First Steps in ITK
Some notes

February 15, 2020

Contents

List of Figures

List of Tables

List of Equations

Listings

1. What is this session about?

In this session, we are going to generate our first ITK program and understand the main structure (class) it uses to represent data, `Image`. Mainly, we will be using the ITK guide `InsightSoftwareGuide-Book1-*.pdf`[1], where '`*`' means current ITK version), especifically sections called "Getting Started with ITK", "Image", and the chapter on "Iterators".

1.1. Learning outcomes

- To know how to use the method for generating an executable program that uses the ITK framework by configuring and generating with `ccmake` and building with `make`.
- To understand the `itk::Image` class and know how to create an image from the scratch.
- To insight into fundamental metadata associated to image file formats commonly used in medical applications.
- To understand how iterators work and create an average filter using iterators.

2. Getting Started with ITK

We are going to work in a "CMake-like style", so we will create a directory for each task, e.g. `task01`, and inside this directory we will create a new one named `build` which acts as the *build directory* of our project. The `task*` directory includes a `CMakeLists.txt` file and a source file written in C++. The first file to place in the source directory is a `CMakeLists.txt` file that would be used by CMake to generate a Makefile (if you are using Linux/UNIX platform) or a Visual Studio workspace (if you are using Microsoft Windows). The second source file to be created is the actual C++ program.

Once both files are in your directory you can run CMake in order to configure your project. Run `ccmake` in the directory where you want the object and executable files to be placed (`build` directory). If the source directory is not the same as this build directory, in fact this is our case, you have to specify it as an argument on the command line. For example, if the current working directory is `task01` we will do the steps shown in Listing 1.

```
PROJECT_PATH/task01/$> mkdir build && cd build
PROJECT_PATH/task01/build$> cmake ..
% or
PROJECT_PATH/task01/build$> ccmake ..
```

Listing 1: Steps for configuring and building a program with Cmake (or Ccmake).

Both the terminal-based Cmake and the "GUI" (`ncurses`) version CCMake will require you to specify the directory where ITK was built in the CMake variable `ITK_DIR`. In my system, this directory is placed at `/usr/local/ITK/InsightToolkit-4.13.0/ITK-build`. The ITK binary directory contains a file named `ITKConfig.cmake` that was generated by Cmake during the ITK configuration process. From this file, CMake recovers all the information required to configure your new ITK project.

After configuring and generating, on UNIX/Linux systems the project has to be compiled by typing `make` in the terminal provided that the current working directory is equal to the project's build directory: `.../task01/build`.

Now we have described the process of generating a executable file in UNIX/Linux platforms, we just need to know the contents of the two required files: `CMakeLists.txt` and `HelloWorld.cxx` to actually generate the executable file. You can find this two files in the `ITK_DIR/Examples/Installation`

[1] The ITK guide is split into two files `InsightSoftwareGuide-Book1-*.pdf` [1] and `InsightSoftwareGuide-Book2-*.pdf` [2], from now on we'll call them *guide1* and *guide2*, respectively.

directory. In the web page of the course, concretely in the section corresponding with this practice, you can find the `CMakeLists.txt` file we'll be using in our course. This file tells Ccmake to generate a `Makefile` file that allows to link our project with ITK and VTK libraries.

Task 1

In this task our goal is to generate our first executable program and we just need to look up on the section "Getting Started with ITK" in the guide. You can see in figure 1 a screenshot of the CMake's `ncurses` GUI with the configuration options ready to generate the project. You must remember after generating the project files, on UNIX/Linux systems the project has to be compiled by typing `make` in the terminal provided that the current directory is set to the project's binary directory.

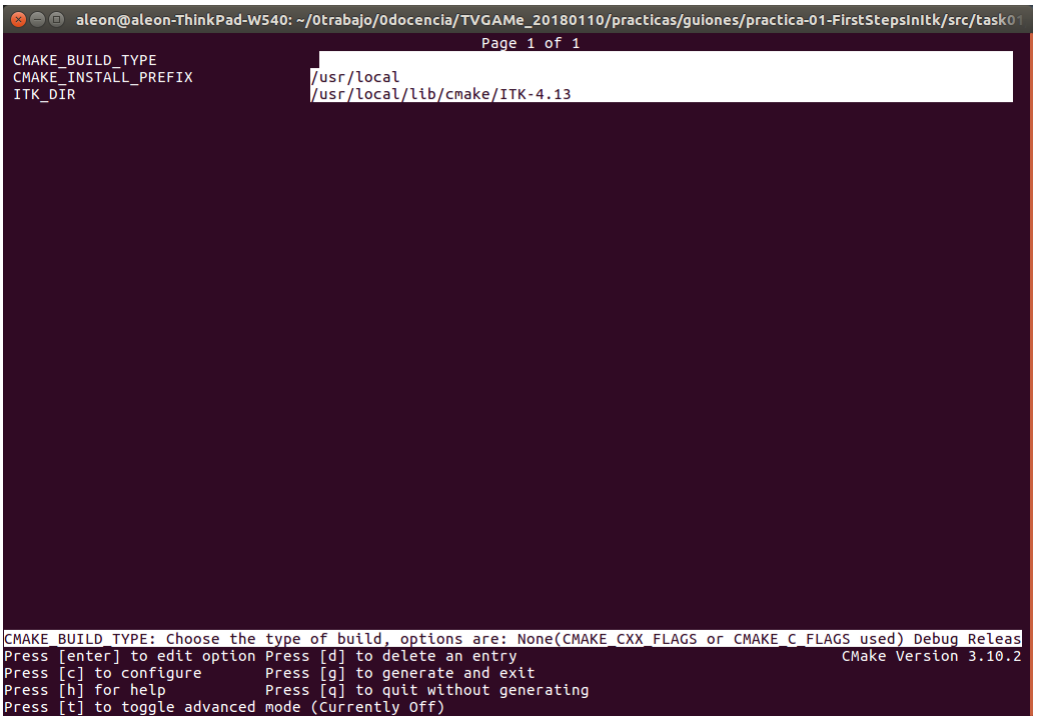

Figure 1: A project configured and ready to generate.

3. System organization

This section is extracted from the guide, mainly from chapter 3 called "System Overview", and shows the relevant information we would need in future sessions. You can look up on the guide for the rest of the information. The Insight Toolkit consists of several subsystems:

Data Representation and Access . Two principal classes are used to represent data: the `itk::Image` and `itk::Mesh` classes. We will focus on the first one.

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

IO Framework . Associated with the data processing pipeline are *sources*, filters that initiate the pipeline, and *mappers*, filters that terminate the pipeline. The standard examples of sources

and mappers are *readers* and *writers* respectively. Readers input data (typically from a file), and writers output data from the pipeline.

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

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

3.1. Data representation

There are two main types of data represented in ITK: *images* and *meshes*. This functionality is implemented in the classes `itk::Image` and `itk::Mesh`, both of which are subclasses of `itk::DataObject` .

`itk::Image` represents an n-dimensional, regular sampling of data. The sampling direction is parallel to direction matrix axes, and the origin of the sampling, inter-pixel spacing, and the number of samples in each direction (i.e., image dimension) must be specified. The sample, or pixel, type in ITK is arbitrary—a template parameter `TPixel` specifies the type upon *template instantiation*.

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:

`LargestPossibleRegion`. The whole image.

`BufferedRegion`. The portion of the image retained in memory.

`RequestedRegion`. The portion of the region requested by a filter or other class when operating on the image.

The `itk::Mesh` class represents a n-dimensional *unstructured grid*. The *topology* of the mesh is represented by a set of cells, each one defined by a type and a 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.

Mesh is a subclass of `itk::PointSet` . The PointSet class can be used to represent point clouds and has no associated topology.

3.2. Data Processing Pipeline

While *data objects* are used to represent data, *process objects* are classes that operate on data objects and may produce new data objects. Process objects are categorized 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, e.g. VTK by means of ImageToVTKImageFilter class. Typically process objects are connected together using the `SetInput()` and `GetOutput()` methods.

3.3. 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 representation of those objects. The capabilities provided by the spatial objects framework supports their use in object segmentation, registration, surface/volume rendering, and other display and analysis functions.

Currently implemented types of spatial objects include: Blob, Ellipse, Group, Image, Line, Surface, and Tube. The `itk::Scene` object is used to hold a list of spatial objects that may in turn have children. Each spatial object can be assigned a color property. Using the nominal spatial object capabilities, methods such as marching cubes or mutual information registration can be applied to objects regardless of their internal representation. By having a common API, the same method can be used to register a parametric representation of a hearth with an individual's CT data or to register two segmentations of a liver (hígado).

4. Image

This section is mainly extracted from a chapter on "Data Representation" from the ITK *guide1*. Particularly, we will be working on the contents of section "Image".

4.1. Creating an image manually

In this section we will tackle with the task of declaring and instantiating an object from the image class (`itk::Image`). We would usually leave this task to a *reader object* but it is a good idea to know the parameters required by this class, at least once in our lives! You can read the code in `ITK_DIR/Examples/DataRepresentation/Image/Image1.cxx` . The two first parameters which we would have to define are the type used to represent the pixel and the dimension that the image will have. With these two parameters we can instantiate the `Image` class as shown in listing 2. Then we can create an `Image` object and point it with a *smart pointer*.

```
// Creating two different instances of the Image class
typedef unsigned short PixelType;
typedef itk::Image<PixelType,2> ImageType2D;
typedef itk::Image<PixelType,3> ImageType3D;

// Creating two objects from the previously instantiated Image class
ImageType2D::Pointer image2D = ImageType2D::New();
ImageType3D::Pointer image3D = ImageType3D::New();
```

Listing 2: Instantiation of `itk::Image` class and creation of two objects pointed by `image2D` and `image3D` .

ITK forces us to define a third parameter since images exist in combination with one or more *regions*. As previously mentioned, a region is a subset of the image and indicates a portion that may be processed by other classes in the system, usually filter classes. Manually creating an image requires that the image is instantiated as previously shown, and then we must associate at least one region to it. When an image is created manually, the user is responsible for defining the *image size* and the *index*. These two parameters make it possible to process selected regions instead of the whole image.

A region is defined by two classes: `itk::Index` and `itk::Size` . The *origin* of the region within the image is defined by the `Index` . This index is expressed in *pixel integer coordinates* and indicates where

the *image grid* starts. The `Index` is represented by a n-dimensional array where each component is an integer indicating where is placed in *image coordinates* the first pixel of the region. The dimension of the array is determined by the `ImageType` instantiation of class `ITK::Image`. The extent, or *size*, of the region is defined by `Size`. The size indicates the number of pixels of the image along each dimension.

Once you have created `Index` and `Size`, you use this two instances to create a `itk::ImageRegion` object, and then you use the `ITK::Image`'s method `SetRegions()` to set the region for the previously created image. In the code you can notice that `IndexType`, `SizeType` and `RegionType` are defined according to `ImageType[2D,3D]`, which have been previously defined.

In listing 3 you can see the rest of the steps needed to have two Images ready to work with. The `SetRegions()` method sets the *LargestPossibleRegion*, *BufferedRegion*, and *RequestedRegion* simultaneously, and the `Allocate()` method makes the memory allocation for the image.

```
// Create two different instances of the Image class
typedef unsigned short PixelType;
typedef itk::Image<PixelType,2> ImageType2D;
typedef itk::Image<PixelType,3> ImageType3D;

// Create two objects from the previously instantiated Image class
ImageType2D::Pointer image2D = ImageType2D::New();
ImageType3D::Pointer image3D = ImageType3D::New();

// Define where the image grid starts for the 2D image
ImageType2D::IndexType start2D;
start2D[0] = 0;  // first coordinate on X in image space
start2D[1] = 0;  // first coordinate on Y in image space

// Define the number of pixels/voxels in each dimension for the 2D image
ImageType2D::SizeType size2D;
size2D[0] = 720; // number of pixels along X
size2D[1] = 348; // number of pixels along Y (A tribute to Hercules Graphics Card
    resolution!)

// Create a region for the 2D image using IndexType and SizeType
ImageType2D::RegionType region2D;
region2D.SetSize( size2D );
region2D.SetIndex( start2D );

// Do almost the same for the 3D image
ImageType3D::IndexType start3D;
start3D[0] = 0;  // first coordinate on X in image space
start3D[1] = 0;  // first coordinate on Y in image space
start3D[2] = 0;  // first coordinate on Z in image space

ImageType3D::SizeType size3D;
size3D[0] = 256; // number of voxels along X
size3D[1] = 256; // number of voxels along Y
size3D[2] = 20; // number of voxels along Z

ImageType3D::RegionType region3D;
region3D.SetSize( size3D );
region3D.SetIndex( start3D );

// Assign a valid region to each image and allocate memory
image2D->SetRegions( region2D );
image2D->Allocate();

image3D->SetRegions( region3D );
image3D->Allocate();
```

Listing 3: The listing shows all the steps needed to get an image ready to work with.

4.2. Reading an Image from a file

The most common operation that would be done at the begining of a ITK program will be to read and image. Despite we will be practicing a lot with reading images in a later lab session, now we are going to understand the basics about this. Image reader class is what we need to read an image. We need an image type to instantiate an 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.

The minimal information required by the reader is the filename of the image to be loaded into memory. This is provided through the `SetFileName()` method. The file format here is inferred from the filename extension.

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.

Access to the newly read image can be gained by calling the `GetOutput()` method on the reader. Any attempt to access image data before the reader executes will yield an image with no pixel data. You can review the code in `ITK_DIR/Examples/DataRepresentation/Image/Image2.cxx`.

Task 2

In this task we are going to learn how to read an image from a file and then show it using ItkVtkGlue capabilities. Our code, `showImage.cpp` for instance, reads and image passed by argument and displays it using `QuickView` [2]. We will replicate the structure of this code many times in order to show the results of our pipelines. Read the code from `Image2.cxx` to get a clue of how to programming `showImage.cpp`, generate the executable and use it to read and display the image `FatMRISlice.png` included in the data directory. Remember to create a directory for Task 2, `task02`, and inside it a `build` directory, next modify adequately the file `CMakeLists.txt` for including the name of the new executable and source code file.

Next, try to read `BrainProtonDensity3Slices.raw`. You get an error, especifically an exception, because ITK tried to create a reader object for the `.raw` image file format among a list of known formats—ITK shows you the list!. Try out to run your program again with `BrainProtonDensity3Slices.mha`. Now it works fine because this is a known file format for ITK. In a later session we will put on deeper with reading and writing different image file formats.

Once you have read and executed `showImage.cpp` try to write a program that reads an image from a file and `Shrinks` the image in a factor of 2, displaying both the read and the shrunk image into a viewer. For understanding `itk::ShrinkImageFilter` seek in the documentation of ITK at itk.org.

4.3. Accessing pixel data

In an image, the individual position of a pixel, which is represented by the pixel center (also called *grid point*), is identified by a unique *index*. An index is an array of integers that defines the position of the pixel along each dimension of the image in *image coordinates*. The `IndexType` is automatically defined by the image and can be accessed using the scope operator `itk::Index`. The length of the array will match the dimensions of the associated image.

[2] Check how to use it on the online documentation.

Having defined a pixel position with an index, it is then possible to get the content of the pixel in the image, `GetPixel()`, and to set the value of the pixel, `SetPixel()`. However, these two methods are relatively slow and should not be used in situations where high-performance access is required. *Image iterators* (we will present then below) are the appropriate mechanism to efficiently access image pixel data. The listing 4 shows the use of these methods.

```
// Create an instance of the Image class
typedef unsigned short PixelType;
typedef itk::Image<PixelType,2> ImageType2D;

// Instantiate an object from the previously instantiated Image class
ImageType2D::Pointer image2D = ImageType2D::New();

// Define where the image grid starts
ImageType2D::IndexType start2D;
start2D[0] = 0;  // first X coordinate in Image Space
start2D[1] = 0;  // first Y coordinate in Image Space

// Define the number of pixels in each dimension
ImageType2D::SizeType size2D;
size2D[0] = 200; // number of pixels along X
size2D[1] = 200; // number of pixels along Y

// Create a region for the two images from start2D and size2D
ImageType2D::RegionType region2D;

region2D.SetSize(size2D);
region2D.SetIndex(start2D);

// Assign a valid region to the image and allocate memory for it
image2D->SetRegions(region2D);
image2D->Allocate();

// Declare an instance of the index type automatically
// defined by the ImageType2D and initialize it.
const ImageType2D::IndexType pixelIndex2D = {{10,10}}; // Position of pixel centered
    at (10,10)

// Get and set pixel values
ImageType2D::PixelType pixelValue = image2D->GetPixel(pixelIndex2D);
image2D->SetPixel(pixelIndex2D,pixelValue + 1);
```

Listing 4: The listing shows how to access pixel data.

4.4. Defining Origing and Spacing

ITK is mainly developed for processing medical image data, hence it is mandatory to associate additional information to images. Particularly useful is the information associated with the *physical spacing between pixels* and the *position of the image in space* with respect to some *World Coordinate System* (WCS), together with the orientation of the axes of this coordinate system (*direction matrix* or *grid rotation matrix* with respect to the origin of the WCS). All this information defines de *geometry of the image* in ITK. If you don't have the image geometry information (spatial information), you will not be able to carry out things like image registration, medical diagnosis, image analysis, feature extraction, assisted radiation therapy or image guided surgery.

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 (grid points). The *grid constant* we will spoke about in the Digital Image Processing (DIP) seminar,

particularly in neighborhood and adjacency relationship, is called *spacing* in ITK. The spacing might vary in each image dimension, instead of the theoretical grid constant that is considered to be 1 and equal in all dimensions for convenience.

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

Read the section "Defining Origin and Spacing" in the *guide1* to understand these different parameters and review the code in the file `ITK_DIR/Examples/DataRepresentation/Image/Image4.cxx`

Task 3

In this task we will write a program to extract information from a image that is provided as an argument and then we will display it. Concretely, we will show the following data: image size for each dimension, origin, spacing, and direction matrix. Look at the code in `image4.cxx` to know how to write the program. We will be working with the following images both as arguments to our program and for editing them:

- `BrainProtonDensity3Slices.mha`
- `BrainProtonDensity3Slices.raw`
- `BrainProtonDensitySliceBorder20.mhd`
- `BrainProtonDensitySliceBorder20.raw`
- `BrainProtonDensitySliceBorder20DirectionPlus30.mhd`
- `BrainProtonDensitySliceBorder20.zraw`
- `BrainProtonDensitySliceBorder20DirectionPlus30.nhdr`
- `BrainProtonDensitySliceBorder20DirectionPlus30.raw`

The images above have been extracted from the `Data` directory which is inside the ITK installation directory. You can read the files `BrainProtonDensitySliceBorder20.mhd` and `BrainProtonDensitySliceBorder20DirectionPlus30.mhd` in a text editor, `gedit` for instance. This two files are *metadata files*, which describe information associated with the real image data.

For example, in `BrainProtonDensitySliceBorder20.mhd` you can see "`NDims = 2`" data field that represents the image dimension, the "`ElementSpacing = 1 1`" that shows the spacing in each dimension, the "`DimSize = 221 257`" that holds the number of pixels in each dimension, and the "`ElementType = MET_UCHAR`" that represent the data type used to save the pixel values in the data file. OK, we have two extra fields particularly reserved to hold information about the file that actually saves the pixel values: "`CompressedData = False`" and "`ElementDataFile = BrainProtonDensitySliceBorder20.raw`".

The `BrainProtonDensitySliceBorder20DirectionPlus30.nhdr` is another kind of metadata file (`.nhdr`) commonly used in medical imaging. You can read it using `gedit` and compare the metadata information with the `.mhd` format. Try out the program you have written with the above images and compare the information it display with the information saved in `.mha` , `.mhd` and `.nhdr` formats.

4.5. Accessing pixel data by an iterator

This section is excerpted from the ITK *guide1*, specifically from chapter on "Iterators". In ITK, we use iterators the same way as usual, especifically to instantiate images with different combinations of pixel type, pixel container type, and dimensionality. Using the ITK iterators instead of accessing data directly through the `itk::Image` interface permits algorithms run much faster and improves neighborhood-based image processing (called *masks* or *kernels* in DIP jargon).

4.5.1. Creating iterators

An iterator constructor requires at least two arguments, a smart pointer to the image to iterate across, and an image region (*iteration region*) where the iterator works. A valid iteration region is any subregion of the image within the current `BufferedRegion` .

There is a const and a non-const version of most ITK image iterators. A non-const iterator cannot be instantiated on a non-const image pointer. Const versions of iterators may read, but may not write pixel values.

4.5.2. Moving iterators

At any time, the iterator will reference one pixel location in the iteration region of the image. *Forward iteration* goes from the beginning of the iteration region to the end of the iteration region. *Reverse iteration*, goes from just past the end of the region back to the beginning. There are two corresponding starting positions for iterators, the *begin* position and the *end* position. Note that the end position is not actually located within the iteration region, $[begin, end)$. This is important to remember because attempting to dereference an iterator at its end position will have undefined results. An iterator first moves across columns, then down rows, then from slice to slice, and so on—as the scanning order of ancient CRT displays. The common use of iterators is by mean of the following methods:

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

`GoToEnd()` . Points the iterator to one position past the last valid element in the region.

`operator++()` . Increments the iterator one position in the positive direction.

`operator--()` . Decrements the iterator one position in the negative direction.

`bool IsAtEnd()` . True if the iterator points to one position past the end of the iteration region.

`bool IsAtBegin()` . True if the iterator points to the first position in the iteration region.

In addition to sequential iteration through the image, some iterators may define random access operators. Unlike the increment operators, random access operators may not be optimized for speed and require some knowledge of the dimensionality of the image and the extent of the iteration region to use properly.

`operator+=( OffsetType )` . Moves the iterator to the pixel position at the current index plus specified `itk::Offset`.

`operator-=( OffsetType )` . Moves the iterator to the pixel position at the current index minus specified `itk::Offset`.

`SetPosition( IndexType )`. Moves the iterator to the given `itk::Index` position.

`IndexType GetIndex()`. Returns the `itk::Index` of the image pixel that the iterator currently references to.

4.5.3. Accessing data

ITK image iterators define two basic methods for reading and writing pixel values and an optimized dereference once method for concrete situations.

`PixelType Get()`. Returns the value of the pixel at the iterator position.

`void Set( PixelType )`. Sets the value of the pixel at the iterator position. Not defined for const versions of iterators.

`PixelType &Value()`. Returns a reference to the pixel at the iterator position. The `Value()` method can be used as either an *lval* or an *rval* in an expression.

The following listing (See listing 5) shows how to use iterators in a simple example:

```
// Create an instance of the Image class
typedef float PixelType;
typedef itk::Image<PixelType,2> ImageType2D;

// Declare a smart pointer to itk::RandomImageSource<Image2DType>
itk::RandomImageSource<Image2DType>::Pointer random;

// Create a new object of type itk::RandomImageSource<Image2DType>
random = itk::RandomImageSource<Image2DType>::New();

// Set the minimum/maximum pixel value
random->SetMin(0.0);
random->SetMax(1.0);

// Define and set the random image's size
Image2DType::SizeValueType size[2];
size[0] = 20;
size[1] = 20;
random->SetSize(size);

// Explicitly invoke the Update() method to update de pipeline.
random->Update();

// Declare an smart pointer to point to a new created object of type Image2DType
Image2DType::Pointer outputImage2D = Image2DType::New();

// Define where the image grid starts
Image2DType::IndexType region2DIndex;
region2DIndex[0] = 0;  // first coordinate on X in image space
region2DIndex[1] = 0;  // first coordinate on Y in image space

// Define the number of pixels in each dimension
Image2DType::SizeType region2DSize;
region2DSize[0] = 20; // number of pixels along X
region2DSize[1] = 20; // number of pixels along Y

// Create a region for the image from 'region2DIndex' and with 'region2DSize'
Image2DType::RegionType region2D;

region2D.SetIndex( region2DIndex );
```

```
region2D.SetSize( region2DSize );

// Assign a valid region to the image and allocate memory
outputImage2D->SetRegions( region2D );
outputImage2D->Allocate();

// Declare and define two new types of iterators across 'Image2DType'
typedef itk::ImageRegionConstIterator< Image2DType > ConstIterator2DType;
typedef itk::ImageRegionIterator< Image2DType > Iterator2DType;

// Make an Image pointer points to the output of random filter
// Remember this filter has as output an itk::Image data rep
Image2DType::Pointer inputImage2D = random->GetOutput();

// Define the iterators across 'inputImage2D' and 'outputImage2D'
ConstIterator2DType in( inputImage2D, inputImage2D->GetRequestedRegion() );
Iterator2DType out( outputImage2D, inputImage2D->GetRequestedRegion() );

for ( in.GoToBegin(), out.GoToBegin(); !in.IsAtEnd(); ++in, ++out ) {
  out.Set( in.Get() * in.Get() );

    // Print stdout the value of the current input and output pixels
    float inputValue = in.Get();
    float outputValue = out.Get();
    std::cout << inputValue << "\t"  << outputValue << std::endl;
}


  return EXIT_SUCCESS;
}
```

Listing 5: Using iterators.

Task 4

Try to generate an executable from the code shown in the listing 5 and note how to manually create an `itk::Image`, how to use an Image pointer to "raw" access to the Image data generated from a filter, and how to use iterators.

5. Image iterators

This section describes iterators that go across rectilinear image regions and reference a single pixel at a time. The `itk::ImageRegionIterator` is the most basic ITK image iterator and the first choice for most applications. The rest of the iterators in this section are specializations of ImageRegionIterator that are designed to make common image processing tasks more efficient or easier to implement.

5.1. `ImageRegionIterator`

You can see the code that shows how to use a `itk::ImageRegionIterator` in `ITK_DIR/Examples/Iterators/ImageRegionIterator.cxx`

The `itk::ImageRegionIterator` is optimized for iteration speed and is the first choice for iterative, pixel-wise operations when location in the image is not important. The application crops a subregion from an image by copying its pixel values into to a second, smaller image. You can find the image argument of `ImageRegionIterator.cxx` in `ITK_DIR/Examples/Data/FatMRISlice.png`.

Try to generate the example and execute it with the parameters indicated at the top of the source code file. Note that we define at the begining of the source code the `inputRegion` and

outputRegion with the same size, $(argv[5], argv[6])$ but with different starting coordinates in image space, $(argv[3], argv[4])$ and $(0, 0)$ respectively. Another interesting point is how to check whether the cropped region is inside the inputImage or not. The following listing shows how they do it (see listing 6.

```
// Check that the region is contained within the input image.
if ( ! reader->GetOutput()->GetRequestedRegion().IsInside( inputRegion ) )
  {
  std::cerr << "Error" << std::endl;
  std::cerr << "The region " << inputRegion << "is not contained within "
            << "the input image region "
            << reader->GetOutput()->GetRequestedRegion() << std::endl;
  return -1;
  }
```

Listing 6: Checking whether a region is contained within an image.

Notice the way they set the geometric coordinates of the outputImage using the methods for getting the origing, GetOrigin() , and spacing, GetSpacing() , of the Image resulting from the reader object. The following listing shows how they do it (look at listing 7. For each dimension, inputOrigin[i] represents the geometric coordinate of the pixel located at index $[0, 0]$ in image coordinates; spacing[i] represents the units (e.g., cm, mm,...) between two neighboring pixels' centre; and inputStart[i] represents the index in image space where the cropped region starts.

```
  const ImageType::SpacingType& spacing = reader->GetOutput()->GetSpacing();
  const ImageType::PointType& inputOrigin = reader->GetOutput()->GetOrigin();
  double    outputOrigin[ Dimension ];

  for(unsigned int i=0; i< Dimension; i++)
    {
    outputOrigin[i] = inputOrigin[i] + spacing[i] * inputStart[i];
    }

  outputImage->SetSpacing( spacing );
  outputImage->SetOrigin(  outputOrigin );
```

Listing 7: Assigning origing and spacing to a new image.

5.2. ImageRegionIteratorWithIndex

The "WithIndex" family of iterators was designed for algorithms that use both the value and the location of image pixels in calculations. Unlike itk::ImageRegionIterator , which calculates an index only when asked for, itk::ImageRegionIteratorWithIndex maintains its index location as a member variable that is updated during the increment or decrement process. Iteration speed is penalized, but the index queries are more efficient.

You can see the code that shows how to use a itk::ImageRegionIteratorWithIndex in ITK_DIR/Examples/Iterators/ImageRegionIteratorWithIndex.cxx

Try to generate and execute that example. Notice the method to copy all the information from a itk::Image of the same ImageType . The listing below shows how they do it (see listing 8.

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

Listing 8: A useful way of creating a new image from another well-formed one.

5.3. ImageLinearIteratorWithIndex

The itk::ImageLinearIteratorWithIndex is designed for line-by-line processing of an image. It walks a linear path along a selected image direction parallel to one of the coordinate axes of the image. Like all image iterators, movement of the ImageLinearIteratorWithIndex is constrained within an image region R. The line l through which the iterator moves is defined by selecting a direction and an origin. The line l extends from the origin to the upper boundary of R. The origin can be moved to any position along the lower boundary of R. This iterator has a new set of methods you can see in the ITK guide. The example ImageLinearIteratorWithIndex2.cxx is worth the effort it take to read it.

5.4. ImageSliceIteratorWithIndex

The itk::ImageSliceIteratorWithIndex class is an extension of itk::ImageLinearIteratorWithIndex from iteration along lines to *iteration along both lines and planes in an image*. A slice is a 2D plane spanned by two vectors pointing along orthogonal coordinate axes. The slice orientation of the slice iterator is defined by specifying its two spanning axes.

SetFirstDirection(). Specifies the first coordinate axis direction of the slice plane.

SetSecondDirection(). Specifies the second coordinate axis direction of the slice plane.

The slice iterator moves line by line using NextLine() and PreviousLine() . The line direction is parallel to the second coordinate axis direction of the slice plane. The code example ImageSliceIteratorWithIndex.cxx calculates the *maximum intensity projection* along one of the coordinate axes of an image volume. The algorithm is straightforward using ImageSliceIteratorWithIndex because we can coordinate movement through a slice of the 3D input image with movement through the 2D planar output.

Here is how the algorithm works. For each 2D slice of the input, iterate through all the pixels line by line. Copy a pixel value to the corresponding position in the 2D output image if it is larger than the value already contained there. When all slices have been processed, the output image is the desired maximum intensity projection. The **projection direction** is read from the command line. The projection image will be the size of the 2D plane orthogonal to the projection direction. Its spanning vectors are the two remaining coordinate axes in the volume. Try to understand how the projection and the outputImage dimensions are calculated in the code showed in the ITK guide.

Run this example on the 3D image ITK_DIR/Examples/Data/BrainProtonDensity3Slices.mha using the z-axis as the axis of projection.

Task 5

Write a program that accepts two arguments, an image file and a kernel size, and applies an average filter to the input image, showing the results using a viewer—use QuickView class.

References

[1] Johnson, H. J., McCormick, M. M., and Ibanez, L. *The ITK Software Guide Book 1: Introduction and Development Guidelines-Volume 1 4th ed.*, 2017.

[2] Johnson, H. J., McCormick, M. M., and Ibanez, L. *The ITK Software Guide Book 2: Design and Functionality-Volume 2 4th ed.*, 2017.