Lecture 19
Write Your Own ITK Filters, Part 2

Methods in Medical Image Analysis - Spring 2018
16-725 (CMU RI): BioE 2630 (Pitt)
Dr. John Galeotti

Based in part on Damion Shelton's slides from 2006

What are “advanced” filters?

- More than one input
- Support progress methods
- Output image is a different size than input
- Multi-threaded

Details, details

- In the interests of time I’m going to gloss over some of the finer details
- I’d like to make you aware of some of the more complicated filter issues, but not scare you away
- See book 1, chapter 8 of the software guide!

Different output size

- Overload GenerateOutputInformation()
- This allows you to change the output image’s:
  - Largest possible region (size in pixels)
  - Origin & spacing
- By default, the output image has the same size, origin, and spacing as the input
- See:
  Modules/Filtering/ImageGrid/include/itkShrinkImageFilter

Propagation of requested region size

- Remember that requested regions propagate back up the pipeline from output to input
- Therefore, it’s likely that if we are messing with the output image size, then we will also need to alter the input requested region

Changing the input requested region

- Overload GenerateInputRequestedRegion()
- Generate the input requested region based on:
  - The output region
  - Our algorithm’s input-padding requirements/preferences
- WARNING: Never set the input requested region larger than the input’s largest possible region!
  - If input image is too small, handle the problem gracefully
  - E.g. throw an exception or degrade output at boundaries
- See:
  Modules/Filtering/ImageGrid/include/itkShrinkImageFilter
  Modules/Filtering/Smoothing/include/SmoothBlurImageFilter
An aside: base class implementations

- In general, when overloading base class functionality you should first call the base class function.
- You do this with a line like this:
  
  ```cpp
  Superclass::GenerateInputRequestedRegion();
  ```
- This ensures that the important framework stuff still happens.

Multi-threaded

- Actually relatively simple.
- Implement `ThreadedGenerateData()` instead of `GenerateData()`.
- A few things look different...

Multi-threaded: overview

- The pipeline framework “chunks” the output image into regions for each thread to process.
- Each thread gets its own region and thread ID.
- Keep in mind that this will not (and cannot) work in all cases.
- Some filters can’t be multi-threaded.

Multi-threaded: output regions

- The output target is now:
  
  ```cpp
  OutputImageRegionType & outputRegionForThread
  ```
- You iterate over this rather than over the entire output image.
- Each thread can read from the `entire input image`.
- Each thread can write to only its `specific output region`.

Multi-threaded: output allocation

- `ThreadedGenerateData()` does NOT allocate the memory for its output image!
- `AllocateOutputs()` is instead responsible for allocating output memory.
- The default `AllocateOutputs()` function:
  - Sets each output’s buffered region = requested region
  - Allocates memory for each buffered region.

Multi-threaded: order of operations

- Execution of multi-threaded filters is controlled by the inherited `GenerateData()`
- `itk::ImageSource::GenerateData()` will:
  1. Call `AllocateOutputs()`.
  2. If `BeforeThreadedGenerateData()` exists, call it.
  3. Divide the output image into chunks, one per thread.
  4. Spawn threads (usually one per CPU core).
  - Each thread executes `ThreadedGenerateData()` on its own particular output region, with its own particular thread ID.
  5. If `AfterThreadedGenerateData()` exists, call it.
ThreadID

- This deserves a special note...
- In the naïve case a thread would not know how many other threads were out there
- If a thread takes a non thread-safe action (e.g., file writing) it’s possible other threads would do the same thing

ThreadID, cont.

- This could cause major problems!
- The software guide suggests:
  1. Don’t do “unsafe” actions in threads - or -
  2. Only let the thread with ID 0 perform unsafe actions

Multiple inputs

- It’s fairly straightforward to create filter that has multiple inputs – we will use 2 inputs as an example
- For additional reference see:
  
  Modules/Filtering/ImageFilterBase/include/itkBinaryFunctorImageFilter

Step 1: Define Number of Inputs

- In the constructor, set:
  
  ```cpp
  this->SetNumberOfRequiredInputs(2);
  ```

Step 2:

- Optional: Write named functions to set inputs 1 and 2, they look something like:

  ```cpp
  SetInputImageMask( const TInputImageMask * imageMask )
  {
    this->SetInput(0, imageMask);
  }
  ```

Step 3

- Implement `GenerateData()` or `ThreadedGenerateData()`
- Caveat: you now have to deal with input regions for both inputs, or N inputs in the arbitrary case
Multiple outputs?

- Not many examples
  - `ImageSource` and `ImageToImageFilter` only recently gained full support for multiple outputs
- Previously, special calls were needed to `ProcessObject`
- The constructor of the filter must:
  - Allocate the extra output, typically by calling `New()`
  - Indicate to the pipeline the # of outputs
- What if the outputs are different types?
  - More complex
  - Example:
    - Also try searching online: itk multiple output filter

Progress reporting

- A useful tool for keeping track of what your filters are doing
- Initialize in `GenerateData` or `ThreadedGenerateData`

```cpp
ProgressReporter progress(
  this,
  threadId,
  outputRegionForThread.GetNumberOfPixels()
);`
```

Progress reporting cont.

- Pointer to the filter

```cpp
ProgressReporter progress(
  this, ThreadID, or 0 for ST
  outputRegionForThread.GetNumberOfPixels()
);`
```

Total pixels or steps (for iterative filters)

Querying progress from outside your filter

- How does your program query the total progress?
- Short answer is to use the inherited method: `ProcessObject::ReportProgress()`
  - All filters (including ones that you write) automatically have this function, since it is provided by `ProcessObject`.
- Typically you create an external observer to both query your filter’s total progress and then update your GUI
  - In particular, you write an observer that calls your filter’s `ReportProgress()` method and then calls some other “short” function to update your GUI accordingly.

Helpful ITK features to use when writing your own filter

- Points and vectors
- VNL math
- Functions
- Conditional iterators
- Other useful ITK filters
Points and Vectors

- `itk::Point` is the representation of a point in n-d space
- `itk::Vector` is the representation of a vector in n-d space
- Both of these are derived from ITK’s non-dynamic array class (meaning their length is fixed)

Interchangability

- You can convert between Points and Vectors in a logical manner:
  - Point + Vector = Point
  - Vector + Vector = Vector
  - Point + Point = Undefined
- This is pretty handy for maintaining clarity, since it distinguishes between the intent of different arrays

Things to do with Points

- Get a vector from the origin to this Point
  - `GetVectorFromOrigin()`
- Get the distance to another Point
  - `EuclideanDistanceTo()`
- Set the location of this point to the midpoint of the vector between two other points
  - `SetToMidPoint()`

Things to do with Vectors

- Get the length (norm) of the vector
  - `GetNorm()`
- Normalize the vector
  - `Normalize()`
- Scale by a scalar value
  - Use the `*` operator

Need more complicated math?

- ITK includes a copy of the VNL numerics library
- You can get `vnl_vector` objects from both Points and Vectors by calling `Get_vnl_vector()`
  - Ex: You can build a rotation matrix by knowing basis vectors

VNL

- VNL could easily occupy an entire lecture
- Extensive documentation is available at: http://vxl.sourceforge.net/
- Click on the the VXL book link and look at chapter 6
Things VNL can do

- Dot products
  ```cpp
dot_product(G1.Get_vnl_vector(),
             C12.Get_vnl_vector())
  ```
- Create a matrix
  ```cpp
vnl_matrix_fixed<
  double,
  NDimensions,
  NDimensions>
  myMatrix;
  ```

More VNL tricks

- If it were just good at simple linear algebra, it wouldn’t be very interesting
- VNL can solve generalized eigenproblems:
  ```cpp
vnl_generalized_eigensystem*
pEigenSys = new vnl_generalized_eigensystem(
  Matrix_1, Matrix_2);
  ```
Solves the generalized eigenproblem
Matrix_1 * x = Matrix_2 * x
(Matrix_2 will often be the identity matrix)

VNL take home message

- VNL can do a lot more cool stuff that you do not want to write from scratch
  - SVD
  - Quaternions
  - C++ can work like Matlab!
  - It’s worth spending the time to learn VNL
    - Especially true for C++ programmers!
    - But Python programmers may rather learn NumPy:
      http://www.scipy.org/NumPy_Tutorial

Change of topic

- Next we’ll talk about how ITK encapsulates the general idea of functions
- Generically, functions map a point in their domain to a point in their range

Functions

- ITK has a generalized function class called FunctionBase
  ```cpp
  itk::FunctionBase< TInput, TOutput >
  ```
- By itself it’s pretty uninteresting, and it’s purely virtual

What good is FunctionBase?

- It enforces an interface...
  ```cpp
  virtual OutputType Evaluate (const InputType &input) const = 0
  ```
- The evaluate call is common to all derived classes; pass it an object in the domain and you get an object in the range
Spatial functions

- Spatial functions are functions where the domain is the set of N-d points in continuous space
- The base class is `itk::SpatialFunction`
- Note that the range (TOutput) is still templated

Spatial function example

- `GaussianSpatialFunction` evaluates an N-d Gaussian
- It forms the basis for `GaussianImageSource`, which evaluates the function at all of the pixels in an image and stores the value

Interior-exterior spatial functions

- These are a further specialization of spatial functions, where the range is enforced to be of type `bool`
- Semantically, the output value is taken to mean “inside” the function if true and “outside” the function if false

IE spatial function example

- `itk::ConicShellInteriorExteriorSpatialFunction` let's you determine whether or not a point lies within the volume of a truncated cone
- `itk::SphereSpatialFunction` does the same thing for a N-d sphere (circle, sphere, hypersphere...) - note a naming inconsistency here

Image functions

- Image functions are functions where the domain is the pixels within an image
- The function evaluates based on the value of a pixel accessed by its position in:
  - Physical space (via `Evaluate`
  - Discrete data space (via `EvaluateAtIndex`
  - Continuous data space (via `EvaluateAtContinuousIndex`

Image function examples

- `itk::BinaryThresholdImageFunction` returns true if the value being accessed lies within the bounds of a lower and upper threshold
- `itk::InterpolateImageFunction` is the base class for image functions that allow you to access subpixel interpolated values
Hey - this is messy...

- You might be wondering why there are so many levels in this hierarchy
- The goal is to enforce conceptual similarity in order to better organize the toolkit
- In particular, the interior-exterior functions have a specific reason for existence

Change of topic

- You may have observed that we have (at least) two ways of determining whether or not a point/pixel is “included” in some set
  - Within the bounds of a spatial function
  - Within a threshold defined by an image function
- Useful for, e.g., connected component labeling...

Conditional iterators

- One way to think about iterators is that they return all of the objects within a certain set
- With ImageRegionIterators, the set is all pixels within a particular image region
- What about more complicated sets?

The “condition”

- The condition in a ConditionalIterator is the test that you apply to determine whether or not a pixel is included within the set of interest
- Examples:
  - Is the pixel inside a spatial function?
  - Is the pixel within a certain threshold?

Using the condition - brute force

- If the pixel passes the test, it can be accessed by the iterator
- Otherwise, it’s not part of the set
- The brute force implementation is to visit all pixels in an image, apply the test, and return the pixel if it passes

Conditional iterators - UI

- The interface to conditional iterators is consistent with the other iterators:
  - ++ means get the next pixel
  - GetIndex() returns the index of the current pixel
  - IsAtEnd() returns true if there are no more pixels to access
Conditional iterators - guts

• What’s happening “underneath” may be quite complex, in general:
  1. Start at some pixel
  2. Find the next pixel
  3. Next pixel exists? Return it, otherwise we’re finished and IsAtEnd() returns true.
  4. Go to 2.

Special case - connected regions

• For small regions within large, high-dimension images, applying this test everywhere is needlessly expensive
• Moreover, the brute-force method can’t handle region growing, where the “condition” is based on neighbor inclusion (in an iterative sense)

Flood filled iterators

• Flood filled iterators get around these limitations by performing an N-d flood fill of a connected region where all of the pixels meet the “condition”

  • FloodFilledSpatialFunctionConditionalIterator
  • FloodFilledImageFunctionConditionalIterator

How they work

• Create the iterator and specify an appropriate function
• You need a seed pixel(s) to start the flood - set these a priori or find them automatically with FindSeedPixel(s)
• Start using the iterator as you normally would

“Drawing” geometric objects

• Given an image, spatial function, and seed position:
  TITType sfi = TITType(outputImage, spatialFunc, seedPos);
  for( ; !( sfi.IsAtEnd() ); ++sfi)
  {
    sfi.Set(255);
  }
• This code sets all pixels “inside” the function to 255
• The cool part: the function can be arbitrarily complex - we don’t care!

Flood filled spatial function example

• Here we’ll look at some C++ code:
  • itkFloodFilledSpatialFunctionExample.cxx found in the InsightApplications downloadable archive of examples.
  • This code illustrates a subtlety of spatial function iterators - determining pixel inclusion by vertex/corner/center inclusion
  • Inclusion is determined by the “inclusion strategy”
Useful ITK filters

- These are filters that solve particularly common problems that arise in image processing
- You can use these filters at least 2 ways:
  - In addition to your own filters
  - Inside of your own filters
  - Don’t re-invent the wheel!
- This list is not comprehensive (obviously)
- Specific filter documentation is an EFTR

Padding an image

- Problem: you need to add extra pixels outside of an image (e.g., prior to running a filter)
- Solution: `PadImageFilter` and its derived classes
Cropping an image

- Problem: trimming image data from the outside edges of an image (the inverse of padding)
- Solution: `CropImageFilter`

Rescaling image intensity

- Problem: you need to translate between two different pixel types, or need to shrink or expand the dynamic range of a particular pixel type
- Solutions:
  - `RescaleIntensityImageFilter`
  - `IntensityWindowingImageFilter`

Computing image derivatives

- Problem: you need to compute the derivative at each pixel in an image
- Solution: `DerivativeImageFilter`, which is a wrapper for the neighborhood tools discussed in a previous lecture
- See also `LaplacianImageFilter`

Compute the mirror image

- Problem: you want to mirror an image about a particular axis or axes
- Solution: `FlipImageFilter` - you specify flipping on a per-axis basis

Rearrange the axes in an image

- Problem: the coordinate system of your image isn’t what you want; the x axis should be z, and so on
- Solution: `PermuteAxesImageFilter` - you specify which input axis maps to which output axis

Resampling an image

- Problem: you want to apply an arbitrary coordinate transformation to an image, with the output being a new image
- Solution: `ResampleImageFilter` - you control the transform and interpolation technique
  - (This is used when doing registration)
Getting a lower dimension image

Problem: you have read time-series volume data as a single 4D image, and want a 3D “slice” of this data (one frame in time), or want a 2D slice of a 3D image, etc.

Solution: ExtractImageFilter - you specify the region to extract and the “index” within the parent image of the extraction region