Chapter 5

Algorithms

This manual is available on-line at the URL

`http://www.geom.umn.edu/locate/pisces/`

The on-line version may be more current, and may contain features not contained in the printed version. For example an the on-line version contains an interactive interface to Pisces via the World-Wide Web.

An Adaptive Grid Algorithm (AGrid)

Overview

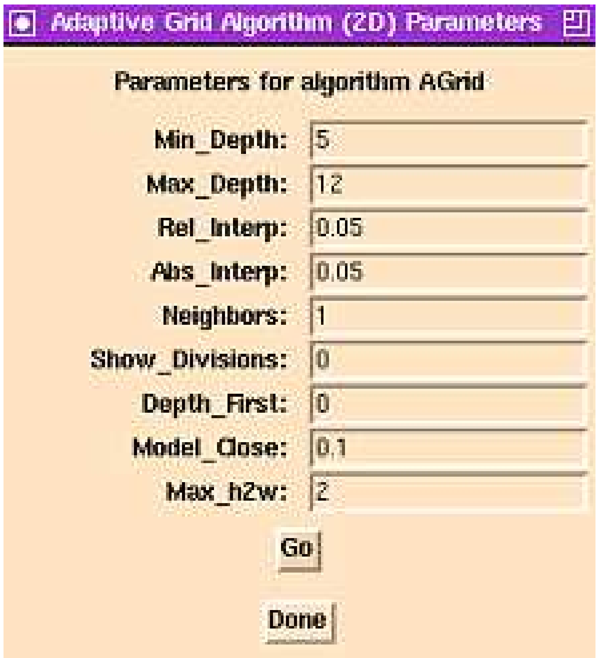
This algorithm locates a level set of an arbitrary function of two variables. It should locate all components of the level set, even fairly small ones, and treat singularities well (though not perfectly).

5.0.3 How Adaptive Grid Works

A standard method of locating the zero set of a function is to break the domain into a grid (like graph paper), and then for each square in the grid, look along the edges of the square for zeros of the function. This is usually done by evaluating the function at the corners of the square and looking for edges that have a sign change; provided the function is continuous, there will be a zero somewhere along such an edge. The position of the zero is approximated using the values of the function at the corners, usually via linear interpolation. Once the positions of the zeros along the edges of the squares are located, these zeros are joined by line segments through the interior of the square. Taken over all the squares in the grid, these line segments represent an approximation to the actual zero set of the function. To get a better approximation, one uses a finer grid.

The main drawback of this approach is that in order to obtain improved resolution, the grid must be made finer over the entire domain, including part of the domain that are far from the zero set, so a lot of work will be done even where it is unnecessary. The main idea of AGrid is to begin with a fairly course initial grid, and then refine it only where it is necessary. This allows for higher resolution in the neighborhood of singularities, for example, while still allowing for lower resolution in places where the level curve can be represented adequately by longer linear segments.

An algorithm of this sort requires a subdivision criterion that determines when a portion of the grid should be divided, and when subdivision is no longer necessary. AGrid employs several different tests in order to decide if a subdivision is needed, and these are divided into three groups: tests based on the behavior of the function along the edges of the grid; tests based on the line segments used to approximate the zero set within a region of the grid; and tests based on the subdivisions occurring within neighboring regions of the grid. If any of the tests along any edge of the region requires subdivision, the region is subdivided. Likewise, if the approximation to the zero set requires more subdivision, or if the neighboring regions suggest that subdivision is needed, then the region will be subdivided, otherwise it is no longer divided and the approximations to the zero set that it contains are added into the final representation of the zero set. The tests themselves are controlled by the parameters described below.



The panel that controls the adaptive grid algorithm.

5.0.4 Controlling the algorithm from Pisces

Min_Depth

The AGrid algorithm uses a triangular mesh which is obtained by taking a rectangular domain and dividing it into two triangles. Each of these triangles is then subdivided, and the resulting triangles subdivided again, and so on for a total of Min_Depth times (giving 2*2^`Min_Depth` initial triangles). A setting of 5 typically gives a good starting configuration.

Max_Depth

This value is the maximum number of times that a triangle will be divided. That is, if a triangle is the result of subdividing one of the initial triangles `Max_Depth` times, then it will not be subdivided further even if the other tests say it should. Setting `Max_Depth` equal to `Min_Depth` makes AGrid work essentially like the traditional (non-adaptive) algorithm. A value of around 12 to 15 is usually sufficient.

Rel_Interp and **Abs_Interp**

These parameters control how accurate an interpolated zero must be in order to be considered good enough to use. When the endpoints of an edge have different signs, the location of the zero between them is approximated by linear interpolation between the values of the function at the endpoints of the edge. The value of the function at this point is computed and checked to see how close it is to zero; if it is greater than `Abs_Interp` in absolute value, then it is not considered to be good enough, and the triangle will be subdivided. Otherwise, the value is compared to the average of the absolute values of the values of the function at the endpoints; if the proposed zero is less (in absolute value) than `Rel_Interp` times the average of the endpoints, then the zero is accepted, otherwise the triangle will be subdivided. A zero must pass both tests in order to be considered accurate. The reason for `Rel_Interp` is that on an edge where the value of the function is uniformly small, more accuracy is needed when determining the zero, so we test for zero relative to the values of the function nearby, and we always require the zero to be within `Abs_Interp`, regardless.

If the zeros along the edges of a triangle all pass these tests, then an approximation to the zero set within the triangle is given by the line segment joining the zeros on the boundary edges. Before this approximation is accepted, it is also tested for accuracy; its midpoint is determined, and the function value at this point is calculated, and it is tested against `Abs_Interp`, and `Rel_Interp` (here the average of the absolute values of the function at each of the corners of the triangles is used for comparison). If the value passes both tests, then the approximated zero set is accepted, otherwise the triangle is subdivided.

Reasonable values for `Abs_Interp` and `Rel_Interp` depend on the function you are actually using, but 0.01 for `Abs_Interp` and 0.05 for `Rel_Interp` make good starting places. Increase these values if subdivisions are occurring where they are not needed, or decrease them if not enough division occurs near the zero set.

Neighbors

This variable takes a value of 0 or 1, with 1 meaning that subdivisions in one triangle may cause subdivision in neighboring triangles, while 0 means that subdivisions do not propagate to neighboring regions. The idea is that if great detail is needed in one area, then we need to treat the nearby area with more care. Setting `Neighbors` to 1 can cause the results to be more accurate, and allows the other parameters to be less strict, but at the cost of causing more work to be done when it might not really be necessary. The actual test performed is: if a region is being subdivided and its neighbor is more than twice the size of the resulting triangles, then the neighbor will be subdivided as well. Such subdivisions may propagate outward to neighbors that are farther away, as well.

Show_Divisions

This variable takes a value of 0 or 1, with 1 meaning that AGrid will output the boundaries of the triangles that it has used in the grid, and 0 meaning that the triangles are not shown. This is sometimes useful as a means of seeing where AGrid is spending most of its energy, particularly as you try to adjust the other parameters.

Depth_First

This variable takes a value of 0 or 1, with 1 meaning that each triangle is divided to as small as it needs to be before the next triangle is processed, while 0 means that all the triangles of a given size are handled before smaller triangles are processed. Ideally, the results should be the same, but if `Neighbors` is set to 1, the amount of processing time may be affected.

Model_Close and **Max_h2w**

For an edge where the sign of the function differs on the endpoints, we know there is a zero somewhere on the edge; but if the signs are the same, there still might be a zero on the edge, so we need a criterion for when to subdivide such edges. First, we compute a point on the edge that is likely to be where the function is closed to zero along the edge (this calculation is based on the values of the function at the endpoints). The function is evaluated at this point; if its sign is different from that at the endpoints, then the triangle will be subdivided. Otherwise, the value is divided by the length of the edge; if the result is greater than `Max_h2w`, then the function is considered to be too far from zero over that edge, and we assume there is no zero along that edge, so no subdivision is required. Otherwise, we compare the value of the function at that point to the value of the linear function connecting the values at the endpoints of the edge; if these differ by more than `Model_Close` times the width of the edge, then the triangle will be subdivided.

The idea here is that we are using a linear approximation to model the actual function, and if it is close enough to the value of the function, then the fact that the linear model doesn't have a zero indicates that the actual function doesn't have a zero on the edge. (The actual formula used is a bit more complicated than the one given above, and is explained in detail in the source code.)

Reasonable values for `Max_h2w` and `Model_Close` depend on the actual function you are using, but values of 2 for `Max_h2w` and 0.1 for `Model_Close` are good starting points. Decreasing either of these parameters will cause more subdivisions to occur. Note that `Abs_Interp` and `Rel_Interp` only come into play on edges where the function values at the endpoints have opposite signs, and `Model_Close` and `Max_h2w` only come into play when they have the same sign.

5.0.5 Known Bugs

None. Please send bug reports to `software@geom.umn.edu`.

5.0.6 Acknowledgements

The algorithm was implemented by Daniel B. Krech, who also developed the portion of the algorithm that subdivides a triangle based on the size of its neighbors.

This code is based on an algorithm developed by Davide Cervone at the Geometry Center, though this implementation does not include all the features of the complete algorithm. The full algorithm uses tangent information to improve the results, and to locate and identify singularities and non-transverse zeros.

5.1 A Box Algorithm (Box)

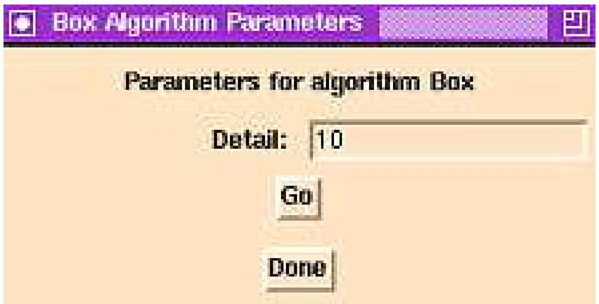
5.1.1 Overview

This algorithm locates a level set of an arbitrary function from *R^n -> R*, although only the dimensions *n=2* and *n=3* are implemented. It should locate all components of the level set that are sufficiently large, but will typically misrepresent the topology of a level set in the vicinity of a singularity.

5.1.2 How the Box Algorithm Works

For functions of two variables, the Box Algorithm locates the zero set of a function by breaking the domain into a fixed set of rectangular cells. For each square in the grid, look along the edges of the square for zeros of the function. This is done by evaluating the function at the corners of the square and looking for edges that have a sign change; provided the function is continuous, there will be a zero somewhere along such an edge. The position of the zero is approximated using the values of the function at the corners via linear interpolation. Once the positions of the zeros along the edges of the squares are located, these zeros are joined by line segments through the interior of the square. Taken over all the squares in the grid, these line segments represent an approximation to the actual zero set of the function. To get a better approximation, one uses a finer grid. See the explanation of the so-called simple algorithm to get a more complete description of the geometry behind the two-variable case.

For functions of three variables, the process is similar, except that cells are now cubes. We linearly approximate the zero set by finding points on each cell edge that are approximately zero, and then we connect these points with a polygon. The surface is formed as the union of these polygons.



The panel that controls the box algorithm.

5.1.3 Controlling the algorithm from Pisces

Detail

The box algorithm uses a uniform subdivision of the bounding box into cells. The computational domain will be divided into `Detail`$\hat{}n$ cells, n=*2,3*.

5.1.4 Known Bugs

None. Please send bug reports to `software@geom.umn.edu`.

5.1.5 Acknowledgements

The algorithm was implemented by Daniel B. Krech, who was supervised by Davide Cervone.

5.2 Geisow's Algorithm (Geisow)

5.2.1 Overview

This algorithm (also known as the "2D Bernstein Method") traces the level set of a POLYNOMIAL function of two variables. That is, given a polynomial from *R^2->R*, this function computes the algebraic variety associated to that polynomial. This algorithm should NOT be used on functions that are not polynomials!

5.2.2 How Geisow's Algorithm Works

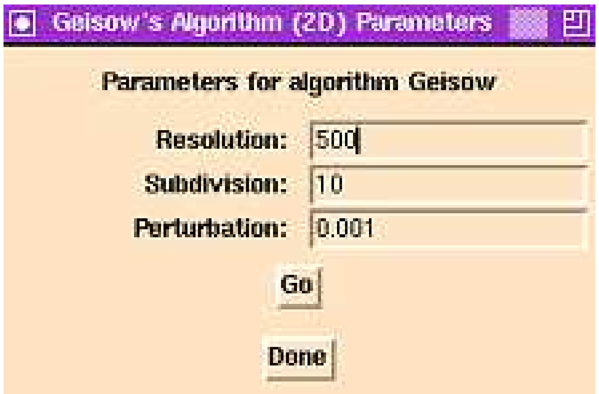
Geisow's algorithm traces the zero set of an arbitrary polynomial curve in a specified region of *x,y* space. It represents the polynomial in the Bernstein basis

{ choose*(N,i)(1-x)^{N-i}x^i* },
because the coefficients of the polynomial in that basis have the property that if ALL Bernstein coefficients have the same sign (pos or neg), then the polynomial does not have any zeros on the interval *[0,1]*. (The converse does not hold: there are Bernstein polynomials with coefficients of different signs that have no zeros on *[0,1]*.) **The algorithm is therefore:**

```
Compute coefficients of polynomial in Bernstein form.
Check coeffs of 2D Bernstein polynomial
if they all have the same sign,
   poly has no zeros ==> DONE
else
{
   look at first partial derivatives (as Bernstein polys)
   if they have coeffs of same sign
      poly is monotone, therefore find zero and draw soln.
   else
   {
       if we are in a small region of space,
           draw linear approx to poly
       else
           chop 2D region into 4 subregions
           recursively generate curve for those regions
   }
}
```

End result: We either

1. draw linear approximation to level curve
2. give up because we've reached resolution limit
3. throw the rectangle out because there is no root in the rectangle



The panel that controls Geisow's algorithm.

5.2.3 Controlling the Algorithm From Pisces

The Pisces panel to Geisow's algorithm has 3 parameters:

Resolution

This parameter determines the size of a "small" region of space. Specifically, let L = minimum side length of initial bounding box. We will subdivide the initial bounding box into subregions until we have a subregion with side length less than L /`Resolution`. We will then no longer divide that region.
EXAMPLE: If you begin with a bounding box of unit length and if `Resolution` = 100, then we will stop subdividing space when we have a region with side length >1/100.
DEFAULT VALUE: 500

Subdivision

This parameter is similar to the one above, except that if we let L as above, then if we have a region of space with side length less than L / `Subdivision`, then we assume that we can approximate the solution curve linearly on this domain. If `Subdivision` is equal to `Resolution`, then we will not fill in any of the solution until we have subdivided as far

as possible. Usually, however, we want to put in pieces of the solution curve (when applicable) before subdividing to the limit, so we typically set `Subdivision` to be less than `Resolution`.
DEFAULT VALUE: 10

Perturbation

Sometimes symmetries in the polynomial being studied give us a misleading view of how the algorithm behaves. For example, if the polynomial is singular at the origin, then the algorithm will "locate" the singularity by virtue of the fact that the singularity was on one of the grid points for the algorithm. In order to restore genericity, we perturb the initial grid randomly. The size of this perturbation will be smaller than the value of `Perturbation`.
EXAMPLE: A user has the bounding box set for *[0,1]x[0,1]* with `Perturbation`=0.1. Then the algorithm will compute a new bounding box with xmin chosen randomly in the interval *[-0.1, 0.1]*, with xmax chosen randomly in the interval *[0.9, 1.1]*, and similarly for ymin and ymax.

5.2.4 Known Bugs

Coefficients for the polynomial are determined numerically, so any coefficient that has a magnitude less than about 1.0e-8 will get truncated to 0.

The routine does not verify that the current function is a polynomial. This is because a function like *x + y^2 + A*sin(x)* may or may not be a polynomial, depending on whether $A=0$, and whether x is a variable or a parameter.

Please send bug reports to `software@geom.umn.edu`.

5.2.5 Acknowledgements

The algorithm was implemented by Frederick J. Wicklin.

The code is based on an algorithm developed by A. Geisow in his 1982 PhD thesis, "Surface Interrogation," University of East Anglia, (UK)

Dr. Ralph Martin (Dept of Computing Maths, University of Wales College of Cardiff, Senghennydd Rd, Cardiff, CF2 4AG, United Kingdom) kindly gave the Geometry Center code that implemented this algorithm. The code was originally written for Sun PHIGS. Many thanks to Drs. Geisow and Martin for agreeing to allow their code to be incorporated into the Geometry Center code.

5.3 A Simple Algorithm (Simple)

5.3.1 Overview

This algorithm traces the level set of a function of two variables. This algorithm is not intended to be used when an accurate approximation to a level curve is required. Rather, this algorithm is a simple example of how to write an algorithm in Pisces.

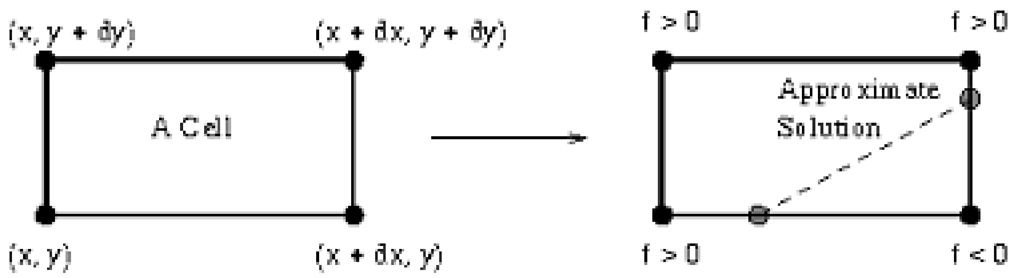
5.3.2 How the Simple Algorithm Works

The idea is, well, simple. The user specifies a bounding box and two integers. The two integers determine the number of sub-boxes into which the bounding box is subdivided. We will call the sub-boxes "cells."

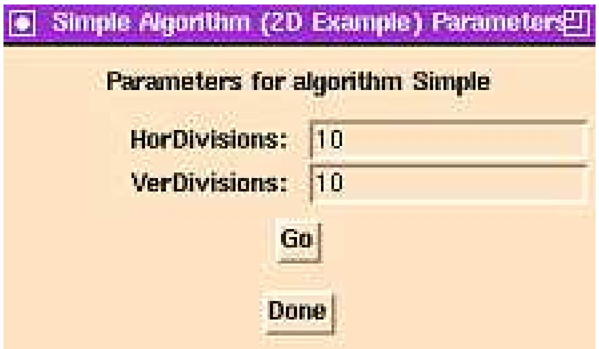
The algorithm subdivides the bounding box into equal sized cells, then evaluates the function at the corner of each cell.

For each cell, if all of the function evaluations have the same sign (all positive or all negative), then we assume that there is no solution to *f=0* within the cell.

If at least two corners of a cell have different signs, then evaluate the function along each edge and solve for points along the edge where *f=0*. In this implementation, we only use a linear approximation to the solution. The collection of points where *f=0* are then connected to each other (in the order they were generated) with line segments.



Schematic diagram of the simple algorithm.



The panel that controls the "simple algorithm."

5.3.3 Controlling the algorithm from Pisces

HorDivisions

This parameter determines the number of intervals into which the horizontal side of the bounding box is divided.
EXAMPLE: If the bounding box is *[0,1]x[-1,1]* and `HorDivisions` equals 4, then we will subdivide the interval [0,1] into four segments: [0,0.25], [0.25,0.5], [0.5,0.75], and [0.75,1]. The endpoints of these segments will be used as gridpoints for evaluating the model.

VerDivisions

This parameter determines the number of intervals into which the vertical side of the bounding box is divided.

Known Bugs

None, but the algorithm is extremely wasteful in the allocation of memory and in the evaluation of the model. This was done intentionally in order to make the algorithm as easy to understand as possible. Please send bug reports to `software@geom.umn.edu`.

Algorithm Implemented by

Frederick J. Wicklin

5.4 Predictor-Corrector Algorithm for Curves

5.4.1 Overview

This algorithm traces the level set of a an arbitrary smooth function f: R^n -> R^{n-1}. When n=2, it pays close attention to the existence and local resolution of singularities (points where the level curve does not have a well-defined tangent line).

5.4.2 How Predictor-Corrector works

The basic idea is to

1. find one point on the curve;
2. compute a tangent vector to the curve at that point;
3. step out a small amount along the tangent vector;

4. relocate the curve at a new point

5. go to step 2

Typically, steps 1 and 4 are accomplished by using Newton's method to iterate a point near the curve onto the curve. Step 2 is accomplished by solving the linear system $Df(x)\ T = 0$ where $DF(x)$ is the $n\ x\ (n\text{-}1)$ Jacobian matrix of f evaluated at a point x on the curve, and where T is the tangent vector to $f=0$ at the point x.

This particular implementation attempts to intelligently implement this basic strategy by

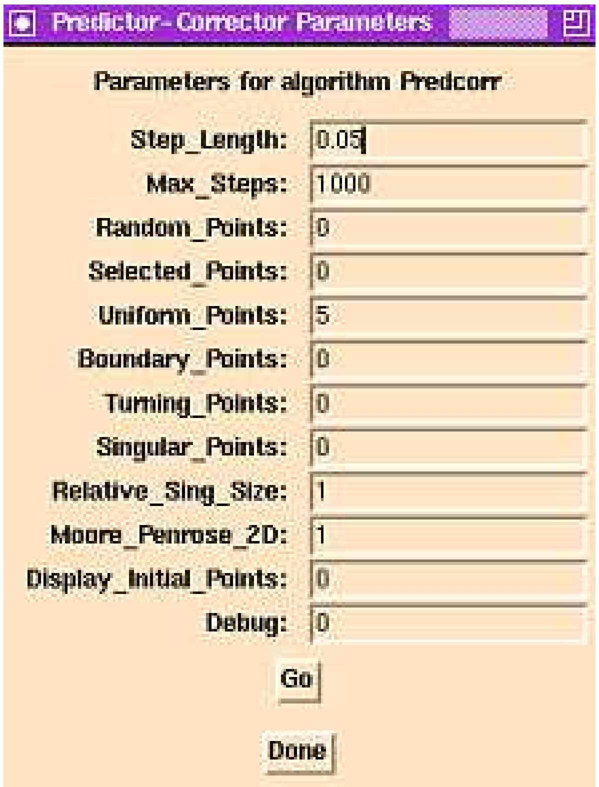
1. Finding special points on the curve to use as initial starting points for the tracing. These "special points" include

   - turning points (where one of the derivatives to the curve vanishes)
   - boundary points (where $f=0$ intersects a bounding box)
   - singular points (for planar curves, where $f=0$ and $grad(f)=(0,0)$)

   We allow the user to select which types of initial points to search for and use. Choices include the special points described above as well as points obtained by iterating onto the curve seeds corresponding to points on a uniform grid, to random points, or to points selected by the user.

2. Detecting and locally resolving planar singularities. By this we mean that we excise a neighborhood of the singularity and replace that region by a local representation of $f=0$ near the singularity. This uses ideas from singularity theory, differential geometry, and algebraic geometry.

3. As we trace the curve, we keep track of where we are relative to any "special points" we may have computed. If we enter a neighborhood of a special point, then we intelligently connect $f=0$ to the point.



The panel that controls the predictor-corrector algorithm.

5.4.3 Controlling the Algorithm from Pisces

The Pisces panel to the predictor corrector algorithm has the following parameters which may be set from the algorithm window:

Step_Length

This field should contain a positive numerical value. It indicates the step length for the predictor part of the algorithm. That is, we take a step of length `Step_Length` in the direction tangent to the curve, before using Newton's method to correct back onto the curve. Due to this correction, the actual length of the line segments outputted by the algorithm will vary somewhat at different points along the curve. If `Optimize_2D` is off

or if the model domain is of dimension greater than 2, each segment will have length AT LEAST `Step_Length`.
Default value: 0.05

Max_Steps

This field should contain a positive numerical value. It indicates the maximum number of steps that the algorithm will attempt in each direction from each initial condition.
Default value: 1000

The following parameters allow the user to choose among various methods of selecting initial points. At least one of these fields must be nonzero in order to obtain output. Several or all of them may be selected at once.

Random_Points

This field should contain a nonnegative integer value. This value specifies the number of random points (all within the domain boundaries of the model) that the algorithm will generate. Each of these points is then used as a seed for Newton's method, which attempts to iterate these seeds onto the curve. In a model with domain dimension n, these iterations are confined to an n-1 dimensional subspace of the domain in order to obtain a fully determined system. If the random point is p, this hyperplane is chosen as the n-1 dimensional subspace orthogonal to the tangent vector at p to the level curve f(x) = f(p). Newton's method works somewhat differently if n = 2 and `Optimize_2D` is set to 1. See documentation for `Optimize_2D` below. *Default value:* 50

Selected_Points

This field takes a boolean value 0 (false) or 1 (true). If the value is nonzero, the user will be prompted to enter points, coordinate by coordinate, at the Pisces command line. The user is allowed to enter as many points as desired. The algorithm attempts to iterate the points onto the curve using Newton's method.
Default value: 0

Uniform_Points

This field should contain a nonnegative integer value. A value of 0 indicates that uniform grid points should not be used as seeds. Given a positive value m for `Uniform_Points` in an n-dimensional model, the algorithm constructs a uniform grid with m^n points, and attempts to

iterate each point onto the curve using Newton's method. Specifically, the algorithm divides the domain space into m^n uniform n-cubes, and places a grid point at the center of each cube.
Default value: 4

Boundary_Points

This field should contain a nonnegative integer value. If the value is positive, the algorithm sets up a grid along each boundary face of the domain space, and attempts to iterate each point onto the curve under the constraint that all points must remain along that boundary. Given a value m for `Boundary_Points`, the grid used is uniform as described above, with m^(n-1) points per boundary. All initial points found by this method will be along the boundaries.
Default value: 3

Turning_Points

This field should contain a nonnegative integer value. If the value is positive, the algorithm sets up a uniform grid (where the Turning_Points parameter specifies the fineness of the grid, as described in #4 above), and attempts to iterate each point onto a turning point of the curve with respect to each axis. A turning point is defined as a point where the tangent to the curve either is not uniquely defined (a singularity) or is perpendicular to one of the axes. For each grid point, a separate series of iterations is required for each axis to which a turning point might correspond.
Default value: 0

WARNING: When using models with a large number (n) of dimensions, be careful not to use large values (m) for `Uniform_Points`, `Boundary_Points`, and `Turning_Points`. The values in these fields do not indicate the total number of seeds to use; they represent the fineness of the grid. The functions which find uniform and turning points use m^n seed points, while the one which calculates boundary points uses m^(n-1) seed points per boundary. Large values of `Turning_Points` (say, above 10 for three or more dimensions) may cause the algorithm to be especially slow, since turning point iterations are relatively complex and since a separate set of these iterations must be executed for each coordinate axis for each point.

Singular_Points

This field should contain a nonnegative integer value. If the value is positive, the algorithm sets up a uniform grid (where the `Singular_Points` parameter specifies the fineness of the grid, as described above under uniform points), and attempts to iterate each point onto a singularity of the function. In general, we define a singularity as a point where the Jacobian of f is rank deficient. Singularity detection and handling is currently implemented only in 2 dimensions, so setting this parameter to a nonzero value with a higher dimensional model will cause an error message. In two dimensions, a singularity is simply a point where $f=0$ and both partial derivatives of f are 0. Singularity searches are based upon Newton's method, but are more complicated than those described above – both because the system is overdetermined and because the search conditions degenerate as the algorithm approaches a higher-order singularity, causing the search to become unstable.

We deal with the latter problem by using an algorithm based on an extension of ideas presented in a paper by W. Govaerts, which dynamically adds search conditions as it approaches a higher-order singularity. When a singularity is found by this routine, a region surrounding it is "desingularized." That is, a local approximation to the curve in this neighborhood is computed and drawn. The singularity point itself is not used as an initial point, but near-singularity points found at the boundary of the desingularization region are used. Singularity detection is generally slow compared to other methods of finding initial points. The above warning applies, even though $n=2$.

Relative_Sing_Size

This field takes a real positive value. It is used to specify the size of the desingularization region: the singular radius is computed as the product of the `Relative_Sing_Size` times `Step_Length`. For a higher order singularity, the desingularization region is a square centered at the singularity with side length of twice the singular radius. For a node, the singularity-handling routines trace a distance equal to the singular radius in each of the four tangent directions.
Default value: 1

Moore_Penrose_2D

This field takes a boolean value 0 (false) or 1 (true). The value of this field has no effect unless the model is planar. If this option is turned on and the model is planar, the predictor-corrector algorithm uses a specialized routine for implementing Newton's method, both for determining initial points from random, uniform, or user-selected seeds, and for correcting back onto the curve after taking a predictor step (the specialized functions are not used while searching for boundary points, turning points, or singularities). These functions are referred to as optimized because all formulas are hard-coded for the special case of two dimensions (in which many of the formulas become degenerate). LU decomposition and backsolve routines are not used.

The 2D search methods also differ from the general ones (when used for a planar model) in that they use a different variant of Newton's method. These functions are called in cases when Newton's method is in fact being used to solve a system which is underdetermined by one equation. Classically, Newton's method requires a fully determined system (i.e., it involves the inversion of a square matrix). The general procedure solves this problem by introducing the additional condition that the iterations occur within a hyperplane perpendicular to a previously determined vector (the vector used is typically a tangent vector; when stepping along the curve, it is the vector indicating the direction in which the step was taken). The "optimized" functions do not add this extra condition; they implement Newton's method on an underdetermined system, using the method of the Moore-Penrose inverse described in Allgower and Georg. For the planar case, this simply involves recomputing the gradient at each iteration, and taking a Newton's step in that direction. Our experimentation shows that (after we fixed a seemingly endless string of bugs in these optimized functions) the optimized functions tend to give better results than their general counterparts, and to execute slightly faster. However, the behavior of the optimized functions is sometimes less predictable, and we have found a few cases where the general functions gave better results.
Default value: 1

Display_Initial_Points

This field should contain either 0 (false) or 1 (true). If it is 0, only the actual curve computed by the algorithm will be displayed. When this field is set to 1, the initial points that were used to trace the curve will also be displayed as red dots. In addition, the algorithm will output its bounding box (separate from Geomview's bounding box) in black when this option is set.
Default Value: 0

Debug

This field should contain either 0 (false) or 1 (true). When `Debug` is 1, a very large quantity of unimportant information will be displayed on the standard output. This output describes whether or not Newton's method successfully converged to each initial point, and how the line segments were traced between the initial points. This option is not intended for general use, although some people find the debugging output strangely interesting.

There are also some graphical effects when `Debug` is set to 1. After the algorithm computes all of its initial points, it selects an appropriate subset of those initial points to be used in the tracing of the curve. When both Display_Initial_Points and `Debug` are set to 1, the initial points that were NOT selected to be used to trace the curve will also be displayed. Points that were too close to another point will be displayed as blue dots, and points that were outside of the bounding box will be displayed in green.

If you want your X server to spend minutes on end outputting text debugging information, then try setting `Debug` to 2 (which will output everything you get with `Debug` set to 1, and more).
Default Value: 0

5.4.4 Known Bugs

We are not aware of any programming bugs in the new Predictor-Corrector. However, the singularity detection and handling routines are not yet in finished form. They generally work, but sometimes fail to accurately locate or handle singularities. These routines will be improved in the future.

You may also encounter one of the known output bugs. None of these bugs are intrinsic to the Predictor-Corrector algorithm; they are caused by the Pisces graphics code or by Geomview. Most of the glaring output bugs have been fixed, but a few remain. In particular, if you use Geomview as your output device, you will occasionally be informed that Geomview can't seek back far enough on its input pipe (and no image will be drawn).

Future improvements to predictor-corrector will include improved singularity detection and improved singularity handling (tracing within the desingularization region).

Please send bug reports to `software@geom.umn.edu`.

5.4.5 Acknowledgements

This algorithm was implemented as part of the Geometry Center's 1995 Summer Institute for Undergraduates. The Pisces team worked under the direction of Frederick J. Wicklin and with Center apprentice Nicolas Vera on this algorithm and on the related singularity algorithms. The linear algebra routines in used by this algorithm were written by Vera and Wicklin.

The summer students were

- Erik Streed, streed@geom.umn.edu
- Ron Dror, dror@geom.umn.edu
- John Golden, jgolden@geom.umn.edu
- Brian Johnson, bwj@geom.umn.edu

For questions about this implementation, contact Frederick J. Wicklin A good starting reference on the predictor-corrector algorithm is
Numerical Continuation Methods, An Introduction,
by Eugene L. Allgower and Kurt Georg,
published by Springer-Verlag in 1990
for the Springer Series in Computational Mathematics, Number 13

5.5 Recursive Surface Algorithm (RSurf)

5.5.1 Overview

This algorithm approximates the zero level set of a function of three variables. That is, given a function from *R ˆ3->R*, this algorithm computes the level set of zero associated to that function.

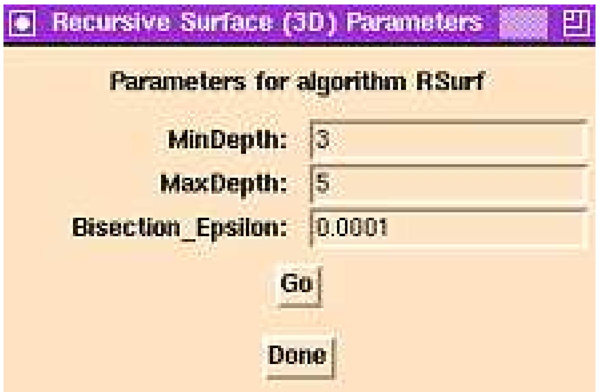
5.5.2 How Recursive Surface Works

The basic step of RSurf is to subdivide a cube into eight smaller cubes. The initial cube is the bounding domain. The algorithm starts by replacing the initial cube and all that result until all the cubes have some minimum required depth. A cube's depth is the depth it has in the octree that results from the cube replacement. The algorithm continues by subdividing when a cube has not reached the maximum allowed depth and, the eight numbers that result when you evaluated the function at the vertices of the cube do not have the same sign.

Once a cube has reached its maximum depth, a polygonal approximation to the surface is made by:

1. Start at the first edge with a sign crossing.
2. Look up which edges can be next based on what the current edge is and its orientation.
3. Decide which of these three edges have the sign crossing.
4. Continue until back to the first edge.
5. The polygon is then made up of the traversed edges crossings.

The algorithm is currently implemented recursively.



The panel that controls the recursive surface algorithm.

5.5.3 Controlling the algorithm from Pisces

The Pisces panel to the RSurf algorithm has 3 parameters:

MinDepth

All cubes are divided until they have at least minimum depth.
Example: If `MinDepth` is 3, then the original cube (the bounding domain) is divided into 8 cubes, each of the 8 into 8 cubes, and each of the resulting 64 into 8 cubes resulting in 512 cubes filling the original bounding domain.
Default Value: 4

MaxDepth

No cubes are allowed to have depth greater than the maximum depth.
Example: If the `MaxDepth` is 5, then if a cube has depth 5 it will not be subdivided any further, but rather a polygonal approximation to the surface will result instead.
Default Value: 6

Bisection_Epsilon

When bisecting the edge of a cube to find out where the surface crosses the edge, the bisection process stops on an interval of `Bisection_Epsilon`.

Default Value: 0.0001

5.5.4 Known Bugs

Sometimes a memory allocation error in RSurf will cause Pisces to crash. If you can systematically reproduce this crash, we would like that information so that we can fix the problem. Please send bug reports to `software@geom.umn.edu`.

5.5.5 Acknowledgements

The algorithm was implemented by Daniel B. Krech. The algorithm was inspired after working on implementing a brute force method for finding curves in the plane. The method involved dividing each "scan line" up into n segments, doing bisection on any segment where the functions sign changed and drawing the pixel that results from the bisection. This "scanning algorithm" was suggested by Auden Holme.