Contouring with C^1 data (part 2)

Josh Levenberg

February 20, 2003

Intro

Last week I talked about how 2D contouring was like 1D contouring. Today I will go into more depth about 2D contouring and then say how 3D contouring is like 2D contouring.

To review, let $f : [0,1]^n \to \mathbf{R}$ be defined by a 'black box' that answers what the value or gradient is at any queried point (today, n will be either 2 or 3). We want to construct an $n-1$ manifold (possibly with boundary) within $[0,1]^n$ corresponding to $f^{-1}(0)$. The approach is to first construct a piecewise-cubic approximation to f and then contour each cubic.

2D

For the 2D case, this breaks down into the following steps:

1. Adaptive meshing
   - Using binary triangle trees
   - Unfortunately, hard to estimate minimum gradient of f restricted to the contour $f^{-1}(0)$ on a given triangle
2. Spline interpolation
   (a) Splines used are Bézier triangles defined in barycentric coordinates
   (b) Clough-Tocher on triangles, Sibson Split-Square on squares
   (c) Solves the degree of freedom problem so we can get C^1 continuity
   (d) Reproduces quadratics exactly; by considering adjacent triangles, can compute a cross boundary derivative that will reproduce cubics exactly (by 'minimizing the C^2 discontinuity')
   (e) Gives $O(h^3)$ (or $O(h^4)$ if you reproduce cubics) error for side length h
3. Contouring cubics using [Grandine and Kleine 1997]

(a) Define a 2-d direction as "up."

(b) Define the height function by dot product with the up vector.

(c) Consider the points where the contour is perpendicular to the up vector, these corespond to critical points of the height function on the contour.

(d) Figure out the points where the contours intersect the boundary. Add these to the list of critical points.

(e) Figure out the topology at the critical points. Between any two critical values, the contours are topologically simple.

(f) Use cubic root finding to find a bunch of points on the contours.

(g) Compute the normal and curvature at those points.

(h) Find cubic splines that match [de Boor, Höllig, and Sabin 1987] — 6th order accurate!

(i) Join the splines together according to the topology

An operation used at several points in step 3 is finding all the roots of $f_t(s) = f(s,t)$ for particular values of t. f_t is represented as a cubic Bézier spline. Its coefficients are cubic functions of t that we compute in step 3(a). Given a t, we evaluate the coefficients of f_t and then find its roots using the 1D root finder.

Step 3(c) requires solving an equation like:

$$\begin{aligned} f &= 0 \\ \frac{\partial f}{\partial s} &= 0 \end{aligned}$$

for f a cubic in s and t. I use a generalization of the 1D root finder, [Sherbrooke and Patrikalakis 1993] modified for cubics instead of bicubics. Unfortunately, this only has linear convergence, but I have a another modification that I think will give quadratic convergence.

Steps 3(f-h) are in a loop to do adaptive sampling. It measures the error at step (h) and performs further subdivision if the error is too large (this is epsilon 3 below).

Three sources of error:

1. approximation error: from using spline instead of original function:

$$\left.\begin{array}{l}\text{2 with an adaptive mesh}\\ \text{4 with a regular grid}\end{array}\right\}\text{ times as many triangles}$$
$$\text{to scale triangles by } \tfrac{1}{2}$$
$$\text{to reduce the error by a factor of }\left\{\begin{array}{l}\text{8 with standard Clough-Tocher}\\ \text{16 if you reproduce cubics}\end{array}\right.$$

2. root finding error: whenever we execute the operation "find all roots at a given height": quadratic convergence so time is $O(\log -(\log \text{error}))$. This is the same amount of time as the simultaneous solver, assuming my modifications give quadratic convergence there too.

3. spline fitting error: when we construct the splines representing the solution in step 8, error is $O(h^6)$ where h is stepsize

Algorithmic complexity is approximately:

$$O\left(\text{(length of solution)} * \frac{1}{\sqrt[3]{\text{error1}}}\right)$$

function and gradient evaluations plus:

$$O\left(\text{(length of solution)} * \frac{1}{\sqrt[3]{\text{error1}}} * \left(\log\left(-\log\frac{\text{error2}}{\sqrt[3]{\text{error1}}}\right) + \frac{\sqrt[3]{\text{error1}}}{\sqrt[6]{\text{error3}}}\right)\right)$$

other work. If you can reproduce cubics, the $\sqrt[3]{\text{error1}}$ are replaced by $\sqrt[4]{\text{error1}}$.

Total error is approx error1+error2+error3, so generally error1 >> error2, error3

Note that I haven't done careful timing to verify these running time estimates.

Note that there are many bad cases: consider $f(x,y) = y(x^2+y^2-1)$. Problems happen whenever the gradient is 0. My solution is to look at the behavior of the function $2*\epsilon$ above and below the problem area and then extrapolate to the problem point.

3D

Contouring is often used to construct surfaces from 3D data. The most famous algorithm is marching cubes. This took a regular (cube) grid of data and returned a list of triangles. Its main idea was pretty simple: create a look-up table of the 256 possible values (which fall into 14 different classes) for the signs of f at the corners of the cube (note $256 = 2^{2^n}$ where the dimension is $n = 3$).

Marching cubes, as originally published, suck. It requires a large table that is hard to generate reliably. It is impractical in higher dimensions ($n = 4$ requires a table of size 65536). It required a uniform rather than adaptive grid. It was patented. And, it did not even genrate a mesh without holes. The problem was that the table entries did not correspond to any globally defined interpolant. As a result, adjacent cubes can disagree on their common face.

Many papers have been written on how to fix this, but the easiest and simplest is to use tetrahedrons instead of cubes. Now you only need a 16 entry table (and there are only 3 different cases after symmetry). It scales to higher dimensions (table size is 2^{n+1} and the only cases are $0, 1, \ldots, n$ positive corners, and the k and $n-k$ cases are symmetric). Since there is a single linear interpolant for each tetrahedron that agrees on common faces of adjacent tetrahedrons, consistency is assured. Note that a cube can be divided into 6 tetrahdra without introducing any additional vertices.

Turns out "marching tetrahedra" can be extended to an adaptive grid using, for example, a generalization of binary triangle trees. Maubach has extended binary triangle trees to simplicies of higher dimensions. Instead of splitting diamonds consisting of two triangles sharing an edge, you need to split the wheel of tetrahedra sharing an edge.

There is also generalizations of Clough-Tocher interpolation to tetrahedra, so you can do piecewise-cubic approximation on this adaptive mesh. It is most natural to represent this approximation with Bézier tetrahedra.

To contour this approximation, the algorithm:

1. Picks an 'up' direction and height function $h(x, y, z)$ that simply evaluates the dot product with the up direction

2. Finds all critical points (that is: where f is zero and the gradient ∇f is parallel to the up vector)

3. If ∇f is zero anywhere f is zero, exit with an error

4. Finds the minimums and maximums of the contour (with respect to h) restricted to the boundary of the domain. These are added to the list of critical points.

5. Sorts the critical points into increasing height.

6. For each critical point (in order):

   (a) Find the principle axes of the quadratic approximation to the contour. If the second derivative of f is zero, we probably need to pick a new up vector and start over.

   (b) Computes the *index* of the critical point

   (c) If the index is zero, a neighborhood of the critical point is added to the solution-so-far.

   (d) Otherwise some curves are found connecting this critical point to the solution-so-far

7. The resulting net of curves captures the topology of the contour, but may need to be refined in order to reduce the error

8. Finally, triangular cubic Bézier patches are found that have boundaries in the net of curves and meet with C^1 continuity. [J. Peters 1990]

This is best explained with the example of a torus and a lot of pictures. The hardest part is finding curves contained within the contour in steps 6c and 7. Actually, the last step is pretty hard, but someone already wrote a paper on how to do it.