ITK Filters: How to Write Them

Methods in Medical Image Analysis - Spring 2020
16-725 (CMU RI): BioE 2630 (Pitt)
Dr. John Galeotti

Based in part on Damion Shelton’s slides from 2006

Where we are

- You should understand
  - What the pipeline is and how to connect filters together to perform sequential processing
  - How to move through images using iterators
  - How to access specific pixels based on their location in data space or physical space
What we’ll cover

- How to write your own filter that can fit into the pipeline
- For reference, read Chapters 6 & 8 from book 1 of the ITK Software Guide

Is it hard or easy?

- Writing filters can be really, really easy
- But, it can also be tricky at times
- Remember, don’t panic!
“Cheat” as much as possible!

- Never, ever, ever, write a filter from scratch
- Unless you’re doing something really odd, find a filter close to what you want and work from there
- Recycling the general framework will save you a lot of time and reduce errors

Much of the filter is already written

- Most of the interface for an \texttt{ImageToImageFilter} is already coded by the base classes
- For example, \texttt{SetInput} and \texttt{GetOutput} are not functions you have to write
- You should never have to worry about particulars of the pipeline infrastructure.
The simple case

- You can write a filter with only one\(^*\) function!
  - (* well, sort of)
- Overload \texttt{GenerateData(void)} to produce output given some input
- We’ll look at \texttt{BinomialBlurImageFilter} as an example
  - Located in SimpleITK-build/ITK/Modules/Filtering/Smoothing/include

The header - stuff that’s “always there”

- \texttt{itkNewMacro} sets up the object factory (for reference counted smart pointers)
- \texttt{itkTypeMacro} allows you to use run time type information
- \texttt{itkGetConstMacro} and \texttt{itkSetMacro} are used to access private member variables
The header cont.

- Prototypes for functions you will overload:
  
  ```cpp
  void PrintSelf(std::ostream& os, 
                 Indent indent) const;
  
  void GenerateData(void);
  ```

- For multi-threaded filters, the latter will instead be:

  ```cpp
  ThreadedGenerateData(void);
  ```

More header code

- You will also see:
  - Many typedefs, some of which are particularly important:
    ```cpp
    Self
    Superclass
    Pointer
    ConstPointer
    ```
  - Constructor and destructor prototypes
  - Member variables (in this example, only one)

- Things not typically necessary:
  ```cpp
  GenerateInputRequestedRegion()
  ```
  - Concept checking stuff
Pay attention to...

- `#ifdef #define #endif` are used to enforce single inclusion of header code
- Use of `namespace itk`
- The three lines at the bottom starting with:
  ```
  #ifndef ITK_MANUAL_INSTANTIATION
  ```
  control whether the `.hxx` file should be included with the `.h` file.
- There are often three lines just before that, starting with `#if ITK_TEMPLATE_EXPLICIT`, which allow for explicitly precompiling certain combinations of template parameters.

Does this seem complex?

- That’s why I suggested always starting with an existing class
- You may want to use find and replace to change the class name and edit from there
- Moving on to the `.hxx` file...
The constructor

- In BinomialBlurImageFilter, the constructor doesn’t do much
  - Initialize the member variable

GenerateData()

- This is where most of the action occurs
- GenerateData() is called during the pipeline update process
- It’s responsible for allocating the output image (though the pointer already exists) and filling the image with interesting data
Accessing the input and output

- First, we get the pointers to the input and output images

```c
InputImageConstPointer inputPtr =
    this->GetInput(0);

OutputImagePointer outputPtr =
    this->GetOutput(0);
```

Filters can have multiple inputs or outputs, in this case we only have one of each

Allocating the output image

```c
outputPtr->SetBufferedRegion(
    outputPtr->GetRequestedRegion());

outputPtr->Allocate();
```
The meat of GenerateData()

- Make a temporary copy of the input image
- Repeat the desired number of times for each dimension:
  - Iterate forward through the image, averaging each pixel with its following neighbor
  - Iterate backward through the image, averaging each pixel with its preceding neighbor
- Copy the temp image’s contents to the output
- We control the number of repetitions with `m_Repetitions`

PrintSelf

- *PrintSelf* is a function present in all classes derived from *itk::Object* which permits easy display of the “state” of an object (i.e. all of its member variables)
- ITK’s testing framework requires that you implement this function for any class containing non-inherited member variables
  - Otherwise your code will fail the “PrintSelf test”...
  - If you try to contribute your code to ITK
- **Important:** users should call `Print()` instead of `PrintSelf()`
PrintSelf, cont.

- First, we call the base class implementation
  ```cpp
  Superclass::PrintSelf(os, indent);
  ```
  This is the only time you should ever call `PrintSelf()` directly!

- And second we print all of our member variables
  ```cpp
  os << indent << "Number of Repetitions: " << m_Repetitions << std::endl;
  ```

Questions?

- How can we make multithreaded filters?
- What if the input and output images are not the same size? E.g., convolution edge effects, subsampling, etc.
- What about requested regions?

We’ll address these questions when we discuss advanced filters
Another Question for Today

How do I deal with neighborhoods in N-Dimensions...

Such as for convolution?

Neighborhoods in ITK

- An ITK neighborhood can be any collection of pixels that have a fixed relationship to the “center” based on offsets in data space.
  - Not limited to the max- or min-connected immediately neighboring pixels!
- See 6.4 in the ITK Software Guide, book 1
Neighborhoods in ITK, cont.

- In general, the neighborhood is not completely arbitrary
  - Neighborhoods are rectangular, defined by a “radius” in N-dimensions
  - Shaped Neighborhoods are more arbitrary, defined by a list of offsets from the center
- The first form is most useful for mathematical morphology kinds of operations, convolution, etc.

Neighborhood iterators

- The cool & useful thing about neighborhoods is that they can be used with neighborhood iterators to allow efficient access to pixels “around” a target pixel in an image
Neighborhood iterators

- Remember that I said access via pixel indices was slow?
  - Get current index = $l$
  - Upper left pixel index $l_{UL} = l - (1,1)$
  - Get pixel at index $l_{UL}$
- Neighborhood iterators solve this problem by doing pointer arithmetic based on offsets

Neighborhood layout

- Neighborhoods have one primary vector parameter, their “radius” in N-dimensions
- The side length along a particular dimension $i$ is $2 \times \text{radius}_i + 1$
- Note that the side length is always odd because the center pixel always exists
Neighborhoods have another parameter called **stride** which is the spacing (in data space) along a particular axis between adjacent pixels in the neighborhood.

- In the previous numbering scheme, stride in $Y$ is amount then index value changes when you move in $Y$.
- In our example, $\text{Stride}_x = 1$, $\text{Stride}_y = 5$. 

A 3x5 neighborhood in 2D

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

Stride
Neighborhood pixel access

- The **lexicographic** numbering on the previous diagram is important!
  - It’s ND
  - It’s how you index (access) that particular pixel when using a neighborhood iterator
- This will be clarified in a few slides...

NeighborhoodIterator access

- Neighborhood iterators are created using:
  - The radius of the neighborhood
  - The image that will be traversed
  - The region of the image to be traversed
- Their syntax largely follows that of other iterators (++, IsAtEnd(), etc.)
Neighborhood pixel access, cont.

Let's say there's some region of an image that has the following pixel values:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2</td>
<td>1.3</td>
<td>1.8</td>
<td>1.4</td>
<td>1.1</td>
</tr>
<tr>
<td>1.8</td>
<td>1.1</td>
<td>0.7</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2.1</td>
<td>1.9</td>
<td>1.7</td>
<td>1.4</td>
<td>2.0</td>
</tr>
</tbody>
</table>

Pixel access, cont.

- Now assume that we place the neighborhood iterator over this region and start accessing pixels.
- What happens?
Pixel access, cont.

*myNeigh.GetPixel(7) returns 0.7*
so does *myNeigh.GetCenterPixel()*

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2</td>
<td>1.3</td>
<td>1.8</td>
<td>1.4</td>
<td>1.1</td>
<td>1.8</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>1.8</td>
<td>1.1</td>
<td>0.7</td>
<td>1.0</td>
<td>1.0</td>
<td>1.7</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>2.1</td>
<td>1.9</td>
<td>1.7</td>
<td>1.4</td>
<td>2.0</td>
<td>1.0</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>33</td>
</tr>
</tbody>
</table>

Intensity of currently underlying pixel in the image

*lexicographic index within neighborhood*

Get the length & stride length of the iterator:

*Size()* returns the #pixels in the neighborhood
Ex: find the center pixel’s index:

```c
unsigned int c = iterator.Size() / 2;
```

*GetStride()* returns the stride of dimension N:

```c
unsigned int s = iterator.GetStride(1);
```
Pixel access, cont.

\[\text{myNeigh.GetPixel}(c) \text{ returns 0.7}\]
\[\text{myNeigh.GetPixel}(c-1) \text{ returns 1.1}\]

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2</td>
<td>1.3</td>
<td>1.8</td>
<td>1.4</td>
<td>1.1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1.8</td>
<td>1.1</td>
<td>0.7</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>2.1</td>
<td>1.9</td>
<td>1.7</td>
<td>1.4</td>
<td>2.0</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

Pixel access, cont.

\[\text{myNeigh.GetPixel}(c-s) \text{ returns 1.8}\]
\[\text{myNeigh.GetPixel}(c-s-1) \text{ returns 1.3}\]

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2</td>
<td>1.3</td>
<td>1.8</td>
<td>1.4</td>
<td>1.1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1.8</td>
<td>1.1</td>
<td>0.7</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>2.1</td>
<td>1.9</td>
<td>1.7</td>
<td>1.4</td>
<td>2.0</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>
The ++ method

- In Image-Region Iterators, the ++ method moves the focus of the iterator on a per pixel basis
- In Neighborhood Iterators, the ++ method moves the center pixel of the neighborhood and therefore implicitly shifts the entire neighborhood

An aside: “regular” iterators

- Regular ITK Iterators are also lexicographic
  - That is how they, too, are ND
- The stride parameters are for the entire image
- Conceptual parallel between:
  - ITK mapping a neighborhood to an image pixel in an image
  - Lexicographically unwinding a kernel for an image
- The linear pointer arithmetic is very fast!
  - Remember, all images are stored linearly in RAM
Convolution (ahem, correlation)!

To do correlation we need 3 things:

1. A kernel
2. A way to access a region of an image the same size as the kernel
3. A way to compute the inner product between the kernel and the image region

Item 1 - the kernel

- A **NeighborhoodOperator** is a set of pixel values that can be applied to a Neighborhood to perform a user-defined operation (i.e. convolution kernel, morphological structuring element)

- **NeighborhoodOperator** is derived from **Neighborhood**
Item 2 - image access method

- We already showed that this is possible using the neighborhood iterator
- Just be careful setting it up so that it’s the same size as your kernel

Item 3 - inner product method

- The `NeighborhoodInnerProduct` computes the inner product between two neighborhoods
- Since `NeighborhoodOperator` is derived from `Neighborhood`, we can compute the IP of the kernel and the image region
Good to go?

1. Create an interesting operator to form a kernel
2. Move a neighborhood through an image
3. Compute the IP of the operator and the neighborhood at each pixel in the image

Voila – correlation in N-dimensions

---

Inner product example

```cpp
itk::NeighborhoodInnerProduct<ImageType> IP;

itk::DerivativeOperator<TPixel,
    ImageType::ImageDimension>
operator ;

operator->SetOrder(1);
operator->SetDirection(0);
operator->CreateDirectional();

itk::NeighborhoodIterator<ImageType> iterator(
    operator->GetRadius(),
    myImage,
    myImage->GetRequestedRegion() ) ;
```
Inner product example, cont.

```cpp
iterator.SetToBegin();
while ( ! iterator. IsAtEnd () )
{
    std::cout << "Derivative at index "
    << iterator.GetIndex ()
    << " is " << IP(iterator, operator)
    << std::endl;
    ++iterator;
}
```

Note

- No explicit reference to dimensionality in neighborhood iterator
- Therefore easy to make N-d
This suggests a filter...

- **NeighborhoodOperatorImageFilter** wraps this procedure into a filter that operates on an input image
- So, if the main challenge is coming up with an interesting neighborhood operator, ITK can do the rest

Your arch-nemesis...
image boundaries

- One obvious problem with inner product techniques is what to do when you reach the edge of your image
- Is the operation undefined?
- Does the image wrap?
- Should we assume the rest of the world is empty/full/something else?
ImageBoundaryCondition

- Subclasses of ImageBoundaryCondition can be used to tell neighborhood iterators what to do if part of the neighborhood is not in the image.

ConstantBoundaryCondition

- The rest of the world is filled with some constant value of your choice.
- The default is 0.
- Be careful with the value you choose - you can (for example) detect edges that aren’t really there.
PeriodicBoundaryCondition

- The image wraps, so that if I exceed the length of a particular axis, I wrap back to 0 and start over again.
- If you enjoy headaches, imagine this in 3D.
- This isn’t a bad idea, but most medical images are not actually periodic.

ZeroFluxNeumannBoundaryCondition

- This is the default boundary condition.
- Simply returns the closest in-bounds pixel value to the requested out-of-bounds location.
- Important result: the first derivative across the boundary is zero.
  - Thermodynamic motivation
  - Useful for solving certain classes of diff. eq.
Using boundary conditions

- `NeighborhoodIterator` automatically determines whether or not it needs to enable bounds checking when it is created (i.e. constructed).

- `SetNeedToUseBoundaryCondition(true/false)`
  - Manually forces or disables bounds checking

- `OverrideBoundaryCondition()`
  - Changes which boundary condition is used
  - Can be called on both:
    - `NeighborhoodIterator`
    - `NeighborhoodOperatorImageFilter`

Last Major Question
(for today, anyway)

How do I do math with different pixel types...
Answer: numeric traits

- Provide various bits of numerical information about arbitrary pixel types.
- Usage scenario:
  - “What is the max value of the current pixel type?”
- Need to know these things at compile time, but templated pixel types make this hard.
- Numeric traits provide answers that are “filled in” at compilation for our pixel type.

**itk::NumericTraits**

- NumericTraits is class that is specialized to provide information about pixel types
- Examples include:
  - Min and max, epsilon and infinity values
  - Definitions of Zero and One
    - (i.e., Additive and multiplicative identities)
  - **IsPositive**, **IsNegative** functions
- See also:
  - Modules/ThirdParty/VNL/src/vxl/vcl/emulation/vcl_limits.h
  - http://www.itk.org/Doxygen/html/classitk_1_1NumericTraits.html
  - http://www.itk.org/Wiki/ITK/Examples/SimpleOperations/numericTraits
Using traits

- What’s the maximum value that can be represented by an `unsigned char`?
  - `itk::NumericTraits<unsigned char>::max()

- What about for our pixel type?
  - `itk::NumericTraits<PixelType>::max()

Get used to coding like this!

Excerpt from
http://www.itk.org/Wiki/ITK/Examples/SimpleOperations/NumericTraits

```cpp
#include "itkNumericTraits.h"

// ...
std::cout << "Min: " << itk::NumericTraits<float>::min() << std::endl;
std::cout << "Max: " << itk::NumericTraits<float>::max() << std::endl;
std::cout << "Zero: " << itk::NumericTraits<float>::Zero << std::endl;
std::cout << "Zero: " << itk::NumericTraits<float>::ZeroValue() << std::endl;
std::cout << "Is -1 negative? " << itk::NumericTraits<float>::IsNegative(-1) << std::endl;
std::cout << "Is 1 negative? " << itk::NumericTraits<float>::IsNegative(1) << std::endl;
std::cout << "One: " << itk::NumericTraits<float>::One << std::endl;
std::cout << "Epsilon: " << itk::NumericTraits<float>::epsilon() << std::endl;
std::cout << "Infinity: " << itk::NumericTraits<float>::infinity() << std::endl;
// ...
```