Creation and Demonstration of a Framework for Handling Paths in ITK

John Galeotti and George Stetten

Carnegie Mellon University
galeotti+miccai@cs.cmu.edu

Abstract. A hierarchy of path data types and basic path filters were added to ITK, providing a general framework for curves that map a scalar value to a point in n-dimensional space. The framework supports curves that are either continuous (parametric curves) or discrete (chain-codes). Example usage of the entire framework is demonstrated using a previously published 2D active contour algorithm that was converted to ITK.

Introduction

ITK was originally designed to operate on image and mesh data types. A significant number of segmentation algorithms, however, utilize active contours or other path-type data objects [1-3]. Our research required the extensive use of paths as well. Unfortunately, at the onset of our path-based research such support was completely lacking in ITK. Therefore, Galeotti created a general-purpose hierarchy of path classes in ITK, as well as filters to operate on them. He also created specialized classes to meet our specific research requirements.

The present paper reviews the process undergone to add generic support for paths to ITK and explains the general usage of the path framework within ITK. An example implementation of a previously published 2D active contour algorithm is then presented.

How We Began

Due to the general-purpose, foundational nature of this work, we consulted with the ITK developers' community to insure the general usefulness of the path hierarchy we were developing. The suggestions and criticisms that arose both at the ITK working conferences and in ongoing discussions across the ITK developers' email list were invaluable to this end.

It was decided that to be as generally useful as possible, the only assumptions that should be made by the basic path API should be that (1) a path maps a scalar input to a point along a contiguous ND curve (although the curve dimension may be fixed in some path types) and that accordingly (2) any path can be mapped to a linearly-ordered sequence of connected indices in ND image space. We choose to enforce both of these assumptions by use of an abstract path base class. We placed no other conceptual restrictions on the algorithms and data types used to represent a path. A path may be open, closed, or self-crossing.

Based on feedback obtained from other developers, it was also decided that paths should follow the ITK FunctionBase API. To enable path filters, however, the Path base class was made to descend from the ITK DataObject class rather than the FunctionBase class. Therefore, compliance with the FunctionBase API had to be manually implemented to allow path usage in template arguments requiring data-types compliant with the FunctionBase API.

Path Types

For the authors' purposes, two basic types of paths were required: chain-codes and parametric paths. These have very different algorithmic representations and associated data structures, and so a base class was created for each (both base classes descend from the abstract path base class). Figure 1 shows the final hierarchy of path classes that we implemented in ITK.

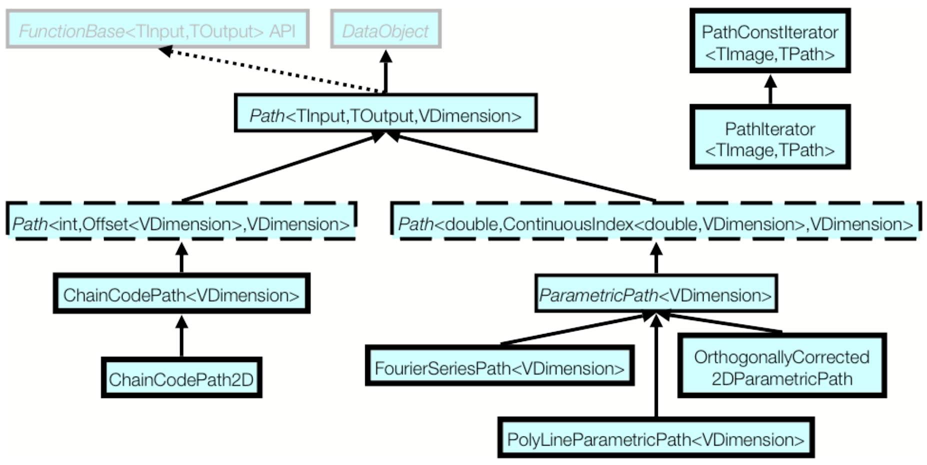

FunctionBase<TInput,TOutput> API
DataObject
PathConstIterator <TImage,TPath>
Path<TInput,TOutput,VDimension>
PathIterator <TImage,TPath>
Path<int,Offset<VDimension>,VDimension>
Path<double,ContinuousIndex<double,VDimension>,VDimension>
ChainCodePath<VDimension>
ParametricPath<VDimension>
ChainCodePath2D
FourierSeriesPath<VDimension>
OrthogonallyCorrected 2DParametricPath
PolyLineParametricPath<VDimension>

Fig. 1. Path data types added to ITK. The two classes at the top with light gray text were already a part of ITK. Classes diagrammed in boxes with thin borders are abstract base classes that were created to unify the framework and to simplify the development of additional path classes by others. Classes diagrammed in boxes with thick, bold borders are complete (instantiatable) classes that were created to be directly used by others. Boxes with dashed borders are used to show the template parameters used by the two basic path types.

Chain Codes

Chain-codes represent a path as a sequence of offsets between adjoining locations on a rectangular lattice. Whether these locations represent voxel-centers or voxel-vertices is unspecified. Both of these interpretations are equally valid, so long as one or the other interpretation is used consistently for the creation, processing, utilization, and visualization of a given chain-code. Chain-codes can, therefore, be used both to segment between voxels and to trace through the centers of voxels. In ITK, the locations visited by a chain-code were implemented using offsets, which are vectors in integer space. A chain-code with n offset-steps can be denoted as a sequence

$$\mathbf{P} = (\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3, \ldots \mathbf{u}_n) \tag{1}$$

whose elements are the individual step vectors. An arbitrary step $\mathbf{u}_i$ is a vector from a given index to one of its neighbors.

$$\mathbf{u}_i = \begin{pmatrix} \mathbf{u}_{i_1} \\ \vdots \\ \mathbf{u}_{i_N} \end{pmatrix} = \begin{pmatrix} \Delta x_1 \\ \vdots \\ \Delta x_N \end{pmatrix}, \quad \Delta x_i \in \{-1, 0, 1\} \tag{2}$$

For a 2D chain-code,

$$\mathbf{u}_i \in \left\{ \binom{0}{1}, \binom{1}{1}, \binom{1}{0}, \binom{1}{-1}, \binom{0}{-1}, \binom{-1}{-1}, \binom{-1}{0}, \binom{-1}{1} \right\} \tag{3}$$

as illustrated in Figure 2.

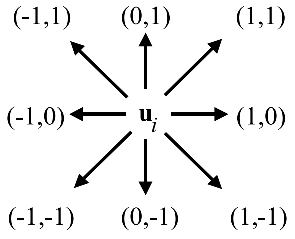


Fig. 2. Step $\mathbf{u}_i$ in a 2D chain-code.

The individual offset vectors (or steps) of a specific chain-code $\mathbf{P}$ may be represented by adding an index to $\mathbf{P}$ as $\mathbf{P}_i = \mathbf{u}_i$, where $\mathbf{u}_i$ is the ith step of $\mathbf{P}$. Note that $\mathbf{P}$ (without an index) is an entire chain-code, but $\mathbf{P}_i$ (with an index) is only one step of $\mathbf{P}$.

The matrix $\mathbf{P}$ and its constituent steps $\mathbf{P}_i$ are relative displacements from a starting index location $\mathbf{s}$, resulting in a terminal location $\mathbf{e}$.

$$\mathbf{e} = \mathbf{s} + \sum_{i=1}^{n} \mathbf{P}_i \tag{4}$$

The combination of a chain-code $\mathbf{P}$ and a starting location $\mathbf{s}$ is designated as $\mathbf{P}^{\mathrm{s}}$. $\mathbf{P}_i^{\mathrm{s}}$ designates a specific step of a specific chain-code placed at a specific starting location. Like $\mathbf{P}_i$, $\mathbf{P}_i^{\mathrm{s}}$ is the offset value of a step, but unlike $\mathbf{P}_i$, the location of $\mathbf{P}_i^{\mathrm{s}}$ can be calculated because the starting location of the step's chain-code is known. The absolute location in an image of the chain-code step $\mathbf{P}_i^{\mathrm{s}}$ can be represented as

$$\mathbf{p}_i^{\mathbf{s}} = \mathbf{s} + \sum_{j=1}^{i} \mathbf{P}_j \tag{5}$$

Note that capital **P** is used either by itself to denote an *entire* chain-code or with a subscript to denote the *offset* vector of a single step of a chain-code, while lowercase **p** is used to denote the *location* to which a *single* step of a chain-code points (and therefore requires that the starting location of that chain-code be specified). Because $\mathbf{p}_i^{\mathbf{s}}$ is defined by a sequential iterative process, chain-code location is optimized for sequential access rather than random access.

A chain-code can be *open* or *closed*. If a chain-code is closed, it begins and ends on the same pixel,

$$\mathbf{e} = \mathbf{s} \tag{6}$$

or, put another way, the sum of its steps is the zero-vector,

$$\sum_{i=1}^{n} \mathbf{P}_i = \mathbf{0} \tag{7}$$

Chain-codes may cross themselves, in which case, for some j and k, $j \neq k$, $\mathbf{p}_j^{\mathbf{s}} = \mathbf{p}_k^{\mathbf{s}}$. If a path is closed and does not cross itself, it can be constrained to proceed in a clockwise or counterclockwise direction, permitting unambiguous determination of inside vs. outside for any given step.

The ITK ChainCodePath base class is fully defined, and supports all of the functionality described by the math above. We only implemented one specialized child of the ChainCodePath class. It is a 2D chain-code that stores offsets using Freeman Codes [4], which sequentially enumerate all possible offsets and then store the offsets using their integer labels. Freeman codes require less storage capacity than normal chain codes and can simplify the rotation of individual path offsets (rotation is done by addition modulo 8). Freeman codes are also very useful for debugging, since a Freeman code can be printed to a terminal with a single character used for each step of the path.

Parametric paths

Parametric paths are represented by an algebraically defined curve parameterized over a scalar input. As opposed to chain-codes, parametric paths provide efficient random access but comparatively poor sequential access to image indices, due to the difficulty of knowing how much to increment the parametric input to reach the next sequential voxel along the path. Because of their algebraic nature, most parametric paths have a generally well-defined derivative with respect to their input. The ParametricPath abstract base class establishes a specialized API, complete with many default member-function implementations. The default member functions can greatly simplify the creation of other parametric path types by others. New parametric path types can, however, override the default implementations to take advantage of any efficiency gains that new path types may make possible. The ParametricPath base class currently has three fully defined instantiatable children. FourierSeriesPath represents a closed path by its Fourier coefficients in each dimension; it has continuous well-defined derivatives with respect to its input. PolyLineParametricPath represents a path as a series of vertices connected by line segments; it provides a simple means of creating a path that can then be converted to other path types. Finally, Orthogonally-Corrected2DParametricPath represents a path by the combination of another 2D parametric path and a list of orthogonal offsets to be evenly spaced along the other path. It simplifies the deformation of some types of 2D paths to ease path-based segmentations. The orthogonally corrected path implements the orthogonal corrections on-the-fly by linearly interpolating an offset value for a requested input value and evaluating the original path at the requested input; the interpolated offset is then added to the evaluated path position to produce the new, corrected path position, as shown in Figure 3.

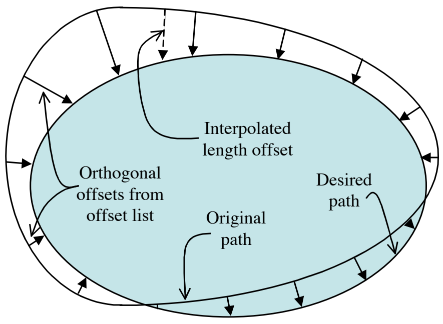


Fig. 3. Orthogonally corrected path.

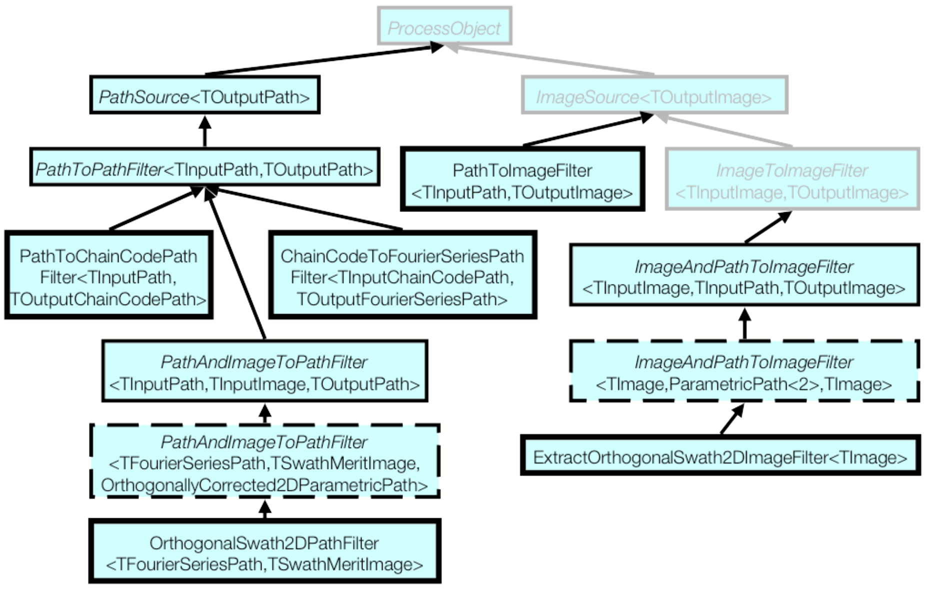
Path Iterators

While a path must store some representation of a sequence of connected indices, it was decided that a path should not store a current input value corresponding to a current position along itself. Instead, the notion of path position was delegated to a new path iterator class, capable of sequentially visiting each index along a path. The primary reason for doing so was to make possible the creation of constant (read only) path objects that can still be traced. Path iterators also allow different pieces of code to easily iterate concurrently over a single path. Operations such as testing for path equality are also simplified.

For the purpose of simplicity to the user, it was decided that a single path iterator class should be able to iterate over any type of path. To enable this, the path API requires all path classes to have an input incrementing function. The idea is that while a path does not store a current input value, a path should know how much to increment any given input value to make the path's output move to the next discrete index along the path. For chain-codes, such a function trivially returns the value of 1; for generic parametric paths, such a function must iterate over a converging region of input in a manner similar to Bresenham's algorithm [5]. For efficiency, the input incrementing function returns the index-space offset resulting from its increment of the input.

Path Filters

To integrate paths into the ITK pipeline architecture, it was necessary to expand the architecture to handle the new path data types. We added several basic path filters to ITK (as well as a complete and previously published path-based segmentation algorithm that demonstrates ITK path usage). Figure 4 shows the final hierarchy of path filters that we implemented in ITK.



ProcessObject
PathSource<TOutputPath>
ImageSource<TOutputImage>
PathToPathFilter<TInputPath,TOutputPath>
PathToImageFilter <TInputPath,TOutputImage>
ImageToImageFilter <TInputImage,TOutputImage>
PathToChainCodePath Filter<TInputPath, TOutputChainCodePath>
ChainCodeToFourierSeriesPath Filter<TInputChainCodePath, TOutputFourierSeriesPath>
ImageAndPathToImageFilter <TInputImage,TInputPath,TOutputImage>
PathAndImageToPathFilter <TInputPath,TInputImage,TOutputPath>
ImageAndPathToImageFilter <TImage,ParametricPath<2>,TImage>
PathAndImageToPathFilter <TFourierSeriesPath,TSwathMeritImage, OrthogonallyCorrected2DParametricPath>
ExtractOrthogonalSwath2DImageFilter<TImage>
OrthogonalSwath2DPathFilter <TFourierSeriesPath,TSwathMeritImage>

Fig. 4. Path filters added to ITK. The classes at the top with light gray text were already a part of ITK. Classes diagrammed in boxes with thin borders are abstract base classes that were created to unify the framework and to simplify the development of additional path-filter classes by others. Filter classes diagrammed in boxes with thick, bold borders are complete (instantiatable) filters that were created to be directly used by others. Boxes with dashed borders are used to show the template parameters used by children classes.

ITK's design places a large burden on the developer of low-level code in order to ease the use and extension of ITK by most typical users. Therefore, it is not surprising that the main difficulty in supporting path filters was the creation of the path source and path-to-path filter base classes. Other filter base classes that required implementation were the path-and-image-to-path filter and the image-and-path-to-image filter. These four base classes added the necessary code and API to ITK to support the pipelining of path objects through ITK filters, and almost any filter that utilizes paths will descend from one of these classes. One additional "foundational" filter, used to convert a path to an image, was added to enable path visualization.

In addition to the above filters, two conversion filters were written to convert between different types of paths. One converts any type of path into a chain-code. It allows the user to specify whether the resulting chain-code must be maximally connected or whether it can be minimally (vertex) connected. In 3D, the voxels of a maximally-connected chain-code will have adjoining faces (6 neighbors) while the voxels of a minimally connected chain-code may only be connected at their vertices (26 neighbors).

The other conversion filter converts any closed chain-code into a Fourier series path, making it very useful for path smoothing. Although the filter could have been implemented to directly convert from any type of closed path into a Fourier series path, a much more efficient implementation was possible by requiring that non-chain-code-type paths be converted into chain-code paths before being passed to the filter. The filter allows the specification of the number of Fourier series harmonics that should be calculated from the input chain-code; a higher number of harmonics results in a more accurate conversion while a lower number of harmonics results in more path smoothing. Regardless of the number of harmonics used, conversion of a chain code into a Fourier series path is a very useful way to acquire the advantages of parametric paths for a path originally computed as a chain code.

One more basic filter was added to ITK to extract an orthogonal "swath" image from an input path through an input image. The orthogonal-swath filter traverses along a parametric path, interpolating image pixels orthogonal to the path at regularly spaced intervals of the path's input (see Figure 5). Each interval corresponds to a single location on the x-axis of the swath image. The y-axis of the swath image corresponds to orthogonal signed distance from the path to an interpolated pixel; positive y-values correspond to points to the left when walking down the path, and the center row of the swath image (y=0) corresponds to pixels that lie directly on the path. A swath image can be very useful for examining the neighborhood around a path, as would typically be necessary when deforming a path to segment an object.

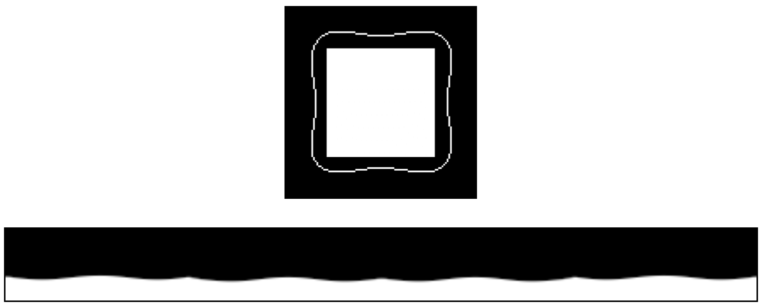


Fig. 5. A path around a white box (above) and the path's orthogonal swath (below).

Example Implementation of a 2D Active Contour Algorithm

A previously published 2D active contour algorithm [6] was added to ITK to provide example usage of the path framework. The algorithm works by finding the optimal orthogonal offsets for evenly spaced points along the initial path, with the requirement that neighboring offsets differ in value by at most one.

The algorithm is implemented in OrthogonalSwath2DPathFilter, with example usage code now available in the ITK file itkOrthogonalSwath2DPathFilterTest.cxx. To use the algorithm, first ExtractOrthogonalSwath2DImageFilter filter must used to extract a swath image from around a Fourier-smoothed initial closed path. Each pixel in the swath image is then evaluated by a merit function image filter (such as a ITK's vertical derivative filter) to produce a merit swath image, which is used as input for OrthogonalSwath2DPathFilter. Within that filter, dynamic programming [7] is used to find the optimum connected sequence of rows that the path should follow through the (merit) swath image. The resulting optimal contour is returned as an orthogonally corrected parametric path.

In a typical usage application, a user may use a pointing device to input a poly-line path that is converted to an intermediate chain-code and then converted to a Fourier series path to use as input to the swath-extraction filter. A sequence of ITK image filters may then be used to produce a merit swath image from the swath image. The merit swath image is fed to OrthogonalSwath2DPathFilter, which will produce a orthogonally-corrected parametric path. Finally, the initial and final paths can be converted into images for visualization with the path-to-image filter. The resultant output for the data shown in Figure 5 is shown in Figure 6.

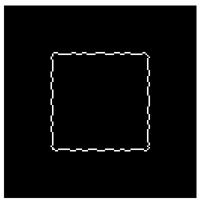


Fig. 6. Result of applying the example algorithm to the data shown in Figure 5.

Conclusion

In this paper we have presented a hierarchy of path data types and basic path filters that were added to ITK to provide a general framework for handling curves. Two types of curves were implemented. Curves that are continuous (parametric curves) provide efficient random access but comparatively poor sequential access to image indices. Discrete curves (chain-codes) are optimized for sequential access rather than random access. Filters were implemented to convert from one type of path to the other, and to integrate paths with the rest of the image-processing framework in ITK. A previously published 2D active contour algorithm was also converted to ITK, and its usage demonstrates the usage of the entire path framework.

Acknowledgements

The authors would like to thank everyone who has contributed to ITK, and especially Luis Ibáñez for his guidance in creating the path data types. This work has been supported in part by a contract with the National Library of Medicine and a NSF Graduate Student Fellowship.

References

1. Chen, C., Huang, T., and Arrot, M., 1994. Modeling, Analysis, and Visualization of Left Ventricle Shape and Motion by Hierarchical Decomposition, PAMI, 16(4), pp. 342-356.
2. Geiger, D., Gupta, A., Costa, L., and Vlontzos, J., 1995. Dynamic Programming for Detecting, Tracking, and Matching Deformable Contours, PAMI 17(3), pp. 294-302.
3. Gunn, S., Nixon, M., 1997. A Robust Snake Implementation: A Dual Active Contour, PAMI 19(1), pp. 63-68.
4. Freeman, H., 1961. On the encoding of arbitrary geometric configurations, IRE Trans. Electronic Computers EC-10, pp. 260-268.
5. Bresenham, J., 1965. Algorithm for Computer Control of a Digital Plotter, IBM Systems Journal, 4 (1), pp. 25-30.
6. Stetten, G., Drezek, R., 2001. Active Fourier Contour Applied to Real Time 3D Ultrasound of the Heart, International Journal of Image and Graphics, 1(4), pp. 647-658.
7. Montanari, U., 1971. On the optimal detection of curves in noisy pictures. Communications of the ACM, 14(5), pp. 335-345.