ContourExtractor2DImageFilter: A subpixel-precision image isocontour extraction filter.

Release 1.0

Zachary Pincus

February 1, 2006

zpincus-at-stanford.edu

Abstract

This document describes itk::ContourExtractor2DImageFilter, an implementation of the "marching squares" 2D contour-finding method using the Insight Toolkit ITK www.itk.org. This filter takes as input a 2D image (not necessarily binarized) and a contour value, and outputs zero or more itk::PolyLineParametricPath objects which represent the image contours at that value.

In ITK, two-dimensional isosurfaces can be extracted from a binarized 3D image using the filter itk::BinaryMask3DMeshSource, which is an implementation of the classic "Marching Cubes" algorithm.[1]. Currently, ITK contains no analogue for extracting 1-dimensional isovalue contours from 2D image data, which renders the itk::Path hierarchy less useful than it could otherwise be. I have produced a relatively simple implementation of a marching squares method, called itk::ContourExtractor2DImageFilter, to remedy this deficit. This filter is a subclass of itk::ImageToPathFilter, which I created to fill another minor gap in the ITK path hierarchy and to make it easier to create other such path-finding algorithms.

1 Algorithm Overview

This algorithm aims to estimate, with sub-pixel precision, isovalued contours of an input image. Thus, the algorithm must for any adjacent pair of pixels, decide whether the contour line passes between those pixels (e.g. is one pixel above the contour value and the other below), and if so, precisely where the line passes. This is accomplished by application of the marching squares method, which is well-known and covered in depth in many locations (e.g. http://www.essi.fr/~lingrand/MarchingCubes). In brief, the image is divided into a set of 2-pixel by 2-pixel squares. There are sixteen possible square configurations if all that is taken into account is whether the pixel at each vertex is above or below the desired contour value. For each configuration, the position of the contour segment that passes through that square can be uniquely determined. Thus, an implementation of this algorithm must (a) iterate over all such squares in the input image, (b) for each square, determine its "square type" and where (or whether) a contour segment should pass through the square (e.g. if the left two vertices of the square are below the contour value and the right two are above, then a contour should pass roughly through the middle of the square, from bottom to top),

(c) estimate the precise position of the endpoints of the contour segment via some sort of interpolation, and (d) add that segment to the correct contour.

2 Ambiguities

There are several ambiguous cases in the above algorithm. This section will clarify how those ambiguities are handled in itk::ContourExtractor2DImageFilter.

First, note that contours are directed: they start at some location and go to some other location. Thus, for any of the square types above, the algorithm must decide in which direction the new segment should travel. By default, this implementation draws segments such that (when viewed from tail to head) below-contour-values are on the left of the contour and above-contour-values are on the right. In other words, the image gradient is (approximately) a 90^o right rotation of the segment. This means that "islands" of high values surrounded by below-contour-values will be circled by clockwise contours. This behavior can be reversed by calling a filter method, however.

Second, note that while contours must in the general case be closed, it is not possible to properly determine how to close a contour that intersects with an image edge. Thus, these contour segments are left open. This can be easily detected by determining whether the last vertex in the contour is the same as the first — if so, then the contour is closed.

Third, the "proper" behavior when a square is encountered with two diagonally-opposed pixels above and the other two below the contour value. Either the contour surrounding the high-valued pixels can be made continuous, or the contour surrounding the low-valued pixels can be made continuous. If the contours are being used to encircle "objects", then the question is whether high-valued objects on a low-valued background should be considered to be vertex-connected (8-connected), or face-connected (4-connected). If high-valued pixels are vertex connected, then low-valued objects surrounded by a high-valued background are necessarily face-connected. By default this implementation treats low-valued pixels as vertex-connected, but this too can be toggled.

Finally, the location of a contour on a plateau is ambiguous. Imagine an image with all zero values and one pixel with value one in the center. If the zero-level contour is sought, where should it be placed? This algorithm treats a vertex of a square to be above-contour-value if it is strictly greater than the contour value, which has the effect of placing contours at the edge of a plateau nearest to the step up. Thus, seeking the zero-level contour of the image described above will provide a contour that encircles the single "one" pixel. However, seeking the one-level contour of the image will not produce any contours at all because no squares will be detected in the image with a vertex strictly greater than the contour value. This behavior cannot be toggled in the current implementation. The correct usage is simply to specify a non-degenerate contour value (e.g. 0.5 in the above case).

3 Implementation

Several implementation details are described in this section.

The fundamental difficulty in implementing marching squares is defining a data structure which will allow a newly-determined contour segment to be properly connected with the rest of that contour. In this case, I have chosen to use two hash tables: one to associate each growing contour with its starting position, and one to associate each contour with its ending position. If a segment is added that starts where an extant contour

ends, the new segment is appended to the existing segment. If a segment is added that ends where an extant contour starts, the new segment is prepended to the existing contour. If no contour start or end is found, a new contour is created and added to the tables. Finally, if an existing contour ends where the new arc starts and an existing contour starts where the new arc ends, then those two contours must be joined. There are several subtleties to this latter case.

First, if the two existing contours are actually the same contour, then that contour must be closed and removed from the "growing contours" has tables. If the two existing contours are different, then they must be merged into one. I have elected to control the merging such that the "younger" contour is merged into the older one. This ensures that contours are output in the order that they were first encountered in the image (left to right, top to bottom), which makes the output of this algorithm more predictable. This means that each contour must store its "creation number" so it is possible to decide which is oldest. Because contours are stored in a vcl_deque (to facilitate addition to the contour at either the front of the back), I have subclassed vcl_deque to also store the additional creation number parameter.

A small point to consider in implementation is how to properly march over each 2-by-2 pixel square in the image exactly once. I have done this by using a shaped neighborhood iterator which visits four pixels at once: the neighborhood "center pixel" and the rest of the square that contains that pixel as its top-left vertex. This means that the square protrudes one pixel in "front" and one pixel "below" the center of the neighborhood, so to hit each image square once and not protrude beyond the image, the filter shrinks the iteration region by one pixel at the right and bottom. This filter also supports custom-defined regions. This is a little tricky, since paths have no "RequestedRegion" member variable to store this sort of information. This, the filter itself defines a SetRequestedRegion() method. This seems better than requiring the input image to have the RequestedRegion attribute set, since that breaks ITK convention even more badly with regard to how region information is propagated.

One final implementation note: this filter is a subclass of itk::ImageToPathFilter, a new class that I am also contributing. This class subclasses from itk::PathSource, and is a close analogue of ImageToImageFilter or PathToImageFilter (etc.).

References

[1] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics*, 21(4):163–169, July 1987. (document)