision for the shift estimate.

9.2 Disparity Map Estimation Framework

Taking figure 9.1 as a starting point, we can generalize the approach by letting the user choose:

- the similarity measure;
- the geometric transform to be estimated (see definition 4);

In order to do this, we will use the ITK registration framework locally on a set of nodes. Once the disparity is estimated on a set of nodes, we will use it to generate a deformation field: the dense, regular vector field which gives the translation to be applied to a pixel of the secondary image to be positioned on its homologous point of the master image.

9.3 Simple Disparity Map Estimation

The source code for this example can be found in the file
`Examples/DisparityMap/SimpleDisparityMapEstimationExample.cxx`.

This example demonstrates the use of the `otb::DisparityMapEstimationMethod`, along with the `otb::NearestPointDeformationFieldGenerator`. The first filter performs deformation estimation according to a given transform, using embedded ITK registration framework. It takes as input a possibly non regular point set and produces a point set with associated point data representing the deformation.

The second filter generates a deformation field by using nearest neighbor interpolation on the deformation values from the point set. More advanced methods for deformation field interpolation are also available.

The first step toward the use of these filters is to include the proper header files.

```
#include "otbDisparityMapEstimationMethod.h"
#include "itkTranslationTransform.h"
#include "itkNormalizedCorrelationImageToImageMetric.h"
#include "itkWindowedSincInterpolateImageFunction.h"
#include "itkZeroFluxNeumannBoundaryCondition.h"
#include "itkGradientDescentOptimizer.h"
#include "otbNearestPointDeformationFieldGenerator.h"
#include "itkWarpImageFilter.h"
```

Then we must decide what pixel type to use for the image. We choose to do all the computation in floating point precision and rescale the results between 0 and 255 in order to export PNG images.

```
  typedef double        PixelType;
  typedef unsigned char OutputPixelType;
```

The images are defined using the pixel type and the dimension. Please note that the `otb::NearestPointDeformationFieldGenerator` generates a `otb::VectorImage` to represent the deformation field in both image directions.

```
  typedef otb::Image<PixelType,Dimension>       ImageType;
  typedef otb::Image<OutputPixelType,Dimension> OutputImageType;
```

The next step is to define the transform we have chosen to model the deformation. In this example the deformation is modeled as a `itk::TranslationTransform`.

```
  typedef itk::TranslationTransform<double,Dimension> TransformType;
  typedef TransformType::ParametersType ParametersType;
```

Then we define the metric we will use to evaluate the local registration between the fixed and the moving image. In this example we choosed the `itk::NormalizedCorrelationImageToImageMetric`.

```
  typedef itk::NormalizedCorrelationImageToImageMetric<ImageType,
                ImageType> MetricType;
```

Disparity map estimation implies evaluation of the moving image at non-grid position. Therefore, an interpolator is needed. In this example we choosed the `itk::WindowedSincInterpolateImageFunction`.

```
typedef itk::Function::HammingWindowFunction<3> WindowFunctionType;
typedef itk::ZeroFluxNeumannBoundaryCondition<ImageType> ConditionType;
typedef itk::WindowedSincInterpolateImageFunction<ImageType,3,
             WindowFunctionType,ConditionType ,double> InterpolatorType;
```

To perform local registration, an optimizer is needed. In this example we choosed the `itk::GradientDescentOptimizer`.

```
typedef itk::GradientDescentOptimizer OptimizerType;
```

Now we will define the point set to represent the point where to compute local disparity.

```
typedef itk::PointSet<ParametersType,Dimension> PointSetType;
```

Now we define the disparity map estimation filter.

```
typedef otb::DisparityMapEstimationMethod<ImageType,
             ImageType,PointSetType> DMEstimationType;
typedef DMEstimationType::SizeType SizeType;
```

The input image reader also has to be defined.

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

Two readers are instantiated : one for the fixed image, and one for the moving image.

```
ReaderType::Pointer fixedReader = ReaderType::New();
ReaderType::Pointer movingReader = ReaderType::New();

fixedReader->SetFileName(argv[1]);
movingReader->SetFileName(argv[2]);
fixedReader->Update();
movingReader->Update();
```

We will the create a regular point set where to compute the local disparity.

```
  SizeType fixedSize =
    fixedReader->GetOutput()->GetLargestPossibleRegion().GetSize();
  unsigned int NumberOfXNodes = (fixedSize[0]-2*atoi(argv[7])-1)
    /atoi(argv[5]);
  unsigned int NumberOfYNodes = (fixedSize[1]-2*atoi(argv[7])-1)
    /atoi(argv[6]);

  ImageType::IndexType firstNodeIndex;
  firstNodeIndex[0] = atoi(argv[7]);
  firstNodeIndex[1] = atoi(argv[7]);

  PointSetType::Pointer  nodes = PointSetType::New();
  unsigned int nodeCounter = 0;

  for(unsigned int x=0; x<NumberOfXNodes; x++)
    {
      for(unsigned int y=0; y<NumberOfYNodes; y++)
{
  PointType p;
  p[0] = firstNodeIndex[0]+x*atoi(argv[5]);
  p[1] = firstNodeIndex[1]+y*atoi(argv[6]);
  nodes->SetPoint( nodeCounter++,p);
}
    }
```

We build the transform, interpolator, metric and optimizer for the disparity map estimation filter.

```
  TransformType::Pointer transform = TransformType::New();

  OptimizerType::Pointer optimizer = OptimizerType::New();
  optimizer->MinimizeOn();
  optimizer->SetLearningRate(atof(argv[9]));
  optimizer->SetNumberOfIterations(atoi(argv[10]));

  InterpolatorType::Pointer interpolator = InterpolatorType::New();

  MetricType::Pointer metric = MetricType::New();
  metric->SetSubtractMean(true);
```

We then set up the disparity map estimation filter. This filter will perform a local registration at each point of the given point set using the ITK registration framework. It will produce a point set whose point data reflects the disparity locally around the associated point.

Point data will contains the following data :

1. The final metric value found in the registration process,

2. the deformation value in the first image direction,
3. the deformation value in the second image direction,
4. the final parameters of the transform.

Please note that in the case of a itk::TranslationTransform, the deformation values and the transform parameters are the same.

```
DMEstimationType::Pointer dmestimator = DMEstimationType::New();

dmestimator->SetTransform(transform);
dmestimator->SetOptimizer(optimizer);
dmestimator->SetInterpolator(interpolator);
dmestimator->SetMetric(metric);

SizeType windowSize, explorationSize;
explorationSize.Fill(atoi(argv[7]));
windowSize.Fill(atoi(argv[8]));

dmestimator->SetWinSize(windowSize);
dmestimator->SetExploSize(explorationSize);
```

The initial transform parameters can be set via the SetIntialTransformParameters() method. In our case, we simply fill the parameter array with null values.

```
DMEstimationType::ParametersType
  initialParameters(transform->GetNumberOfParameters() );
initialParameters[0] = 0.0;
initialParameters[1] = 0.0;
dmestimator->SetInitialTransformParameters(initialParameters);
```

Now we can set the input for the deformation field estimation filter. Fixed image can be set using the SetFixedImage() method, moving image can be set using the SetMovingImage(), and input point set can be set using the SetPointSet() method.

```
dmestimator->SetFixedImage(fixedReader->GetOutput());
dmestimator->SetMovingImage(movingReader->GetOutput());
dmestimator->SetPointSet(nodes);
```

Once the estimation has been performed by the otb::DisparityMapEstimationMethod, one can generate the associated deformation field (that means translation in first and second image direction). It will be represented as a otb::VectorImage.

```
typedef otb::VectorImage<PixelType,Dimension> DeformationFieldType;
```

For the deformation field estimation, we will use the `otb::NearestPointDeformationFieldGenerator`. This filter will perform a nearest neighbor interpolation on the deformation values in the point set data.

```
typedef otb::NearestPointDeformationFieldGenerator<PointSetType,
             DeformationFieldType> GeneratorType;
```

The disparity map estimation filter is instanciated.

```
GeneratorType::Pointer generator = GeneratorType::New();
```

We must then specify the input point set using the `SetPointSet()` method.

```
generator->SetPointSet(dmestimator->GetOutput());
```

One must also specify the origin, size and spacing of the output deformation field.

```
generator->SetOutputOrigin(fixedReader->GetOutput()->GetOrigin());
generator->SetOutputSpacing(fixedReader->GetOutput()->GetSpacing());
generator->SetOutputSize(fixedReader->GetOutput()
 ->GetLargestPossibleRegion().GetSize());
```

The local registration process can lead to wrong deformation values and transform parameters. To Select only points in point set for which the registration process was succesful, one can set a threshold on the final metric value : points for which the absolute final metric value is below this threshold will be discarded. This threshold can be set with the `SetMetricThreshold()` method.

```
generator->SetMetricThreshold(atof(argv[11]));
```

The following classes provide similar functionality:

- otb::NNearestPointsLinearInterpolateDeformationFieldGenerator
- otb::BSplinesInterpolateDeformationFieldGenerator
- otb::NearestTransformDeformationFieldGenerator
- otb::NNearestTransformsLinearInterpolateDeformationFieldGenerator
- otb::BSplinesInterpolateTransformDeformationFieldGenerator

Now we can warp our fixed image according to the estimated deformation field. This will be performed by the itk::WarpImageFilter. First, we define this filter.

```
typedef itk::WarpImageFilter<ImageType,ImageType,
             DeformationFieldType> ImageWarperType;
```

Then we instantiate it.

```
ImageWarperType::Pointer warper = ImageWarperType::New();
```

We set the input image to warp using the SetInput() method, and the deformation field using the SetDeformationField() method.

```
warper->SetInput(movingReader->GetOutput());
warper->SetDeformationField(generator->GetOutput());
warper->SetOutputOrigin(fixedReader->GetOutput()->GetOrigin());
warper->SetOutputSpacing(fixedReader->GetOutput()->GetSpacing());
```

In order to write the result to a PNG file, we will rescale it on a proper range.

```
typedef itk::RescaleIntensityImageFilter<ImageType,
             OutputImageType> RescalerType;

RescalerType::Pointer outputRescaler = RescalerType::New();
outputRescaler->SetInput(warper->GetOutput());
outputRescaler->SetOutputMaximum(255);
outputRescaler->SetOutputMinimum(0);
```

We can now write the image to a file. The filters are executed by invoking the `Update()` method.

```
typedef otb::ImageFileWriter<OutputImageType> WriterType;

WriterType::Pointer outputWriter = WriterType::New();
outputWriter->SetInput(outputRescaler->GetOutput());
outputWriter->SetFileName(argv[4]);
outputWriter->Update();
```

We also want to write the deformation field along the first direction to a file. To achieve this we will use the `otb::MultiToMonoChannelExtractROI` filter.

```
typedef otb::MultiToMonoChannelExtractROI<PixelType,
             PixelType> ChannelExtractionFilterType;

ChannelExtractionFilterType::Pointer channelExtractor
  = ChannelExtractionFilterType::New();

channelExtractor->SetInput(generator->GetOutput());
channelExtractor->SetChannel(1);

RescalerType::Pointer fieldRescaler = RescalerType::New();
fieldRescaler->SetInput(channelExtractor->GetOutput());
fieldRescaler->SetOutputMaximum(255);
fieldRescaler->SetOutputMinimum(0);

WriterType::Pointer fieldWriter = WriterType::New();
fieldWriter->SetInput(fieldRescaler->GetOutput());
fieldWriter->SetFileName(argv[3]);
fieldWriter->Update();
```

Figure 9.2 shows the result of applying disparity map estimation on a regular point set, followed by deformation field estimation and fixed image resampling on an Ikonos image. The moving image is the fixed image warped with a sinusoidal deformation with a 3-pixels amplitude and a 170-pixels period. Please note that there are more efficient ways to interpolate the deformation field than nearest neighbor, including BSplines fitting.

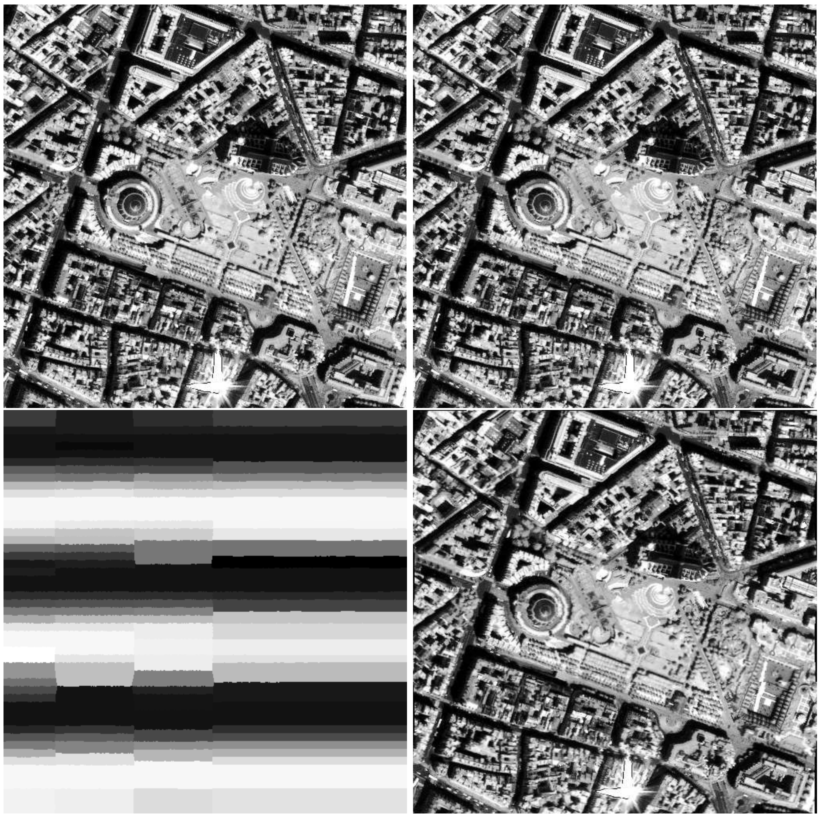


Figure 9.2: From left to right and top to bottom: fixed input image, moving image with a sinusoid deformation, estimated deformation field in the horizontal direction, resampled moving image.

CHAPTER

TEN

Ortho-registration

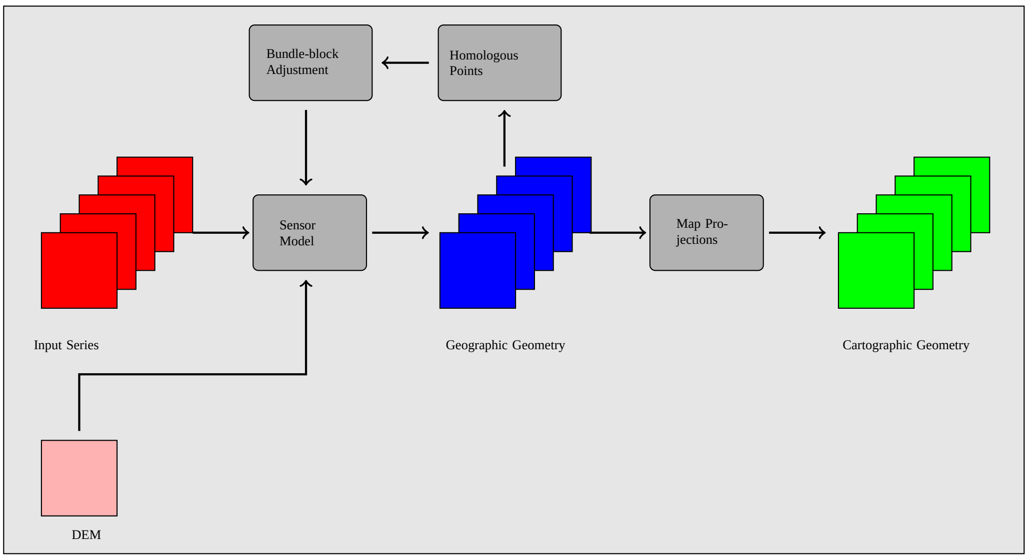


Figure 10.1: Image Ortho-registration Procedure.

This chapter introduces the functionnalities available in OTB for image ortho-registration. We define ortho-registration as the procedure allowing to transform an image in sensor geometry to a geographic or cartographic projection.

Figure 10.1 shows a synoptic view of the different steps involved in a classical ortho-registration processing chain able to deal with image series. These steps are the following:

- Sensor modelling: the geometric sensor model allows to convert image coordinates (line, column) into geographic coordinates (latitude, longitude); a rigorous modelling needs a digital elevation model (DEM) in order to take into account the terrain topography.

- Bundle-block adjustment: in the case of image series, the geometric models and their parameters can be refined by using homologous points between the images. This is an optional step and not currently implemented in OTB.
- Map projection: this step allows to go from geographic coordinates to some specific cartographic projection as Lambert, Mercator or UTM.

10.1 Sensor Models

A sensor model is a set of equations giving the relationship between image pixel (l,c) coordinates and ground (X,Y) coordinates for every pixel in the image. Typically, the ground coordinates are given in a geographic projection (latitude, longitude). The sensor model can be expressed either from image to ground – forward model – or from ground to image – inverse model. This can be written as follows:

$$\begin{array}{ll} \textit{Forward} & \\ X = f_x(l,c,h,\vec{\theta}) & Y = f_y(l,c,h,\vec{\theta}) \\ & \\ \textit{Inverse} & \\ l = g_l(X,Y,h,\vec{\theta}) & c = g_c(X,Y,h,\vec{\theta}) \end{array}$$

Where $\vec{\theta}$ is the set of parameters which describe the sensor and the acquisition geometry (platform altitude, viewing angle, focal length for optical sensors, doppler centroid for SAR images, etc.).

In OTB, sensor models are implemented as `itk::Transforms` (see section 8.6 for details), which are the appropiate way to express coordinate changes. The base class for sensor models is `otb::SensorModelBase` from which the classes `otb::InverseSensorModel` and `otb::ForwardSensorModel` inherit.

As one may note from the model equations, the height of the ground, h, must be known. Usually, that means that a Digital Elevation Model, DEM, will be used.

10.1.1 Types of Sensor Models

There exist two main types of sensor models. On one hand, we have the so-called *physical models*, which are rigorous, complex, eventually highly non-linear equations of the sensor geometry. As such, they are difficult to inverse (obtain the inverse model from the forward one and vice-versa). They have the significant advantage of having parameters with physical meaning (angles, distances, etc.). They are specific of each sensor, which means that a library of models is required in the software. A library which has to be updated every time a new sensor is available.

On the other hand, we have general analytical models, which approximate the physical models. These models can take the form of polynomials or ratios of polynomials, the so-called rational polynomial functions or Rational Polynomial Coefficients, RPC, also known as *Rapid Positioning Capability*. Since they are approximations, they are less accurate than the physical models. However, the achieved accuracy is usually high: in the case of Pléiades, RPC models have errors lower than 0.02 pixels with respect to the physical model. Since these models have a standard form they are easier to use and implement. However, they have the drawback of having parameters (coefficients, actually) without physical meaning.

OTB, through the use of the OSSIM library – `http://www.ossim.org` – offers models for most of current sensors either through a physical or an analytical approach. This is transparent for the user, since the geometrical model for a given image is instantiated using the information stored in its meta-data.

10.1.2 Using Sensor Models

The transformation of an image in sensor geometry to geographic geometry can be done using the following steps.

1. Read image meta-data and instantiate the model with the given parameters.
2. Define the ROI in ground coordinates (this is your output pixel array)
3. Iterate through the pixels of coordinates (X,Y):
   (a) Get h from the DEM
   (b) Compute $(c,l) = G(X,Y,h,\vec{\theta})$
   (c) Interpolate pixel values if (c,l) are not grid coordinates.

Actually, in OTB, you don't have to manually instantiate the sensor model which is appropriate to your image. That is, you don't have to manually choose a SPOT5 or a Quickbird sensor model. This task is automatically performed by the `otb::ImageFileReader` class in a similar way as the image format recognition is done. The appropriate sensor model will then be included in the image meta-data, so you can access it when needed.

The source code for this example can be found in the file
`Examples/Projections/SensorModelExample.cxx`.

This example illustrates how to use the sensor model read from image meta-data in order to perform ortho-rectification. This is a very basic, step-by-step example, so you understand the different components involved in the process. When performing real ortho-rectifications, you can use the example presented in section 10.3.

We will start by including the header file for the inverse sensor model.

```
#include "otbInverseSensorModel.h"
```

As explained before, the first thing to do is to create the sensor model in order to transform the ground coordinates in sensor geometry coordinates. The geoetric model will automatically be created by the image file reader. So we begin by declaring the types for the input image and the image reader.

```
  typedef otb::Image<unsigned int, 2>     ImageType;
  typedef otb::ImageFileReader<ImageType>  ReaderType;
  ReaderType::Pointer                  reader=ReaderType::New();
  reader->SetFileName(argv[1]);

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

We have just instantiated the reader and set the file name, but the image data and meta-data has not yet been accessed by it. Since we need the creation of the sensor model and all the image information (size, spacing, etc.), but we do not want to read the pixel data – it could be huge – we just ask the reader to generate the output information needed.

```
  reader->GenerateOutputInformation();

  std::cout << "Original input imagine spacing: "<<
    reader->GetOutput()->GetSpacing() << std::endl;;
```

We can now instantiate the sensor model – an inverse one, since we want to convert ground coordinates to sensor geometry. Note that the choice of the specific model (SPOT5, Ikonos, etc.) is done by the reader and we only need to instantiate a generic model.

```
  typedef otb::InverseSensorModel<double>  ModelType;
  ModelType::Pointer   model= ModelType::New();
```

The model is parameterized by passing to it the *keyword list* containing the needed information.

```
  model->SetImageGeometry(reader->GetOutput()->GetImageKeywordlist());
```

Since we can not be sure that the image we read actually has sensor model information, we must check for the model validity.

```
if(!model)
  {
  std::cerr << "Unable to create a model" << std::endl;
  return 1;
  }
```

The types for the input and output coordinate points can be now declared. The same is done for the index types.

```
ModelType::OutputPointType inputPoint;
typedef itk::Point <double, 2>   PointType;
PointType  outputPoint;
```

```
ImageType::IndexType   currentIndex;
ImageType::IndexType   currentIndexBis;
ImageType::IndexType   pixelIndexBis;
```

We will now create the output image over which we will iterate in order to transform ground coordinates to sensor coordinates and get the corresponding pixel values.

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

ImageType::PixelType  pixelValue;

ImageType::IndexType    start;
start[0]=0;
start[1]=0;

ImageType::SizeType    size;
size[0]=atoi(argv[5]);
size[1]=atoi(argv[6]);
```

The spacing in y direction is negative since origin is the upper left corner.

```
ImageType::SpacingType    spacing;
```

```
spacing[0]=0.00001;
spacing[1]=-0.00001;


ImageType::PointType  origin;
origin[0]=strtod(argv[3], NULL);         //longitude
origin[1]=strtod(argv[4], NULL);         //lattitude

ImageType::RegionType  region;

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

outputImage->SetOrigin(origin);
outputImage->SetRegions(region);
outputImage->SetSpacing(spacing);
outputImage->Allocate();
```

We will now instantiate an extractor filter in order to get input regions by manual tiling.

```
typedef itk::ExtractImageFilter<ImageType,ImageType>   ExtractType;
ExtractType::Pointer   extract=ExtractType::New();
```

Since the transformed coordinates in sensor geometry may not be integer ones, we will need an interpolator to retrieve the pixel values (note that this assumes that the input image was correctly sampled by the acquisition system).

```
typedef itk::LinearInterpolateImageFunction<ImageType, double>
                                                  InterpolatorType;
InterpolatorType::Pointer interpolator=InterpolatorType::New();
```

We proceed now to create the image writer. We will also use a writer plugged to the output of the extractor filter which will write the temporary extracted regions. This is just for monitoring the process.

```
typedef otb::Image<unsigned char, 2>    CharImageType;
typedef otb::ImageFileWriter<CharImageType>  CharWriterType;
typedef otb::ImageFileWriter<ImageType>  WriterType;
WriterType::Pointer                  extractorWriter=WriterType::New();
CharWriterType::Pointer                  writer=CharWriterType::New();
extractorWriter->SetFileName("image_temp.jpeg");
extractorWriter->SetInput(extract->GetOutput());
```

Since the output pixel type and the input pixel type are different, we will need to rescale the intensity values before writing them to a file.

```
typedef itk::RescaleIntensityImageFilter<ImageType,CharImageType>
                                                        RescalerType;
RescalerType::Pointer                  rescaler=RescalerType::New();
rescaler->SetOutputMinimum(10);
rescaler->SetOutputMaximum(255);
```

The tricky part starts here. Note that this example is only intended for pedagogic purposes and that you do not need to proceed as this. See the example in section 10.3 in order to code ortho-rectification chains in a very simple way.

You want to go on? OK. You have been warned.

We will start by declaring an image region iterator and some convenience variables.

```
typedef itk::ImageRegionIteratorWithIndex<ImageType>IteratorType;

unsigned int NumberOfStreamDivisions;
if (atoi(argv[7])==0)
  {
  NumberOfStreamDivisions=10;
  }
else
  {
  NumberOfStreamDivisions=atoi(argv[7]);
  }


unsigned int count=0;
unsigned int It, j, k;
int max_x, max_y, min_x, min_y;
ImageType::IndexType    iterationRegionStart;
ImageType::SizeType    iteratorRegionSize;
ImageType::RegionType  iteratorRegion;
```

The loop starts here.

```
for(count=0;count<NumberOfStreamDivisions;count++)
  {
  iteratorRegionSize[0]=atoi(argv[5]);
```

```
    if (count==NumberOfStreamDivisions-1)
      {
      iteratorRegionSize[1]=(atoi(argv[6]))-((int)(((atoi(argv[6]))/
      NumberOfStreamDivisions)+0.5))*(count);
      iterationRegionStart[1]=(atoi(argv[5]))-(iteratorRegionSize[1]);
      }
    else
      {
      iteratorRegionSize[1]=(int)(((atoi(argv[6]))/
   NumberOfStreamDivisions)+0.5);
      iterationRegionStart[1]=count*iteratorRegionSize[1];
      }
    iterationRegionStart[0]=0;
    iteratorRegion.SetSize(iteratorRegionSize);
    iteratorRegion.SetIndex(iterationRegionStart);
```

We create an array for storing the pixel indexes.

```
    unsigned int pixelIndexArrayDimension= iteratorRegionSize[0]*iteratorRegionSize[1]*2;
    int *pixelIndexArray=new int[pixelIndexArrayDimension];
    int *currentIndexArray=new int[pixelIndexArrayDimension];
```

We create an iterator for each piece of the image, and we iterate over them.

```
    IteratorType outputIt(outputImage, iteratorRegion);

    It=0;
    for (outputIt.GoToBegin(); !outputIt.IsAtEnd(); ++outputIt)
      {
```

We get the current index.

```
      currentIndex=outputIt.GetIndex();
```

We transform the index to physical coordinates.

```
      outputImage->TransformIndexToPhysicalPoint(currentIndex, outputPoint);
```

We use the sensor model to get the pixel coordinates in the input image and we transform this coodinates to an index. Then we store the index in the array. Note that the `TransformPoint()` method of the model has been overloaded so that it can be used with a 3D point when the height of the ground point is known (DEM availability).

```
    inputPoint = model->TransformPoint(outputPoint);

    pixelIndexArray[It]=static_cast<int>(inputPoint[0]);
    pixelIndexArray[It+1]=static_cast<int>(inputPoint[1]);

    currentIndexArray[It]=static_cast<int>(currentIndex[0]);
    currentIndexArray[It+1]=static_cast<int>(currentIndex[1]);

    It=It+2;
    }
```

By this point, we have stored all the indexes we need for the piece of image we are processing. Now we can compute the bounds of the area in the input image we need to extract.

```
  max_x=pixelIndexArray[0];
  min_x=pixelIndexArray[0];
  max_y=pixelIndexArray[1];
  min_y=pixelIndexArray[1];



  for (j=0;j<It;j++)
    {
    if(j%2==0 && pixelIndexArray[j]>max_x){max_x=pixelIndexArray[j];}
    if(j%2==0 && pixelIndexArray[j]<min_x){min_x=pixelIndexArray[j];}
    if(j%2!=0 && pixelIndexArray[j]>max_y){max_y=pixelIndexArray[j];}
    if(j%2!=0 && pixelIndexArray[j]<min_y){min_y=pixelIndexArray[j];}
    }
```

We can now set the parameters for the extractor using a little bit of margin in order to cope with irregular geometric distortions which could be due to topography, for instance.

```
  ImageType::RegionType             extractRegion;

  ImageType::IndexType       extractStart;

  if (min_x<10 && min_y<10)
    {
    extractStart[0]=0;
    extractStart[1]=0;
    }
```

```
    else
      {
      extractStart[0]=min_x-10;
      extractStart[1]=min_y-10;
      }

    ImageType::SizeType       extractSize;

    extractSize[0]=(max_x-min_x)+20;
    extractSize[1]=(max_y-min_y)+20;
    extractRegion.SetSize(extractSize);
    extractRegion.SetIndex(extractStart);

    extract->SetExtractionRegion(extractRegion);
    extract->SetInput(reader->GetOutput());
    extractorWriter->Update();
```

We give the input image to the interpolator and we loop through the index array in order to get the corresponding pixel values. Note that for every point we check whether it is inside the extracted region.

```
    interpolator->SetInputImage(extract->GetOutput());

    for ( k=0; k<It/2; k++)
      {
      pixelIndexBis[0]= pixelIndexArray[2*k];
      pixelIndexBis[1]= pixelIndexArray[2*k+1];
      currentIndexBis[0]= currentIndexArray[2*k];
      currentIndexBis[1]= currentIndexArray[2*k+1];

      if (interpolator->IsInsideBuffer(pixelIndexBis))
{
pixelValue=int (interpolator->EvaluateAtIndex(pixelIndexBis));
}
      else
{
pixelValue=0;
}

      outputImage->SetPixel(currentIndexBis,pixelValue);
      }
    delete pixelIndexArray;
    delete currentIndexArray;

    }
```

So we are done. We can now write the output image to a file after performing the intensity rescaling.

```
writer->SetFileName(argv[2]);

rescaler->SetInput(outputImage);

writer->SetInput(rescaler->GetOutput());
writer->Update();
```

10.1.3 Limits of the Approach

As you may understand by now, accurate geo-referencing needs accurate DEM and also accurate sensor models and parameters. In the case where we have several images acquired over the same area by different sensors or different geometric configurations, geo-referencing (geographical coordinates) or ortho-rectification (cartographic coordinates) is not usually enough. Indeed, when working with image series we usually want to compare them (fusion, change detection, etc.) at the pixel level.

Since common DEM and sensor parameters do not allow for such an accuracy, we have to use clever strategies to improve the co-registration of the images. The classical one consists in refining the sensor parameters by taking homologous points between the images to co-register. This is called bundle block adjustment and will be implemented in comming versions of OTB.

Even if the model parameters are refined, errors due to DEM accuracy can not be eliminated. In this case, image to image registration can be applied. This approaches are presented in chapters 8 and 9.

10.2 Map Projections

Map projections describe the link between geographic coordinates and cartographic ones. So map projections allow to represent a 2-dimensional manifold of a 3-dimensional space (the Earth surface) in a 2-dimensional space (a map which used to be a sheet of paper!). This geometrical transformation doesn't have a unique solution, so over the cartography history, every country or region in the world has been able to express the belief of being the center of the universe. In other words, every cartographic projection tries to minimize the distortions of the 3D to 2D transformation for a given point of the Earth surface[1].

In OTB the `otb::MapProjection` class is derived from the `itk::Transform` class, so the coordinate transformation points are overloaded with map projection equations. The `otb::MapProjection` class is templated over the type of cartographic projection, which is provided by the OSSIM library. In order to hide the complexity of the approach, some type definitions for the more common projections are given in the file `otbMapProjections.h` file.

You will seldom use a map projection by itself, but rather in an ortho-rectification framework. An example is given in the next section.

10.3 Ortho-rectification with OTB

The source code for this example can be found in the file
`Examples/Projections/OrthoRectificationExample.cxx`.

This example demonstrates the use of the `otb::OrthoRectificationFilter`. This filter is intended to orthorectify images which are in a distributor format with the appropriate meta-data describing the sensor model. In this example, we will choose to use an UTM projection for the output image.

The first step toward the use of these filters is to include the proper header files: the one for the ortho-rectification filter and the one defining the different projections available in OTB.

```
#include "otbOrthoRectificationFilter.h"
#include "otbMapProjections.h"
```

We will start by defining the types for the images, the image file reader and the image file writer. The writer will be a `otb::StreamingImageFileWriter` which will allow us to set the number of stream divisions we want to apply when writing the output image, which can be very large.

```
  typedef otb::Image<unsigned char, 2>    CharImageType;
  typedef otb::Image<unsigned int, 2>     ImageType;
  typedef otb::VectorImage<unsigned int, 2>     VectorImageType;
  typedef otb::ImageFileReader<VectorImageType>  ReaderType;
  typedef otb::StreamingImageFileWriter<VectorImageType>  WriterType;


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

[1]We proposed to optimize an OTB map projection for Toulouse, but we didn't get any help from OTB users.

```
reader->SetFileName(argv[1]);
writer->SetFileName(argv[2]);
```

We can now proceed to declare the type for the ortho-rectification filter. The class `otb::OrthoRectificationFilter` is templated over the input and the output image types as well as over the cartographic projection. We define therefore the type of the projection we want, which is an UTM projection for this case.

```
typedef otb::UtmInverseProjection utmMapProjectionType ;
typedef otb::OrthoRectificationFilter<ImageType, ImageType,
                            utmMapProjectionType> OrthoRectifFilterType ;


OrthoRectifFilterType::Pointer  orthoRectifFilter =
                                            OrthoRectifFilterType::New();
```

Now we need to instanciate the map projection, set the *zone* and *hemisphere* parameters and pass this projection to the orthorectification filter.

```
utmMapProjectionType::Pointer utmMapProjection =
                                            utmMapProjectionType::New();
utmMapProjection->SetZone(atoi(argv[3]));
utmMapProjection->SetHemisphere(*(argv[4]));
orthoRectifFilter->SetMapProjection(utmMapProjection);
```

Wiring the orthorectification filter into a PerBandImageFilter allows to orthrectify images with multiple bands seamlesly.

```
typedef otb::PerBandVectorImageFilter<VectorImageType,
        VectorImageType, OrthoRectifFilterType> PerBandFilterType;
PerBandFilterType::Pointer perBandFilter=PerBandFilterType::New();
perBandFilter->SetFilter(orthoRectifFilter);
perBandFilter->SetInput(reader->GetOutput());
```

Using the user-provided information, we define the output region for the image generated by the orthorectification filter.

```
ImageType::IndexType start;
start[0]=0;
start[1]=0;
orthoRectifFilter->SetOutputStartIndex(start);

ImageType::SizeType size;
size[0]=atoi(argv[7]);
size[1]=atoi(argv[8]);
orthoRectifFilter->SetSize(size);

ImageType::SpacingType spacing;
spacing[0]=atof(argv[9]);
spacing[1]=atof(argv[10]);
orthoRectifFilter->SetOutputSpacing(spacing);

ImageType::PointType origin;
origin[0]=strtod(argv[5], NULL);
origin[1]=strtod(argv[6], NULL);
orthoRectifFilter->SetOutputOrigin(origin);
```

We can now set plug the ortho-rectification filter to the writer and set the number of tiles we want to split the output image in for the writing step.

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

writer->SetTilingStreamDivisions();
```

Finally, we trigger the pipeline execution by calling the `Update()` method on the writer. Please note that the ortho-rectification filter is derived from the `otb::StreamingResampleImageFilter` in order to be able to compute the input image regions which are needed to build the output image. Since the resampler applies a geometric transformation (scale, rotation, etc.), this region computation is not trivial.

```
writer->Update();
```

CHAPTER

ELEVEN

Radiometry

Remote sensing is not just a matter of taking pictures, but also – mostly – a matter of measuring physical values. In order to properly deal with physical magnitudes, the numerical values provided by the sensors have to be calibrated. After that, several indices with physical meaning can be computed.

Calibration functionnalities (absolute and relative) and even atmospheric correction routines will be available in future versions of OTB. Please note that the 6S Radiative Transfer Code[1] is already included in the OTB source code and compiles out of the box. Calibration and atmospheric corrections in OTB will be based on it.

In the current version of OTB, several vegetation indices are already available. They are presented in this chapter.

11.1 Vegetation Index

11.1.1 Introduction

A vegetation index is a quantitative measure used to measure biomass or vegetative vigor, usually formed from combinations of several spectral bands, whose values are added, divided, or multiplied in order to yield a single value that indicates the amount or vigor of vegetation.

11.1.2 NDVI

NDVI was one of the most successful of many attempts to simply and quickly identify vegetated areas and their *condition*, and it remains the most well-known and used index to detect live green plant canopies in multispectral remote sensing data. Once the feasibility to detect vegetation had been demonstrated, users tended to also use the NDVI to quantify the photosynthetic capacity of

[1] http://6s.ltdri.org/

plant canopies. This, however, can be a rather more complex undertaking if not done properly.

The source code for this example can be found in the file
`Examples/Radiometry/NDVIRAndNIRVegetationIndexImageFilter.cxx`.

The following example illustrates the use of the `otb::RAndNIRVegetationIndexImageFilter` with the use of the Normalized Difference Vegatation Index (NDVI). NDVI computes the difference between the NIR channel, noted L_{NIR}, and the red channel, noted L_r radiances reflected from the surface and transmitted through the atmosphere:

$$\mathbf{NDVI} = \frac{L_{NIR} - L_r}{L_{NIR} + L_r} \tag{11.1}$$

With the `otb::RAndNIRVegetationIndexImageFilter` class the filter inputs are one channel images: one inmage represents the NIR channel, the the other the NIR channel.

Let's look at the minimal code required to use this algorithm. First, the following header defining the `otb::RAndNIRVegetationIndexImageFilter` class must be included.

```
#include "otbRAndNIRVegetationIndexImageFilter.h"
```

The image types are now defined using pixel types the dimension. Input and output images are defined as `otb::Image`.

```
  const unsigned int                                 Dimension = 2;
  typedef double                                     InputPixelType;
  typedef float                                      OutputPixelType;
  typedef otb::Image<InputPixelType,Dimension>       InputRImageType;
  typedef otb::Image<InputPixelType,Dimension>       InputNIRImageType;
  typedef otb::Image<OutputPixelType,Dimension>      OutputImageType;
```

The NDVI (Normalized Difference Vegetation Index) is instantiated using the images pixel type as template parameters. It is implemented as a functor class which will be passed as a parameter to an `otb::RAndNIRVegetationIndexImageFilter`.

```
  typedef otb::Functor::NDVI< InputPixelType,
                              InputPixelType,
                              OutputPixelType>   FunctorType;
```

The `otb::RAndNIRVegetationIndexImageFilter` type is instantiated using the images types and the NDVI functor as template parameters.

```
  typedef otb::RAndNIRVegetationIndexImageFilter<InputRImageType,
                                                 InputNIRImageType,
                                                 OutputImageType,
                                                 FunctorType>
                              RAndNIRVegetationIndexImageFilterType;
```

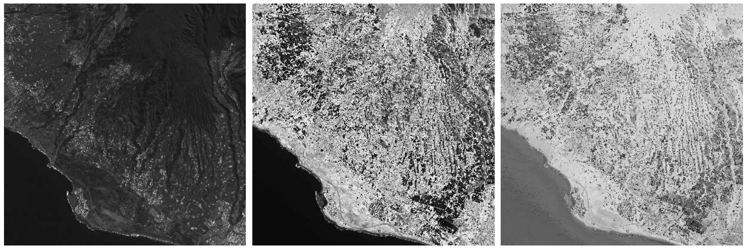


Figure 11.1: NDVI input images on the left (Red channel and NIR channel), on the right the result of the algorithm.

Now the input images are set and a name is given to the output image.

```
readerR->SetFileName( argv[1] );
readerNIR->SetFileName( argv[2] );
writer->SetFileName( argv[3]  );
```

We set the processing pipeline: filter inputs are linked to the reader output and the filter output is linked to the writer input.

```
filter->SetInputR( readerR->GetOutput() );
filter->SetInputNIR( readerNIR->GetOutput() );

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

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 images `NDVI_3.hdr` and `NDVI_4.hdr` (images kindly and free of charge given by SISA and CNES) provided in the directory `Examples/Data`.

11.1.3 ARVI

The source code for this example can be found in the file
`Examples/Radiometry/ARVIMultiChannelRAndBAndNIRVegetationIndexImageFilter.cxx`.

The following example illustrates the use of the otb::MultiChannelRAndBAndNIR VegetationIndexImageFilter with the use of the Atmospherically Resistant Vegetation Index (ARVI). ARVI is an improved version of the NDVI that is more resistent to the atmospheric effect. In addition to the red and NIR channels (used in the NDVI), the ARVI takes advantage of the presence of the blue channel to accomplish a self-correction process for the atmospheric effect on the red channel. For this, it uses the difference in the radiance between the blue and the red channels to correct the radiance in the red channel. Let's define ρ_{NIR}^{*}, ρ_{r}^{*}, ρ_{b}^{*} the normalized radiances (that is to say the radiance normalized to reflectance units) of red, blue and NIR channels respectively. ρ_{rb}^{*} is defined as

$$\rho_{rb}^{*} = \rho_{r}^{*} - \gamma * (\rho_{b}^{*} - \rho_{r}^{*}) \tag{11.2}$$

The ARVI expression is

$$\mathbf{ARVI} = \frac{\rho_{NIR}^{*} - \rho_{rb}^{*}}{\rho_{NIR}^{*} + \rho_{rb}^{*}} \tag{11.3}$$

This formula can be simplified with :

$$\mathbf{ARVI} = \frac{L_{NIR} - L_{rb}}{L_{NIR} + L_{rb}} \tag{11.4}$$

For more details, refer to Faufman and Tanré work [51].

With the `otb::MultiChannelRAndBAndNIRVegetationIndexImageFilter` class the input has to be a multi channel image and the user has to specify index channel of the red, blue and NIR channel.

Let's look at the minimal code required to use this algorithm. First, the following header defining the `otb::MultiChannelRAndBAndNIRVegetationIndexImageFilter` class must be included.

```
#include "otbMultiChannelRAndBAndNIRVegetationIndexImageFilter.h"
```

The image types are now defined using pixel types and dimension. The input image is defined as an `otb::VectorImage`, the output is a `otb::Image`.

```
  const unsigned int                                 Dimension = 2;
  typedef double                                     InputPixelType;
  typedef float                                      OutputPixelType;
  typedef otb::VectorImage<InputPixelType ,Dimension> InputImageType;
  typedef otb::Image<OutputPixelType,Dimension>       OutputImageType;
```

The ARVI (Atmospherically Resistant Vegetation Index) is instantiated using the image pixel types as template parameters.

```
typedef  otb::Functor::ARVI< InputPixelType,
                             InputPixelType,
                InputPixelType,
                                 OutputPixelType >        FunctorType;
```

The `otb::MultiChannelRAndBAndNIRVegetationIndexImageFilter` type is defined using the image types and the ARVI functor as template parameters. We then instantiate the filter itself.

```
typedef otb::MultiChannelRAndBAndNIRVegetationIndexImageFilter
                                    <InputImageType,
                                     OutputImageType,
                                     FunctorType      >
                   MultiChannelRAndBAndNIRVegetationIndexImageFilterType;

MultiChannelRAndBAndNIRVegetationIndexImageFilterType::Pointer
  filter = MultiChannelRAndBAndNIRVegetationIndexImageFilterType::New();
```

Now the input image is set and a name is given to the output image.

```
reader->SetFileName( argv[1] );
writer->SetFileName( argv[2] );
```

The three used index bands (red, blue and NIR) are declared.

```
filter->SetRedIndex(::atoi(argv[5]));
filter->SetBlueIndex(::atoi(argv[6]));
filter->SetNIRIndex(::atoi(argv[7]));
```

The γ parameter is set. The `otb::MultiChannelRAndBAndNIRVegetationIndexImageFilter` class sets the default value of γ to 0.5. This parameter is used to reduce the atmospheric effect on a global scale.

```
filter->GetFunctor().SetGamma(::atof(argv[8]));
```

The filter input is linked to the reader output and the filter output is linked to the writer input.

```
filter->SetInput( reader->GetOutput() );

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

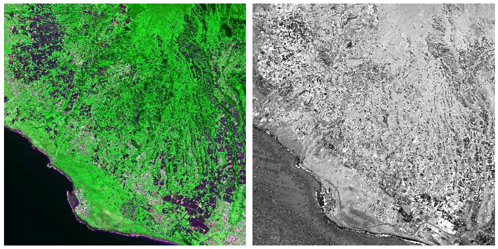


Figure 11.2: ARVI result on the right with the left image in input.

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 `IndexVegetation.hd` (image kindly and free of charge given by SISA and CNES) and γ=0.6 provided in the directory `Examples/Data`.

11.2 Atmospheric Corrections

The source code for this example can be found in the file
`Examples/Radiometry/AtmosphericCorrectionSequencement.cxx`.

The following example illustrates the application of atmospheric corrections to an optical multispectral image similar to Pleiades. These corrections are made in four steps :

- digital number to luminance correction;
- luminance to refletance image conversion;
- atmospheric correction for TOA (top of atmosphere) to TOC (top of canopy) reflectance estimation;
- correction of the adjency effects taking into account the neighborhood contribution.

The manipulation of each class used for the different steps and the link with the 6S radiometry library will be explained.

Let's look at the minimal code required to use this algorithm. First, the following header defining the `otb::AtmosphericCorrectionSequencement` class must be included. For the numerical to luminance image, luminance to refletance image, and reflectance to atmospheric correction image corrections and the neighborhood correction, four header files are required.

```
#include "otbImageToLuminanceImageFilter.h"
#include "otbLuminanceToReflectanceImageFilter.h"
#include "otbReflectanceToSurfaceReflectanceImageFilter.h"
#include "otbSurfaceAdjencyEffect6SCorrectionSchemeFilter.h"
```

This chain uses the 6S radiative transfer code to compute radiometric parameters. To manipulate 6S data, three classes are needed (the first one to store the metadata, the second one that calls 6S class and generates the information which will be stored in the last one).

```
#include "otbAtmosphericCorrectionParameters.h"
#include "otbAtmosphericCorrectionParametersTo6SAtmosphericRadiativeTerms.h"
#include "otbAtmosphericRadiativeTerms.h"
```

Image types are now defined using pixel types and dimension. The input image is defined as an `otb::VectorImage`, the output image is a `otb::VectorImage`. To simplify, input and output image types are the same one.

```
  const unsigned int                                     Dimension = 2;
  typedef double                                         PixelType;
  typedef otb::VectorImage<PixelType,Dimension>          ImageType;
```

The `GenerateOutputInformation()` reader method is called to know the number of component per pixel of the image. It is recommended to place `GenerateOutputInformation` calls in a `try/catch` block in case errors occur and exceptions are thrown.

```
  reader->SetFileName(argv[1]);
  try
    {
      reader->GenerateOutputInformation();
    }
  catch( itk::ExceptionObject & excep )
    {
      std::cerr << "Exception caught !" << std::endl;
      std::cerr << excep << std::endl;
    }
```

The `otb::ImageToLuminanceImageFilter` type is defined and instancied. This class uses a functor applied to each component of each pixel ($\mathbf{X^k}$) whose formula is:

$$\mathbf{L^k_{TOA}} = \frac{X^k}{\alpha_k} + \beta_k. \quad (11.5)$$

Where :

- $\mathbf{L^k_{TOA}}$ is the incident luminance (in $W.m^{-2}.sr^{-1}.\mu m^{-1}$);
- $\mathbf{X^k}$ is the measured digital number (ie. the input image pixel component);
- α_k is the absolute calibration gain for the channel k;
- β_k is the absolute calibration bias for the channel k.

```
typedef otb::ImageToLuminanceImageFilter<ImageType,ImageType>
  ImageToLuminanceImageFilterType;

ImageToLuminanceImageFilterType::Pointer filterImageToLuminance
  = ImageToLuminanceImageFilterType::New();
```

Here, α and β are read from an ASCII file given in input, stored in a vector and passed to the class.

```
filterImageToLuminance->SetAlpha(alpha);
filterImageToLuminance->SetBeta(beta);
```

The `otb::LuminanceToReflectanceImageFilter` type is defined and instancied. This class used a functor applied to each component of each pixel of the luminance filter output ($\mathbf{L^k_{TOA}}$):

$$\rho^k_{TOA} = \frac{\pi.\mathbf{L^k_{TOA}}}{E^k_S.cos(\theta_S).d/d_0}. \quad (11.6)$$

Where :

- $\mathbf{rho^k_{TOA}}$ is the reflectance measured by the sensor;
- θ_S is the zenithal solar angle in degrees;
- E^k_S is the solar illumination out of the atmosphere measured at a distance d_0 from the Earth;
- d/d_0 is the ratio between the Earth-Sun distance at the acquisition date and the mean Earth-Sun distance. The ratio can be directly given to the class or computed using a 6S routine. In the last case (that is the one of this example), the user has to precise the month and the day of the acquisition.

```
typedef otb::LuminanceToReflectanceImageFilter<ImageType,ImageType>
                              LuminanceToReflectanceImageFilterType;
LuminanceToReflectanceImageFilterType::Pointer filterLuminanceToReflectance
  = LuminanceToReflectanceImageFilterType::New();
```

The solar illumination is read from a ASCII file given in input, stored in a vector and given to the class. Day, month and zenital solar angle are inputs and can be directly given to the class.

```
filterLuminanceToReflectance->SetZenithalSolarAngle(
  static_cast<double>(atof(argv[6])));
filterLuminanceToReflectance->SetDay(atoi(argv[7]));
filterLuminanceToReflectance->SetMonth(atoi(argv[8]));
filterLuminanceToReflectance->SetSolarIllumination(solarIllumination);
```

At this step of the chain, radiometric informations are nedeed. Those informations will be computed from different parameters stored in a `otb::AtmosphericCorrectionParameters` class intance. This *container* will be given to an `otb::AtmosphericCorrectionParametersTo6SAtmosphericRadiativeTerms` class instance which will call a 6S routine that will compute the needed radiometric informations and store them in a `otb::AtmosphericRadiativeTerms` class instance. For this, `otb::AtmosphericCorrectionParametersTo6SAtmosphericRadiativeTerms` `otb::AtmosphericCorrectionParameters` and `otb::AtmosphericRadiativeTerms` types are defined and instancied.

```
typedef otb::AtmosphericCorrectionParametersTo6SAtmosphericRadiativeTerms
                    AtmosphericCorrectionParametersTo6SRadiativeTermsType;

typedef otb::AtmosphericCorrectionParameters
                                      AtmosphericCorrectionParametersType;

typedef otb::AtmosphericRadiativeTerms
                                            AtmosphericRadiativeTermsType;
```

The `otb::AtmosphericCorrectionParameters` class needs several parameters :

- The zenithal and azimutal solar angles that describe the solar incidence configuration (in degrees);
- The zenithal and azimuthal viewing angles that describe the viewing direction (in degrees);
- The month and the day of the acquisition;
- The atmospheric pressure;
- The water vapor amount, that is, the total water vapor content over vertical atmospheric column;

- The ozone amount that is the Stratospheric ozone layer content;
- The aerosol model that is the kind of particles (no aerosol, continental, maritime, urban, desertic);
- The aerosol optical thickness at 550 nm that is the is the Radiative impact of aerosol for the reference wavelength 550 nm;
- The filter function that is the values of the filter function for one spectral band, from λ_{inf} to λ_{sup} by step of 2.5 nm. One filter function by channel is required. This last parameter are read in text files, the other one are directly given to the class.

```
dataAtmosphericCorrectionParameters->SetSolarZenithalAngle(
  static_cast<double>(atof(argv[6])));

dataAtmosphericCorrectionParameters->SetSolarAzimutalAngle(
  static_cast<double>(atof(argv[9])));

dataAtmosphericCorrectionParameters->SetViewingZenithalAngle(
  static_cast<double>(atof(argv[10])));

dataAtmosphericCorrectionParameters->SetViewingAzimutalAngle(
  static_cast<double>(atof(argv[11])));

dataAtmosphericCorrectionParameters->SetMonth(atoi(argv[8]));

dataAtmosphericCorrectionParameters->SetDay(atoi(argv[7]));

dataAtmosphericCorrectionParameters->SetAtmosphericPressure(
  static_cast<double>(atof(argv[12])));

dataAtmosphericCorrectionParameters->SetWaterVaporAmount(
  static_cast<double>(atof(argv[13])));

dataAtmosphericCorrectionParameters->SetOzoneAmount(
  static_cast<double>(atof(argv[14])));

AerosolModelType aerosolModel =
  static_cast<AerosolModelType>(::atoi(argv[15]));

dataAtmosphericCorrectionParameters->SetAerosolModel(aerosolModel);

dataAtmosphericCorrectionParameters->SetAerosolOptical(
  static_cast<double>(atof(argv[16])));
```

Once those parameters are loaded and stored in the AtmosphericCorrectionParameters instance class, it is given in input of an instance of AtmosphericCorrectionParametersTo6SAtmosphericRadiativeTerms that will compute the needed radiometric informations.

```
AtmosphericCorrectionParametersTo6SRadiativeTermsType::Pointer
  filterAtmosphericCorrectionParametersTo6SRadiativeTerms =
  AtmosphericCorrectionParametersTo6SRadiativeTermsType::New();

filterAtmosphericCorrectionParametersTo6SRadiativeTerms->SetInput(
  dataAtmosphericCorrectionParameters );
```

The output of this class will be an instance of the AtmosphericRadiativeTerms class. This class contains (for each channel of the image)

- The Intrinsic atmospheric reflectance that takes into account for the molecular scattering and the aerosol scattering attenuated by water vapor absorption;
- The spherical albedo of the atmosphere;
- The total gaseous transmission (for all species);
- The total transmittance of the atmosphere from sun to ground (downward transmittance) and from ground to space sensor (upward transmittance).

Atmospheric corrections can now start. First, an instance of `otb::ReflectanceToSurfaceReflectanceImageFilter` is created.

```
typedef otb::ReflectanceToSurfaceReflectanceImageFilter<ImageType,
  ImageType> ReflectanceToSurfaceReflectanceImageFilterType;

ReflectanceToSurfaceReflectanceImageFilterType::Pointer
  filterReflectanceToSurfaceReflectanceImageFilter
 = ReflectanceToSurfaceReflectanceImageFilterType::New();
```

The aim of the atmospheric correction is to invert the surface reflectance (for each pixel of the input image) from the TOA reflectance and from simulations of the atmospheric radiative functions corresponding to the geometrical conditions of the observation and to the atmospheric components. The process required to be applied on each pixel of the image, band by band with the following formula:

$$\rho_S^{unif} = \frac{\mathbf{A}}{1 + Sx\mathbf{A}} \tag{11.7}$$

Where,

$$\mathbf{A} = \frac{\rho_{TOA} - \rho_{atm}}{T(\mu_S).T(\mu_V).t_g^{allgas}} \tag{11.8}$$

With :

- ρ_{TOA} is the reflectance at the top of the atmosphere;

- ρ_S^{unif} is the ground reflectance under asumption of a lambertian surface and an uniform environment;
- ρ_{atm} is the intrinsic atmospheric reflectance;
- t_g^{allgas} is the soherical albedo of the atmosphere;
- $T(\mu_S)$ is the downward transmittance;
- $T(\mu_V)$ is the upward transmittance.

All those parameters are contained in the AtmosphericCorrectionParametersTo6SRadiativeTerms output.

```
filterReflectanceToSurfaceReflectanceImageFilter->
  SetAtmosphericRadiativeTerms(
    filterAtmosphericCorrectionParametersTo6SRadiativeTerms->GetOutput() );
```

Next (and last step) is the neighborhood correction. For this, the SurfaceAdjencyEffect6SCorrectionSchemeFilter class is used. The previous surface reflectance inversion is performed under the assumption of a homogeneous ground environment. The following step allows to correct the adjacency effect on the radiometry of pixels. The method is based on the decomposition of the observed signal as the summation of the own contribution of the target pixel and of the contribution of neighbored pixels moderated by their distance to the target pixel. A simplified relation may be :

$$\rho S = \frac{\rho_S^{unif}.T(\mu_V) - <\rho S>.t_d(\mu_V)}{exp(-\delta/\mu_V)} \tag{11.9}$$

With :

- ρ_S^{unif} is the ground reflectance under asumption of an homogeneous environment;
- $T(\mu_V)$ is the upward transmittance;
- $t_d(\mu_S)$ is the upward diffus transmittance;
- $exp(-\delta/\mu_V)$ is the upward direct transmittance;
- ρ_S is the environment contribution to the pixel target reflectance in the total observed signal.

  $$\rho S = \sum j \sum i f(r(i,j)) \times \rho_S^{unif}(i,j) \tag{11.10}$$

  where,

  - r(i,j) is the distance between the pixel(i,j) and the central pixel of the window in *km*;

– f(r) is the global environment function.

$$f(r) = \frac{t_d^R(\mu_V).f_R(r) + t_d^A(\mu_V).f_A(r)}{t_d(\mu_V)} \tag{11.11}$$

The neighborhood consideration window size is given by the window radius. An instance of `otb::SurfaceAdjencyEffect6SCorrectionSchemeFilter` is created.

```
typedef otb::SurfaceAdjencyEffect6SCorrectionSchemeFilter<ImageType,
  ImageType>  SurfaceAdjencyEffect6SCorrectionSchemeFilterType;
SurfaceAdjencyEffect6SCorrectionSchemeFilterType::Pointer
  filterSurfaceAdjencyEffect6SCorrectionSchemeFilter
  = SurfaceAdjencyEffect6SCorrectionSchemeFilterType::New();
```

The needs four input informations:

- Radiometric informations (the output of the AtmosphericCorrectionParameter-sTo6SRadiativeTerms filter);
- The zenithal viewing angle;
- The neighborhood window radius;
- The pixel spacing in kilometers.

At this step, each filter of the chain is instancied and every one has its input paramters set. A name can be given to the output image and each filter can linked to other to create the final processing chain.

```
writer->SetFileName(argv[2]);

filterImageToLuminance->SetInput(reader->GetOutput());
filterLuminanceToReflectance->SetInput(filterImageToLuminance->GetOutput());
filterReflectanceToSurfaceReflectanceImageFilter->SetInput(
  filterLuminanceToReflectance->GetOutput());
filterSurfaceAdjencyEffect6SCorrectionSchemeFilter->SetInput(
  filterReflectanceToSurfaceReflectanceImageFilter->GetOutput());

writer->SetInput(
  filterSurfaceAdjencyEffect6SCorrectionSchemeFilter->GetOutput());
```

The invocation of the `Update()` method on the writer triggers the execution of the pipeline. It is recommended to place this call 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;
  }
```

CHAPTER

TWELVE

Image Fusion

Satellite sensors present an important diversity in terms of characteristics. Some provide a high spatial resolution while other focus on providing several spectral bands. The fusion process brings the information from different sensors with different characteristics together to get the best of both worlds.

Most of the fusion methods in the remote sensing community deal with the *pansharpening technique*. This fusion combines the image from the PANchromatic sensor of one satellite (high spatial resolution data) with the multispectral (XS) data (lower resolution in several spectral bands) to generate images with a high resolution and several spectral bands. Several advantages make this situation easier:

- PAN and XS images are taken simultaneously from the same satellite (or with a very short delay);
- the imaged area is common to both scenes;
- many satellites provide these data (SPOT 1-5, Quickbird, Pleiades)

This case is well-studied in the literature and many methods exist. Only very few are available in OTB now but this should evolve soon.

12.1 Simple Pan Sharpening

A simple way to view the pan-sharpening of data is to consider that, at the same resolution, the panchromatic channel is the sum of the XS channel. After putting the two images in the same geometry, after orthorectification (see chapter 10) with an oversampling of the XS image, we can proceed to the data fusion.

The idea is to apply a low pass filter to the panchromatic band to give it a spectral content (in the Fourier domain) equivalent to the XS data. Then we normalize the XS data with this low-pass panchromatic and multiply the result with the original panchromatic band.

The process is described on figure 12.1.

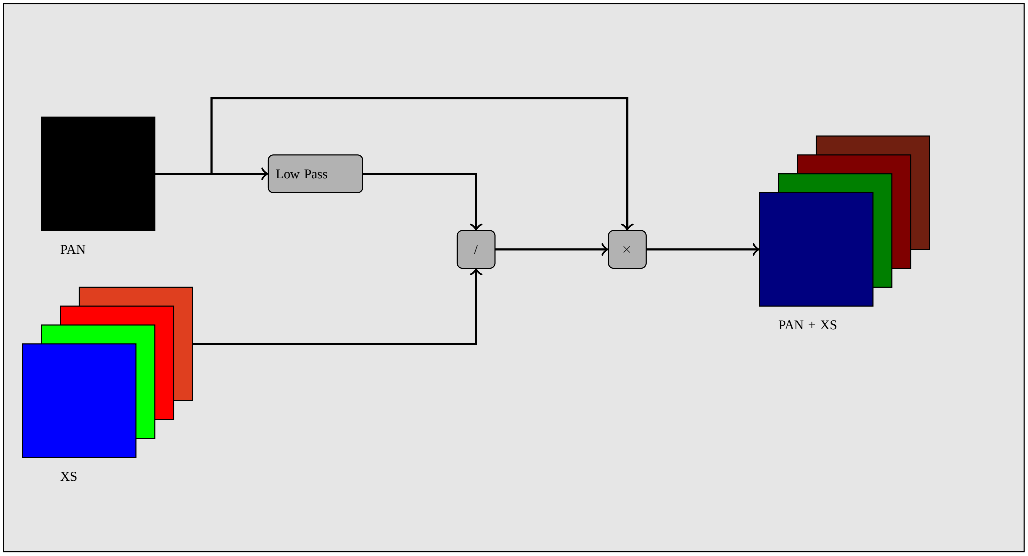


Figure 12.1: Simple pan-sharpening procedure.

The source code for this example can be found in the file
`Examples/Fusion/PanSharpeningExample.cxx`.

Here we are illustrating the use of the `otb::SimpleRcsPanSharpeningFusionImageFilter` for PAN-sharpening. This example takes a PAN and the corresponding XS images as input. These images are supposed to be registered.

The fusion operation is defined as

$$\frac{XS}{\mathrm{Filtered}(PAN)} PAN \tag{12.1}$$

Figure 12.2 shows the result of applying this PAN sharpening filter to a Quickbird image.

We start by including the required header and declaring the main function:

```
#include "otbImage.h"
#include "otbVectorImage.h"
#include "otbImageFileReader.h"
#include "otbStreamingImageFileWriter.h"
#include "otbSimpleRcsPanSharpeningFusionImageFilter.h"

int main( int argc, char* argv[] )
{
```

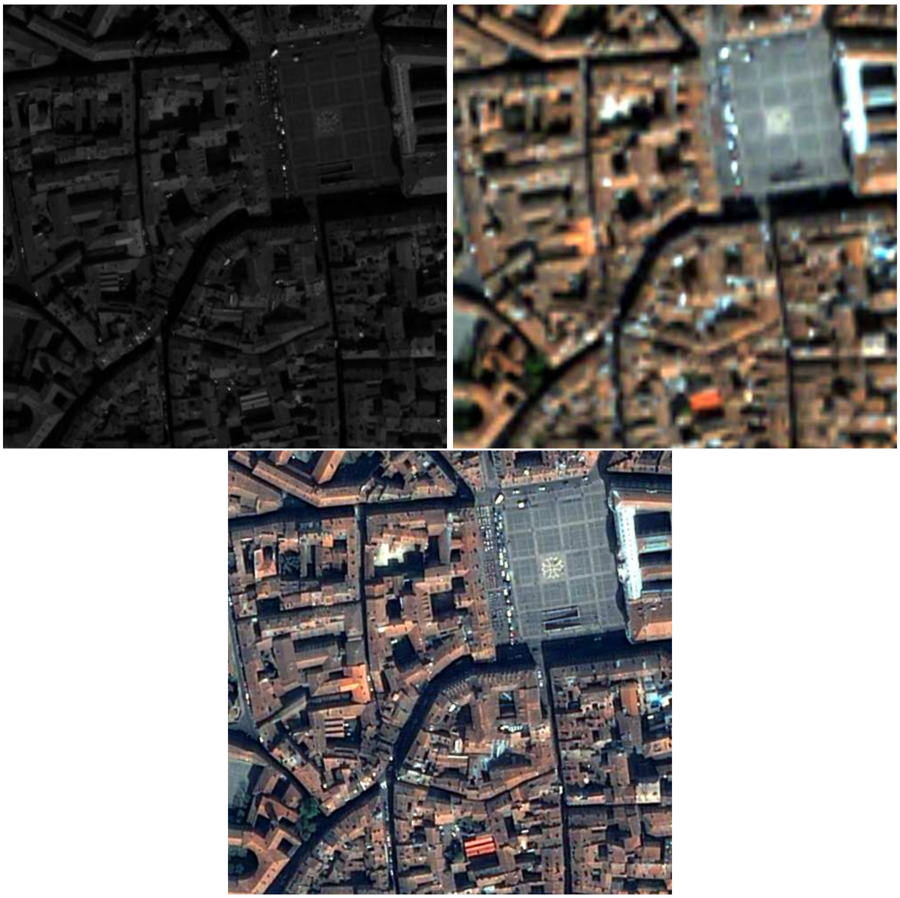


Figure 12.2: Result of applying the `otb::SimpleRcsPanSharpeningFusionImageFilter` to orthorectified Quickbird image. From left to right : original PAN image, original XS image and the result of the PAN sharpening

We declare the different image type used here as well as the image reader. Note that, the reader for the PAN image is templated by an `otb::Image` while the XS reader uses an `otb::VectorImage`.

```
typedef otb::Image<double, 2>     ImageType;
typedef otb::VectorImage<double, 2>     VectorImageType;
typedef otb::ImageFileReader<ImageType>  ReaderType;
typedef otb::ImageFileReader<VectorImageType>  ReaderVectorType;
typedef otb::VectorImage<unsigned int, 2>     VectorIntImageType;


ReaderVectorType::Pointer      readerXS=ReaderVectorType::New();
ReaderType::Pointer      readerPAN=ReaderType::New();
```

We pass the filenames to the readers

```
readerPAN->SetFileName(argv[1]);
readerXS->SetFileName(argv[2]);
```

We declare the fusion filter an set its inputs using the readers:

```
typedef otb::SimpleRcsPanSharpeningFusionImageFilter
    <ImageType,VectorImageType,VectorIntImageType> FusionFilterType;
FusionFilterType::Pointer fusion = FusionFilterType::New();
fusion->SetPanInput(readerPAN->GetOutput());
fusion->SetXsInput(readerXS->GetOutput());
```

And finally, we declare the writer and call its `Update()` method to trigger the full pipeline execution.

```
typedef otb::StreamingImageFileWriter<VectorIntImageType>  WriterType;
WriterType::Pointer      writer=WriterType::New();
writer->SetFileName(argv[3]);
writer->SetInput(fusion->GetOutput());
writer->Update();
```

12.2 Bayesian Data Fusion

The source code for this example can be found in the file
`Examples/Fusion/BayesianFusionImageFilter.cxx`.

The following example illustrates the use of the `otb::BayesianFusionFilter`. The Bayesian data fusion relies on the idea that variables of interest, denoted as vector $\mathbf{Z}$, cannot be directly observed. They are linked to the observable variables $\mathbf{Y}$ through the following error-like model.

$$\mathbf{Y} = g(\mathbf{Z}) + \mathbf{E} \tag{12.2}$$

where g(**Z**) is a set of functionals and **E** is a vector of random errors that are stochastically independent from **Z**. This algorithm uses elementary probability calculus, and several assumptions to compute the data fusion. For more explication see Fasbender, Radoux and Bogaert's publication [27]. Three images are used :

- a panchromatic image,
- a multispectral image resampled at the panchromatic image spatial resolution,
- a multispectral image resampled at the panchromatic image spatial resolution, using, e.g. a cubic interpolator.
- a float : λ, the meaning of the weight to be given to the panchromatic image compared to the multispectral one.

Let's look at the minimal code required to use this algorithm. First, the following header defining the otb::BayesianFusionFilter class must be included.

```
#include "otbBayesianFusionFilter.h"
```

The image types are now defined using pixel types and particular dimension. The panchromatic image is defined as an `otb::Image` and the multispectral one as `otb::VectorImage`.

```
typedef double                                              InternalPixelType;
const   unsigned int                                        Dimension = 2;
typedef otb::Image< InternalPixelType, Dimension >          PanchroImageType;
typedef otb::VectorImage< InternalPixelType, Dimension >    MultiSpecImageType;
```

The Bayesian data fusion filter type is instantiated using the images types as a template parameters.

```
typedef otb::BayesianFusionFilter< MultiSpecImageType,
                                       MultiSpecImageType,
                                   PanchroImageType,
                                   OutputImageType      >    BayesianFusionFilterType;
```

Next the filter is created by invoking the `New()` method and assigning the result to a `itk::SmartPointer`.

```
BayesianFusionFilterType::Pointer bayesianFilter = BayesianFusionFilterType::New();
```

Now the multi spectral image, the interpolated multi spectral image and the panchromatic image are given as inputs to the filter.

Figure 12.3: Input images used for this example (©European Space Imaging).

```
bayesianFilter->SetMultiSpect(       multiSpectReader->GetOutput() );
bayesianFilter->SetMultiSpectInterp( multiSpectInterpReader->GetOutput() );
bayesianFilter->SetPanchro(          panchroReader->GetOutput() );

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

The BayesianFusionFilter requires defining one parameter : λ. The λ parameter can be used to tune the fusion toward either a high color consistency or sharp details. Typical λ value range in $[0.5, 1[$, where higher values yield sharper details. by default λ is set at 0.9999.

```
bayesianFilter->SetLambda( atof(argv[9]) );
```

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 images `multiSpect.tif` , `multiSpectInterp.tif` and `panchro.tif` provided in the directory `Examples/Data`. The results obtained for 2 different values for λ are shown in figure 12.3.

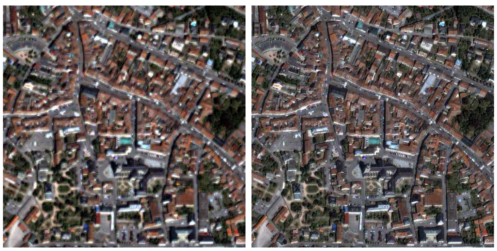


Figure 12.4: Fusion results for the Bayesian Data Fusion filter for $\lambda = 0.5$ on the left and $\lambda = 0.9999$ on the right.

CHAPTER

THIRTEEN

Feature Extraction

13.1 Introduction

Under the term *Feature Extraction* we include several techniques aiming to detect or extract informations of low level of abstraction from images. These *features* can be objects : points, lines, etc. They can also be measures : moments, textures, etc.

13.2 Interest Points

13.2.1 Harris detector

The source code for this example can be found in the file
`Examples/FeatureExtraction/HarrisExample.cxx`.

This example illustrates the use of the `otb::HarrisImageFilter`.

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

```
#include "otbHarrisImageFilter.h"
```

The `otb::HarrisImageFilter` is templated over the input and output image types, so we start by defining:

```
    typedef otb::HarrisImageFilter<InputImageType,
                                   InputImageType>   HarrisFilterType;
```

The `otb::HarrisImageFilter` needs some parameters to operate. The derivative computation is performed by a convolution with the derivative of a Gaussian kernel of variance σ_D

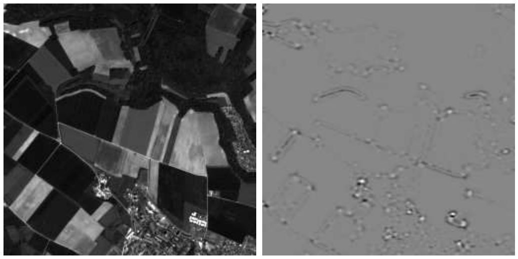


Figure 13.1: Result of applying the `otb::HarrisImageFilter` to a Spot 5 image.

(derivation scale) and the smoothing of the image is performed by convolving with a Gaussian kernel of variance σ_I (integration scale). This allows the computation of the following matrix:

$$\mu(\mathbf{x},\sigma_I,\sigma_D) = \sigma_D^2 g(\sigma_I) \star \begin{bmatrix} L_x^2(\mathbf{x},\sigma_D) & L_xL_y(\mathbf{x},\sigma_D) \\ L_xL_y(\mathbf{x},\sigma_D) & L_y^2(\mathbf{x},\sigma_D) \end{bmatrix} \tag{13.1}$$

The output of the detector is

$$det(\mu) - \alpha trace^2(\mu).$$

```
harris->SetSigmaD( SigmaD );
harris->SetSigmaI( SigmaI );
harris->SetAlpha( Alpha );
```

Figure 13.1 shows the result of applying the interest point detector to a small patch extracted from a Spot 5 image.

The output of the `otb::HarrisImageFilter` is an image where,for each pixel, we obtain the intensity of the detection. Often, the user may want to get access to the set of points for which the output of the detector is higher than a given threshold. This can be obtained by using the `otb::HarrisImageToPointSetFilter`. This filter is only templated over the input image type, the output being a `itk::PointSet` with pixel type equal to the image pixel type.

```
typedef otb::HarrisImageToPointSetFilter<InputImageType>   FunctionType;
```

We declare now the filter and a pointer to the output point set.

```
typedef FunctionType::OutputPointSetType   OutputPointSetType;


FunctionType::Pointer        harrisPoints    = FunctionType::New();
OutputPointSetType::Pointer  pointSet = OutputPointSetType::New();
```

The `otb::HarrisImageToPointSetFilter` takes the same parameters as the `otb::HarrisImageFilter` and an additional parameter : the threshold for the point selection.

```
harrisPoints->SetInput( 0,reader->GetOutput() );
harrisPoints->SetSigmaD( SigmaD );
harrisPoints->SetSigmaI( SigmaI );
harrisPoints->SetAlpha( Alpha );
harrisPoints->SetLowerThreshold( 10 );
pointSet = harrisPoints->GetOutput();
```

We can now iterate through the obtained pointset and access the coordinates of the points. We start by accessing the container of the points which is encapsulated into the point set (see section 5.2 for more information on using `itk::PointSets`) and declaring an iterator to it.

```
typedef OutputPointSetType::PointsContainer ContainerType;
ContainerType* pointsContainer = pointSet->GetPoints();
typedef ContainerType::Iterator IteratorType;
IteratorType itList = pointsContainer->Begin();
```

And we get the points coordinates

```
while( itList != pointsContainer->End() )
  {
  typedef OutputPointSetType::PointType           OutputPointType;
  OutputPointType pCoordinate = (itList.Value());
  std::cout << pCoordinate << std::endl;
  ++itList;
  }
```

13.2.2 SIFT detector

The source code for this example can be found in the file
`Examples/FeatureExtraction/SIFTExample.cxx`.

This example illustrates the use of the `otb::ImageToSIFTKeyPointSetFilter`. The Scale-Invariant Feature Transform (or SIFT) is an algorithm in computer vision to detect and describe local features in images. The algorithm was published by David Lowe [59]. The detection and description of local image features can help in object recognition and image registration. The SIFT features are local and based on the appearance of the object at particular interest points, and are invariant to image scale and rotation. They are also robust to changes in illumination, noise, occlusion and minor changes in viewpoint.

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

```
#include "otbImageToSIFTKeyPointSetFilter.h"
```

The otb::ImageToSIFTKeyPointSetFilter is templated over its input image type and the output point set type. Therefore, we start by defining the needed types.

```
typedef otb::Image<RealType,Dimension> ImageType;
typedef itk::VariableLengthVector<RealType> RealVectorType;
typedef otb::ImageFileReader<ImageType> ReaderType;
typedef itk::PointSet<RealVectorType,Dimension> PointSetType;
typedef otb::ImageToSIFTKeyPointSetFilter<ImageType,PointSetType> ImageToSIFTKeyPointSetFilterType;
```

Since the SIFT detector produces a point set, we will need iterators for the coordinates of the points and the data associated with them.

```
typedef PointSetType::PointsContainer PointsContainerType;
typedef PointsContainerType::Iterator PointsIteratorType;
typedef PointSetType::PointDataContainer PointDataContainerType;
typedef PointDataContainerType::Iterator PointDataIteratorType;
```

We can now instantiate the reader and the SIFT filter and plug the pipeline.

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

reader->SetFileName(infname);


filter->SetInput(0,reader->GetOutput());
```

The SIFT filter needs the following parameters:

- the number of octaves, that is, the number of levels of undersampling,
- the number of scales (blurring) per octave,
- the low contrast threshold to be applied to each point for the detection on the difference of Gaussians image,
- the threshold on the responses to consider a point as an edge.

```
filter->SetOctavesNumber(octaves);
filter->SetScalesNumber(scales);

filter->SetDoGThreshold(threshold);
filter->SetEdgeThreshold(ratio);
```

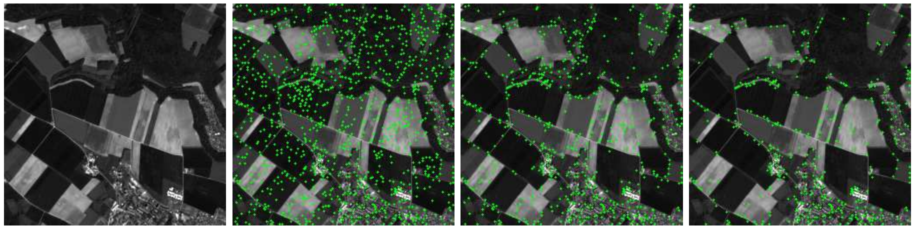


Figure 13.2: Result of applying the `otb::ImageToSIFTKeyPointSetFilter` to a Spot 5 image. Left to right: original image and SIFT with thresholds 0, 1 and 2 respectively.

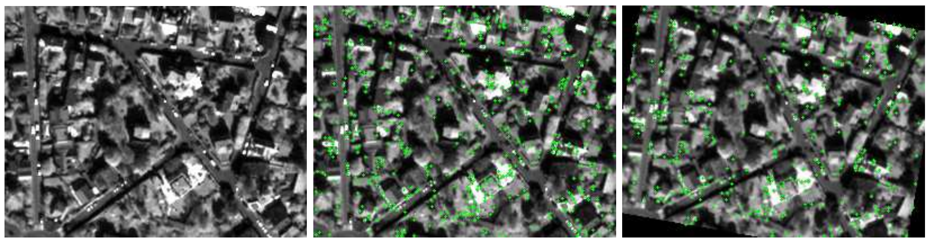


Figure 13.3: Result of applying the `otb::ImageToSIFTKeyPointSetFilter` to a high resolution image image. Left to right: original image and SIFT on the original and a rotated image respectively.

Figure 13.2 shows the result of applying the SIFT point detector to a small patch extracted from a Spot 5 image using different threshold values. Figure 13.3 shows the result of applying the SIFT point detector to a small patch extracted from a Spot 5 image using different threshold values.

13.3 Alignments

The source code for this example can be found in the file
`Examples/FeatureExtraction/AlignmentsExample.cxx`.

This example illustrates the use of the `otb::ImageToPathListAlignFilter`. This filter allows to extract meaninful alignments. Alignments (that is edges and lines) are detected using the *Gestalt* approach proposed by Desolneux et al. [25]. In this context, an event is considered meaningful if the expectation of its occurrence would be very small in a random image. One can thus consider that in a random image the direction of the gradient of a given point is uniformly distributed, and that neighbouring pixels have a very low probability of having the same gradient direction. This algorithm gives a set of straight line segments defined by the two extremity coordinates under the form of a `std::list` of `itk::PolyLineParametricPath`.

The first step required to use this filter is to include its header.

```
#include "otbImageToPathListAlignFilter.h"
```

In order to visualize the detected alignments, we will use the facility class `otb::DrawPathFilter` which draws a `itk::PolyLineParametricPath` on top of a given image.

```
#include "itkPolyLineParametricPath.h"
#include "otbDrawPathFilter.h"
```

The `otb::ImageToPathListAlignFilter` is templated over the input image type and the output path type, so we start by defining:

```
  typedef itk::PolyLineParametricPath< Dimension >  PathType;
  typedef otb::ImageToPathListAlignFilter<InputImageType,PathType>
                                                    ListAlignFilterType;
```

Next, we build the pipeline.

```
  ListAlignFilterType::Pointer alignFilter = ListAlignFilterType::New();

  alignFilter->SetInput( reader->GetOutput() );
```

We can choose the number of accepted false alarms in the detection with the method `SetEps()` for which the parameter is of the form $-log10(\text{max. number of false alarms})$.

```
  alignFilter->SetEps( atoi(argv[3]) );
```

As stated, above, the `otb::DrawPathFilter`, is useful for drawint the detected alignments. This class is templated over the input image and path types and also on the output image type.

```
  typedef otb::DrawPathFilter< InputImageType, PathType,
                               OutputImageType >  DrawPathFilterType;
```

We will now go through the list of detected paths and feed them to the `otb::DrawPathFilter` inside a loop. We will use a list iterator inside a `while` statement.

```
  typedef ListAlignFilterType::OutputPathListType ListType;


  ListType* pathList = alignFilter->GetOutput();

  ListType::Iterator listIt = pathList->Begin();
```

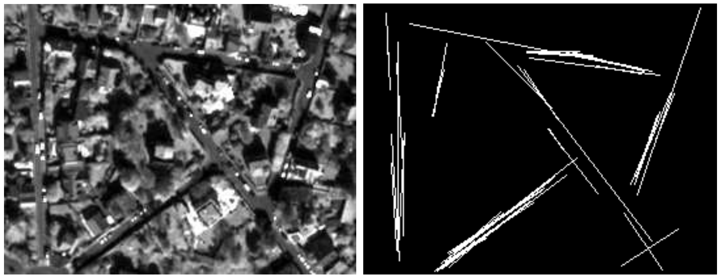


Figure 13.4: Result of applying the `otb::ImageToPathListAlignFilter` to a VHR image of a suburb.

We define a dummy image will be iteratively fed to the `otb::DrawPathFilter` after the drawing of each alignment.

```
InputImageType::Pointer backgroundImage = reader->GetOutput();
```

We iterate through the list and write the result to a file.

```
while( listIt != pathList->End())
  {

  DrawPathFilterType::Pointer drawPathFilter = DrawPathFilterType::New();
  drawPathFilter->SetImageInput( backgroundImage );
  drawPathFilter->SetPathInput( listIt.Get() );

  drawPathFilter->SetValue( itk::NumericTraits<OutputPixelType>::max() );
  drawPathFilter->Update();

  backgroundImage = drawPathFilter->GetOutput();

  ++listIt;

  }

writer->SetInput( backgroundImage );
```

Figure 13.4 shows the result of applying the alignment detection to a small patch extracted from a VHR image.

13.4 Lines

13.4.1 Line Detection

The source code for this example can be found in the file
`Examples/FeatureExtraction/RatioLineDetectorExample.cxx`.

This example illustrates the use of the `otb::RatioLineDetectorImageFilter`. This filter is used for line detection in SAR images. Its principle is described in [87]: a line is detected if two parallel edges are present in the images. These edges are detected with the ratio of means detector.

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

```
#include "otbLineRatioDetectorImageFilter.h"
```

Then we must decide what pixel type to use for the image. We choose to make all computations with floating point precision and rescale the results between 0 and 255 in order to export PNG images.

```
  typedef  float  InternalPixelType;
  typedef  unsigned char  OutputPixelType;
```

The images are defined using the pixel type and the dimension.

```
  typedef otb::Image< InternalPixelType,  2 >   InternalImageType;
  typedef otb::Image< OutputPixelType,  2 >   OutputImageType;
```

The filter can be instantiated using the image types defined above.

```
  typedef otb::LineRatioDetectorImageFilter< InternalImageType, InternalImageType >  FilterType;
```

An `otb::ImageFileReader` class is also instantiated in order to read image data from a file.

```
  typedef otb::ImageFileReader< InternalImageType >  ReaderType;
```

An `otb::ImageFileWriter` is instantiated in order to write the output image to a file.

```
  typedef otb::ImageFileWriter< OutputImageType >  WriterType;
```

The intensity rescaling of the results will be carried out by the `itk::RescaleIntensityImageFilter` which is templated by the input and output image types.

```
typedef itk::RescaleIntensityImageFilter< InternalImageType,
                                          OutputImageType > RescalerType;
```

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 same is done for the rescaler and the writer.

```
RescalerType::Pointer rescaler = RescalerType::New();
WriterType::Pointer writer = WriterType::New();
```

The `itk::RescaleIntensityImageFilter` needs to know which is the minimu and maximum values of the output generated image. Those can be chosen in a generic way by using the `NumericTraits` functions, since they are templated over the pixel type.

```
rescaler->SetOutputMinimum( itk::NumericTraits< OutputPixelType >::min());
rescaler->SetOutputMaximum( itk::NumericTraits< OutputPixelType >::max());
```

The image obtained with the reader is passed as input to the `otb::LineRatioDetectorImageFilter`. The pipeline is built as follows.

```
filter->SetInput( reader->GetOutput() );
rescaler->SetInput( filter->GetOutput() );
writer->SetInput( rescaler->GetOutput() );
```

The methods `SetLengthLine()` and `SetWidthLine()` allow to set the minimum length and the typical witdh of the lines which are to be detected.

```
filter->SetLengthLine( atoi(argv[4]) );
filter->SetWidthLine( atoi(argv[5]) );
```

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();
```

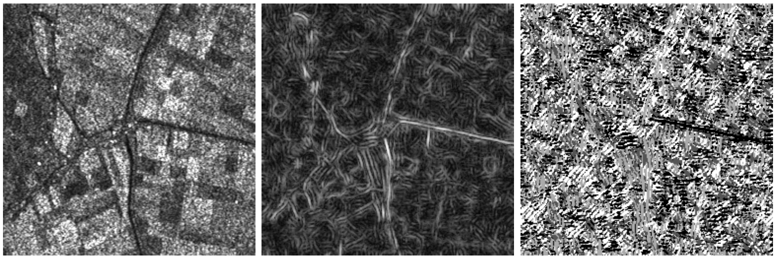


Figure 13.5: Result of applying the `otb::LineRatioDetectorImageFilter` to a SAR image. From left to right : original image, line intensity and edge orientation.

We can also obtain the direction of the lines by invoking the `GetOutputDirection()` method.

```
rescaler->SetInput( filter->GetOutputDirection() );
writer->SetInput( rescaler->GetOutput() );
writer->Update();
```

shows the result of applying the LineRatio edge detector filter to a SAR image.

The following classes provide similar functionality:

- `otb::LineCorrelationDetectorImageFilter`

The source code for this example can be found in the file
`Examples/FeatureExtraction/CorrelationLineDetectorExample.cxx`.

This example illustrates the use of the `otb::CorrelationLineDetectorImageFilter`. This filter is used for line detection in SAR images. Its principle is described in [87]: a line is detected if two parallel edges are present in the images. These edges are detected with the correlation of means detector.

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

```
#include "otbLineCorrelationDetectorImageFilter.h"
```

Then we must decide what pixel type to use for the image. We choose to make all computations with floating point precision and rescale the results between 0 and 255 in order to export PNG images.

```
typedef  float  InternalPixelType;
typedef  unsigned char  OutputPixelType;
```

The images are defined using the pixel type and the dimension.

```
typedef otb::Image< InternalPixelType,  2 >   InternalImageType;
typedef otb::Image< OutputPixelType,  2 >   OutputImageType;
```

The filter can be instantiated using the image types defined above.

```
typedef otb::LineCorrelationDetectorImageFilter< InternalImageType,
                                        InternalImageType >  FilterType;
```

An `otb::ImageFileReader` class is also instantiated in order to read image data from a file.

```
typedef otb::ImageFileReader< InternalImageType >  ReaderType;
```

An `otb::ImageFileWriter` is instantiated in order to write the output image to a file.

```
typedef otb::ImageFileWriter< OutputImageType >  WriterType;
```

The intensity rescaling of the results will be carried out by the `itk::RescaleIntensityImageFilter` which is templated by the input and output image types.

```
typedef itk::RescaleIntensityImageFilter< InternalImageType,
                                          OutputImageType > RescalerType;
```

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 same is done for the rescaler and the writer.

```
RescalerType::Pointer rescaler = RescalerType::New();
WriterType::Pointer writer = WriterType::New();
```

The `itk::RescaleIntensityImageFilter` needs to know which is the minimu and maximum values of the output generated image. Those can be chosen in a generic way by using the `NumericTraits` functions, since they are templated over the pixel type.

```
rescaler->SetOutputMinimum( itk::NumericTraits< OutputPixelType >::min());
rescaler->SetOutputMaximum( itk::NumericTraits< OutputPixelType >::max());
```

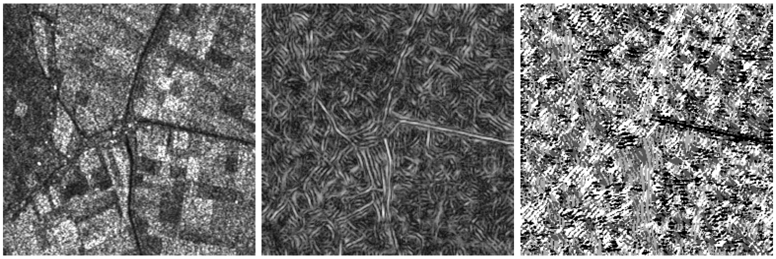


Figure 13.6: Result of applying the `otb::LineCorrelationDetectorImageFilter` to a SAR image. From left to right : original image, line intensity and edge orientation.

The image obtained with the reader is passed as input to the `otb::LineCorrelationDetectorImageFilter`. The pipeline is built as follows.

```
filter->SetInput( reader->GetOutput() );
rescaler->SetInput( filter->GetOutput() );
writer->SetInput( rescaler->GetOutput() );
```

The methods `SetLengthLine()` and `SetWidthLine()` allow to set the minimum length and the typical witdh of the lines which are to be detected.

```
filter->SetLengthLine( atoi(argv[4]) );
filter->SetWidthLine( atoi(argv[5]) );
```

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();
```

We can also obtain the direction of the lines by invoking the `GetOutputDirections()` method.

```
rescaler->SetInput( filter->GetOutputDirection() );
writer->SetInput( rescaler->GetOutput() );
writer->Update();
```

shows the result of applying the LineCorrelation edge detector filter to a SAR image.

The following classes provide similar functionality:

- `otb::LineCorrelationDetectorImageFilter`

The source code for this example can be found in the file
`Examples/FeatureExtraction/AssymmetricFusionOfLineDetectorExample.cxx`.

This example illustrates the use of the `otb::AssymmetricFusionOfLineDetectorImageFilter`.

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

```
#include "otbAssymmetricFusionOfLineDetectorImageFilter.h"
```

Then we must decide what pixel type to use for the image. We choose to make all computations with floating point precision and rescale the results between 0 and 255 in order to export PNG images.

```
  typedef  float  InternalPixelType;
  typedef  unsigned char  OutputPixelType;
```

The images are defined using the pixel type and the dimension.

```
  typedef otb::Image< InternalPixelType,  2 >   InternalImageType;
  typedef otb::Image< OutputPixelType,  2 >   OutputImageType;
```

The filter can be instantiated using the image types defined above.

```
  typedef otb::AssymmetricFusionOfLineDetectorImageFilter< InternalImageType,
                                            InternalImageType >  FilterType;
```

An `otb::ImageFileReader` class is also instantiated in order to read image data from a file.

```
  typedef otb::ImageFileReader< InternalImageType >  ReaderType;
```

An `otb::ImageFileWriter` is instantiated in order to write the output image to a file.

```
  typedef otb::ImageFileWriter< OutputImageType >  WriterType;
```

The intensity rescaling of the results will be carried out by the `itk::RescaleIntensityImageFilter` which is templated by the input and output image types.

```
  typedef itk::RescaleIntensityImageFilter< InternalImageType,
                                         OutputImageType > RescalerType;
```

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 same is done for the rescaler and the writer.

```
RescalerType::Pointer rescaler = RescalerType::New();
WriterType::Pointer writer = WriterType::New();
```

The `itk::RescaleIntensityImageFilter` needs to know which is the minimu and maximum values of the output generated image. Those can be chosen in a generic way by using the `NumericTraits` functions, since they are templated over the pixel type.

```
rescaler->SetOutputMinimum( itk::NumericTraits< OutputPixelType >::min());
rescaler->SetOutputMaximum( itk::NumericTraits< OutputPixelType >::max());
```

The image obtained with the reader is passed as input to the `otb::AssymetricFusionOfDetectorImageFilter`. The pipeline is built as follows.

```
filter->SetInput( reader->GetOutput() );
rescaler->SetInput( filter->GetOutput() );
writer->SetInput( rescaler->GetOutput() );
```

The methods `SetLengthLine()` and `SetWidthLine()` allow to set the minimum length and the typical witdh of the lines which are to be detected.

```
filter->SetLengthLine( atoi(argv[3]) );
filter->SetWidthLine( atoi(argv[4]) );
```

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();
```

Figure 13.7 shows the result of applying the AssymetricFusionOf edge detector filter to a SAR image.

13.4.2 Segment Extraction

The source code for this example can be found in the file
`Examples/FeatureExtraction/LocalHoughExample.cxx`.

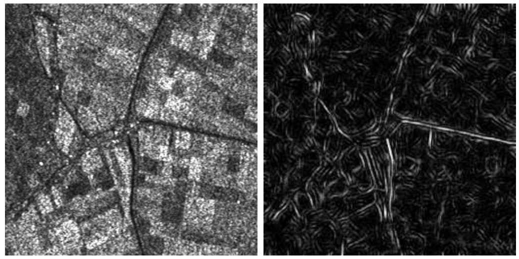


Figure 13.7: Result of applying the otb::AssymetricFusionOfDetectorImageFilter to a SAR image. From left to right : original image, line intensity.

This example illustrates the use of the otb::ExtractSegmentsImageFilter.

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

```
#include "otbLocalHoughFilter.h"
#include "otbDrawLineSpatialObjectListFilter.h"
#include "otbLineSpatialObjectList.h"
```

Then we must decide what pixel type to use for the image. We choose to make all computations with floating point precision and rescale the results between 0 and 255 in order to export PNG images.

```
  typedef  float  InternalPixelType;
  typedef  unsigned char  OutputPixelType;
```

The images are defined using the pixel type and the dimension.

```
  typedef otb::Image< InternalPixelType,  2 >   InternalImageType;
  typedef otb::Image< OutputPixelType,  2 >   OutputImageType;
```

The filter can be instantiated using the image types defined above.

```
  typedef otb::LocalHoughFilter< InternalImageType >      LocalHoughType;
  typedef otb::DrawLineSpatialObjectListFilter< InternalImageType,
                                        OutputImageType >  DrawLineListType;
```

An otb::ImageFileReader class is also instantiated in order to read image data from a file.

```
  typedef otb::ImageFileReader< InternalImageType >  ReaderType;
```

An `otb::ImageFileWriter` is instantiated in order to write the output image to a file.

```
typedef otb::ImageFileWriter< OutputImageType >  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();

LocalHoughType::Pointer localHough= LocalHoughType::New();

DrawLineListType::Pointer drawLineList= DrawLineListType::New();
```

The same is done for the writer.

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

The image obtained with the reader is passed as input to the `otb::ExtractSegmentsImageFilter`. The pipeline is built as follows.

```
localHough->SetInput( reader->GetOutput() );


drawLineList->SetInput( reader->GetOutput() );
drawLineList->SetInputLineSpatialObjectList( localHough->GetOutput() );
writer->SetFileName( argv[2] );
writer->SetInput( drawLineList->GetOutput() );
writer->Update();
```

Figure 13.8 shows the result of applying the `otb::LocalHoughImageFilter`.

13.5 Geometric Moments

13.5.1 Complex Moments

The complex geometric moments are defined as:

$$c_{pq} = \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} (x+iy)^p (x-iy)^q f(x,y) dxdy, \tag{13.2}$$

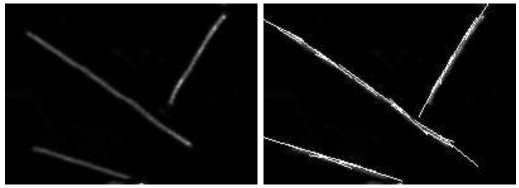


Figure 13.8: Result of applying the `otb::LocalHoughImageFilter`. From left to right : original image, extracted segments.

where x and y are the coordinates of the image $f(x,y)$, i is the imaginary unit and $p+q$ is the order of c_{pq}. The geometric moments are particularly useful in the case of scale changes.

Complex Moments for Images

The source code for this example can be found in the file
`Examples/FeatureExtraction/ComplexMomentImageExample.cxx`.

This example illustrates the use of the `otb::ComplexMomentImageFunction`.

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

```
#include "otbComplexMomentImageFunction.h"
```

The `otb::ComplexMomentImageFunction` is templated over the input image type and the output complex type value, so we start by defining:

```
typedef std::complex<float>                                   ComplexType;
typedef otb::ComplexMomentImageFunction<InputImageType,ComplexType>   CMType;

CMType::Pointer cmFunction =CMType::New();
```

Next, we plug the input image into the complex moment fucntion and we set its parameters.

```
reader->Update();
cmFunction->SetInputImage( reader->GetOutput() );
cmFunction->SetQ(Q);
cmFunction->SetP(P);
```

We can chose the pixel of the image which will used as center for the moment computation

```
InputImageType::IndexType center;
center[0]=50;
center[1]=50;
```

We can also choose the size of the neighborhood around the center pixel for the moment computation.

In order to get the value of the moment, we call the `EvaluateAtIndex` method.

```
ComplexType Result = cmFunction->EvaluateAtIndex(center);

std::cout << "The moment of order (" << P << "," << Q <<
                    ") is equal to " << Result << std:: endl;
```

Complex Moments for Paths

The source code for this example can be found in the file `Examples/FeatureExtraction/ComplexMomentPathExample.cxx`.

The complex moments can be computed on images, but sometimes we are interested in computing them on shapes extracted from images by segmentation algorithms. These shapes can be represented by `itk::Path`s. This example illustrates the use of the `otb::ComplexMomentPathFunction` for the computation of complex geometric moments on ITK paths.

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

```
#include "otbComplexMomentPathFunction.h"
```

The `otb::ComplexMomentPathFunction` is templated over the input path type and the output complex type value, so we start by defining:

```
const   unsigned int      Dimension = 2;

typedef itk::PolyLineParametricPath< Dimension > PathType;

typedef std::complex<double>                             ComplexType;
typedef otb::ComplexMomentPathFunction<PathType,ComplexType>   CMType;

CMType::Pointer cmFunction =CMType::New();
```

Next, we set the parameters of the plug the input path into the complex moment function and we set its parameters.

```
cmFunction->SetInputPath( path );
cmFunction->SetQ(Q);
cmFunction->SetP(P);
```

Since the paths are defined in physical coordinates, we do not need to set the center for the moment computation as we did with the `otb::ComplexMomentImageFunction`. The same applies for the size of the neighborhood around the center pixel for the moment computation. The moment computation is triggered by calling the `Evaluate` method.

```
ComplexType Result = cmFunction->Evaluate();

std::cout << "The moment of order (" << P << "," << Q <<
                    ") is equal to " << Result << std:: endl;
```

13.5.2 Hu Moments

Using the algebraic moment theory, H. Ming-Kuel obtained a family of 7 invariants with respect to planar transformations called Hu invariants, [42]. Those invariants can be seen as nonlinear combinations of the complex moments. Hu invariants have been very much used in object recognition during the last 30 years, since they are invariant to rotation, scaling and translation. [32] gives their expressions :

$$
\begin{array}{cccc}
\phi_1 = c_{11}; & \phi_2 = c_{20}c_{02}; & \phi_3 = c_{30}c_{03}; & \phi_4 = c_{21}c_{12}; \\
\phi_5 = Re(c_{30}c_{12}^3); & \phi_6 = Re(c_{21}c_{12}^2); & \phi_7 = Im(c_{30}c_{12}^3). &
\end{array}
\tag{13.3}
$$

[29] have used these invariants for the recognition of aircraft silhouettes. Flusser and Suk have used them for image registration, [47].

Hu Moments for Images

The source code for this example can be found in the file
`Examples/FeatureExtraction/HuMomentImageExample.cxx`.

This example illustrates the use of the `otb::HuMomentImageFunction`.

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

```
#include "otbHuImageFunction.h"
```

The `otb::HuImageFunction` is templated over the input image type and the output (real) type value, so we start by defining:

```
typedef float                             MomentType;
typedef otb::HuImageFunction<InputImageType,
                                         MomentType>   HuType;

HuType::Pointer hmFunction =HuType::New();
```

We can choose the region and the pixel of the image which will used as coordinate origin for the moment computation

```
InputImageType::RegionType   region;
InputImageType::SizeType     size;
InputImageType::IndexType    start;

start[0] = 0;
start[1] = 0;
size[0] = 50;
size[1] = 50;


reader->Update();
InputImageType::Pointer image = reader->GetOutput();

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

image->SetRegions(region);
image->Update();

InputImageType::IndexType center;
center[0]=start[0]+size[0]/2;
center[1]=start[1]+size[1]/2;
```

Next, we plug the input image into the complex moment function and we set its parameters.

```
hmFunction->SetInputImage( image );
hmFunction->SetMomentNumber(mMomentNumber);
```

In order to get the value of the moment, we call the `EvaluateAtIndex` method.

```
MomentType Result = hmFunction->EvaluateAtIndex(center);
```

```
    std::cout << "The moment of order " << mMomentNumber <<
                    " is equal to " << Result << std:: endl;
```

The following classes provide similar functionality:

- otb::HuPathFunction

13.5.3 Flusser Moments

The Hu invariants have been modified and improved by several authors. Flusser used these moments in order to produce a new family of descriptors of order higher than 3, [32]. These descriptors are invariant to scale and rotation. They have the following expressions:

$$\begin{array}{ccc} \psi_1 = c_{11} = \phi_1; & \psi_2 = c_{21}c_{12} = \phi_4; & \psi_3 = Re(c_{20}c_{12}^2) = \phi_6; \\ \psi_4 = Im(c_{20}c_{12}^2); & \psi_5 = Re(c_{30}c_{12}^3) = \phi_5; & \psi_6 = Im(c_{30}c_{12}^3) = \phi_7. \\ \psi_7 = c_{22}; & \psi_8 = Re(c_{31}c_{12}^2); & \psi_9 = Im(c_{31}c_{12}\ 2); \\ \psi_{10} = Re(c_{40}c_{12}^4); & \psi_{11} = Im(c_{40}c_{12}^2). & \end{array} \tag{13.4}$$

Examples

Flusser Moments for Images

The source code for this example can be found in the file
`Examples/FeatureExtraction/FlusserMomentImageExample.cxx`.

This example illustrates the use of the `otb::FlusserMomentImageFunction`.

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

```
#include "otbFlusserImageFunction.h"
```

The `otb::FlusserImageFunction` is templated over the input image type and the output (real) type value, so we start by defining:

```
    typedef float                                   MomentType;
    typedef otb::FlusserImageFunction<InputImageType,
                                            MomentType>   FlusserType;

    FlusserType::Pointer fmFunction =FlusserType::New();
```

We can choose the region and the pixel of the image which will used as coordinate origin for the moment computation

```
InputImageType::RegionType   region;
InputImageType::SizeType     size;
InputImageType::IndexType    start;

start[0] = 0;
start[1] = 0;
size[0] = 50;
size[1] = 50;


reader->Update();
InputImageType::Pointer image = reader->GetOutput();

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

image->SetRegions(region);
image->Update();

InputImageType::IndexType center;
center[0]=start[0]+size[0]/2;
center[1]=start[1]+size[1]/2;
```

Next, we plug the input image into the complex moment function and we set its parameters.

```
fmFunction->SetInputImage( image );
fmFunction->SetMomentNumber(mMomentNumber);
```

In order to get the value of the moment, we call the `EvaluateAtIndex` method.

```
MomentType Result = fmFunction->EvaluateAtIndex(center);

std::cout << "The moment of order " << mMomentNumber <<
                  " is equal to " << Result << std:: endl;
```

The following classes provide similar functionality:

- `otb::FlusserPathFunction`

13.6 Road extraction

Road extraction is a critical feature for an efficient use of high resolution satellite images. There are many applications of road extraction: update of GIS database, reference for image registration, help for identification algorithms and rapid mapping for example. Road network can be used to register an optical image with a map or an optical image with a radar image for example. Road network extraction can help for other algorithms: isolated building detection, bridge detection. In these cases, a rough extraction can be sufficient. In the context of response to crisis, a fast mapping is necessary: within 6 hours, infrastructures for the designated area are required. Within this timeframe, a manual extraction is inconceivable and an automatic help is necessary.

13.6.1 Road extraction filter

The source code for this example can be found in the file
`Examples/FeatureExtraction/ExtractRoadExample.cxx`.

The easiest way to use the road extraction filter provided by OTB is to use the composite filter. If a modification in the pipeline is required to adapt to a particular situation, the step by step example, described in the next section can be adapted.

This example demonstrates the use of the `otb::RoadExtractionFilter`. This filter is a composite filter achieving road extraction according to the algorithm adapted by E. Christophe and J. Inglada [15] from an original method proposed in [55].

The first step toward the use of this filter is the inclusion of the proper header files.

```
#include "otbPolyLineParametricPathWithValue.h"
#include "otbRoadExtractionFilter.h"
#include "otbDrawPathListFilter.h"
```

Then we must decide what pixel type to use for the image. We choose to do all the computation in floating point precision and rescale the results between 0 and 255 in order to export PNG images.

```
  typedef double        InputPixelType;
  typedef unsigned char OutputPixelType;
```

The images are defined using the pixel type and the dimension. Please note that the `otb::RoadExtractionFilter` needs an `otb::VectorImage` as input to handle multispectral images.

```
  typedef otb::VectorImage<InputPixelType,Dimension> InputVectorImageType;
```

```
typedef otb::Image<InputPixelType,Dimension> InputImageType;
typedef otb::Image<OutputPixelType,Dimension> OutputImageType;
```

We define the type of the polyline that the filter produces. We use the `otb::PolyLineParametricPathWithValue`, which allows the filter to produce a likehood value along with each polyline. The filter is able to produce `itk::PolyLineParametricPath` as well.

```
typedef otb::PolyLineParametricPathWithValue<InputPixelType,Dimension> PathType;
```

Now we can define the `otb::RoadExtractionFilter` that takes a multi-spectral image as input and produces a list of polylines.

```
typedef otb::RoadExtractionFilter<InputVectorImageType,
                                  PathType> RoadExtractionFilterType;
```

We also define an `otb::DrawPathListFilter` to draw the output polylines on an image, taking their likehood values into account.

```
typedef otb::DrawPathListFilter<InputImageType, PathType,
                                InputImageType> DrawPathFilterType;
```

The intensity rescaling of the results will be carried out by the `itk::RescaleIntensityImageFilter` which is templated by the input and output image types.

```
typedef itk::RescaleIntensityImageFilter<InputImageType,
                                         OutputImageType> RescalerType;
```

An `otb::ImageFileReader` class is also instantiated in order to read image data from a file. Then, an `otb::ImageFileWriter` is instantiated in order // to write the output image to a file.

```
typedef otb::ImageFileReader<InputVectorImageType> ReaderType;
typedef otb::ImageFileWriter<OutputImageType> WriterType;
```

The different filters composing our pipeline are created by invoking their `New()` methods, assigning the results to smart pointers.

```
ReaderType::Pointer reader = ReaderType::New();
RoadExtractionFilterType::Pointer roadExtractionFilter
  = RoadExtractionFilterType::New();
DrawPathFilterType::Pointer drawingFilter = DrawPathFilterType::New();
RescalerType::Pointer rescaleFilter = RescalerType::New();
WriterType::Pointer writer = WriterType::New();
```

The `otb::RoadExtractionFilter` needs to have a reference pixel corresponding to the spectral content likely to represent a road. This is done by passing a pixel to the filter. Here we suppose that the input image has four spectral bands.

```
InputVectorImageType::PixelType ReferencePixel;
ReferencePixel.SetSize(4);
ReferencePixel.SetElement(0,::atof(argv[3]));
ReferencePixel.SetElement(1,::atof(argv[4]));
ReferencePixel.SetElement(2,::atof(argv[5]));
ReferencePixel.SetElement(3,::atof(argv[6]));
roadExtractionFilter->SetReferencePixel(ReferencePixel);
```

We must also set the alpha parameter of the filter which allows us to tune the width of the roads we want to extract. Typical value is 1.0 and should be working in most situations.

```
roadExtractionFilter->SetAlpha(atof(argv[7]));
```

All other parameter should not influence the results too much in most situation and can be kept at the default value.

The amplitude threshold parameter tunes the sensitivity of the vectorization step. A typical value is $5 \cdot 10^{-5}$.

```
roadExtractionFilter->SetAmplitudeThreshold(atof(argv[8]));
```

The tolerance threshold tunes the sensitivity of the path simplification step. Typical value is 1.0.

```
roadExtractionFilter->SetTolerance(atof(argv[9]));
```

Roads are not likely to have sharp turns. Therefore we set the max angle parameter, as well as the link angular threshold. The value is typicaly $\frac{\pi}{8}$.

```
roadExtractionFilter->SetMaxAngle(atof(argv[10]));
roadExtractionFilter->SetAngularThreshold(atof(argv[10]));
```

The `otb::RoadExtractionFilter` performs two odd path removing operations at different stage of its execution. The first mean distance threshold and the second mean distance threshold set their criterion for removal. Path are removed if their mean distance between nodes is to small, since such path coming from previous filters are likely to be tortuous. The first removal operation as a typical mean distance threshold parameter of 1.0, and the second of 10.0.

```
roadExtractionFilter->SetFirstMeanDistanceThreshold(atof(argv[11]));
roadExtractionFilter->SetSecondMeanDistanceThreshold(atof(argv[12]));
```

The `otb::RoadExtractionFilter` is able to link path whose ends are near according to an euclidean distance criterion. The threshold for this distance to link a path is the distance threshold parameter. A typical value is 25.

```
roadExtractionFilter->SetDistanceThreshold(atof(argv[13]));
```

We will now create a black background image to draw the resulting polyline on. To achieve this we need to know the size of our input image. Therefore we trigger the `GenerateOutputInformation()` of the reader.

```
reader->GenerateOutputInformation();
InputImageType::Pointer blackBackground = InputImageType::New();
blackBackground->SetRegions(reader->GetOutput()->GetLargestPossibleRegion());
blackBackground->Allocate();
blackBackground->FillBuffer(0);
```

We tell the `otb::DrawPathListFilter` to try to use the likehood value embedded within the polyline as a value for drawing this polyline if possible.

```
drawingFilter->UseInternalPathValueOn();
```

The `itk::RescaleIntensityImageFilter` needs to know which is the minimum and maximum values of the output generated image. Those can be chosen in a generic way by using the `NumericTraits` functions, since they are templated over the pixel type.

```
rescaleFilter->SetOutputMinimum(itk::NumericTraits< OutputPixelType >::min());
rescaleFilter->SetOutputMaximum(itk::NumericTraits< OutputPixelType >::max());
```

Now it is time for some pipeline wiring.

```
roadExtractionFilter->SetInput(reader->GetOutput());
drawingFilter->SetInput(blackBackground);
drawingFilter->SetInputPath(roadExtractionFilter->GetOutput());
rescaleFilter->SetInput(drawingFilter->GetOutput());
```

The update of the pipeline is triggered by the `Update()` method of the rescale intensity filter.

```
rescaleFilter->Update();
```

Figure 13.9 shows the result of applying the road extraction filter to a fusionned Quickbird image.

13.6.2 Step by step road extraction

The source code for this example can be found in the file
`Examples/FeatureExtraction/ExtractRoadByStepsExample.cxx`.

This example illustrates the details of the `otb::RoadExtractionFilter`. This filter, described in the previous section, is a composite filter that includes all the steps below. Individual filters can be replaced to design a road detector targeted at SAR images for example.

The spectral angle is used to compute a grayscale image from the multispectral original image using `otb::SpectralAngleDistanceImageFilter`. The spectral angle is illustrated on Figure 13.10. Pixels corresponding to roads are in darker color.

```
typedef otb::SpectralAngleDistanceImageFilter<MultiSpectralImageType,
                                              InternalImageType> SAFilterType;
SAFilterType::Pointer saFilter = SAFilterType::New();
saFilter->SetReferencePixel(pixelRef);
saFilter->SetInput(multispectralReader->GetOutput());
```

Figure 13.9: Result of applying the `otb::RoadExtractionFilter` to a fusionned Quickbird image. From left to right : original image, extracted road with their likehood values (color are inverted for display).

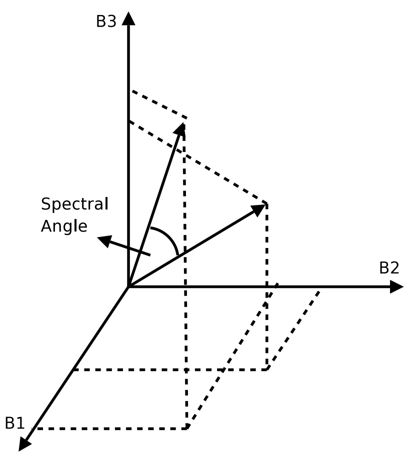


Figure 13.10: Illustration of the spectral angle for one pixel of a three-band image. One of the vector is the reference pixel and the other is the current pixel.

A square root is applied to the spectral angle image in order to enhance contrast between darker pixels (which are pixels of interest) with `itk::SqrtImageFilter`.

```
typedef itk::SqrtImageFilter<InternalImageType,InternalImageType> SqrtFilterType;
SqrtFilterType::Pointer sqrtFilter = SqrtFilterType::New();
sqrtFilter->SetInput(saFilter->GetOutput());
```

Use the Gaussian gradient filter compute the gradient direction and intensity (`itk::GradientRecursiveGaussianImageFilter`).

```
double sigma = alpha*(1.2/resolution+1);
typedef itk::GradientRecursiveGaussianImageFilter<InternalImageType,
             VectorImageType> GradientFilterType;
GradientFilterType::Pointer gradientFilter = GradientFilterType::New();
gradientFilter->SetSigma(sigma);
gradientFilter->SetInput(sqrtFilter->GetOutput());
```

Compute the scalar product of the neighboring pixels and keep the minimum value and the direction with `otb::NeighborhoodScalarProductFilter`. This is the line detector described in [55].

```
typedef otb::NeighborhoodScalarProductFilter<VectorImageType,
             InternalImageType,InternalImageType> NeighborhoodScalarProductType;
NeighborhoodScalarProductType::Pointer scalarFilter
  = NeighborhoodScalarProductType::New();
scalarFilter->SetInput(gradientFilter->GetOutput());
```

The resulting image is passed to the `otb::RemoveIsolatedByDirectionFilter` filter to remove pixels with no neighbor having the same direction.

```
typedef otb::RemoveIsolatedByDirectionFilter<InternalImageType,
             InternalImageType,InternalImageType> RemoveIsolatedByDirectionType;
RemoveIsolatedByDirectionType::Pointer removeIsolatedByDirectionFilter
                                  = RemoveIsolatedByDirectionType::New();
removeIsolatedByDirectionFilter->SetInput(scalarFilter->GetOutput());
removeIsolatedByDirectionFilter
  ->SetInputDirection(scalarFilter->GetOutputDirection());
```

We remove pixels having a direction corresponding to bright lines as we know that after the spectral angle, roads are in darker color with the `otb::RemoveWrongDirectionFilter` filter.

```
typedef otb::RemoveWrongDirectionFilter<InternalImageType,
             InternalImageType,InternalImageType> RemoveWrongDirectionType;
RemoveWrongDirectionType::Pointer removeWrongDirectionFilter
  = RemoveWrongDirectionType::New();
removeWrongDirectionFilter->SetInput(removeIsolatedByDirectionFilter->GetOutput());
removeWrongDirectionFilter->SetInputDirection(scalarFilter->GetOutputDirection());
```

We remove pixels which are not maximum on the direction perpendicular to the road direction with the `otb::NonMaxRemovalByDirectionFilter`.

```
typedef otb::NonMaxRemovalByDirectionFilter<InternalImageType,
             InternalImageType,InternalImageType> NonMaxRemovalByDirectionType;
NonMaxRemovalByDirectionType::Pointer nonMaxRemovalByDirectionFilter
                              = NonMaxRemovalByDirectionType::New();
nonMaxRemovalByDirectionFilter->SetInput(removeWrongDirectionFilter->GetOutput());
nonMaxRemovalByDirectionFilter
  ->SetInputDirection(scalarFilter->GetOutputDirection());
```

Extracted road are vectorized into polylines with `otb::VectorizationPathListFilter`.

```
typedef otb::VectorizationPathListFilter<InternalImageType,
             InternalImageType,PathType> VectorizationFilterType;
VectorizationFilterType::Pointer vectorizationFilter
  = VectorizationFilterType::New();
vectorizationFilter->SetInput(nonMaxRemovalByDirectionFilter->GetOutput());
vectorizationFilter->SetInputDirection(scalarFilter->GetOutputDirection());
vectorizationFilter->SetAmplitudeThreshold(atof(argv[8]));
```

However, this vectorization is too simple and need to be refined to be usable. First, we remove all aligned points to make one segment with `otb::SimplifyPathListFilter`. Then we break the polylines which have sharp angles as they are probably not road with `otb::BreakAngularPathListFilter`. Finally we remove path which are too short with `otb::RemoveTortuousPathListFilter`.

```
typedef otb::SimplifyPathListFilter<PathType> SimplifyPathType;
SimplifyPathType::Pointer simplifyPathListFilter = SimplifyPathType::New();
simplifyPathListFilter->SetTolerance(1.0);
simplifyPathListFilter->SetInput(vectorizationFilter->GetOutput());

typedef otb::BreakAngularPathListFilter<PathType> BreakAngularPathType;
BreakAngularPathType::Pointer breakAngularPathListFilter
  = BreakAngularPathType::New();
breakAngularPathListFilter->SetMaxAngle(M_PI/8.);
```

```
breakAngularPathListFilter->SetInput(simplifyPathListFilter->GetOutput());

typedef otb::RemoveTortuousPathListFilter<PathType> RemoveTortuousPathType;
RemoveTortuousPathType::Pointer removeTortuousPathListFilter
                        = RemoveTortuousPathType::New();
removeTortuousPathListFilter->SetMeanDistanceThreshold(1.0);
removeTortuousPathListFilter->SetInput(breakAngularPathListFilter->GetOutput());
```

Polylines within a certain range are linked (`otb::LinkPathListFilter`) to try to fill gaps due to occultations by vehicules, trees, etc. before simplifying polylines (`otb::SimplifyPathListFilter`) and removing the shortest ones with `otb::RemoveTortuousPathListFilter`.

```
typedef otb::LinkPathListFilter<PathType> LinkPathType;
LinkPathType::Pointer linkPathListFilter = LinkPathType::New();
linkPathListFilter->SetDistanceThreshold(25.0/resolution);
linkPathListFilter->SetAngularThreshold(M_PI/8);
linkPathListFilter->SetInput(removeTortuousPathListFilter->GetOutput());

SimplifyPathType::Pointer simplifyPathListFilter2 = SimplifyPathType::New();
simplifyPathListFilter2->SetTolerance(1.0);
simplifyPathListFilter2->SetInput(linkPathListFilter->GetOutput());

RemoveTortuousPathType::Pointer removeTortuousPathListFilter2
                        = RemoveTortuousPathType::New();
removeTortuousPathListFilter2->SetMeanDistanceThreshold(10.0);
removeTortuousPathListFilter2->SetInput(simplifyPathListFilter2->GetOutput());
```

A value can be associated with each polyline according to pixel values under the polyline with `otb::LikehoodPathListFilter`. A higher value will mean a higher likelihood to be a road.

```
typedef otb::LikehoodPathListFilter<PathType,
             InternalImageType> PathListToPathListWithValueType;
PathListToPathListWithValueType::Pointer pathListConverter
              = PathListToPathListWithValueType::New();
pathListConverter->SetInput(removeTortuousPathListFilter2->GetOutput());
pathListConverter->SetInputImage(nonMaxRemovalByDirectionFilter->GetOutput());
```

A black background image is built to draw the path on.

```
InternalImageType::Pointer output = InternalImageType::New();
output->SetRegions(multispectralReader->GetOutput()
   ->GetLargestPossibleRegion());
output->Allocate();
output->FillBuffer(0.0);
```

Polylines are drawn on a black background image with `otb::DrawPathListFilter`. The `SetUseIternalValues()` tell the drawing filter to draw the path with its likehood value.

```
typedef otb::DrawPathListFilter<InternalImageType, PathType,
             InternalImageType> DrawPathType;
DrawPathType::Pointer drawPathListFilter = DrawPathType::New();
drawPathListFilter->SetInput(output);
drawPathListFilter->SetInputPath(pathListConverter->GetOutput());
drawPathListFilter->SetUseInternalPathValue(true);
```

The output from the drawing filter contains very small values (likehood values). Therefore the image has to be rescaled to be viewed. The whole pipeline is executed by invoking the `Update()` method on this last filter.

```
typedef itk::RescaleIntensityImageFilter<InternalImageType,
             InternalImageType> RescalerType;
RescalerType::Pointer rescaler = RescalerType::New();
rescaler->SetOutputMaximum(255);
rescaler->SetOutputMinimum(0);
rescaler->SetInput(drawPathListFilter->GetOutput());
rescaler->Update();
```

Figures 13.11 and 13.12 show the result of applying the road extraction by steps to a fusionned Quickbird image. The result image is a RGB composition showing the extracted path in red. Full processing took about 3 seconds for each image.

Figure 13.11: Result of applying the road extraction by steps pipeline to a fusionned Quickbird image. From left to right : original image, extracted road with their likehood values.

Figure 13.12: Result of applying the road extraction by steps pipeline to a fusionned Quickbird image. From left to right : original image, extracted road with their likehood values.

CHAPTER

FOURTEEN

Image Segmentation

Segmentation of remote sensing 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 objects to be segmented.

The Insight Toolkit provides a basic set of algorithms that can be used to develop and customize a full segmentation application. They are therefore available in the Orfeo Toolbox. Some of the most commonly used segmentation components are described in the following sections.

14.1 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.

14.1.1 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 example can be found in the file
`Examples/Segmentation/ConnectedThresholdImageFilter.cxx`.

The following example illustrates the use of the `itk::ConnectedThresholdImageFilter`. This filter uses the flood fill iterator. Most of the algorithmic complexity of a region growing method comes from visiting neighboring pixels. The flood fill iterator assumes this responsibility and greatly simplifies the implementation of the region growing algorithm. Thus the algorithm is left to establish a criterion to decide whether a particular pixel should be included in the current region or not.

The criterion used by the ConnectedThresholdImageFilter is based on an interval of intensity values provided by the user. Values of lower and upper threshold should be provided. The region growing algorithm includes those pixels whose intensities are inside the interval.

$$I(\mathbf{X}) \in [\text{lower}, \text{upper}] \tag{14.1}$$

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. In this particular example we use the `itk::CurvatureFlowImageFilter`, hence we need to include its header file.

```
#include "itkCurvatureFlowImageFilter.h"
```

We declare the image type based on a particular pixel type and dimension. In this case the `float` type is used for the pixels due to the requirements of the smoothing filter.

```
  typedef   float           InternalPixelType;
  const     unsigned int    Dimension = 2;
  typedef otb::Image< InternalPixelType, Dimension >  InternalImageType;
```

The smoothing filter is instantiated using the image type as a template parameter.

```
  typedef itk::CurvatureFlowImageFilter< InternalImageType, InternalImageType >
    CurvatureFlowImageFilterType;
```

Then the filter is created by invoking the `New()` method and assigning the result to a `itk::SmartPointer`.

```
CurvatureFlowImageFilterType::Pointer smoothing =
                       CurvatureFlowImageFilterType::New();
```

We now declare the type of the region growing filter. In this case it is the ConnectedThresholdImageFilter.

```
typedef itk::ConnectedThresholdImageFilter< InternalImageType,
                                  InternalImageType > ConnectedFilterType;
```

Then we construct one filter of this class using the `New()` method.

```
ConnectedFilterType::Pointer connectedThreshold = ConnectedFilterType::New();
```

Now it is time to connect a simple, linear pipeline. A file reader is added at the beginning of the pipeline and a cast filter and writer are added at the end. The cast filter is required to convert `float` pixel types to integer types since only a few image file formats support `float` types.

```
smoothing->SetInput( reader->GetOutput() );
connectedThreshold->SetInput( smoothing->GetOutput() );
caster->SetInput( connectedThreshold->GetOutput() );
writer->SetInput( caster->GetOutput() );
```

The CurvatureFlowImageFilter requires a couple of parameters to be defined. The following are typical values, however they may have to be adjusted depending on the amount of noise present in the input image.

```
smoothing->SetNumberOfIterations( 5 );
smoothing->SetTimeStep( 0.125 );
```

The ConnectedThresholdImageFilter has two main parameters to be defined. They are the lower and upper thresholds of the interval in which intensity values should fall in order to be included in the region. Setting these two values too close will not allow enough flexibility for the region to grow. Setting them too far apart will result in a region that engulfs the image.

```
connectedThreshold->SetLower(  lowerThreshold  );
connectedThreshold->SetUpper(  upperThreshold  );
```

The output of this filter is a binary image with zero-value pixels everywhere except on the extracted region. The intensity value set inside the region is selected with the method `SetReplaceValue()`

Structure	Seed Index	Lower	Upper	Output Image
Road	$(110,38)$	50	100	Second from left in Figure 14.1
Shadow	$(118,100)$	0	10	Third from left in Figure 14.1
Building	$(169,146)$	220	255	Fourth from left in Figure 14.1

Table 14.1: Parameters used for segmenting some structures shown in Figure 14.1 with the filter `itk::ConnectedThresholdImageFilter`.

```
connectedThreshold->SetReplaceValue(
             itk::NumericTraits<OutputPixelType>::max() );
```

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 structure to be segmented. The seed is passed in the form of a `itk::Index` to the `SetSeed()` method.

```
connectedThreshold->SetSeed( index );
```

The invocation of the `Update()` method on the writer triggers the execution of the pipeline. It is usually wise to put update calls in a `try/catch` block in case errors occur and exceptions are thrown.

```
try
  {
  writer->Update();
  }
catch( itk::ExceptionObject & excep )
  {
  std::cerr << "Exception caught !" << std::endl;
  std::cerr << excep << std::endl;
  }
```

Let's run this example using as input the image `QB_Suburb.png` provided in the directory `Examples/Data`. We can easily segment the major structures by providing seeds in the appropriate locations and defining values for the lower and upper thresholds. Figure 14.1 illustrates several examples of segmentation. The parameters used are presented in Table 14.1.

Notice that some objects are not being completely segmented. This illustrates the vulnerability of the region growing methods when the structures to be segmented do not have a homogeneous statistical distribution over the image space. You may want to experiment with different values of the lower and upper thresholds to verify how the accepted region will extend.

Another option for segmenting regions is to take advantage of the functionality provided by the ConnectedThresholdImageFilter for managing multiple seeds. The seeds can be passed one by one to the filter using the `AddSeed()` method. You could imagine a user interface in which

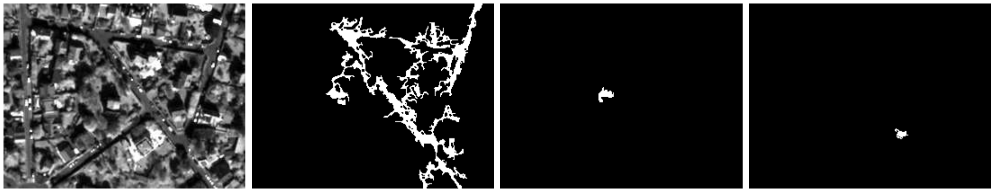


Figure 14.1: Segmentation results for the ConnectedThreshold filter for various seed points.

an operator clicks on multiple points of the object to be segmented and each selected point is passed as a seed to this filter.

14.1.2 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 example can be found in the file
`Examples/Segmentation/OtsuThresholdImageFilter.cxx`.

This example illustrates how to use the `itk::OtsuThresholdImageFilter`.

```
#include "itkOtsuThresholdImageFilter.h"
```

The next step is to decide which pixel types to use for the input and output images.

```
  typedef  unsigned char  InputPixelType;
  typedef  unsigned char  OutputPixelType;
```

The input and output image types are now defined using their respective pixel types and dimensions.

```
  typedef otb::Image< InputPixelType,  2 >   InputImageType;
  typedef otb::Image< OutputPixelType, 2 >   OutputImageType;
```

The filter type can be instantiated using the input and output image types defined above.

```
  typedef itk::OtsuThresholdImageFilter<
               InputImageType, OutputImageType >  FilterType;
```

An `otb::ImageFileReader` class is also instantiated in order to read image data from a file. (See Section 6 on page 95 for more information about reading and writing data.)

```
typedef otb::ImageFileReader< InputImageType >  ReaderType;
```

An `otb::ImageFileWriter` is instantiated in order to write the output image to a file.

```
typedef otb::ImageFileWriter< InputImageType >  WriterType;
```

Both the filter and the reader are created by invoking their `New()` methods and assigning the result to `itk::SmartPointer`s.

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

The image obtained with the reader is passed as input to the OtsuThresholdImageFilter.

```
filter->SetInput( reader->GetOutput() );
```

The method `SetOutsideValue()` defines the intensity value to be assigned to those pixels whose intensities are outside the range defined by the lower and upper thresholds. The method `SetInsideValue()` defines the intensity value to be assigned to pixels with intensities falling inside the threshold range.

```
filter->SetOutsideValue( outsideValue );
filter->SetInsideValue(  insideValue  );
```

The method `SetNumberOfHistogramBins()` defines the number of bins to be used for computing the histogram. This histogram will be used internally in order to compute the Otsu threshold.

```
filter->SetNumberOfHistogramBins( 128 );
```

The execution of the filter is triggered by invoking the `Update()` method. If the filter's output has been passed as input to subsequent filters, the `Update()` call on any posterior filters in the pipeline will indirectly trigger the update of this filter.

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

We print out here the Threshold value that was computed internally by the filter. For this we invoke the `GetThreshold` method.

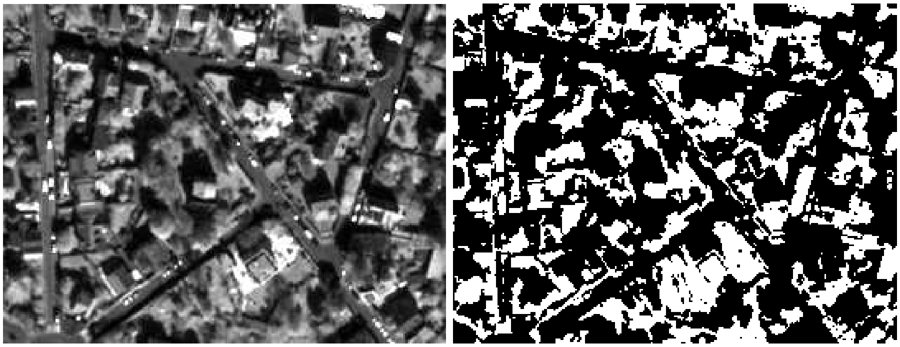


Figure 14.2: Effect of the OtsuThresholdImageFilter.

```
int threshold = filter->GetThreshold();
std::cout << "Threshold = " << threshold << std::endl;
```

Figure 14.2 illustrates the effect of this filter. 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.

The following classes provide similar functionality:

- itk::ThresholdImageFilter

The source code for this example can be found in the file
Examples/Segmentation/OtsuMultipleThresholdImageFilter.cxx.

This example illustrates how to use the itk::OtsuMultipleThresholdsCalculator.

```
#include "itkOtsuMultipleThresholdsCalculator.h"
```

OtsuMultipleThresholdsCalculator calculates thresholds for a give histogram so as to maximize the between-class variance. We use ScalarImageToHistogramGenerator to generate histograms

```
typedef itk::Statistics::ScalarImageToHistogramGenerator< InputImageType >
  ScalarImageToHistogramGeneratorType;
typedef itk::OtsuMultipleThresholdsCalculator<
  ScalarImageToHistogramGeneratorType::HistogramType >   CalculatorType;
```

Once thresholds are computed we will use BinaryThresholdImageFilter to segment the input image into segments.

```
typedef itk::BinaryThresholdImageFilter< InputImageType, OutputImageType >
  FilterType;
```

```
ScalarImageToHistogramGeneratorType::Pointer scalarImageToHistogramGenerator =
  ScalarImageToHistogramGeneratorType::New();
CalculatorType::Pointer calculator = CalculatorType::New();
FilterType::Pointer filter = FilterType::New();

scalarImageToHistogramGenerator->SetNumberOfBins(128);
int nbThresholds = argc-2;
calculator->SetNumberOfThresholds( nbThresholds );
```

The pipeline will look as follows:

```
scalarImageToHistogramGenerator->SetInput(reader->GetOutput());
calculator->SetInputHistogram(scalarImageToHistogramGenerator->GetOutput());
filter->SetInput( reader->GetOutput() );
writer->SetInput(filter->GetOutput());
```

Thresholds are obtained using the `GetOutput` method

```
const CalculatorType::OutputType &thresholdVector = calculator->GetOutput();
CalculatorType::OutputType::const_iterator itNum = thresholdVector.begin();

for(; itNum < thresholdVector.end(); itNum++)
  {
  std::cout << "OtsuThreshold["
    << (int)(itNum - thresholdVector.begin())
    << "] = " <<
    static_cast<itk::NumericTraits<CalculatorType::MeasurementType>::PrintType>
    (*itNum) << std::endl;

  }
```

Figure 14.3 illustrates the effect of this filter.

The following classes provide similar functionality:

- `itk::ThresholdImageFilter`

14.1.3 Neighborhood Connected

The source code for this example can be found in the file
`Examples/Segmentation/NeighborhoodConnectedImageFilter.cxx`.

The following example illustrates the use of the `itk::NeighborhoodConnectedImageFilter`. This filter is a close variant of the `itk::ConnectedThresholdImageFilter`. On one hand, the ConnectedThresholdImageFilter accepts a pixel in the region if its intensity is in the interval

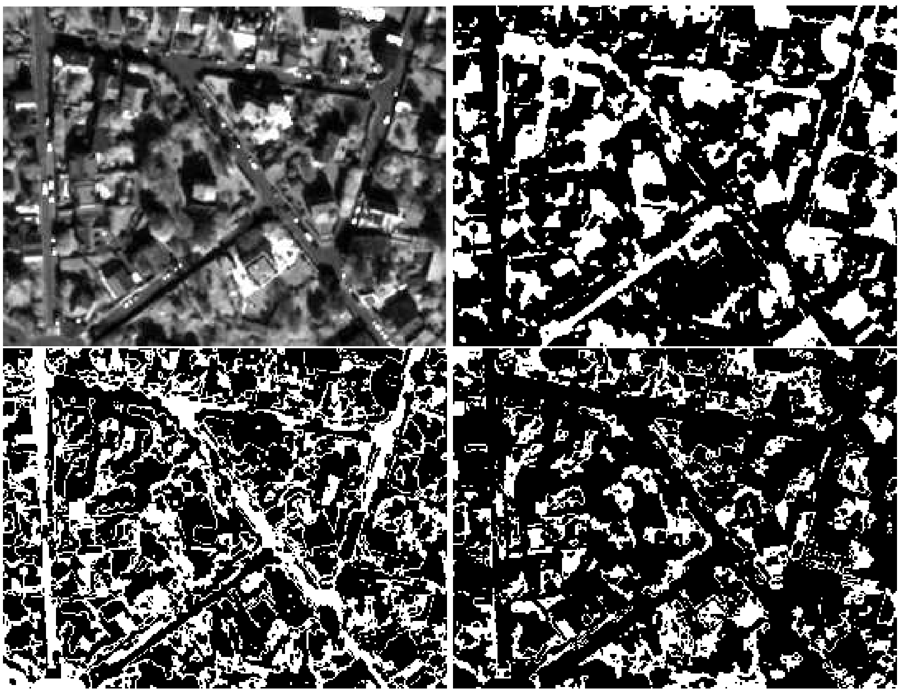


Figure 14.3: Effect of the OtsuMultipleThresholdImageFilter.

defined by two user-provided threshold values. The NeighborhoodConnectedImageFilter, on the other hand, will only accept a pixel if **all** its neighbors have intensities that fit in the interval. The size of the neighborhood to be considered around each pixel is defined by a user-provided integer radius.

The reason for considering the neighborhood intensities instead of only the current pixel intensity is that small structures are less likely to be accepted in the region. The operation of this filter is equivalent to applying the ConnectedThresholdImageFilter followed by mathematical morphology erosion using a structuring element of the same shape as the neighborhood provided to the NeighborhoodConnectedImageFilter.

```
#include "itkNeighborhoodConnectedImageFilter.h"
```

The `itk::CurvatureFlowImageFilter` is used here to smooth the image while preserving edges.

```
#include "itkCurvatureFlowImageFilter.h"
```

We now define the image type using a particular pixel type and image dimension. In this case the `float` type is used for the pixels due to the requirements of the smoothing filter.

```
  typedef   float           InternalPixelType;
  const     unsigned int    Dimension = 2;
  typedef otb::Image< InternalPixelType, Dimension >  InternalImageType;
```

The smoothing filter type is instantiated using the image type as a template parameter.

```
  typedef   itk::CurvatureFlowImageFilter<InternalImageType, InternalImageType>
    CurvatureFlowImageFilterType;
```

Then, the filter is created by invoking the `New()` method and assigning the result to a `itk::SmartPointer`.

```
  CurvatureFlowImageFilterType::Pointer smoothing =
                         CurvatureFlowImageFilterType::New();
```

We now declare the type of the region growing filter. In this case it is the NeighborhoodConnectedImageFilter.

```
  typedef itk::NeighborhoodConnectedImageFilter<InternalImageType,
                                    InternalImageType > ConnectedFilterType;
```

One filter of this class is constructed using the `New()` method.

```
ConnectedFilterType::Pointer neighborhoodConnected = ConnectedFilterType::New();
```

Now it is time to create a simple, linear data processing pipeline. A file reader is added at the beginning of the pipeline and a cast filter and writer are added at the end. The cast filter is required to convert `float` pixel types to integer types since only a few image file formats support `float` types.

```
smoothing->SetInput( reader->GetOutput() );
neighborhoodConnected->SetInput( smoothing->GetOutput() );
caster->SetInput( neighborhoodConnected->GetOutput() );
writer->SetInput( caster->GetOutput() );
```

The CurvatureFlowImageFilter requires a couple of parameters to be defined. The following are typical values for $2D$ images. However they may have to be adjusted depending on the amount of noise present in the input image.

```
smoothing->SetNumberOfIterations( 5 );
smoothing->SetTimeStep( 0.125 );
```

The NeighborhoodConnectedImageFilter requires that two main parameters are specified. They are the lower and upper thresholds of the interval in which intensity values must fall to be included in the region. Setting these two values too close will not allow enough flexibility for the region to grow. Setting them too far apart will result in a region that engulfs the image.

```
neighborhoodConnected->SetLower(  lowerThreshold  );
neighborhoodConnected->SetUpper(  upperThreshold  );
```

Here, we add the crucial parameter that defines the neighborhood size used to determine whether a pixel lies in the region. The larger the neighborhood, the more stable this filter will be against noise in the input image, but also the longer the computing time will be. Here we select a filter of radius 2 along each dimension. This results in a neighborhood of 5×5 pixels.

```
InternalImageType::SizeType   radius;

radius[0] = 2;   // two pixels along X
radius[1] = 2;   // two pixels along Y

neighborhoodConnected->SetRadius( radius );
```

As in the ConnectedThresholdImageFilter we must now provide the intensity value to be used for the output pixels accepted in the region and at least one seed point to define the initial region.

```
neighborhoodConnected->SetSeed( index );
neighborhoodConnected->SetReplaceValue( 255 );
```

Structure	Seed Index	Lower	Upper	Output Image
Road	$(110,38)$	50	100	Second from left in Figure 14.4
Shadow	$(118,100)$	0	10	Third from left in Figure 14.4
Building	$(169,146)$	220	255	Fourth from left in Figure 14.4

Table 14.2: Parameters used for segmenting some structures shown in Figure 14.4 with the filter `itk::NeighborhoodConnectedThresholdImageFilter`.

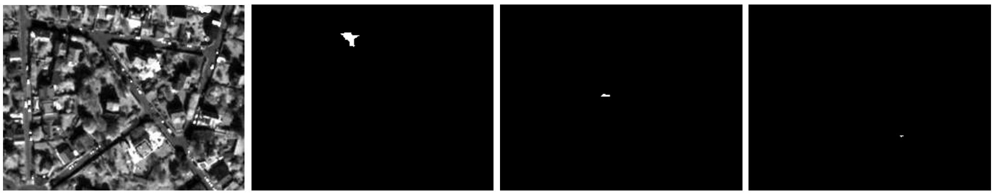


Figure 14.4: Segmentation results for the NeighborhoodConnectedThreshold filter for various seed points.

The invocation of the `Update()` method on the writer triggers the execution of the pipeline. It is usually wise to put update calls in a `try/catch` block in case errors occur and exceptions are thrown.

```
try
  {
  writer->Update();
  }
catch( itk::ExceptionObject & excep )
  {
  std::cerr << "Exception caught !" << std::endl;
  std::cerr << excep << std::endl;
  }
```

Let's run this example using as input the image `QB_Suburb.png` provided in the directory `Examples/Data`. We can easily segment the major structures by providing seeds in the appropriate locations and defining values for the lower and upper thresholds. Figure 14.4 illustrates several examples of segmentation. The parameters used are presented in Table 14.2.

As with the ConnectedThresholdImageFilter, several seeds could be provided to the filter by using the `AddSeed()` method. Compare the output of Figure 14.4 with those of Figure 14.1 produced by the ConnectedThresholdImageFilter. You may want to play with the value of the neighborhood radius and see how it affect the smoothness of the segmented object borders, the size of the segmented region and how much that costs in computing time.

14.1.4 Confidence Connected

The source code for this example can be found in the file
`Examples/Segmentation/ConfidenceConnected.cxx`.

The following example illustrates the use of the `itk::ConfidenceConnectedImageFilter`. The criterion used by the ConfidenceConnectedImageFilter is based on simple statistics of the current region. First, the algorithm computes the mean and standard deviation of intensity values for all the pixels currently included in the region. A user-provided factor is used to multiply the standard deviation and define a range around the mean. Neighbor pixels whose intensity values fall inside the range are accepted and included in the region. When no more neighbor pixels are found that satisfy the criterion, the algorithm is considered to have finished its first iteration. At that point, the mean and standard deviation of the intensity levels are recomputed using all the pixels currently included in the region. This mean and standard deviation defines a new intensity range that is used to visit current region neighbors and evaluate whether their intensity falls inside the range. This iterative process is repeated until no more pixels are added or the maximum number of iterations is reached. The following equation illustrates the inclusion criterion used by this filter,

$$I(\mathbf{X}) \in [m - f\sigma, m + f\sigma] \tag{14.2}$$

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. 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 otb::Image< InternalPixelType, Dimension >  InternalImageType;
```

The smoothing filter type is instantiated using the image type as a template parameter.

```
typedef itk::CurvatureFlowImageFilter< InternalImageType, InternalImageType >
  CurvatureFlowImageFilterType;
```

Next the filter is created by invoking the `New()` method and assigning the result to a `itk::SmartPointer`.

```
CurvatureFlowImageFilterType::Pointer smoothing =
                       CurvatureFlowImageFilterType::New();
```

We now declare the type of the region growing filter. In this case it is the ConfidenceConnectedImageFilter.

```
typedef itk::ConfidenceConnectedImageFilter<InternalImageType, InternalImageType>
  ConnectedFilterType;
```

Then, we construct one filter of this class using the `New()` method.

```
ConnectedFilterType::Pointer confidenceConnected = ConnectedFilterType::New();
```

Now it is time to create a simple, linear pipeline. A file reader is added at the beginning of the pipeline and a cast filter and writer are added at the end. The cast filter is required here to convert `float` pixel types to integer types since only a few image file formats support `float` types.

```
smoothing->SetInput( reader->GetOutput() );
confidenceConnected->SetInput( smoothing->GetOutput() );
caster->SetInput( confidenceConnected->GetOutput() );
writer->SetInput( caster->GetOutput() );
```

The CurvatureFlowImageFilter requires defining two parameters. The following are typical values. However they may have to be adjusted depending on the amount of noise present in the input image.

```
smoothing->SetNumberOfIterations( 5 );
smoothing->SetTimeStep( 0.125 );
```

The ConfidenceConnectedImageFilter requires defining two parameters. First, the factor f that the defines how large the range of intensities will be. Small values of the multiplier will restrict the inclusion of pixels to those having very similar intensities to those in the current region. Larger values of the multiplier will relax the accepting condition and will result in more generous growth of the region. Values that are too large will cause the region to grow into neighboring regions that may actually belong to separate structures.

```
confidenceConnected->SetMultiplier( 2.5 );
```

The number of iterations is specified based on the homogeneity of the intensities of the object to be segmented. Highly homogeneous regions may only require a couple of iterations. Regions with ramp effect, may require more iterations. In practice, it seems to be more important to carefully select the multiplier factor than the number of iterations. However, keep in mind that there is no reason to assume that this algorithm should converge to a stable region. It is possible that by letting the algorithm run for more iterations the region will end up engulfing the entire image.

```
confidenceConnected->SetNumberOfIterations( 5 );
```

The output of this filter is a binary image with zero-value pixels everywhere except on the extracted region. The intensity value to be set inside the region is selected with the method `SetReplaceValue()`

```
confidenceConnected->SetReplaceValue( 255 );
```

The initialization of the algorithm requires the user to provide a seed point. It is convenient to select this point to be placed in a *typical* region of the structure to be segmented. A small neighborhood around the seed point will be used to compute the initial mean and standard deviation for the inclusion criterion. The seed is passed in the form of a `itk::Index` to the `SetSeed()` method.

```
confidenceConnected->SetSeed( index );
```

The size of the initial neighborhood around the seed is defined with the method `SetInitialNeighborhoodRadius()`. The neighborhood will be defined as an N-dimensional rectangular region with $2r+1$ pixels on the side, where r is the value passed as initial neighborhood radius.

```
confidenceConnected->SetInitialNeighborhoodRadius( 2 );
```

The invocation of the `Update()` method on the writer triggers the execution of the pipeline. It is recommended to place update calls in a `try/catch` block in case errors occur and exceptions are thrown.

```
try
  {
  writer->Update();
  }
catch( itk::ExceptionObject & excep )
  {
```

Structure	Seed Index	Lower	Upper	Output Image
Road	$(110,38)$	50	100	Second from left in Figure 14.1
Shadow	$(118,100)$	0	10	Third from left in Figure 14.1
Building	$(169,146)$	220	255	Fourth from left in Figure 14.1

Table 14.3: Parameters used for segmenting some structures shown in Figure 14.1 with the filter `itk::ConnectedThresholdImageFilter`.

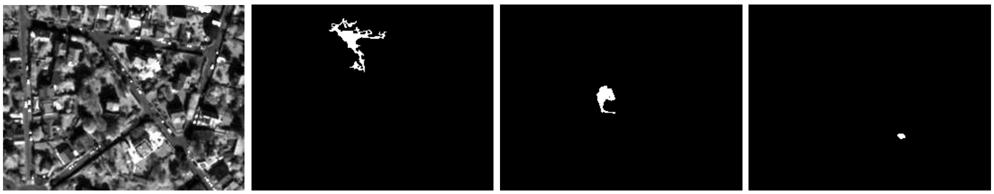


Figure 14.5: Segmentation results for the ConfidenceConnected filter for various seed points.

```
std::cerr << "Exception caught !" << std::endl;
std::cerr << excep << std::endl;
}
```

Let's now run this example using as input the image `QB_Suburb.png` provided in the directory `Examples/Data`. We can easily segment structures by providing seeds in the appropriate locations. For example

14.2 Segmentation Based on Watersheds

14.2.1 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 [100, 101].

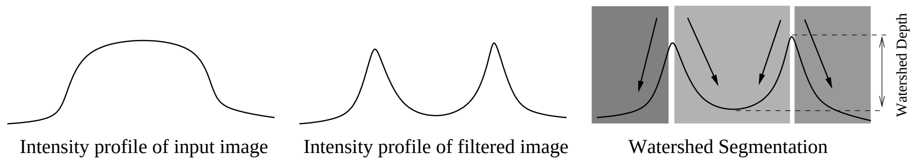


Figure 14.6: A fuzzy-valued boundary map, from an image or set of images, is segmented using local minima and catchment basins.

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 14.6. 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" [52, 53, 31] in the graph of f. In the 1D case ($U \subset \Re$), the watershed boundaries are the local maxima of f, and the results of the watershed segmentation is trivial. For higher-dimensional image domains, the watershed boundaries are not simply local phenomena; they depend on the shape of the entire watershed.

The drawback of watershed segmentation is that it produces a region for each local minimum—in practice too many regions—and an over segmentation results. To alleviate this, we can establish a minimum watershed depth. The watershed depth is the difference in height between the watershed minimum and the lowest boundary point. In other words, it is the maximum depth of water a region could hold without flowing into any of its neighbors. Thus, a watershed segmentation algorithm can sequentially combine watersheds whose depths fall below the minimum until all of the watersheds are of sufficient depth. This depth measurement can be combined with other saliency measurements, such as size. The result is a segmentation containing regions whose boundaries and size are significant. Because the merging process is sequential, it produces a hierarchy of regions, as shown in Figure 14.7. 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 [101].

There are two different algorithms commonly used to implement watersheds: top-down and bottom-up. The top-down, gradient descent strategy was chosen for ITK because we want to consider the output of multi-scale differential operators, and the f in question will therefore have floating point values. The bottom-up strategy starts with seeds at the local minima in the image and grows regions outward and upward at discrete intensity levels (equivalent to a

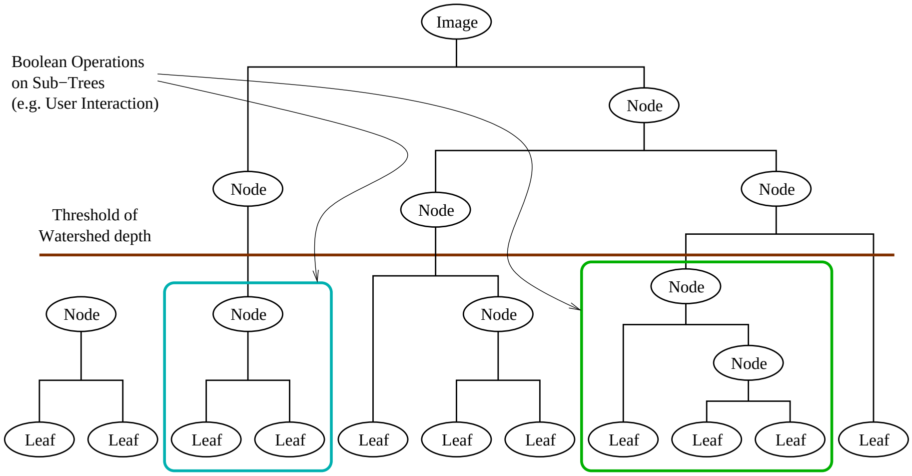


Figure 14.7: A watershed segmentation combined with a saliency measure (watershed depth) produces a hierarchy of regions. Structures can be derived from images by either thresholding the saliency measure or combining subtrees within the hierarchy.

sequence of morphological operations and sometimes called *morphological watersheds* [79].) This limits the accuracy by enforcing a set of discrete gray levels on the image.

Figure 14.8 shows how the ITK image-to-image watersheds filter is constructed. The filter is actually a collection of smaller filters that modularize the several steps of the algorithm in a mini-pipeline. The segmenter object creates the initial segmentation via steepest descent from each pixel to local minima. Shallow background regions are removed (flattened) before segmentation using a simple minimum value threshold (this helps to minimize oversegmentation 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

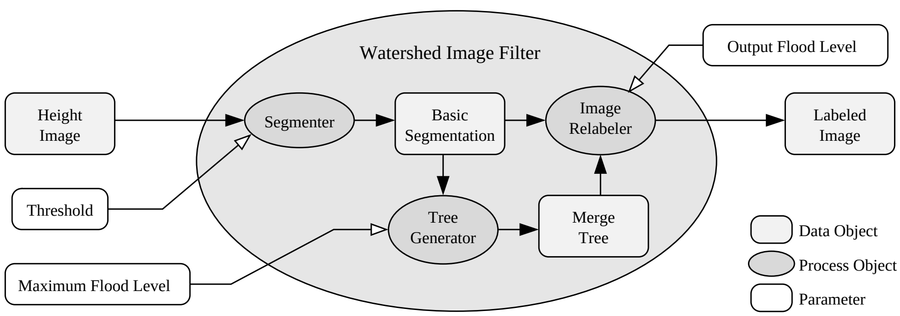


Figure 14.8: The construction of the Insight watersheds filter.

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 14.8 connected to their relevant processing stages.

14.2.2 Using the ITK Watershed Filter

The source code for this example can be found in the file
`Examples/Segmentation/WatershedSegmentation.cxx`.

The following example illustrates how to preprocess and segment images using the `itk::WatershedImageFilter`. Note that the care with which the data is preprocessed will greatly affect the quality of your result. Typically, the best results are obtained by preprocessing the original image with an edge-preserving diffusion filter, such as one of the anisotropic diffusion filters, or with the bilateral image filter. As noted in Section 14.2.1, the height function used as input should be created such that higher positive values correspond to object boundaries. A suitable height function for many applications can be generated as the gradient magnitude of the image to be segmented.

The `itk::VectorGradientMagnitudeAnisotropicDiffusionImageFilter` class is used to smooth the image and the `itk::VectorGradientMagnitudeImageFilter` is used to generate the height function. We begin by including all preprocessing filter header files and the header file for the WatershedImageFilter. We use the vector versions of these filters because the input data is a color image.

```
#include "itkVectorGradientAnisotropicDiffusionImageFilter.h"
#include "itkVectorGradientMagnitudeImageFilter.h"
#include "itkWatershedImageFilter.h"
```

We now declare the image and pixel types to use for instantiation of the filters. All of these filters expect real-valued pixel types in order to work properly. The preprocessing stages are done directly on the vector-valued data and the segmentation is done using floating point scalar data. Images are converted from RGB pixel type to numerical vector type using `itk::VectorCastImageFilter`. Please pay attention to the fact that we are using `itk::Images` since the `itk::VectorGradientMagnitudeImageFilter` has some internal typedefs which make polymorfism impossible.

```
  typedef itk::RGBPixel<unsigned char>   RGBPixelType;
  typedef otb::Image<RGBPixelType, 2>    RGBImageType;
  typedef itk::Vector<float, 3>          VectorPixelType;
  typedef itk::Image<VectorPixelType, 2> VectorImageType;
  typedef itk::Image<unsigned long, 2>   LabeledImageType;
  typedef itk::Image<float, 2>           ScalarImageType;
```

The various image processing filters are declared using the types created above and eventually used in the pipeline.

```
typedef otb::ImageFileReader<RGBImageType> FileReaderType;
typedef itk::VectorCastImageFilter<RGBImageType, VectorImageType>
  CastFilterType;
typedef itk::VectorGradientAnisotropicDiffusionImageFilter<VectorImageType,
  VectorImageType>  DiffusionFilterType;
typedef itk::VectorGradientMagnitudeImageFilter<VectorImageType,float,ScalarImageType>
  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 7.6.2 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 for that converts scalar images to RGB images.

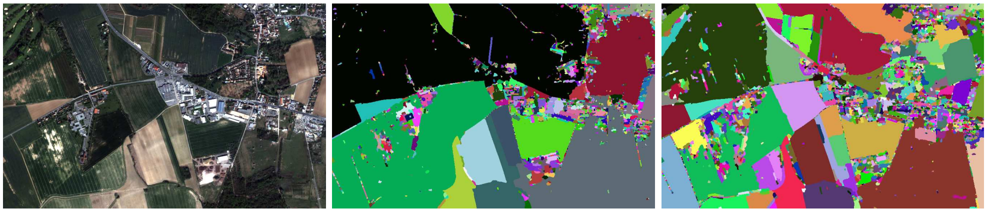


Figure 14.9: Segmented RGB image. At left is the original image. The image in the middle was generated with parameters: conductance = 2.0, iterations = 10, threshold = 0.0, level = 0.05, principal components = on. The image on the right was generated with parameters: conductance = 2.0, iterations = 10, threshold = 0.001, level = 0.15, principal components = off.

```
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 14.9 shows output from the example code. Note that a critical difference between the two segmentations is the mode of the gradient magnitude calculation.

A note on the computational complexity of the watershed algorithm is warranted. Most of the complexity of the ITK implementation lies in generating the hierarchy. Processing times for this stage are non-linear with respect to the number of catchment basins in the initial segmentation. This means that the amount of information contained in an image is more significant than the number of pixels in the image. A very large, but very flat input take less time to segment than a very small, but very detailed input.

14.3 Level Set Segmentation

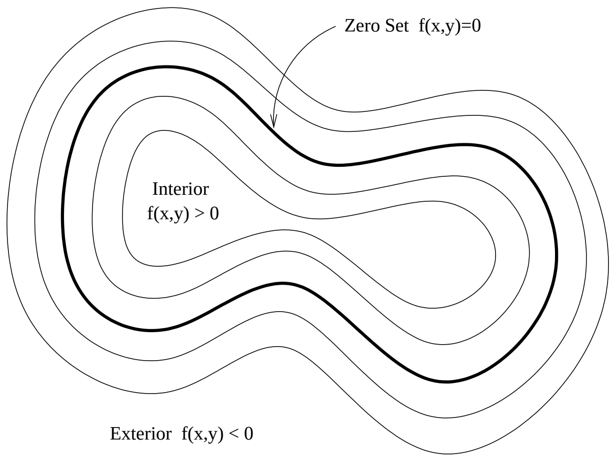


Figure 14.10: Concept of zero set in a level set.

The paradigm of the level set is that it is a numerical method for tracking the evolution of contours and surfaces. Instead of manipulating the contour directly, the contour is embedded as the zero level set of a higher dimensional function called the level-set function, $\psi(\mathbf{X},\mathbf{t})$. The level-set function is then evolved under the control of a differential equation. At any time, the evolving contour can be obtained by extracting the zero level-set $\Gamma((\mathbf{X}),\mathbf{t}) = \{\psi(\mathbf{X},\mathbf{t}) = \mathbf{0}\}$ from the output. The main advantages of using level sets is that arbitrarily complex shapes can be modeled and topological changes such as merging and splitting are handled implicitly.

Level sets can be used for image segmentation by using image-based features such as mean intensity, gradient and edges in the governing differential equation. In a typical approach, a contour is initialized by a user and is then evolved until it fits the form of an object 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 [80].

The following sections introduce practical examples of some of the level set segmentation methods available in ITK. The remainder of this section describes features common to all of these filters except the `itk::FastMarchingImageFilter`, which is derived from a different code framework. Understanding these features will aid in using the filters more effectively.

Each filter makes use of a generic level-set equation to compute the update to the solution ψ of the partial differential equation.

$$\frac{d}{dt}\psi = -\alpha\mathbf{A}(\mathbf{x})\cdot\nabla\psi - \beta P(\mathbf{x}) \mid \nabla\psi \mid + \gamma Z(\mathbf{x})\kappa \mid \nabla\psi \mid \tag{14.3}$$

where $\mathbf{A}$ is an advection term, P is a propagation (expansion) term, and Z is a spatial modifier term for the mean curvature κ. The scalar constants α, β, and γ weight the relative influence of each of the terms on the movement of the interface. A segmentation filter may use all of these terms in its calculations, or it may omit one or more terms. If a term is left out of the equation, then setting the corresponding scalar constant weighting will have no effect.

All of the level-set based segmentation filters *must* operate with floating point precision to pro-

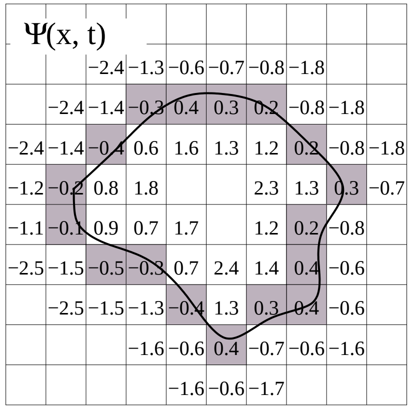


Figure 14.11: The implicit level set surface Γ is the black line superimposed over the image grid. The location of the surface is interpolated by the image pixel values. The grid pixels closest to the implicit surface are shown in gray.

duce valid results. The third, optional template parameter is the *numerical type* used for calculations and as the output image pixel type. The numerical type is `float` by default, but can be changed to `double` for extra precision. A user-defined, signed floating point type that defines all of the necessary arithmetic operators and has sufficient precision is also a valid choice. You should not use types such as `int` or `unsigned char` for the numerical parameter. If the input image pixel types do not match the numerical type, those inputs will be cast to an image of appropriate type when the filter is executed.

Most filters require two images as input, an initial model $\psi(\mathbf{X}, \mathbf{t} = \mathbf{0})$, and a *feature image*, which is either the image you wish to segment or some preprocessed version. You must specify the isovalue that represents the surface Γ in your initial model. The single image output of each filter is the function ψ at the final time step. It is important to note that the contour representing the surface Γ is the zero level-set of the output image, and not the isovalue you specified for the initial model. To represent Γ using the original isovalue, simply add that value back to the output.

The solution Γ is calculated to subpixel precision. The best discrete approximation of the surface is therefore the set of grid positions closest to the zero-crossings in the image, as shown in Figure 14.11. The `itk::ZeroCrossingImageFilter` operates by finding exactly those grid positions and can be used to extract the surface.

There are two important considerations when analyzing the processing time for any particular level-set segmentation task: the surface area of the evolving interface and the total distance that the surface must travel. Because the level-set equations are usually solved only at pixels near the surface (fast marching methods are an exception), the time taken at each iteration depends on the number of points on the surface. This means that as the surface grows, the solver will slow down proportionally. Because the surface must evolve slowly to prevent numerical instabilities

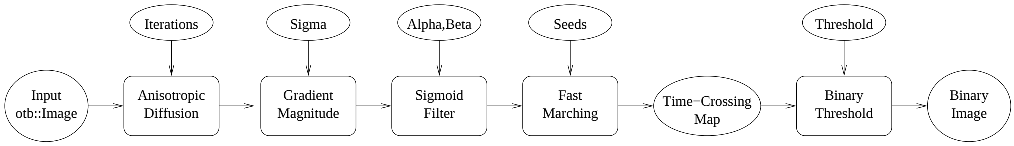


Figure 14.12: Collaboration diagram of the FastMarchingImageFilter applied to a segmentation task.

in the solution, the distance the surface must travel in the image dictates the total number of iterations required.

Some level-set techniques are relatively insensitive to initial conditions and are therefore suitable for region-growing segmentation. Other techniques, such as the `itk::LaplacianSegmentationLevelSetImageFilter`, can easily become “stuck” on image features close to their initialization and should be used only when a reasonable prior segmentation is available as the initialization. For best efficiency, your initial model of the surface should be the best guess possible for the solution.

14.3.1 Fast Marching Segmentation

The source code for this example can be found in the file
`Examples/Segmentation/FastMarchingImageFilter.cxx`.

When the differential equation governing the level set evolution has a very simple form, a fast evolution algorithm called fast marching can be used.

The following example illustrates the use of the `itk::FastMarchingImageFilter`. This filter implements a fast marching solution to a simple level set evolution problem. In this example, the speed term used in the differential equation is expected to be provided by the user in the form of an image. This image is typically computed as a function of the gradient magnitude. Several mappings are popular in the literature, for example, the negative exponential $exp(-x)$ and the reciprocal $1/(1+x)$. In the current example we decided to use a Sigmoid function since it offers a good deal of control parameters that can be customized to shape a nice speed image.

The mapping should be done in such a way that the propagation speed of the front will be very low close to high image gradients while it will move rather fast in low gradient areas. This arrangement will make the contour propagate until it reaches the edges of anatomical structures in the image and then slow down in front of those edges. The output of the FastMarchingImageFilter is a *time-crossing map* that indicates, for each pixel, how much time it would take for the front to arrive at the pixel location.

The application of a threshold in the output image is then equivalent to taking a snapshot of the contour at a particular time during its evolution. It is expected that the contour will take a longer time to cross over the edges of a particular 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 14.12 shows the major components involved in the application of the FastMarchingImageFilter to a segmentation task. It involves an initial stage of smoothing using the `itk::CurvatureAnisotropicDiffusionImageFilter`. The smoothed image is passed as the input to the `itk::GradientMagnitudeRecursiveGaussianImageFilter` and then to the `itk::SigmoidImageFilter`. Finally, the output of the FastMarchingImageFilter is passed to a `itk::BinaryThresholdImageFilter` in order to produce a binary mask representing the segmented object.

The code in the following example illustrates the typical setup of a pipeline for performing segmentation with fast marching. First, the input image is smoothed using an edge-preserving filter. Then the magnitude of its gradient is computed and passed to a sigmoid filter. The result of the sigmoid filter is the image potential that will be used to affect the speed term of the differential equation.

Let's start by including the following headers. First we include the header of the CurvatureAnisotropicDiffusionImageFilter that will be used for removing noise from the input image.

```
#include "itkCurvatureAnisotropicDiffusionImageFilter.h"
```

The headers of the GradientMagnitudeRecursiveGaussianImageFilter and SigmoidImageFilter are included below. Together, these two filters will produce the image potential for regulating the speed term in the differential equation describing the evolution of the level set.

```
#include "itkGradientMagnitudeRecursiveGaussianImageFilter.h"
#include "itkSigmoidImageFilter.h"
```

Of course, we will need the `otb::Image` class and the FastMarchingImageFilter class. Hence we include their headers.

```
#include "otbImage.h"
#include "itkFastMarchingImageFilter.h"
```

The time-crossing map resulting from the FastMarchingImageFilter will be thresholded using the BinaryThresholdImageFilter. We include its header here.

```
#include "itkBinaryThresholdImageFilter.h"
```

Reading and writing images will be done with the `otb::ImageFileReader` and `otb::ImageFileWriter`.

```
#include "otbImageFileReader.h"
#include "otbImageFileWriter.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 otb::Image< InternalPixelType, Dimension >  InternalImageType;
```

The output image, on the other hand, is declared to be binary.

```
typedef unsigned char OutputPixelType;
typedef otb::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.

```
thresholder->SetLowerThreshold(           0.0  );
thresholder->SetUpperThreshold( timeThreshold  );

thresholder->SetOutsideValue(  0  );
thresholder->SetInsideValue(  255 );
```

We instantiate reader and writer types in the following lines.

```
typedef  otb::ImageFileReader< InternalImageType > ReaderType;
typedef  otb::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.

```
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 14.12 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. However they may have to be adjusted depending on the amount of noise present in the input image.

```
smoothing->SetTimeStep( 0.125 );
smoothing->SetNumberOfIterations(  10 );
smoothing->SetConductanceParameter( 2.0 );
```

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.

```
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 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 structure to be segmented. Then, let's call $K2$ an average value of the gradient magnitude in the middle of the structure. These two values indicate the dynamic range that we want to map to the interval $[0:1]$ in the speed image. We want the sigmoid to map $K1$ to 0.0 and $K2$ to 1.0. Given that $K1$ is expected to be higher than $K2$ and we want to map those values to 0.0 and 1.0 respectively, we want to select a negative value for alpha so that the sigmoid function will also do an inverse intensity mapping. This mapping will produce a speed image such that the level set will march rapidly on the homogeneous region and will definitely stop on the contour. The suggested value for beta is $(K1+K2)/2$ while the suggested value for alpha is $(K2-K1)/6$, which must be a negative number. In our simple example the values are provided by the user from the command line arguments. The user can estimate these values by observing the gradient magnitude image.

```
sigmoid->SetAlpha( alpha );
sigmoid->SetBeta(  beta  );
```

The FastMarchingImageFilter requires the user to provide a seed point from which the contour will expand. The user can actually pass not only one seed point but a set of them. A good set of seed points increases the chances of segmenting a complex object without missing parts. The use of multiple seeds also helps to reduce the amount of time needed by the front to visit a whole object and hence reduces the risk of leaks on the edges of regions visited earlier. For example, when segmenting an elongated object, it is undesirable to place a single seed at one extreme of the object since the front will need a long time to propagate to the other end of the object. Placing several seeds along the axis of the object will probably be the best strategy to ensure that the entire object is captured early in the expansion of the front. One of the important properties of level sets is their natural ability to fuse several fronts implicitly without any extra bookkeeping. The use of multiple seeds takes good advantage of this property.

The seeds are passed stored in a container. The type of this container is defined as `NodeContainer` among the FastMarchingImageFilter traits.

```
typedef FastMarchingFilterType::NodeContainer          NodeContainer;
typedef FastMarchingFilterType::NodeType               NodeType;
NodeContainer::Pointer seeds = NodeContainer::New();
```

Nodes are created as stack variables and initialized with a value and an `itk::Index` position.

```
NodeType node;
const double seedValue = 0.0;

node.SetValue( seedValue );
node.SetIndex( seedPosition );
```

The list of nodes is initialized and then every node is inserted using the `InsertElement()`.

```
seeds->Initialize();
seeds->InsertElement( 0, node );
```

The set of seed nodes is now passed to the FastMarchingImageFilter with the method `SetTrialPoints()`.

```
fastMarching->SetTrialPoints(  seeds  );
```

The FastMarchingImageFilter requires the user to specify the size of the image to be produced as output. This is done using the `SetOutputSize()`. Note that the size is obtained here from the output image of the smoothing filter. The size of this image is valid only after the `Update()` methods of this filter has been called directly or indirectly.

```
fastMarching->SetOutputSize(
         reader->GetOutput()->GetBufferedRegion().GetSize() );
```

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
  {
```

Structure	Seed Index	σ	α	β	Threshold	Output Image from left
Road	$(91, 176)$	0.5	-0.5	3.0	100	First
Shadow	$(118, 100)$	1.0	-0.5	3.0	100	Second
Building	$(145, 21)$	0.5	-0.5	3.0	100	Third

Table 14.4: Parameters used for segmenting some structures shown in Figure 14.14 using the filter FastMarchingImageFilter. All of them used a stopping value of 100.

```
    writer->Update();
    }
  catch( itk::ExceptionObject & excep )
    {
    std::cerr << "Exception caught !" << std::endl;
    std::cerr << excep << std::endl;
    }
```

Now let's run this example using the input image `QB_Suburb.png` provided in the directory `Examples/Data`. We can easily segment structures by providing seeds in the appropriate locations. The following table presents the parameters used for some structures.

Figure 14.13 presents the intermediate outputs of the pipeline illustrated in Figure 14.12. 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.

The following classes provide similar functionality:

- `itk::ShapeDetectionLevelSetImageFilter`
- `itk::GeodesicActiveContourLevelSetImageFilter`
- `itk::ThresholdSegmentationLevelSetImageFilter`
- `itk::CannySegmentationLevelSetImageFilter`
- `itk::LaplacianSegmentationLevelSetImageFilter`

See the ITK Software Guide for examples of the use of these classes.

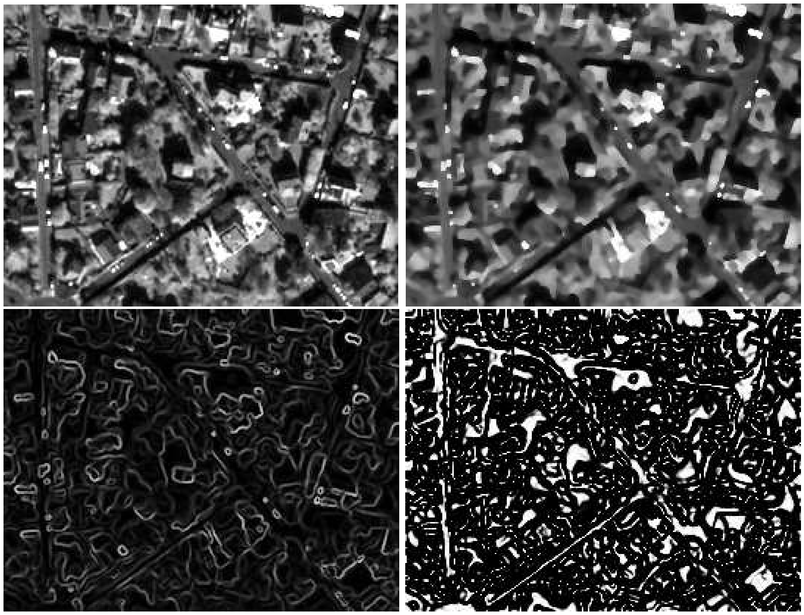


Figure 14.13: Images generated by the segmentation process based on the FastMarchingImageFilter. From left to right and top to bottom: input image to be segmented, image smoothed with an edge-preserving smoothing filter, gradient magnitude of the smoothed image, sigmoid of the gradient magnitude. This last image, the sigmoid, is used to compute the speed term for the front propagation

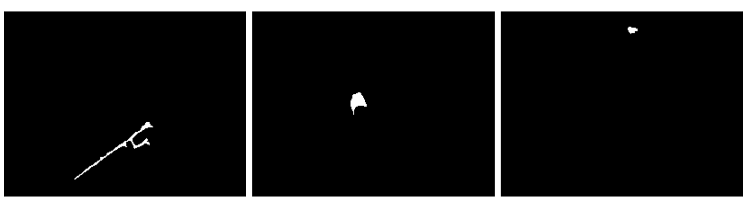


Figure 14.14: Images generated by the segmentation process based on the FastMarchingImageFilter. From left to right: segmentation of the road, shadow, building.

CHAPTER

FIFTEEN

Multi-scale Analysis

15.1 Introduction

In this chapter, the tools for multi-scale and multi-resoltuion processing (analysis, synthesis and fusion) will be presented. Most of the algorithms are based on pyramidal approaches. These approaches were first used for image compression and they are based on the fact that, once an image has been low-pass filtered it does not have details beyond the cut-off frequency of the low-pass filter any more. Therefore, the image can be subsampled – decimated – without any loss of information.

A pyramidal decomposition is thus performed applying the following 3 steps in an iterative way:

1. Low pas filter the image I_n in order to produce $F(I_n)$;
2. Compute the difference $D_n = I_n - F(I_n)$ which corresponds to the details at level n;
3. Subsample $F_(I_n)$ in order to obtain I_{n+1}.

The result is a series of decrasing resolution images I_k and a series of decreasing resolution details D_k.

15.2 Morphological Pyramid

If the smoothing filter used in the pyramidal analysis is a morphological filter, one cannot safely subsample the filtered image without loss of information. However, by keeping the details possibly lost in the down-sampling operation, such a decomposition can be used.

The Morphological Pyramid is an approach to such a decomposition. It's computation process is an iterative analysis involving smoothing by the morphological filter, computing the details lost in the smoothing, down-sampling the current image, and computing the details lost in the down-sampling.

The source code for this example can be found in the file
`Examples/MultiScale/MorphologicalPyramidAnalysisFilterExample.cxx`.

This example illustrates the use of the `otb::MorphologicalPyramidAnalyseFilter`.

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

```
#include "otbMorphologicalPyramidAnalysisFilter.h"
```

The mathematical morphology filters to be used have also to be included here.

```
#include "otbOpeningClosingMorphologicalFilter.h"
#include "itkBinaryBallStructuringElement.h"
```

As usual, we start by defining the types needed for the pixels, the images, the image reader and the image writer.

```
      const unsigned int Dimension = 2;
      typedef unsigned char InputPixelType;
      typedef unsigned char OutputPixelType;

      typedef otb::Image<InputPixelType,Dimension> InputImageType;
      typedef otb::Image<OutputPixelType,Dimension> OutputImageType;

      typedef otb::ImageFileReader<InputImageType> ReaderType;
      typedef otb::ImageFileWriter<OutputImageType> WriterType;
```

Now, we define the types needed for the morphological filters which will be used to build the morphological pyramid. The first thing to do is define the structuring element, which in our case, will be a `itk::BinaryBallStructuringElement` which is templated over the pixel type and the dimension of the image.

```
      typedef itk::BinaryBallStructuringElement<InputPixelType,
                                 Dimension> StructuringElementType;
```

We can now define the type of the filter to be used by the morphological pyramid. In this case, we choose to use an `otb::OpeningClosingMorphologicalFilter` which is just the concatenation of an opening and a closing. This filter is theplated over the input and output image types and the structurung element type that we just define above.

```
typedef otb::OpeningClosingMorphologicalFilter<InputImageType,
                           InputImageType,StructuringElementType>
                                 OpeningClosingFilterType;
```

We can finally define the type of the morpholoical pyramid filter. The filter is templated over the input and output mage types and the *lowpas* morphological filter to be used.

```
typedef otb::MorphologicalPyramidAnalysisFilter<InputImageType,
                        OutputImageType,OpeningClosingFilterType>
                                        PyramidFilterType;
```

Since the `otb::MorphologicalPyramidAnalyseFilter` generates a list of images as output, it is useful to have an iterator to access the images. This is done as follows :

```
typedef PyramidFilterType::OutputImageListType::Iterator
                                       ImageListIterator;
```

We can now instantiate the reader in order to access the input image which has to be analysed.

```
ReaderType::Pointer reader = ReaderType::New();
reader->SetFileName(inputFilename);
```

We instantiate the morphological pyramid analysis filter and set its parameters which are:

- the number of iterations or levels of the pyramid;
- the subsample scale or decimation factor between two successive pyramid levels.

After that, we plug the pipeline and run it by calling the `Update()` method.

```
PyramidFilterType::Pointer pyramid = PyramidFilterType::New();
pyramid->SetNumberOfLevels(numberOfLevels);
pyramid->SetDecimationRatio(decimationRatio);
pyramid->SetInput(reader->GetOutput());
pyramid->Update();
```

The morphological pyramid has 5 types of output:

- the analysed image at each level of the pyramid through the `GetOutput()` method;

- the brighter details extracted from the filtering operation through the `GetSupFilter()` method;
- the darker details extracted from the filtering operation through the `GetInfFilter()` method;
- the brighter details extracted from the resampling operation through the `GetSupDeci()` method;
- the darker details extracted from the resampling operation through the `GetInfDeci()` method; to decimation

Each one of these methods provides a list of images (one for each level of analysis), so we can iterate through the image lists by using iterators.

```
ImageListIterator itAnalyse = pyramid->GetOutput()->Begin();
ImageListIterator itSupFilter = pyramid->GetSupFilter()->Begin();
ImageListIterator itInfFilter = pyramid->GetInfFilter()->Begin();
ImageListIterator itInfDeci = pyramid->GetSupDeci()->Begin();
ImageListIterator itSupDeci =  pyramid->GetInfDeci()->Begin();
```

We can now instantiate a writer and use it to write all the images to files.

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

      int i=1;

      // Writing the results images
      std::cout<<(itAnalyse!=(pyramid->GetOutput()->End()))<<std::endl;
      while(itAnalyse!=pyramid->GetOutput()->End())
{
  writer->SetInput(itAnalyse.Get());
  writer->SetFileName(argv[0*4+i+1]);
  writer->Update();

  writer->SetInput(itSupFilter.Get());
  writer->SetFileName(argv[1*4+i+1]);
  writer->Update();

  writer->SetInput(itInfFilter.Get());
  writer->SetFileName(argv[2*4+i+1]);
  writer->Update();
```

Figure 15.1: Test image for the morphological pyramid.

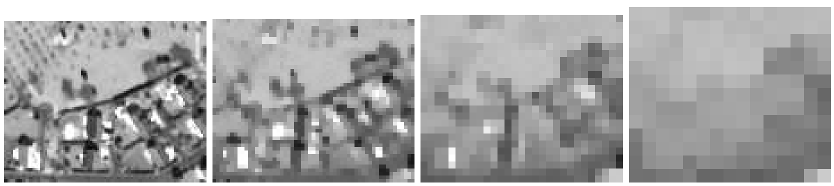


Figure 15.2: Result of the analysis for 4 levels of the pyramid.

```
  writer->SetInput(itInfDeci.Get());
  writer->SetFileName(argv[3*4+i+1]);
  writer->Update();

  writer->SetInput(itSupDeci.Get());
  writer->SetFileName(argv[4*4+i+1]);
  writer->Update();

  ++itAnalyse;
  ++itSupFilter;
  ++itInfFilter;
  ++itInfDeci;
  ++itSupDeci;
  ++i;
}
```

Figure 15.1 shows the test image to be processed by the morphological pyramid.

Figure 15.2 shows the 4 levels of analysis of the image.

Figure 15.3 shows the 4 levels of bright details.

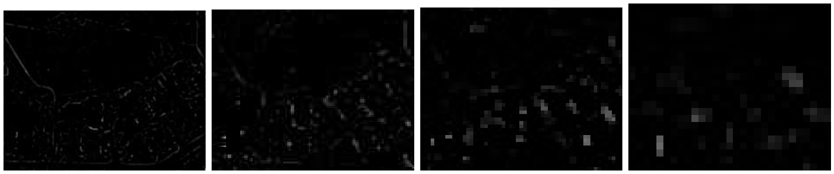


Figure 15.3: Bright details for 4 levels of the pyramid.

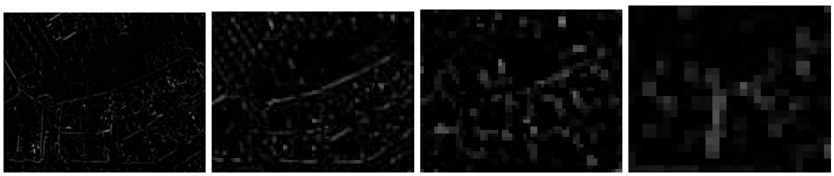


Figure 15.4: Dark details for 4 levels of the pyramid.

Figure 15.4 shows the 4 levels of dark details.

Figure 15.5 shows the 4 levels of bright decimation details.

Figure 15.6 shows the 4 levels of dark decimation details.

The source code for this example can be found in the file
`Examples/MultiScale/MorphologicalPyramidSynthesisFilterExample.cxx`.

This example illustrates the use of the `otb::MorphologicalPyramidSynthesisFilter`.

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

```
#include "otbMorphologicalPyramidSynthesisFilter.h"
```

The mathematical morphology filters to be used have also to be included here, as well as the `otb::MorphologicalPyramidAnalyseFilter` in order to perform the analysis step.

```
#include "otbMorphologicalPyramidAnalysisFilter.h"
#include "otbOpeningClosingMorphologicalFilter.h"
#include "itkBinaryBallStructuringElement.h"
```

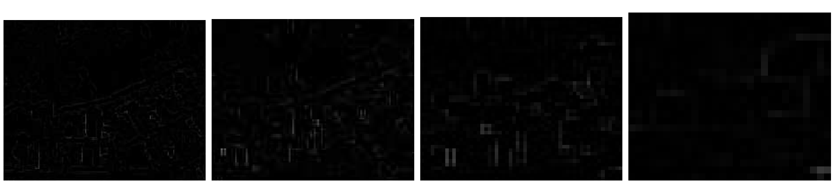


Figure 15.5: Bright decimation details for 4 levels of the pyramid.

Figure 15.6: Dark decimation details for 4 levels of the pyramid.

As usual, we start by defining the types needed for the pixels, the images, the image reader and the image writer.

```
const unsigned int Dimension = 2;
typedef unsigned char InputPixelType;
typedef unsigned char OutputPixelType;

typedef otb::Image<InputPixelType,Dimension> InputImageType;
typedef otb::Image<OutputPixelType,Dimension> OutputImageType;

typedef otb::ImageFileReader<InputImageType> ReaderType;
typedef otb::ImageFileWriter<OutputImageType> WriterType;
```

Now, we define the types needed for the morphological filters which will be used to build the morphological pyramid. The first thing to do is define the structuring element, which in our case, will be a `itk::BinaryBallStructuringElement` which is templated over the pixel type and the dimension of the image.

```
typedef itk::BinaryBallStructuringElement<InputPixelType,Dimension>
                                          StructuringElementType;
```

We can now define the type of the filter to be used by the morphological pyramid. In this case, we choose to use an `otb::OpeningClosingMorphologicalFilter` which is just the concatenation of an opening and a closing. This filter is theplated over the input and output image types and the structurung element type that we just define above.

```
typedef otb::OpeningClosingMorphologicalFilter<InputImageType,
                             InputImageType,StructuringElementType>
                                   OpeningClosingFilterType;
```

We can now define the type of the morpholoical pyramid filter. The filter is templated over the input and output mage types and the *lowpas* morphological filter to be used.

```
typedef otb::MorphologicalPyramidAnalysisFilter<InputImageType,
                          OutputImageType,OpeningClosingFilterType>
                                    PyramidAnalysisFilterType;
```

We can finally define the type of the morpholoical pyramid synthesis filter. The filter is templated over the input and output mage types.

```
typedef otb::MorphologicalPyramidSynthesisFilter<InputImageType,
                                                  OutputImageType>
                                PyramidSynthesisFilterType;
```

We can now instantiate the reader in order to access the input image which has to be analysed.

```
ReaderType::Pointer reader = ReaderType::New();
reader->SetFileName(inputFilename);
```

We instantiate the morphological pyramid analysis filter and set its parameters which are:

- the number of iterations or levels of the pyramid;
- the subsample scale or decimation factor between two successive pyramid levels.

After that, we plug the pipeline and run it by calling the `Update()` method.

```
PyramidAnalysisFilterType::Pointer pyramidAnalysis =
                  PyramidAnalysisFilterType::New();
pyramidAnalysis->SetNumberOfLevels(numberOfLevels);
pyramidAnalysis->SetDecimationRatio(decimationRatio);
pyramidAnalysis->SetInput(reader->GetOutput());
pyramidAnalysis->Update();
```

Once the analysis step is finished we can proceed to the synthesis of the image from its different levels of decomposition. The morphological pyramid has 5 types of output:

- the Analysisd image at each level of the pyramid through the `GetOutput()` method;
- the brighter details extracted from the filtering operation through the `GetSupFilter()` method;
- the darker details extracted from the filtering operation through the `GetInfFilter()` method;

- the brighter details extracted from the resampling operation through the `GetSupDeci()` method;
- the darker details extracted from the resampling operation through the `GetInfDeci()` method; to decimation

This outputs can be used as input of the synthesis filter by using the appropriate methods.

```
PyramidSynthesisFilterType::Pointer pyramidSynthesis = PyramidSynthesisFilterType::New()
pyramidSynthesis->SetInput(pyramidAnalysis->GetOutput()->Back());
pyramidSynthesis->SetSupFilter(pyramidAnalysis->GetSupFilter());
pyramidSynthesis->SetSupDeci(pyramidAnalysis->GetSupDeci());
pyramidSynthesis->SetInfFilter(pyramidAnalysis->GetInfFilter());
pyramidSynthesis->SetInfDeci(pyramidAnalysis->GetInfDeci());
```

After that, we plug the pipeline and run it by calling the `Update()` method.

```
pyramidSynthesis->Update();
```

We finally instatiate a the writer in order to save the result image to a file.

```
WriterType::Pointer writer = WriterType::New();
writer->SetFileName(outputFilename);
writer->SetInput(pyramidSynthesis->GetOutput()->Back());
writer->Update();
```

Since the synthesis operation is applied on the result of the analysis, the input and the output images should be identical. This is the case as shown in figure 15.7.

Of course, in a real application, a specific processing will be applied after the analysis and before the synthesis to, for instance, denoise the image by removing pixels at the finer scales, etc.

15.2.1 Morphological Pyramid Exploitation

One of the possible uses of the morphological pyramid is the segmentation of objects – regions – of a particular scale.

Figure 15.7: Result of the morphological pyramid analysis and synthesis. Left: original image. Right: result of applying the analysis and the synthesis steps.

The source code for this example can be found in the file
`Examples/MultiScale/MorphologicalPyramidSegmenterExample.cxx`.

This example illustrates the use of the `otb::MorphologicalPyramid::Segmenter`. This class performs the segmentation of a detail image extracted from a morphological pyramid analysis. The Segmentation is perfomed using the `itk::ConnectedThresholdImageFilter`. The seeds are extracted from the image using the `otb::ImageToPointSetFilter`. The thresolds are set by using quantiles computed with the HistogramGenerator.

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

```
#include "otbMorphologicalPyramidSegmenter.h"
```

As usual, we start by defining the types needed for the pixels, the images, the image reader and the image writer. Note that, for this example, an RGB image will be created to store the results of the segmentation.

```
const unsigned int Dimension = 2;
typedef double InputPixelType;
typedef unsigned short LabelPixelType;
typedef itk::RGBPixel<unsigned char>   RGBPixelType;


typedef otb::Image<InputPixelType,Dimension> InputImageType;
typedef otb::Image<LabelPixelType,Dimension> LabelImageType;
typedef otb::Image<RGBPixelType, 2>    RGBImageType;

typedef otb::ImageFileReader<InputImageType> ReaderType;
typedef otb::ImageFileWriter<RGBImageType> WriterType;
```

We define now the segmenter. Please pay attention to the fact that this class belongs to the `morphologicalPyramid` namespace.

```
typedef otb::MorphologicalPyramid::Segmenter<InputImageType,
                                              LabelImageType>
                                        SegmenterType;
```

We instantiate the readers which will give us access to the image of details produced by the morphological pyramid analysis and the original image (before analysis) which is used in order to produce segmented regions which are sharper than what would have been obtained with the detail image only.

```
ReaderType::Pointer reader = ReaderType::New();
reader->SetFileName(inputFilename);
ReaderType::Pointer reader2 = ReaderType::New();
reader2->SetFileName(originalFilename);
```

We instantiate the segmenter and set its parameters as follows. We plug the output of the readers for the details image and the original image; we set the boolean variable which controls wether the segmented details are bright or dark; we set the quantile used to threshold the details image in order to obtain the seed points for the segmentation; we set the quantile for setting the threshold for the region growing segmentation; and finally, we set the minimum size for a segmented region to be kept in the final result.

```
SegmenterType::Pointer segmenter = SegmenterType::New();
segmenter->SetDetailsImage(reader->GetOutput());
segmenter->SetOriginalImage(reader2->GetOutput());
segmenter->SetSegmentDarkDetailsBool(segmentDark);
segmenter->SetSeedsQuantile(seedsQuantile);
segmenter->SetConnectedThresholdQuantile(segmentationQuantile);
segmenter->SetMinimumObjectSize(minObjectSize);
```

The output of the segmenter is an image of integer labels, where a label denotes membership of a pixel in a particular segmented region. This value is usually coded using 16 bits. 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 for that converts scalar images to RGB images.

Figure 15.8: Morphological pyramid segmentation. From left to right: original image, image of bright details and result of the sementation.

```
typedef itk::Functor::ScalarToRGBPixelFunctor<LabelPixelType>
                                                    ColorMapFunctorType;
typedef itk::UnaryFunctorImageFilter<LabelImageType,
                    RGBImageType, ColorMapFunctorType> ColorMapFilterType;
ColorMapFilterType::Pointer colormapper = ColorMapFilterType::New();
```

We can now plug the final segment of the pipeline by using the color mapper and the image file writer.

```
colormapper->SetInput(segmenter->GetOutput());
WriterType::Pointer writer = WriterType::New();
writer->SetInput(colormapper->GetOutput());
writer->SetFileName(outputFilename1);
writer->Update();
```

Figure 15.8 shows the results of the segmentation of the image of bright details obtained with the morphological pyramid analysis.

This same approach can be applied to all the levels of the morphological pyramid analysis.

The source code for this example can be found in the file
`Examples/MultiScale/MorphologicalPyramidSegmentationExample.cxx`.

This example illustrates the use of the `otb::MorphologicalSegmentationFilter`. This filter performs a segmentation of the details `supFilter` and `infFilter` extracted with the morphological pyramid. The segmentation algorithm used is based on seeds extraction using the `otb::ImageToPointSetFilter`, followed by a connected threshold segmentation using the `itk::ConnectedThresholdImageFilter`. The threshold for seeds extraction and segmentation are computed using quantiles. A pre processing step is applied by multiplying the full resolution brighter details (resp. darker details) with the original image (resp. the inverted original image). This perfoms an enhancement of the regions contour precision. The details from the pyramid are set via the `SetBrighterDetails()` and `SetDarkerDetails()` methods. The brighter and darker details depend on the filter used in the pyramid analysis. If the `otb::OpeningClosingMorphologicalFilter` filter is used, then the brighter details are those from the `supFilter` image list, whereas if the `otb::ClosingOpeningMorphologicalFilter` filter is used, the brighter details are those from the `infFilter` list. The output of the segmentation filter is a single segmentation images list, containing first the brighter details segmentation from higher scale to lower, and then the darker details in the same order. The attention of the user is drawn to the fact that since the label filter used internally will deal with a large number of labels, the `OutputPixelType` is required to be sufficiently precise. Unsigned short or Unsigned long would be a good choice, unless the user has a very good reason to think that a less precise type will be sufficient. The first step to use this filter is to include its header file.

```
#include "otbMorphologicalPyramidSegmentationFilter.h"
```

The mathematical morphology filters to be used have also to be included here, as well as the morphological pyramid analysis filter.

```
#include "otbOpeningClosingMorphologicalFilter.h"
#include "itkBinaryBallStructuringElement.h"
#include "otbMorphologicalPyramidAnalysisFilter.h"
```

As usual, we start by defining the types for the pixels, the images, the reader and the writer. We also define the types needed for the morphological pyramid analysis.

```
const unsigned int Dimension = 2;
typedef unsigned char InputPixelType;
typedef unsigned short OutputPixelType;

typedef otb::Image<InputPixelType,Dimension> InputImageType;
typedef otb::Image<OutputPixelType,Dimension> OutputImageType;

typedef otb::ImageFileReader<InputImageType> ReaderType;
typedef otb::ImageFileWriter<OutputImageType> WriterType;

typedef itk::BinaryBallStructuringElement<InputPixelType,Dimension>
                                     StructuringElementType;
typedef otb::OpeningClosingMorphologicalFilter<InputImageType,
                              InputImageType,StructuringElementType>
                                    OpeningClosingFilterType;
```

```
typedef otb::MorphologicalPyramidAnalysisFilter<InputImageType,
                             InputImageType,OpeningClosingFilterType>
                                             PyramidFilterType;
```

We can now define the type for the `otb::MorphologicalPyramidSegmentationFilter` which is templated over the input and output image types.

```
typedef otb::MorphologicalPyramidSegmentationFilter<InputImageType,
                                                    OutputImageType>
                                      SegmentationFilterType;
```

Since the output of the segmentation filter is a list of images, we define an iterator type which will be used to access the segmented images.

```
typedef SegmentationFilterType::OutputImageListIteratorType
                                      OutputListIteratorType;
```

The following code snippet shows how to read the input image and perform the morphological pyramid analysis.

```
ReaderType::Pointer reader = ReaderType::New();
reader->SetFileName(inputFilename);

PyramidFilterType::Pointer pyramid = PyramidFilterType::New();
pyramid->SetNumberOfLevels(numberOfLevels);
pyramid->SetDecimationRatio(decimationRatio);
pyramid->SetInput(reader->GetOutput());
```

We can now instantiate the segmentation filter and set its parameters. As one can see, the `SetReferenceImage()` is used to pass the original image in order to obtain sharp region boundaries. Using the `SetBrighterDetails()` and `SetDarkerDetails()` the output of the analysis is passed to the filter. Finally, the parameters for the segmentation are set by using the `SetSeedsQuantile()`, `SetConnectedThresholdQuantile()` and `SetMinimumObjectSize()` methods.

```
SegmentationFilterType::Pointer segmentation = SegmentationFilterType::New();
segmentation->SetReferenceImage(reader->GetOutput());
segmentation->SetBrighterDetails(pyramid->GetSupFilter());
```

```
      segmentation->SetDarkerDetails(pyramid->GetInfFilter());
      segmentation->SetSeedsQuantile(seedsQuantile);
      segmentation->SetConnectedThresholdQuantile(segmentationQuantile);
      segmentation->SetMinimumObjectSize(minObjectSize);
```

The pipeline is executed bu calling the `Update()` method.

```
      segmentation->Update();
```

Finally, we get an iterator to the list generated as output for the segmentation and we use it to iterate through the list and write the images to files.

```
      OutputListIteratorType it = segmentation->GetOutput()->Begin();
      WriterType::Pointer writer;
      int index = 1;
      std::stringstream oss;
      while(it!=segmentation->GetOutput()->End())
{
  oss<<outputFilenamePrefix<<index<<"."<<outputFilenameSuffix;
  writer = WriterType::New();
  writer->SetInput(it.Get());
  writer->SetFileName(oss.str().c_str());
  writer->Update();
  std::cout<<oss.str()<<" file written."<<std::endl;
  oss.str("");
  ++index;
  ++it;
}
```

The user will pay attention to the fact that the list contains first the brighter details segmentation from higher scale to lower, and then the darker details in the same order.

CHAPTER

SIXTEEN

Change Detection

16.1 Introduction

Change detection techniques try to detect and locate areas which have changed between two or more observations of the same scene. These changes can be of different types, with different origins and of different temporal length. This allows to distinguish different kinds of applications:

- *land use monitoring*, which corresponds to the characterization of the evolution of the vegetation, or its seasonal changes;
- *natural resources management*, which corresponds mainly to the characterisation of the evolution of the urban areas, the evolution of the deforestation, etc.
- *damage mapping*, which corresponds to the location of damages caused by natural or industrial disasters.

From the point of view of the observed phenomena, one can distinguish 2 types of changes whose nature is rather different: the abrupt changes and the progressive changes, which can eventually be periodic. From the data point of view, one can have:

- Image pairs before and after the event. The applications are mainly the abrupt changes.
- Multi-temporal image series on which 2 types on changes may appear:
  - The slow changes like for instance the erosion, vegetation evolution, etc. The knowledge of the studied phenomena and of their consequences on the geometrical and radiometrical evolution at the different dates is a very important information for this kind of analysis.
  - The abrupt changes may pose different kinds of problems depending on whether the date of the change is known in the image series or not. The detection of areas

affected by a change occurred at a known date may exploit this a priori information in order to split the image series into two sub-series (before an after) and use the temporal redundancy in order to improve the detection results. On the other hand, when the date of the change is not known, the problem has a higher difficulty.

From this classification of the different types of problems, one can infer 4 cases for which one can look for algorithms as a function of the available data:

1. Abrupt changes in an image pair. This is no doubt the field for which more work has been done. One can find tools at the 3 classical levels of image processing: data level (differences, ratios, with or without pre-filtering, etc.), feature level (edges, targets, etc.), and interpretation level (post-classification comparison).
2. Abrupt changes within an image series and a known date. One can rely on bi-date techniques, either by fusing the images into 2 stacks (before and after), or by fusing the results obtained by different image couples (one after and one before the event). One can also use specific discontinuity detection techniques to be applied in the temporal axis.
3. Abrupt changes within an image series and an unknown date. This case can be seen either as a generalization of the preceding one (testing the N-1 positions for N dates) or as a particular case of the following one.
4. Progressive changes within an image series. One can work in two steps:
   (a) detect the change areas using stability criteria in the temporal areas;
   (b) identify the changes using prior information about the type of changes of interest.

16.1.1 Surface-based approaches

In this section we discuss about the damage assessment techniques which can be applied when only two images (before/after) are available.

As it has been shown in recent review works [18, 60, 73, 74], a relatively high number of methods exist, but most of them have been developed for optical and infrared sensors. Only a few recent works on change detection with radar images exist [82, 9, 66, 44, 24, 6, 46]. However, the intrinsic limits of passive sensors, mainly related to their dependence on meteorological and illumination conditions, impose severe constraints for operational applications. The principal difficulties related to change detection are of four types:

1. In the case of radar images, the speckle noise makes the image exploitation difficult.
2. The geometric configuration of the image acquisition can produce images which are difficult to compare.

3. Also, the temporal gap between the two acquisitions an thus the sensor aging and the inter-calibration are sources of variability which are difficult to deal with.
4. Finally, the normal evolution of the observed scenes must not be confused with the changes of interest.

The problem of detecting abrupt changes between a pair of images is the following: Let I_1, I_2 be two images acquired at different dates t_1, t_2; we aim at producing a thematic map which shows the areas where changes have taken place.

Three main categories of methods exist:

- Strategy 1: Post Classification Comparison

  The principle of this approach [21] is two obtain two land-use maps independently for each date and comparing them.

- Strategy 2: Joint classification

  This method consists in producing the change map directly from a joint classification of both images.

- Strategy 3: Simple detectors

  The last approach consists in producing an image of change likelihood (by differences, ratios or any other approach) and thresholding it in order to produce the change map.

Because of its simplicity and its low computation overhead, the third strategy is the one which has been chosen for the processing presented here.

16.2 Change Detection Framework

The source code for this example can be found in the file
`Examples/ChangeDetection/ChangeDetectionFrameworkExample.cxx`.

This example illustrates the Change Detector framework implemented in OTB. This framework uses the generic programming approach. All change detection filters are `otb::BinaryFunctorNeighborhoodImageFilter`s, that is, they are filters taking two images as input and providing one image as output. The change detection computation itself is performed on a the neighborhood of each pixel of the input images.

The first step required to build a change detection filter is to include the header of the parent class.

```
#include "otbBinaryFunctorNeighborhoodImageFilter.h"
```

The change detection operation itself is one of the templates of the change detection filters and takes the form of a function, that is, something accepting the syntax `foo()`. This can be implemented using classical C/C++ functions, but it is preferable to implement it using C++ functors. These are classical C++ classes which overload the `()` operator. This allows to use them with the same syntax as C/C++ functions.

Since change detectors operate on neighborhoods, the functor call will take 2 arguments which are `itk::ConstNeighborhoodIterator`s.

The change detector functor is templated over the types of the input iterators and the output result type. The core of the change detection is implemented in the `operator()` section.

```
template< class TInput1, class TInput2, class TOutput>
class MyChangeDetector
{
public:
  // The constructor and destructor.
  MyChangeDetector() {};
  ~MyChangeDetector() {};
  // Change detection operation
  inline TOutput operator()( const TInput1 & itA,
                             const TInput2 & itB)
  {

    TOutput result = 0.0;

    for(unsigned long pos = 0; pos< itA.Size(); ++pos)
      {

      result += static_cast<TOutput>(itA.GetPixel(pos)-itB.GetPixel(pos));


      }
    return static_cast<TOutput>( result/itA.Size() );
  }
};
```

The interest of using functors is that complex operations can be performed using internal `protected` class methods and that class variables can be used to store information so different pixel locations can access to results of previous computations.

The next step is the definition of the change detector filter. As stated above, this filter will inherit from `otb::BinaryFunctorNeighborhoodImageFilter` which is templated over the 2 input image types, the output image type and the functor used to perform the change detection operation.

Inside the class only a few `typedef`s and the constructors and destructors have to be declared.

```
template <class TInputImage1, class TInputImage2, class TOutputImage>
```

```
class ITK_EXPORT MyChangeDetectorImageFilter :
    public otb::BinaryFunctorNeighborhoodImageFilter<
            TInputImage1,TInputImage2,TOutputImage,
            MyChangeDetector<
                   typename itk::ConstNeighborhoodIterator<TInputImage1>,
                   typename itk::ConstNeighborhoodIterator<TInputImage2>,
   typename TOutputImage::PixelType>   >
{
public:
  /** Standard class typedefs. */
  typedef MyChangeDetectorImageFilter  Self;
  typedef typename otb::BinaryFunctorNeighborhoodImageFilter<
      TInputImage1,TInputImage2,TOutputImage,
          MyChangeDetector<
               typename itk::ConstNeighborhoodIterator<TInputImage1>,
               typename itk::ConstNeighborhoodIterator<TInputImage2>,
               typename TOutputImage::PixelType>
  >  Superclass;
  typedef itk::SmartPointer<Self>   Pointer;
  typedef itk::SmartPointer<const Self>  ConstPointer;

  /** Method for creation through the object factory. */
  itkNewMacro(Self);

protected:
  MyChangeDetectorImageFilter() {}
  virtual ~MyChangeDetectorImageFilter() {}

private:
  MyChangeDetectorImageFilter(const Self&); //purposely not implemented
  void operator=(const Self&); //purposely not implemented

};
```

Pay attention to the fact that no `.txx` file is needed, since filtering operation is implemented in the `otb::BinaryFunctorNeighborhoodImageFilter` class. So all the algorithmics part is inside the functor.

We can now write a program using the change detector.

As usual, we start by defining the image types. The internal computations will be performed with floating point precision, while the output image will be stored using one byte per pixel.

```
  typedef float InternalPixelType;
  typedef unsigned char OutputPixelType;
  typedef otb::Image<InternalPixelType, Dimension>  InputImageType1;
  typedef otb::Image<InternalPixelType, Dimension>  InputImageType2;
```

```
typedef otb::Image<InternalPixelType, Dimension>  ChangeImageType;
typedef otb::Image<OutputPixelType, Dimension>    OutputImageType;
```

We declare the readers, the writer, but also the `itk::RescaleIntensityImageFilter` which will be used to rescale the result before writing it to a file.

```
typedef otb::ImageFileReader< InputImageType1 >  ReaderType1;
typedef otb::ImageFileReader< InputImageType2 >  ReaderType2;
typedef otb::ImageFileWriter< OutputImageType >  WriterType;
typedef itk::RescaleIntensityImageFilter< ChangeImageType,
                                          OutputImageType > RescalerType;
```

The next step is declaring the filter for the change detection.

```
typedef MyChangeDetectorImageFilter<
                              InputImageType1,
                              InputImageType2,
                              ChangeImageType  >       FilterType;
```

We connect the pipeline.

```
reader1->SetFileName( inputFilename1  );
reader2->SetFileName( inputFilename2  );
writer->SetFileName( outputFilename );
rescaler->SetOutputMinimum( itk::NumericTraits< OutputPixelType >::min());
rescaler->SetOutputMaximum( itk::NumericTraits< OutputPixelType >::max());

filter->SetInput1( reader1->GetOutput() );
filter->SetInput2( reader2->GetOutput() );
filter->SetRadius( atoi(argv[3]) );

rescaler->SetInput( filter->GetOutput() );
writer->SetInput( rescaler->GetOutput() );
```

And that is all.

16.3 Simple Detectors

16.3.1 Mean Difference

The simplest change detector is based on the pixel-wise differencing of image values:

$$I_D(i,j) = I_2(i,j) - I_1(i,j). \tag{16.1}$$

Figure 16.1: Images used for the change detection. Left: Before the flood. Right: during the flood.

In order to make the algorithm robust to noise, one actually uses local means instead of pixel values.

The source code for this example can be found in the file
`Examples/ChangeDetection/DiffChDet.cxx`.

This example illustrates the class `otb::MeanDifferenceImageFilter` for detecting changes between pairs of images. This filter computes the mean intensity in the neighborhood of each pixel of the pair of images to be compared and uses the difference of means as a change indicator. This example will use the images shown in figure 16.1. These correspond to the near infrared band of two Spot acquisitions before and during a flood.

We start by including the corresponding header file.

```
#include "otbMeanDifferenceImageFilter.h"
```

We start by declaring the types for the two input images, the change image and the image to be stored in a file for visualization.

```
  typedef float InternalPixelType;
  typedef unsigned char OutputPixelType;
  typedef otb::Image<InternalPixelType, Dimension>  InputImageType1;
  typedef otb::Image<InternalPixelType, Dimension>  InputImageType2;
  typedef otb::Image<InternalPixelType, Dimension>  ChangeImageType;
  typedef otb::Image<OutputPixelType, Dimension>  OutputImageType;
```

We can now declare the types for the readers and the writer.

```
typedef otb::ImageFileReader< InputImageType1 >  ReaderType1;
typedef otb::ImageFileReader< InputImageType2 >  ReaderType2;
typedef otb::ImageFileWriter< OutputImageType >  WriterType;
```

The change detector will give positive and negative values depending on the sign of the difference. We are usually interested only in the asbolute value of the difference. For this purpose, we will use the `itk::AbsImageFilter`. Also, before saving the image to a file in, for instance, PNG format, we will rescale the results of the change detection in order to use all the output pixel type range of values.

```
typedef itk::AbsImageFilter< ChangeImageType,
                                          ChangeImageType > AbsType;
typedef itk::RescaleIntensityImageFilter< ChangeImageType,
                                          OutputImageType > RescalerType;
```

The `otb::MeanDifferenceImageFilter` is templated over the types of the two input images and the type of the generated change image.

```
typedef otb::MeanDifferenceImageFilter<
                                InputImageType1,
                                InputImageType2,
                                ChangeImageType  >       FilterType;
```

The different elements of the pipeline can now be instantiated.

```
ReaderType1::Pointer reader1 = ReaderType1::New();
ReaderType2::Pointer reader2 = ReaderType2::New();
WriterType::Pointer writer = WriterType::New();
FilterType::Pointer   filter = FilterType::New();
AbsType::Pointer absFilter = AbsType::New();
RescalerType::Pointer rescaler = RescalerType::New();
```

We set the parameters of the different elements of the pipeline.

```
reader1->SetFileName( inputFilename1  );
reader2->SetFileName( inputFilename2  );
writer->SetFileName( outputFilename );
rescaler->SetOutputMinimum( itk::NumericTraits< OutputPixelType >::min());
rescaler->SetOutputMaximum( itk::NumericTraits< OutputPixelType >::max());
```

The only parameter for this change detector is the radius of the window used for computing the mean of the intensities.

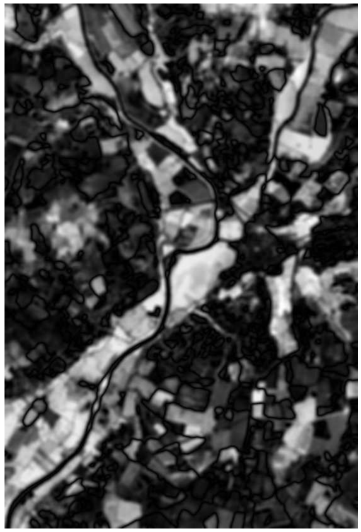


Figure 16.2: Result of the mean difference change detector

```
filter->SetRadius( atoi(argv[4]) );
```

We build the pipeline by plugging all the elements together.

```
filter->SetInput1( reader1->GetOutput() );
filter->SetInput2( reader2->GetOutput() );
absFilter->SetInput( filter->GetOutput() );
rescaler->SetInput( absFilter->GetOutput() );
writer->SetInput( rescaler->GetOutput() );
```

Since the processing time of large images can be long, it is interesting to monitor the evolution of the computation. In order to do so, the change detectors can use the command/observer design pattern. This is easily done by attaching an observer to the filter.

```
typedef otb::CommandProgressUpdate<FilterType> CommandType;

CommandType::Pointer observer = CommandType::New();
filter->AddObserver(itk::ProgressEvent(), observer);
```

Figure 16.2 shows the result of the change detection by difference of local means.

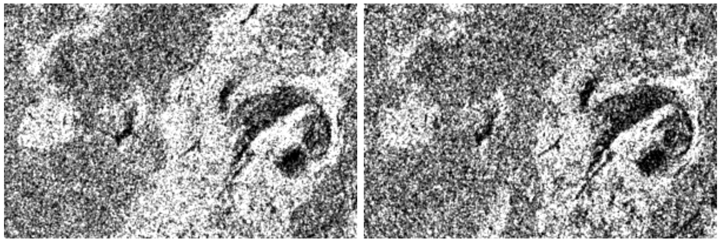


Figure 16.3: Images used for the change detection. Left: Before the eruption. Right: after the eruption.

16.3.2 Ratio Of Means

This detector is similar to the previous one except that it uses a ratio instead of the difference:

$$I_R(i,j) = \frac{I_2(i,j)}{I_1(i,j)}. \tag{16.2}$$

The use of the ratio makes this detector robust to multiplicative noise which is a good model for the speckle phenomenon which is present in radar images.

In order to have a bounded and normalized detector the following expression is actually used:

$$I_R(i,j) = 1 - min\left(\frac{I_2(i,j)}{I_1(i,j)}, \frac{I_1(i,j)}{I_2(i,j)}\right). \tag{16.3}$$

The source code for this example can be found in the file
`Examples/ChangeDetection/RatioChDet.cxx`.

This example illustrates the class `otb::MeanRatioImageFilter` for detecting changes between pairs of images. This filter computes the mean intensity in the neighborhood of each pixel of the pair of images to be compared and uses the ratio of means as a change indicator. This change indicator is then normalized between 0 and 1 by using the classical

$$r = 1 - min\{\frac{\mu_A}{\mu_B}, \frac{\mu_B}{\mu_A}\}, \tag{16.4}$$

where μ_A and μ_B are the local means. This example will use the images shown in figure 16.3. These correspond to 2 Radarsat fine mode acquisitions before and after a lava flow resulting from a volcanic eruption.

We start by including the corresponding header file.

```
#include "otbMeanRatioImageFilter.h"
```

We start by declaring the types for the two input images, the change image and the image to be stored in a file for visualization.

```
typedef float InternalPixelType;
typedef unsigned char OutputPixelType;
typedef otb::Image<InternalPixelType, Dimension>  InputImageType1;
typedef otb::Image<InternalPixelType, Dimension>  InputImageType2;
typedef otb::Image<InternalPixelType, Dimension>  ChangeImageType;
typedef otb::Image<OutputPixelType, Dimension>  OutputImageType;
```

We can now declare the types for the readers. Since the images can be vey large, we will force the pipeline to use streaming. For this purpose, the file writer will be streamed. This is achieved by using the `otb::StreamingImageFileWriter` class.

```
typedef otb::ImageFileReader< InputImageType1 >  ReaderType1;
typedef otb::ImageFileReader< InputImageType2 >  ReaderType2;
typedef otb::StreamingImageFileWriter< OutputImageType >  WriterType;
```

The change detector will give a normalized result between 0 and 1. In order to store the result in PNG format we will rescale the results of the change detection in order to use all the output pixel type range of values.

```
typedef itk::ShiftScaleImageFilter< ChangeImageType,
                                    OutputImageType > RescalerType;
```

The `otb::MeanRatioImageFilter` is templated over the types of the two input images and the type of the generated change image.

```
typedef otb::MeanRatioImageFilter<
                              InputImageType1,
                              InputImageType2,
                              ChangeImageType  >      FilterType;
```

The different elements of the pipeline can now be instantiated.

```
ReaderType1::Pointer reader1 = ReaderType1::New();
ReaderType2::Pointer reader2 = ReaderType2::New();
WriterType::Pointer writer = WriterType::New();
FilterType::Pointer   filter = FilterType::New();
RescalerType::Pointer rescaler = RescalerType::New();
```

We set the parameters of the different elements of the pipeline.

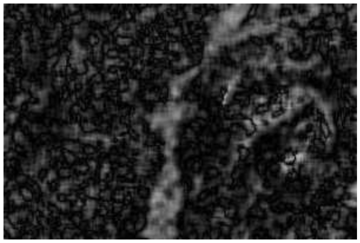


Figure 16.4: Result of the ratio of means change detector

```
reader1->SetFileName( inputFilename1  );
reader2->SetFileName( inputFilename2  );
writer->SetFileName( outputFilename );
float scale = itk::NumericTraits< OutputPixelType >::max();
rescaler->SetScale( scale );
```

The only parameter for this change detector is the radius of the window used for computing the mean of the intensities.

```
filter->SetRadius( atoi(argv[4]) );
```

We build the pipeline by plugging all the elements together.

```
filter->SetInput1( reader1->GetOutput() );
filter->SetInput2( reader2->GetOutput() );


rescaler->SetInput( filter->GetOutput() );
writer->SetInput( rescaler->GetOutput() );
```

Figure 16.4 shows the result of the change detection by ratio of local means.

16.4 Statistical Detectors

16.4.1 Distance between local distributions

This detector is similar to the ratio of means detector (seen in the previous section page 380). Nevertheless, instead of the comparison of means, the comparison is performed to the complete distribution of the two Random Variables (RVs) [44].

The detector is based on the Kullback-Leibler distance between probability density functions (pdfs). In the neighborhood of each pixel of the pair of images I_1 and I_2 to be compared, the distance between local pdfs f_1 and f_2 of RVs X_1 and X_2 is evaluated by:

$$\mathcal{K}(X_1, X_2) = K(X_1|X_2) + K(X_2|X_1) \tag{16.5}$$

$$\text{with} \qquad K(X_j|X_i) = \int_{\mathbb{R}} \log \frac{f_{X_i}(x)}{f_{X_j}(x)} f_{X_i}(x) dx, \qquad i, j = 1, 2. \tag{16.6}$$

In order to reduce the computational time, the local pdfs f_1 and f_2 are not estimated through histogram computations but rather by a cumulant expansion, namely the Edgeworth expansion, with is based on the cumulants of the RVs:

$$f_X(x) = \left(1 + \frac{\kappa_{X;3}}{6} H_3(x) + \frac{\kappa_{X;4}}{24} H_4(x) + \frac{\kappa_{X;5}}{120} H_5(x) + \frac{\kappa_{X;6} + 10\kappa_{X;3}^2}{720} H_6(x)\right) \mathcal{G}_X(x). \tag{16.7}$$

In eq. (16.7), $\mathcal{G}_X$ stands for the Gaussian pdf which has the same mean and variance as the RV X. The $\kappa_{X;k}$ coefficients are the cumulants of order k, and $H_k(x)$ are the Chebyshev-Hermite polynomials of order k (see [46] for deeper explanations).

The source code for this example can be found in the file
`Examples/ChangeDetection/KullbackLeiblerDistanceChDet.cxx`.

This example illustrates the class `otb::KullbackLeiblerDistanceImageFilter` for detecting changes between pairs of images. This filter computes the Kullback-Leibler distance between probability density functions (pdfs). In fact, the Kullback-Leibler distance is itself approximated through a cumulant-based expansion, since the pdfs are approximated through an Edgeworth series. The Kullback-Leibler distance is evaluated by:

$$\begin{aligned} K_{\text{Edgeworth}}(X_1|X_2) = \frac{1}{12}\frac{\kappa_{X_1;3}^2}{\kappa_{X_1;2}^2} + \frac{1}{2}\left(\log\frac{\kappa_{X_2;2}}{\kappa_{X_1;2}} - 1 + \frac{1}{\kappa_{X_2;2}}\left(\kappa_{X_1;1} - \kappa_{X_2;1} + \kappa_{X_1;2}^{1/2}\right)^2\right) \\ - \left(\kappa_{X_2;3}\frac{a_1}{6} + \kappa_{X_2;4}\frac{a_2}{24} + \kappa_{X_2;3}^2\frac{a_3}{72}\right) - \frac{1}{2}\frac{\kappa_{X_2;3}^2}{36}\left(c_6 - 6\frac{c_4}{\kappa_{X_1;2}} + 9\frac{c_2}{\kappa_{X_2;2}^2}\right) \\ - 10\frac{\kappa_{X_1;3}\kappa_{X_2;3}\left(\kappa_{X_1;1} - \kappa_{X_2;1}\right)\left(\kappa_{X_1;2} - \kappa_{X_2;2}\right)}{\kappa_{X_2;2}^6} \end{aligned} \tag{16.8}$$

where

$$
\begin{aligned}
a_1 &= c_3 - 3\frac{\alpha}{\kappa_{X_2;2}} \\
a_2 &= c_4 - 6\frac{c_2}{\kappa_{X_2;2}} + \frac{3}{\kappa_{X_2;2}^2} \\
a_3 &= c_6 - 15\frac{c_4}{\kappa_{X_2;2}} + 45\frac{c_2}{\kappa_{X_2;2}^2} - \frac{15}{\kappa_{X_2;2}^3} \\
c_2 &= \alpha^2 + \beta^2 \\
c_3 &= \alpha^3 + 3\alpha\beta^2 \\
c_4 &= \alpha^4 + 6\alpha^2\beta^2 + 3\beta^4 \\
c_6 &= \alpha^6 + 15\alpha^4\beta^2 + 45\alpha^2\beta^4 + 15\beta^6 \\
\alpha &= \frac{\kappa_{X_1;1} - \kappa_{X_2;1}}{\kappa_{X_2;2}} \\
\beta &= \frac{\kappa_{X_1;2}^{1/2}}{\kappa_{X_2;2}}.
\end{aligned}
$$

$\kappa_{X_i;1}$, $\kappa_{X_i;2}$, $\kappa_{X_i;3}$ and $\kappa_{X_i;4}$ are the cumulants up to order 4 of the random variable X_i $(i = 1,2)$. This example will use the images shown in figure 16.3. These correspond to 2 Radarsat fine mode acquisitions before and after a lava flow resulting from a volcanic eruption.

The program itself is very similar to the ratio of means detector, implemented in `otb::MeanRatioImageFilter`, in section 16.3.2. Nevertheless the corresponding header file has to be used instead.

```
#include "otbKullbackLeiblerDistanceImageFilter.h"
```

The `otb::KullbackLeiblerDistanceImageFilter` is templated over the types of the two input images and the type of the generated change image, in a similar way as the `otb::MeanRatioImageFilter`. It is the only line to be changed from the ratio of means change detection example to perform a change detection through a distance between distributions...

```
typedef otb::KullbackLeiblerDistanceImageFilter<ImageType,
     ImageType,ImageType> FilterType;
```

The different elements of the pipeline can now be instantiated. Follow the ratio of means change detector example.

The only parameter for this change detector is the radius of the window used for computing the cumulants.

```
FilterType::Pointer filter = FilterType::New();
filter->SetRadius( (winSize-1)/2 );
```

Figure 16.5: Result of the Kullback-Leibler change detector

The pipeline is built by plugging all the elements together.

```
filter->SetInput1( reader1->GetOutput() );
filter->SetInput2( reader2->GetOutput() );
```

Figure 16.5 shows the result of the change detection by computing the Kullback-Leibler distance between local pdf through an Edgeworth approximation.

16.4.2 Local Correlation

The correlation coefficient measures the likelihood of a linear relationship between two random variables:

$$\begin{aligned} I_\rho(i,j) &= \frac{1}{N}\frac{\sum_{i,j}(I_1(i,j)-m_{I_1})(I_2(i,j)-m_{I_2})}{\sigma_{I_1}\sigma_{I_2}} \\ &= \sum_{(I_1(i,j),I_2(i,j))}\frac{(I_1(i,j)-m_{I_1})(I_2(i,j)-m_{I_2})}{\sigma_{I_1}\sigma_{I_2}}p_{ij} \end{aligned} \quad (16.9)$$

where $I_1(i,j)$ and $I_2(i,j)$ are the pixel values of the 2 images and p_{ij} is the joint probability density. This is like using a linear model:

$$I_2(i,j) = (I_1(i,j)-m_{I_1})\frac{\sigma_{I_2}}{\sigma_{I_1}} + m_{I_2} \quad (16.10)$$

for which we evaluate the likelihood with p_{ij}.

With respect to the difference detector, this one will be robust to illumination changes.

The source code for this example can be found in the file
`Examples/ChangeDetection/CorrelChDet.cxx`.

This example illustrates the class `otb::CorrelationChangeDetector` for detecting changes between pairs of images. This filter computes the correlation coefficient in the neighborhood

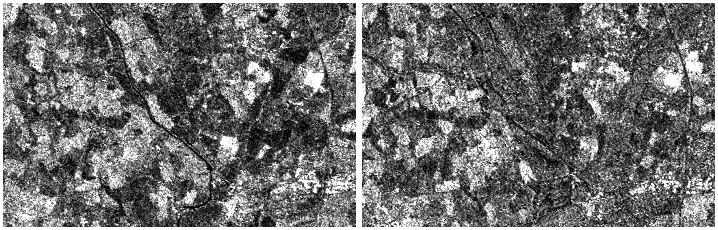


Figure 16.6: Images used for the change detection. Left: Before the flood. Right: during the flood.

of each pixel of the pair of images to be compared. This example will use the images shown in figure 16.6. These correspond to two ERS acquisitions before and during a flood.

We start by including the corresponding header file.

```
#include "otbCorrelationChangeDetector.h"
```

We start by declaring the types for the two input images, the change image and the image to be stored in a file for visualization.

```
  typedef float InternalPixelType;
  typedef unsigned char OutputPixelType;
  typedef otb::Image<InternalPixelType, Dimension>  InputImageType1;
  typedef otb::Image<InternalPixelType, Dimension>  InputImageType2;
  typedef otb::Image<InternalPixelType, Dimension>  ChangeImageType;
  typedef otb::Image<OutputPixelType, Dimension>  OutputImageType;
```

We can now declare the types for the readers. Since the images can be vey large, we will force the pipeline to use streaming. For this purpose, the file writer will be streamed. This is achieved by using the `otb::StreamingImageFileWriter` class.

```
  typedef otb::ImageFileReader< InputImageType1 >  ReaderType1;
  typedef otb::ImageFileReader< InputImageType2 >  ReaderType2;
  typedef otb::StreamingImageFileWriter< OutputImageType >  WriterType;
```

The change detector will give a response which is normalized between 0 and 1. Before saving the image to a file in, for instance, PNG format, we will rescale the results of the change detection in order to use all the output pixel type range of values.

```
  typedef itk::ShiftScaleImageFilter< ChangeImageType,
                                      OutputImageType > RescalerType;
```

The `otb::CorrelationChangeDetector` is templated over the types of the two input images and the type of the generated change image.

```
typedef otb::CorrelationChangeDetector<
                                InputImageType1,
                                InputImageType2,
                                ChangeImageType  >       FilterType;
```

The different elements of the pipeline can now be instantiated.

```
ReaderType1::Pointer reader1 = ReaderType1::New();
ReaderType2::Pointer reader2 = ReaderType2::New();
WriterType::Pointer writer = WriterType::New();
FilterType::Pointer   filter = FilterType::New();
RescalerType::Pointer rescaler = RescalerType::New();
```

We set the parameters of the different elements of the pipeline.

```
reader1->SetFileName( inputFilename1  );
reader2->SetFileName( inputFilename2  );
writer->SetFileName( outputFilename );

float scale = itk::NumericTraits< OutputPixelType >::max();
rescaler->SetScale( scale );
```

The only parameter for this change detector is the radius of the window used for computing the correlation coefficient.

```
filter->SetRadius( atoi(argv[4]) );
```

We build the pipeline by plugging all the elements together.

```
filter->SetInput1( reader1->GetOutput() );
filter->SetInput2( reader2->GetOutput() );
rescaler->SetInput( filter->GetOutput() );
writer->SetInput( rescaler->GetOutput() );
```

Since the processing time of large images can be long, it is interesting to monitor the evolution of the computation. In order to do so, the change detectors can use the command/observer design pattern. This is easily done by attaching an observer to the filter.

```
typedef otb::CommandProgressUpdate<FilterType> CommandType;

CommandType::Pointer observer = CommandType::New();
filter->AddObserver(itk::ProgressEvent(), observer);
```

Figure 16.7 shows the result of the change detection by local correlation.

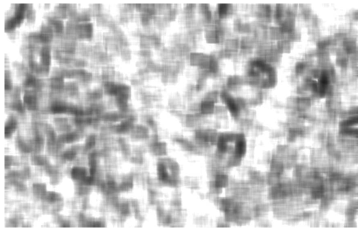


Figure 16.7: Result of the correlation change detector

16.5 Multi-Scale Detectors

16.5.1 Kullback-Leibler Distance between distributions

This technique is an extension of the distance between distributions change detector presented in section 16.4.1. Since this kind of detector is based on cumulants estimations through a sliding window, the idea is just to upgrade the estimation of the cumulants by considering new samples as soon as the sliding window is increasing in size.

Let's consider the following problem: how to update the moments when a $N+1^{th}$ observation x_{N+1} is added to a set of observations $\{x_1, x_2, \ldots, x_N\}$ already considered. The evolution of the central moments may be characterized by:

$$\begin{aligned} \mu_{1,[N]} &= \frac{1}{N} s_{1,[N]} \\ \mu_{r,[N]} &= \frac{1}{N} \sum_{\ell=0}^{r} \binom{r}{\ell} \left(-\mu_{1,[N]}\right)^{r-\ell} s_{\ell,[N]}, \end{aligned} \tag{16.11}$$

where the notation $s_{r,[N]} = \sum_{i=1}^{N} x_i^r$ has been used. Then, Edgeworth series is updated also by transforming moments to cumulants by using:

$$\begin{aligned} \kappa_{X;1} &= \mu_{X;1} \\ \kappa_{X;2} &= \mu_{X;2} - \mu_{X;1}^2 \\ \kappa_{X;3} &= \mu_{X;3} - 3\mu_{X;2}\mu_{X;1} + 2\mu_{X;1}^3 \\ \kappa_{X;4} &= \mu_{X;4} - 4\mu_{X;3}\mu_{X;1} - 3\mu_{X;2}^2 + 12\mu_{X;2}\mu_{X;1}^2 - 6\mu_{X;1}^4. \end{aligned} \tag{16.12}$$

It yields a set of images that represent the change measure according to an increasing size of the analysis window.

The source code for this example can be found in the file
`Examples/ChangeDetection/KullbackLeiblerProfileChDet.cxx`.

This example illustrates the class `otb::KullbackLeiblerProfileImageFilter` for detecting changes between pairs of images, according to a range of window size. This example is very similar, in its principle, to all of the change detection examples, especially the distance between distributions one (section 16.4.1) which uses a fixed window size.

The main differences are:

1. a set of window range instead of a fixed size of window;
2. an output of type `otb::VectorImage`.

Then, the program begins with the `otb::VectorImage` and the `otb::KullbackLeiblerProfileImageFilter` header files in addition to those already details in the `otb::MeanRatioImageFilter` example.

```
#include "otbVectorImage.h"
#include "otbKullbackLeiblerProfileImageFilter.h"
```

The `otb::KullbackLeiblerProfileImageFilter` is templated over the types of the two input images and the type of the generated change image (which is now of multi-components), in a similar way as the `otb::KullbackLeiblerDistanceImageFilter`.

```
typedef otb::Image<PixelType,Dimension> ImageType;
typedef otb::VectorImage<PixelType,Dimension> VectorImageType;
typedef otb::KullbackLeiblerProfileImageFilter<ImageType,
     ImageType,VectorImageType> FilterType;
```

The different elements of the pipeline can now be instantiated in the same way as the ratio of means change detector example.

Two parameters are now required to give the minimum and the maximum size of the analysis window. The program will begin by performing change detection through the smaller window size and then applying moments update of eq. (16.11) by incrementing the radius of the analysis window (i.e. add a ring of width 1 pixel around the current neightborhood shape). The process is applied until the larger window size is reached.

```
FilterType::Pointer filter = FilterType::New();
filter->SetRadius( (winSizeMin-1)/2,(winSizeMax-1)/2 );
filter->SetInput1( reader1->GetOutput() );
filter->SetInput2( reader2->GetOutput() );
```

Figure 16.8 shows the result of the change detection by computing the Kullback-Leibler distance between local pdf through an Edgeworth approximation.

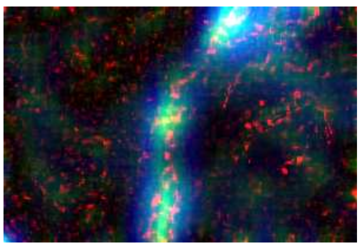


Figure 16.8: Result of the Kullback-Leibler profile change detector, colored composition including the first, 12th and 24th channel of the generated output.

CHAPTER

SEVENTEEN

Classification

17.1 Introduction

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[28]) 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 17.1 illustrates this general framework.

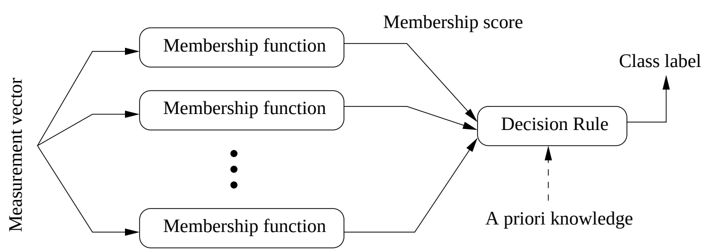


Figure 17.1: Simple conceptual classifier.

This framework closely follows that of Duda and Hart[28]. 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 formulated various classification tasks by using different membership functions and incorporating task specific requirements and prior knowledge

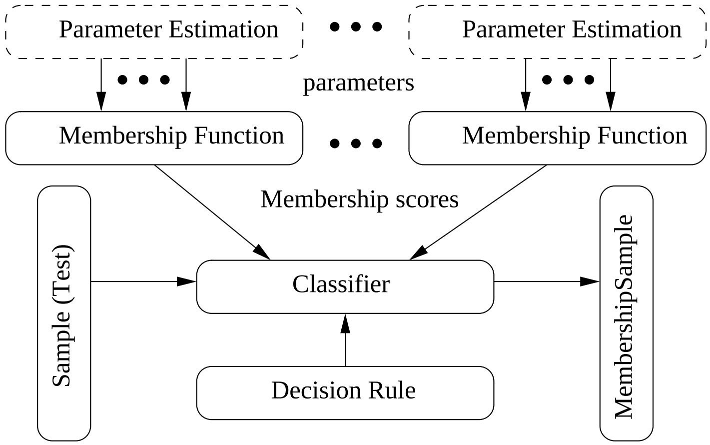


Figure 17.2: Statistical classification framework.

into the decision rule. For example, instead of using probability density functions as membership functions, through distance functions and a minimum value decision rule (which assigns a class from the distance function that returns the smallest value) users can achieve a least squared error classifier. As another example, users can add a rejection scheme to the decision rule so that even in a situation where the membership scores suggest a "winner", a measurement vector can be flagged as ill defined. Such a rejection scheme can avoid risks of assigning a class label without a proper win margin.

17.1.1 k-d Tree Based k-Means Clustering

The source code for this example can be found in the file
`Examples/Classification/KdTreeBasedKMeansClustering.cxx`.

K-means clustering is a popular clustering algorithm because it is simple and usually converges to a reasonable solution. The k-means algorithm works as follows:

1. Obtains the initial k means input from the user.
2. Assigns each measurement vector in a sample container to its closest mean among the k number of means (i.e., update the membership of each measurement vectors to the nearest of the k clusters).
3. Calculates each cluster's mean from the newly assigned measurement vectors (updates the centroid (mean) of k clusters).
4. Repeats step 2 and step 3 until it meets the termination criteria.

The most common termination criteria is that if there is no measurement vector that changes its cluster membership from the previous iteration, then the algorithm stops.

The `itk::Statistics::KdTreeBasedKmeansEstimator` is a variation of this logic. The k-means clustering algorithm is computationally very expensive because it has to recalculate the mean at each iteration. To update the mean values, we have to calculate the distance between k means and each and every measurement vector. To reduce the computational burden, the KdTreeBasedKmeansEstimator uses a special data structure: the k-d tree (`itk::Statistics::KdTree`) with additional information. The additional information includes the number and the vector sum of measurement vectors under each node under the tree architecture.

With such additional information and the k-d tree data structure, we can reduce the computational cost of the distance calculation and means. Instead of calculating each measurement vectors and k means, we can simply compare each node of the k-d tree and the k means. This idea of utilizing a k-d tree can be found in multiple articles [3] [68] [50]. Our implementation of this scheme follows the article by the Kanungo et al [50].

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 this k-means algorithm requires a `itk::Statistics::KdTree` object as an input, we include the KdTree class header file. As mentioned above, we need a k-d tree with the vector sum and the number of measurement vectors. Therefore we use the `itk::Statistics::WeightedCentroidKdTreeGenerator` instead of the `itk::Statistics::KdTreeGenerator` that generate a k-d tree without such additional information.

```
#include "itkKdTree.h"
#include "itkWeightedCentroidKdTreeGenerator.h"
```

The KdTreeBasedKmeansEstimator class is the implementation of the k-means algorithm. It does not create k clusters. Instead, it returns the mean estimates for the k clusters.

```
#include "itkKdTreeBasedKmeansEstimator.h"
```

To generate the clusters, we must create k instances of `itk::Statistics::EuclideanDistance` function as the membership functions for each cluster and plug that—along with a sample—into an `itk::Statistics::SampleClassifier` object to get a `itk::Statistics::MembershipSample` that stores pairs of measurement vectors and their associated class labels (k labels).

```
#include "itkMinimumDecisionRule.h"
#include "itkEuclideanDistance.h"
#include "itkSampleClassifier.h"
```

We will fill the sample with random variables from two normal distribution using the `itk::Statistics::NormalVariateGenerator`.

```
#include "itkNormalVariateGenerator.h"
```

Since the `NormalVariateGenerator` class only supports 1-D, we define our measurement vector type as one component vector. We then, create a `ListSample` object for data inputs. Each measurement vector is of length 1. We set this using the `SetMeasurementVectorSize()` method.

```
typedef itk::Vector< double, 1 > MeasurementVectorType;
typedef itk::Statistics::ListSample< MeasurementVectorType > SampleType;
SampleType::Pointer sample = SampleType::New();
sample->SetMeasurementVectorSize( 1 );
```

The following code snippet creates a NormalVariateGenerator object. Since the random variable generator returns values according to the standard normal distribution (The mean is zero, and the standard deviation is one), before pushing random values into the `sample`, we change the mean and standard deviation. We want two normal (Gaussian) distribution data. We have two for loops. Each `for` loop uses different mean and standard deviation. Before we fill the `sample` with the second distribution data, we call `Initialize(random seed)` method, to recreate the pool of random variables in the `normalGenerator`.

To see the probability density plots from the two distribution, refer to the Figure 17.3.

```
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;
```

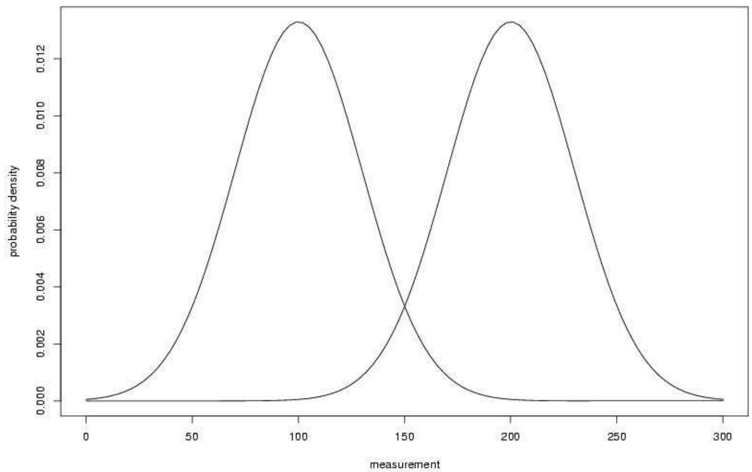


Figure 17.3: Two normal distributions' probability density plot (The means are 100 and 200, and the standard deviation is 30)

```
for ( unsigned int i = 0 ; i < 100 ; ++i )
{
  mv[0] = ( normalGenerator->GetVariate() * standardDeviation ) + mean;
  sample->PushBack( mv );
}
```

We create a k-d tree.

```
typedef itk::Statistics::WeightedCentroidKdTreeGenerator< SampleType >
    TreeGeneratorType;
TreeGeneratorType::Pointer treeGenerator = TreeGeneratorType::New();

treeGenerator->SetSample( sample );
treeGenerator->SetBucketSize( 16 );
treeGenerator->Update();
```

Once we have the k-d tree, it is a simple procedure to produce k mean estimates.

We create the KdTreeBasedKmeansEstimator. Then, we provide the initial mean values using the `SetParameters()`. Since we are dealing with two normal distribution in a 1-D space, the size of the mean value array is two. The first element is the first mean value, and the second is the second mean value. If we used two normal distributions in a 2-D space, the size of array would be four, and the first two elements would be the two components of the first normal distribution's mean vector. We plug-in the k-d tree using the `SetKdTree()`.

The remaining two methods specify the termination condition. The estimation process stops when the number of iterations reaches the maximum iteration value set by the `SetMaximumIteration()`, or the distances between the newly calculated mean (centroid) values and previous ones are within the threshold set by the `SetCentroidPositionChangesThreshold()`. The final step is to call the `StartOptimization()` method.

The for loop will print out the mean estimates from the estimation process.

```
typedef TreeGeneratorType::KdTreeType TreeType;
typedef itk::Statistics::KdTreeBasedKmeansEstimator<TreeType> EstimatorType;
EstimatorType::Pointer estimator = EstimatorType::New();

EstimatorType::ParametersType initialMeans(2);
initialMeans[0] = 0.0;
initialMeans[1] = 0.0;

estimator->SetParameters( initialMeans );
estimator->SetKdTree( treeGenerator->GetOutput() );
estimator->SetMaximumIteration( 200 );
estimator->SetCentroidPositionChangesThreshold(0.0);
estimator->StartOptimization();

EstimatorType::ParametersType estimatedMeans = estimator->GetParameters();

for ( unsigned int i = 0 ; i < 2 ; ++i )
{
  std::cout << "cluster[" << i << "] " << std::endl;
  std::cout << "    estimated mean : " << estimatedMeans[i] << std::endl;
}
```

If we are only interested in finding the mean estimates, we might stop. However, to illustrate how a classifier can be formed using the statistical classification framework. We go a little bit further in this example.

Since the k-means algorithm is an minimum distance classifier using the estimated k means and the measurement vectors. We use the EuclideanDistance class as membership functions. Our choice for the decision rule is the `itk::Statistics::MinimumDecisionRule` that returns the index of the membership functions that have the smallest value for a measurement vector.

After creating a SampleClassifier object and a MinimumDecisionRule object, we plug-in the `decisionRule` and the `sample` to the classifier. Then, we must specify the number of classes that will be considered using the `SetNumberOfClasses()` method.

The remainder of the following code snippet shows how to use user-specified class labels. The classification result will be stored in a MembershipSample object, and for each measurement vector, its class label will be one of the two class labels, 100 and 200 (`unsigned int`).

```
typedef itk::Statistics::EuclideanDistance< MeasurementVectorType >
    MembershipFunctionType;
typedef itk::MinimumDecisionRule DecisionRuleType;
DecisionRuleType::Pointer decisionRule = DecisionRuleType::New();

typedef itk::Statistics::SampleClassifier< SampleType > ClassifierType;
ClassifierType::Pointer classifier = ClassifierType::New();

classifier->SetDecisionRule( (itk::DecisionRuleBase::Pointer) decisionRule);
classifier->SetSample( sample );
classifier->SetNumberOfClasses( 2 );

std::vector< unsigned int > classLabels;
classLabels.resize( 2 );
classLabels[0] = 100;
classLabels[1] = 200;

classifier->SetMembershipFunctionClassLabels( classLabels );
```

The `classifier` is almost ready to do the classification process except that it needs two membership functions that represents two clusters respectively.

In this example, the two clusters are modeled by two Euclidean distance functions. The distance function (model) has only one parameter, its mean (centroid) set by the `SetOrigin()` method. To plug-in two distance functions, we call the `AddMembershipFunction()` method. Then invocation of the `Update()` method will perform the classification.

```
std::vector< MembershipFunctionType::Pointer > membershipFunctions;
MembershipFunctionType::OriginType origin( sample->GetMeasurementVectorSize() );
int index = 0;
for ( unsigned int i = 0 ; i < 2 ; i++ )
{
  membershipFunctions.push_back( MembershipFunctionType::New() );
  for ( unsigned int j = 0 ; j < sample->GetMeasurementVectorSize(); j++ )
  {
    origin[j] = estimatedMeans[index++];
  }
  membershipFunctions[i]->SetOrigin( origin );
  classifier->AddMembershipFunction( membershipFunctions[i].GetPointer() );
}

classifier->Update();
```

The following code snippet prints out the measurement vectors and their class labels in the `sample`.

```
  ClassifierType::OutputType* membershipSample = classifier->GetOutput();
  ClassifierType::OutputType::ConstIterator iter = membershipSample->Begin();

  while ( iter != membershipSample->End() )
  {
    std::cout << "measurement vector = " << iter.GetMeasurementVector()
        << "class label = " << iter.GetClassLabel()
        << std::endl;
    ++iter;
  }
```

17.1.2 K-Means Classification

Simple version

The source code for this example can be found in the file
`Examples/Classification/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 "otbImage.h"
#include "otbImageFileReader.h"
#include "otbImageFileWriter.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 `otb::ImageFileReader` needed for reading the input image, create one and set its input filename.

```
  typedef signed short       PixelType;
  const unsigned int          Dimension = 2;

  typedef otb::Image<PixelType, Dimension > ImageType;

  typedef otb::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 [0:255], and when it is used for holding four classes, the non-contiguous labels will be (0,64,128,192). The selection of the mode to use is done with the method `SetUseContiguousLabels()`.

```
const unsigned int useNonContiguousLabels = atoi( argv[3] );

kmeansFilter->SetUseNonContiguousLabels( useNonContiguousLabels );
```

For each one of the classes we must provide a tentative initial value for the mean of the class. Given that this is a scalar image, each one of the means is simply a scalar value. Note however that in a general case of K-Means, the input image would be a vector image and therefore the means will be vectors of the same dimension as the image pixels.

```
for( unsigned k=0; k < numberOfInitialClasses; k++ )
  {
  const double userProvidedInitialMean = atof( argv[k+argoffset] );
  kmeansFilter->AddClassWithInitialMean( userProvidedInitialMean );
  }
```

The `itk::ScalarImageKmeansImageFilter` is predefined for producing an 8 bits scalar image as output. This output image contains labels associated to each one of the classes in the K-Means algorithm. In the following lines we use the `OutputImageType` in order to instantiate the type of a `otb::ImageFileWriter`. Then create one, and connect it to the output of the classification filter.

```
typedef KMeansFilterType::OutputImageType  OutputImageType;

typedef otb::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 17.4 illustrates the effect of this filter with three classes. The means can be estimated by ScalarImageKmeansModelEstimator.cxx.

The source code for this example can be found in the file
`Examples/Classification/ScalarImageKmeansModelEstimator.cxx`.

This example shows how to compute the KMeans model of an Scalar Image.

The `itk::Statistics::KdTreeBasedKmeansEstimator` 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. One of the drawbacks of this technique is that the

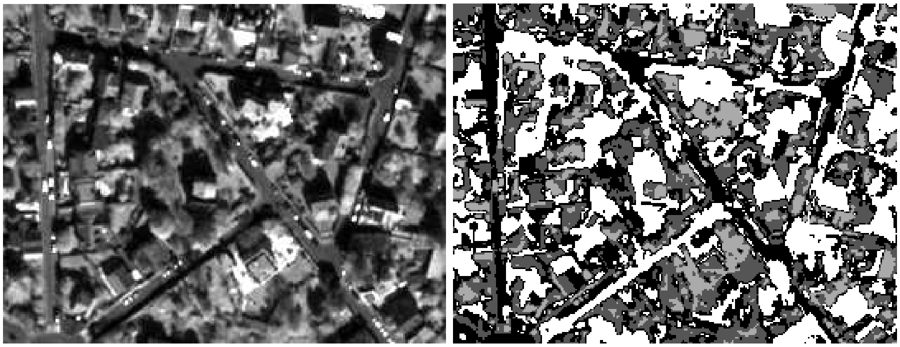


Figure 17.4: Effect of the KMeans classifier. Left: original image. Right: image of classes.

spatial distribution of the pixels is not considered at all. It is common therefore to combine the classification resulting from K-Means with other segmentation techniques that will use the classification as a prior and add spatial information to it in order to produce a better segmentation.

```
// Create a List from the scalar image
typedef itk::Statistics::ScalarImageToListAdaptor< ImageType >   AdaptorType;

AdaptorType::Pointer adaptor = AdaptorType::New();

adaptor->SetImage(  reader->GetOutput() );



// Define the Measurement vector type from the AdaptorType
typedef AdaptorType::MeasurementVectorType  MeasurementVectorType;


// Create the K-d tree structure
typedef itk::Statistics::WeightedCentroidKdTreeGenerator<
                                                    AdaptorType >
                                                            TreeGeneratorType;

TreeGeneratorType::Pointer treeGenerator = TreeGeneratorType::New();

treeGenerator->SetSample( adaptor );
treeGenerator->SetBucketSize( 16 );
treeGenerator->Update();



typedef TreeGeneratorType::KdTreeType TreeType;
typedef itk::Statistics::KdTreeBasedKmeansEstimator<TreeType> EstimatorType;
```

```
EstimatorType::Pointer estimator = EstimatorType::New();

const unsigned int numberOfClasses = 4;

EstimatorType::ParametersType initialMeans( numberOfClasses );
initialMeans[0] = 25.0;
initialMeans[1] = 125.0;
initialMeans[2] = 250.0;

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 < numberOfClasses ; ++i )
  {
  std::cout << "cluster[" << i << "] " << std::endl;
  std::cout << "    estimated mean : " << estimatedMeans[i] << std::endl;
  }
```

General approach

The source code for this example can be found in the file
`Examples/Classification/KMeansImageClassificationExample.cxx`.

The K-Means classification proposed by ITK for images is limited to scalar images and is not streamed. In this example, we show how the use of the `otb::KMeansImageClassificationFilter` allows for a simple implementation of a K-Means classification application. We will start by including the appropirate header file.

```
#include "otbKMeansImageClassificationFilter.h"
```

We will assume double precision input images and will also define the type for the labeled pixels.

```
const unsigned int     Dimension = 2;
typedef double         PixelType;
typedef unsigned short LabeledPixelType;
```

Our classifier will be genric enough to be able to process images with any number of bands. We read the images as `otb::VectorImages`. The labeled image will be a scalar image.

```
typedef otb::VectorImage<PixelType,Dimension> ImageType;
typedef otb::Image<LabeledPixelType,Dimension> LabeledImageType;
```

We can now define the type for the classifier filter, which is templated over its input and output image types.

```
typedef otb::KMeansImageClassificationFilter<ImageType,LabeledImageType>
                                                  ClassificationFilterType;
typedef ClassificationFilterType::KMeansParametersType KMeansParametersType;
```

And finally, we define the reader and the writer. Since the images to classify can be very big, we will use a streamed writer which will trigger the streaming ability of the classifier.

```
typedef otb::ImageFileReader<ImageType> ReaderType;
typedef otb::StreamingImageFileWriter<LabeledImageType> WriterType;
```

We instantiate the classifier and the reader objects and we set their parameters. Please note the call of the `GenerateOutputInformation()` method on the reader in order to have available the information about the input image (size, number of bands, etc.) without needing to actually read the image.

```
ClassificationFilterType::Pointer filter = ClassificationFilterType::New();

ReaderType::Pointer reader = ReaderType::New();
reader->SetFileName(infname);
reader->GenerateOutputInformation();
```

The classifier needs as input the centroids of the classes. We declare the parameter vector, and we read the centroids from the arguments of the program.

```
const unsigned int sampleSize = ClassificationFilterType::MaxSampleDimension;
const unsigned int parameterSize = nbClasses * sampleSize;
 KMeansParametersType parameters;

 parameters.SetSize(parameterSize);
 parameters.Fill(0);
```

```
  for(unsigned int i = 0; i<nbClasses;++i)
    {
      for(unsigned int j = 0; j <
    reader->GetOutput()->GetNumberOfComponentsPerPixel();++j)
{
  parameters[i*sampleSize+j]=
    atof(argv[4+i*
      reader->GetOutput()->GetNumberOfComponentsPerPixel()
      +j]);
}
    }

  std::cout<<"Parameters: "<<parameters<<std::endl;
```

We set the parameters for the classifier, we plug the pipeline and trigger its execution by updating the output of the writer.

```
filter->SetCentroids(parameters);
filter->SetInput(reader->GetOutput());

WriterType::Pointer writer = WriterType::New();
writer->SetInput(filter->GetOutput());
writer->SetFileName(outfname);
writer->Update();
```

17.1.3 Bayesian Plug-In Classifier

The source code for this example can be found in the file
`Examples/Classification/BayesianPluginClassifier.cxx`.

In this example, we present a system that places measurement vectors into two Gaussian classes. The Figure 17.5 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::EuclideanDistance`. Since the membership function is different, the membership function requires a different set of parameters, mean vectors and covariance matrices. We choose the `itk::Statistics::MeanCalculator` (sample mean) and the `itk::Statistics::CovarianceCalculator` (sample covariance) for the estimation algorithms of the two parameters. If we want more robust estimation algorithm, we can replace these estimation algorithms with more alternatives without changing other components in the classifier system.

It is a bad idea to use the same sample for test and training (parameter estimation) of the parameters. However, for simplicity, in this example, we use a sample for test and training.

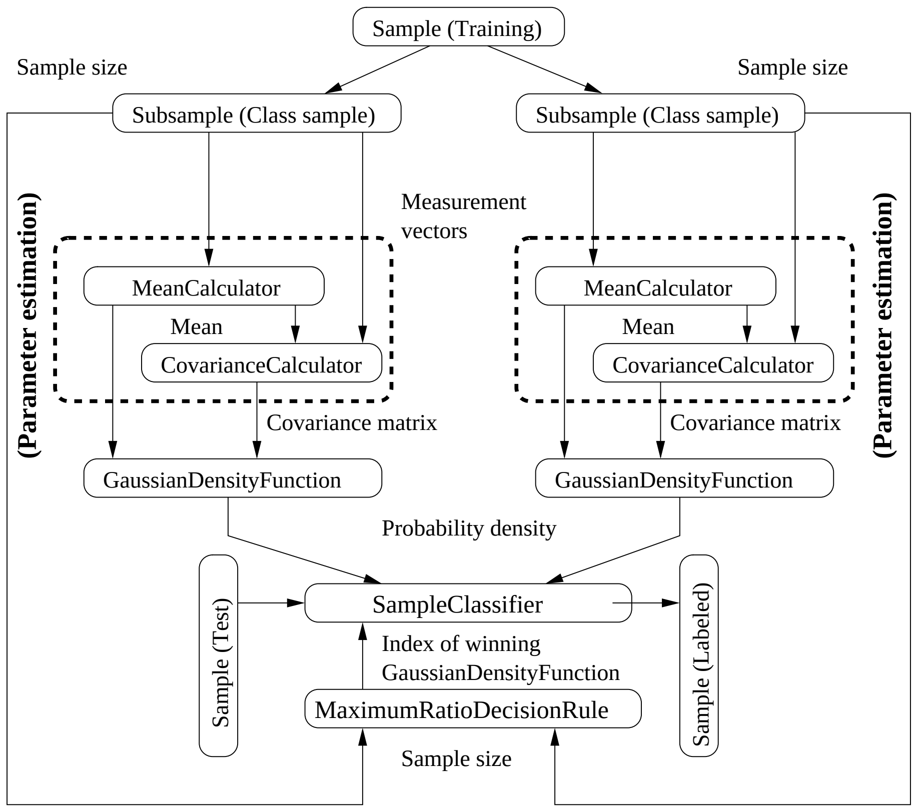


Figure 17.5: Bayesian plug-in classifier for two Gaussian classes.

We use the itk::Statistics::ListSample as the sample (test and training). The itk::Vector is our measurement vector class. To store measurement vectors into two separate sample containers, we use the itk::Statistics::Subsample objects.

```
#include "itkVector.h"
#include "itkListSample.h"
#include "itkSubsample.h"
```

The following two files provides us the parameter estimation algorithms.

```
#include "itkMeanCalculator.h"
#include "itkCovarianceCalculator.h"
```

The following files define the components required by ITK statistical classification framework: the decision rule, the membership function, and the classifier.

```
#include "itkMaximumRatioDecisionRule.h"
#include "itkGaussianDensityFunction.h"
#include "itkSampleClassifier.h"
```

We will fill the sample with random variables from two normal distribution using the itk::Statistics::NormalVariateGenerator.

```
#include "itkNormalVariateGenerator.h"
```

Since the NormalVariateGenerator class only supports 1-D, we define our measurement vector type as a one component vector. We then, create a ListSample object for data inputs.

We also create two Subsample objects that will store the measurement vectors in sample into two separate sample containers. Each Subsample object stores only the measurement vectors belonging to a single class. This class sample will be used by the parameter estimation algorithms.

```
  typedef itk::Vector< double, 1 > MeasurementVectorType;
  typedef itk::Statistics::ListSample< MeasurementVectorType > SampleType;
  SampleType::Pointer sample = SampleType::New();
  sample->SetMeasurementVectorSize( 1 ); // length of measurement vectors
                                         // in the sample.

  typedef itk::Statistics::Subsample< SampleType > ClassSampleType;
  std::vector< ClassSampleType::Pointer > classSamples;
  for ( unsigned int i = 0 ; i < 2 ; ++i )
    {
    classSamples.push_back( ClassSampleType::New() );
    classSamples[i]->SetSample( sample );
    }
```

The following code snippet creates a NormalVariateGenerator object. Since the random variable generator returns values according to the standard normal distribution (the mean is zero, and the standard deviation is one) before pushing random values into the `sample`, we change the mean and standard deviation. We want two normal (Gaussian) distribution data. We have two for loops. Each for loop uses different mean and standard deviation. Before we fill the `sample` with the second distribution data, we call `Initialize(random seed)` method, to recreate the pool of random variables in the `normalGenerator`. In the second for loop, we fill the two class samples with measurement vectors using the `AddInstance()` method.

To see the probability density plots from the two distributions, refer to Figure 17.3.

```
typedef itk::Statistics::NormalVariateGenerator NormalGeneratorType;
NormalGeneratorType::Pointer normalGenerator = NormalGeneratorType::New();

normalGenerator->Initialize( 101 );

MeasurementVectorType mv;
double mean = 100;
double standardDeviation = 30;
SampleType::InstanceIdentifier id = 0UL;
for ( unsigned int i = 0 ; i < 100 ; ++i )
  {
  mv.Fill( (normalGenerator->GetVariate() * standardDeviation ) + mean);
  sample->PushBack( mv );
  classSamples[0]->AddInstance( id );
  ++id;
  }

normalGenerator->Initialize( 3024 );
mean = 200;
standardDeviation = 30;
for ( unsigned int i = 0 ; i < 100 ; ++i )
  {
  mv.Fill( (normalGenerator->GetVariate() * standardDeviation ) + mean);
  sample->PushBack( mv );
  classSamples[1]->AddInstance( id );
  ++id;
  }
```

In the following code snippet, notice that the template argument for the MeanCalculator and CovarianceCalculator is `ClassSampleType` (i.e., type of Subsample) instead of SampleType (i.e., type of ListSample). This is because the parameter estimation algorithms are applied to the class sample.

```
typedef itk::Statistics::MeanCalculator< ClassSampleType > MeanEstimatorType;
typedef itk::Statistics::CovarianceCalculator< ClassSampleType >
  CovarianceEstimatorType;
```

```
std::vector< MeanEstimatorType::Pointer > meanEstimators;
std::vector< CovarianceEstimatorType::Pointer > covarianceEstimators;

for ( unsigned int i = 0 ; i < 2 ; ++i )
  {
  meanEstimators.push_back( MeanEstimatorType::New() );
  meanEstimators[i]->SetInputSample( classSamples[i] );
  meanEstimators[i]->Update();

  covarianceEstimators.push_back( CovarianceEstimatorType::New() );
  covarianceEstimators[i]->SetInputSample( classSamples[i] );
  covarianceEstimators[i]->SetMean( meanEstimators[i]->GetOutput() );
  covarianceEstimators[i]->Update();
  }
```

We print out the estimated parameters.

```
for ( unsigned int i = 0 ; i < 2 ; ++i )
  {
  std::cout << "class[" << i << "] " << std::endl;
  std::cout << "    estimated mean : "
            << *(meanEstimators[i]->GetOutput())
            << "    covariance matrix : "
            << *(covarianceEstimators[i]->GetOutput()) << std::endl;
  }
```

After creating a SampleClassifier object and a MaximumRatioDecisionRule object, we plug in the `decisionRule` and the `sample` to the classifier. Then, we specify the number of classes that will be considered using the `SetNumberOfClasses()` method.

The MaximumRatioDecisionRule requires a vector of *a priori* probability values. Such *a priori* probability will be the $P(\omega_i)$ of the following variation of the Bayes decision rule:

$$\text{Decide } \omega_i \text{ if } \frac{p(\overrightarrow{x}|\omega_i)}{p(\overrightarrow{x}|\omega_j)} > \frac{P(\omega_j)}{P(\omega_i)} \text{ for all } j \neq i \tag{17.1}$$

The remainder of the code snippet shows how to use user-specified class labels. The classification result will be stored in a MembershipSample object, and for each measurement vector, its class label will be one of the two class labels, 100 and 200 (`unsigned int`).

```
typedef itk::Statistics::GaussianDensityFunction< MeasurementVectorType >
  MembershipFunctionType;
typedef itk::MaximumRatioDecisionRule DecisionRuleType;
DecisionRuleType::Pointer decisionRule = DecisionRuleType::New();
```

```
DecisionRuleType::APrioriVectorType aPrioris;
aPrioris.push_back( classSamples[0]->GetTotalFrequency()
                    / sample->GetTotalFrequency() ) ;
aPrioris.push_back( classSamples[1]->GetTotalFrequency()
                    / sample->GetTotalFrequency() ) ;
decisionRule->SetAPriori( aPrioris );

typedef itk::Statistics::SampleClassifier< SampleType > ClassifierType;
ClassifierType::Pointer classifier = ClassifierType::New();

classifier->SetDecisionRule( (itk::DecisionRuleBase::Pointer) decisionRule);
classifier->SetSample( sample );
classifier->SetNumberOfClasses( 2 );

std::vector< unsigned int > classLabels;
classLabels.resize( 2 );
classLabels[0] = 100;
classLabels[1] = 200;
classifier->SetMembershipFunctionClassLabels(classLabels);
```

The `classifier` is almost ready to perform the classification except that it needs two membership functions that represent the two clusters.

In this example, we can imagine that the two clusters are modeled by two Euclidean distance functions. The distance function (model) has only one parameter, the mean (centroid) set by the `SetOrigin()` method. To plug-in two distance functions, we call the `AddMembershipFunction()` method. Then invocation of the `Update()` method will perform the classification.

```
std::vector< MembershipFunctionType::Pointer > membershipFunctions;
for ( unsigned int i = 0 ; i < 2 ; i++ )
  {
  membershipFunctions.push_back(MembershipFunctionType::New());
  membershipFunctions[i]->SetMean( meanEstimators[i]->GetOutput() );
  membershipFunctions[i]->
    SetCovariance( covarianceEstimators[i]->GetOutput() );
  classifier->AddMembershipFunction(membershipFunctions[i].GetPointer());
  }

classifier->Update();
```

The following code snippet prints out pairs of a measurement vector and its class label in the `sample`.

```
ClassifierType::OutputType* membershipSample = classifier->GetOutput();
ClassifierType::OutputType::ConstIterator iter = membershipSample->Begin();
```

```
while ( iter != membershipSample->End() )
  {
  std::cout << "measurement vector = " << iter.GetMeasurementVector()
            << "class label = " << iter.GetClassLabel() << std::endl;
  ++iter;
  }
```

17.1.4 Expectation Maximization Mixture Model Estimation

The source code for this example can be found in the file
`Examples/Classification/ExpectationMaximizationMixtureModelEstimator.cxx`.

In this example, we present ITK's implementation of the expectation maximization (EM) process to generate parameter estimates for a two Gaussian component mixture model.

The Bayesian plug-in classifier example (see Section 17.1.3) used two Gaussian probability density functions (PDF) to model two Gaussian distribution classes (two models for two class). However, in some cases, we want to model a distribution as a mixture of several different distributions. Therefore, the probability density function ($p(x)$) of a mixture model can be stated as follows :

$$p(x) = \sum_{i=0}^{c} \alpha_i f_i(x) \tag{17.2}$$

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 provide 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 17.3.

```
  typedef itk::Statistics::NormalVariateGenerator NormalGeneratorType;
```

```
NormalGeneratorType::Pointer normalGenerator = NormalGeneratorType::New();

normalGenerator->Initialize( 101 );

MeasurementVectorType mv;
double mean = 100;
double standardDeviation = 30;
for ( unsigned int i = 0 ; i < 100 ; ++i )
  {
  mv[0] = ( normalGenerator->GetVariate() * standardDeviation ) + mean;
  sample->PushBack( mv );
  }

normalGenerator->Initialize( 3024 );
mean = 200;
standardDeviation = 30;
for ( unsigned int i = 0 ; i < 100 ; ++i )
  {
  mv[0] = ( normalGenerator->GetVariate() * standardDeviation ) + mean;
  sample->PushBack( mv );
  }
```

In the following code snippet notice that the template argument for the MeanCalculator and CovarianceCalculator is `ClassSampleType` (i.e., type of Subsample) instead of `SampleType` (i.e., type of ListSample). This is because the parameter estimation algorithms are applied to the class sample.

```
typedef itk::Array< double > ParametersType;
ParametersType params( 2 );

std::vector< ParametersType > initialParameters( numberOfClasses );
params[0] = 110.0;
params[1] = 800.0;
initialParameters[0] = params;

params[0] = 210.0;
params[1] = 850.0;
initialParameters[1] = params;

typedef itk::Statistics::GaussianMixtureModelComponent< SampleType >
  ComponentType;

std::vector< ComponentType::Pointer > components;
for ( unsigned int i = 0 ; i < numberOfClasses ; i++ )
  {
  components.push_back( ComponentType::New() );
  (components[i])->SetSample( sample );
```

```
    (components[i])->SetParameters( initialParameters[i] );
    }
```

We run the estimator.

```
  typedef itk::Statistics::ExpectationMaximizationMixtureModelEstimator<
                           SampleType > EstimatorType;
  EstimatorType::Pointer estimator = EstimatorType::New();

  estimator->SetSample( sample );
  estimator->SetMaximumIteration( 200 );

  itk::Array< double > initialProportions(numberOfClasses);
  initialProportions[0] = 0.5;
  initialProportions[1] = 0.5;

  estimator->SetInitialProportions( initialProportions );

  for ( unsigned int i = 0 ; i < numberOfClasses ; i++)
    {
    estimator->AddComponent( (ComponentType::Superclass*)
                             (components[i]).GetPointer() );
    }

  estimator->Update();
```

We then print out the estimated parameters.

```
  for ( unsigned int i = 0 ; i < numberOfClasses ; i++ )
    {
    std::cout << "Cluster[" << i << "]" << std::endl;
    std::cout << "    Parameters:" << std::endl;
    std::cout << "         " << (components[i])->GetFullParameters()
              << std::endl;
    std::cout << "    Proportion: ";
    std::cout << "         " << (*estimator->GetProportions())[i] << std::endl;
    }
```

17.1.5 Classification using Markov Random Fields

Markov Random Fields are probabilistic models that use the statistical dependency between pixels in a neighborhood to infeer the value of a give pixel.

ITK framework

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 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 [7].

The source code for this example can be found in the file
`Examples/Classification/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 "otbImage.h"
#include "itkFixedArray.h"
#include "otbImageFileReader.h"
#include "otbImageFileWriter.h"
#include "itkScalarToArrayCastImageFilter.h"
```

The following headers are related to the statistical classification classes.

```
#include "itkMRFImageFilter.h"
#include "itkDistanceToCentroidMembershipFunction.h"
#include "itkMinimumDecisionRule.h"
#include "itkImageClassifierBase.h"
```

First we define the pixel type and dimension of the image that we intend to classify. With this image type we can also declare the `otb::ImageFileReader` needed for reading the input image, create one and set its input filename.

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

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

```
typedef otb::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 17.1.2.

```
typedef unsigned char       LabelPixelType;

typedef otb::Image<LabelPixelType, Dimension > LabelImageType;

typedef otb::ImageFileReader< LabelImageType > LabelReaderType;
LabelReaderType::Pointer labelReader = LabelReaderType::New();
labelReader->SetFileName( inputLabelImageFileName );
```

Since the Markov Random Field algorithm is defined in general for images whose pixels have multiple components, that is, images of vector type, we must adapt our scalar image in order to satisfy the interface expected by the `MRFImageFilter`. We do this by using the `itk::ScalarToArrayCastImageFilter`. With this filter we will present our scalar image as a vector image whose vector pixels contain a single component.

```
typedef itk::FixedArray<LabelPixelType,1>  ArrayPixelType;

typedef otb::Image< ArrayPixelType, Dimension > ArrayImageType;

typedef itk::ScalarToArrayCastImageFilter<
                   ImageType, ArrayImageType > ScalarToArrayFilterType;

ScalarToArrayFilterType::Pointer
  scalarToArrayFilter = ScalarToArrayFilterType::New();
scalarToArrayFilter->SetInput( reader->GetOutput() );
```

With the input image type `ImageType` and labeled image type `LabelImageType` we instantiate the type of the `itk::MRFImageFilter` that will apply the Markov Random Field algorithm in order to refine the pixel classification.

```
typedef itk::MRFImageFilter< ArrayImageType, LabelImageType > MRFFilterType;

MRFFilterType::Pointer mrfFilter = MRFFilterType::New();

mrfFilter->SetInput( scalarToArrayFilter->GetOutput() );
```

We set now some of the parameters for the MRF filter. In particular, the number of classes to be used during the classification, the maximum number of iterations to be run in this filter and the error tolerance that will be used as a criterion for convergence.

```
mrfFilter->SetNumberOfClasses( numberOfClasses );
mrfFilter->SetMaximumNumberOfIterations( numberOfIterations );
mrfFilter->SetErrorTolerance( 1e-7 );
```

The smoothing factor represents the tradeoff between fidelity to the observed image and the smoothness of the segmented image. Typical smoothing factors have values between 1 5. This factor will multiply the weights that define the influence of neighbors on the classification of a given pixel. The higher the value, the more uniform will be the regions resulting from the classification refinement.

```
mrfFilter->SetSmoothingFactor( smoothingFactor );
```

Given that the MRF filter needs 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 needs a decision rule to be set by the user. Note that we must use `GetPointer()` in the call of the `SetDecisionRule()` method because we are passing a SmartPointer, and smart pointer cannot perform polymorphism, we must then extract the raw pointer that is associated to the smart pointer. This extraction is done with the GetPointer() method.

```
typedef itk::MinimumDecisionRule DecisionRuleType;

DecisionRuleType::Pointer  classifierDecisionRule = DecisionRuleType::New();

classifier->SetDecisionRule( classifierDecisionRule.GetPointer() );
```

We now instantiate the membership functions. In this case we use the `itk::Statistics::DistanceToCentroidMembershipFunction` class templated over the pixel type of the vector image, which in our example happens to be a vector of dimension 1.

```
typedef itk::Statistics::DistanceToCentroidMembershipFunction<
                                                    ArrayPixelType >
                                                       MembershipFunctionType;

typedef MembershipFunctionType::Pointer MembershipFunctionPointer;
```

```
double meanDistance = 0;
vnl_vector<double> centroid(1);
for( unsigned int i=0; i < numberOfClasses; i++ )
  {
  MembershipFunctionPointer membershipFunction =
                                     MembershipFunctionType::New();

  centroid[0] = atof( argv[i+numberOfArgumentsBeforeMeans] );

  membershipFunction->SetCentroid( centroid );

  classifier->AddMembershipFunction( membershipFunction );
  meanDistance += static_cast< double > (centroid[0]);
  }
meanDistance /= numberOfClasses;
```

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 Markov 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 `otb::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 otb::ImageFileWriter< OutputImageType > WriterType;

WriterType::Pointer writer = WriterType::New();

writer->SetInput( intensityRescaler->GetOutput() );

writer->SetFileName( outputImageFileName );
```

We are now ready for triggering the execution of the pipeline. This is done by simply invoking the `Update()` method in the writer. This call will propagate the update request to the reader and then to the MRF filter.

```
try
  {
  writer->Update();
```

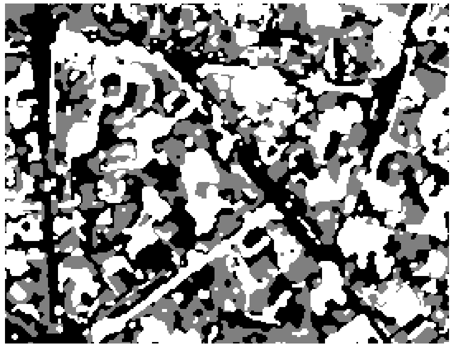


Figure 17.6: Effect of the MRF filter.

```
    }
  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 17.6 illustrates the effect of this filter with four 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 described in section 17.1.2. The obtained result can be compared with the one of figure 17.4 to see the interest of using the MRF approach in order to ensure the regularization of the classified image.

OTB framework

The ITK approach was considered not to be flexible enough for some remote sensing applications. Therefore, we decided to implement our own framework.

The source code for this example can be found in the file
`Examples/Markov/MarkovClassification1Example.cxx`.

This example illustrates the details of the `otb::MarkovRandomFieldFilter`. This filter is an application of the Markov Random Fields for classification, segmentation or restauration.

This example applies the `otb::MarkovRandomFieldFilter` to classify an image into four classes defined by their mean and variance. The optimization is done using an Metropolis algorithm with a random sampler. The regularization energy is defined by a Potts model and the fidelity by a Gaussian model.

The first step toward the use of this filter is the inclusion of the proper header files.

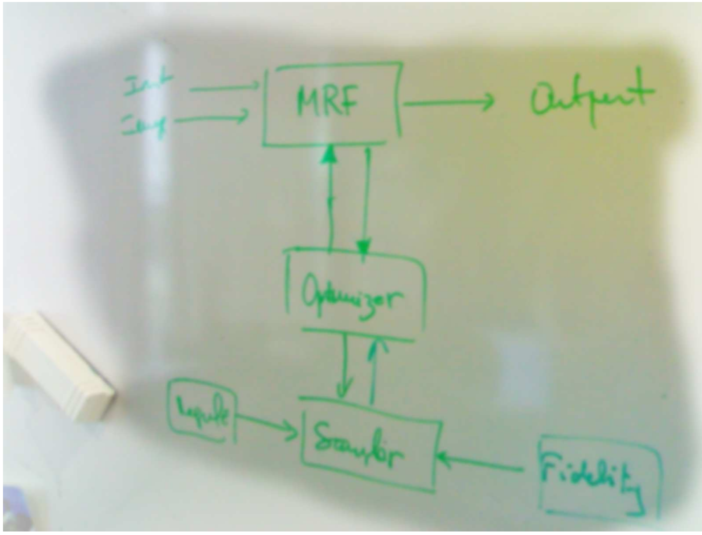


Figure 17.7: OTB Markov Framework.

```
#include "otbMRFEnergyPotts.h"
#include "otbMRFEnergyGaussianClassification.h"
#include "otbMRFOptimizerMetropolis.h"
#include "otbMRFSamplerRandom.h"
```

Then we must decide what pixel type to use for the image. We choose to make all computations with double precision. The labelled image is of type unsigned char which allows up to 256 different classes.

```
  const unsigned int Dimension = 2;

  typedef double InternalPixelType;
  typedef unsigned char LabelledPixelType;
  typedef otb::Image<InternalPixelType, Dimension>  InputImageType;
  typedef otb::Image<LabelledPixelType, Dimension>    LabelledImageType;
```

We define a reader for the image to be classified, an initialisation for the classification (which could be random) and a writer for the final classification.

```
  typedef otb::ImageFileReader< InputImageType >  ReaderType;
  typedef otb::ImageFileWriter< LabelledImageType >  WriterType;

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

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

const char * inputFilename  = argv[1];
const char * outputFilename = argv[2];

reader->SetFileName( inputFilename );
writer->SetFileName( outputFilename );
```

Finally, we define the different classes necessary for the Markov classification. A `otb::MarkovRandomFieldFilter` is instanciated, this is the main class which connect the other to do the Markov classification.

```
typedef otb::MarkovRandomFieldFilter
<InputImageType,LabelledImageType> MarkovRandomFieldFilterType;
```

An `otb::MRFSamplerRandomMAP`, which derives from the `otb::MRFSampler`, is instanciated. The sampler is in charge of proposing a modification for a given site. The `otb::MRFSamplerRandomMAP`, randomly pick one possible value according to the MAP probability.

```
typedef otb::MRFSamplerRandom< InputImageType, LabelledImageType> SamplerType;
```

An `otb::MRFOptimizerMetropoli`, which derives from the `otb::MRFOptimizer`, is instanciated. The optimizer is in charge of accepting or rejecting the value proposed by the sampler. The `otb::MRFSamplerRandomMAP`, accept the proposal according to the variation of energy it causes and a temperature parameter.

```
typedef otb::MRFOptimizerMetropolis OptimizerType;
```

Two energy, deriving from the `otb::MRFEnergy` class need to be instanciated. One energy is required for the regularization, taking into account the relashionship between neighborhing pixels in the classified image. Here it is done with the `otb::MRFEnergyPotts` which implement a Potts model.

The second energy is for the fidelity to the original data. Here it is done with an `otb::MRFEnergyGaussianClassification` class, which defines a gaussian model for the data.

```
typedef otb::MRFEnergyPotts
<LabelledImageType, LabelledImageType>  EnergyRegularizationType;
```

```
typedef otb::MRFEnergyGaussianClassification
<InputImageType, LabelledImageType>  EnergyFidelityType;
```

The different filters composing our pipeline are created by invoking their `New()` methods, assigning the results to smart pointers.

```
MarkovRandomFieldFilterType::Pointer markovFilter = MarkovRandomFieldFilterType::New();
EnergyRegularizationType::Pointer energyRegularization = EnergyRegularizationType::New();
EnergyFidelityType::Pointer energyFidelity = EnergyFidelityType::New();
OptimizerType::Pointer optimizer = OptimizerType::New();
SamplerType::Pointer sampler = SamplerType::New();
```

Parameter for the `otb::MRFEnergyGaussianClassification` class, meand and standard deviation are created.

```
unsigned int nClass = 4;
energyFidelity->SetNumberOfParameters(2*nClass);
EnergyFidelityType::ParametersType parameters;
parameters.SetSize(energyFidelity->GetNumberOfParameters());
parameters[0]=10.0; //Class 0 mean
parameters[1]=10.0; //Class 0 stdev
parameters[2]=80.0;//Class 1 mean
parameters[3]=10.0; //Class 1 stdev
parameters[4]=150.0; //Class 2 mean
parameters[5]=10.0; //Class 2 stdev
parameters[6]=220.0;//Class 3 mean
parameters[7]=10.0; //Class 3 stde
energyFidelity->SetParameters(parameters);
```

Parameters are given to the different class an the sampler, optimizer and energies are connected with the Markov filter.

```
OptimizerType::ParametersType param(1);
param.Fill(atof(argv[5]));
optimizer->SetParameters(param);
markovFilter->SetNumberOfClasses(nClass);
markovFilter->SetMaximumNumberOfIterations(atoi(argv[4]));
markovFilter->SetErrorTolerance(0.0);
markovFilter->SetLambda(atof(argv[3]));
markovFilter->SetNeighborhoodRadius(1);
```

```
markovFilter->SetEnergyRegularization(energyRegularization);
markovFilter->SetEnergyFidelity(energyFidelity);
markovFilter->SetOptimizer(optimizer);
markovFilter->SetSampler(sampler);
```

The pipeline is connected. An `itk::RescaleIntensityImageFilter` rescale the classified image before saving it.

```
markovFilter->SetInput(reader->GetOutput());

typedef itk::RescaleIntensityImageFilter
    < LabelledImageType, LabelledImageType > RescaleType;
RescaleType::Pointer rescaleFilter = RescaleType::New();
rescaleFilter->SetOutputMinimum(0);
rescaleFilter->SetOutputMaximum(255);

rescaleFilter->SetInput( markovFilter->GetOutput() );

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

Finally, the pipeline execution is trigerred.

```
writer->Update();
```

Figure 17.8 shows the output of the Markov Random Field classification after 20 iterations with a random sampler and a Metropolis optimizer.

The source code for this example can be found in the file
`Examples/Markov/MarkovClassification2Example.cxx`.

Using a similar structure as the previous program and the same energy function, we are now going to slightly alter the program to use a different sampler and optimizer. The proposed sample is proposed randomly according to the MAP probability and the optimizer is the ICM which accept the proposed sample if it enable a reduction of the energy.

First, we need to include header specific to these class:

```
#include "otbMRFSamplerRandomMAP.h"
#include "otbMRFOptimizerICM.h"
```

And to declare these new type:

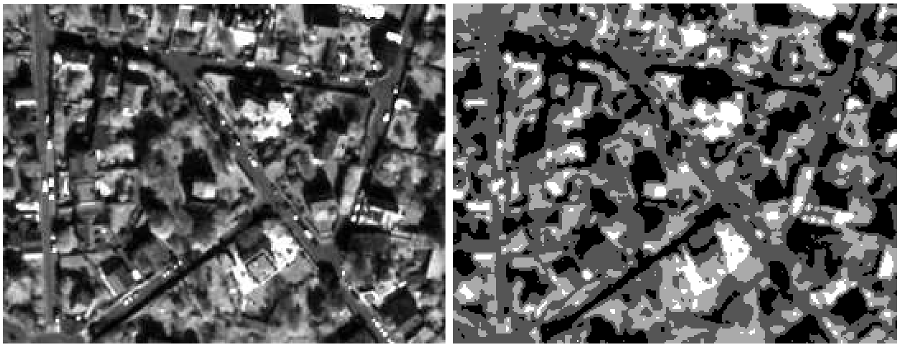


Figure 17.8: Result of applying the otb::MarkovRandomFieldFilter to an extract from a PAN Quickbird image for classification. The result is obtained after 20 iterations with a random sampler and a Metropolis optimizer. From left to right : original image, classification.

```
  typedef otb::MRFSamplerRandomMAP< InputImageType, LabelledImageType> SamplerType;
//   typedef otb::MRFSamplerRandom< InputImageType, LabelledImageType> SamplerType;
```

```
  typedef otb::MRFOptimizerICM OptimizerType;
```

As the otb::MRFOptimizerICM does not have any parameters, the call to optimizer->SetParameters() must be removed

Apart from these, no further modification is required.

Figure 17.9 shows the output of the Markov Random Field classification after 5 iterations with a MAP random sampler and an ICM optimizer.

The source code for this example can be found in the file
Examples/Markov/MarkovRegularizationExample.cxx.

This example illustrates the use of the otb::MarkovRandomFieldFilter. to regularize a classification obtained previously by another classifier. Here we will apply the regularization to the output of an SVM classifier presented in 17.3.5.

The reference image and the starting image are both going to be the original classification. Both regularization and fidelity energy are defined by Potts model.

The convergence of the Markov Random Field is done with a random sampler and a Metropolis model as in example 1. As you should get use to the general program structure to use the MRF framework, we are not going to repeat the entire example. However, remember you can find the full source code for this example in your OTB source directory.

To find the number of classes available in the original image we use the itk::LabelStatisticsImageFilter and more particularly the method

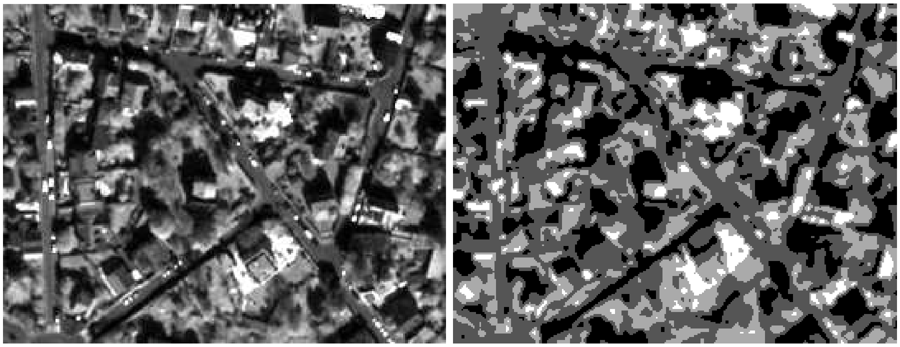


Figure 17.9: Result of applying the `otb::MarkovRandomFieldFilter` to an extract from a PAN Quickbird image for classification. The result is obtained after 5 iterations with a MAP random sampler and an ICM optimizer. From left to right : original image, classification.

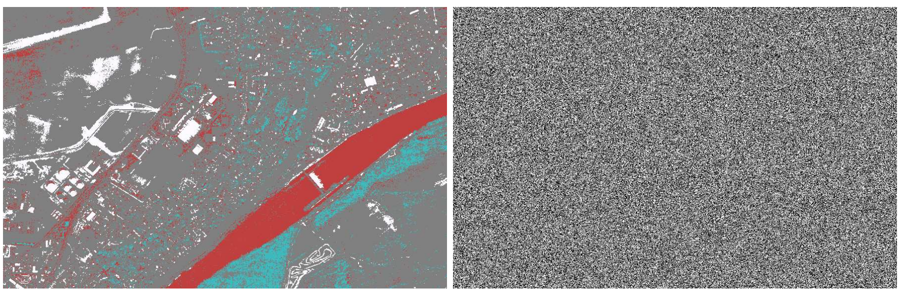


Figure 17.10: Result of applying the `otb::MarkovRandomFieldFilter` to regularized the result of another classification. From left to right : original classification, regularized classification

`GetNumberOfLabels()`.

```
typedef itk::LabelStatisticsImageFilter
    <LabelledImageType, LabelledImageType> LabelledStatType;
LabelledStatType::Pointer labelledStat = LabelledStatType::New();
labelledStat->SetInput(reader->GetOutput());
labelledStat->SetLabelInput(reader->GetOutput());
labelledStat->Update();

unsigned int nClass = labelledStat->GetNumberOfLabels();
```

Figure 17.10 shows the output of the Markov Random Field regularization on the classification output of another method.

17.2 Statistical Segmentations

17.2.1 Stochastic Expectation Maximization

The Stochastic Expectation Maximization (SEM) approach is a stochastic version of the EM mixture estimation seen on section 17.1.4. It has been introduced by [14] to prevent convergence of the EM approach from local minima. It avoids the analytical maximization issued by integrating a stochastic sampling procedure in the estimation process. It induces an almost sure (a.s.) convergence to the algorithm.

From the initial two step formulation of the EM mixture estimation, the SEM may be decomposed into 3 steps:

1. **E-step**, calculates the expected membership values for each measurement vector to each classes.
2. **S-step**, performs a stochastic sampling of the membership vector to each classes, according to the membership values computed in the E-step.
3. **M-step**, updates the parameters of the membership probabilities (parameters to be defined through the class `itk::Statistics::ModelComponentBase` and its inherited classes).

The implementation of the SEM has been turned to a contextual SEM in the sense where the evaluation of the membership parameters is conditioned to membership values of the spatial neighborhood of each pixels.

The source code for this example can be found in the file
`Examples/Learning/SEMModelEstimatorExample.cxx`.

In this example, we present OTB's implementation of SEM, through the class `otb::SEMClassifier`. This class performs a stochastic version of the EM algorithm, but instead of inheriting from `itk::ExpectationMaximizationMixtureModelEstimator`, we choosed to inherit from `itk::Statistics::ListSample< TSample >`, in the same way as `otb::SVMClassifier`.

The program begins with `otb::VectorImage` and outputs `itb::Image`. Then appropriate header files have to be included:

```
#include "otbImage.h"
#include "otbVectorImage.h"
#include "otbImageFileReader.h"
#include "otbImageFileWriter.h"
```

`otb::SEMClassifier` performs estimation of mixture to fit the initial histogram. Actually, mixture of Gaussian pdf can be performed. Those generic pdf are treated in `otb::Statistics::ModelComponentBase`. The Gaussian model is taken in charge with the class `otb::Statistics::GaussianModelComponent`.

```
#include "otbGaussianModelComponent.h"
#include "otbSEMClassifier.h"
```

Input/Output images type are define in a classical way. In fact, a `itk::VariableLengthVector` is to be considered for the templated `MeasurementVectorType`, which will be used in the `ListSample` interface.

```
typedef double PixelType;

typedef otb::VectorImage< PixelType, 2 > ImageType;
typedef otb::ImageFileReader< ImageType > ReaderType;

typedef itk::Image< unsigned char, 2 > OutputImageType;
typedef otb::ImageFileWriter< OutputImageType > WriterType;
```

Once the input image is opened, the classifier may be initialised by `SmartPointer`.

```
typedef otb::SEMClassifier< ImageType, OutputImageType > ClassifType;
ClassifType::Pointer classifier = ClassifType::New();
```

Then, it follows, classical initialisations of the pipeline.

```
classifier->SetNumberOfClasses( numberOfClasses ) ;
classifier->SetMaximumIteration( numberOfIteration );
classifier->SetNeighborhood( neighborhood );
classifier->SetTerminationThreshold( terminationThreshold );
classifier->SetSample( reader->GetOutput() );
```

When an initial segmentation is available, the classifier may use it as image (of type `OutputImageType`) or as a `itk::SampleClassifier` result (of type `itk::Statistics::MembershipSample< SampleType >`).

```
if ( fileNameImgInit != NULL )
{
  typedef otb::ImageFileReader< OutputImageType > ImgInitReaderType;
  ImgInitReaderType::Pointer segReader = ImgInitReaderType::New();
  segReader->SetFileName( fileNameImgInit );
  segReader->Update();
  classifier->SetClassLabels( segReader->GetOutput() );
}
```

By default, `otb::SEMClassifier` performs initialisation of `ModelComponentBase` by as many instanciation of `otb::Statistics::GaussianModelComponent` as the number of classes to estimate in the mixture. Nevertheless, the user may add specific distribution into the mixture estimation. It is permited by the use of `AddComponent` for the given class number and the specific distribution.

```
typedef ClassifType::ClassSampleType ClassSampleType;
typedef otb::Statistics::GaussianModelComponent< ClassSampleType >
    GaussianType;

for ( int i = 0; i < numberOfClasses; i++ )
  classifier->AddComponent( i, GaussianType::New() );
```

Once the pipeline is instanciated. The segmentation by itself may be launched by using the `Update` function.

```
try {
  classifier->Update() ;
}
```

The segmentation may outputs a result of type `itk::Statistics::MembershipSample< SampleType >` as it is the case for the `otb::SVMClassifier`. But when using `GetOutputImage` the output is directly an Image.

Only for visualization purposes, we choose to rescale the image of classes before saving it to a file. We will use the `itk::RescaleIntensityImageFilter` for this purpose.

```
typedef itk::RescaleIntensityImageFilter< OutputImageType,
  OutputImageType > RescalerType;
RescalerType::Pointer rescaler = RescalerType::New();

rescaler->SetOutputMinimum( itk::NumericTraits< unsigned char >::min());
rescaler->SetOutputMaximum( itk::NumericTraits< unsigned char >::max());

rescaler->SetInput( classifier->GetOutputImage() );

WriterType::Pointer writer = WriterType::New();
writer->SetFileName( fileNameOut );
writer->SetInput( rescaler->GetOutput() );
writer->Update();
```

Figure 17.11 shows the result of the SEM segmentation with 4 different classes and a contextual neighborhood of 3 pixels.

As soon as the segmentation is performed by an iterative stochastic process, it is worth verifying the output status: does the segmentation ends when it has converged or just at the limit of the iteration numbers.

```
std::cerr << "Program terminated with a ";
if ( classifier->GetTerminationCode() == ClassifType::CONVERGED )
  std::cerr << "converged ";
else
  std::cerr << "not-converged ";
std::cerr << "code...\n";
```

Figure 17.11: SEM Classification results.

The text output gives for each class the parameters of the pdf (e.g. mean of each component of the class and there covariance matrix, in the case of a Gaussian mixture model).

```
classifier->Print( std::cerr );
```

17.3 Support Vector Machines

Kernel based learning methods in general and the Support Vector Machines (SVM) in particular, have been introduced in the last years in learning theory for classification and regression tasks, [88]. SVM have been successfully applied to text categorization, [48], and face recognition, [67]. Recently, they have been successfully used for the classification of hyperspectral remote-sensing images, [8].

Simply stated, the approach consists in searching for the separating surface between 2 classes by the determination of the subset of training samples which best describes the boundary between the 2 classes. These samples are called support vectors and completely define the classification system. In the case where the two classes are nonlinearly separable, the method uses a kernel expansion in order to make projections of the feature space onto higher dimensionality spaces where the separation of the classes becomes linear.

17.3.1 Mathematical formulation

This section reminds the basic principles of SVM learning and classification. A good tutorial on SVM can be found in, [10].

We have N samples represented by the couple $(y_i, \mathbf{x}_i), i = 1 \ldots N$ where $y_i \in \{-1, +1\}$ is the class label and $\mathbf{x}_i \in \mathbb{R}^n$ is the feature vector of dimension n. A classifier is a function

$$f(\mathbf{x}, \alpha) : \mathbf{x} \mapsto y$$

where α are the classifier parameters. The SVM finds the optimal separating hyperplane which fulfills the following constraints :

- The samples with labels $+1$ and -1 are on different sides of the hyperplane.
- The distance of the closest vectors to the hyperplane is maximised. These are the support vectors (SV) and this distance is called the margin.

The separating hyperplane has the equation

$$\mathbf{w} \cdot \mathbf{x} + b = 0;$$

with $\mathbf{w}$ being its normal vector and x being any point of the hyperplane. The orthogonal distance to the origin is given by $\frac{|b|}{\|\mathbf{w}\|}$. Vectors located outside the hyperplane have either $\mathbf{w} \cdot \mathbf{x} + b > 0$ or $\mathbf{w} \cdot \mathbf{x} + b < 0$.

Therefore, the classifier function can be written as

$$f(\mathbf{x}, \mathbf{w}, b) = sgn(\mathbf{w} \cdot \mathbf{x} + b).$$

The SVs are placed on two hyperplanes which are parallel to the optimal separating one. In order to find the optimal hyperplane, one sets $\mathbf{w}$ and b :

$$\mathbf{w} \cdot \mathbf{x} + b = \pm 1.$$

Since there must not be any vector inside the margin, the following constraint can be used:

$$\mathbf{w} \cdot \mathbf{x}_i + b \geq +1 \text{ if } y_i = +1;$$

$$\mathbf{w} \cdot \mathbf{x}_i + b \leq -1 \text{ if } y_i = -1;$$

which can be rewritten as

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1 \geq 0 \quad \forall i.$$

The orthogonal distances of the 2 parallel hyperplanes to the origin are $\frac{|1-b|}{\|\mathbf{w}\|}$ and $\frac{|-1-b|}{\|\mathbf{w}\|}$. Therefore the modulus of the margin is equal to $\frac{2}{\|\mathbf{w}\|}$ and it has to be maximised.

Thus, the problem to be solved is:

- Find $\mathbf{w}$ and b which minimise $\left\{\frac{1}{2}\|\mathbf{w}\|^2\right\}$
- under the constraint : $y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 \quad i = 1 \ldots N$.

This problem can be solved by using the Lagrange multipliers with one multiplier per sample. It can be shown that only the support vectors will have a positive Lagrange multiplier.

In the case where the two classes are not exactly linearly separable, one can modify the constraints above by using

$$\mathbf{w}\cdot\mathbf{x}_i + b \geq +1 - \xi_i \text{ if } y_i = +1;$$

$$\mathbf{w}\cdot\mathbf{x}_i + b \leq -1 + \xi_i \text{ if } y_i = -1;$$

$$\xi_i \geq 0 \;\; \forall i.$$

If $\xi_i > 1$, one considers that the sample is wrong. The function which has then to be minimised is $\frac{1}{2}\|\mathbf{w}\|^2 + C\left(\sum_i \xi_i\right);$, where C is a tolerance parameter. The optimisation problem is the same than in the linear case, but one multiplier has to be added for each new constraint $\xi_i \geq 0$.

If the decision surface needs to be non-linear, this solution cannot be applied and the kernel approach has to be adopted.

One drawback of the SVM is that, in their basic version, they can only solve two-class problems. Some works exist in the field of multi-class SVM (see [2, 91], and the comparison made by [41]), but they are not used in our system.

For problems with $N > 2$ classes, one can choose either to train N SVM (one class against all the others), or to train $N \times (N-1)$ SVM (one class against each of the others). In the second approach, which is the one that we use, the final decision is taken by choosing the class which is most often selected by the whole set of SVM.

17.3.2 Learning With PointSets

The source code for this example can be found in the file
`Examples/Learning/SVMPointSetModelEstimatorExample.cxx`.

This example illustrates the use of the `otb::SVMPointSetModelEstimator` in order to perform the SVM learning from an `itk::PointSet` data structure.

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

```
#include "otbSVMPointSetModelEstimator.h"
```

In the framework of supervised learning and classification, we will always use feature vectors for the characterization of the classes. On the other hand, the class labels are scalar values. Here, we start by defining the type of the features as the `PixelType`, which will be used to define the feature `VectorType`. We also declare the type for the labels.

```
  typedef float                          PixelType;
  typedef std::vector<PixelType>         VectorType;
  typedef int                            LabelPixelType;
```

We can now proceed to define the point sets used for storing the features and the labels.

```
typedef itk::PointSet< VectorType,  Dimension >      FeaturePointSetType;

typedef itk::PointSet< LabelPixelType,  Dimension >  LabelPointSetType;


FeaturePointSetType::Pointer fPSet = FeaturePointSetType::New();
LabelPointSetType::Pointer   lPSet = LabelPointSetType::New();
```

We will need to get access to the data stored in the point sets, so we define the appropriate for the points and the points containers used by the point sets (see the section 5.2 for more information oin haw to use point sets).

```
typedef FeaturePointSetType::PointType    FeaturePointType;
typedef LabelPointSetType::PointType      LabelPointType;

typedef FeaturePointSetType::PointsContainer    FeaturePointsContainer;
typedef LabelPointSetType::PointsContainer      LabelPointsContainer;

FeaturePointsContainer::Pointer fCont = FeaturePointsContainer::New();
LabelPointsContainer::Pointer lCont = LabelPointsContainer::New();
```

We need now to build the training set for the SVM learning. In this simple example, we will build a SVM who classes points depending on which side of the line $x = y$ they are located. We start by generating 500 random points.

```
int lowest = 0;
int range = 1000;

for(unsigned int pointId = 0; pointId<500; pointId++)
  {

  FeaturePointType fP;
  LabelPointType lP;

  int x_coord = lowest+static_cast<int>(range*(rand()/(RAND_MAX + 1.0)));
  int y_coord = lowest+static_cast<int>(range*(rand()/(RAND_MAX + 1.0)));
```

We set the coordinates of the points. They are the same for the feature vector and for the label.

```
  fP[0] = x_coord;
  fP[1] = y_coord;

  lP[0] = x_coord;
  lP[1] = y_coord;
```

We push the features in the vector after a normalization which is useful for SVM convergence.

```
VectorType feature;
feature.push_back(static_cast<PixelType>((x_coord*1.0-lowest)/range));
feature.push_back(static_cast<PixelType>((y_coord*1.0-lowest)/range));
```

We decide on the label for each point.

```
LabelPixelType label;

if(x_coord < y_coord)
  label= -1;
else
  label = 1;
```

And we insert the points in the points containers.

```
fCont->InsertElement( pointId , fP );
fPSet->SetPointData( pointId, feature );


lCont->InsertElement( pointId , lP );
lPSet->SetPointData( pointId, label );


}
```

After the loop, we set the points containers to the point sets.

```
fPSet->SetPoints( fCont );
lPSet->SetPoints( lCont );
```

Up to now, we have only prepared the data for the SVM learning. We can now create the SVM model estimator. This class is templated over the feature and the label point set types.

```
typedef otb::SVMPointSetModelEstimator< FeaturePointSetType,
                                      LabelPointSetType >   EstimatorType;


EstimatorType::Pointer estimator = EstimatorType::New();
```

The next step consists in setting the point sets for the estimator and the number of classes for the model. The feture point set is set using the `SetInputPointSet` and the label point set is set with the `SetTrainingPointSet` method.

```
estimator->SetInputPointSet( fPSet );
estimator->SetTrainingPointSet( lPSet );
estimator->SetNumberOfClasses( 2 );
```

The model estimation is triggered by calling the `Update` method.

```
estimator->Update();
```

Finally, we can save the result of the learning to a file.

```
estimator->SaveModel("svm_model.svm");
```

The `otb::otbSVMModel` class provides several accessors in order to get some information about the result of the learning step. For instance, one can get the number of support vectors kept to define the separation surface by using the `GetNumberOfSupportVectors()`. This can be very useful to detect some kind of overlearning (the number of support vectors is close to the number of examples). One can also get the SVs themselves by calling the `GetSupportVectors()`. The α values for the support vectors can be accessed by using the `GetAlpha()` method. Finally the `Evaluate()` method will return the result of the classification of a sample and the `EvaluateHyperplaneDistance()` will return the distance of the sample to the separating surface (or surfaces in the case of multi-class problems).

17.3.3 PointSet Classification

The source code for this example can be found in the file
`Examples/Learning/SVMPointSetClassificationExample.cxx`.

This example illustrates the use of the `otb::SVMClassifier` class for performing SVM classification on pointsets. The first thing to do is include the header file for the class. Since the `otb::SVMClassifier` takes `itk::ListSample`s as input, the class `itk::PointSetToListAdaptor` is needed.

We start by including the needed header files.

```
#include "itkPointSetToListAdaptor.h"
#include "itkListSample.h"
#include "otbSVMClassifier.h"
```

In the framework of supervised learning and classification, we will always use feature vectors for the characterization of the classes. On the other hand, the class labels are scalar values. Here, we start by defining the type of the features as the `PixelType`, which will be used to define the feature `VectorType`. We also declare the type for the labels.

```
typedef float                              InputPixelType;

typedef std::vector<InputPixelType>        InputVectorType;
typedef int                                LabelPixelType;
```

We can now proceed to define the point sets used for storing the features and the labels.

```
typedef itk::PointSet< InputVectorType,  Dimension >    MeasurePointSetType;

typedef itk::PointSet< LabelPixelType,  Dimension >     LabelPointSetType;
```

We will need to get access to the data stored in the point sets, so we define the appropriate for the points and the points containers used by the point sets (see the section 5.2 for more information on how to use point sets).

```
typedef MeasurePointSetType::PointType    MeasurePointType;
typedef LabelPointSetType::PointType    LabelPointType;

typedef MeasurePointSetType::PointsContainer      MeasurePointsContainer;
typedef LabelPointSetType::PointsContainer      LabelPointsContainer;


MeasurePointSetType::Pointer tPSet = MeasurePointSetType::New();
MeasurePointsContainer::Pointer tCont = MeasurePointsContainer::New();
```

We need now to build the test set for the SVM. In this simple example, we will build a SVM who classes points depending on which side of the line $x = y$ they are located. We start by generating 500 random points.

```
int lowest = 0;
int range = 1000;

for(pointId = 0; pointId<100; pointId++)
  {

  MeasurePointType tP;
```

```
int x_coord = lowest+static_cast<int>(range*(rand()/(RAND_MAX + 1.0)));
int y_coord = lowest+static_cast<int>(range*(rand()/(RAND_MAX + 1.0)));

std::cout << "coords : " << x_coord << " " << y_coord << std::endl;
tP[0] = x_coord;
tP[1] = y_coord;
```

We push the features in the vector after a normalization which is useful for SVM convergence.

```
InputVectorType measure;
measure.push_back(static_cast<InputPixelType>((x_coord*1.0-lowest)/range));
measure.push_back(static_cast<InputPixelType>((y_coord*1.0-lowest)/range));
```

And we insert the points in the points container.

```
tCont->InsertElement( pointId , tP );
tPSet->SetPointData( pointId, measure );

}
```

After the loop, we set the points container to the point set.

```
tPSet->SetPoints( tCont );
```

Once the pointset is ready, we must transform it to a sample which is compatible with the classification framework. We will use a `itk::Statistics::PointSetToListAdaptor` for this task. This class is templated over the point set type used for storing the measures.

```
typedef itk::Statistics::PointSetToListAdaptor< MeasurePointSetType >
  SampleType;
  SampleType::Pointer sample = SampleType::New();
```

After instantiation, we can set the point set as an imput of our sample adaptor.

```
sample->SetPointSet( tPSet );
```

Now, we need to declare the SVM model which is to be used by the classifier. The SVM model is templated over the type of value used for the measures and the type of pixel used for the labels.

```
typedef otb::SVMModel< SampleType::MeasurementVectorType::ValueType,
                                                  LabelPixelType > ModelType;

ModelType::Pointer model = ModelType::New();
```

After instantiation, we can load a model saved to a file (see section 17.3.2 for an example of model estimation and storage to a file).

```
model->LoadModel(argv[1]);
```

We have now all the elements to create a classifier. The classifier is templated over the sample type (the type of the data to be classified) and the label type (the type of the output of the classifier).

```
typedef otb::SVMClassifier< SampleType, LabelPixelType > ClassifierType ;

ClassifierType::Pointer classifier = ClassifierType::New() ;
```

We set the classifier parameters : number of classes, SVM model, the sample data. And we trigger the classification process by calling the `Update` method.

```
int numberOfClasses = model->GetNumberOfClasses();
classifier->SetNumberOfClasses(numberOfClasses) ;
classifier->SetModel( model );
classifier->SetSample(sample.GetPointer()) ;
classifier->Update() ;
```

After the classification step, we usually want to get the results. The classifier gives an output under the form of a sample list. This list supports the classical STL iterators.

```
    ClassifierType::OutputType* membershipSample =
      classifier->GetOutput() ;

    ClassifierType::OutputType::ConstIterator m_iter =
      membershipSample->Begin() ;
    ClassifierType::OutputType::ConstIterator m_last =
      membershipSample->End() ;
```

We will iterate through the list, get the labels and compute the classification error.

```
    double error = 0.0;
    pointId = 0;
    while (m_iter != m_last)
      {
```

We get the label for each point.

```
      ClassifierType::ClassLabelType label = m_iter.GetClassLabel();
```

And we compare it to the corresponding one of the test set.

```
      InputVectorType measure;

      tPSet->GetPointData(pointId, &measure);

      ClassifierType::ClassLabelType expectedLabel;
      if(measure[0] < measure[1])
expectedLabel= -1;
      else
expectedLabel = 1;

      double dist = fabs(measure[0] - measure[1]);

      if(label != expectedLabel )
error++;

      std::cout << int(label) << "/" << int(expectedLabel) << " --- " << dist << std::endl;
```

Figure 17.12: Images used for the estimation of the SVM model. Left: RGB image. Right: image of labels.

```
    ++pointId;
    ++m_iter ;
    }

  std::cout << "Error = " << error/pointId << " % " << std::endl;
```

17.3.4 Learning With Images

The source code for this example can be found in the file
`Examples/Learning/SVMImageModelEstimatorExample.cxx`.

This example illustrates the use of the `otb::SVMImageModelEstimator` class. This class allows the estimation of a SVM model (supervised learning) from a feature image and an image of labels. In this example, we will train an SVM to separate between water and non-water pixels by using the RGB values only. The images used for this example are shown in figure 17.12. The first thing to do is include the header file for the class.

```
#include "otbSVMImageModelEstimator.h"
```

We define the types for the input and training images. Even if the input image will be an RGB image, we can read it as a 3 component vector image. This simplifies the interfacing with OTB's SVM framework.

```
  typedef unsigned char      InputPixelType;
  const   unsigned int       Dimension = 2;

  typedef otb::VectorImage< InputPixelType,  Dimension >        InputImageType;

  typedef otb::Image< InputPixelType,  Dimension >     TrainingImageType;
```

The `otb::SVMImageModelEstimator` class is templated over the input (features) and the training (labels) images.

```
typedef otb::SVMImageModelEstimator< InputImageType,
                                     TrainingImageType >   EstimatorType;
```

As usual, we define the readers for the images.

```
typedef otb::ImageFileReader< InputImageType > InputReaderType;
typedef otb::ImageFileReader< TrainingImageType > TrainingReaderType;

InputReaderType::Pointer inputReader = InputReaderType::New();
TrainingReaderType::Pointer trainingReader = TrainingReaderType::New();
```

We read the images. It is worth to note that, in order to ensure the pipeline coherence, the output of the objects which preceed the model estimator in the pipeline, must be up to date, so we call the corresponding `Update` methods.

```
inputReader->SetFileName( inputImageFileName );
trainingReader->SetFileName( trainingImageFileName );

inputReader->Update();
trainingReader->Update();
```

We can now instantiate the model estimator and set its parameters.

```
EstimatorType::Pointer svmEstimator = EstimatorType::New();

svmEstimator->SetInputImage( inputReader->GetOutput() );
svmEstimator->SetTrainingImage( trainingReader->GetOutput() );
svmEstimator->SetNumberOfClasses( 2 );
```

The model estimation procedure is triggered by calling the estimator's `Update` method.

```
svmEstimator->Update();
```

Finally, the estimated model can be saved to a file for later use.

```
svmEstimator->SaveModel(outputModelFileName);
```

17.3.5 Image Classification

The source code for this example can be found in the file
`Examples/Classification/SVMImageClassificationExample.cxx`.

In previous examples, we have used the `otb::SVMClassifier`, which uses the ITK classification framework. This good for compatibility with the ITK framework, but introduces the limitations of not being able to use streaming and being able to know at compilation time the number of bands of the image to be classified. In OTB we have avoided this limitation by developping the `otb::SVMImageClassificationFilter`. In this example we will illustrate its use. We start by including the appropriate header file.

```
#include "otbSVMImageClassificationFilter.h"
```

We will assume double precision input images and will also define the type for the labeled pixels.

```
  const unsigned int     Dimension = 2;
  typedef double         PixelType;
  typedef unsigned short LabeledPixelType;
```

Our classifier will be genric enough to be able to process images with any number of bands. We read the images as `otb::VectorImages`. The labeled image will be a scalar image.

```
  typedef otb::VectorImage<PixelType,Dimension> ImageType;
  typedef otb::Image<LabeledPixelType,Dimension> LabeledImageType;
```

We can now define the type for the classifier filter, which is templated over its input and output image types.

```
  typedef otb::SVMImageClassificationFilter<ImageType,LabeledImageType>
                                                             ClassificationFilterType;
  typedef ClassificationFilterType::ModelType ModelType;
```

And finally, we define the reader and the writer. Since the images to classify can be very big, we will use a streamed writer which will trigger the streaming ability of the classifier.

```
  typedef otb::ImageFileReader<ImageType> ReaderType;
  typedef otb::StreamingImageFileWriter<LabeledImageType> WriterType;
```

We instantiate the classifier and the reader objects and we set the existing SVM model obtained in a previous training step.

```
ClassificationFilterType::Pointer filter = ClassificationFilterType::New();

ReaderType::Pointer reader = ReaderType::New();
reader->SetFileName(infname);

ModelType::Pointer model = ModelType::New();
model->LoadModel(modelfname);

filter->SetModel(model);
```

We plug the pipeline and trigger its execution by updating the output of the writer.

```
filter->SetInput(reader->GetOutput());

WriterType::Pointer writer = WriterType::New();
writer->SetInput(filter->GetOutput());
writer->SetFileName(outfname);
writer->Update();
```

The source code for this example can be found in the file
`Examples/Learning/SVMImageEstimatorClassificationMultiExample.cxx`.

This example illustrates the OTB's multi-class SVM capabilities. The theory behind this kind of classification is out of the scope of this guide. In OTB, the multi-class SVM classification is used in the same way as the two-class one. Figure 17.13 shows the image to be classified and the associated ground truth, which is composed of 4 classes.

The following header files are needed for the program:

```
#include "otbSVMImageModelEstimator.h"
#include "itkImageToListAdaptor.h"
#include "itkListSample.h"
#include "otbSVMClassifier.h"
```

We define the types for the input and training images. Even if the input image will be an RGB image, we can read it as a 3 component vector image. This simplifies the interfacing with OTB's SVM framework.

```
  typedef unsigned short      InputPixelType;
  const   unsigned int       Dimension = 2;
```

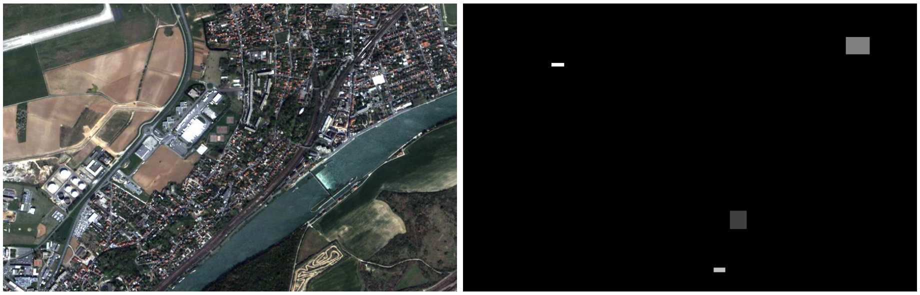


Figure 17.13: Images used for the estimation of the SVM model. Left: RGB image. Right: image of labels.

```
typedef otb::VectorImage< InputPixelType, Dimension >   InputImageType;
```

```
typedef otb::Image< InputPixelType,  Dimension >     TrainingImageType;
```

The `otb::SVMImageModelEstimator` class is templated over the input (features) and the training (labels) images.

```
typedef otb::SVMImageModelEstimator< InputImageType,
                                     TrainingImageType >   EstimatorType;
```

As usual, we define the readers for the images.

```
typedef otb::ImageFileReader< InputImageType > InputReaderType;
typedef otb::ImageFileReader< TrainingImageType > TrainingReaderType;

InputReaderType::Pointer inputReader = InputReaderType::New();
TrainingReaderType::Pointer trainingReader = TrainingReaderType::New();
```

We read the images. It is worth to note that, in order to ensure the pipeline coherence, the output of the objects which preceed the model estimator in the pipeline, must be up to date, so we call the corresponding `Update` methods.

```
inputReader->SetFileName( inputImageFileName );
trainingReader->SetFileName( trainingImageFileName );

inputReader->Update();
trainingReader->Update();
```

We can now instantiate the model estimator and set its parameters.

```
EstimatorType::Pointer svmEstimator = EstimatorType::New();

svmEstimator->SetInputImage( inputReader->GetOutput() );
svmEstimator->SetTrainingImage( trainingReader->GetOutput() );
svmEstimator->SetNumberOfClasses( 4 );
```

The model estimation procedure is triggered by calling the estimator's `Update` method.

```
svmEstimator->Update();
```

We can now proceed to the image classification. We start by declaring the type of the image to be classified. ITK's classification framework needs the type of the pixel to be of fixed type, so we declare the following types.

```
typedef otb::Image< itk::FixedArray<InputPixelType,3>,
                         Dimension >       ClassifyImageType;


typedef otb::ImageFileReader< ClassifyImageType  >         ClassifyReaderType;
```

We can now read the image by calling the `Update` method of the reader.

```
ClassifyReaderType::Pointer cReader = ClassifyReaderType::New();

cReader->SetFileName( inputImageFileName  );

cReader->Update();
```

The image has now to be transformed to a sample which is compatible with the classification framework. We will use a `itk::Statistics::ImageToListAdaptor` for this task. This class is templated over the image type used for storing the measures.

```
typedef itk::Statistics::ImageToListAdaptor< ClassifyImageType > SampleType;
SampleType::Pointer sample = SampleType::New();
```

After instantiation, we can set the image as an imput of our sample adaptor.

```
sample->SetImage(cReader->GetOutput());
```

Now, we need to declare the SVM model which is to be used by the classifier. The SVM model is templated over the type of value used for the measures and the type of pixel used for the labels. The model is obtained from the model estimator by calling the `GetModel` method.

```
typedef InputPixelType LabelPixelType ;

typedef otb::SVMModel< InputPixelType, LabelPixelType > ModelType;

ModelType::Pointer model = svmEstimator->GetModel();
```

We have now all the elements to create a classifier. The classifier is templated over the sample type (the type of the data to be classified) and the label type (the type of the output of the classifier).

```
typedef otb::SVMClassifier< SampleType, LabelPixelType > ClassifierType ;

ClassifierType::Pointer classifier = ClassifierType::New() ;
```

We set the classifier parameters : number of classes, SVM model, the sample data. And we trigger the classification process by calling the `Update` method.

```
int numberOfClasses = model->GetNumberOfClasses();
classifier->SetNumberOfClasses(numberOfClasses) ;
classifier->SetModel( model );
classifier->SetSample(sample.GetPointer()) ;
classifier->Update() ;
```

After the classification step, we usually want to get the results. The classifier gives an output under the form of a sample list. This list supports the classical STL iterators. Therefore, we will create an output image and fill it up with the results of the classification. The pixel type of the output image is the same as the one used for the labels.

```
typedef ClassifierType::ClassLabelType             OutputPixelType;
typedef otb::Image< OutputPixelType, Dimension >        OutputImageType;

OutputImageType::Pointer outputImage = OutputImageType::New();
```

We allocate the memory for the output image using the information from the input image.

```
typedef itk::Index<Dimension>         myIndexType;
typedef itk::Size<Dimension>          mySizeType;
typedef itk::ImageRegion<Dimension>        myRegionType;

mySizeType size;
size[0] = cReader->GetOutput()->GetRequestedRegion().GetSize()[0];
size[1] = cReader->GetOutput()->GetRequestedRegion().GetSize()[1];

myIndexType start;
start[0] = 0;
start[1] = 0;

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

outputImage->SetRegions( region );
outputImage->Allocate();
std::cout << "---" << std::endl;
```

We can now declare the interators on the list that we get at the output of the classifier as well as the iterator to fill the output image.

```
ClassifierType::OutputType* membershipSample =
  classifier->GetOutput() ;
ClassifierType::OutputType::ConstIterator m_iter =
  membershipSample->Begin() ;
ClassifierType::OutputType::ConstIterator m_last =
  membershipSample->End() ;

typedef itk::ImageRegionIterator< OutputImageType>  OutputIteratorType;
```

```
    OutputIteratorType  outIt( outputImage,
   outputImage->GetBufferedRegion() );

    outIt.GoToBegin();
```

We will iterate through the list, get the labels and assign pixel values to the output image.

```
    while (m_iter != m_last && !outIt.IsAtEnd())
    {
    outIt.Set(m_iter.GetClassLabel());
    ++m_iter ;
    ++outIt;
    }
    std::cout << "---" << std::endl;
```

Only for visualization purposes, we choose a color mapping to the image of classes before saving it to a file. 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 for that converts scalar images to RGB images.

```
    typedef itk::RGBPixel<unsigned char>   RGBPixelType;
    typedef otb::Image<RGBPixelType, 2>    RGBImageType;
    typedef itk::Functor::ScalarToRGBPixelFunctor<unsigned long>
      ColorMapFunctorType;
    typedef itk::UnaryFunctorImageFilter<OutputImageType,
    RGBImageType, ColorMapFunctorType> ColorMapFilterType;
    ColorMapFilterType::Pointer colormapper = ColorMapFilterType::New();

    colormapper->SetInput( outputImage );
```

We can now create an image file writer and save the image.

```
    typedef otb::ImageFileWriter<RGBImageType> WriterRescaledType;

    WriterRescaledType::Pointer writerRescaled = WriterRescaledType::New();

    writerRescaled->SetFileName( outputRescaledImageFileName  );
```

Figure 17.14: Result of the SVM classification . Left: RGB image. Right: image of classes.

```
writerRescaled->SetInput( colormapper->GetOutput() );

writerRescaled->Update();
```

Figure 17.14 shows the result of the SVM classification.

17.3.6 Generic Kernel SVM

OTB has developed a specific interface for user-defined kernels. A function $k(\cdot,\cdot)$ is considered to be a kernel when:

$$\begin{aligned} \forall g(\cdot) \in \mathcal{L}^2(\mathbb{R}^n) \quad & \text{so that} \quad \int g(x)^2 dx \text{ be finite,} \\ & \text{then} \quad \int k(x,y)\, g(x)\, g(y)\, dxdy \geqslant 0, \end{aligned} \tag{17.3}$$

which is known as the *Mercer condition*.

When defined through the OTB, a kernel is a class that inherits from `GenericKernelFunctorBase`. Several virtual functions have to be overloaded:

- The `Evaluate` function, which implements the behavior of the kernel itself. For instance, the classical linear kernel could be re-implemented with:

```
double
MyOwnNewKernel
::Evaluate ( const svm_node * x, const svm_node * y,
             const svm_parameter & param ) const
{
        return this->dot(x,y);
}
```

This simple example shows that the classical dot product is already implemented into `otb::GenericKernelFunctorBase::dot()` as a protected function.

- The `Update()` function which synchronizes local variables and their integration into the initial SVM procedure. The following examples will show the way to use it.

Some pre-defined generic kernels have already been implemented in OTB:

- `otb::MixturePolyRBFKernelFunctor` which implements a linear mixture of a polynomial and a RBF kernel;
- `otb::NonGaussianRBFKernelFunctor` which implements a non gaussian RBF kernel;
- `otb::SpectralAngleKernelFunctor`, a kernel that integrates the Spectral Angle, instead of the Euclidean distance, into an inverse multiquadric kernel. This kernel may be appropriated when using multispectral data.
- `otb::ChangeProfileKernelFunctor`, a kernel which is dedicated to the supervized classification of the multiscale change profile presented in section 16.5.1.

Learning with User Defined Kernels

The source code for this example can be found in the file
`Examples/Learning/SVMGenericKernelImageModelEstimatorExample.cxx`.

This example illustrates the modifications to be added to the use of `otb::SVMImageModelEstimator` in order to add a user defined kernel. This initial program has been explained in section 17.3.4.

The first thing to do is to include the header file for the new kernel.

```
#include "otbSVMImageModelEstimator.h"
#include "otbMixturePolyRBFKernelFunctor.h"
```

Once the `otb::SVMImageModelEstimator` is instanciated, it is possible to add the new kernel and its parameters.

Then in addition to the initial code:

```
    EstimatorType::Pointer svmEstimator = EstimatorType::New();

    svmEstimator->SetSVMType( C_SVC );
    svmEstimator->SetInputImage( inputReader->GetOutput() );
    svmEstimator->SetTrainingImage( trainingReader->GetOutput() );
    svmEstimator->SetNumberOfClasses( 4 );
```

The instanciation of the kernel is to be implemented. The kernel which is used here is a linear combination of a polynomial kernel and an RBF one. It is written as

$$\mu k_1(x,y) + (1-\mu)k_2(x,y)$$

with $k_1(x,y) = (\gamma_1 x \cdot y + c_0)^d$ and $k_2(x,y) = \exp\left(-\gamma_2 \|x-y\|^2\right)$. Then, the specific parameters of this kernel are:

- `Mixture` (μ),
- `GammaPoly` (γ_1),
- `CoefPoly` (c_0),
- `DegreePoly` (d),
- `GammaRBF` (γ_2).

Their instanciations are achieved through the use of the `SetValue` function.

```
otb::MixturePolyRBFKernelFunctor myKernel;
myKernel.SetValue( "Mixture", 0.5 );
myKernel.SetValue( "GammaPoly", 1.0 );
myKernel.SetValue( "CoefPoly", 0.0 );
myKernel.SetValue( "DegreePoly", 1 );
myKernel.SetValue( "GammaRBF", 1.5 );
myKernel.Update();
```

Once the kernel's parameters are affected and the kernel updated, the connection to `otb::SVMImageModelEstimator` takes place here.

```
svmEstimator->SetKernelFunctor( &myKernel );
svmEstimator->SetKernelType( GENERIC );
```

The model estimation procedure is triggered by calling the estimator's `Update` method.

```
    svmEstimator->Update();
```

The rest of the code remains unchanged...

```
    svmEstimator->SaveModel(outputModelFileName);
```

In the file `outputModelFileName` a specific line will appear when using a generic kernel. It gives the name of the kernel and its parameters name and value.

Classification with user defined kernel

The source code for this example can be found in the file
`Examples/Learning/SVMGenericKernelImageClassificationExample.cxx`.

This example illustrates the modifications to be added to use the `otb::SVMClassifier` class for performing SVM classification on images with a user-defined kernel. In this example, we will use an SVM model estimated in the previous section to separate between water and non-water pixels by using the RGB values only. The images used for this example are shown in figure 17.12. The first thing to do is include the header file for the class as well as the header of the appropriated kernel to be used.

```
#include "otbSVMClassifier.h"
#include "otbMixturePolyRBFKernelFunctor.h"
```

We need to declare the SVM model which is to be used by the classifier. The SVM model is templated over the type of value used for the measures and the type of pixel used for the labels.

```
    typedef otb::SVMModel< PixelType, LabelPixelType > ModelType;
    ModelType::Pointer model = ModelType::New();
```

After instantiation, we can load a model saved to a file (see section 17.3.4 for an example of model estimation and storage to a file).

When using a user defined kernel, an explicit instanciation has to be performed.

```
    otb::MixturePolyRBFKernelFunctor myKernel;
model->SetKernelFunctor( &myKernel );
```

Then, the rest of the classification program remains unchanged.

```
    model->LoadModel( modelFilename );
```

17.3.7 Multi-band, streamed classification

The source code for this example can be found in the file
`Examples/Classification/SVMImageClassifierExample.cxx`.

In previous examples, we have used the `otb::SVMClassifier`, which uses the ITK classification framework. This good for compatibility with the ITK framework, but introduces the limitations of not being able to use streaming and being able to know at compilation time the number of bands of the image to be classified. In OTB we have avoided this limitation by developping the `otb::SVMImageClassificationFilter`. In this example we will illustrate its use. We start by including the appropriate header file.

```
#include "otbSVMImageClassificationFilter.h"
```

We will assume double precision input images and will also define the type for the labeled pixels.

```
  const unsigned int     Dimension = 2;
  typedef double         PixelType;
  typedef unsigned short LabeledPixelType;
```

Our classifier will be genric enough to be able to process images with any number of bands. We read the images as `otb::VectorImage`s. The labeled image will be a scalar image.

```
  typedef otb::VectorImage<PixelType,Dimension> ImageType;
  typedef otb::Image<LabeledPixelType,Dimension> LabeledImageType;
```

We can now define the type for the classifier filter, which is templated over its input and output image types.

```
  typedef otb::SVMImageClassificationFilter<ImageType,LabeledImageType>
                                              ClassificationFilterType;
  typedef ClassificationFilterType::ModelType ModelType;
```

And finally, we define the reader and the writer. Since the images to classify can be very big, we will use a streamed writer which will trigger the streaming ability of the classifier.

```
  typedef otb::ImageFileReader<ImageType> ReaderType;
  typedef otb::StreamingImageFileWriter<LabeledImageType> WriterType;
```

We instantiate the classifier and the reader objects and we set the existing SVM model obtained in a previous training step.

```
  ClassificationFilterType::Pointer filter = ClassificationFilterType::New();

  ReaderType::Pointer reader = ReaderType::New();
  reader->SetFileName(infname);

  ModelType::Pointer model = ModelType::New();
```

```
model->LoadModel(modelfname);

filter->SetModel(model);
```

We plug the pipeline and trigger its execution by updating the output of the writer.

```
filter->SetInput(reader->GetOutput());

WriterType::Pointer writer = WriterType::New();
writer->SetInput(filter->GetOutput());
writer->SetFileName(outfname);
writer->Update();
```

17.4 Kohonen's Self Organizing Map

The Self Organizing Map, SOM, introduced by Kohonen is a non-supervised neural learning algorithm. The map is composed of neighboring cells which are in competition by means of mutual interactions and they adapt in order to match characteristic patterns of the examples given during the learning. The SOM is usually on a plane (2D).

The algorithm implements a nonlinear projection from a high dimensional feature space to a lower dimension space, usually 2D. It is able to find the correspondence between a set of structured data and a network of much lower dimension while keeping the topological relationships existing in the feature space. Thanks to this topological organization, the final map presents clusters and their relationships.

17.4.1 The algorithm

Kohonen's SOM is usually represented as an array of cells where each cell is, i, associated to a feature (or weight) vector $\underline{m_i} = [m_{i1}, m_{i2}, \cdots, m_{in}]^T \in \mathbb{R}^n$ (figure 17.15).

A cell (or neuron) in the map is a good detector for a given input vector $\underline{x} = [x_1, x_2, \cdots, x_n]^T \in \mathbb{R}^n$ if the latter is *close* to the former. This distance between vectors can be represented by the scalar product $\underline{x}^T \cdot \underline{m_i}$, but for most of the cases other distances can be used, as for instance the Euclidean one. The cell having the weight vector closest to the input vector is called the *winner*.

Learning

The goal of the learning step is to get a map which is representative of an input example set. It is an iterative procedure which consists in passing each input example to the map, testing the response of each neuron and modifying the map to get it closer to the examples.

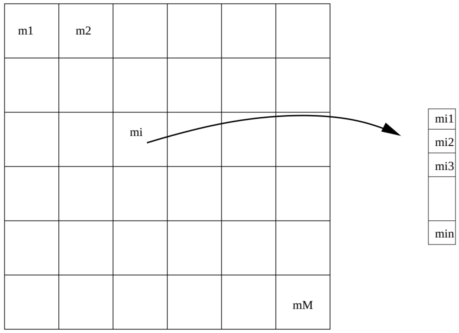


Figure 17.15: Kohonen's Self Organizing Map

Algorithm 1 *SOM learning:*

1. $t = 0$.
2. *Initialize the weight vectors of the map (randomly, for instance).*
3. *While* $t <$ *number of iterations, do:*
   (a) $k = 0$.
   (b) *While* $k <$ *number of examples, do:*
      i. *Find the vector* $\underline{m}_i(t)$ *which minimizes the distance* $d(\underline{x}_k, \underline{m}_i(t))$
      ii. *For a neighborhood* $N_c(t)$ *around the winner cell, apply the transformation:*

$$\underline{m}_i(t+1) = \underline{m}_i(t) + \beta(t)\left[\underline{x}_k(t) - \underline{m}_i(t)\right] \tag{17.4}$$

      iii. $k = k+1$
   (c) $t = t+1$.

In 17.4, $\beta(t)$ is a decreasing function with the geometrical distance to the winner cell. For instance:

$$\beta(t) = \beta_0(t) e^{-\frac{\|\underline{r}_i - \underline{r}_c\|^2}{\sigma^2(t)}}, \tag{17.5}$$

with $\beta_0(t)$ and $\sigma(t)$ decreasing functions with time and $\underline{r}$ the cell coordinates in the output – map – space.

Therefore the algorithm consists in getting the map closer to the learning set. The use of a neighborhood around the winner cell allows the organization of the map into areas which specialize in the recognition of different patterns. This neighborhood also ensures that cells which are topologically close are also close in terms of the distance defined in the feature space.

17.4.2 Building a color table

The source code for this example can be found in the file
`Examples/Learning/SOMExample.cxx`.

This example illustrates the use of the `otb::SOM` class for building Kohonen's Self Organizing Maps.

We will use the SOM in order to build a color table from an input image. Our input image is coded with 3×8 bits and we would like to code it with only 16 levels. We will use the SOM in order to learn which are the 16 most representative RGB values of the input image and we will assume that this is the optimal color table for the image.

The first thing to do is include the header file for the class. We will also need the header files for the map itself and the activation map builder whose utility will be explained at the end of the example.

```
#include "otbSOMMap.h"
#include "otbSOM.h"
#include "otbSOMActivationBuilder.h"
```

Since the `otb::SOM` class uses a distance, we will need to include the header file for the one we want to use

```
#include "itkEuclideanDistance.h"
```

The Self Organizing Map itself is actually an N-dimensional image where each pixel contains a neuron. In our case, we decide to build a 2-dimensional SOM, where the neurons store RGB values with floating point precision.

```
  const unsigned int Dimension = 2;
  typedef double PixelType;
  typedef otb::VectorImage< PixelType, Dimension > ImageType;
  typedef ImageType::PixelType                     VectorType;
```

The distance that we want to apply between the RGB values is the Euclidean one. Of course we could choose to use other type of distance, as for instance, a distance defined in any other color space.

```
  typedef itk::Statistics::EuclideanDistance< VectorType > DistanceType;
```

We can now define the type for the map. The `otb::SOMMap::`class is templated over the neuron type – `PixelType` here –, the distance type and the number of dimensions. Note that the number of dimensions of the map could be different from the one of the images to be processed.

```
typedef otb::SOMMap< VectorType, DistanceType, Dimension > MapType;
```

We are going to perform the learning directly on the pixels of the input image. Therefore, the image type is defined using the same pixel type as we used for the map. We also define the type for the imge file reader.

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

Since the `otb::SOM` class works on lists of samples, it will need to access the input image through an adaptor. Its type is defined as follows:

```
typedef itk::Statistics::ListSample< VectorType > SampleListType;
```

We can now define the type for the SOM, which is templated over the input sample list and the type of the map to be produced and the two functors that hold the training behavior.

```
typedef otb::Functor::CzihoSOMLearningBehaviorFunctor
                                                        LearningBehaviorFunctorType;
typedef otb::Functor::CzihoSOMNeighborhoodBehaviorFunctor
                                                    NeighborhoodBehaviorFunctorType;
typedef otb::SOM< SampleListType, MapType,
                     LearningBehaviorFunctorType, NeighborhoodBehaviorFunctorType >
                                                                            SOMType;
```

As an alternative to standart `SOMType`, one can decide to use an `otb::PeriodicSOM`, which behaves like `otb::SOM` but is to be considered to as a torus instead of a simple map. Hence, the neighborhood behavior of the winning neuron does not depend on its location on the map... `otb::PeriodicSOM` is defined in otbPeriodicSOM.h.

We can now start building the pipeline. The first step is to instantiate the reader and pass its output to the adaptor.

```
ReaderType::Pointer reader = ReaderType::New();
reader->SetFileName(inputFileName);
reader->Update();

SampleListType::Pointer sampleList = SampleListType::New();
```

```
sampleList->SetMeasurementVectorSize( reader->GetOutput()->GetVectorLength() );

itk::ImageRegionIterator< ImageType > imgIter ( reader->GetOutput(),
reader->GetOutput()->GetBufferedRegion() );
imgIter.GoToBegin();

itk::ImageRegionIterator< ImageType > imgIterEnd ( reader->GetOutput(),
   reader->GetOutput()->GetBufferedRegion() );
imgIterEnd.GoToEnd();

do {
  sampleList->PushBack( imgIter.Get() );
  ++imgIter;
} while ( imgIter != imgIterEnd );
```

We can now instantiate the SOM algorithm and set the sample list as input.

```
SOMType::Pointer som = SOMType::New();
som->SetListSample( sampleList );
```

We use a `SOMType::SizeType` array in order to set the sizes of the map.

```
SOMType::SizeType size;
size[0]=sizeX;
size[1]=sizeY;
som->SetMapSize(size);
```

The initial size of the neighborhood of each neuron is set in the same way.

```
SOMType::SizeType radius;
radius[0] = neighInitX;
radius[1] = neighInitY;
som->SetNeighborhoodSizeInit(radius);
```

The other parameters are the number of iterations, the initial and the final values for the learning rate – β – and the maximum initial value for the neurons (the map will be randomly initialized).

Figure 17.16: Result of the SOM learning. Left: RGB image. Center: SOM. Right: Activation map

```
som->SetNumberOfIterations(nbIterations);
som->SetBetaInit(betaInit);
som->SetBetaEnd(betaEnd);
som->SetMaxWeight(static_cast<PixelType>(initValue));
```

Now comes the intialisation of the functors.

```
LearningBehaviorFunctorType learningFunctor;
learningFunctor.SetIterationThreshold( radius, nbIterations );
som->SetBetaFunctor( learningFunctor );

NeighborhoodBehaviorFunctorType neighborFunctor;
som->SetNeighborhoodSizeFunctor( neighborFunctor );
som->Update();
```

Finally, we set up the las part of the pipeline where the plug the output of the SOM into the writer. The learning procedure is triggered by calling the `Update()` method on the writer. Since the map is itself an image, we can write it to disk with an `otb::ImageFileWriter`.

Figure 17.16 shows the result of the SOM learning. Since we have performed a learning on RGB pixel values, the produced SOM can be interpreted as an optimal color table for the input image. It can be observed that the obtained colors are topologically organised, so similar colors are also close in the map. This topological organisation can be exploited to further reduce the number of coding levels of the pixels without performing a new learning: we can subsample the map to get a new color table. Also, a bilinear interpolation between the neurons can be used to increase the number of coding levels.

We can now compute the activation map for the input image. The activation map tells us how many times a given neuron is activated for the set of examples given to the map. The activation map is stored as a scalar image and an integer pixel type is usually enough.

```
typedef unsigned char OutputPixelType;

typedef otb::Image<OutputPixelType,Dimension> OutputImageType;
typedef otb::ImageFileWriter<OutputImageType> ActivationWriterType;
```

In a similar way to the `otb::SOM` class the `otb::SOMActivationBuilder` is templated over the sample list given as input, the SOM map type and the activation map to be built as output.

```
typedef otb::SOMActivationBuilder< SampleListType, MapType,
                                            OutputImageType> SOMActivationBuilderType;
```

We instantiate the activation map builder and set as input the SOM map build before and the image (using the adaptor).

```
SOMActivationBuilderType::Pointer somAct
                                     = SOMActivationBuilderType::New();
somAct->SetInput(som->GetOutput());
somAct->SetListSample( sampleList );
somAct->Update();
```

The final step is to write the activation map to a file.

```
if ( actMapFileName != NULL )
  {
    ActivationWriterType::Pointer actWriter = ActivationWriterType::New();
    actWriter->SetFileName(actMapFileName);
```

The righthand side of figure 17.16 shows the activation map obtained.

17.4.3 SOM Classification

The source code for this example can be found in the file
`Examples/Learning/SOMClassifierExample.cxx`.

This example illustrates the use of the `otb::SOMClassifier` class for performing a classification using an existing Kohonen's Self Organizing. Actually, the SOM classification consists only in the attribution of the winner neuron index to a given feature vector.

We will use the SOM created in section 17.4.2 and we will assume that each neuron represents a class in the image.

The first thing to do is include the header file for the class.

```
#include "otbSOMClassifier.h"
```

As for the SOM learning step, we must define the types for the `otb::SOMMap`, and therefore, also for the distance to be used. We will also define the type for the SOM reader, which is actually an `otb::ImageFileReader::`which the appropiate image type.

```
  typedef itk::Statistics::EuclideanDistance<PixelType> DistanceType;
  typedef otb::SOMMap<PixelType,DistanceType,Dimension> SOMMapType;
  typedef otb::ImageFileReader<SOMMapType> SOMReaderType;
```

The classification will be performed by the `otb::SOMClassifier::`, which, as most of the classifiers, works on `itk::Statistics::ListSample`s. In order to be able to perform an image classification, we will need to use the `itk::Statistics::ImageToListAdaptor` which is templated over the type of image to be adapted. The `SOMClassifier` is templated over the sample type, the SOMMap type and the pixel type for the labels.

```
  typedef itk::Statistics::ListSample< PixelType> SampleType;
  typedef otb::SOMClassifier<SampleType,SOMMapType,LabelPixelType>
                                                           ClassifierType;
```

The result of the classification will be stored on an image and saved to a file. Therefore, we define the types needed for this step.

```
  typedef otb::Image<LabelPixelType, Dimension >  OutputImageType;
  typedef otb::ImageFileWriter<OutputImageType>  WriterType;
```

We can now start reading the input image and the SOM given as inputs to the program. We instantiate the readers as usual.

```
  ReaderType::Pointer reader = ReaderType::New();
  reader->SetFileName( imageFilename  );
  reader->Update();

  SOMReaderType::Pointer somreader = SOMReaderType::New();
  somreader->SetFileName(mapFilename);
  somreader->Update();
```

The conversion of the input data from image to list sample is easily done using the adaptor.

```
  SampleType::Pointer sample = SampleType::New();

  itk::ImageRegionIterator<InputImageType> it(reader->GetOutput(),reader->GetOutput()->GetLarg

    it.GoToBegin();

    while(!it.IsAtEnd())
      {
sample->PushBack(it.Get());
++it;
      }
```

The classifier can now be instantiated. The input data is set by using the `SetSample()` method and the SOM si set using the `SetMap()` method. The classification is triggered by using the `Update()` method.

```
  ClassifierType::Pointer classifier = ClassifierType::New() ;
  classifier->SetSample(sample.GetPointer());
  classifier->SetMap(somreader->GetOutput());
  classifier->Update() ;
```

Once the classification has been performed, the sample list obtained at the output of the classifier must be converted into an image. We create the image as follows :

```
  OutputImageType::Pointer outputImage = OutputImageType::New();
  outputImage->SetRegions( reader->GetOutput()->GetLargestPossibleRegion());
  outputImage->Allocate();
```

We can now get a pointer to the classification result.

```
ClassifierType::OutputType* membershipSample = classifier->GetOutput();
```

And we can declare the iterators pointing to the front and the back of the sample list.

```
ClassifierType::OutputType::ConstIterator m_iter =  membershipSample->Begin();
ClassifierType::OutputType::ConstIterator m_last =  membershipSample->End();
```

We also declare an `itk::ImageRegionIterator::`in order to fill the output image with the class labels.

```
typedef itk::ImageRegionIterator< OutputImageType>  OutputIteratorType;

OutputIteratorType  outIt(outputImage,outputImage->GetLargestPossibleRegion());
```

We iterate through the sample list and the output image and assign the label values to the image pixels.

```
outIt.GoToBegin();

while (m_iter != m_last && !outIt.IsAtEnd())
  {
  outIt.Set(m_iter.GetClassLabel());
  ++m_iter ;
  ++outIt;
  }
```

Finally, we write the classified image to a file.

```
WriterType::Pointer writer = WriterType::New();
writer->SetFileName(outputFilename);
writer->SetInput(outputImage);
writer->Update();
```

Figure 17.17 shows the result of the SOM classification.

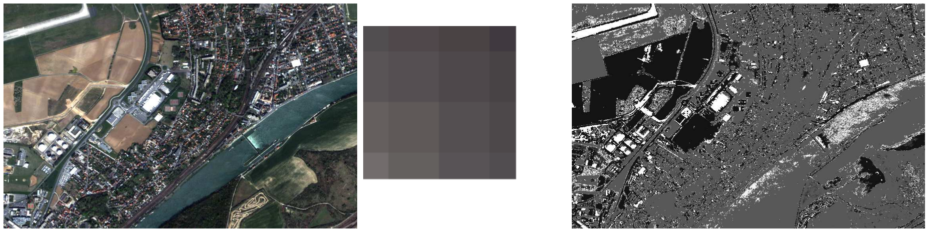


Figure 17.17: Result of the SOM learning. Left: RGB image. Center: SOM. Right: Classified Image

17.4.4 Multi-band, streamed classification

The source code for this example can be found in the file
`Examples/Classification/SOMImageClassificationExample.cxx`.

In previous examples, we have used the `otb::SOMClassifier`, which uses the ITK classification framework. This good for compatibility with the ITK framework, but introduces the limitations of not being able to use streaming and being able to know at compilation time the number of bands of the image to be classified. In OTB we have avoided this limitation by developping the `otb::SOMImageClassificationFilter`. In this example we will illustrate its use. We start by including the appropriate header file.

```
#include "otbSOMImageClassificationFilter.h"
```

We will assume double precision input images and will also define the type for the labeled pixels.

```
  const unsigned int     Dimension = 2;
  typedef double         PixelType;
  typedef unsigned short LabeledPixelType;
```

Our classifier will be genric enough to be able to process images with any number of bands. We read the images as `otb::VectorImages`. The labeled image will be a scalar image.

```
  typedef otb::VectorImage<PixelType,Dimension> ImageType;
  typedef otb::Image<LabeledPixelType,Dimension> LabeledImageType;
```

We can now define the type for the classifier filter, which is templated over its input and output image types and the SOM type.

```
typedef otb::SOMMap<ImageType::PixelType> SOMMapType;
typedef otb::SOMImageClassificationFilter<ImageType,
             LabeledImageType,SOMMapType> ClassificationFilterType;
```

And finally, we define the readers (for the input image and theSOM) and the writer. Since the images, to classify can be very big, we will use a streamed writer which will trigger the streaming ability of the classifier.

```
typedef otb::ImageFileReader<ImageType> ReaderType;
typedef otb::ImageFileReader<SOMMapType> SOMReaderType;
typedef otb::StreamingImageFileWriter<LabeledImageType> WriterType;
```

We instantiate the classifier and the reader objects and we set the existing SOM obtained in a previous training step.

```
ClassificationFilterType::Pointer filter = ClassificationFilterType::New();

ReaderType::Pointer reader = ReaderType::New();
reader->SetFileName(infname);

SOMReaderType::Pointer somreader = SOMReaderType::New();
somreader->SetFileName(somfname);
somreader->Update();

filter->SetMap(somreader->GetOutput());
```

We plug the pipeline and trigger its execution by updating the output of the writer.

```
filter->SetInput(reader->GetOutput());

WriterType::Pointer writer = WriterType::New();
writer->SetInput(filter->GetOutput());
writer->SetFileName(outfname);
writer->Update();
```

CHAPTER

EIGHTEEN

Image Visualization

Even if OTB is not a visualization toolkit as for instance VTK (*The Visualization Toolkit* `http://www.vtk.org`), some simple functionnalities for image visualization are given in the toolbox. Indeed, for algorithm prototyping, it is sometimes more useful to *see* the result on the screen, than saving it to a file and then open in with an external viewer.

OTB provides the `otb::ImageViewer` class which is compatible with the pipeline and can therefore replace the `otb::ImageFileWriter::`during proto-typing phases.

The source code for this example can be found in the file
`Examples/Visu/VisuExample1.cxx`.

This example shows the use of the `otb::ImageViewer` class for image visualization. As usual, we start by including the header file for the class.

```
#include "otbImageViewer.h"
```

We will build a very simple pipeline where a reader gets an image from a file and gives it to the viewer. We define the types for the pixel, the image and the reader. The viewer class is templated over the scalar component of the pixel type.

```
  typedef int PixelType;
  typedef otb::VectorImage< PixelType, 2 > ImageType;
  typedef otb::ImageFileReader< ImageType > ReaderType;
  typedef otb::ImageViewer< PixelType >    ViewerType;
```

We create the objects.

```
  ViewerType::Pointer  lViewer = ViewerType::New();
  ReaderType::Pointer       lReader = ReaderType::New();
  lReader->SetFileName(inputFilename);
  lReader->Update();
```

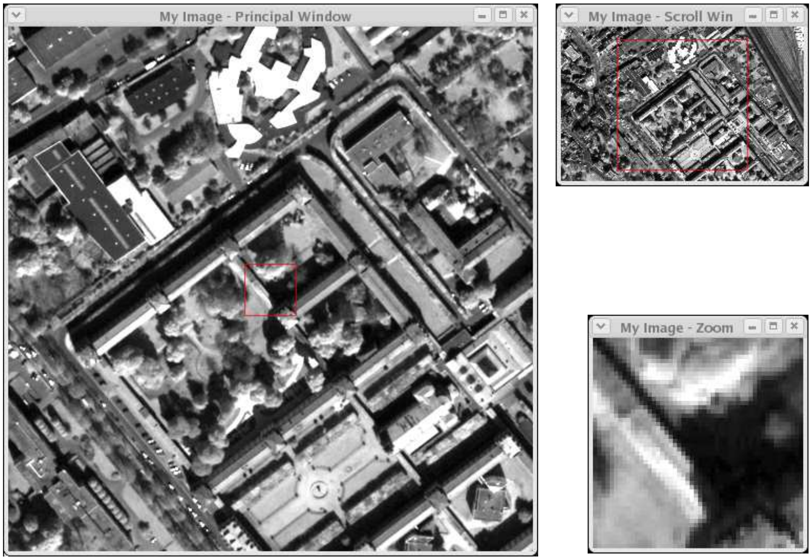


Figure 18.1: Example of image visualization.

We can choose a label for the windows created by the viewer.

```
lViewer->SetLabel( "My Image" );
```

We can now plug the pipeline and trigger the visualization by using the `Show` method.

```
lViewer->SetImage( lReader->GetOutput() );

lViewer->Show();
```

The last step consists in starting the GUI event loop by calling the appropiate FLTK method.

```
Fl::run();
```

The the `otb::ImageViewer` class creates 3 windows (see figure 18.1) for an improved visualization of large images. This procedure is inspired from the navigation window of the Gimp and other image visualization tools. The navigation window is called here *scroll* window and it shows the complete image but subsampled to a lower resolution. The pricipal window shows the region marked by a red rectangle in the scroll window using the real resolution of the image. Finally, a zoom window displays the region inside the red rectangle shown in the principal window. A mouse click on a pixel of the scroll (respectively, the pricipal window) updates the rectangle prosition and, therefore, the region viewed in the principal (respectively, the zoom) window. The zoom rate can be modified by using the mous wheel.

Part IV

Developper's guide

CHAPTER

NINETEEN

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 19.2 describes the programming interface common to most ITK image iterators. Sections 19.3–19.4 document specific ITK iterator types and provide examples of how they are used.

19.1 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 `otb::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.

19.2 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.

19.2.1 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 5.1 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 `otb::Image`.

```
typedef otb::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() );
```

19.2.2 Moving Iterators

An iterator is described as *walking* its iteration region. At any time, the iterator will reference, or "point to", one pixel location in the N-dimensional (ND) image. *Forward iteration* goes from the beginning of the iteration region to the end of the iteration region. *Reverse iteration*, goes from just past the end of the region back to the beginning. There are two corresponding starting positions for iterators, the *begin* position and the *end* position. An iterator can be moved directly to either of these two positions using the following methods.

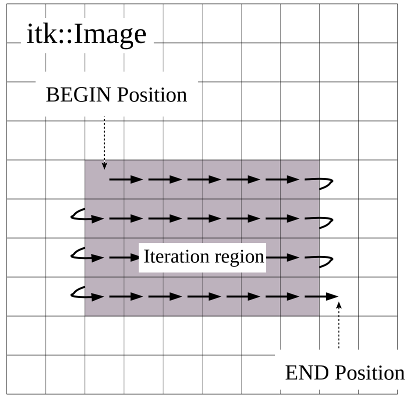


Figure 19.1: Normal path of an iterator through a 2D image. The iteration region is shown in a darker shade. An arrow denotes a single iterator step, the result of one ++ operation.

- **`GoToBegin()`** Points the iterator to the first valid data element in the region.
- **`GoToEnd()`** Points the iterator to *one position past* the last valid element in the region.

Note that the end position is not actually located within the iteration region. This is important to remember because attempting to dereference an iterator at its end position will have undefined results.

ITK iterators are moved back and forth across their iterations using the decrement and increment operators.

- **`operator++()`** Increments the iterator one position in the positive direction. Only the prefix increment operator is defined for ITK image iterators.
- **`operator--()`** Decrements the iterator one position in the negative direction. Only the prefix decrement operator is defined for ITK image iterators.

Figure 19.1 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.

19.2.3 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 20 on page 505 for more information about image adaptors).

19.2.4 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() );
  }
```

19.3 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.

19.3.1 ImageRegionIterator

The source code for this example can be found in the file
`Examples/Iterators/ImageRegionIterator.cxx`.

The `itk::ImageRegionIterator` is optimized for iteration speed and is the first choice for iterative, pixel-wise operations when location in the image is not important. ImageRegionIterator is the least specialized of the ITK image iterator classes. It implements all of the methods described in the preceding section.

The following example illustrates the use of `itk::ImageRegionConstIterator` and ImageRegionIterator. Most of the code constructs introduced apply to other ITK iterators as well. This simple application crops a subregion from an image by copying its pixel values into to a second, smaller image.

We begin by including the appropriate header files.

```
#include "itkImageRegionConstIterator.h"
#include "itkImageRegionIterator.h"
```

Next we define a pixel type and corresponding image type. ITK iterator classes expect the image type as their template parameter.

```
const unsigned int Dimension = 2;

typedef unsigned char PixelType;
```

```
typedef otb::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 5.1).

```
ImageType::RegionType inputRegion;

ImageType::RegionType::IndexType inputStart;
ImageType::RegionType::SizeType  size;

inputStart[0] = ::atoi( argv[3] );
inputStart[1] = ::atoi( argv[4] );

size[0]  = ::atoi( argv[5] );
size[1]  = ::atoi( argv[6] );

inputRegion.SetSize( size );
inputRegion.SetIndex( inputStart );
```

The destination region in the output image is defined using the input region size, but a different start index. The starting index for the destination region is the corner of the newly generated image.

```
ImageType::RegionType outputRegion;

ImageType::RegionType::IndexType outputStart;

outputStart[0] = 0;
outputStart[1] = 0;

outputRegion.SetSize( size );
outputRegion.SetIndex( outputStart );
```

After reading the input image and checking that the desired subregion is, in fact, contained in the input, we allocate an output image. It is fundamental to set valid values to some of the basic image information during the copying process. In particular, the starting index of the output region is now filled up with zero values and the coordinates of the physical origin are computed as a shift from the origin of the input image. This is quite important since it will allow us to later register the extracted region against the original image.

```
ImageType::Pointer outputImage = ImageType::New();
outputImage->SetRegions( outputRegion );
```

```
const ImageType::SpacingType& spacing = reader->GetOutput()->GetSpacing();
const ImageType::PointType& inputOrigin = reader->GetOutput()->GetOrigin();
double   outputOrigin[ Dimension ];

for(unsigned int i=0; i< Dimension; i++)
  {
  outputOrigin[i] = inputOrigin[i] + spacing[i] * inputStart[i];
  }

outputImage->SetSpacing( spacing );
outputImage->SetOrigin(  outputOrigin );
outputImage->Allocate();
```

The necessary images and region definitions are now in place. All that is left to do is to create the iterators and perform the copy. Note that image iterators are not accessed via smart pointers so they are light-weight objects that are instantiated on the stack. Also notice how the input and output iterators are defined over the *same corresponding region*. Though the images are different sizes, they both contain the same target subregion.

```
ConstIteratorType inputIt(   reader->GetOutput(), inputRegion  );
IteratorType      outputIt(  outputImage,         outputRegion );

for ( inputIt.GoToBegin(), outputIt.GoToBegin(); !inputIt.IsAtEnd();
      ++inputIt, ++outputIt)
  {
  outputIt.Set(  inputIt.Get()  );
  }
```

The `for` loop above is a common construct in ITK/OTB. 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 `QB_Suburb.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 QB_Suburb.png ImageRegionIteratorOutput.png 20 70 210 140
```

The output is the cropped subregion shown in Figure 19.2.

19.3.2 ImageRegionIteratorWithIndex

The source code for this example can be found in the file
`Examples/Iterators/ImageRegionIteratorWithIndex.cxx`.

Figure 19.2: Cropping a region from an image. The original image is shown at left. The image on the right is the result of applying the ImageRegionIterator example code.

The “WithIndex” family of iterators was designed for algorithms that use both the value and the location of image pixels in calculations. Unlike `itk::ImageRegionIterator`, which calculates an index only when asked for, `itk::ImageRegionIteratorWithIndex` maintains its index location as a member variable that is updated during the increment or decrement process. Iteration speed is penalized, but the index queries are more efficient.

The following example illustrates the use of ImageRegionIteratorWithIndex. The algorithm mirrors a 2D image across its *x*-axis (see `itk::FlipImageFilter` for an ND version). The algorithm makes extensive use of the `GetIndex()` method.

We start by including the proper header file.

```
#include "itkImageRegionIteratorWithIndex.h"
```

For this example, we will use an RGB pixel type so that we can process color images. Like most other ITK image iterator, ImageRegionIteratorWithIndex class expects the image type as its single template parameter.

```
  const unsigned int Dimension = 2;

  typedef itk::RGBPixel< unsigned char > RGBPixelType;
  typedef otb::Image< RGBPixelType, Dimension >  ImageType;

  typedef itk::ImageRegionIteratorWithIndex< ImageType > IteratorType;
```

An `ImageType` smart pointer called `inputImage` points to the output of the image reader. After updating the image reader, we can allocate an output image of the same size, spacing, and origin as the input image.

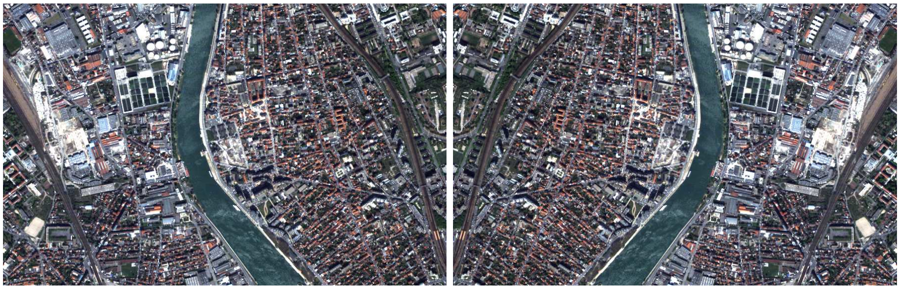


Figure 19.3: Results of using ImageRegionIteratorWithIndex to mirror an image across an axis. The original image is shown at left. The mirrored output is shown at right.

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

Next we create the iterator that walks the output image. This algorithm requires no iterator for the input image.

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

This axis flipping algorithm works by iterating through the output image, querying the iterator for its index, and copying the value from the input at an index mirrored across the *x*-axis.

```
ImageType::IndexType requestedIndex =
              outputImage->GetRequestedRegion().GetIndex();
ImageType::SizeType requestedSize =
              outputImage->GetRequestedRegion().GetSize();

for ( outputIt.GoToBegin(); !outputIt.IsAtEnd(); ++outputIt)
  {
  ImageType::IndexType idx = outputIt.GetIndex();
  idx[0] =  requestedIndex[0] + requestedSize[0] - 1 - idx[0];
  outputIt.Set( inputImage->GetPixel(idx) );
  }
```

Let's run this example on the image `ROI_QB_MUL_2.tif` found in the `Examples/Data` directory. Figure 19.3 shows how the original image has been mirrored across its *x*-axis in the output.

19.3.3 ImageLinearIteratorWithIndex

The source code for this example can be found in the file
`Examples/Iterators/ImageLinearIteratorWithIndex.cxx`.

The `itk::ImageLinearIteratorWithIndex` is designed for line-by-line processing of an image. It walks a linear path along a selected image direction parallel to one of the coordinate axes of the image. This iterator conceptually breaks an image into a set of parallel lines that span the selected image dimension.

Like all image iterators, movement of the ImageLinearIteratorWithIndex is constrained within an image region R. The line ℓ through which the iterator moves is defined by selecting a direction and an origin. The line ℓ extends from the origin to the upper boundary of R. The origin can be moved to any position along the lower boundary of R.

Several additional methods are defined for this iterator to control movement of the iterator along the line ℓ and movement of the origin of ℓ.

- **`NextLine()`** Moves the iterator to the beginning pixel location of the next line in the image. The origin of the next line is determined by incrementing the current origin along the fastest increasing dimension of the subspace of the image that excludes the selected dimension.
- **`PreviousLine()`** Moves the iterator to the *last valid pixel location* in the previous line. The origin of the previous line is determined by decrementing the current origin along the fastest increasing dimension of the subspace of the image that excludes the selected dimension.
- **`GoToBeginOfLine()`** Moves the iterator to the beginning pixel of the current line.
- **`GoToEndOfLine()`** Move the iterator to *one past* the last valid pixel of the current line.
- **`IsAtReverseEndOfLine()`** Returns true if the iterator points to *one position before* the beginning pixel of the current line.
- **`IsAtEndOfLine()`** Returns true if the iterator points to *one position past* the last valid pixel of the current line.

The following code example shows how to use the ImageLinearIteratorWithIndex. It implements the same algorithm as in the previous example, flipping an image across its x-axis. Two line iterators are iterated in opposite directions across the x-axis. After each line is traversed, the iterator origins are stepped along the y-axis to the next line.

Headers for both the const and non-const versions are needed.

```
#include "itkImageLinearConstIteratorWithIndex.h"
#include "itkImageLinearIteratorWithIndex.h"
```

The RGB image and pixel types are defined as in the previous example. The ImageLinearIteratorWithIndex class and its const version each have single template parameters, the image type.

```
typedef itk::ImageLinearIteratorWithIndex< ImageType >       IteratorType;
typedef itk::ImageLinearConstIteratorWithIndex< ImageType >  ConstIteratorType;
```

After reading the input image, we allocate an output image that of the same size, spacing, and origin.

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

Next we create the two iterators. The const iterator walks the input image, and the non-const iterator walks the output image. The iterators are initialized over the same region. The direction of iteration is set to 0, the *x* dimension.

```
ConstIteratorType inputIt( inputImage, inputImage->GetRequestedRegion() );
IteratorType outputIt( outputImage, inputImage->GetRequestedRegion() );

inputIt.SetDirection(0);
outputIt.SetDirection(0);
```

Each line in the input is copied to the output. The input iterator moves forward across columns while the output iterator moves backwards.

```
for ( inputIt.GoToBegin(),  outputIt.GoToBegin(); ! inputIt.IsAtEnd();
      outputIt.NextLine(),  inputIt.NextLine())
  {
  inputIt.GoToBeginOfLine();
  outputIt.GoToEndOfLine();
  --outputIt;
  while ( ! inputIt.IsAtEndOfLine() )
    {
    outputIt.Set( inputIt.Get() );
    ++inputIt;
    --outputIt;
    }
  }
```

Running this example on `ROI_QB_MUL_1.tif` produces the same output image shown in Figure 19.3.

19.4 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

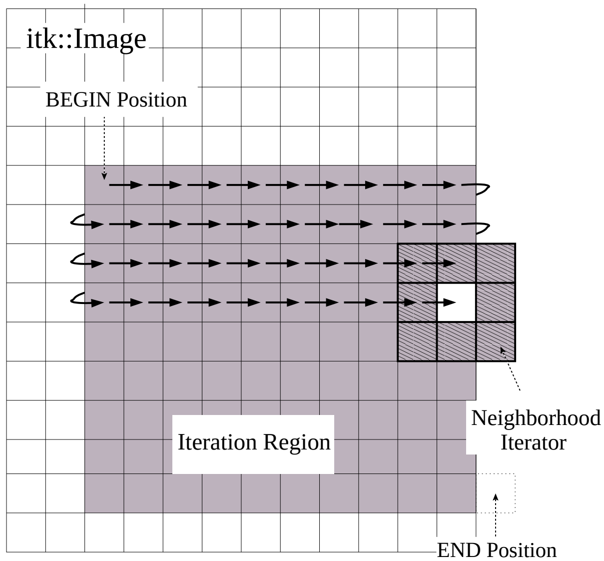


Figure 19.4: Path of a $3x3$ neighborhood iterator through a 2D image region. The extent of the neighborhood is indicated by the hashing around the iterator position. Pixels that lie within this extent are accessible through the iterator. An arrow denotes a single iterator step, the result of one ++ operation.

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

Neighborhood iterators use the same operators defined in Section 19.2 and the same code constructs as normal iterators for looping through an image. Figure 19.4 shows a neighborhood iterator moving through an iteration region. This iterator defines a $3x3$ neighborhood around each pixel that it visits. The *center* of the neighborhood iterator is always positioned over its current index and all other neighborhood pixel indices are referenced as offsets from the center index. The pixel under the center of the neighborhood iterator and all pixels under the shaded area, or *extent*, of the iterator can be dereferenced.

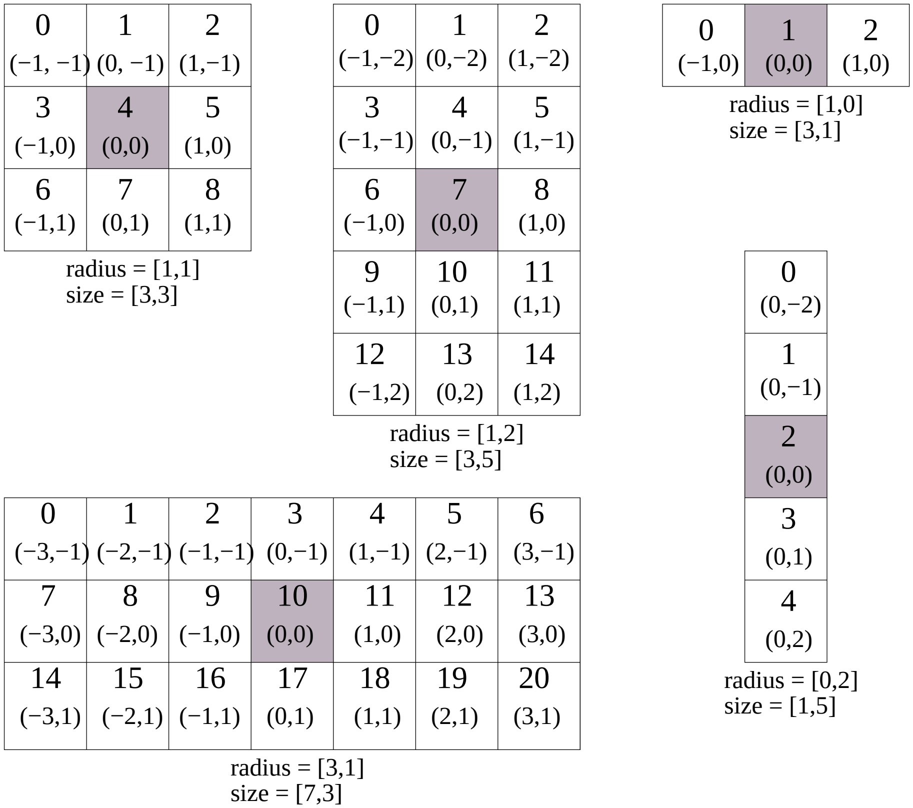


Figure 19.5: Several possible 2D neighborhood iterator shapes are shown along with their radii and sizes. A neighborhood pixel can be dereferenced by its integer index (top) or its offset from the center (bottom). The center pixel of each iterator is shaded.

In addition to the standard image pointer and iteration region (Section 19.2), 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 19.5 shows the relationship between the radius of the iterator and the size of the neighborhood for a variety of 2D iterator shapes.

The radius of the neighborhood iterator is queried after construction by calling the `GetRadius()` method. Some other methods provide some useful information about the iterator and its underlying image.

- **`SizeType GetRadius()`** Returns the ND radius of the neighborhood as an `itk::Size`.
- **`const ImageType *GetImagePointer()`** Returns the pointer to the image referenced by the iterator.
- **`unsigned long Size()`** Returns the size in number of pixels of the neighborhood.

The neighborhood iterator interface extends the normal ITK iterator interface for setting and getting pixel values. One way to dereference pixels is to think of the neighborhood as a linear array where each pixel has a unique integer index. The index of a pixel in the array is determined by incrementing from the upper-left-forward corner of the neighborhood along the fastest increasing image dimension: first column, then row, then slice, and so on. In Figure 19.5, the unique integer index is shown at the top of each pixel. The center pixel is always at position $n/2$, where n is the size of the array.

- **`PixelType GetPixel(const unsigned int i)`** Returns the value of the pixel at neighborhood position `i`.
- **`void SetPixel(const unsigned int i, PixelType p)`** Sets the value of the pixel at position `i` to `p`.

Another way to think about a pixel location in a neighborhood is as an ND offset from the neighborhood center. The upper-left-forward corner of a 3*x*3*x*3 neighborhood, for example, can be described by offset $(-1,-1,-1)$. The bottom-right-back corner of the same neighborhood is at offset $(1,1,1)$. In Figure 19.5, 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 19.4, 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 19.4.1.

- **`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.

19.4.1 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 example can be found in the file
`Examples/Iterators/NeighborhoodIterators1.cxx`.

This example uses the `itk::NeighborhoodIterator` to implement a simple Sobel edge detection algorithm [35]. 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 otb::Image< PixelType, 2 >  ImageType;
typedef otb::ImageFileReader< ImageType > ReaderType;

typedef itk::ConstNeighborhoodIterator< ImageType > NeighborhoodIteratorType;
typedef itk::ImageRegionIterator< ImageType>        IteratorType;
```

The following code creates and executes the OTB image reader. The `Update` call on the reader object is surrounded by the standard `try/catch` blocks to handle any exceptions that may be thrown by the reader.

```
ReaderType::Pointer reader = ReaderType::New();
reader->SetFileName( argv[1] );
try
  {
  reader->Update();
  }
catch ( itk::ExceptionObject &err)
  {
  std::cout << "ExceptionObject caught !" << std::endl;
  std::cout << err << std::endl;
  return -1;
  }
```

We can now create a neighborhood iterator to range over the output of the reader. For Sobel edge-detection in 2D, we need a square iterator that extends one pixel away from the neighborhood center in every dimension.

```
NeighborhoodIteratorType::RadiusType radius;
radius.Fill(1);
NeighborhoodIteratorType it( radius, reader->GetOutput(),
                             reader->GetOutput()->GetRequestedRegion() );
```

The following code creates an output image and iterator.

```
ImageType::Pointer output = ImageType::New();
output->SetRegions(reader->GetOutput()->GetRequestedRegion());
```

```
output->Allocate();

IteratorType out(output, reader->GetOutput()->GetRequestedRegion());
```

Sobel edge detection uses weighted finite difference calculations to construct an edge magnitude image. Normally the edge magnitude is the root sum of squares of partial derivatives in all directions, but for simplicity this example only calculates the x component. The result is a derivative image biased toward maximally vertical edges.

The finite differences are computed from pixels at six locations in the neighborhood. In this example, we use the iterator `GetPixel()` method to query the values from their offsets in the neighborhood. The example in Section 19.4.1 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 otb::Image< WritePixelType, 2 > WriteImageType;
typedef otb::ImageFileWriter< WriteImageType > WriterType;
```

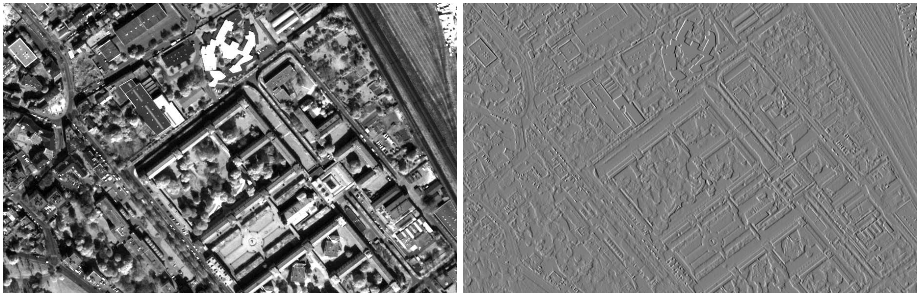


Figure 19.6: Applying the Sobel operator to an image (left) produces x (right) derivative image.

```
typedef itk::RescaleIntensityImageFilter<
            ImageType, WriteImageType > RescaleFilterType;

RescaleFilterType::Pointer rescaler = RescaleFilterType::New();

rescaler->SetOutputMinimum(   0 );
rescaler->SetOutputMaximum( 255 );
rescaler->SetInput(output);

WriterType::Pointer writer = WriterType::New();
writer->SetFileName( argv[2] );
writer->SetInput(rescaler->GetOutput());
try
  {
  writer->Update();
  }
catch ( itk::ExceptionObject &err)
  {
  std::cout << "ExceptionObject caught !" << std::endl;
  std::cout << err << std::endl;
  return -1;
  }
```

The center image of Figure 19.6 shows the output of the Sobel algorithm applied to `Examples/Data/ROI_QB_PAN_1.tif`.

Convolution filtering: Sobel operator

The source code for this example can be found in the file
`Examples/Iterators/NeighborhoodIterators2.cxx`.

In this example, the Sobel edge-detection routine is rewritten using convolution filtering. Convolution filtering is a standard image processing technique that can be implemented numerically as the inner product of all image neighborhoods with a convolution kernel [35] [13]. 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 19.6. Note that x-direction operator produces the same output image as in the previous example.

Optimizing iteration speed

The source code for this example can be found in the file
`Examples/Iterators/NeighborhoodIterators3.cxx`.

This example illustrates a technique for improving the efficiency of neighborhood calculations by eliminating unnecessary bounds checking. As described in Section 19.4, 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 and OTB 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 example can be found in the file
`Examples/Iterators/NeighborhoodIterators4.cxx`.

We now introduce a variation on convolution filtering that is useful when a convolution kernel is separable. In this example, we create a different neighborhood iterator for each axial direction of the image and then take separate inner products with a 1D discrete Gaussian kernel. The idea of using several neighborhood iterators at once has applications beyond convolution filtering and may improve efficiency when the size of the whole neighborhood relative to the portion of the neighborhood used in calculations becomes large.

The only new class necessary for this example is the Gaussian operator.

```
#include "itkGaussianOperator.h"
```

The Gaussian operator, like the Sobel operator, is instantiated with a pixel type and a dimensionality. Additionally, we set the variance of the Gaussian, which has been read from the command line as standard deviation.

```
  itk::GaussianOperator< PixelType, 2 > gaussianOperator;
  gaussianOperator.SetVariance( ::atof(argv[3]) * ::atof(argv[3]) );
```

The only further changes from the previous example are in the main loop. Once again we use the results from face calculator to construct a loop that processes boundary and non-boundary image regions separately. Separable convolution, however, requires an additional, outer loop over all the image dimensions. The direction of the Gaussian operator is reset at each iteration of the outer loop using the new dimension. The iterators change direction to match because they are initialized with the radius of the Gaussian operator.

Input and output buffers are swapped at each iteration so that the output of the previous iteration becomes the input for the current iteration. The swap is not performed on the last iteration.

```
  ImageType::Pointer input = reader->GetOutput();
  for (unsigned int i = 0; i < ImageType::ImageDimension; ++i)
    {
    gaussianOperator.SetDirection(i);
    gaussianOperator.CreateDirectional();

    faceList = faceCalculator(input, output->GetRequestedRegion(),
                              gaussianOperator.GetRadius());
```

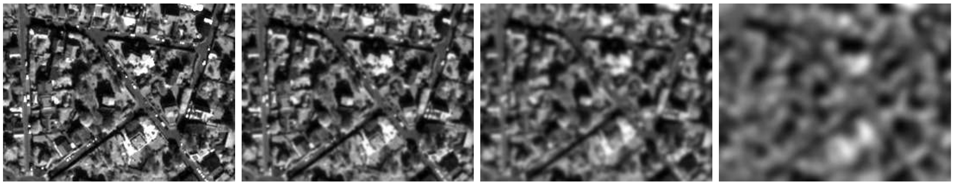


Figure 19.7: Results of convolution filtering with a Gaussian kernel of increasing standard deviation σ (from left to right, $\sigma = 0$, $\sigma = 1$, $\sigma = 2$, $\sigma = 5$). Increased blurring reduces contrast and changes the average intensity value of the image, which causes the image to appear brighter when rescaled.

```
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 19.7 shows the results of Gaussian blurring the image `Examples/Data/QB_Suburb.png` using increasing kernel widths.

Random access iteration

The source code for this example can be found in the file
`Examples/Iterators/NeighborhoodIterators6.cxx`.

Some image processing routines do not need to visit every pixel in an image. Flood-fill and connected-component algorithms, for example, only visit pixels that are locally connected to one another. Algorithms such as these can be efficiently written using the random access capabilities of the neighborhood iterator.

The following example finds local minima. Given a seed point, we can search the neighborhood of that point and pick the smallest value *m*. While *m* is not at the center of our current neighborhood, we move in the direction of *m* and repeat the analysis. Eventually we discover a local minimum and stop. This algorithm is made trivially simple in ND using an ITK neighborhood iterator.

To illustrate the process, we create an image that descends everywhere to a single minimum: a positive distance transform to a point. The details of creating the distance transform are not relevant to the discussion of neighborhood iterators, but can be found in the source code of this example. Some noise has been added to the distance transform image for additional interest.

The variable `input` is the pointer to the distance transform image. The local minimum algorithm is initialized with a seed point read from the command line.

```
ImageType::IndexType index;
index[0] = ::atoi(argv[2]);
index[1] = ::atoi(argv[3]);
```

Next we create the neighborhood iterator and position it at the seed point.

```
NeighborhoodIteratorType::RadiusType radius;
radius.Fill(1);
NeighborhoodIteratorType it(radius, input, input->GetRequestedRegion());

it.SetLocation(index);
```

Searching for the local minimum involves finding the minimum in the current neighborhood, then shifting the neighborhood in the direction of that minimum. The `for` loop below records the `itk::Offset` of the minimum neighborhood pixel. The neighborhood iterator is then moved using that offset. When a local minimum is detected, `flag` will remain false and the `while` loop will exit. Note that this code is valid for an image of any dimensionality.

```
bool flag = true;
while ( flag == true )
  {
  NeighborhoodIteratorType::OffsetType nextMove;
  nextMove.Fill(0);

  flag = false;

  PixelType min = it.GetCenterPixel();
  for (unsigned i = 0; i < it.Size(); i++)
    {
    if ( it.GetPixel(i) < min )
      {
      min = it.GetPixel(i);
      nextMove = it.GetOffset(i);
```

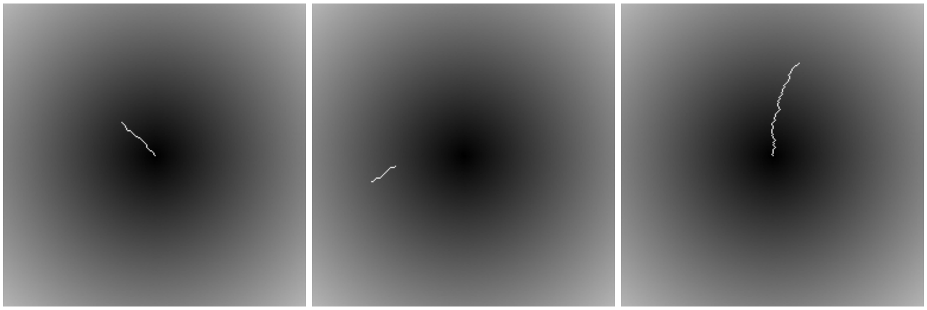


Figure 19.8: Paths traversed by the neighborhood iterator from different seed points to the local minimum. The true minimum is at the center of the image. The path of the iterator is shown in white. The effect of noise in the image is seen as small perturbations in each path.

```
        flag = true;
        }
      }
    it.SetCenterPixel( 255.0 );
    it += nextMove;
    }
```

Figure 19.8 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.

19.4.2 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 [35], [13], et al.). The examples that follow implement erosion and dilation.

Shaped neighborhoods: morphological operations

The source code for this example can be found in the file
`Examples/Iterators/ShapedNeighborhoodIterators1.cxx`.

This example uses `itk::ShapedNeighborhoodIterator` to implement a binary erosion algorithm. If we think of an image I as a set of pixel indices, then erosion of I by a smaller set E, called the *structuring element*, is the set of all indices at locations x in I such that when E is positioned at x, every element in E is also contained in I.

This type of algorithm is easy to implement with shaped neighborhood iterators because we can use the iterator itself as the structuring element E and move it sequentially through all positions *x*. The result at *x* is obtained by checking values in a simple iteration loop through the neighborhood stencil.

We need two iterators, a shaped iterator for the input image and a regular image iterator for writing results to the output image.

```
#include "itkConstShapedNeighborhoodIterator.h"
#include "itkImageRegionIterator.h"
```

Since we are working with binary images in this example, an `unsigned char` pixel type will do. The image and iterator types are defined using the pixel type.

```
  typedef unsigned char PixelType;
  typedef otb::Image< PixelType, 2 >  ImageType;

  typedef itk::ConstShapedNeighborhoodIterator<
                                          ImageType
                                            > ShapedNeighborhoodIteratorType;

  typedef itk::ImageRegionIterator< ImageType> IteratorType;
```

Refer to the examples in Section 19.4.1 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 19.4.1 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 19.4.1 is created and used as before.

```
  typedef itk::NeighborhoodAlgorithm::ImageBoundaryFacesCalculator<
                                                ImageType > FaceCalculatorType;

  FaceCalculatorType faceCalculator;
  FaceCalculatorType::FaceListType faceList;
  FaceCalculatorType::FaceListType::iterator fit;

  faceList = faceCalculator( reader->GetOutput(),
                             output->GetRequestedRegion(),
                             radius );
```

Now we initialize some variables and constants.

```
IteratorType out;

const PixelType background_value = 0;
const PixelType foreground_value = 255;
const float rad = static_cast<float>(element_radius);
```

The outer loop of the algorithm is structured as in previous neighborhood iterator examples. Each region in the face list is processed in turn. As each new region is processed, the input and output iterators are initialized on that region.

The shaped iterator that ranges over the input is our structuring element and its active stencil must be created accordingly. For this example, the structuring element is shaped like a circle of radius `element_radius`. Each of the appropriate neighborhood offsets is activated in the double `for` loop.

```
for ( fit=faceList.begin(); fit != faceList.end(); ++fit)
  {
  ShapedNeighborhoodIteratorType it( radius, reader->GetOutput(), *fit );
  out = IteratorType( output, *fit );

  // Creates a circular structuring element by activating all the pixels less
  // than radius distance from the center of the neighborhood.

  for (float y = -rad; y <= rad; y++)
    {
    for (float x = -rad; x <= rad; x++)
      {
      ShapedNeighborhoodIteratorType::OffsetType off;

      float dis = ::sqrt( x*x + y*y );
      if (dis <= rad)
        {
        off[0] = static_cast<int>(x);
        off[1] = static_cast<int>(y);
        it.ActivateOffset(off);
        }
      }
    }
```

The inner loop, which implements the erosion algorithm, is fairly simple. The `for` loop steps the input and output iterators through their respective images. At each step, the active stencil of the shaped iterator is traversed to determine whether all pixels underneath the stencil contain the foreground value, i.e. are contained within the set I. Note the use of the stencil iterator, `ci`, in performing this check.

```
  // Implements erosion
```

```
for (it.GoToBegin(), out.GoToBegin(); !it.IsAtEnd(); ++it, ++out)
  {
  ShapedNeighborhoodIteratorType::ConstIterator ci;

  bool flag = true;
  for (ci = it.Begin(); ci != it.End(); ci++)
    {
    if (ci.Get() == background_value)
      {
      flag = false;
      break;
      }
    }
  if (flag == true)
    {
    out.Set(foreground_value);
    }
  else
    {
    out.Set(background_value);
    }
  }
}
```

The source code for this example can be found in the file
`Examples/Iterators/ShapedNeighborhoodIterators2.cxx`.

The logic of the inner loop can be rewritten to perform dilation. Dilation of the set I by E is the set of all x such that E positioned at x contains at least one element in I.

```
// Implements dilation
for (it.GoToBegin(), out.GoToBegin(); !it.IsAtEnd(); ++it, ++out)
  {
  ShapedNeighborhoodIteratorType::ConstIterator ci;

  bool flag = false;
  for (ci = it.Begin(); ci != it.End(); ci++)
    {
    if (ci.Get() != background_value)
      {
      flag = true;
      break;
      }
    }
  if (flag == true)
    {
    out.Set(foreground_value);
    }
```

Figure 19.9: The effects of morphological operations on a binary image using a circular structuring element of size 4. Left: original image. Right: dilation.

```
      else
        {
        out.Set(background_value);
        }
      }
    }
```

The output image is written and visualized directly as a binary image of `unsigned chars`. Figure 19.9 illustrates the results of dilation on the image `Examples/Data/BinaryImage.png`. Applying erosion and dilation in sequence effects the morphological operations of opening and closing.

CHAPTER

TWENTY

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 19).

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.

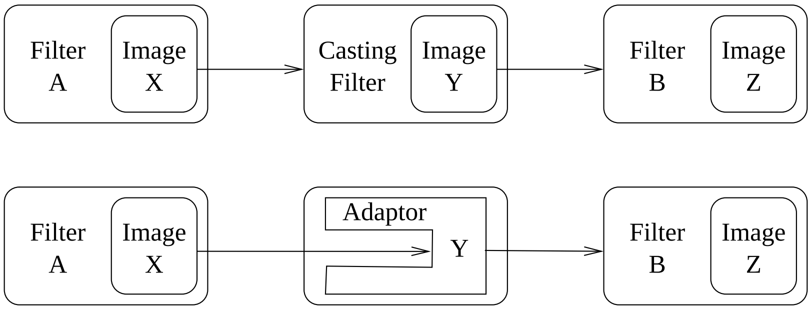


Figure 20.1: The difference between using a CastImageFilter and an ImageAdaptor. ImageAdaptors convert pixel values when they are accessed by iterators. Thus, they do not produces an intermediate image. In the example illustrated by this figure, the *Image Y* is not created by the ImageAdaptor; instead, the image is simulated on the fly each time an iterator from the filter downstream attempts to access the image data.

20.1 Image Casting

The source code for this example can be found in the file
`Examples/DataRepresentation/Image/ImageAdaptor1.cxx`.

This example illustrates how the `itk::ImageAdaptor` can be used to cast an image from one pixel type to another. In particular, we will *adapt* an `unsigned char` image to make it appear as an image of pixel type `float`.

We begin by including the relevant headers.

```
#include "otbImage.h"
#include "itkImageAdaptor.h"
```

First, we need to define a *pixel accessor* class that does the actual conversion. Note that in general, the only valid operations for pixel accessors are those that only require the value of the input pixel. As such, neighborhood type operations are not possible. A pixel accessor must provide methods `Set()` and `Get()`, and define the types of `InternalPixelType` and `ExternalPixelType`. The `InternalPixelType` corresponds to the pixel type of the image to be adapted (`unsigned char` in this example). The `ExternalPixelType` corresponds to the pixel type we wish to emulate with the ImageAdaptor (`float` in this case).

```
class CastPixelAccessor
{
public:
  typedef unsigned char InternalType;
  typedef float         ExternalType;
```

```
  static void Set(InternalType & output, const ExternalType & input)
    {
      output = static_cast<InternalType>( input );
    }

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

The CastPixelAccessor class simply applies a `static_cast` to the pixel values. We now use this pixel accessor to define the image adaptor type and create an instance using the standard `New()` method.

```
  typedef unsigned char  InputPixelType;
  const   unsigned int   Dimension = 2;
  typedef otb::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 otb::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 `otb::Image` class.

20.2 Adapting RGB Images

The source code for this example can be found in the file
`Examples/DataRepresentation/Image/ImageAdaptor2.cxx`.

This example illustrates how to use the `itk::ImageAdaptor` to access the individual components of an RGB image. In this case, we create an ImageAdaptor that will accept a RGB image as input and presents it as a scalar image. The pixel data will be taken directly from the red channel of the original image.

As with the previous example, the bulk of the effort in creating the image adaptor is associated with the definition of the pixel accessor class. In this case, the accessor converts a RGB vector to a scalar containing the red channel component. Note that in the following, we do not need to define the `Set()` method since we only expect the adaptor to be used for reading data from the image.

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

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

The `Get()` method simply calls the `GetRed()` method defined in the `itk::RGBPixel` class.

Now we use the internal pixel type of the pixel accessor to define the input image type, and then proceed to instantiate the ImageAdaptor type.

```
  typedef RedChannelPixelAccessor::InternalType  InputPixelType;
  const   unsigned int   Dimension = 2;
  typedef otb::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 otb::ImageFileReader< ImageType >   ReaderType;
ReaderType::Pointer reader = ReaderType::New();

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

We create an `itk::RescaleIntensityImageFilter` and an `otb::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 otb::Image< unsigned char, Dimension >   OutputImageType;
typedef itk::RescaleIntensityImageFilter< ImageAdaptorType,
                                          OutputImageType
                                             >   RescalerType;

RescalerType::Pointer rescaler = RescalerType::New();
typedef otb::ImageFileWriter< OutputImageType >   WriterType;
WriterType::Pointer writer = WriterType::New();
```

Now we connect the adaptor as the input to the rescaler and set the parameters for the intensity rescaling.

```
rescaler->SetOutputMinimum(  0  );
rescaler->SetOutputMaximum( 255 );

rescaler->SetInput( adaptor );
writer->SetInput( rescaler->GetOutput() );
```

Finally, we invoke the `Update()` method on the writer and take precautions to catch any exception that may be thrown during the execution of the pipeline.

```
try
  {
  writer->Update();
  }
catch( itk::ExceptionObject & excp )
  {
  std::cerr << "Exception caught " << excp << std::endl;
  return 1;
  }
```

ImageAdaptors for the green and blue channels can easily be implemented by modifying the pixel accessor of the red channel and then using the new pixel accessor for instantiating the type of an image adaptor. The following define a green channel pixel accessor.

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

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

A blue channel pixel accessor is similarly defined.

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

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

20.3 Adapting Vector Images

The source code for this example can be found in the file
`Examples/DataRepresentation/Image/ImageAdaptor3.cxx`.

This example illustrates the use of `itk::ImageAdaptor` to obtain access to the components of a vector image. Specifically, it shows how to manage pixel accessors containing internal parameters. In this example we create an image of vectors by using a gradient filter. Then, we use an image adaptor to extract one of the components of the vector image. The vector type used by the gradient filter is the `itk::CovariantVector` class.

We start by including the relevant headers.

```
#include "itkCovariantVector.h"
#include "itkGradientRecursiveGaussianImageFilter.h"
```

A pixel accessors class may have internal parameters that affect the operations performed on input pixel data. Image adaptors support parameters in their internal pixel accessor by using the assignment operator. Any pixel accessor which has internal parameters must therefore implement the assignment operator. The following defines a pixel accessor for extracting components from a vector pixel. The `m_Index` member variable is used to select the vector component to be returned.

```
class VectorPixelAccessor
{
public:
  typedef itk::CovariantVector<float,2>   InternalType;
  typedef                      float      ExternalType;

  void operator=( const VectorPixelAccessor & vpa )
    {
      m_Index = vpa.m_Index;
    }
  ExternalType Get( const InternalType & input ) const
    {
      return static_cast<ExternalType>( input[ m_Index ] );
    }
  void SetIndex( unsigned int index )
    {
      m_Index = index;
    }
private:
  unsigned int m_Index;
};
```

The `Get()` method simply returns the *i*-th component of the vector as indicated by the index. The assignment operator transfers the value of the index member variable from one instance of the pixel accessor to another.

In order to test the pixel accessor, we generate an image of vectors using the `itk::GradientRecursiveGaussianImageFilter`. This filter produces an output image of `itk::CovariantVector` pixel type. Covariant vectors are the natural representation for gradients since they are the equivalent of normals to iso-values manifolds.

```
  typedef unsigned char  InputPixelType;
  const   unsigned int   Dimension = 2;
  typedef otb::Image< InputPixelType,  Dimension  >   InputImageType;
  typedef itk::CovariantVector< float, Dimension  >   VectorPixelType;
  typedef otb::Image< VectorPixelType, Dimension  >   VectorImageType;
  typedef itk::GradientRecursiveGaussianImageFilter< InputImageType,
                                          VectorImageType> GradientFilterType;

  GradientFilterType::Pointer gradient = GradientFilterType::New();
```

We instantiate the ImageAdaptor using the vector image type as the first template parameter and the pixel accessor as the second template parameter.

```
typedef itk::ImageAdaptor<  VectorImageType,
                            VectorPixelAccessor > ImageAdaptorType;

ImageAdaptorType::Pointer adaptor = ImageAdaptorType::New();
```

The index of the component to be extracted is specified from the command line. In the following, we create the accessor, set the index and connect the accessor to the image adaptor using the `SetPixelAccessor()` method.

```
VectorPixelAccessor  accessor;
accessor.SetIndex( atoi( argv[3] ) );
adaptor->SetPixelAccessor( accessor );
```

We create a reader to load the image specified from the command line and pass its output as the input to the gradient filter.

```
typedef otb::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() );
```

20.4 Adaptors for Simple Computation

The source code for this example can be found in the file
`Examples/DataRepresentation/Image/ImageAdaptor4.cxx`.

Image adaptors can also be used to perform simple pixel-wise computations on image data. The following example illustrates how to use the `itk::ImageAdaptor` for image thresholding.

A pixel accessor for image thresholding requires that the accessor maintain the threshold value. Therefore, it must also implement the assignment operator to set this internal parameter.

```
class ThresholdingPixelAccessor
{
public:
```

```
  typedef unsigned char      InternalType;
  typedef unsigned char      ExternalType;

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

  void operator=( const ThresholdingPixelAccessor & vpa )
    {
      m_Threshold = vpa.m_Threshold;
    }
private:
  InternalType m_Threshold;
};
```

The `Get()` method returns one if the input pixel is above the threshold and zero otherwise. The assignment operator transfers the value of the threshold member variable from one instance of the pixel accessor to another.

To create an image adaptor, we first instantiate an image type whose pixel type is the same as the internal pixel type of the pixel accessor.

```
  typedef ThresholdingPixelAccessor::InternalType     PixelType;
  const   unsigned int   Dimension = 2;
  typedef otb::Image< PixelType,  Dimension >   ImageType;
```

We instantiate the ImageAdaptor using the image type as the first template parameter and the pixel accessor as the second template parameter.

```
  typedef itk::ImageAdaptor<  ImageType,
                              ThresholdingPixelAccessor > ImageAdaptorType;

  ImageAdaptorType::Pointer adaptor = ImageAdaptorType::New();
```

The threshold value is set from the command line. A threshold pixel accessor is created and connected to the image adaptor in the same manner as in the previous example.

```
  ThresholdingPixelAccessor  accessor;
  accessor.SetThreshold( atoi( argv[3] ) );
  adaptor->SetPixelAccessor( accessor );
```

Figure 20.2: 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 100, while the image on the right corresponds to a threshold of 200.

We create a reader to load the input image and connect the output of the reader as the input to the adaptor.

```
typedef otb::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 20.2 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.

20.5 Adaptors and Writers

Image adaptors will not behave correctly when connected directly to a writer. The reason is that writers tend to get direct access to the image buffer from their input, since image adaptors do not have a real buffer their behavior in this circumstances is incorrect. You should avoid instantiating the `ImageFileWriter` or the `ImageSeriesWriter` over an image adaptor type.

CHAPTER

TWENTYONE

Streaming and Threading

Streaming and threading are a complex issue in computing in general. This chapter provides the keys to help you understand how it is working so you can make the right choices later.

21.1 Introduction

First, you have to be aware that streaming and threading are two different things even if they are linked to a certain extent. In OTB:

- Streaming describes the ability to combine the processing of several portion of a big image and to make the output identical as what you would have gotten if the whole image was processed at once. Streaming is compulsory when you're processing gigabyte images.
- Threading is the ability to process simultaneously different parts of the image. Threading will give you some benefits only if you have a fairly recent processor (dual, quad core and some older P4).

To sum up: streaming is good if you have big images, threading is good if you have several processing units.

However, these two properties are not unrelated. Both rely on the filter ability to process parts of the image and combine the result, that what the ThreadedGenerateData() method can do.

21.2 Streaming and threading in OTB

For OTB, streaming is pipeline related while threading is filter related. If you build a pipeline where one filter is not streamable, the whole pipeline is not streamable: at one point, you would hold the entire image in memory. Whereas you will benefit from a threaded filter even if the rest of the pipeline is made of non-threadable filters (the processing time will be shorter for this particular filter).

Even if you use a non streamed writer, each filter which has a ThreadedGenerateData() will split the image into two and send each part to one thread and you will notice two calls to the function.

If you have some particular requirement and want to use only one thread, you can call the SetNumberOfThreads() method on each of your filter.

When you are writing your own filter, you have to follow some rules to make your filter streamable and threadable. Some details are provided in sections 22.3 and 22.4.

21.3 Division strategies

The division of the image occurs generally at the writer level. Different strategies are available and can be specified explicitly. In OTB, these are referred as *splitter*. Several available splitters are:

- `itk::ImageRegionSplitter`
- `itk::ImageRegionMultidimensionalSplitter`
- `otb::ImageRegionNonUniformMultidimensionalSplitter`

You can add your own strategies based on these examples.

To change the splitting strategy of the writer, you can use the following model:

```
typedef otb::ImageRegionNonUniformMultidimensionalSplitter<3> splitterType;
splitterType::Pointer splitter=splitterType::New() ;
writer->SetRegionSplitter(splitter);
```

CHAPTER

TWENTYTWO

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.

22.1 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 `otb::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.

22.2 Overview of Filter Creation

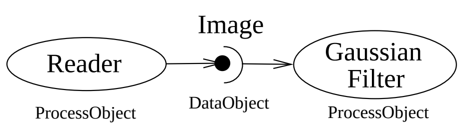


Figure 22.1: Relationship between DataObject and ProcessObject.

Filters are defined with respect to the type of data they input (if any), and the type of data they output (if any). The key to writing a ITK filter is to identify the number and types of input and output. Having done so, there are often superclasses that simplify this task via class derivation. For example, most filters in ITK take a single image as input, and produce a single image on output. The superclass `itk::ImageToImageFilter` is a convenience class that provide most of the functionality needed for such a filter.

Some common base classes for new filters include:

- `ImageToImageFilter`: the most common filter base for segmentation algorithms. Takes an image and produces a new image, by default of the same dimensions. Override `GenerateOutputInformation` to produce a different size.
- `UnaryFunctorImageFilter`: used when defining a filter that applies a function to an image.
- `BinaryFunctorImageFilter`: used when defining a filter that applies an operation to two images.
- `ImageFunction`: a functor that can be applied to an image, evaluating $f(x)$ at each point in the image.
- `MeshToMeshFilter`: a filter that transforms meshes, such as tessellation, polygon reduction, and so on.
- `LightObject`: abstract base for filters that don't fit well anywhere else in the class hierarchy. Also useful for "calculator" filters; ie. a sink filter that takes an input and calculates a result which is retrieved using a `Get()` method.

Once the appropriate superclass is identified, the filter writer implements the class defining the methods required by most all ITK objects: `New()`, `PrintSelf()`, and protected constructor, copy constructor, delete, and operator=, and so on. Also, don't forget standard typedefs like `Self`, `Superclass`, `Pointer`, and `ConstPointer`. Then the filter writer can focus on the most important parts of the implementation: defining the API, data members, and other implementation details of the algorithm. In particular, the filter writer will have to implement either a `GenerateData()` (non-threaded) or `ThreadedGenerateData()` method. (See Section 3.2.7 for an overview of multi-threading in ITK.)

An important note: the GenerateData() method is required to allocate memory for the output. The ThreadedGenerateData() method is not. In default implementation (see `itk::ImageSource`, a superclass of `itk::ImageToImageFilter`) `GenerateData()` allocates memory and then invokes `ThreadedGenerateData()`.

One of the most important decisions that the developer must make is whether the filter can stream data; that is, process just a portion of the input to produce a portion of the output. Often superclass behavior works well: if the filter processes the input using single pixel access, then the default behavior is adequate. If not, then the user may have to a) find a more specialized superclass to derive from, or b) override one or more methods that control how the filter operates during pipeline execution. The next section describes these methods.

22.3 Streaming Large Data

The data associated with multi-dimensional images is large and becoming larger. This trend is due to advances in scanning resolution, as well as increases in computing capability. Any

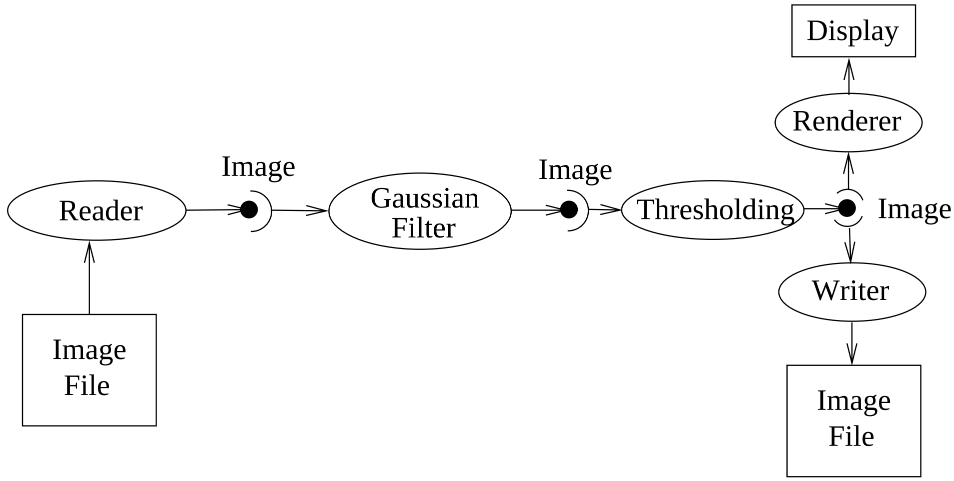


Figure 22.2: The Data Pipeline

practical segmentation and registration software system must address this fact in order to be useful in application. ITK addresses this problem via its data streaming facility.

In ITK, streaming is the process of dividing data into pieces, or regions, and then processing this data through the data pipeline. Recall that the pipeline consists of process objects that generate data objects, connected into a pipeline topology. The input to a process object is a data object (unless the process initiates the pipeline and then it is a source process object). These data objects in turn are consumed by other process objects, and so on, until a directed graph of data flow is constructed. Eventually the pipeline is terminated by one or more mappers, that may write data to storage, or interface with a graphics or other system. This is illustrated in figures 22.1 and 22.2.

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.

22.3.1 Overview of Pipeline Execution

The pipeline execution process performs several important functions.

1. It determines which filters, in a pipeline of filters, need to execute. This prevents redundant execution and minimizes overall execution time.

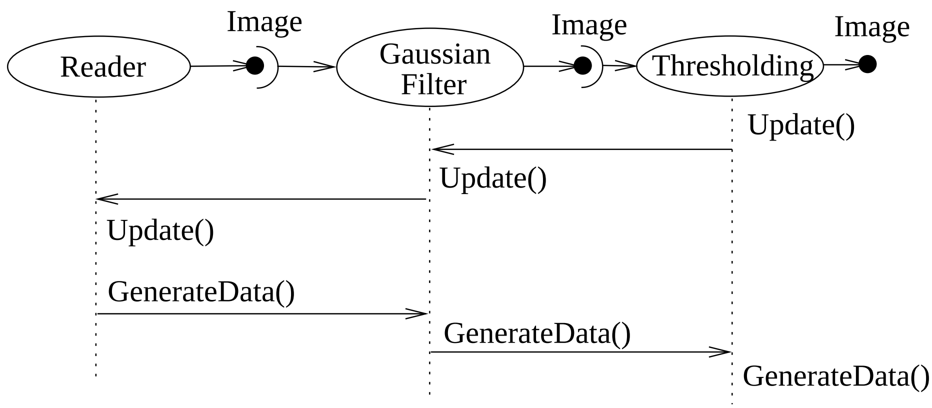


Figure 22.3: Sequence of the Data Pipeline updating mechanism

2. It initializes the (filter's) output data objects, preparing them for new data. In addition, it determines how much memory each filter must allocate for its output, and allocates it.

3. The execution process determines how much data a filter must process in order to produce an output of sufficient size for downstream filters; it also takes into account any limits on memory or special filter requirements. Other factors include the size of data processing kernels, that affect how much data input data (extra padding) is required.

4. It subdivides data into subpieces for multithreading. (Note that the division of data into subpieces is exactly same problem as dividing data into pieces for streaming; hence multithreading comes for free as part of the streaming architecture.)

5. It may free (or release) output data if filters no longer need it to compute, and the user requests that data is to be released. (Note: a filter's output data object may be considered a "cache". If the cache is allowed to remain (`ReleaseDataFlagOff()`) between pipeline execution, and the filter, or the input to the filter, never changes, then process objects downstream of the filter just reuse the filter's cache to re-execute.)

To perform these functions, the execution process negotiates with the filters that define the pipeline. Only each filter can know how much data is required on input to produce a particular output. For example, a shrink filter with a shrink factor of two requires an image twice as large (in terms of its x-y dimensions) on input to produce a particular size output. An image convolution filter would require extra input (boundary padding) depending on the size of the convolution kernel. Some filters require the entire input to produce an output (for example, a histogram), and have the option of requesting the entire input. (In this case streaming does not work unless the developer creates a filter that can request multiple pieces, caching state between each piece to assemble the final output.)

Ultimately the negotiation process is controlled by the request for data of a particular size (i.e., region). It may be that the user asks to process a region of interest within a large image, or that memory limitations result in processing the data in several pieces. For example, an application may compute the memory required by a pipeline, and then use `itk::StreamingImageFilter` to break the data processing into several pieces. The data request is propagated through the pipeline in the upstream direction, and the negotiation process configures each filter to produce output data of a particular size.

The secret to creating a streaming filter is to understand how this negotiation process works, and how to override its default behavior by using the appropriate virtual functions defined in `itk::ProcessObject`. The next section describes the specifics of these methods, and when to override them. Examples are provided along the way to illustrate concepts.

22.3.2 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 Largest-PossibleRegion).
- `GenerateOutputRequestedRegion(DataObject *output)` gives the subclass a chance to define how to set the requested regions for each of its outputs, given this output's requested region. The default implementation is to make all the output requested regions the same. A subclass may need to override this method if each output is a different resolution. This method is only overridden if a filter has multiple outputs.
- `GenerateInputRequestedRegion()` gives the subclass a chance to request a larger requested region on the inputs. This is necessary when, for example, a filter requires more data at the "internal" boundaries to produce the boundary values - due to kernel operations or other region boundary effects.

`itk::RGBGibbsPriorFilter` is an example of a filter that needs to invoke `EnlargeOutputRequestedRegion()`. The designer of this filter decided that the filter should operate on all the data. Note that a subtle interplay between this method and `GenerateInputRequestedRegion()` is occurring here. The default behavior of `GenerateInputRequestedRegion()` (at least for `itk::ImageToImageFilter`) is to set the input RequestedRegion to the output's ReqestedRegion. Hence, by overriding the method `EnlargeOutputRequestedRegion()` to set the output to the LargestPossibleRegion, effectively sets the input to this filter to the LargestPossibleRegion (and probably causing all upstream filters to process their LargestPossibleRegion as well. This means that the filter, and therefore the pipeline, does not stream. This could be fixed by reimplementing the filter with the notion of streaming built in to the algorithm.)

`itk::GradientMagnitudeImageFilter` is an example of a filter that needs to invoke `GenerateInputRequestedRegion()`. It needs a larger input requested region because a kernel is required to compute the gradient at a pixel. Hence the input needs to be "padded out" so the filter has enough data to compute the gradient at each output pixel.

UpdateOutputData()

`UpdateOutputData()` is the third and final method as a result of the `Update()` method. The purpose of this method is to determine whether a particular filter needs to execute in order to bring its output up to date. (A filter executes when its `GenerateData()` method is invoked.) Filter execution occurs when a) the filter is modified as a result of modifying an instance variable; b) the input to the filter changes; c) the input data has been released; or d) an invalid RequestedRegion was set previously and the filter did not produce data. Filters execute in order in the downstream direction. Once a filter executes, all filters downstream of it must also execute.

`DataObject::UpdateOutputData()` is delegated to the DataObject's source (i.e., the ProcessObject that generated it) only if the DataObject needs to be updated. A comparison of modified time, pipeline time, release data flag, and valid requested region is made. If any one of these conditions indicate that the data needs regeneration, then the source's `ProcessObject::UpdateOutputData()` is invoked. These calls are made recursively up the pipeline until a source filter object is encountered, or the pipeline is determined to be up to date and valid. At this point, the recursion unrolls, and the execution of the filter proceeds. (This means that the output data is initialized, StartEvent is invoked, the filters `GenerateData()` is called, EndEvent is invoked, and input data to this filter may be released, if requested. In addition, this filter's InformationTime is updated to the current time.)

The developer will never override `UpdateOutputData()`. The developer need only write the `GenerateData()` method (non-threaded) or `ThreadedGenerateData()` method. A discussion of threading follows in the next section.

22.4 Threaded Filter Execution

Filters that can process data in pieces can typically multi-process using the data parallel, shared memory implementation built into the pipeline execution process. To create a multithreaded filter, simply define and implement a `ThreadedGenerateData()` method. For example, a itk::ImageToImageFilter would create the method:

```
void ThreadedGenerateData(const OutputImageRegionType&
                          outputRegionForThread, int threadId)
```

The key to threading is to generate output for the output region given (as the first parameter in the argument list above). In ITK, this is simple to do because an output iterator can be created using the region provided. Hence the output can be iterated over, accessing the corresponding input pixels as necessary to compute the value of the output pixel.

Multi-threading requires caution when performing I/O (including using cout or cerr) or invoking events. A safe practice is to allow only thread id zero to perform I/O or generate events. (The thread id is passed as argument into ThreadedGenerateData()). If more than one thread tries to write to the same place at the same time, the program can behave badly, and possibly even deadlock or crash.

22.5 Filter Conventions

In order to fully participate in the ITK pipeline, filters are expected to follow certain conventions, and provide certain interfaces. This section describes the minimum requirements for a filter to integrate into the ITK framework.

The class declaration for a filter should include the macro ITK_EXPORT, so that on certain platforms an export declaration can be included.

A filter should define public types for the class itself (Self) and its Superclass, and const and non-const smart pointers, thus:

```
typedef ExampleImageFilter                Self;
typedef ImageToImageFilter<TImage,TImage> Superclass;
typedef SmartPointer<Self>                Pointer;
typedef SmartPointer<const Self>          ConstPointer;
```

The Pointer type is particularly useful, as it is a smart pointer that will be used by all client code to hold a reference-counted instantiation of the filter.

Once the above types have been defined, you can use the following convenience macros, which permit your filter to participate in the object factory mechanism, and to be created using the canonical ::New():

```
/** Method for creation through the object factory. */
itkNewMacro(Self);

/** Run-time type information (and related methods). */
itkTypeMacro(ExampleImageFilter, ImageToImageFilter);
```

The default constructor should be `protected`, and provide sensible defaults (usually zero) for all parameters. The copy constructor and assignment operator should be declared `private` and not implemented, to prevent instantiating the filter without the factory methods (above).

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

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

22.5.1 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.

22.5.2 Useful Macros

Many convenience macros are provided by ITK, to simplify filter coding. Some of these are described below:

itkStaticConstMacro Declares a static variable of the given type, with the specified initial value.

itkGetMacro Defines an accessor method for the specified scalar data member. The convention is for data members to have a prefix of `m_`.

itkSetMacro Defines a mutator method for the specified scalar data member, of the supplied type. This will automatically set the `Modified` flag, so the filter stage will be executed on the next `Update()`.

itkBooleanMacro Defines a pair of `OnFlag` and `OffFlag` methods for a boolean variable `m_Flag`.

itkGetObjectMacro, itkSetObjectMacro Defines an accessor and mutator for an ITK object. The Get form returns a smart pointer to the object.

Much more useful information can be learned from browsing the source in `Code/Common/itkMacro.h` and for the `itk::Object` and `itk::LightObject` classes.

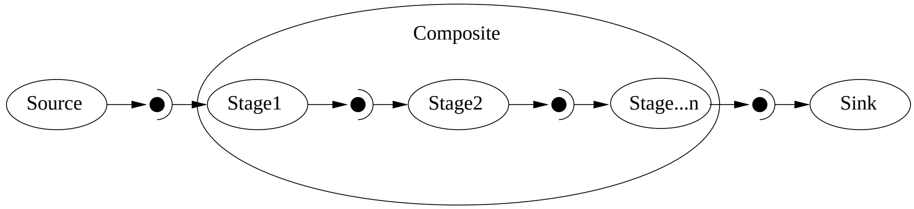


Figure 22.4: A Composite filter encapsulates a number of other filters.

22.6 How To Write A Composite Filter

In general, most ITK/OTB filters implement one particular algorithm, whether it be image filtering, an information metric, or a segmentation algorithm. In the previous section, we saw how to write new filters from scratch. However, it is often very useful to be able to make a new filter by combining two or more existing filters, which can then be used as a building block in a complex pipeline. This approach follows the Composite pattern [33], 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 22.4, 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.

22.6.1 Implementing a Composite Filter

There are a few considerations to take into account when implementing a composite filter. All the usual requirements for filters apply (as discussed above), but the following guidelines should be considered:

1. The template arguments it takes must be sufficient to instantiate all of the component filters. Each component filter needs a type supplied by either the implementor or the enclosing class. For example, an `ImageToImageFilter` normally takes an input and output image type (which may be the same). But if the output of the composite filter is a classified image, we need to either decide on the output type inside the composite filter, or restrict the choices of the user when she/he instantiates the filter.

2. The types of the component filters should be declared in the header, preferably with `protected` visibility. This is because the internal structure normally should not be visible to users of the class, but should be to descendent classes that may need to modify or customize the behavior.

3. The component filters should be private data members of the composite class, as in `FilterType::Pointer`.

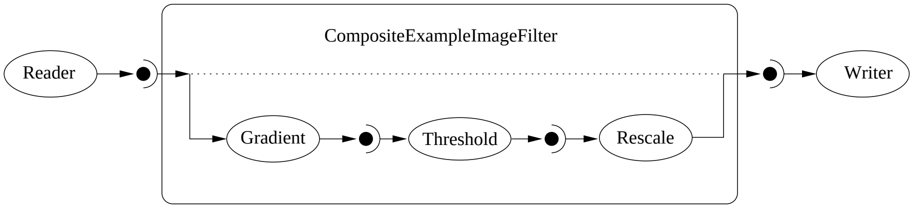


Figure 22.5: Example of a typical composite filter. Note that the output of the last filter in the internal pipeline must be grafted into the output of the composite filter.

4. The default constructor should build the pipeline by creating the stages and connect them together, along with any default parameter settings, as appropriate.
5. The input and output of the composite filter need to be grafted on to the head and tail (respectively) of the component filters.

This grafting process is illustrated in Figure 22.5.

22.6.2 A Simple Example

The source code for this example can be found in the file
`Examples/Filtering/CompositeFilterExample.cxx`.

The composite filter we will build combines three filters: a gradient magnitude operator, which will calculate the first-order derivative of the image; a thresholding step to select edges over a given strength; and finally a rescaling filter, to ensure the resulting image data is visible by scaling the intensity to the full spectrum of the output image type.

Since this filter takes an image and produces another image (of identical type), we will specialize the ImageToImageFilter:

```
#include "itkImageToImageFilter.h"
```

Next we include headers for the component filters:

```
#include "itkGradientMagnitudeImageFilter.h"
#include "itkThresholdImageFilter.h"
#include "itkRescaleIntensityImageFilter.h"
```

Now we can declare the filter itself. It is within the OTB namespace, and we decide to make it use the same image type for both input and output, thus the template declaration needs only one parameter. Deriving from `ImageToImageFilter` provides default behavior for several important aspects, notably allocating the output image (and making it the same dimensions as the input).

```
namespace otb {

template <class TImageType>
class ITK_EXPORT CompositeExampleImageFilter :
    public itk::ImageToImageFilter<TImageType, TImageType>
{
public:
```

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

```
  typedef CompositeExampleImageFilter               Self;
  typedef itk::ImageToImageFilter<TImageType,TImageType> Superclass;
  typedef itk::SmartPointer<Self>                        Pointer;
  typedef itk::SmartPointer<const Self>                  ConstPointer;
```

Here we declare an alias (to save typing) for the image's pixel type, which determines the type of the threshold value. We then use the convenience macros to define the Get and Set methods for this parameter.

```
  typedef typename TImageType::PixelType PixelType;

  itkGetMacro( Threshold, PixelType);
  itkSetMacro( Threshold, PixelType);
```

Now we can declare the component filter types, templated over the enclosing image type:

```
protected:

  typedef itk::ThresholdImageFilter< TImageType > ThresholdType;
  typedef itk::GradientMagnitudeImageFilter< TImageType, TImageType >
                                                           GradientType;
  typedef itk::RescaleIntensityImageFilter< TImageType, TImageType >
                                                           RescalerType;
```

The component filters are declared as data members, all using the smart pointer types.

```
  typename GradientType::Pointer     m_GradientFilter;
  typename ThresholdType::Pointer    m_ThresholdFilter;
  typename RescalerType::Pointer     m_RescaleFilter;

  PixelType m_Threshold;
};

} /* namespace otb */
```

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

```
template <class TImageType>
CompositeExampleImageFilter<TImageType>
::CompositeExampleImageFilter()
{
  m_GradientFilter = GradientType::New();
  m_ThresholdFilter = ThresholdType::New();
  m_RescaleFilter = RescalerType::New();

  m_ThresholdFilter->SetInput( m_GradientFilter->GetOutput() );
  m_RescaleFilter->SetInput( m_ThresholdFilter->GetOutput() );

  m_Threshold = 1;

  m_RescaleFilter->SetOutputMinimum(
                     itk::NumericTraits<PixelType>::NonpositiveMin());
  m_RescaleFilter->SetOutputMaximum(itk::NumericTraits<PixelType>::max());
}
```

The `GenerateData()` is where the composite magic happens. First, we connect the first component filter to the inputs of the composite filter (the actual input, supplied by the upstream stage). Then we graft the output of the last stage onto the output of the composite, which ensures the filter regions are updated. We force the composite pipeline to be processed by calling `Update()` on the final stage, then graft the output back onto the output of the enclosing filter, so it has the result available to the downstream filter.

```
template <class TImageType>
void
CompositeExampleImageFilter<TImageType>::
GenerateData()
{
  m_GradientFilter->SetInput( this->GetInput() );

  m_ThresholdFilter->ThresholdBelow( this->m_Threshold );

  m_RescaleFilter->GraftOutput( this->GetOutput() );
  m_RescaleFilter->Update();
  this->GraftOutput( m_RescaleFilter->GetOutput() );
}
```

Finally we define the `PrintSelf` method, which (by convention) prints the filter parameters. Note how it invokes the superclass to print itself first, and also how the indentation prefixes each line.

```
template <class TImageType>
void
CompositeExampleImageFilter<TImageType>::
PrintSelf( std::ostream& os, itk::Indent indent ) const
{
  Superclass::PrintSelf(os,indent);

  os
    << indent << "Threshold:" << this->m_Threshold
    << std::endl;
}

} /* end namespace otb */
```

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 22.5.

Part V

Appendix

CHAPTER

TWENTYTHREE

Frequently Asked Questions

23.1 Introduction

23.1.1 What is OTB?

OTB, the ORFEO Toolbox is a library of image processing algorithms developed by CNES in the frame of the ORFEO Accompaniment Program. OTB is based on the medical image processing library ITK, `http://www.itk.org`, and offers particular functionalities for remote sensing image processing in general and for high spatial resolution images in particular.

OTB provides:

- image access: optimized read/write access for most of remote sensing image formats, meta-data access, simple visualization;
- sensor geometry: sensor models, cartographic projections;
- radiometry: atmospheric corrections, vegetation indices;
- filtering: blurring, denoising, enhancement;
- fusion: image pansharpening;
- feature extraction: interest points, alignments, lines;
- image segmentation: region growing, watershed, level sets;
- classification: K-means, SVM, Markov random fields;
- change detection.

Many of these functionalities are provided by ITK and have been tested and documented for the use with remote sensing data.

23.1.2 What is ORFEO?

ORFEO stands for Optical and Radar Federated Earth Observation. In 2001 a cooperation program was set between France and Italy to develop ORFEO, an Earth observation dual system with metric resolution: Italy is in charge of COSMO-Skymed the radar component development, and France of PLEIADES the optic component.

The PLEIADES optic component is composed of two ”small satellites” (mass of one ton) offering a spatial resolution at nadir of 0.7 m and a field of view of 20 km. Their great agility enables a daily access all over the world, essentially for defense and civil security applications, and a coverage capacity necessary for the cartography kind of applications at scales better than those accessible to SPOT family satellites. Moreover, PLEIADES will have stereoscopic acquisition capacity to meet the fine cartography needs, notably in urban regions, and to bring more information when used with aerial photography.

The ORFEO ”targeted” acquisition capacities made it a system particularly adapted to defense or civil security missions, as well as critical geophysical phenomena survey such as volcanic eruptions, which require a priority use of the system resources.

With respect to the constraints of the franco-italian agreement, cooperations have been set up for the PLEIADES optical component with Sweden, Belgium, Spain and Austria.

Where can I get more information about ORFEO?

At the PLEIADES HR web site: `http://smsc.cnes.fr/PLEIADES/`.

23.1.3 What is the ORFEO Accompaniment Program?

Beside the Pleiades (PHR) and Cosmo-Skymed (CSK) systems developments forming ORFEO, the dual and bilateral system (France - Italy) for Earth Observation, the ORFEO Accompaniment Program was set up, to prepare, accompany and promote the use and the exploitation of the images derived from these sensors.

The creation of a preparatory program is needed because of :

- the new capabilities and performances of the ORFEO systems (optical and radar high resolution, access capability, data quality, possibility to acquire simultaneously in optic and radar),
- the implied need of new methodological developments : new processing methods, or adaptation of existing methods,
- the need to realize those new developments in very close cooperation with the final users, the integration of new products in their systems.

This program was initiated by CNES mid-2003 and will last until 2009. It consists in two parts, between which it is necessary to keep a strong interaction:

- A Methodological part,
- A Thematic part.

This Accompaniment Program uses simulated data (acquired during airborne campaigns) and satellite images quite similar to Pleiades (as QuickBird and Ikonos), used in a communal way on a set of special sites. The validation of specified products and services will be realized with those simulated data

Apart from the initial cooperation with Italy, the ORFEO Accompaniment Program enlarged to Belgium, with integration of Belgian experts in the different WG as well as a participation to the methodological part.

Where can I get more information about the ORFEO Accompaniment Program?

Go to the following web site: `http://smsc.cnes.fr/PLEIADES/A_prog_accomp.htm`.

23.1.4 Who is responsible for the OTB development?

The French Centre National d'Études Spatiales, CNES, initiated the ORFEO Toolbox and is responsible for the specification of the library. CNES funds the industrial development contracts and research contracts needed for the evolution of OTB.

23.2 Licence

23.2.1 Which is the OTB licence?

OTB is distributed under a free software licence:
`http://www.cecill.info/licences/Licence_CeCILL_V2-en.html`.

23.2.2 If I write an application using OTB am I forced to distribute that application?

No. The license gives you the option to distribute your application if you want to. You do not have to exercise this option in the license.

23.2.3 If I wanted to distribute an application using OTB what license would I need to use?

The CeCILL licence.

23.2.4 I am a commercial user. Is there any restriction on the use of OTB?

OTB can be used internally (”in-house”) without restriction, but only redistributed in other software that is under the CeCILL licence.

23.3 Getting OTB

23.3.1 Who can download the OTB?

Anybody can download the OTB at no cost.

23.3.2 Where can I download the OTB?

Go to `http://otb.cnes.fr` and follow the ”download OTB” link. You will have access to the OTB source code and to the Software User’s Guide.

23.4 Installing OTB

23.4.1 Which platforms are supported

OTB is a multi-platform library. It has successfully been installed on the following platforms:

- Linux/Unix with GCC (2.95.X, 3.3.X, 4.1.X, 4.2.X).
- Windows with Microsoft Visual Studio C++ 7.1 .NET 2003.
- Windows with Microsoft Visual Studio C++ 8.0 .NET 2005.
- Windows with MinGW. (mingw + msys at `http://www.mingw.org`)
- Cygwin. (`http://www.cygwin.com`)

Support for the following platforms is planned:

- Windows with Microsoft Visual Studio C++ 6.0.

23.4.2 Which libraries/packages are needed before installing OTB?

- CMake (`http://www.cmake.org`)
- GDAL (`http://www.gdal.org`)
- Fltk (`http://www.fltk.org`)

23.4.3 Main steps

In order to install OTB on your system follow these steps (in the given order):

1. Install CMake.
2. Install GDAL.
3. Install Fltk using the CMake scripts. Do not use the `configure` approach or the project files for Microsoft Visual Studio shipped with Fltk.
4. Install OTB using CMake for the configuration.

We assume that you will install everything on a directory called `INSTALL_DIR`, which usually is `/usr/local`, `/home/jordi/local` or whatever you want. Make sure that you have downloaded the source code for:

- CMake (`http://www.cmake.org`)
- GDAL (`http://www.gdal.org`)
- Fltk (`http://www.fltk.org`)

Unix/Linux Platforms

Important note: on some Linux distributions (eg. Debian, Ubuntu, Fedora), you may use the official packages for CMake, GDAL and Fltk. Once you have installed these packages, you can skip to step 4.

1. Install GDAL

```
cd INSTALL_DIR
gunzip gdal.1.4.2.tar.gz
tar xvf gdal.1.4.2.tar
cd gdal.1.4.2
./configure --prefix=INSTALL_DIR
make
make install
```

It seems to be a bug in the GDAL install procedure: if you are installing it without root privileges, even if your `INSTALL_DIR` is a directory for which you have the write permissions, GDAL tries to copy the python bindings together with the Python site packages, which are usually somewhere in /usr/lib.

Actually, since this is the last step in the GDAL install procedure, when you get the error message, the GDAL libs and header files are already installed, so you can safely ignore the error.

The `--without-python` option passed to the `configure` step avoids this. However, some users may want to have Python bindings, so recommending this option for the install may not be OK for everybody.

2. Install CMake

```
cd INSTALL_DIR
gunzip cmake-2.4.7.tar.gz
tar xvf cmake-2.4.7.tar
cd cmake-2.4.7
./configure --prefix=INSTALL_DIR
make
make install
```

In order to properly use cmake, add `INSTALL_DIR/bin` to your path with `export PATH=$PATH:INSTALL_DIR/bin` or something similar.

3. Install Fltk (optional) using CMake (do not use the configure script)

```
    cd INSTALL_DIR
    bunzip2 fltk-1.1.7-source.tar.bz2 OR
    gunzip fltk-1.1.7-source.tar.gz
    tar xvf fltk-1.1.7-source.tar
    mkdir Fltk-binary
    cd Fltk-binary
    ccmake ../fltk-1.1.7
    --> follow the CMake instructions, in particular:
        --> set CMAKE_INSTALL_PREFIX to INSTALL_DIR within CMake
--> set BUILD_EXAMPLES to ON within CMake
--> generate the configuration with 'g'
    make
    make install
    --> check that the examples located in
    INSTALL_DIR/Fltk-binary/bin work, in particular, the fractals
    example which makes use of the OpenGL library needed by OTB.
```

You can choose not to install Fltk but in this case, you will not be able to compile the visualization features of OTB.

4. Install OTB

```
     cd INSTALL_DIR
     gunzip OrfeoToolbox-2.0.0.tgz
     tar xvf OrfeoToolbox-2.0.0.tar
     mkdir OTB-Binary
     cd OTB-Binary
     ccmake ../OrfeoToolbox-2.0.0
     --> follow the CMake instructions, in particular:
 --> set BUILD_EXAMPLES to ON within CMake
 --> set BUILD_SHARED_LIBS to ON within CMake
 --> set BUILD_TESTING to OFF within CMake
 --> set CMAKE_INSTALL_PREFIX to INSTALL_DIR within CMake
 --> set GDAL_INCLUDE_DIRS to INSTALL_DIR/include within CMake
 --> set GDAL_LIBRARY_DIRS to INSTALL_DIR/lib within CMake
 --> set OTB_USE_EXTERNAL_ITK to OFF within CMake
 --> set FLTK_DIR to INSTALL_DIR/Fltk-Binary within CMake OR
     if you do not have FLTK press 't' to change to advanced
             mode and set OTB_USE_VISU to OFF
 --> generate the configuration with 'g'
      make
```

If you want a faster compilation and don't want the compilation of the examples, you can set `BUILD_EXAMPLES` to `OFF`. Some plateforms apparently have more difficulties with shared libraries, if you experience any problem with that, you can set `BUILD_SHARED_LIBS` to `OFF` but the built size might reach 1 GB.

After these steps, you have the source of OTB in `INSTALL_DIR/OrfeoToolbox-2.0.0` and the compiled binaries and libraries in `INSTALL_DIR/OTB-Binary`. Keeping the sources is important as most programs you will designed will need an access to the txx files during compilation. However, the binaries directory knows were its sources are and you will need to point only to the `INSTALL_DIR/OTB-Binary` when the `cmake` for your program will ask you where the OTB is.

If you want to put OTB in a standard location, you can proceed with:

```
make install
```

but this is only optional.

Microsoft Visual Studio C++ 7.1

1. Install GDAL

   MSVC++ 7.1 project files are needed to compile GDAL.

   These files can be downloaded at `http://vterrain.org/dist/gdal132_vc71.zip`.

   Then, unzip it to your GDAL folder, and it will create a folder (named "VisualStudio").

   Load the solution (.sln file) and build the gdal project.

   More details can be found at `http://vterrain.org/Distrib/gdal.html`.

2. Install Fltk

   Use CMake on Windows to generate MSVC++ 7.1 project files from fltk sources.

   Open the solution and build the fltk project.

3. Install OTB

   Use CMake on Windows to generate MSVC++ 7.1 project files from otb sources.

   Open the solution and build the otb project.

Microsoft Visual Studio C++ 8.0

1. Install GDAL

   Open a MS-DOS prompt.

   Run the VCVARS32.bat script that comes with the compiler (it can be found in Microsoft Visual Studio 8/VC/bin).

   Then, go to the GDAL root directory, and tape :

   ```
   nmake /f makefile.vc
   ```

   Once the build is successful, tape this line to install GDAL :

   ```
   nmake /f makefile.vc install
   ```

   More details about this install can be found at `http://www.gdal.org/gdal_building.html`.

2. Install Fltk

   Use CMake on Windows to generate MSVC++ 8.0 project files from fltk sources.

   Open the solution and build the fltk project.

3. Install OTB

   Use CMake on Windows to generate MSVC++ 8.0 project files from otb sources.

   Open the solution and build the otb project.

MinGW on Windows platform

1. Download the lastest version of mingw and msys at `http://www.mingw.org` and install those two programs.

   Then, launch MinGW : a prompt appears (similar to Linux one).

2. Install GDAL

   To compile GDAL, at configure step, use these options :

   ```
   ./configure -prefix=INSTALL_DIR --host=mingw32 --without-libtool
   --without-python --with-png=internal --with-libtiff=internal
   --with-jpeg=internal
   ```

   Then the usual make and make install.

3. Install Fltk

   Generate MSYS Makefiles with CMake (Windows version) from fltk sources.

   Then, under prompt, tape make and make install where you have generated Makefiles with CMake.

4. Install OTB

   Similar to fltk install.

Cygwin

1. Download the lastest version at `http://www.cygwin.com` and install it. Then, launch it, a prompt appears (similar to Linux one).

2. Install GDAL

   To compile GDAL, at configure step, use these options :

   ```
   ./configure --prefix=INSTALL_DIR --with-png=internal --with-libtiff=internal
   --with-jpeg=internal
   ```

   Then the usual make and make install.

3. Install Fltk

   See Linux part for details (same procedure).

4. Install OTB

   See Linux part for details (same procedure).

That should be all! Otherwise, subscribe to otb-users@googlegroups.com and you will get some help.

23.4.4 Specific platform issues

SunOS/HP UX

Due to a bug in the tar command shipped with some versions of SunOS, problems may appear when configuring, compiling or installing OTB.

See `http://www.gnu.org/software/tar/manual/tar.html#Checksumming` for details on the bug characterization.

The solution is to use the GNU tar command if it is available on your system (gtar).

Linux Debian/Ubuntu

If you used the official gdal package version 1.4.0, the library is named `libgdal1.4.0.so` so you have to create a simlink named `libgdal.so`: `ln -s /usr/lib/libgdal1.4.0.so /usr/lib/libgdal.so`.

Cygwin

Due to an unknown bug, Fltk can't compile on some versions of Cygwin (OpenGL problems).

Put OTB_USE_VISU to OFF to avoid these problems.

Some bugs can appear while compiling GDAL with JPEG2000 files : disable this format to solve the problem.

MSVC++ 8.0

Execution errors can appear on some platforms, using GDAL compiled with MSVC++ 8.0.

This problem can be solved by downloading GDAL binaries for Windows at `http://vterrain.org/Distrib/gdal.html`.

23.5 Using OTB

23.5.1 Where to start ?

OTB presents a large set of features and it is not always easy to start using it. After the installation, you can proceed to the tutorials (in the Software Guide). This should give you a quick overview of the possibilities of OTB and will teach you how to build your own programs.

23.5.2 What is the image size limitation of OTB ?

The maximum physical space a user can allocate depends on her platform. Therefore, image allocation in OTB is restricted by image dimension, size, pixel type and number of bands.

Fortunately, thanks to the streaming mechanism implemented within OTB's pipeline (actually ITK's), this limitation can be bypassed. The use of the `otb::StreamingImageFileWriter` at the end of the pipeline, or the `itk::StreamingImageFilter` at any point of the pipeline will seamlessly break the large, problematic data into small pieces whose allocation is possible. These pieces are processed one afther the other, so that there is not allocation problem anymore. We are often working with images of 25000×25000 pixels.

For the streaming to work, all the filters in the pipeline must be streaming capable (this is the case for most of the filters in OTB). The output image format also need to be streamable (not PNG or JPEG, but TIFF or ENVI, for instance).

To tune the size of the streaming pieces, the OTB has two CMake variables. The first is named OTB_STREAM_IMAGE_SIZE_TO_ACTIVATE_STREAMING. It represents the minimum size of the image in bytes for which streaming may be helpful. The second, OTB_STREAM_MAX_SIZE_BUFFER_FOR_STREAMING, specifies the maximum size in bytes a streaming piece should have. It can be used to compute the optimal number of pieces to break the input data into.

These two parameters have been used in the OTB-Applications/Utils/ applications. Take this as an example of how they can be used. They can also be tuned by the user to match her specific needs.

23.6 Getting help

23.6.1 Is there any mailing list?

Yes. There is a discussion group at `http://groups.google.com/group/otb-users/` where you can get help on the set up and the use of OTB.

23.6.2 Which is the main source of documentation?

The main source of documentation is the OTB Software Guide which can be downloaded at `http://orfeo-toolbox.sourceforge.net/Docs/OTBSoftwareGuide.pdf`. It contains tenths of commented examples and a tutorial which should be a good starting point for any new OTB user. The code source for these examples is distributed with the toolbox. Another information source is the on-line API documentation which is available at `http://orfeo-toolbox.sourceforge.net/Doxygen`.

23.7 Contributing to OTB

23.7.1 I want to contribute to OTB, where to begin?

First, you can send an email to otb@cnes.fr to let us know what functionality you would like to introduce in OTB. If the functionality seems important for the OTB users, we will then discuss on how to send your code, make the necessary adaptions, check with you that the results are correct and finally include it in the next release.

23.7.2 What are the benefits of contributing to OTB?

Besides the satisfaction of contributing to an open source project, we will include the references to relevant papers in the software guide. Having algorithms published in the form of reproducible research helps science move faster and encourages people who needs your algorithms to use them.

You will also benefit from the strengths of OTB: multiplatform, streaming and threading, etc.

23.7.3 What functionality can I contribute?

All functionalities which are useful for remote sensing data are of interest. As OTB is a library, it should be generic algorithms: change, detection, fusion, object detection, segmentation, interpolation, etc.

More specific applications can be contributed to the OTB-Applications package.

23.8 OTB's Roadmap

23.8.1 Which will be the next version of OTB?

OTB's version numbers have 3 digits. The first one is for major versions, the second one is for minor versions and the last one is for bugfixes.

The first version was 1.0.0 in July 2006. Version 1.2.0, 1.4.0 and 1.6.0 were released in between and the current one 2.0.0 was released in December 2007. The next one will probably be 2.2.0.

What is a major version?

A major version of the library implies the addition of high-level functionalities as for instance image registration, object recognition, etc.

What is a minor version?

A minor version is released when low-level functionalities are available within one major functionality, as for instance a new change detector, a new feature extractor, etc.

What is a bugfix version?

A bugfix version is released when significant bugs are identified and fixed.

23.8.2 When will the next version of OTB be available?

We plan to release major new OTB version once a year, that is, version 2.0.0 was available at the end of 2007, version 3.0.0 should be released by the end of 2008, and so on.

23.8.3 What features will the OTB include and when?

There is no detailed plan about the availability of OTB new features, since OTB's content depends on ongoing research work and on feedback from thematic users of the ORFEO Accompaniment Program.

Nevertheless, the main milestones for the OTB development are the following:

- Version 1 (2006):
  - core of the system,
  - IO,
  - basic filtering, segmentation and classification,
  - basic feature extraction,
  - basic change detection.
- Version 2 (2007):
  - geometric corrections,
  - radiometric corrections,
  - registration.
- Version 3 (2008):
  - multi-scale and multi-resolution analysis,
  - object detection and recognition,
  - supervised learning.

- Version 4 (2009):
  - data fusion,
  - spatial reasoning.

BIBLIOGRAPHY

[1] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Professional Computing Series. Addison-Wesley, 2001. 8.6.1

[2] E. L. Allwein, R. E. Schapire, and Y. Singer. Reducing multiclass to binary: A unifying approach for margin classifiers. In *Proc. 17th International Conf. on Machine Learning*, pages 9–16. Morgan Kaufmann, San Francisco, CA, 2000. 17.3.1

[3] K. Alsabti, S. Ranka, and V. Singh. An efficient k-means clustering algorithm. In *First Workshop on High-Performance Data Mining*, 1998. 17.1.1

[4] L. Alvarez and J.-M. Morel. *A Morphological Approach To Multiscale Analysis: From Principles to Equations*, pages 229–254. Kluwer Academic Publishers, 1994. 7.6.2

[5] M. H. Austern. *Generic Programming and the STL:*. Professional Computing Series. Addison-Wesley, 1999. 3.2.1, 8.6.1

[6] Y. Bazi, L. Bruzzone, and F. Melgani. An unsupervised approach based on the generalized Gaussian model to automatic change detection in multitemporal SAR images. *IEEE Trans. Geoscience and Remote Sensing*, 43(4):874–887, April 2005. 16.1.1

[7] J. Besag. On the statistical analysis of dirty pictures. *J. Royal Statist. Soc. B.*, 48:259–302, 1986. 17.1.5

[8] L. Bruzzone and F. Melgani. Support vector machines for classification of hyperspectral remote-sensing images. In *IEEE International Geoscience and Remote Sensing Symposium, IGARSS*, volume 1, pages 506–508, June 2002. 17.3

[9] L. Bruzzone and D. F. Prieto. An adaptive semiparametric and context-based approach to unsupervised change detection in multitemporal remote-sensing images. *IEEE Trans. Image Processing*, 11(4):452–466, April 2002. 16.1.1

[10] C. Burges. A Tutorial on Support Vector Machines for Pattern Recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167, 1998. 17.3.1

[11] R. H. Byrd, P. Lu, and J. Nocedal. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific and Statistical Computing*, 16(5):1190–1208, 1995. 8.8

[12] R. H. B. C. Zhu and J. Nocedal. L-bfgs-b: Algorithm 778: L-bfgs-b, fortran routines for large scale bound constrained optimization. *ACM Transactions on Mathematical Software*, 23(4):550–560, November 1997. 8.8

[13] K. Castleman. *Digital Image Processing*. Prentice Hall, Upper Saddle River, NJ, 1996. 19.4.1, 19.4.2

[14] G. Celeux and J. Diebolt. The SEM algorithm: a probabilistic teacher algorithm derived from the EM algorithm for the mixture problem. *Computational Statistics Quarterly*, 2(1):73–82, 1985. 17.2.1

[15] E. Christophe and J. Inglada. Robust road extraction for high resolution satellite images. In *IEEE International Conference on Image Processing, ICIP07*, 2007. 13.6.1

[16] A. Chung, W. Wells, A. Norbash, and W. Grimson. Multi-modal image registration by minimising kullback-leibler distance. In *MICCAI'02 Medical Image Computing and Computer-Assisted Intervention*, Lecture Notes in Computer Science, pages 525–532, 2002. 8.7.5

[17] A. Collignon, F. Maes, D. Delaere, D. Vandermeulen, P. Suetens, and G. Marchal. Automated multimodality image registration based on information theory. In *Information Processing in Medical Imaging 1995*, pages 263–274. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1995. 8.4

[18] P. R. Coppin, I. Jonckheere, and K. Nachaerts. Digital change detection in ecosystem monitoring: a review. *Int. J. of Remote Sensing*, 24:1–33, 2003. 16.1.1

[19] P. E. Danielsson. Euclidean distance mapping. *Computer Graphics and Image Processing*, 14:227–248, 1980. 7.7

[20] M. H. Davis, A. Khotanzad, D. P. Flamig, and S. E. Harms. A physics-based coordinate transformation for 3-d image matching. *IEEE Transactions on Medical Imaging*, 16(3), June 1997. 8.6.18

[21] P. Deer. *Digital change detection in remotely sensed imagery using fuzzy set theory*. Phd thesis, University of Adelaïde, Australia, Department of Geography and Computer Science, 1998. 16.1.1

[22] R. Deriche. Fast algorithms for low level vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(1):78–87, 1990. 7.2.2, 7.6.1

[23] R. Deriche. Recursively implementing the gaussian and its derivatives. Technical Report 1893, Unite de recherche INRIA Sophia-Antipolis, avril 1993. Research Repport. 7.2.2, 7.6.1

[24] S. Derrode, G. Mercier, and W. Pieczynski. Unsupervised change detection in SAR images using a multicomponent hidden Markov chain model. In *Second Int. Workshop on the Analysis of Multi-temporal Remote Sensing Images*, volume 3, pages 195–203, Ispra, Italy, July 16-18 2003. 16.1.1

[25] A. Desolneux, L. Moisan, and J.-M. Morel. Meaningful alignments. *Int. J. Comput. Vision*, 40(1):7–23, 2000. 13.3

[26] C. Dodson and T. Poston. *Tensor Geometry: The Geometric Viewpoint and its Uses*. Springer, 1997. 8.6.1, 2

[27] J. R. Dominique Fasbender and P. Bogaert. Bayesian Data Fusion for Adaptable Image Pansharpening. *Transactions on Geoscience and Remote Sensing, in press*, 2007. 12.2

[28] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern classification*. A Wiley-Interscience Publication, second edition, 2000. 17.1, 17.1

[29] S. Dudani, K. Breeding, and R. McGhee. Aircraft identification by moments invariants. *IEEE Transanctions on Computers*, 26:39–45, 1977. 13.5.2

[30] V. N. Dvorchenko. Bounds on (deterministic) correlation functions with applications to registration. *IEEE Trans. PAMI*, 5(2):206–213, 1983. 9.1

[31] D. Eberly. *Ridges in Image and Data Analysis*. Kluwer Academic Publishers, Dordrecht, 1996. 14.2.1

[32] J. Flusser. On the independence of rotation moment invariants. *Pattern Recognition*, 33:1405–1410, 2000. 13.5.2, 13.5.3

[33] 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, 6.2, 22.6

[34] 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. 7.6.2

[35] R. Gonzalez and R. Woods. *Digital Image Processing*. Addison-Wesley, Reading, MA, 1993. 19.4.1, 19.4.1, 19.4.2

[36] H. Gray. *Gray's Anatomy*. Merchant Book Company, sixteenth edition, 2003. 5.1.6

[37] S. Grossberg. Neural dynamics of brightness perception: Features, boundaries, diffusion, and resonance. *Perception and Psychophysics*, 36(5):428–456, 1984. 7.6.2

[38] J. Hajnal, D. J. Hawkes, and D. Hill. *Medical Image Registration*. CRC Press, 2001. 8.7.6

[39] W. R. Hamilton. *Elements of Quaternions*. Chelsea Publishing Company, 1969. 8.6.1, 8.6.11, 8.8

[40] M. Holden, D. L. G. Hill, E. R. E. Denton, J. M. Jarosz, T. C. S. Cox, and D. J. Hawkes. Voxel similarity measures for 3d serial mr brain image registration. In A. Kuba, M. Samal, and A. Todd-Pkropek, editors, *Information Processing in Medical Imaging 1999 (IPMI'99)*, pages 472–477. Springer, 1999. 8.7.3

[41] C. Hsu and C. Lin. A comparison of methods for multi-class support vector machines, 2001. 17.3.1

[42] M. K. Hu. Visual Pattern Recognition by moment invariants. *IEEE Transactions on Information Theory*, 8(2):179–187, 1962. 13.5.2

[43] J. Inglada. Similarity Measures for Multisensor Remote Sensing Images. In *International Geoscience and Remote Sensing Symposium, IGARSS 2002, CD-ROM*, 2002. 9.1.2

[44] J. Inglada. Change detection on SAR images by using a parametric estimation of the Kullback-Leibler divergence. In *IEEE Int. Conf. Geosci. Remote Sensing*, Toulouse, France, July, 21-25 2003. 16.1.1, 16.4.1

[45] J. Inglada and A. Giros. On the possibility of automatic multi-sensor image registration. *IEEE Trans. Geoscience and Remote Sensing*, 42(10), Oct. 2004. 8.4

[46] J. Inglada and G. Mercier. A New Statistical Similarity Measure for Change Detection in Multitemporal SAR Images and its Extension to Multiscale Change Analysis. *IEEE Trans. Geosci. Remote Sensing*, 45(5):1432–1446, May 2007. 16.1.1, 16.4.1

[47] J.Flusser and T. Suk. A moment based approach to registration of image with affine geometric distortion. *IEEE Transactions Geoscience Remote Sensing*, 32(2):382–387, 1994. 13.5.2

[48] T. Joachims. Text Categorization with Support Vector Machines: Learning with Many Relevant Features. Technical report, Computer Science of The University of dortmund, Nov. 1997. 17.3

[49] C. J. Joly. *A Manual of Quaternions*. MacMillan and Co. Limited, 1905. 8.6.11

[50] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. Piatko, R. Silverman, and A. Y. Wu. An efficient k-means clustering algorithm: Analysis and implementation. 17.1.1

[51] Y. J. Kaufman and D. Tanré. Atmospherically Resistant Vegetation Index (ARVI) for EOS-MODIS. *Transactions on Geoscience and Remote Sensing*, 40(2):261–270, Mar. 1992. 11.1.3

[52] J. Koënderink and A. van Doorn. The Structure of Two-Dimensional Scalar Fields with Applications to Vision. *Biol. Cybernetics*, 33:151–158, 1979. 14.2.1

[53] 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. 14.2.1

[54] C. Kuglin and D. Hines. The phase correlation image alignment method. In *IEEE Conference on Cybernetics and Society*, pages 163–165, 1975. 9.1

[55] V. Lacroix and M. Acheroy. Feature extraction using the constrained gradient. *ISPRS Journal of Photogrammetry & Remote Sensing*, 53:85–94, 1998. 13.6.1, 13.6.2

[56] J. Lee. Digital image enhancement and noise filtering by use of local statistics. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2:165–168, 1980. 7.6.3

[57] T. Lindeberg. *Scale-Space Theory in Computer Science*. Kluwer Academic Publishers, 1994. 7.6.1

[58] H. Lodish, A. Berk, S. Zipursky, P. Matsudaira, D. Baltimore, and J. Darnell. *Molecular Cell Biology*. W. H. Freeman and Company, 2000. 5.1.6, 8.6.1

[59] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004. 13.2.2

[60] D. Lu, P. Mausel, E. Brondizio, and E. Moran. Change detection techniques. *Int. J. of Remote Sensing*, 25(12):2365–2407, 2004. 16.1.1

[61] F. Maes, A. Collignon, D. Meulen, G. Marchal, and P. Suetens. Multi-modality image registration by maximization of mutual information. *IEEE Trans. on Med. Imaging*, 16:187–198, 1997. 8.4

[62] D. Malacara. *Color Vision and Colorimetry: Theory and Applications*. SPIE PRESS, 2002. 5.1.6, 5.1.6

[63] D. Mattes, D. R. Haynor, H. Vesselle, T. K. Lewellen, and W. Eubank. Non-rigid multi-modality image registration. In *Medical Imaging 2001: Image Processing*, pages 1609–1620, 2001. 8.6.17, 8.7.4

[64] 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, Jan. 2003. 8.6.17

[65] D. Musser and A. Saini. *STL Tutorial and Reference Guide*. Professional Computing Series. Addison-Wesley, 1996. 3.2.1

[66] V. Onana, E. Trouvé, G. Mauris, J. Rudant, and P. Frison. Change detection in urban context with multitemporal ERS-SAR images by using data fusion approach. In *IEEE Int. Conf. Geosci. Remote Sensing*, Toulouse, France, July, 21-25 2003. 16.1.1

[67] E. Osuna, R. Freund, and F. Girosi. Training support vector machines:an application to face detection, 1997. 17.3

[68] D. Pelleg and A. 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. 17.1.1

[69] G. P. Penney, J. Weese, J. A. Little, P. Desmedt, D. L. G. Hill, and D. J. Hawkes. A comparision of similarity measures for use in 2d-3d medical image registration. *IEEE Transactions on Medical Imaging*, 17(4):586–595, August 1998. 8.7.3

[70] P. Perona and J. Malik. Scale-space and edge detection using anisotropic diffusion. *IEEE Transactions on Pattern Analysis Machine Intelligence*, 12:629–639, 1990. 7.6.2, 7.6.2, 7.6.2

[71] 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, Aug. 2003. 8.7.4

[72] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C*. Cambridge University Press, second edition, 1992. 8.8

[73] R. J. Radke, S. Andra, O. Al-Kofahi, and B. Roysman. Image change detection algorithms: a systematic survey. *IEEE Trans. Image Processing*, 14(3):294–307, March 2005. 16.1.1

[74] J. A. Richards. Analysis of remotely sensed data: the formative decades and the fututre. *IEEE Trans. Geoscience and Remote Sensing*, 43(3):422–432, 2005. 16.1.1

[75] K. Rohr, M. Fornefett, and H. S. Stiehl. Approximating thin-plate splines for elastric registration: Integration of landmark errors and orientation attributes. In A. Kuba, M. Samal, and A. Todd-Pkropek, editors, *Information Processing in Medical Imaging 1999 (IPMI'99)*, pages 252–265. Springer, 1999. 8.6.18

[76] K. Rohr, H. S. Stiehl, R. Sprengel, T. M. Buzug, J. Weese, and M. H. Kuhn. Landmark-based elastic registration using approximating thin-plate splines. *IEEE Transactions on Medical Imaging*, 20(6):526–534, June 1997. 8.6.18

[77] 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. 8.6.17

[78] G. Sapiro and D. Ringach. Anisotropic diffusion of multivalued images with applications to color filtering. *IEEE Trans. on Image Processing*, 5:1582–1586, 1996. 7.6.2

[79] J. P. Serra. *Image Analysis and Mathematical Morphology*. Academic Press Inc., 1982. 7.5.3, 14.2.1

[80] J. Sethian. *Level Set Methods and Fast Marching Methods*. Cambridge University Press, 1996. 14.3

[81] J. C. Spall. An overview of the simultaneous perturbation method for efficient optimization. *Johns Hopkins APL Technical Digest*, 19:482–492, 1998. 8.8

[82] E. Stabel and P. Fischer. Detection of structural changes in river dynamics by radar-based earth observation methods. In *Proc. of the 1st Biennial Meeting of the Int. Environmental Modelling and Software Society*, volume 1, pages 352–358, Lugano, Switzerland, June 2002. 16.1.1

[83] 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, Mar. 2000. 8.8

[84] B. M. ter Haar Romeny, editor. *Geometry-Driven Diffusion in Computer Vision*. Kluwer Academic Publishers, 1994. 7.6.2

[85] R. Touzi, A. Lopes, and P. Bousquet. A statistical and geometrical edge detector for SAR images. *IEEE Trans. Geoscience and Remote Sensing*, 26(6):764–773, November 1988. 7.4.2

[86] J. Townshend, C. Justice, C. Gurney, and J. McManus. The impact of misregistration on change detection. *IEEE Transactions on Geoscience and Remote Sensing*, 30(5):1054–1060, sept 1992. 9.1.1

[87] F. Tupin, H. Maître, J.-F. Mangin, J.-M. Nicolas, and E. Pechersky. Detection of linear features in SAR images: application to road network extraction. *IEEE Transactions on Geoscience and Remote Sensing*, 36(2):434–453, Mar. 1998. 13.4.1, 13.4.1

[88] V. Vapnik. *Statistical learning theory*. John Wiley and Sons, NewYork, 1998. 17.3

[89] P. Viola and W. M. Wells III. Alignment by maximization of mutual information. *IJCV*, 24(2):137–154, 1997. 8.4, 8.7.4

[90] J. Weickert, B. ter Haar Romeny, and M. 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. 7.6.2

[91] J. Weston and C. Watkins. Multi-class support vector machines, 1998. 17.3.1

[92] R. 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. 7.6.2

[93] R. T. Whitaker. *Geometry-Limited Diffusion*. PhD thesis, The University of North Carolina, Chapel Hill, North Carolina 27599-3175, 1993. 7.6.2, 7.6.2

[94] R. 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. 7.6.2

[95] R. T. Whitaker and G. Gerig. *Vector-Valued Diffusion*, pages 93–134. Kluwer Academic Publishers, 1994. 7.6.2, 7.6.2

[96] R. T. Whitaker and S. M. Pizer. Geometry-based image segmentation using anisotropic diffusion. In Y.-L. O, A. Toet, H. Heijmans, D. Foster, and P. Meer, editors, *Shape in Picture: The mathematical description of shape in greylevel images*. Springer Verlag, Heidelberg, 1993. 7.6.2

[97] R. T. Whitaker and S. M. Pizer. A multi-scale approach to nonuniform diffusion. *Computer Vision, Graphics, and Image Processing: Image Understanding*, 57(1):99–110, January 1993. 7.6.2

[98] 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. 7.6.2

[99] G. Wyszecki. *Color Science: Concepts and Methods, Quantitative Data and Formulae*. Wiley-Interscience, 2000. 5.1.6, 5.1.6

[100] T. Yoo, U. Neumann, H. Fuchs, S. Pizer, T. Cullip, J. Rhoades, and R. Whitaker. Direct visualization of volume data. *IEEE Computer Graphics and Applications*, 12(4):63–71, 1992. 14.2.1

[101] T. Yoo, S. 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. 14.2.1, 14.2.1

[102] T. S. Yoo and J. 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. 7.6.2

INDEX