Bisection in 1D, 2D, and 3D

David Eberly
Geometric Tools, LLC
http://www.geometrictools.com/

Created: March 2, 1999
Last Modified: March 2, 2008

Contents

1 1D Bisection

The most success I have had is with bisection. The 1D case which most people are familiar with is implemented in `bisect1.h`

```
class Bisect1
{
public:
    typedef float (*Function)(float);

    Bisect1 (Function _F, int _max_level, float _tolerance);

    int Bisect (float x0, float x1);
    float zero;

private:
// input data and functions
    Function F;
    int max_level;
    float tolerance;
};
```

An example of how to use it is

```
#include <iostream.h>
#include <math.h>
#include "bisect1.h"

// Bisection should produce a root at (r2)
// Output:
// root (+0.707031)
// func (-0.000076)

static float r2 = 1.0/sqrt(2.0);

float F (float x) { return x-r2; }

int main ()
{
    Bisect1 bs(F,16,1e-04);
    if ( bs.bisect(0,1) ) {
        float x = bs.zero[0];
        cout << "root at " << x << endl;
        cout << "func (" << F(x) << ')' << endl;
    }
    else
        cout << "root not found" << endl;
    return 0;
}
```

There are no tricks in the code–just straightforward 1D bisection algorithm which keeps track of signs at end points of intervals.

2 2D Bisection

Bisection in 2 dimensions amounts to finding zeros of two equations in two unknowns, say $F(x, y) = 0$ and $G(x, y) = 0$ on a rectangular region in the plane. The algorithm is effectively a quadtree decomposition of the region into subregions, each subregion (presumably) having $F \neq 0$ or $G \neq 0$. I say presumably since the algorithm checks the signs of F and G at the vertices of the subregions. If F has the same sign at the four vertices, then the algorithm stops processing that subregion (maybe missing roots).

The class declaration is

```
class Bisect2
{
public:
    typedef float (*Function)(float,float);

    Bisect2 (Function _F, Function _G, int _max_level, float _tolerance);

    int Bisect (float x0, float y0, float x1, float y1);
    float zero[2];

private:
    typedef struct _b2node {
        float x, y, f, g;
        struct _b2node* xnext;
        struct _b2node* ynext;
    } Bisect2_node;

    // input data and functions
    Function F, G;
    int level, max_level;
    float tolerance;

    // fixed storage to avoid stack depletion during recursion
    float x0, xm, x1, y0, ym, y1;
    float F00, F10, F01, F11, F0m, F1m, Fm0, Fm1, Fmm;
    float G00, G10, G01, G11, G0m, G1m, Gm0, Gm1, Gmm;
    int netsign;
    Bisect2_node* temp;
```

```
        // the graph and recursion routine for building it
        Bisect2_node *graph;
        int BisectRecurse (Bisect2_node* n00);

        // utility functions
        int Sign (float v) { return (v > 0 ? +1 : (v < 0 ? -1 : 0)); }
        int Abs (int v) { return (v >= 0 ? v : -v); }
};
```

An example of how to use it is

```
#include <iostream.h>
#include <math.h>
#include "bisect2.h"

// Bisection should produce a root at (r2,r3).
// Output:
// root (+0.707031,+0.577393)
// func (-0.000076,+0.000042)

static float r2 = 1.0/sqrt(2.0);
static float r3 = 1.0/sqrt(3.0);

float F (float x, float y) { return x-r2; }
float G (float x, float y) { return y-r3; }

int main ()
{
    Bisect2 bs(F,G,16,1e-04);
    if ( bs.bisect(0,0,1,1) ) {
        float x = bs.zero[0], y = bs.zero[1];
        cout << "root at (" << x << ',' << y << ')' << endl;
        cout << "func (" << F(x,y) << ',' << G(x,y) << ')' << endl;
    }
    else
        cout << "root not found" << endl;

return 0;
}
```

In order to reduce recalculation of the functions at subregion vertices, I save the values in a graph structure. Each node has up to four arcs, one each for left, right, up, or down neighbors. Initially the graph has four nodes representing the four vertices of the rectangle to be searched. The algorithm is recursive (the quadtree approach).

- If F or G does not change sign at the four vertices, the region is rejected for further processing.
- If there is a sign change in both functions, then the functions are evaluated at the center of the

rectangle. If the values at the center are close enough to zero, then that point is saved in `zero[]` and the methods terminate (and gracefully remove the graph structure created to that time).

- If at the center the functions are not close enough to zero, then the rectangle is divided into four subregions whose vertices consist of the original four, the center point, and the bisectors of each edge of the original region. Five nodes are added to the graph which store the function values at those new nodes.
- The algorithm is repeated on each subregion until either a root is found or a maximum number of levels has been reached in the quadtree. (Tolerance for closeness to zero and maximum level are inputs to the class constructor.)

3 3D Bisection

Bisection in 3 dimensions amounts to finding zeros of three equations in three unknowns, say $F(x,y,z) = 0$, $G(x,y,z) = 0$, and $H(x,y,z) = 0$ on a rectangular solid in space. The algorithm is effectively an octree decomposition of the region into subregions, each subregion (presumably) having $F \neq 0$ or $G \neq 0$ or $H \neq 0$. Again I say presumably since the algorithm checks the signs of F, G, and H at the vertices of the subregions. If F has the same sign at the eight vertices, then the algorithm stops processing that subregion (maybe missing roots).

The class declaration is

```
class Bisect3
{
public:
    typedef float (*Function)(float,float,float);

    Bisect3 (Function _F, Function _G, Function _H, int _max_level,
        float _tolerance);

    int Bisect (float x0, float y0, float z0, float x1, float y1, float z1);
    float zero[3];

private:
    typedef struct _b3node {
        float x, y, z, f, g, h;
        struct _b3node* xnext;
        struct _b3node* ynext;
        struct _b3node* znext;
    } Bisect3_node;

    // input data and functions
    Function F, G, H;
    int level, max_level;
    float tolerance;
    // vertex and midpoint locations
    float x0, xm, x1, y0, ym, y1, z0, zm, z1;

    // vertices
    float F000, F100, F010, F110, F001, F101, F011, F111;
    float G000, G100, G010, G110, G001, G101, G011, G111;
    float H000, H100, H010, H110, H001, H101, H011, H111;

    // edges
    float F00m, F10m, F01m, F11m, F0m0, F1m0, F0m1, F1m1, Fm00, Fm10, Fm01,
        Fm11;
    float G00m, G10m, G01m, G11m, G0m0, G1m0, G0m1, G1m1, Gm00, Gm10, Gm01,
        Gm11;
    float H00m, H10m, H01m, H11m, H0m0, H1m0, H0m1, H1m1, Hm00, Hm10, Hm01,
        Hm11;

    // faces
    float F0mm, Fm0m, Fmm0, F1mm, Fm1m, Fmm1;
    float G0mm, Gm0m, Gmm0, G1mm, Gm1m, Gmm1;
    float H0mm, Hm0m, Hmm0, H1mm, Hm1m, Hmm1;

    // center
    float Fmmm, Gmmm, Hmmm;

    int netsign;
    Bisect3_node* temp;

    // the graph and recursion routine for building it
    Bisect3_node* graph;
    int BisectRecurse (Bisect3_node* n000);

    // utility functions
    int Sign (float v) { return (v > 0 ? +1 : (v < 0 ? -1 : 0)); }
    int Abs (int v) { return (v >= 0 ? v : -v); }
};
```

An example of how to use it is

```
#include <iostream.h>
#include <math.h>
#include "bisect3.h"

// Bisection should produce a root at (r2,r3,r5).
// Output:
// root (+0.707031,+0.577393,+0.447266)
// func (-0.000076,+0.000042,+0.000052)

static float r2 = 1.0/sqrt(2.0);
static float r3 = 1.0/sqrt(3.0);
static float r5 = 1.0/sqrt(5.0);

float F (float x, float y, float z) { return x-r2; }
float G (float x, float y, float z) { return y-r3; }
float H (float x, float y, float z) { return z-r5; }

int main ()
{
    Bisect3 bs(F,G,H,16,1e-04);
    if ( bs.bisect(0,0,0,1,1,1) ) {
        float x = bs.zero[0], y = bs.zero[1], z = bs.zero[2];
        cout << "root at (" << x << ',' << y << ',' << z << ')' << endl;
        cout << "func (" << F(x,y,z) << ',' << G(x,y,z) << ',' << H(x,y,z)
             << ')' << endl;
    }
    else
        cout << "root not found" << endl;

return 0;
}
```

In order to reduce recalculation of the functions at subregion vertices, I save the values in a graph structure. Each node has up to six arcs, one for each of the six possible axis directions along which neighbors lie. Initially the graph has eight nodes representing the eight vertices of the rectangular region to be searched. The algorithm is recursive (the octree approach).

- If F, G, or H does not change sign at the eight vertices, the region is rejected for further processing.
- If there is a sign change in all functions, then the functions are evaluated at the center of the region. If the values at the center are close enough to zero, then that point is saved in `zero[]` and the methods terminate (and gracefully remove the graph structure created to that time).
- If at the center the functions are not close enough to zero, then the rectangle is divided into eight subregions whose vertices consist of the original eight, the region center point, the centers of each face of the original region, and the bisections of each edge of the original region. Nineteen nodes are added to the graph which store the function values at those new nodes (1 center, 6 faces, 12 edges).
- The algorithm is repeated on each subregion until either a root is found or a maximum number of levels has been reached in the octree. (Tolerance for closeness to zero and maximum level are inputs to the class constructor.)

The code for 3D bisection is overwhelming. The main problem is making sure the graph is modified properly.