Isotopic Meshing of Intersection of Implicit Surfaces in Higher Dimension

David Cadé

September 15, 2008

Contents

1 Introduction

The problem I have attempted to solve is to compute a mesh, that is, a collection of line segments, of the curve or the set of curves resulting from the intersection of $(n-1)$ implicit surfaces in $\mathbb{R}^n$, with the restriction that a tangent vector exists at every point of this curve. This way, there cannot be any singularities on it. The problem of finding and identifying the singularities of a curve algorithmically is an open problem.

To do this, I have adapted the algorithm in the paper of Simon Plantinga and Gert Vegter [6] that gets an isotopic approximation of the contour generator of a 3D implicit surface. The contour generator is the intersection between the implicit surface f and the implicit surface defined by the derivative of f in the view direction. This algorithm first solves an equation of 3 equations in 3 unknowns using the Interval Newton Method to get initial points on every component of the curve, and then, for each of these points, it will follow the curve by iteratively getting a next point on the curve near enough to be sure that it is isotopic to the line segment. So I first adapted the algorithm to make it work with two implicit surfaces and subsequently I have changed it to make it work in any dimension. This needed some profound changes. I have then established the correctness of my algorithm. The solving process is slow, so I have then tried to get a better way to do it.

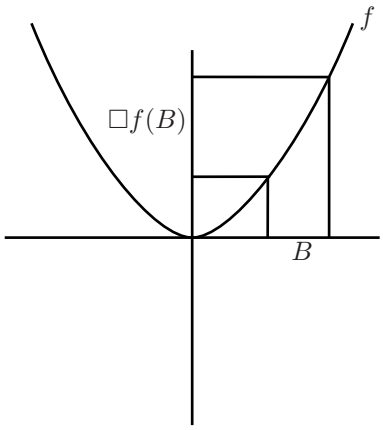

Figure 1: Interval arithmetic : the box B and its image under $\Box f$

Related work There exists another way to do what I attempt by using a method that subdivides the space, and then deduces on each box where the curve goes through. An algorithm that works like that in 2D has been written in another paper by S. Plantinga and G. Vegter [7]. Tracing the intersection of two implicit surfaces isotopically has also been tackled by Bernard Mourrain et al. [5], that works only with algebraic implicit functions and uses the Bernstein representation of the polynomials.

In the following, we will first review the fundamental results required in the algorithm. Then, we will outline the algorithm and prove the correctness. Then we will discuss some implementation issues.

2 Preliminaries

We will first discuss interval arithmetic that permits to prevent rounding errors on floating point numbers, and then present the general solver we will use, the Interval Newton Method.

2.1 Interval arithmetic

Let $f : \mathbb{R}^m \to \mathbb{R}^n$ be a function. An *inclusion function* $\Box f$ is a function from the set of m-boxes to the set of n-boxes, such that for every m-box B, if $x \in B$, then $f(x) \in \Box f(B)$.

An inclusion function is *convergent* if the width of $\Box f(B)$ converges to 0 when the width of B converges to 0.

For example, let us take the function $f(x) = x^2$ from $\mathbb{R}$ to $\mathbb{R}$. A convergent inclusion function for f is :

$$\Box f([a,b]) = \begin{cases} \left[\min(a^2,b^2), \max(a^2,b^2)\right], & \text{if} \quad a \cdot b \geq 0 \\ \left[0, \max(a^2,b^2)\right], & \text{if} \quad a \cdot b < 0 \end{cases}$$

For every basic operator, like addition or division, and function, there exists an inclusion function and we assume that for the functions we will use there

exists an inclusion function, and we will denote them by the same name as the function itself.

There are several libraries that implement interval arithmetic, like C-XSC [2], or PROFIL/BIAS [3]. The two of them work by using a new type that can store intervals, and by using the upper and lower rounding features of the IEEE 754 spec.

2.2 Interval Newton Method

The Interval Newton Method gets all the roots of a function $f : \mathbb{R}^n \to \mathbb{R}^n$ in a box I.

If $0 \notin f(I)$, then we are sure that there is no point where $f = 0$ inside I. So we first subdivide I until the boxes are "small enough", while discarding boxes where we have this property.

The size of the "small enough" boxes to use influences the speed of the algorithm, but does not change the guarantee to get all solutions.

We then refine the solutions on the remaining small boxes by using a Newton method. For the Newton step we solve :

$$f(x) + J(I)(z - x) = 0$$

Where x is the centre of I, J is the jacobian matrix of f and $J(I)$ is the interval matrix of J over I. We then get an interval Y that contains every root z of f that are in I and then we can use Y to refine I. We know also that there is an unique root of f in I if $Y \subset I$. More details can be found in [4].

There also are several implementations of this algorithm, like ALIAS [1] which uses the PROFIL/BIAS library as backend to do it, and C-XSC.

2.3 Miscellaneous notations

For a function $f : \mathbb{R}^3 \to \mathbb{R}$, f_x, f_y, and f_z denotes the 3 derivatives relative to the three cartesian coordinates, i.e. $f_x = \frac{\partial f}{\partial x}$.

For a function $f : \mathbb{R}^n \to \mathbb{R}$, f_{x_i} denotes the derivative relative to the i^{th} cartesian coordinate, i.e. $f_{x_i} = \frac{\partial f}{\partial x_i}$.

3 The algorithm

Let $f^{(1)}, \ldots, f^{(d-1)}$ be $d - 1$ functions of $\mathbb{R}^d \to \mathbb{R}$. We construct an isotopic approximation of the intersection of the following hypersurfaces :

$$\left\{ x \in \mathbb{R}^d | f^{(1)}(x) = 0 \wedge \ldots \wedge f^{(d-1)}(x) = 0 \right\}$$

The first step of my work was to adapt the algorithm of [6] that found an isotopic approximation of the contour generator of an implicit surface defined by the function f, which is the intersection of $f = 0$ and $f_z = 0$. This part was relatively easy, it was for the most part replacing f_z by g in the algorithm. The second step was to adapt it to any number of functions. This demanded more work.

The functions $f^{(1)}, \ldots, f^{(d-1)}$ must be at least C^2 to guarantee the existence of the derivatives and their continuity, and at each point of the intersection curve there must exist a tangent. So each point of the curve is *regular*.

We shall prove now that the curve is regular:

To get a tangent vector to the intersection curve, I have used a generalization of the cross-product.

Let $(e_1, \ldots, e_d)$ be the canonical orthonormal base of $\mathbb{R}^d$, that is to say

$$e_i = (0, \ldots, 0, \underbrace{1}_{\text{at } i^{\text{th}} \text{position}}, 0, \ldots, 0))$$

By using the Cramer's rule, we get a vector perpendicular to every $\nabla f^{(i)}$ at p, so let $(w_i)_{1 \leq i \leq d} : \mathbb{R}^d \to \mathbb{R}$ be defined as these determinants:

$$w_i(p) = \begin{vmatrix} f_{x_1}^{(1)}(p) & \cdots & f_{x_{i-1}}^{(1)}(p) & f_{x_{i+1}}^{(1)}(p) & \cdots & f_{x_d}^{(1)}(p) \\ \vdots & & \vdots & \vdots & & \vdots \\ f_{x_1}^{(d-1)}(p) & \cdots & f_{x_{i-1}}^{(d-1)}(p) & f_{x_{i+1}}^{(d-1)}(p) & \cdots & f_{x_d}^{(d-1)}(p) \end{vmatrix}$$

A tangent vector at the point p is defined by:

$$w(p) = \sum_{i=1}^{d} (-1)^{i+1} w_i(p) e_i$$

Let f be the function $f(x_1, \ldots, x_{d-1}) \mapsto (f^{(1)}(x_1), \ldots, f^{(d-1)}(x_{d-1}))$. Let us take a point p_0 on the curve, p is regular: $w(p_0) \neq 0$. So there exists an i where $w_i(p_0) \neq 0$, and then the following matrix is invertible:

$$\begin{bmatrix} f_{x_1}^{(1)}(p_0) & \cdots & f_{x_{i-1}}^{(1)}(p_0) & f_{x_{i+1}}^{(1)}(p_0) & \cdots & f_{x_d}^{1}(p_0) \\ \vdots & & \vdots & \vdots & & \vdots \\ f_{x_1}^{(d-1)}(p_0) & \cdots & f_{x_{i-1}}^{(d-1)}(p_0) & f_{x_{i+1}}^{(d-1)}(p_0) & \cdots & f_{x_d}^{(d-1)}(p_0) \end{bmatrix}$$

By using the Implicit Function Theorem on f, we have a continuously differentiable function $g = (g_1, \ldots, g_{d-2})$ from $\mathbb{R}^{d-2}$ to $\mathbb{R}$, an open set $U \subset \mathbb{R}^{d-1}$ and $p_0 \in U$, such that the function

$$h : x_i \mapsto (g_1(x_i), \ldots, g_{i-1}(x_i), x_i, g_i(x_i), \ldots, g_{d-2}(x_i))$$

is a parametrization of the intersection curve $f = 0$ locally in U, that is to say $f(h(x_i)) = 0$ for every $h(x_i) \in U$.

The curve is then regular locally near p_0, and that for all p_0. So the curve is regular.

The algorithm proceeds in two steps :

- Getting (at least) a point on every component of the curve.
- For each point found, trace the component of the curve containing this point.

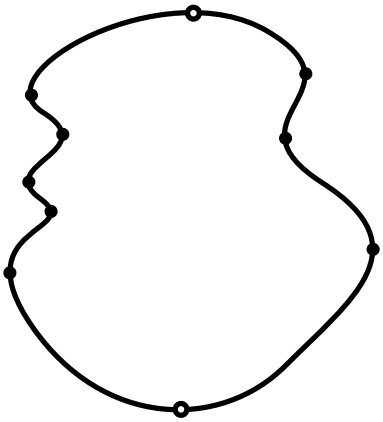

Figure 2: A curve that has more points where $w_1 = 0$ (black dots) than $w_2 = 0$ (white dots).

3.1 Getting initial points

We will suppose that the curve is bounded in an interval the user will give us, so the components of the curve are closed and bounded, so there is at least one point where the first component of the tangent vector is zero on every component of the curve.

Let $S : \mathbb{R}^d \to \mathbb{R}^d$ be the function

$$S(p) = \begin{pmatrix} f^{(1)}(p) \\ \vdots \\ f^{(d-1)}(p) \\ w_1(p) \end{pmatrix}$$

The roots of the function S will be the initial points, and can be retrieved with the Interval Newton Method.

If the curve is not bounded, we can get the non-bounded components by searching additional initial points on the boundaries of the box.

Alternative method I have also thought of an alternative method to get these points.

The above way to get the initial points does depend only on the first component of the tangent, but sometimes a curve has a "privileged" direction in which the number of solutions to the above equation would be less than in the first direction. See figure 2. In order to get that direction we subdivide the space and we count the number of boxes B for each direction i where we have $0 \in w_i(B)$.

We subdivide the space until in each box B one of these condition holds:

- There exists a i where $0 \notin f_i(B)$, that is to say that the curve does not go through this box at all, and don't keep it in memory,
- We have $0 \in f_i(B)$, and there exists a i where $0 \notin \langle w(B), e_i \rangle$, that is to say the curve is parameterizable in the direction e_i. We keep then this

box in memory with a list of every i where the curve is parameterizable in the direction i.

We also stop subdividing if the box becomes too small, by using an user defined ε parameter.

Let $S_j : \mathbb{R}^d \to \mathbb{R}^d$ be the function

$$S_j(p) = \begin{pmatrix} f^{(1)}(p) \\ \vdots \\ f^{(d-1)}(p) \\ w_j(p) \end{pmatrix}$$

These functions are similar to the S function above, but solve it for the ith component of the tangent.

If $w_j(B) = 0$, then the curve is not parameterizable in B in the direction j.

Then we count the number of boxes where the curve is not parameterizable for each direction, and let j be one of the directions that correspond to the smallest number of boxes.

Then we solve $S_j = 0$ on all the boxes where the curve is not parameterizable in the direction j with an Interval Newton Method, and get the initial points.

This permits to get heuristically a "best direction" where there should be the least number of solutions found, and also not to solve on the entire space.

3.2 Tracing

After getting the initial points, we compute for each of them a set of segments that represent the curve. This set of segments is represented by a list of points $p_i, 0 \leq i \leq N$, and the segments are every $p_i p_{i+1}$ and $p_N p_0$. This last segment is not included if the component of the curve is not closed, of course.

This piecewise linear approximation will be isotopic to the component of the curve on which the point lies.

So let p_0 be a point on the curve. We got also a user-defined parameter δ which is the step size. We follow the tangent at the point p_i : $p_{i+1} = p_i + \delta \frac{w(p_i)}{\|w(p_i)\|}$, and get it "near the curve", that gives a candidate point for the next point on the list. If it is a "good approximation", then we continue at the point p_{i+1} and put $\delta := 1.2\delta$, else we retry with a smaller δ.

We need also to know when we get back at the first point. If we are in range of p_0, that's to say that $\|p_i - p_0\| \leq \delta$, we set p_0 as the next candidate point.

3.2.1 Getting near the curve

I will describe here a function that takes a point, p, and moves it near the curve.

Let $(N_i)_{1 \leq i \leq d-1}$ be the functions :

$$N_i(p) = p - \frac{f_i(p)}{\|\nabla f_i(p)\|^2} \nabla f_i(p)$$

N_i follows the gradient of f_i to get near the hyper-surface $f_i = 0$.

Let $p_0 = p$ and let p_{n+1} be $p_{n+1} = N_{d-1}(N_{d-2}(\dots N_1(p_n)\dots))$.

This sequence goes nearer and nearer to the curve, if we have good conditions on the gradients of the functions f_i near the points.

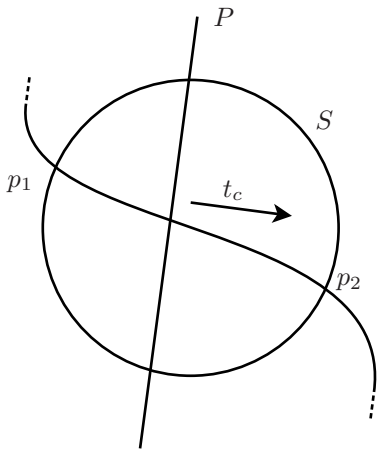

Figure 3: The sphere S for the points p_1 and p_2, a plane P perpendicular to t_c

When the distance $\|p_{n+1} - p_n\|$ is sufficiently small, i.e. inferior to an ε we have chosen, we set p to p_{n+1}.

3.2.2 Is it a good approximation?

The following function test the following property: If p_1 and p_2 are two points that are on the curve, is the curve between p_1 and p_2 isotopic to the line segment that connects these two points ?

The way it is done in [6] is not really adaptable to more than 3 dimensions, so I had to change completely the way it is done.

Let S be the sphere of center $\frac{p_1+p_2}{2}$ and radius $\frac{\|p_2-p_1\|}{2}$. Let B be the bounding box of this sphere, and let

$$I = \langle w(B), w(B) \rangle$$

If $I > 0$, the curve is isotopic to the segment.

Proof :

Let c be the center of the sphere S, and let be $t_c = \frac{w(c)}{\|w(c)\|}$.

Take a plane P perpendicular to t_c, and suppose there are two points q_1 and q_2 of the curve inside the intersection of the sphere S and the plane P.

The sphere being convex, the line segment $\gamma(s) = q_1 + s(q_2 - q_1)$ for $s \in [0, 1]$ stays in the sphere and on the plane P.

For all i, we have $f_i(q_1) = f_i(q_2) = 0$, and according to the Mean Value Theorem applied to $f_i \circ \gamma$, there is a point $c_i \in]0, 1[$ where $(f_i \circ \gamma)'(c_i) = 0$, that is to say $\langle \nabla f_i(\gamma(c_i)), q_2 - q_1 \rangle = 0$.

So, let r_i be $\gamma(c_i)$. We have $r_i \in S$. We have $w(B) = \sum_{i=1}^{d} (-1)^{i+1} w_i(B) e_i$, and we have

$$t_j = \begin{vmatrix} f_{x_1}^{(1)}(r_1) & \cdots & f_{x_{j-1}}^{(1)}(r_1) & f_{x_{j+1}}^{(1)}(r_1) & \cdots & f_{x_d}^{(1)}(r_1) \\ \vdots & & \vdots & \vdots & & \vdots \\ f_{x_1}^{(d-1)}(r_{d-1}) & \cdots & f_{x_{j-1}}^{(d-1)}(r_{d-1}) & f_{x_{j+1}}^{(d-1)}(r_{d+1}) & \cdots & f_{x_d}^{(d-1)}(r_{d-1}) \end{vmatrix} \in w_j(B)$$

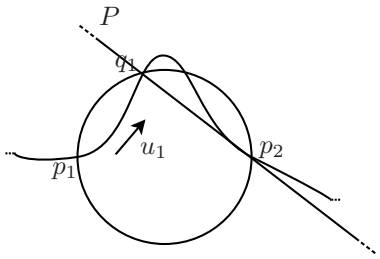

Figure 4: The curve escaping the sphere

and

$$t = \sum_{i=1}^{d} (-1)^{i+1} t_i e_i$$

This vector t is perpendicular to every $\nabla f_i(r_i)$, and then we have $\langle t, t_c \rangle = 0$, and $I \not> 0$.

So if $I > 0$ then there is at most one point in every plane perpendicular to t_c.

Remark we can interchange t_c above with any vector in $w(B)$.

Now we need to prove that the curve cannot escape the sphere, so there is exactly one point of the curve in every plane perpendicular to t_c that intersects S. Then there can be only one component of the curve in the sphere S, and is clearly isotopic to the line segment p_1p_2.

If the curve does not stay in the sphere, then there exists q_1, the first point of the curve between p_1 on the boundary of S. We shall prove $I \not> 0$ in this case.

Let u_1 be the mean of the tangent between p_1 and q_1, and for all $1 \leq i \leq d$, by using the Mean Value Theorem, there is a point between p_1 and q_1 on the curve where the ith component of the tangent is the ith component of u_1. So the point u_1 is in $w(B)$. The plane P perpendicular to u_1 going through q_1 goes also through p_2, and so by using the same proof as in the first part with $u_1 \in w(B)$ and P, we prove that $0 \in I$. So if the curve does not stay in the sphere, $I \not> 0$. Then we have proven that the curve stays in the sphere, and is isotopic to the line segment p_1p_2.

4 Implementation

I will first comment on how I have designed the program that solves this problem, and then discuss some problems encountered.

4.1 Design

I have used C++ to develop this implementation.

A central class in the code is the `ScalFunc` class that represents a scalar function of $\mathbb{R}^n \to \mathbb{R}$.

There is a `Solver` class that takes an array of $d-1$ of these and solves the S or S_i functions. And the `Tracer` creates a `ContourData` that is basically a list of

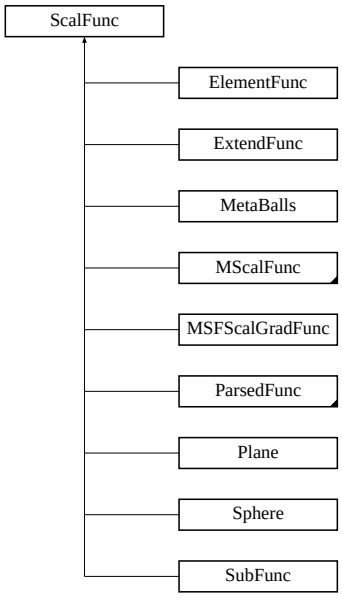

Figure 5: `Scalfunc` and some of its derivatives

points for every point it is given using the algorithm we discussed in the previous section. Then the "glue" between the `Tracer` and the `Solver`, the `Mesher` takes only the list of functions in parameter and gets the complete approximation. It permits also to visualize the data after it has been calculated.

4.1.1 Functions

Functions are represented by the class `ScalFunc` which is an abstract class containing 3 functions :

1. `f` which gets the interval value at the given box,
2. `gf` which gets the interval gradient vector at the given box,
3. `Bounds` which gives the bounds of the function.

I have done many classes that derive from this, some simple ones like `Plane` or `Sphere`, also `Metaballs`, and a more complex one that uses bison and flex to parse a string, and do some automatic derivation.

To achieve this I needed to implement another class, `Element`, where there the derivative is also an `Element`. It is designed to be easy to add more functions to the existing ones by just deriving your new operators or functions directly. See the figure 6).

4.1.2 Solver

To get the initial points, we need to call a solver. At the beginning I was using the C-XSC as the interval arithmetic library and I used the Interval Netwon

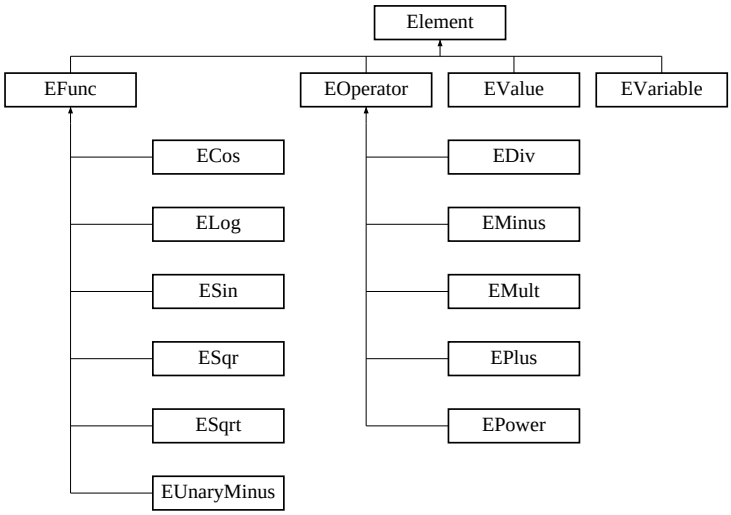

Figure 6: The derivatives of the `Element` class

Method solver from this library, but I have found some strange behavior with non-algebraic curves, like not finding some points that should have been found.

I therefore use the ALIAS library, that is much more complex to configure to get what you want (the solving function has more than 10 parameters!), but seems to work flawlessly most of the time. When the curve is complicated, it takes much more time and gives also sometimes some false positives, or even run out of memory.

The solver is encapsulated in the `Solver` class, which will keep the points the solver has found.

I have also implemented a `FileSolver` class which fetches the solutions from a file, thus just having to solve the function once, and after just getting the results from a file where we would have saved the points.

I have also implemented the Alternative Method I have talked about in the previous section.

4.1.3 Tracer

That class keeps all the lists of points representing the set of line segments. When given a point, it will calculate the approximation of the curve and add it to its list.

4.1.4 Mesher

This class encapsulates the `Solver` and the `Tracer`, and makes them work together. It fetches the solutions from the `Solver` and give them to the `Tracer`.

After having calculated the approximation, it is able to save it to a file, to display it on screen, or to save it in an image file.

Intersection of parametric surfaces I have also created a `Parametric` class that derives from this one, which takes the 6 functions $f^{(x)}$, $f^{(y)}$, $f^{(z)}$, $g^{(x)}$,

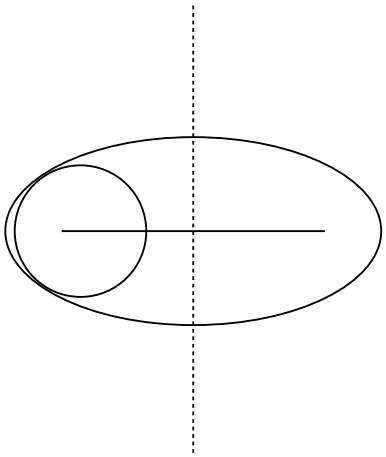

Figure 7: An ellipse, its medial axis (the plain line), its symmetry set (the union of the dashed line and the medial axis), and a circle tangent to the ellipse in two points

$g^{(y)}$,$g^{(z)}$ from $\mathbb{R}^2 \to \mathbb{R}$, which are the components of 2 parametric surfaces $f = (f^{(x)}, f^{(y)}, f^{(z)})$ and $g = (g^{(x)}, g^{(y)}, g^{(z)})$.

The problem of finding the intersection of these 2 surfaces can be reduced at finding the intersection of the 3 following hyper-surfaces in dimension 4:

$$\begin{cases} f^{(x)}(u,v) - g^{(x)}(u',v') \\ f^{(y)}(u,v) - g^{(y)}(u',v') \\ f^{(z)}(u,v) - g^{(z)}(u',v') \end{cases}$$

The `Parametric` class creates these functions, and solves this problem. Then it transforms the (u, v, u', v') points it have found in $(f(u, v))$. This way we can use the Mesher's drawing routines.

I have first tried to intersect a plane parametrized by $(u, v) \mapsto (u, v, 0)$ and a sphere parametrized by $(u, v) \mapsto (\sin u \sin v, \cos u \sin v, \cos v)$.

The number of points in the approximation reached the limit, because the tracer turned around without seeing the first point, because the points (u, v, u', v') and $(u, v, u' + 2\pi, v')$ are different. In order not to have these big lists, if it goes out of the bounding box, it stops the calculation for this component.

Another way, more elegant I think, but harder, would be to derive the `INTERVAL` type to do some modulo calculations. I have not pursued this idea because it is out of the scope of this report.

Symmetry set and Medial axis The medial axis of a curve C is the closure of the set of the centers of circles that are contained in C and are tangent to C in two or more points.

The medial axis is included in the symmetry set, which is the closure of the set of the centers of the circles that are tangent to C in two or more points.

In order to get the medial axis of an implicit curve f, we could begin to try to calculate the symmetry set, which can be represented by the following equations in x, y, x', y':

- $f(x, y) = 0$,
- $f(x', y') = 0$,
- $\left\langle \frac{\nabla f(x,y)}{\|\nabla f(x,y)\|} - \frac{\nabla f(x',y')}{\|\nabla f(x',y')\|}, \begin{pmatrix} x - x' \\ y - y' \end{pmatrix} \right\rangle = 0.$

But this set of points also includes the points where $x = x'$ and $y = y'$, that we do not want. So we needed to add another equation:

- $u((x - x')^2 + (y - y')^2) = 1$,

This equation just says that the two points (x, y) and (x', y') are at distance $1/u$ from each other. We have then a set of 4 equations in 5 dimensions.

In this set of equations, we have also divisions by the norm of the gradient of f, that is 0 at least one time inside of a closed curve, so I first needed a way to fetch the solutions of the solver only on parts where $\|\nabla f(x, y)\| \neq 0$, by doing a subdivision of the space until on all nodes that would contain the function the gradient is not 0.

But this approach required too much calculations, the solver does take too much time to get the solution points that I have gave up.

The symmetry set can also be found by using the 6 equations in x, y, x_1, y_1, x_2, y_2:

- $f(x_1, y_1) = 0$,
- $f(x_2, y_2) = 0$,
- $d(f(x_1, y_1), f(x, y)) = d(f(x_2, y_2), f(x, y))$ with d the Cartesian distance function,
- $\begin{pmatrix} x_1 - x \\ y_1 - y \end{pmatrix}$ is parallel to $\nabla f(x_1, y_1)$. (Can be calculated with a 2x2 determinant for example)
- $\begin{pmatrix} x_2 - x \\ y_2 - y \end{pmatrix}$ is parallel to $\nabla f(x_2, y_2)$.

And also adding an equation to not get solutions where $(x_1, y_1) = (x_2, y_2)$:

- $u((x_1 - x_2)^2 + (y_1 - y_2)^2) = 1$,

The solver was not able to solve it because it also ran out of memory.

4.2 Issues

The biggest issue was the solver, because in higher dimensions it was not able to solve the functions I gave him. So I have tried to find a way to find a method that would not use the Interval Newton Method.

I have then tried to generalize the subdivision method in [7] or [5], without success. I have also had some other ideas that did not really work well.

Method	Normal		Alternative		
	Solving	Tracing	Subdividing	Solving	Tracing
2 spheres	10ms	10ms	0ms	10ms	30ms
1 sphere + A	130ms	100ms	30ms	80ms	100ms
$A + B$	5060ms	11650ms	510ms	1440ms	6270ms
Parametric	430ms	10ms	120ms	200ms	10ms
Metaballs	720ms	100ms	10ms	430ms	130ms

Figure 8: Benchmarking results

4.3 Benchmarks

The only way I have found to speed up the solving process is the "alternative method" I have described in the solving step of the algorithm.

I have tested and benchmarked the 2 methods in 3 dimensions with A and B defined by:

$$A(x, y, z) = \sin 5x + \sin 5y + 0.5x^2 + y^2 + z^2 - 3$$

$$B(x, y, z) = \sin 2x + \sin 3z + x^2 + 0.5y^2 + z^2 - 5$$

A metaball of weight w, radius r, and center c is defined by the function:

$$m(w, r, c)(p) = we^{\frac{\|p-c\|^2}{r^2}}$$

I have tested also in 2 dimensions with a function that is the sum of 7 metaballs.

I have then also tested the algorithm in 4 dimensions by using the intersection of 2 parametric surfaces:

- The plane $z = 0$:
$$\begin{cases} f^{(x)}(u, v) = u \\ f^{(y)}(u, v) = v \\ f^{(z)}(u, v) = 0 \end{cases}$$
- The unit sphere:
$$\begin{cases} g^{(x)}(u, v) = \sin u \sin v \\ g^{(y)}(u, v) = \cos u \sin v \\ g^{(z)}(u, v) = \cos v \end{cases}$$

Which is the problem I have also described in the section about the parametric surfaces.

The figures 9, 10, 11 are outputs of my program. Each of them have big squares that represent the solutions that the solver gave, and little squares for each point of the set of points.

The figure 8 presents the result of the benchmarking.

So the algorithm works on any dimension without any modification: the metaballs example is the intersection of only one curve in 2 dimensions, the examples involving A and B are intersections of 2 curves in 3 dimensions, the parametric example is the intersection of 3 curves in 4 dimensions.

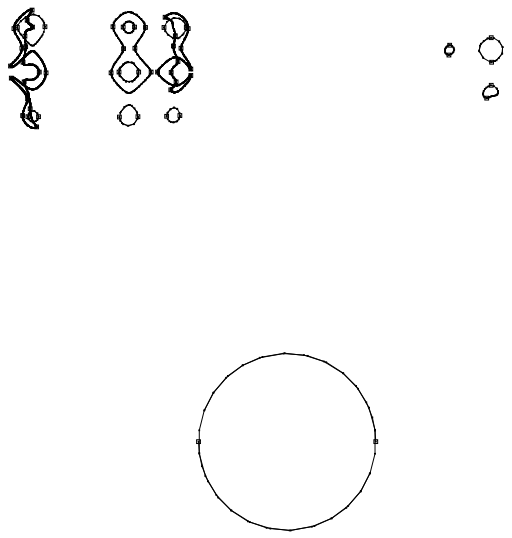

Figure 9: A intersected with B, A intersected with a sphere, and a sphere intersected with a sphere in 3 dimensions, the curves have been projected on the x, y axis.

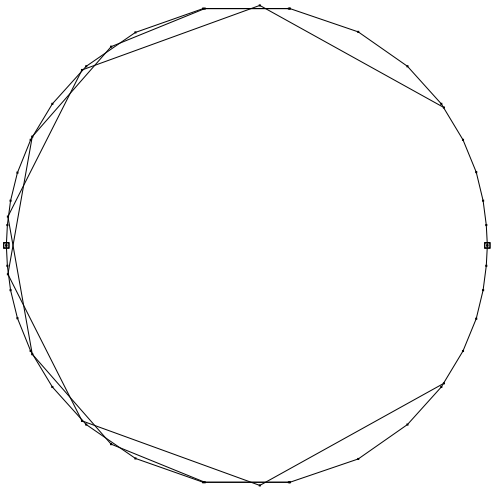

Figure 10: The intersection of a parametric sphere and a parametric plane.

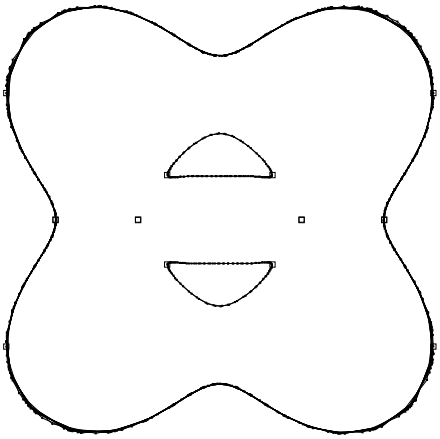

Figure 11: The metaballs in 2 dimensions

As can be seen, on most of the examples, the global time is better when using my alternative method. On all examples, the solving is done faster, the subdividing step help getting the solving process done faster.

On the example $A+B$, it reduced the number of solutions found from 49 to 35, which explains the speed up in the tracing step.

5 Conclusion

We have an algorithm that can mesh an intersection of $d-1$ implicit surfaces in dimension d, the resulting mesh being isotopic to the initial curve. We have also a new algorithm to find one point on every component of the intersection curve.

There are many ways to continue the work I have done, first we could try to take care of finding the discontinuities of the curve, and find an isotopic approximation everywhere where it is not near a singular point. Then we could try to identify what kind of singular point it is and approximate it correctly.

We could also try to get an algorithm completely general that can mesh the intersection of n implicit functions in dimension d.

References

[1] ALIAS, `http://www-sop.inria.fr/coprin/logiciels/ALIAS/`.

[2] C-XSC, `http://www.rz.uni-karlsruhe.de/~iam/html/language/cxsc/cxsc.html`.

[3] PROFIL/BIAS, `http://www.ti3.tu-harburg.de/knueppel/profil/index_e.html`.

[4] E. Hansen and R. I. Greenberg. An interval Newton method. *Applied Mathematics and Computation*, 12:89–98, 1983.

[5] Chen Liang, Bernard Mourrain, and Jean-Pascal Pavone. *Subdivision Methods for the Topology of 2d and 3d Implicit Curves*. HAL - CCSD, November 15 2007.

[6] Simon Plantinga and Gert Vegter. Computing contour generators of evolving implicit surfaces. *ACM Transactions on Graphics*, 25(4):1243–1280, October 2006.

[7] Simon Plantinga and Gert Vegter. Isotopic meshing of implicit surfaces. *The Visual Computer*, 23(1):45–58, 2007.

Acknowledgments I want to thank :

- Gert Vegter and Simon Plantinga for monitoring me.
- J.-P. Merlet for his insightful reply to one of my mails and his ALIAS software.