The ORFEO Tool Box Software Guide
Updated for OTB-5.2.1

OTB Development Team

January 24, 2016

http://www.orfeo-toolbox.org
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 program¹ 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 mid 2013. 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 users. Lastly, this part aims at raising future users awareness, through practical demonstrations and validations.

¹http://smsc.cnes.fr/PLEIADES/A_prog_accomp.htm
The Methodological part objective is the definition and the development of tools for the operational exploitation of the 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, CNES² 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 methodological 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 ITK³ (Insight Toolkit).

²http://www.cnes.fr
³http://www.itk.org
LIST OF FIGURES
LIST OF TABLES
Part I

Introduction
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 online at https://www.orfeo-toolbox.org/doxygen.

1.1 Organization

This software guide is divided into several 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 is a short guide with gradual difficulty to get you start programming with OTB. Part III describes the system from the user point of view. Dozens of examples are used to illustrate important system features. Part ?? 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 ??).

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 the first few tutorials in Part II to get
familiar with the build process and the general program organization, follow by reading Chapter 4, which provides an overview of some of the key concepts in the system, and then review the examples in Part III. 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 Website. 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 latest official OTB release (available on the OTB Web site).

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

```
http://www.orfeo-toolbox.org/
```

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 bookmarked 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.

There are two other ways of getting the OTB source code:

- Clone the current release with Git from the OTB git server, (master branch)
1.3. Software Organization

- Clone the latest revision with Git from the OTB git server (develop branch).

These last two options need a proper Git installation. To get source code from Git, do:

```
    git clone https://git@git.orfeo-toolbox.org/git/otb.git
```

Using Git, you can easily navigate through the different versions. For instance, this brings you to the stable branch for version 5.2:

```
    git checkout release-5.2
```

And this brings you to the latest development version:

```
    git checkout develop
```

There is also a mirror of OTB official repository on GitHub. You can find more information on the OTB git workflow in the wiki.

1.3.2 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.

The OTB contains the following subdirectories:

- **OTB/Modules**—the heart of the software; the location of the majority of the source code.
- **OTB/CMake**—internal files used during the configuration process.
- **OTB/Copyright**—the copyright information of OTB and all the dependencies included in the OTB source tree.
- **OTB/Examples**—a suite of simple, well-documented examples used by this guide and to illustrate important OTB concepts.
- **OTB/Superbuild**—CMake scripts to automatically download, patch, build and install important dependencies of OTB (ITK, OSSIM, GDAL to name a few).
- **OTB/Utilities**—small programs used for the maintenance of OTB.

OTB is organized into different modules, each one covering different part of image processing. It is therefore important to understand the source code directory structure—found in **OTB/Modules**—.
• OTB/Modules/Adapters—Adapters for Boost, Curl, Gdal, OpenThreads and Ossim.

• OTB/Modules/Applications—a set of applications modules that can be launched in different ways (command-line, graphical interface, Python/Java), refer to the OTB Cookbook for more information.

• OTB/Modules/Core—core classes, macro definitions, typedefs, and other software constructs central to OTB.

• OTB/Modules/Detection—detection of clouds, roads.

• OTB/Modules/Feature—various local descriptors and features.

• OTB/Modules/Filtering—basic image processing filters.

• OTB/Modules/Fusion—image fusion algorithms, as for instance, pansharpening.

• OTB/Modules/Hyperspectral—hyperspectral images analysis.

• OTB/Modules/IO—classes that support the reading and writing of data.

• OTB/Modules/Learning—several functionalities for supervised learning and classification.

• OTB/Modules/OBIA—Object Based Image Analysis filters and data structures.

• OTB/Modules/Radiometry—classes allowing to compute vegetation indices and radiometric corrections.

• OTB/Modules/Registration—classes for registration of images or other data structures to each other.

• OTB/Modules/Remote—Functions to fetch remote modules.

• OTB/Modules/Segmentation—several functionalities for image segmentation.

• OTB/Modules/ThirdParty—Modules that import OTB’s dependencies.

• OTB/Modules/Wrappers—Applications wrappers with several access points (command-line, QT Gui, SWIG...).

See also chapter ?? for more information about how to write modules.

1.3.3 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://www.orfeo-toolbox.org/doxygen/.

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.)

1.3.4 Data

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

1.4 The OTB Community and Support

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 Community

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.

Orfeo ToolBox is currently in the incubation stage of being part of the OSGeo foundation. Within the ORFEO ToolBox community we act respectfully toward others in line with the OSGeo Code of Conduct.

1.5 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 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 for better integration of new products in their systems.

This program was initiated by CNES mid-2003 and will last until 2010 at least. It consists in two parts, between which it is necessary to keep a strong interaction:

• A Thematic part

• A Methodological part.

1 http://www.osgeo.org/
2 http://www.osgeo.org/code_of_conduct
3 http://smsc.cnes.fr/PLEIADES/A_prog_accomp.htm
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\textsuperscript{4} decided to develop the \textit{ORFEO ToolBox} (OTB), a set of algorithms encapsulated in a software library. The goals of the OTB is to capitalize a methodological \textit{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 implemented in C++ and is mainly based on ITK\textsuperscript{5} (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 literally copied from ITK’s Software Guide (with permission).
- Many examples are taken from ITK.

\textbf{1.5.1 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

\textsuperscript{4}http://www.cnes.fr
\textsuperscript{5}http://www.itk.org
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 Mollah 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.
There are two ways to install OTB library on your system: installing from a binary distribution or compiling from sources. You can find information about the installation of binary packages for OTB and Monteverdi in the OTB-Cookbook. This chapter covers installation from sources, also known as compiling.

OTB has been developed and tested across different combinations of operating systems, compilers, and hardware platforms including MS-Windows, Linux on Intel-compatible hardware and Mac OSX. It is known to work with the following compilers in 32/64 bit:

- Visual Studio 2010 and higher compiler on MS-Windows
- GCC 4.1 and higher on Unix/Linux systems
- Clang on MacOSX (10.8 and higher) systems

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 supports several generators to produce the compilation scripts, depending on the platform and compiler. It can use:

- Makefiles for Unix systems
- Visual Studio workspaces for Windows
- NMake Makefiles for Windows
- Ninja scripts
and many more ...

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.

There are (at least) two ways to use CMake:

- Using the command ccmake (on Unix) or cmake-gui (on Windows): it provides an interactive mode in which 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 is the easiest way to start.

- Using the command cmake: it is a non-interactive polyvalent tool designed for scripting. It can run both configure and generate steps.

As shown in figure 2.1, CMake has a different interfaces according to your system. Refer to section 2.1 for Linux build instructions, ?? for Mac OS X, and 2.2 for Windows.

For more information on CMake, check:

http://www.cmake.org

OTB depends on a number of external libraries. Some are mandatory, meaning that OTB cannot be compiled without them, while others are optional and can be activated or not during the build process. See table 2.1 for the full list of dependencies.

2.1 Linux and Mac OS X

2.1.1 Setting up the build environment

The first thing to do is to create a directory for working with OTB. This guide will use ~/OTB but you are free to choose something else. In this directory, there will be three locations:

- ~/OTB/otb for the source file obtained from the git repository
- ~/OTB/build for the intermediate build objects, CMake specific files, libraries and binaries.
- ~/OTB/install, the installation directory for OTB once it is built. A system location (/usr/local for example) can also be used, but installing locally is more flexible and does not require root access.
Figure 2.1: CMake interface. Top) `ccmake`, the UNIX version based on curses. Bottom) `CMakeSetup`, the MS-Windows version based on MFC.
<table>
<thead>
<tr>
<th>Library</th>
<th>Web site</th>
<th>Mandatory</th>
<th>Minimum version</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITK</td>
<td><a href="http://www.itk.org">http://www.itk.org</a></td>
<td>yes</td>
<td>4.6.0</td>
</tr>
<tr>
<td>GDAL</td>
<td><a href="http://www.gdal.org">http://www.gdal.org</a></td>
<td>yes</td>
<td>1.10</td>
</tr>
<tr>
<td>OSSIM</td>
<td><a href="http://www.ossim.org">http://www.ossim.org</a></td>
<td>yes</td>
<td>1.8.20-3</td>
</tr>
<tr>
<td>Curl</td>
<td><a href="http://www.curl.haxx.se">http://www.curl.haxx.se</a></td>
<td>no</td>
<td>-</td>
</tr>
<tr>
<td>FFTW</td>
<td><a href="http://www.fftw.org">http://www.fftw.org</a></td>
<td>no</td>
<td>-</td>
</tr>
<tr>
<td>libgeotiff</td>
<td><a href="http://trac.osgeo.org/geotiff/">http://trac.osgeo.org/geotiff/</a></td>
<td>yes</td>
<td>-</td>
</tr>
<tr>
<td>OpenJPEG</td>
<td><a href="http://code.google.com/p/openjpeg/">http://code.google.com/p/openjpeg/</a></td>
<td>no</td>
<td>-</td>
</tr>
<tr>
<td>boost</td>
<td><a href="http://www.boost.org">http://www.boost.org</a></td>
<td>yes</td>
<td>-</td>
</tr>
<tr>
<td>opentheads</td>
<td><a href="http://www.openscenegraph.org">http://www.openscenegraph.org</a></td>
<td>yes</td>
<td>-</td>
</tr>
<tr>
<td>Mapnik</td>
<td><a href="http://www.mapnik.org">http://www.mapnik.org</a></td>
<td>no</td>
<td>-</td>
</tr>
<tr>
<td>tinyXML</td>
<td><a href="http://www.grinninglizard.com/tinyxml">http://www.grinninglizard.com/tinyxml</a></td>
<td>yes</td>
<td>-</td>
</tr>
<tr>
<td>6S</td>
<td><a href="http://6s.ltdri.org">http://6s.ltdri.org</a></td>
<td>no</td>
<td>-</td>
</tr>
<tr>
<td>SiftFast</td>
<td><a href="http://libsift.sourceforge.net">http://libsift.sourceforge.net</a></td>
<td>no</td>
<td>-</td>
</tr>
<tr>
<td>MuParser</td>
<td><a href="http://www.muparser.sourceforge.net">http://www.muparser.sourceforge.net</a></td>
<td>no</td>
<td>-</td>
</tr>
<tr>
<td>MuParserX</td>
<td><a href="http://muparserx.beltoforion.de">http://muparserx.beltoforion.de</a></td>
<td>no</td>
<td>3.0.5</td>
</tr>
<tr>
<td>libSVM</td>
<td><a href="http://www.csie.ntu.edu.tw/~cjlin/libsvm">http://www.csie.ntu.edu.tw/~cjlin/libsvm</a></td>
<td>no</td>
<td>2.0</td>
</tr>
<tr>
<td>Qt</td>
<td><a href="http://qt-project.org/">http://qt-project.org/</a></td>
<td>no</td>
<td>4</td>
</tr>
<tr>
<td>OpenCV</td>
<td><a href="http://opencv.org">http://opencv.org</a></td>
<td>no</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 2.1: External libraries used in OTB.
To setup this structure, the following commands can be used:

```
$ mkdir ~/OTB
$ cd ~/OTB
$ git clone https://git@git.orfeo-toolbox.org/git/otb.git
$ mkdir build
$ mkdir install
```

The OTB project uses a git branching model where `develop` is the current development version. It contains the latest patches and represents the work in progress towards the next release. For more information on OTB and git, including how to decide which branch to want to compile, please see the OTB wiki page at http://wiki.orfeo-toolbox.org/index.php/Git.

Checkout the relevant branch now:

```
$ cd ~/OTB/otb
$ git checkout develop
```

Now you must decide which build method you will use. There are two ways of compiling OTB from sources, depending on how you want to manage dependencies. Both methods rely on CMake.

- **SuperBuild** (go to section 2.1.2). All OTB dependencies are automatically downloaded and compiled. This method is the easiest to use and provides a complete OTB with minimal effort.

- **Normal build** (go to section 2.1.3). OTB dependencies must already be compiled and available on your system. This method requires more work but provides more flexibility.

If you do not know which method to use and just want to compile OTB with all its modules, use SuperBuild.

### 2.1.2 SuperBuild: Build OTB and all dependencies

The SuperBuild is a way of compiling dependencies to a project just before you build the project. Thanks to CMake and its ExternalProject module, it is possible to download a source archive, configure, compile and install it when building the main project. This feature has been used in other CMake-based projects (ITK, Slicer, ParaView,...). In OTB, the SuperBuild is implemented with no impact on the library sources: the sources for SuperBuild are located in the `OTB/SuperBuild` sub-directory. It is made of CMake scripts and source patches that allow to compile all the dependencies necessary for OTB. Once all the dependencies are compiled and installed, the OTB library is built using those dependencies.
CMake variable | Value
---|---
CMAKE\_INSTALL\_PREFIX | Installation directory, target for make install
BUILD\_EXAMPLES | Activate compilation of OTB examples
BUILD\_TESTING | Activate compilation of the tests
OTB\_BUILD\_DEFAULT\_MODULES | Activate all usual modules, required to build the examples
OTB\_USE\_XXX | Activate module XXX
OTBGroup\_XXX | Enable modules in the group XXX
OTB\_DATA\_ROOT | otb-data repository
OTB\_WRAP\_PYTHON | Enable Python wrapper
OTB\_WRAP\_JAVA | Enable Java wrapper

SuperBuild only

| Value
| ---
| Location to download dependencies
| Use the system’s XXX library

Table 2.2: Important CMake configuration variables in OTB

OTB’s compilation is customized by specifying configuration variables. The most important configuration variables are shown in table 2.2. The simplest way to provide configuration variables is via the command line -D option:

$ cd ~/OTB/build
$ cmake -D CMAKE\_INSTALL\_LOCATION=~/OTB/install ../otb/SuperBuild

A pre-load script can also be used with the -C options (see https://cmake.org/cmake/help/v3.4/manual/cmake.1.html#options). Another option is to set variables manually with cmake-gui or ccmake.

During the configuration step, the SuperBuild will detect any existing dependencies installed as systems libraries. Whether to use them can be controlled via the USE\_SYSTEM\_XXX (see table 2.2).

SuperBuild downloads dependencies into the DOWNLOAD\_LOCATION directory, which will be ~/OTB/build/Downloads in our example. Dependencies can be downloaded manually into this directory before the compilation step. This can be useful if you wish to bypass a proxy, intend to compile OTB without an internet connection, or other network constraint. You can find a complete bundle with all dependencies sources on the Orfeo ToolBox website (pick the ‘SuperBuild-archives’ corresponding to the OTB version you want to build):

https://www.orfeo-toolbox.org/packages

You are now ready to compile OTB! Simply use the make command (other targets can be generated with CMake’s -G option):

$ cd ~/OTB/build
$ make
The installation target will copy the binaries and libraries to the installation location:

$ make install

A wiki page detailing the status of SuperBuild on various platforms is also available here: http://wiki.orfeo-toolbox.org/index.php/SuperBuild.

### 2.1.3 Normal build: Build only OTB

Once all OTB dependencies are available on your system, use CMake to generate a Makefile:

$ cd ~/OTB/build
$ cmake -C configuration.cmake ../otb

The script `configuration.cmake` needs to contain dependencies location if CMake cannot find them automatically. This can be done with the `XXX_DIR` variables containing the directories which contain the FindXXX.cmake scripts, or with the `XXXINCLUDEDIR` and `XXXLIBRARY` variables.

Additionally, decide which module you wish to enable, together with tests and examples. Refer to table 2.2 for the list of CMake variables.

Since OTB is modularized, it is possible to only build some modules instead of the whole set. To deactivate a module (and the ones that depend on it) switch off the CMake variable `OTB_BUILD_DEFAULT_MODULES`, configure, and then switch off each `Module_module_name` variable. To provide an overview on how things work, the option `COMPONENTS` of the CMake command `find_package` is used in order to only load the requested modules. This module-specific list prevent CMake from performing a blind search; it is also a convenient way to monitor the dependencies of each module.

```
find_package(OTB COMPONENTS OTBCommon OTBTransform [...])
```

Some of the OTB capabilities are considered as optional, and you can deactivate the related modules thanks to a set of CMake variables starting with `OTB_USE_XXX`. Table 2.3 shows which modules are associated to these variables. It is very important to notice that these variable override the variable `OTB_BUILD_DEFAULT_MODULES`.

You are now ready to compile OTB! Simply use the make command (other targets can be generated with CMake’s `-G` option):

$ make

The installation target will copy the binaries and libraries to the installation location:

$ make install
### Table 2.3: Third parties and related modules.

<table>
<thead>
<tr>
<th>CMake variable</th>
<th>3rd party module</th>
<th>Modules depending on it</th>
</tr>
</thead>
<tbody>
<tr>
<td>OTB_USE_LIBKML</td>
<td>OTBlibkml</td>
<td>OTBKMZWriter OTBIOKML OTBApKMZ</td>
</tr>
<tr>
<td>OTB_USE_QT4</td>
<td>OTBQt4</td>
<td>OTBQtWidget</td>
</tr>
<tr>
<td>OTB_USE_OPENCV</td>
<td>OTBOpenCV</td>
<td>OTBMathParserX OTBAppMathParserX</td>
</tr>
<tr>
<td>OTB_USE_MUPARSERX</td>
<td>OTBMuParserX</td>
<td>OTBMathParserX OTBAppMathParserX</td>
</tr>
<tr>
<td>OTB_USE_OPENJPEG</td>
<td>OTBOpenJPEG</td>
<td>OTBIOJPEG2000</td>
</tr>
<tr>
<td>OTB_USE_CURL</td>
<td>OTBCurl</td>
<td></td>
</tr>
<tr>
<td>OTB_USE_MUPARSER</td>
<td>OTBMuParser</td>
<td>OTBMathParser OTBDempsterShafer OTBAppMathParser OTBAppClassification OTBAppMathParserX OTBAppSegmentation OTBAppClassification OTBRoadExtraction OTBCC8 OTBCCOBIA OTBAppSegmentation OTBMeanShift OTBAppSegmentation OTBMeanShift OTBAppSegmentation</td>
</tr>
<tr>
<td>OTB_USE_LIBSVM</td>
<td>OTBLibSVM</td>
<td>OTBSVMLearning</td>
</tr>
<tr>
<td>OTB_USE_MAPNIK</td>
<td>OTBMapnik</td>
<td>OTBVectorDataRendering</td>
</tr>
<tr>
<td>OTB_USE_6S</td>
<td>OTB6S</td>
<td>OTBOpticalCalibration OTBAppOpticalCalibration OTBSimulation</td>
</tr>
<tr>
<td>OTB_USE_SIFTFAST</td>
<td>OTBSiftFast</td>
<td></td>
</tr>
</tbody>
</table>

### 2.2 Windows

Everything that is needed for OTB development on Windows, including compiling from source, is covered in details on the OTB wiki at:


### 2.3 Known issues

- **openjpeg/ITK**

It is important to know that the OpenJpeg library doesn’t support name mangling since version 2.0. As a consequence, if other libraries linked by your project already contain OpenJpeg, there may be a symbol conflict at run-time. For instance, this was observed with OTB build on a recent ITK version (ver. 4). The ITK library already had a version of OpenJpeg in libitkopenjpeg-*.so, which contained the OpenJpeg symbols un-wrapped. These symbols were also loaded by the GDAL driver but only the first ones were used, which caused a crash.

Hopefully, thanks to the modular architecture of ITK, the library libitkopenjpeg-*.so is not imported anymore inside OTB. However the OpenJPEG headers may be present in ITK include directory. As
the current architecture doesn’t allow to tune include order between modules, the OpenJPEG header from ITK can be included before your own OpenJPEG install. There are two ways to avoid this situation:

- Use an ITK without GDCM nor ITKReview (only these modules depend on OpenJPEG)
- Hide the header openjpeg.h in the ITK include directory.

More information can be found here: http://wiki.orfeo-toolbox.org/index.php/JPEG2000_with_GDAL

- libkml / Ubuntu 12.04

Another issue is related to the official package of libkml under Ubuntu 12.4. Until this problem is addressed, users of this platform should disable the option OTB_USE_KML, so that OTB won’t be built with this third-party.
COMPILING ICE AND MONTEVERDI FROM SOURCE

3.1 Linux and Mac OS X

Compiling ICE and Monteverdi from source follows the same procedure as a regular CMake project: setup a directory for an out-of-source build, configure and compile.

3.1.1 ICE

Setup an out-of-source build environment (this guide uses ~/ICE but feel free to choose anything):

$ mkdir ~/ICE
$ cd ~/ICE
$ git clone https://git@git.orfeo-toolbox.org/git/ice.git
$ mkdir build
$ mkdir install

Remember to checkout the develop branch if you want the current development version:

$ cd ~/ICE/ice
$ git checkout develop

CMake will need to be able to find your OTB installation location. Ensure the OTB_DIR variable is set to FindOTB.cmake file location. For example, if you installed OTB in ~/OTB/install:

$ cd ~/ICE/build
$ cmake -DCMAKE_INSTALL_PREFIX=~/ICE/install \
Chapter 3. Compiling ICE and Monteverdi from source

```bash
-DOTB_DIR=~/OTB/install/lib/cmake/OTB-5.0 \ "
../ice
$ make
$ make install
```

### 3.1.2 Monteverdi

Make sure OTB is compiled with `OTB_USE_QT4` set to ON. Setup another out-of-source build environment for Monteverdi:

```bash
$ mkdir ~/monteverdi
$ cd ~/monteverdi
$ git clone https://git@git.orfeo-toolbox.org/git/monteverdi2.git
$ mkdir build
$ mkdir install
```

Remember to checkout the develop branch if you want the current development version:

```bash
$ cd ~/monteverdi/monteverdi2
$ git checkout develop
```

CMake needs to find both OTB, ICE and QWT installation locations. For example, set an CMake cache pre-population script with the following content:

```bash
# monteverdi-configuration.cmake
set(CMAKE_INSTALL_PREFIX "~/monteverdi/install" CACHE STRING "" FORCE)
set(OTB_DIR "~/OTB/install/lib/cmake/OTB-5.0" CACHE STRING "" FORCE)
set(ICE_INCLUDE_DIR "~/ICE/install/include" CACHE STRING "" FORCE)
set(ICE_LIBRARY "~/ICE/install/lib/libOTBIce.so" CACHE STRING "" FORCE)
set(QWT_INCLUDE_DIR "/usr/include/qwt5-qt4" CACHE STRING "" FORCE)
set(QWT_LIBRARY "/usr/lib64/libqwt.so.5" CACHE STRING "" FORCE)
```

Configure and compile monteverdi:

```bash
$ cd ~/monteverdi/build
$ cmake -C monteverdi-configuration.cmake ..~/monteverdi2
$ make
$ make install
```
3.2 Windows

Everything that is needed for Ice and Monteverdi development on Windows, including compiling from source, is covered in details on the OTB wiki at:

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 community. 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!

4.1 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, another module—OTB-Documents is 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.

### 4.2 Essential System Concepts

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

Generic programming is a method of organizing libraries consisting of generic—or reusable—software components [?]. 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 [?].

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.

4.2.2 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.

\texttt{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).

\texttt{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.

\texttt{itkWin32Header.h} is found in the Utilities/ITK/Code/Common and is used to define operating system parameters to control the compilation process.
4.2.3 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`.

4.2.4 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 theSmartPointer was created is closed. As a result, you should rarely if ever call `Register()` or `Delete()` in OTB. For example:

```cpp
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 aSmartPointer. Raw pointers are used for non-LightObject classes when the need for speed and/or memory demands a smaller, faster class.

### 4.2.5 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 \texttt{itk::ByteSwapper}):

\begin{verbatim}
switch ( \text{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;
}
\end{verbatim}

Note that \texttt{itk::ByteSwapperError} is a subclass of \texttt{itk::ExceptionObject}. (In fact in OTB all exceptions should be derived from \texttt{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 \texttt{ExceptionObject}.

\subsection{Event Handling}

Event handling in OTB is implemented using the Subject/Observer design pattern [?] (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 \texttt{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 \texttt{EventObject}; look in \texttt{itkEventObject.h} to determine which events are available.)

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

\begin{verbatim}
    this->InvokeEvent( ProgressEvent() );
\end{verbatim}

To watch for such an event, registration is required that associates a command (e.g., callback function) with the event: \texttt{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.)

4.2.7 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.

4.3 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.

4.4 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 4.5 on page 34 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.
4.5 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:

```cpp
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.
4.6 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 an \texttt{itk::LineSpatialObject}.

Currently implemented types of spatial objects include: Blob, Ellipse, Group, Image, Line, Surface, and Tube. The \texttt{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, \texttt{itk::TubeSpatialObject} s 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
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 programming, 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.

5.1 Hello world

5.1.1 Linux and Mac OS X

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)

cmake_minimum_required(VERSION 2.6)

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 ${OTB_LIBRARIES})

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. 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.

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

int main(int itkNotUsed(argc), char * itkNotUsed(argv)[])
{
    typedef otb::Image<unsigned short, 2> ImageType;
    ImageType::Pointer image = ImageType::New();
    std::cout << "OTB Hello World !" << std::endl;
    return EXIT_SUCCESS;
}
```

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 created and assigned to an 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 4.2.4 for more information).

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.

5.1.2 Windows

Create a directory (with write access) where to store your work (for example at C:\path\to\MyFirstCode). Organize your repository as it :

- MyFirstCode\src
- MyFirstCode\build

Follow the following steps:

1. Create a CMakeLists.txt into the src repository with the following lines:

```cmake
project(MyFirstProcessing)

cmake_minimum_required(VERSION 2.8)

find_package(OTB REQUIRED)
include(${OTB_USE_FILE})

add_executable(MyFirstProcessing MyFirstProcessing.cxx)

target_link_libraries(MyFirstProcessing ${OTB_LIBRARIES})
```

2. Create a MyFirstProcessing.cxx into the src repository with the following lines:

```cpp
#include "otbImage.h"
#include "otbVectorImage.h"
#include "otbImageFileReader.h"
```
```cpp
#include "otbImageFileWriter.h"
#include "otbMultiToMonoChannelExtractROI.h"

int main(int argc, char* argv[]) {  
  if (argc < 3)  
  {  
    std::cerr << "Usage: " << std::endl;
    std::cerr << argv[0] << " inputImageFile outputImageFile" << std::endl;
    return EXIT_FAILURE;
  }

typedef unsigned short PixelType;
typedef otb::Image <PixelType, 2> ImageType;
typedef otb::VectorImage <PixelType, 2> VectorImageType;
typedef otb::MultiToMonoChannelExtractROI <PixelType, PixelType> FilterType;
typedef otb::ImageFileReader<VectorImageType> ReaderType;
typedef otb::ImageFileWriter<ImageType> WriterType;

FilterType::Pointer filter = FilterType::New();
ReaderType::Pointer reader = ReaderType::New();
WriterType::Pointer writer = WriterType::New();

reader->SetFileName(argv[1]);
filter->SetInput(reader->GetOutput());
writer->SetFileName(argv[2]);
writer->SetInput(filter->GetOutput());

return EXIT_SUCCESS;
}

3. create a file named BuildMyFirstProcessing.bat into the MyFirstCode directory with the following lines:

@echo off

set /A ARGS_COUNT=0
for %%A in (%*) do set /A ARGS_COUNT+=1
if %ARGS_COUNT% NEQ 3 (goto :Usage)

if NOT DEFINED OSGEO4W_ROOT (goto :NoOSGEO4W)

set src_dir=%1
set build_dir=%2
```
set otb_install_dir=%3
set current_dir=%CD%

cd %build_dir%

cmake %src_dir% ^
-DCMAKE_INCLUDE_PATH:PATH="%OSGEO4W_ROOT%\include" ^
-DCMAKE_LIBRARY_PATH:PATH="%OSGEO4W_ROOT%\lib" ^
-DOTB_DIR:PATH=%otb_install_dir% ^
-DCMAKE_CONFIGURATION_TYPES:STRING=Release

cmake --build . --target INSTALL --config Release

cd %current_dir%

goto :END

:Usage
echo You need to provide 3 arguments to the script:
echo 1. path to the source directory
echo 2. path to the build directory
echo 3. path to the installation directory
GOTO :END

:NoOSGEO4W
echo You need to run this script from an OSGeo4W shell
GOTO :END

:END

4. into an OSGeo4W shell, run the configure.bat with the right arguments: full path to your src directory, full path to your build directory, full path to the place where find OTBConfig.cmake file (should be C:\path\to\MyOTBDir\install\lib\otb).

5. into the OSGeo4W shell, open the MyFirstProcessing.sln

6. build the solution

7. into the OSGeo4W shell, go to the bin\Release directory and run MyFirstProcessing.exe. You can try for example with the otb_logo.tif file which can be found into the OTB source.
5.2 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 ${OTB_LIBRARIES})
```

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:

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

int main(int argc, char * argv[])
{
    if (argc != 3)
    {
        std::cerr << "Usage: "
                   << argv[0]
                   << " <input_filename> <output_filename>"
                   << std::endl;
    }

    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::ImageFileWriter` also templated with the image type.

    ```
    typedef otb::ImageFileWriter<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.
5.3 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:

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

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

```cpp
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.

```cpp
writer->Update();
return EXIT_SUCCESS;
```

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

Get one image from the `OTB-Data/Examples` directory from the OTB-Data repository. You can get it either by cloning the OTB data repository (`git clone https://git.orfeo-toolbox.org/otb-data.git`), but that might be quite long as this also gets the data to run the tests. Alternatively, you can get it from `http://www.orfeo-toolbox.org/packages/OTB-Data-Examples.tgz`. Take 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.
### Chapter 5. Building Simple Applications with OTB

ADD_EXECUTABLE(FilteringPipeline FilteringPipeline.cxx)
TARGET_LINK_LIBRARIES(FilteringPipeline ${OTB_LIBRARIES})

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 beginning of the file is similar to the Pipeline.cxx.

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

```cpp
#include "otbImage.h"
#include "otbImageFileReader.h"
#include "otbImageFileWriter.h"
#include "itkGradientMagnitudeImageFilter.h"

int main(int argc, char * argv[])
{
    if (argc != 3)
    {
        std::cerr << "Usage: "
                    << argv[0]
                    << " <input_filename> <output_filename>"
                    << std::endl;
    }

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

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

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

typedef otb::ImageFileWriter<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:

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

Let’s plug the pipeline:

```cpp
filter->SetInput(reader->GetOutput());
writer->SetInput(filter->GetOutput());
```
And finally, we trigger the pipeline execution calling the `Update()` method on the writer

```cpp
writer->Update();
return EXIT_SUCCESS;
```

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 `itk::ImageToImageFilter` class. You will definitely find some useful filters here!

## 5.4 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:

```cpp
ADD_EXECUTABLE(ScalingPipeline ScalingPipeline.cxx)
TARGET_LINK_LIBRARIES(ScalingPipeline ${OTB_LIBRARIES})
```

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::CannyEdgeDetectionImageFilter` and the `itk::RescaleIntensityImageFilter`.

```cpp
#include "otbImage.h"
#include "otbImageFileReader.h"
#include "otbImageFileWriter.h"
#include "itkUnaryFunctorImageFilter.h"
#include "itkCannyEdgeDetectionImageFilter.h"
#include "itkRescaleIntensityImageFilter.h"
```
```c++
int main(int argc, char * argv[]) {
    if (argc != 3) {
        std::cerr << "Usage: "
                  << argv[0]
                  << " <input_filename> <output_filename>" 
                  << std::endl;
    }
}
```

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

```c++
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.

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

```c++
typedef double PixelType;
typedef otb::Image<PixelType, 2> ImageType;
typedef unsigned char OutputPixelType;
typedef otb::Image<OutputPixelType, 2> OutputImageType;
```
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 EXIT_SUCCESS;

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.

5.5 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 $(OTB_LIBRARIES))

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:
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.

```cpp
#include "otbImageFileReader.h"
#include "otbImageFileWriter.h"
#include "otbMultiToMonoChannelExtractROI.h"
#include "itkShiftScaleImageFilter.h"
#include "otbPerBandVectorImageFilter.h"

int main(int argc, char * argv[]) {
    if (argc != 4) {
        std::cerr << "Usage: " << argv[0] << " <input_filename> <output_extract> <output_shifted_scaled>" << std::endl;
    }

    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]);

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

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

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.

```cpp
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 `otb::MultiToMonoChannelExtractROI`. The declaration is as usual:

```cpp
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 extracting 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.

```cpp
reader->UpdateOutputInformation();
extracl Channel->SetExtractionRegion(
    reader->GetOutput()->GetLargestPossibleRegion());
```
We chose the channel number to extract (starting from 1) and we plug the pipeline.

```cpp
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.

```cpp
typedef otb::Image<PixelType, 2> ImageType;
typedef otb::ImageFileWriter<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.

```cpp
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::ImageFileWriter<VectorImageType> VectorWriterType;
VectorWriterType::Pointer writerVector = VectorWriterType::New();

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

writerVector->Update();

return EXIT_SUCCESS;

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

5.6 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 $(OTB_LIBRARIES))

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 \( \sigma_D \) (derivation scale) and the smoothing of the image is performed by convolving with a Gaussian kernel of variance \( \sigma_I \) (integration scale). This allows the computation of the following matrix:
\[ \mu(x, \sigma_I, \sigma_D) = \sigma_D^2 g(\sigma_I) * \begin{bmatrix} L_2^2(x, \sigma_D) & L_3^2(x, \sigma_D) \\ L_3^2(x, \sigma_D) & L_3^2(x, \sigma_D) \end{bmatrix} \]

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

We want to set 3 parameters of this filter through the command line: \( \sigma_D \) (SigmaD), \( \sigma_I \) (SigmaI) and \( \alpha \) (Alpha).

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

Let first add the two following headers:

```cpp
#include "itkMacro.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 \texttt{otb::HarrisImageFilter}. Then we start the usual main function.

```cpp
#include "otbImage.h"
#include "otbImageFileReader.h"
#include "otbImageFileWriter.h"
#include "itkUnaryFunctorImageFilter.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 \texttt{try}.

```cpp
try
{
```

Now, we can declare the \texttt{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:

```cpp
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 \texttt{AddInputImage()} and \texttt{AddOutputImage()} methods.

For the other options, we need to use the \texttt{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

```cpp
parser->SetProgramDescription("This program applies a Harris detector on the input image");
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.

```cpp
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;
}
```
5.6. Parsing command line arguments

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

```c++
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::ImageFileWriter<OutputImageType> WriterType;
WriterType::Pointer writer = WriterType::New();
```

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

```c++
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:

```c++
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.

```c++
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

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

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

Let’s plug the pipeline:
We trigger the pipeline execution calling the `Update()` method on the writer

```cpp
caller->Update();
```

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

```cpp
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.

5.7 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 ?? 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 detailed in ??.

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

ADD_EXECUTABLE(OrthoFusion OrthoFusion.cxx)
TARGET_LINK_LIBRARIES(OrthoFusion ${OTB_LIBRARIES})

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.

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

#include "otbOrthoRectificationFilter.h"
#include "otbGenericMapProjection.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> ";
    std::cout << "<x_ground_upper_left_corner> <y_ground_upper_left_corner> ";
    std::cout << "<x_Size> <y_Size> ";
    std::cout << "<x_groundSamplingDistance> ";
    std::cout << "<y_groundSamplingDistance "
      << "(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::ImageFileWriter<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]);
5.7. Going from raw satellite images to useful products

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

- the UTM zone
- the hemisphere

```cpp
typedef otb::GenericMapProjection<otb::TransformDirection::INVERSE> InverseProjectionType;
InverseProjectionType::Pointer utmMapProjection = InverseProjectionType::New();
utmMapProjection->SetWkt("Utm");
utmMapProjection->SetParameter("Zone", argv[4]);
utmMapProjection->SetParameter("Hemisphere", argv[5]);
```

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

```cpp
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:

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

OrthoRectifFilterType::Pointer orthoRectifPAN = OrthoRectifFilterType::New();
orthoRectifPAN->SetMapProjection(utmMapProjection);
orthoRectifPAN->SetInput(readerPAN->GetOutput());
orthoRectifPAN->SetOutputStartIndex(start);
orthoRectifPAN->SetOutputSize(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.
typedef otb::OrthoRectificationFilter<VectorImageType, DoubleVectorImageType, InverseProjectionType> VectorOrthoRectifFilterType;

VectorOrthoRectifFilterType::Pointer orthoRectifXS = VectorOrthoRectifFilterType::New();

orthoRectifXS->SetMapProjection(utmMapProjection);

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

orthoRectifXS->SetOutputStartIndex(start);
orthoRectifXS->SetOutputSize(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(orthoRectifXS->GetOutput());

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

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

writer->SetAutomaticTiledStreaming();

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

writer->Update();

return EXIT_SUCCESS;
Part III

User’s guide
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`.

### 6.1 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.

#### 6.1.1 Creating an Image

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

```cpp
#include "otbImage.h"

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 an itk::SmartPointer.
```

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

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.
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 an 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.

```c++
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.

### 6.1.2 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.

```c++
#include "otbImageFileReader.h"
```

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

```c++
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.

```c++
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.

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

The minimum information required by the reader is the filename of the image to be loaded in memory. This is provided through the SetFileName() method. The file format here is inferred from
the filename extension. The user may also explicitly specify the data format explicitly using the itk::ImageIO (See Chapter 7.1 99 for more information):

```cpp
cost 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.

```cpp```
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.

```cpp```
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.

### 6.1.3 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.

### 6.1.4 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 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 6.1 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.

```cpp
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.

```cpp
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.

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

```cpp
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.

```cpp
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.

```cpp
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::SmartPointer`s. Point objects are simply declared as instances of any other C++ class. Once the point is declared, its components can be accessed using traditional array notation. In particular, the `[]` operator is available. For efficiency reasons, no bounds checking is performed on the index used to access a particular point component. It is the user’s responsibility to make sure that the index is in the range `{0, Dimension − 1}`.
PointType point;
point[0] = 1.45;       // x coordinate
point[1] = 7.21;       // y coordinate

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.

```cpp
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.

### 6.1.5 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 functionalities 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.

```cpp
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.

```cpp
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.

```cpp
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;

```cpp
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 " << GCPCount << std::endl;
```
One can also get the GCPs by number, as well as their coordinates in image and geographical space.

```cpp
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.

```cpp
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;
}
```
6.1.6 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 \(?, ?\). The human retina possess different types of light sensitive cells. Three of them, known as cones, are sensitive to color \(?\) 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 rhythms are present in every cell of an organism and are known to be exquisitely precise \(?\).

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

\[
\text{ColorA} = \text{ColorB} - \text{ColorC} \tag{6.1}
\]

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 *subtract light*. So when we mention Equation 6.1 we actually mean

\[
\text{ColorB} = \text{ColorA} + \text{ColorC} \tag{6.2}
\]

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 \(?\).

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 \(?\).

---

\(1\)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.
ITK introduces the \texttt{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 \texttt{itk::Vector} in space. For this reason, the RGBPixel lack many of the operators that may be naively expected from it. In particular, there are no defined operations for subtraction or addition.

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

If you decide to interpret RGB images as simply three independent channels then you should rather use the \texttt{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 \texttt{itk::RGBPixel} class, it is necessary to include its header file.

```cpp
#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.

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

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

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

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

```cpp
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.

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

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

6.1.7 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 choosing 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.

```cpp
#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.

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

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

```cpp
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.

```cpp
ImageType::PixelType pixelValue;
pixelValue.Reserve(4);
pixelValue[0] = 1; // Blue component
pixelValue[1] = 6; // Green component
pixelValue[2] = 100; // Red 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.

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

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

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

Let's 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.

### 6.1.8 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.

```cpp
#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.

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

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

```cpp
typedef otb::ImportImageFilter<ImageType> ImportFilterType;
```

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

```cpp
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.

```cpp
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.

```cpp
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,
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.

```cpp
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.

### 6.1.9 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 functionalities 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.

```cpp
#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.

```cpp
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.

```cpp
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()`.

```cpp
ReaderType::Pointer reader = ReaderType::New();
```
We create an image list by using the `New()` method.

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

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

```cpp
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.

```cpp
// 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 randomly accessing 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.

## 6.2 PointSet

### 6.2.1 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`. 

6.2. PointSet

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 \(\{0, \text{Dimension} - 1\}\). Each of the components in the point is associated with space coordinates. The following code illustrates how to instantiate a point and initialize its components.

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

```cpp
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.

```cpp
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.

```cpp
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).

### 6.2.2 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::Point`s. It is more efficient, in general, to manage points by using the access methods provided directly on the points container. The following example illustrates how to interact with the point container and how to use point iterators.

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

```cpp
typedef PointSetType::PointsContainer PointsContainer;
```
6.2. PointSet

The actual type of the PointsContainer depends on what style of PointSet is being used. The dynamic PointSet uses 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 a 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.

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

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

```cpp
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.

```cpp
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.

```cpp
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.

```cpp
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. The Points iterator is not a reference counted class, so it is created directly from the traits without using SmartPointers.

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

---

3If you dig deep enough into the code, you will discover that these iterators are actually ITK wrappers around STL iterators.
The subsequent use of the iterator follows what you may expect from a STL iterator. The iterator to the first point is obtained from the container with the Begin() method and assigned to another iterator.

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

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

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

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

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

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

### 6.2.3 Getting Access to Data in Points

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

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;
```
6.2. PointSet

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.

```cpp
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.

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

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

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

```cpp
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.

```cpp
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.

```cpp
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.

```cpp
class PointSet
{
public:
    void SetPointData(PointData* data)
    {
        // Implementation details
    }
}
```

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.

```cpp
typedef PointDataContainer::Pointer PointDataContainer::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.

```cpp
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.

```cpp
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.

```cpp
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.

### 6.2.4 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 information. The following code shows how vector values can be used as pixel type on the PointSet class. The \texttt{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.

```cpp
#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.

```cpp
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.

```cpp
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 6.2.

```cpp
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->AddPoint(point);
    pointId = pointId + 1;
    tangentSet->AddPoint(tangent);
    tangentId = tangentId + 1;
}```
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.

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

def
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.
6.3 Mesh

6.3.1 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.

```c++
#include "itkMesh.h"
```

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

```c++
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.

```c++
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`.

```c++
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
```
6.3. Mesh

6.3.2 Inserting Cells

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

A \texttt{itk::Mesh} can contain a variety of cell types. Typical cells are the \texttt{itk::LineCell}, \texttt{itk::TriangleCell}, \texttt{itk::QuadrilateralCell} and \texttt{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.

\begin{verbatim}
#include "itkLineCell.h"
\end{verbatim}

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 \texttt{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.

\begin{verbatim}
typedef MeshType::CellType CellType;
\end{verbatim}

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

\begin{verbatim}
typedef itk::LineCell<CellType> LineType;
\end{verbatim}

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 \texttt{CellAutoPointer}. This pointer, based on the \texttt{itk::AutoPointer}, differs in many respects from the \texttt{SmartPointer}. The \texttt{CellAutoPointer} has an internal pointer to the actual object and a boolean flag that indicates if the \texttt{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.

```cpp
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.

```cpp
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.

```cpp
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

\(^4\)Some cell types like polygons have a variable number of points associated with them.
It is simply an index in the range \( \{0, \text{NumberOfPoints} - 1\} \). 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.

```cpp
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.

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

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

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

```cpp
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.

```cpp
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.

```cpp
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.

```cpp
while (cellIterator != end) {
```

```cpp```
6.3.3 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.

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

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

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

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

```cpp
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.

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

typedef MeshType::PointerType PointType;
PointerType 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.

```cpp
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.

```cpp
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.

```cpp
for (unsigned int cellId = 0; cellId < numberOfCells; cellId++)
{
    PixelType value = itk::NumericTraits<PixelType>::Zero;
    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`.

```cpp
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()`.

```cpp
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 \emph{Value()} method used to get the actual value of the data entry. \texttt{PixelType} elements are copied into the local variable \texttt{cellValue}.

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

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

## 6.4 Path

### 6.4.1 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 \texttt{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 ?? for an example in the context of alignment detection. The \texttt{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 \texttt{PolyLineParametricPath} class must be included.

```cpp
#include "itkPolyLineParametricPath.h"
```

The path is instantiated over the dimension of the image.

```cpp
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];

image->TransformPhysicalPointToContinuousIndex(origin, cindex);

path->AddVertex(cindex);

image->TransformPhysicalPointToContinuousIndex(point, cindex);

path->AddVertex(cindex);
CHAPTER
SEVEN

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.

7.1 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 functionalities which are specific to remote sensing images.

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

```cpp
#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.

```cpp
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.

```cpp
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 an `itk::SmartPointer`.

```cpp
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.

```cpp
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.

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

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

To better understand the IO architecture, please refer to Figures 7.1, 7.2, and 7.3.

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

### 7.2 Pluggable Factories

The principle behind the input/output mechanism used in ITK and therefore OTB is known as \textit{pluggable-factories} \[?\]. This concept is illustrated in the UML diagram in Figure 7.1.

\textsuperscript{1}In this example no file format is specified; this program can be used as a general file conversion utility.
Chapter 7. Reading and Writing Images

### Figure 7.2: Use cases of ImageIO factories.

![Use cases of ImageIO factories.](image.png)

### Figure 7.3: Class diagram of the ImageIO factories.

![Class diagram of the ImageIO factories.](image.png)
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 7.2. 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 7.3.

For example, in order to read PNG files, a PNGImageIOFactory is created and registered with the central ImageIOFactory singleton\(^2\) class as illustrated in the left side of Figure 7.2. When the ImageFileReader asks the ImageIOFactory for an ImageIO capable of reading the file identified with `filename` the ImageIOFactory will iterate over the list of registered factories and will ask each one of them is they know how to read the file. The factory that responds affirmatively will be used to create the specific ImageIO instance that will be returned to the ImageFileReader and used to perform the read operations.

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/](http://www.gdal.org/), is encapsulated 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](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 appropriate 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.

\(^2\) Singleton means that there is only one instance of this class in a particular application
7.3 IO Streaming

7.3.1 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 functionality 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::ImageFileWriter has to be used.

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

```cpp
#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.

```cpp
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.

```cpp
typedef otb::ImageFileReader<ImageType> ReaderType;
typedef itk::RescaleIntensityImageFilter<ImageType, ImageType> RescalerType;
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.

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

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

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

```cpp
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.

### 7.3.2 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:

```cpp
#include "otbRAMDrivenAdaptativeStreamingManager.h"
```

The `otb::RAMDrivenAdaptativeStreamingManager` 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 ??

```cpp
// typedef otb::StreamingTraits<ImageType> StreamingTraitsType;
// typedef itk::ImageRegionSplitter<2> SplitterType;
typedef otb::RAMDrivenAdaptativeStreamingManager<ImageType> StreamingManagerType;
```

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

```cpp
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.

```cpp
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.

```cpp
/*
 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.

```cpp
unsigned int piece = 0;
RegionType streamingRegion;
for (piece = 0;
     piece < numberOfStreamDivisions;
     piece++)
{
    /*streamingRegion =
     splitter->GetSplit(piece, numberOfStreamDivisions, largestRegion);
     */
    streamingRegion = streamingManager->GetSplit(piece);
    std::cout << "Processing region: " << streamingRegion << std::endl;
```

We get the region

```cpp
/*streamingRegion =
 splitter->GetSplit(piece, numberOfStreamDivisions, largestRegion);
 */
streamingRegion = streamingManager->GetSplit(piece);
std::cout << "Processing region: " << streamingRegion << std::endl;
```

We ask the reader to provide the region.

```cpp
reader->GetOutput() -> SetRequestedRegion(streamingRegion);
reader->GetOutput() -> PropagateRequestedRegion();
reader->GetOutput() -> UpdateOutputData();
```

We declare an iterator and walk through the region.
7.4. 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.

```c
#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.

```c
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.

```c
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.

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

Finally, execution of the pipeline can be triggered by invoking the `Update()` method in the writer.

```c
writer->Update();
```
You may have noticed that apart from the declaration of the `PixelType` there is nothing in this code that is specific for RGB images. All the actions required to support color images are implemented internally in the `itk::ImageIO` objects.

### 7.5 Reading, Casting and Writing Images

The source code for this 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.

```c++
#include "otbImageFileReader.h"
#include "otbImageFileWriter.h"
#include "itkUnaryFunctorImageFilter.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.

```c++
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.

```c++
typedef otb::ImageFileReader<InputImageType> ReaderType;
typedef otb::ImageFileWriter<OutputImageType> WriterType;
```

Below we instantiate the `RescaleIntensityImageFilter` class that will linearly scale the image intensities.

```c++
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.

```cpp
FilterType::Pointer filter = FilterType::New();
filter->SetOutputMinimum(0);
filter->SetOutputMaximum(255);
```

Then, we create the reader and writer and connect the pipeline.

```cpp
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.

```cpp
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.

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

## 7.6 Extracting Regions

The source code for this 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.

```cpp
#include "otbImageFileReader.h"
#include "otbImageFileWriter.h"
```
The \texttt{otb::ExtractROI} is the filter used to extract a region from an image. Its header is included below.

\begin{verbatim}
#include "otbExtractROI.h"
\end{verbatim}

Image types are defined below.

\begin{verbatim}
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;
\end{verbatim}

The types for the \texttt{otb::ImageFileReader} and \texttt{otb::ImageFileWriter} are instantiated using the image types.

\begin{verbatim}
typedef otb::ImageFileReader<InputImageType> ReaderType;
typedef otb::ImageFileWriter<OutputImageType> WriterType;
\end{verbatim}

The \texttt{ExtractROI} type is instantiated using the input and output pixel types. Using the pixel types as template parameters instead of the image types allows restricting the use of this class to \texttt{otb::Image}s which are used with scalar pixel types. See section 7.8.1 for the extraction of ROIs on \texttt{otb::VectorImage}s. A filter object is created with the \texttt{New()} method and assigned to a \texttt{itk::SmartPointer}.

\begin{verbatim}
typedef otb::ExtractROI<InputImageType::PixelType, OutputImageType::PixelType> FilterType;
FilterType::Pointer filter = FilterType::New();
\end{verbatim}

The \texttt{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 \texttt{itk::RegionOfInterestImageFilter} class).

\begin{verbatim}
filter->SetStartX(atoi(argv[3]));
filter->SetStartY(atoi(argv[4]));
filter->SetSizeX(atoi(argv[5]));
filter->SetSizeY(atoi(argv[6]));
\end{verbatim}

Below, we create the reader and writer using the \texttt{New()} method and assigning the result to a \texttt{SmartPointer}.

\begin{verbatim}
ReaderType::Pointer reader = ReaderType::New();
WriterType::Pointer writer = WriterType::New();
\end{verbatim}

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

\begin{verbatim}
reader->SetFileName(inputFilename);
writer->SetFileName(outputFilename);
\end{verbatim}
7.7. Reading and Writing Vector Images

Below we connect the reader, filter and writer to form the data processing pipeline.

```cpp
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.

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

7.7 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 type of images is the processing of SLC SAR images, which are complex.

7.7.1 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.

```cpp
#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.

```cpp
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.

```cpp
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.

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

Here we simply connect the output of the reader as input to the writer. This simple program could be used for converting complex images from one file format to another.

```cpp
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.

```cpp
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.

### 7.8 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.

```cpp
#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:

```cpp
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.

```cpp
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`.

```cpp
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.

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

We can now connect these readers and writers to filters to create a pipeline. The only thing to take care of is, when executing the program, choosing an output image file format which supports multiband images.

```cpp
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;
}
```
7.8.1 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 

\texttt{otb::MultiChannelExtractROI} and 
\texttt{otb::MultiToMonoChannelExtractROI} which allow the extraction of ROIs from multi-band images stored into \texttt{otb::VectorImage}s. The first one provides a Vector Image as output, while the second one provides a classical \texttt{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.

```cpp
#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.

```cpp
const char * inputFilename = argv[1];
const char * outputFilenameRGB = argv[2];
const char * outputFilenameMIR = argv[3];
unsigned int startX = atoi(argv[4]);
unsigned int startY = atoi(argv[5]);
unsigned int sizeX = atoi(argv[6]);
unsigned int sizeY = atoi(argv[7]);
```

As usual, we define the input and output pixel types.

```cpp
typedef unsigned char InputPixelType;
typedef unsigned char OutputPixelType;
```

First of all, we extract the multiband part by using the \texttt{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 \texttt{otb::VectorImage} type.

```cpp
typedef otb::MultiChannelExtractROI<InputPixelType, OutputPixelType> ExtractROIFilterType;
```

We create the extractor filter by using the \texttt{New} method of the class and we set its parameters.

```cpp
ExtractROIFilterType::Pointer extractROIFilter = ExtractROIFilterType::New();
extractROIFilter->SetStartX(startX);
extractROIFilter->SetStartY(startY);
extractROIFilter->SetSizeX(sizeX);
extractROIFilter->SetSizeY(sizeY);
```