RC 18777 (82115) 3/16/93
Mathematics

Research Report

COMPUTING IMPLICITLY DEFINED SURFACES: TWO PARAMETER CONTINUATION

Michael E. Henderson

IBM Research Division
T. J. Watson Research Center
Yorktown Heights, NY 10598

COMPUTING IMPLICITLY DEFINED SURFACES: TWO PARAMETER CONTINUATION

Michael E. Henderson

IBM Research Division
T. J. Watson Research Center
Yorktown Heights, NY 10598
e-mail: mhender@watson.ibm.com

ABSTRACT: Implicitly defined surface arise in many different contexts, from physical simulation to CAD and visualization. We describe a continuation algorithm for computing implitily defined surfaces. It is similar to analytic continuation, without the drawbacks of that algorithm. It is well suited to surfaces embedded in high dimensional spaces, has a natural step-size selection, and has no difficulties with global topologies which are not planar (i.e. spheres, tori etc.).

Keywords: Surface Continuation, Continuation Methods, Implicitly Defined Surfaces

1. Introduction

Let $\mathcal{S} \subset \mathbb{R}^{n+2}$ be an implicitly defined surface embedded in $\mathbb{R}^n$. That is, each point $u \in \mathcal{S}$ satisfies an equation

$$g(u) = 0$$
$$g : \mathbb{R}^{n+2} \to \mathbb{R}^n$$

Definition 1. *A regular point is a point $u_0 \in \mathcal{S}$ at which the Jacobian $g_u(u_0) : \mathbb{R}^{n+2} \to \mathbb{R}^n$ is of full rank. That is,*

$$\dim(\mathcal{N}(g_u(u_0))) = 2$$
$$\dim(\mathcal{R}(g_u(u_0))) = n$$

This paper describes an algorithm for computing implicitly defined surfaces which have no non-regular (singular) points.

This problem appears in many different contexts. In Computer Graphics and Computer Aided Design, for example, there are two broad categories of surfaces which are used; parameteric surfaces, and implicitly defined surfaces (e.g. Bloomenthal [4] and Hall and Warren [8]). In Scientific Visualization isosurfaces are used to visualize three dimensional scalar fields, such as temperature distributions, or stress in mechanical structure (e.g. Lorensen and Cline [11]). Finally, in physical simulations there are often many more than two parameters. The state of the physical system is therefore an implicitly defined surface (or a higher-dimensional object whose sections are surfaces), embedded in a space of very large dimension. For example, the distribution of surface elevation near a moving boat is parameterized by Reynolds number (boat speed) and Froud number (the size of the boat relative to free surface waves). The main difference between surfaces occuring in physical simulation and those in CAD and visualization is the size of the embedding space n. In the latter case $n = 3$, while the physical problems live in infinite dimensional spaces (Banach spaces) which are approximated by spaces of dimension $1,000,000$ or larger.

The corresponding problem in one lower dimension, computing implicitly defined curves, has been successfully dealt with by conventional continuation methods (for example, [9], [16], [1], and [12]). Codes exist for large classes of problems, for example AUTO [7], PITCON [16], PLTMG [3], etc. The literature on computing surfaces is not as well developed. Allgower [2] proposed an algorithm based on simplicial continuation, which describes the basic algorithm used in CAD and visualization (e.g. "Marching Cubes" [11]). Rheinbolt [15] proposes using a smoothly varying projection of the tangent space onto the surface ("Moving Frame") to "wrap" a grid onto the surface.

The algorithms based on simplicial continuation work well when n is small, and are particularly simple to implement when no adaptive refinement is used. Algorithms based on the algorithm proposed by Rheinboldt are well suited to large n, but it is difficult to implement adaptive step size selection, and to deal with global connections, such as tori and spheres. The surface continuation algorithm presented here is most closely related to Rheinboldt's algorithm, but includes adaptive step sizes in a natural way, and can handle global

connections. However, Rheinboldt's algorithm extends to higher dimensional manifolds, while ours does not.

The surface continuation algorithm is loosely patterned after analytic continuation in complex analysis (e.g. [5]). This constructs an analytic mapping $f : \mathbb{C} \to \mathbb{C}$, given the value of f at an initial point z_0. First f is constructed in a neighborhood of the initial point. This can be done provided f is analytic at z_0, by constructing a power series for f about z_0. The series converges in some circle U_0, centered at z_0. A point different from z_0 is then chosen from this circle, and a new circle U_1 is constructed. At each step f is approximated by a list of circles $\{U_i\}$. One step of the continuation is to choose a point z_{i+1} near the boundary of some circle U_i, compute a new circle U_{i+1} and add it to the list. (Figure 1.)

One of the problems with analytic continuation, which prevents it from being a pratical proceedure, is the power series expansion of the function. Analytic continuation is similar to an initial value problem, in that the function is only defined local to the center of a circle. Small errors in the continuation grow, and limit the accuracy of the computation. Since points on implicit surfaces satisfy an equation, the Implicit Function Theorem can be used instead of the power series expansion. So unlike analytic continuation, any point in circle U_i can be projected onto the surface, to a given tolerance. This eliminates the growth of round-off errors, and allows the surface to be computed to a global tolerance.

Another motivation for the algorithm comes from viewing the surface as a manifold. A two dimensional manifold $\mathcal{M}$, embedded in $\mathbb{R}^n$, is an atlas of mappings $\{\psi_i\}$

$$\psi_i : U_i \to R^n$$

from neighborhoods of the origin $(0,0) \in U_i$, onto the surface, with the constraint that the mappings agree on the overlap of adjacent neighborhoods. A regular implicitly defined surface is a two dimensional manifold. Given a set of points $\{u_i\} \in \mathcal{S}$, the mappings ψ_i are the mappings from the tangent space onto the surface. These mappings exists in some neighborhood U_i of the origin of the tangent space, by the Implicit Function Theorem. Any given set of points therefore determines some subset of the surface. Our algorithm computes a set of points on the surface, and a set of mappings from the tangent spaces which cover the surface.

2. The Geometry of Implicitly Defined Surfaces

We begin by presenting several properties of implicity defined surfaces which be used in the algorithm. These are not new results, for example, see Rheinboldt [14].

2.1. Regularity of a Basic Linear System

One linear system occurs frequently in the algorithm. For large n its solution will dominate the computation.

Lemma 1. *Let u_s and u_t be an orthonormal basis for $\mathcal{N}(g_u(u_0))$ at a regular point u_0 on a surface S. That is*

$$\begin{aligned} g(u_0) &= 0 \\ g_u(u_0)u_s &= 0 \\ g_u(u_0)u_t &= 0 \\ u_s^* u_s &= 1 \\ u_s^* u_t &= 1 \\ u_s^* u_t &\neq 1. \end{aligned}$$

If u_s^a and u_t^a are any two vectors which have non-zero projections onto the null space of $g_u(u_0)$, and whose projections on the nullspace span the nullspace, then the linear system

$$\begin{bmatrix} g_u(u_0) \\ (u_s^a)^* \\ (u_t^a)^* \end{bmatrix}$$

is non-singular.

Proof 1. *Consider the system*

$$\begin{aligned} g_u(u_0)w &= a \\ (u_s^a)^* w &= b \\ (u_t^a)^* w &= c \end{aligned}$$

Since the range of g_u is all of $\mathbb{R}^n$, all solutions of $g_u w = a$ can be expressed as $w = w_p + \alpha u_s + \beta u_t$, where w_p is the unique solution of $g_u w_p = a$ with no component in the null space of g_u. α and β are determined by

$$\begin{pmatrix} (u_s^a)^* u_s & (u_s^a)^* u_t \\ (u_t^a)^* u_s & (u_t^a)^* u_t \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} b \\ c \end{pmatrix},$$

which is invertible given the assumptions on u_s^a and u_t^a.

2.2. Computing a basis for the Tangent Space of the Surface

There is some freedom in choosing a basis for the null space of g_u, which can be resolved by specifying two approximate basis functions (u_s^a, u_t^a). A basis can be computed by solving the non-singular systems

$$\begin{aligned} g_u(u_0)u_s &= 0 & \qquad g_u(u_0)u_t &= 0 \\ (u_s^a)^* u_s &= 1 & \qquad (u_t^a)^* u_t &= 1 \\ (u_t^a)^* u_s &= 0 & \qquad (u_s)^* u_t &= 0 \end{aligned}$$

and then normalizing u_s and u_t. The choice of basis is not critical to the algorithm, and there is no constraint of continuity of basis between neighborhoods.

2.3. Mapping the Tangent Space onto the Surface

Now that we have a basis for the tangent space, we can construct the mapping from the tangent space onto the surface.

Lemma 2. *For each regular point of a surface $\mathcal{S}$,there is a neighborhood of $(0,0) \in \mathbb{R}^2$ in the tangent space, and a unique mapping $a(s,t)$, from the $\mathbb{R}^2 \to \mathbb{R}^{n+2}$, such that*

$$u(s,t) = u_0 + su_s + tu_t + a(s,t) \in \mathcal{S}.$$

Proof 2. *The Implicit Function Theorem applied to the system*

$$\begin{aligned} &g(u_0 + su_s + tu_t + a) = 0 \\ &u_s^* a = 0 \\ &u_t^* a = 0, \end{aligned}$$

with u_s and u_t a basis for the tangent space, gives the result.

A Taylor series for the map $a(s,t)$ can be computed. The lowest order terms are

$$a(s,t) = \frac{1}{2}(a_{ss}s^2 + 2a_{st}st + a_{tt}t^2) + O(s^3, s^2t, st^2, t^3),$$

where

$$\begin{array}{cc} g_u(u_0)a_{ss} = -g_{uu}(u_0)u_s u_s & g_u(u_0)a_{st} = -g_{uu}(u_0)u_s u_t \\ u_s^* a_{ss} = 0 & u_s^* a_{st} = 0 \\ u_t^* a_{ss} = 0 & u_t^* a_{st} = 0 \end{array}$$

$$\begin{aligned} g_u(u_0)a_{tt} &= -g_{uu}(u_0)u_t u_t \\ u_s^* a_{tt} &= 0 \\ u_t^* a_{ss} &= 0 \end{aligned}$$

The third and higher order terms may be found by solving similar systems.

2.4. Estimating the Region in which the Mapping is valid

The Implicit Function Theorem guarentees that a exists in some neighborhood of the origin. There are various ways to estimate the size of this ball. Some of these have been used to determine stepsize in continuation algorithms [6], [17], [10]. An estimate of the size in terms of local quantites is given in [10], pages 23–24. This is basically an estimate of the condition number of g_u, or the distance to the nearest singular point.

Lemma 3. *Let*

$$G(u) = \begin{pmatrix} g_u(u_0 + su_s + tu_t + a) \\ u_s^* a \\ u_t^* a \end{pmatrix}$$

and

$$M_0 \geq \| G_u(u_0) \|$$

$$K_0 \geq \max(\| G_s(u_0) \|, \| G_t(u_0) \|)$$

$$K_1 \geq \| G_{uu}(u_0) \|$$

$$K_2 \geq \max(\| G_{us}(u_0) \|, \| G_{ut}(u_0) \|)$$

$$B = M_0 K_2$$

$$A = M_0^2 K_0 K_1 + B$$

then $a(s,t)$ exists and is unique inside a ball approximately of size

$$\rho(u_0) \sim \frac{\left[1 - \frac{A}{B}\left(1 - \sqrt{1 - \frac{B}{A}}\right)\right]\left[1 - \sqrt{1 - \frac{B}{A}}\right]}{B\sqrt{1 - \frac{B}{A}}}$$

This estimate often underestimates the size of the region. There is a second constraint, that the distance between the tangent space and the surface be small, which is more practical.

In what follows, U_i will always be an ellipse centered at the origin of the tangent space, with axes r_s and r_t. To compute the surface to a given accuracy, we will require that $a(s,t) < \epsilon$ inside U_i. One way to do this, which is similar in spirit to the stepsize selection for one parameter continuation of [6], is to choose r_s and r_t so that

$$|\tfrac{1}{2} a_{ss} r_s^2| = \epsilon$$

$$|\tfrac{1}{2} a_{tt} r_t^2| = \epsilon$$

or,

$$r_s = \sqrt{\tfrac{2\epsilon}{|a_{ss}|}}$$

$$r_t = \sqrt{\tfrac{2\epsilon}{|a_{tt}|}}$$

In principle, one could do better by chosing an arbitrary basis for the tangent space, say

$$\tilde{u}_s = \cos\theta_s u_s + \sin\theta_s u_t$$
$$\tilde{u}_t = \cos\theta_t u_s + \sin\theta_t u_t$$

and choosing θ_s, θ_t, r_s and r_t so that

$$\tfrac{\partial}{\partial\theta_s}|a(r_s\cos\theta_s, r_s\sin\theta_s)| = 0$$
$$|a(r_s\cos\theta_s, r_s\sin\theta_s)| = \epsilon$$
$$\tfrac{\partial}{\partial\theta_t}|a(r_t\cos\theta_t, r_t\sin\theta_t)| = 0$$
$$|a(r_t\cos\theta_t, r_t\sin\theta_t)| = \epsilon$$

however, this is more involved, and the benefit is not clear.

2.5. Estimating variations between neighborhoods

These conditions on the size of the U_i impose a restriction on how much the tangent space can change from neighborhood to neighborhood. We have (without orthonormalization)

$$\begin{aligned} u(s,t) &= u_0 + su_s^0 + tu_t^0 + a(s,t) \\ u_s(s,t) &= u_s^0 + a_s(s,t) \\ u_t(s,t) &= u_t^0 + a_t(s,t) \end{aligned}$$

so,

$$\begin{aligned} (u_s^0)^* u_s(s,t) &= 1 + (u_s^0)^* a_s(s,t) \\ (u_t^0)^* u_t(s,t) &= 1 + (u_t^0)^* a_t(s,t) \end{aligned}$$

In choosing the size of the neighborhood U we have bounded the size of the second derivatives of a, so we can estimate the first derivatives of a. We have

$$\begin{aligned} a(s,t) &\sim \tfrac{1}{2}a_{ss}(0,0)s^2 + a_{st}(0,0)st + \tfrac{1}{2}a_{tt}(0,0)t^2 + O(s^3, s^2t, st^2, t^3) \\ a_s(s,t) &\sim a_{ss}(0,0)s + a_{st}(0,0)t + O(s^2, st, t^2) \\ a_t(s,t) &\sim a_{st}(0,0)s + a_{tt}(0,0)t + O(s^2, st, t^2). \end{aligned}$$

Let ξ bound the second derivatives of $a(s,t)$ in U. If (s,t) are the coordinates of the center of the nearby neighborhood, and the center lies within U, then s and t are $O(\sqrt{\frac{\epsilon}{\xi}})$ in U. This gives

$$\begin{aligned} (u_s^0)^* a(s,t) &= O(\epsilon^{3/2}) \\ (u_t^0)^* a(s,t) &= O(\epsilon^{3/2}), \end{aligned}$$

and

$$\begin{aligned} (u_s^0)^* u_s(s,t) &= 1 + O(\epsilon) \\ (u_t^0)^* u_t(s,t) &= 1 + O(\epsilon) \end{aligned}$$

3. Data Structures

In order to describe the algorithm we first introduce some fundemental data structures. The syntax is that of the C language, but any language which can represent doubly linked lists might be used just as well. For each data structure routines exist for creating, deleting, and modifying it.

3.1. `DomainPoint`

An element of $\mathbb{R}^{n+2}$ will be called a `DomainPoint`,

```
struct DomainPoint  {
                     float  coordinates[n + 2];
                    };
```

3.2. RangePoint

An element of $\mathbb{R}^n$ will be called a `RangePoint`,

```
struct RangePoint   {
                     float   coordinates[n];
                    };
```

3.3. TangentVector

An element of the tangent space of $\mathcal{S}$ will be called a `TangentVector`.

```
struct TangentVector   {
                        float   coordinates[n + 2];
                       };
```

3.4. TangentPlanePoint

A point in the tangent space of the surface will be called a `TangentPlanePoint`. The coordinates of the point are stored, as well as a pointer to the piece of the surface to which the tangent space belongs (a `Disk` is defined below).

```
struct TangentPlanePoint   {
                            float         s;
                            float         t;
                            struct Disk   *disk;
                           };
```

3.5. Disk

A `Disk` is the projection of an elliptical neighborhood U of the origin of the tangent space onto $\mathcal{S}$. (See Figure 2.)

```
struct Disk   {
               struct DomainPoint    *u0;
               float                  rs;
               struct TangentVector  *us;
               float                  rt;
               struct TangentVector  *ut;
               struct Disk           *nextDisk;
               struct Disk           *previousDisk;
               struct Arc            **arcList;
              };
```

The Disk is the set

$$\texttt{Disk} = \left\{ u \mid u = \texttt{u}_0 + s\texttt{u}_\texttt{s} + t\texttt{u}_\texttt{t} + a(s,t),\ u \in \mathcal{S},\ s^2/\texttt{r}_\texttt{s}{}^2 + t^2/\texttt{r}_\texttt{t}{}^2 \le 1 \right\},$$

and the projection of the Disk onto the tangent space of $\mathcal{S}$ at $\texttt{u}_0$, is the elliptical region

$$U = \left\{ u \mid u = \texttt{u}_0 + s\texttt{u}_\texttt{s} + t\texttt{u}_\texttt{t},\ s^2/\texttt{r}_\texttt{s}{}^2 + t^2/\texttt{r}_\texttt{t}{}^2 \le 1 \right\}.$$

The Disk is a part of a doubly linked list of Disk's. The part of the boundary of the Disk which is on the boundary of the computed piece of $\mathcal{S}$ is the union of the arcList. (Arc's are defined below.)

3.6. Arc

The boundary of the computed piece of $\mathcal{S}$ is stored as a list of Arc's,

```
struct Arc   {
              struct Disk  *disk;
              float         θ0;
              float         θ1;
              struct Arc   *nextArc;
              struct Arc   *previousArc;
             };
```

The Arc is the set

$$\texttt{Arc} = \{ u \mid u = \texttt{u}_0 + \texttt{r}_\texttt{s}\cos(\theta)\texttt{u}_\texttt{s} + \texttt{r}_\texttt{t}\sin(\theta)\texttt{u}_\texttt{t} + a(\texttt{r}_\texttt{s}\cos\theta, \texttt{r}_\texttt{t}\sin\theta),\ u \in \mathcal{S},\ \theta \in [\theta_0, \theta_1] \},$$

where the quantites $\texttt{u}_0, \texttt{r}_\texttt{t}$ etc. are the values from the associated Disk disk. The projection of an Arc onto the tangent space of $\mathcal{S}$ is the elliptical arc:

$$\{ u \mid u = \texttt{u}_0 + \texttt{r}_\texttt{s}\cos(\theta)\texttt{u}_\texttt{s} + \texttt{r}_\texttt{t}\sin(\theta)\texttt{u}_\texttt{t},\ \theta \in [\theta_0, \theta_1] \}.$$

3.7. Disk Intersection

One of the basic routines will compute the intersection between two Disk's, and return a list of the points where the boundaries of the two Disk's intersect. A DiskIntersection is the data structure which contains this list

```
struct DiskIntersection  {
                          int           n;
                          struct Disk  *disk1;
                          float        *angleList1;
                          struct Disk  *disk2;
                          float        *angleList2;
                         };
```

`disk1` and `disk2` are the two `Disk`'s whose intersections are represented. n is the total number of intersections, and `*angleList1` and `*angleList2` are lists of the intersections in the terms of the angles on `Arc`'s of the respective `Disk`'s.

3.8. `Surface`

A `Surface` (which is a subset of $\mathcal{S}$), is represented by a list of `Disk`'s, and a list of `Arc`'s. The union of the `Arc`'s is the boundary of the union of the `Disk`'s.

```
struct Surface  {
                 struct Disk  *firstDisk;
                 struct Arc   *firstArc;
                };
```

The data structure representing a surface is illustrated in Figure 3. Initially, no attempt is made to record the connectivity of the boundary of the disks. This information is necessary when constructing an interpolant for the surface, but is not needed to compute the surface. Section 6 descrbes how a triangulation is obtained as the surface is computed.

4. Routines

As for the Data Structures, we describe the basic routines of the algorithm using C syntax. Where this is much longer than the standard mathmatical notation we have substituted the mathematical notation. For example, dot products between `DomainPoint`'s have been written as x^*y.

4.1. Solving the Linear System – `solveLS`

```
struct TangentVector *solveLS(
                                struct DomainPoint    *u0,
                                struct TangentVector  *us,
                                struct TangentVector  *ut,
                                struct RangePoint     *v,
                                float                  σ,
                                float                  τ
                                struct DomainPoint    *wa,
                                int                   *Factor
                              );
```

Given a point $\mathtt{u}_0 \in \mathcal{S}$, a pair of `TangentVector`'s ($\mathtt{u}_s$, $\mathtt{u}_t$) satisfying the assumptions of Lemma 2.1, a `RangePoint` `v`, and two scalars $\sigma \in \mathbb{R}$ and $\tau \in \mathbb{R}$, this routine solves the

non-singular system

$$\begin{aligned} g_u(\mathtt{u}_0)w &= \mathtt{v} \\ \mathtt{u_s}^* w &= \sigma \\ \mathtt{u_t}^* w &= \tau \end{aligned}$$

and returns a pointer to w. An approximate solution $\mathtt{w_a}$ is provided for iterative solvers.

For each particular problem there is a best choice for solving this system. We make no assumption about which method is used, rather consider it to be supplied by the implementation. By default the LINPACK routines DGEF and DGES might be used for small systems. The flag `Factor` is used to save computation. If `Factor=FACTOR`, then an LU decomposition must be done. Otherwise, if `Factor=NO_FACTOR` the previous factorization may be re-used, and only a backsolve performed.

4.2. Solving the Non-Linear System – `solveNLS`

```
struct DomainPoint *solveNLS(
                        struct DomainPoint    *u0,
                        struct TangentVector  *us,
                        struct TangentVector  *ut,
                        struct RangePoint     *v,
                        float                  σ,
                        float                  τ
                        struct DomainPoint    *wa,
                    );
```

This routine solves the system

$$\begin{aligned} g(w) &= v \\ \mathtt{u_s}^* w &= \sigma \\ \mathtt{u_t}^* w &= \tau \end{aligned}$$

and returns a pointer to w. An approximate solution $\mathtt{w_a}$ is provided for iterative solvers. One possible implementation is to use Newton's method, which relies on the `solveLS` routine.

4.3. Creating a `Disk`– `createDisk`

```
struct Disk *createDisk(
                        struct DomainPoint    *u0,

                        struct TangentVector  *usa,

                        struct TangentVector  *uta,
                    );
```

Given a point $\mathtt{u}_0 \in \mathcal{S}$, and the basis $\mathtt{u}_\mathtt{s}^\mathtt{a}$ and $\mathtt{u}_\mathtt{t}^\mathtt{a}$ of a nearby `Disk`'s tangent space, this routine computes the quantities stored in the `Disk` data structure and returns a pointer to the newly created `Disk`.

First a basis for the tangent space is computed

$$
\begin{aligned}
w &= \mathtt{solveLS}(\mathtt{u}_0, \mathtt{u}_\mathtt{s}^\mathtt{a}, \mathtt{u}_\mathtt{t}^\mathtt{a}, \mathtt{0}, \mathtt{1.}, \mathtt{0.}, \mathtt{FACTOR});\\
u_s &= w/|w|\\
w &= \mathtt{solveLS}(\mathtt{u}_0, u_s, \mathtt{u}_\mathtt{t}^\mathtt{a}, \mathtt{0}, \mathtt{0.}, \mathtt{1.}, \mathtt{NO\_FACTOR});\\
u_t &= w/|w|
\end{aligned}
$$

Once a basis is known the radii can be computed ($a_s s$ ans $a_t t$ are computed using an additional call to `solveLS`).

$$r_s = \min(\rho(\mathtt{u}_0), \sqrt{\tfrac{2\epsilon}{|a_{ss}|}})$$

$$r_t = \min(\rho(\mathtt{u}_0), \sqrt{\tfrac{2\epsilon}{|a_{tt}|}})$$

In a practical implementation, the point $\mathtt{u}_0$ is not assumed to lie exactly on the surface. It is first projected onto the surface using the approximate tangents and solveNLS.

4.4. Computing a Point on a `Disk`– `pointOnDisk`

```
struct DomainPoint *pointOnDisk(
                                    struct TangentPlanePoint  *tangentPt;
                                );
```

This routine computes the point u on $\mathcal{S}$ corresponding to the `TangentPlanePoint` by solving the equations

$$
\begin{aligned}
g(u) &= 0\\
\mathtt{u}_\mathtt{s}{}^*(u - \mathtt{u}_0) &= s\\
\mathtt{u}_\mathtt{t}{}^*(u - \mathtt{u}_0) &= t
\end{aligned}
$$

In other words

$$
\begin{aligned}
disk &= (*\mathtt{tangentPt}).\mathtt{disk};\\
s &= (*\mathtt{tangentPt}).\mathtt{s};\\
t &= (*\mathtt{tangentPt}).\mathtt{t};\\
u_0 &= (*\mathtt{disk}).\mathtt{u}_0;\\
u_s &= (*\mathtt{disk}).\mathtt{u}_\mathtt{s};\\
u_t &= (*\mathtt{disk}).\mathtt{u}_\mathtt{t};\\
\sigma &= s + u_s^* u_0;\\
\tau &= t + u_t^* u_0;\\
\mathtt{w}_\mathtt{a} &= u_0 + s u_s + t u_t;\\
u &= \mathtt{solveNLS}(u_0, u_s, u_t, 0, \sigma, \tau, \mathtt{w}_\mathtt{a});
\end{aligned}
$$

4.5. Projecting points from $\mathcal{S}$ onto the tangent space – `projectPoint`

```
struct TangentPlanePoint *projectPoint(
                                        struct DomainPoint  *u,
                                        struct Disk         *disk
                                      );
```

The projection of u, (s,t) is given by

$$\begin{pmatrix} s \\ t \end{pmatrix} = \begin{pmatrix} 1 & \mathtt{u_s}^*\mathtt{u_t} \\ \mathtt{u_t}^*\mathtt{u_s} & 1 \end{pmatrix}^{-1} \begin{pmatrix} \mathtt{u_s}^*(u-\mathtt{u}_0) \\ \mathtt{u_t}^*(u-\mathtt{u}_0) \end{pmatrix}$$

where $\mathtt{u}_0$, $\mathtt{u_s}$ and $\mathtt{u_t}$ are the quantities associated with `disk`. The TangentPlanePoint which is returned has coordinates (s,t), and a pointer to `disk`.

4.6. Intersecting two `Disk`'s – `intersectDisks`

```
struct DiskIntersection *intersectDisks(
                                           struct Disk  *disk1,
                                           struct Disk  *disk2
                                        );
```

This routine takes two `Disk`'s and returns a list of the intersections of thier boundaries.

Let

$$\begin{array}{ll} \mathtt{u}_0 = (*\mathtt{disk1}).\mathtt{u}_0 & \mathtt{v}_0 = (*\mathtt{disk2}).\mathtt{u}_0 \\ \mathtt{u_s} = (*\mathtt{disk1}).\mathtt{u_s} & \mathtt{v}_\sigma = (*\mathtt{disk2}).\mathtt{u_s} \\ \mathtt{u_t} = (*\mathtt{disk1}).\mathtt{u_t} & \mathtt{v}_\tau = (*\mathtt{disk2}).\mathtt{u_t} \\ \mathtt{r_s} = (*\mathtt{disk1}).\mathtt{r_s} & \mathtt{r}_\sigma = (*\mathtt{disk2}).\mathtt{r_s} \\ \mathtt{r_t} = (*\mathtt{disk1}).\mathtt{r_t} & \mathtt{r}_\tau = (*\mathtt{disk2}).\mathtt{r_t} \end{array}$$

The points u on $\mathcal{S}$ which are on the boundaries of both of the `Disk`'s satisfy the system

$$\begin{aligned} g(u) &= 0 \\ u_s^*(u-u_0) &= r_s\cos\theta_0 \\ u_t^*(u-u_0) &= r_t\sin\theta_0 \\ v_\sigma^*(u-v_0) &= r_\sigma\cos\theta_1 \\ v_\tau^*(u-v_0) &= r_\tau\sin\theta_1 \end{aligned}$$

Using the bases for the tangent spaces, and projecting onto the surface with the mappings $a(s,t)$ and $\alpha(\sigma,\tau)$, this is equivalent to solving the system

$$\begin{aligned} \mathtt{u_s}^*(\mathtt{u}_0 + s\mathtt{u_s} + t\mathtt{u_t} + a(s,t)) &= \mathtt{u_s}^*(\mathtt{v}_0 + \sigma\mathtt{v}_\sigma + \tau\mathtt{v}_\tau + \alpha(\sigma,\tau)) \\ \mathtt{u_t}^*(\mathtt{u}_0 + s\mathtt{u_s} + t\mathtt{u_t} + a(s,t)) &= \mathtt{u_t}^*(\mathtt{v}_0 + \sigma\mathtt{v}_\sigma + \tau\mathtt{v}_\tau + \alpha(\sigma,\tau)) \\ (s/\mathtt{r_s})^2 + (t/\mathtt{r_t})^2 &= 1 \\ (\sigma/\mathtt{r}_\sigma)^2 + (\tau/\mathtt{r}_\tau)^2 &= 1 \end{aligned}$$

with

$$\begin{aligned} s &= r_s \cos\theta_0 \\ t &= r_t \sin\theta_0 \\ \sigma &= r_\sigma \cos\theta_1 \\ \tau &= r_\tau \sin\theta_1. \end{aligned}$$

The variables s and t may be eliminated:

$$\begin{pmatrix} s \\ t \end{pmatrix} = A \begin{pmatrix} \sigma \\ \tau \end{pmatrix} + b$$

where,

$$A = \begin{pmatrix} 1 & \mathtt{u_s}^*\mathtt{u_t} \\ \mathtt{u_s}^*\mathtt{u_t} & 1 \end{pmatrix}^{-1} \begin{pmatrix} \mathtt{u_s}^*\mathtt{v}_\sigma & \mathtt{u_s}^*\mathtt{v}_\tau \\ \mathtt{u_t}^*\mathtt{v}_\sigma & \mathtt{u_t}^*\mathtt{v}_\tau \end{pmatrix}$$

$$b = \begin{pmatrix} 1 & \mathtt{u_s}^*\mathtt{u_t} \\ \mathtt{u_s}^*\mathtt{u_t} & 1 \end{pmatrix}^{-1} \begin{pmatrix} \mathtt{u_s}^*(\mathtt{v}_0 - \mathtt{u}_0) + \mathtt{u_s}^*\alpha(\sigma,\tau) \\ \mathtt{u_t}^*(\mathtt{v}_0 - \mathtt{u}_0) + \mathtt{u_t}^*\alpha(\sigma,\tau) \end{pmatrix}$$

Let

$$R_{st} = \begin{pmatrix} 1/r_s^2 & 0 \\ 0 & 1/r_t^2 \end{pmatrix}$$

$$R_{\sigma\tau} = \begin{pmatrix} 1/r_\sigma^2 & 0 \\ 0 & 1/r_\tau^2 \end{pmatrix}$$

Substituting,

$$\begin{pmatrix} \sigma & \tau \end{pmatrix} A^* R_{st} A \begin{pmatrix} \sigma \\ \tau \end{pmatrix} + 2b^* R_{st} A \begin{pmatrix} \sigma \\ \tau \end{pmatrix} + b^* R_{st} b = 1$$

$$\begin{pmatrix} \sigma & \tau \end{pmatrix} R_{\sigma\tau} \begin{pmatrix} \sigma \\ \tau \end{pmatrix} = 1$$

The terms $u_s^*\alpha(\sigma,\tau)$ and $u_t^*\alpha(\sigma,\tau)$ are $O(\epsilon^{3/2})$, so, for an approximate solution, may be neglected (section 1). The remaining, reduced equations are a pair of two quadratic equations in two variables. One of these variables may be eliminated leaving a quartic. For example, for the system

$$\begin{aligned} a_1\sigma^2 + b_1(\tau)\sigma + c_1(\tau,\tau^2) &= 0 \\ a_2\sigma^2 + b_2(\tau)\sigma + c_2(\tau,\tau^2) &= 0 \end{aligned}$$

reduces to the quartic in τ

$$(b_1c_2 - c_1b_2)(b_1a_2 - a_1b_2) + (a_1c_2 - c_1a_2)^2 = 0$$

with

$$\sigma = \frac{a_1 c_2 - a_2 c_1}{a_2 b_1 - a_1 b_2}$$

For each solution (σ, τ) we have

$$\theta_0 = \arctan(t/s)$$
$$\theta_1 = \arctan(\tau/\sigma)$$

If more accuracy is needed these solutions provide an initial guess for an iteration.

4.7. Testing if a point is contained in a `Disk` – `insideOf`

```
Boolean *insideOf(
                    struct DomainPoint  *u,
                    struct Disk         *disk
                  );
```

This routine tests the `DomainPoint` u and returns `TRUE` if it lies inside the `Disk`. The test is straightforward. First the `DomainPoint` is projected onto the tangent space of the `Disk` (using `projectPoint`). Then, if the resulting coordinates (s, t) satisfy

$$\left(\frac{s}{r_s}\right)^2 + \left(\frac{t}{r_t}\right)^2 \leq 1$$

the test returns `TRUE`.

4.8. splitting an `Arc` – `splitArc`

```
struct Arc **splitArc(
                        struct Arc               *arc,
                        struct DiskIntersection  *diskIntersection
                      );
```

This routine takes an `Arc` and the intersection of the `Disk` associated with the `Arc` and some other `Disk`, and returns a list of `Arc`'s whose union is the original `Arc`, and no element of the list crosses an intersection. (See Figure 4.)

The implementation of this routine is somewhat messy, so we will not give details. The chief detail to note is that the original `Arc` is not deleted, and that the `Arc`'s which are returned are linked through the data structure, but are not added to the `Surface`.

4.9. Removing Pieces of Arc's underneath a Disk – removeArcsUnderDisk

```
removeArcsUnderDisk(
                        struct Disk              *disk1,
                        struct Disk              *disk2,
                        struct DiskIntersection  *DI
                    );
```

The routine takes two disks and their intersections, and splits all the Arc's on both Disk's. The pieces which lie inside the Disk's are then removed from the lists. (See Figure 5.)

4.10. Adding a Disk to Surface – addDiskToSurface

```
struct Arc *addDiskToSurface(
                                 struct Disk     *disk,
                                 struct Surface  *surface
                             );
```

This routine appends a Disk to the list of Disk's and Arc's of a Surface.

4.11. Getting the end point of an Arc on the surface – getArcEndPoint

```
struct DomainPoint *getArcEndPoint(
                                 struct Surface        *surface
                                 struct TangentVector  **u_s
                                 struct TangentVector  **u_t
                             );
```

This routine walks down the list of Arc's of a Surface and returns the endpoint of the first Arc it finds. Since the Surface may not be finite, a box is specified, and the first endpoint Arc not outside this box is used. The pointers to the TangentVectors are set to the tangents of the Disk to which the Arc belongs.

4.12. Merging a new Disk with the surface – addDisk

```
addDisk(
            struct Disk     *disk,
            struct Surface  *surface
        );
```

This is the main step of the continuation. A newly computed `Disk` and its boundary are merged with an existing union of `Disk`'s and their boundary. This is simply a matter of removing pieces of the boundaries which are interior to the union, and adding the `Disk` to the list.

The process is

```
1. Create an arc list for the new disk
     createWholeDiskBoundary(newDisk);
2. Loop over all disk's on the surface
     disk = (*surface).firstDisk;
     while(disk != NULL)
      {
       if((DI = intersectDisk(disk,newDisk)) != NULL)
        {
3. Split the arcs and remove underlying pieces
         removeArcsUnderDisk(newDisk,disk,DI);
         freeDiskIntersection(DI);
        };
        disk = disk.nextDisk;
      };
5. Add the new disk and it's arc's to the surface
     addDiskToSurface(newDisk,Surface);
```

This is a simplified algorithm for clarity. As is discussed below, the checking of each pair of `Disk`'s is expensive, and can be avoided.

5. The Algorithm for Computing a `Surface`

With the above data structures and routines the algorithm is easy to state:

```
1. Obtain:
     an initial point on the surface u_0.
     approximate tangent vectors us and ut.
     a box in which the surface is to be computed.

2. Compute the first Disk and its Boundary Arc
       disk = createDisk(u_0,us,ut);

3. Create a surface with the initial Disk
       surface = createSurface(disk);

   Do until there are no more eligable points on the boundary:
```

```
4.   Get a point on the boundary
       while( (newPoint=getArcEndPoint(surface),&us,&ut) != NULL)
        {

5.   Create a Disk for this new point and its boundary Arc.
         newDisk=createDisk(newPoint,us,ut);

6.   Merge the old surface with the new Disk and boundary.
         addDisk(newDisk,surface);
        };
```

6. Implementation

There are two main implementational issues. The first is reducing the work involved in intersecting all pairs of `Disk`'s. The second is obtaining a triangulation of the surface from the list of `Disk`'s and `Arc`'s.

6.1. Intersecting pairs of `Disk`'s

The complexity of updating the boundary of the `Disk`'s can be reduced by creating a modified quadtree data structure for the `Disk`'s. The quadtree provides a fast way of getting a list of the `Disk`'s which intersect a given region (the `Disk` being added). The standard quadtree [13] is used for points in a plane. The modified quadtree stores the projection of the centers of the `Disk`'s and a bounding box, which contains all of the `Disk`'s underneath the node. In this way, large collections of `Disk`'s can be eliminated as being outside the region of interest.

The algorithm as stated is $O(N^2)$, where N is the total number of `Disk`'s computed (for each of the N `Disk`'s $N-1$ other `Disk`'s must be checked). The modified quadtree reduces the complexity of checking the old `Disk`'s to $\log N$, and the total complexity to $N \log N$.

The implementation of the modified quadtree is straightforward. It is created when a surface is created, and when a `Disk` is added to the surface, it is also added to the modified quadtree. Since `Disk`'s are not deleted, the only other quadtree routine necessary returns a list of the `Disk`'s which overlap a region of the plane. In principle this procedure could be extended to higher dimensional spaces, for example an octree and a projection into 3 space.

6.2. Triangulating the surface

The second refinement of the algorithm involves computing a triangulation of the surface. This is obtained by adding a list of `Disk`'s to the `Disk` data structure. When `Disk`'s are intersected against a newly created `Disk` in the routine `addDisk`, this list is created for the new `Disk`. When a `Disk` intersects the new `Disk` it is added to the list. The list is then

sorted, by projecting the vectors between centers onto the tangent space and sorting so that the `Disk`'s are in counterclockwise order. (Figure 6.)

7. Examples

As examples we compute a sphere and a torus (Figures 7 and 8). These cause difficulty for local continuation algorithms because they are finite and have no edges. The continuation must therefore avoid travelling repeatedly around the surface. The algorithm deletes `Arc`'s where the surface overlaps itself, so eventually there are no `Arc`'s left and the algorithm stops.

8. Singular Surfaces

The algorithm as stated assumes that the surface is regular at all points. In practice the only restriction is that no center of a disk generated by the continuation be a singular point. If this happens the tangent space is not uniquely defined, and so no `Disk` can be created. One approach to desingularizing a surface is to "lift" the surface by considering the tangent space to be part of the surface. Instead of $u \in \mathcal{S}$, the lifted points are (u, u_s, u_t), and the defining equation of the lifted surface is

$$\begin{aligned} g(u) &= 0 \\ g_u(u)u_s &= 0 \\ g_u(u)u_t &= 0 \\ u_s.u_t &= 0 \\ u_s.u_s &= 1 \\ u_t.u_t &= 1 \end{aligned}$$

A singular surface whose only singularities are where regular sheets intersect transversally will therefore become regular. Although this sounds like a major project, the lifting actually only has to be done when two disks are intersected. If the dot product between the tangents of the two `Disk`'s (after rotating to make the bases conform) are more than ϵ away from the values for constant tangent space, the `Disk`'s are declared to be non-intersecting, even though the projections of the boundaries onto one tangent space or the other intersect.

9. Conclusion

We have described an algorithm for computing implicitly defined surfaces which is similar in spirit to analytic continuation, but which does not have the practical limititions of that algorithm. Step size control is implemented as a natural part of the algorithm, and the algorithm is global, so that surfaces such as spheres and cylinders can be computed with a minimum of coverage.

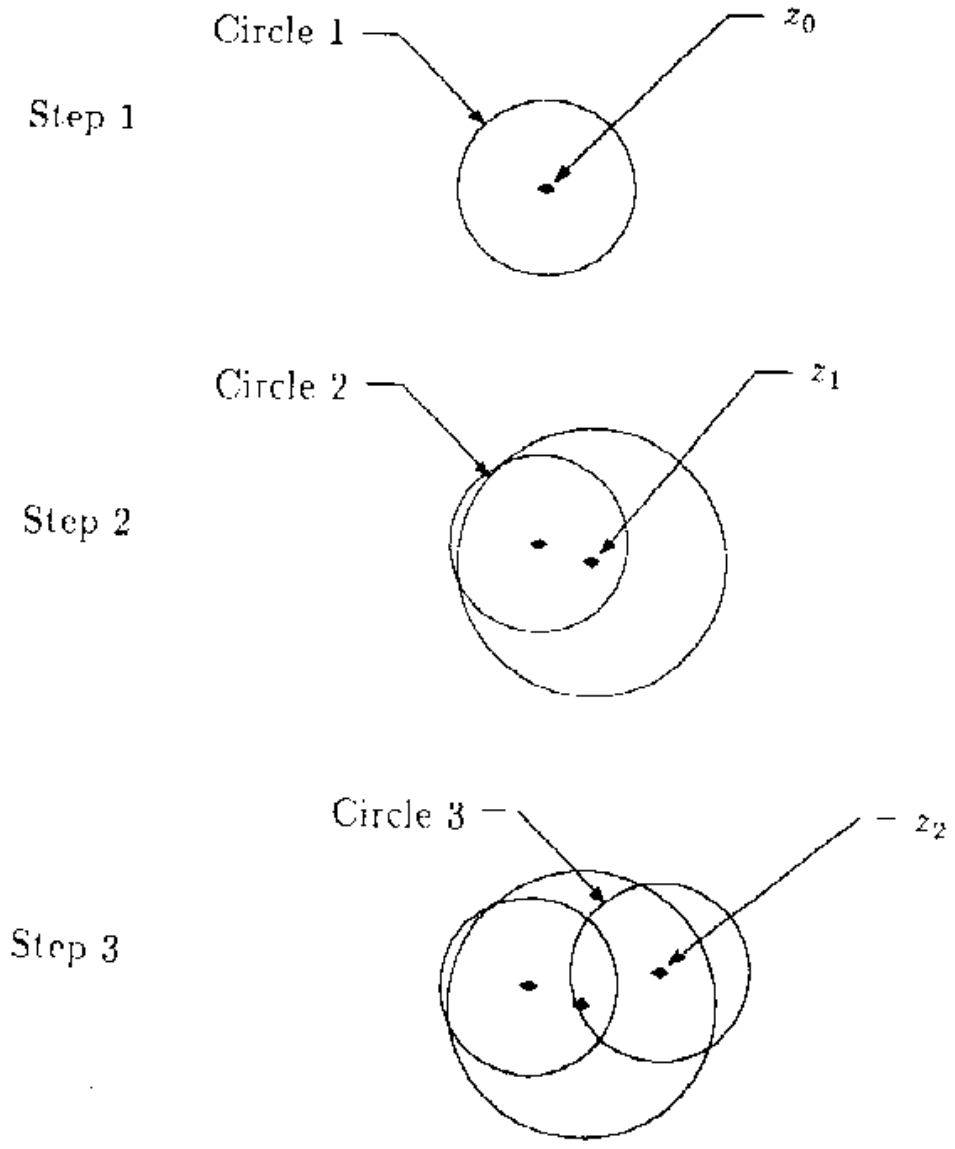

Figure 1: Analytic Continuation.

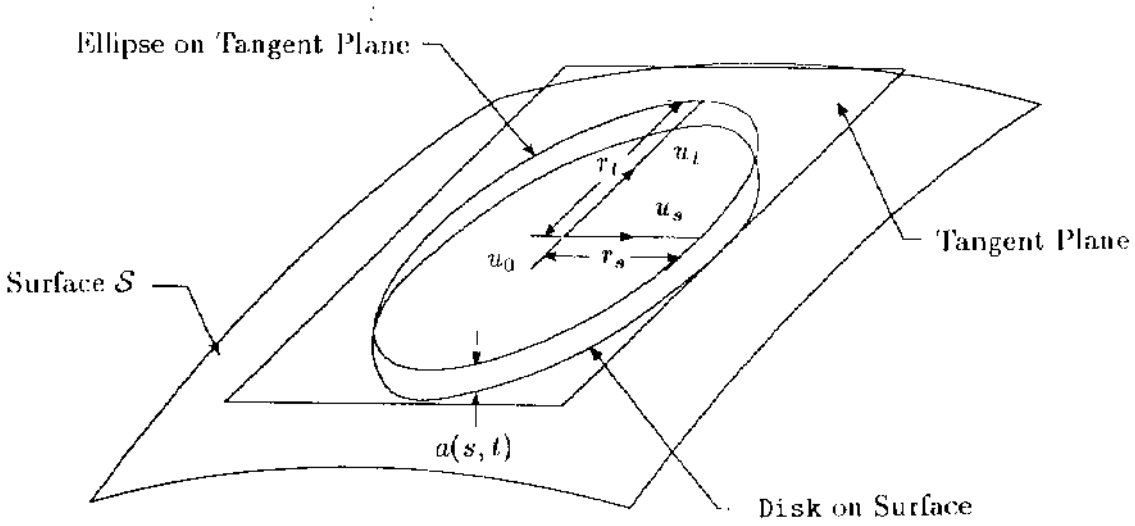

Figure 2: The Disk Data Structure.

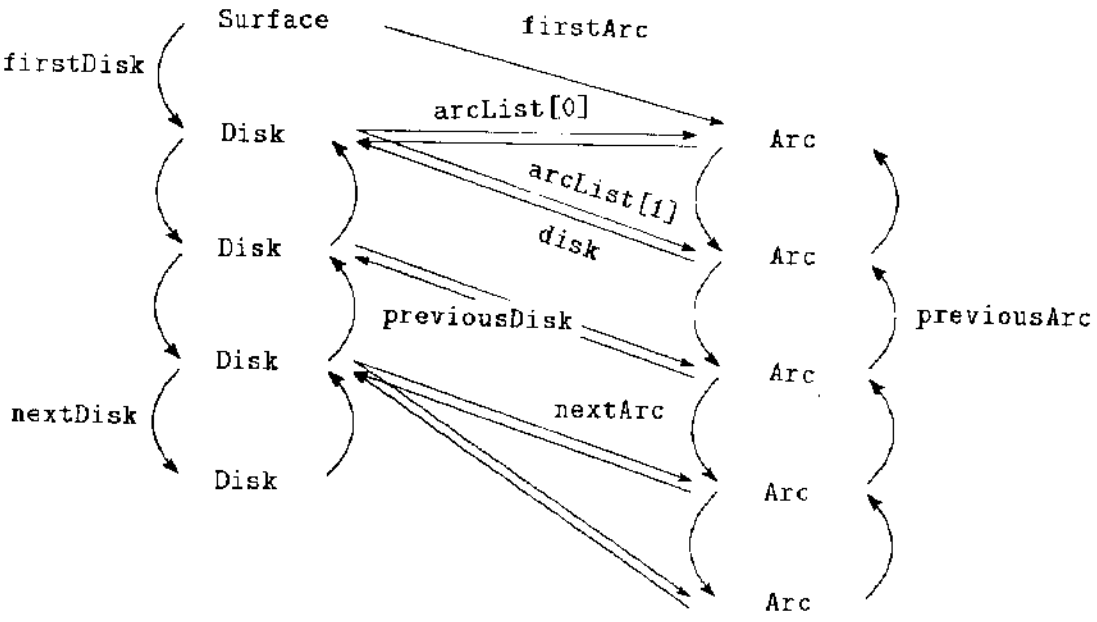

Figure 3: The Surface Data Structure.

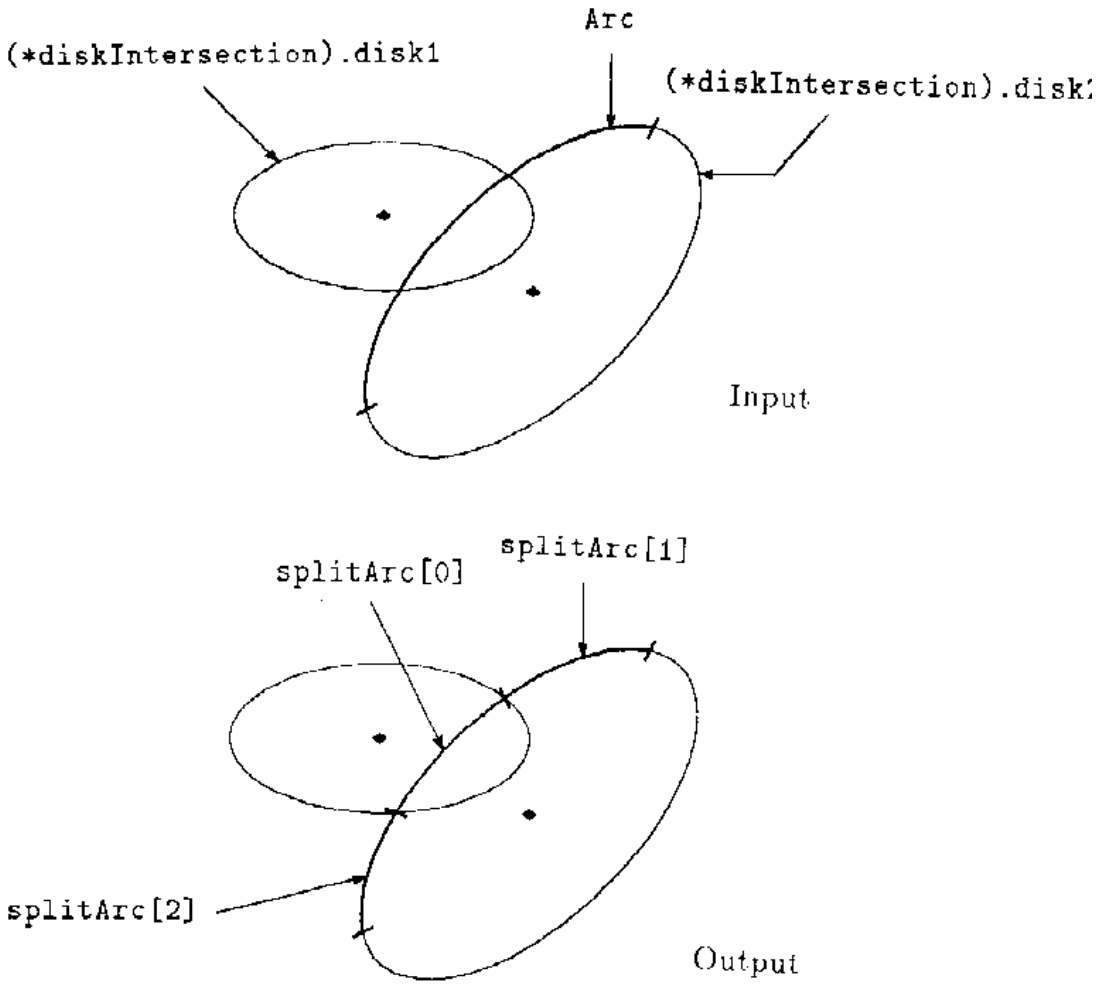

Figure 4: SplitArc

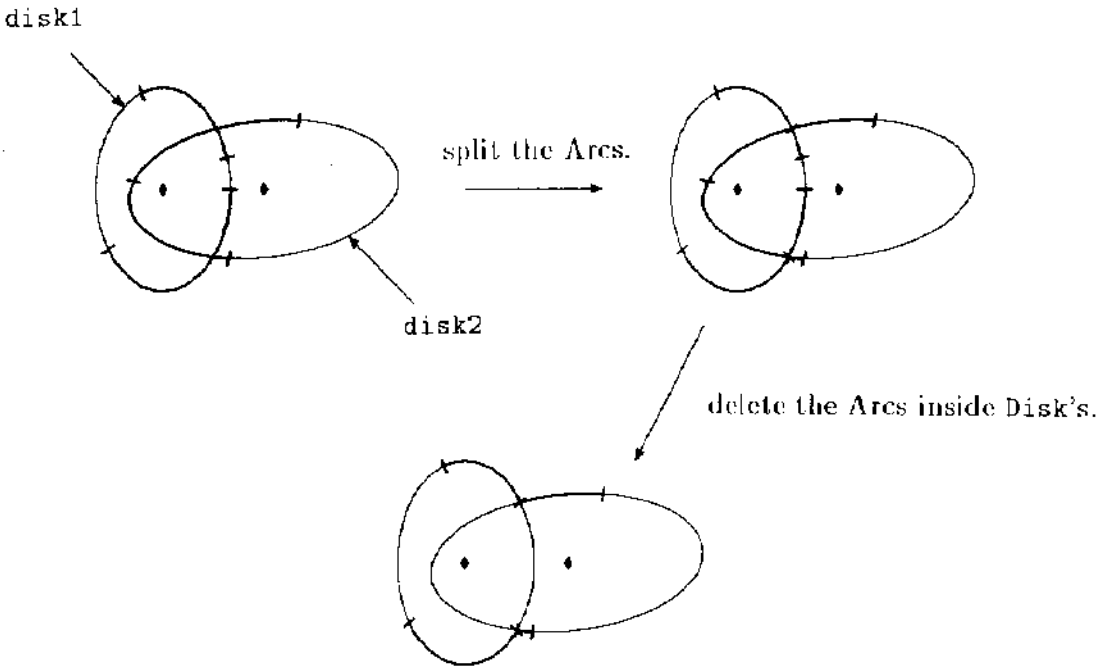

Figure 5: `removeArcsUnderDisk`

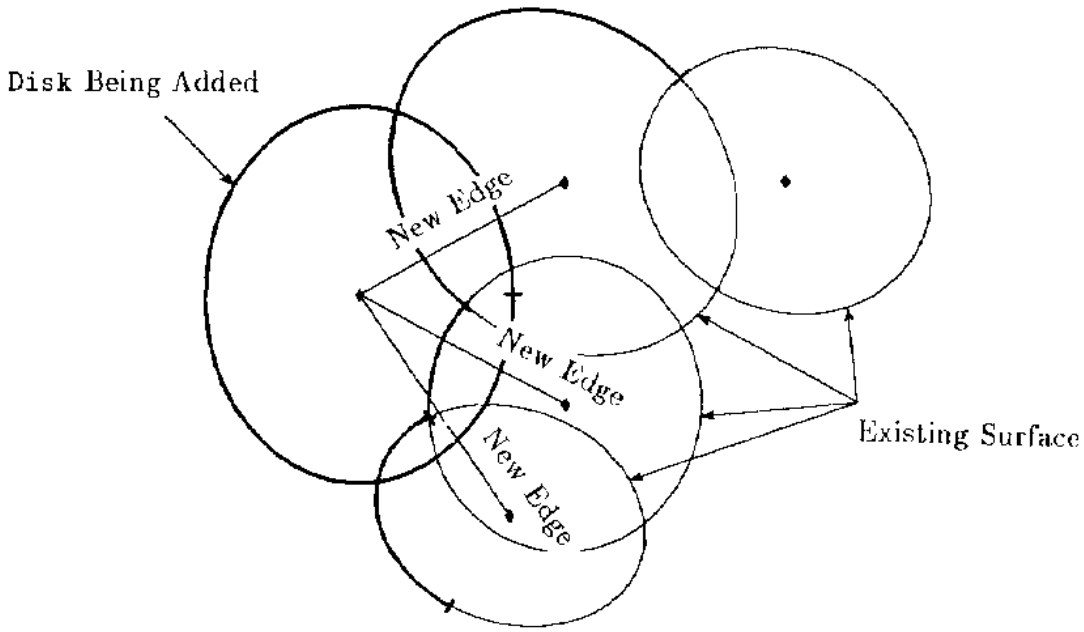

Figure 6: Incrementally Triangulating the Surface.

References

[1] E. L. Allgower and K. Georg, *Simplicial and continuation methods for approximations, fixed points and solutions to systems of equations*, SIAM Review, 22 (1980), pp. 28–85.

[2] E. L. Allgower and P. H. Schmidt, *An algorithm for piecewise-linear approximation of an implicitly defined manifold*, SIAM J. Numer. Anal., 22 (1985), pp. 322–346.

[3] R. E. Bank, *PLTMG: A Software Package for Solving Elliptic Partial Differential Equations*, Frontiers in Applied Mathematics, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1990.

[4] J. Bloomenthal, *Polygonization of implicit surfaces*, Computer Aided Geometric Design, 5 (1988), pp. 341–355.

[5] G. F. Carrier, M. Krook, and C. E. Pearson, *Functions of a Complex Variable, Theory and Practice*, McGraw-Hill, New York, 1966.

[6] C. den Heijer and W. C. Rheinboldt, *On steplength algorithms for a class of continuation methods*, SIAM J. Numer. Anal., 18 (1981), pp. 925–948.

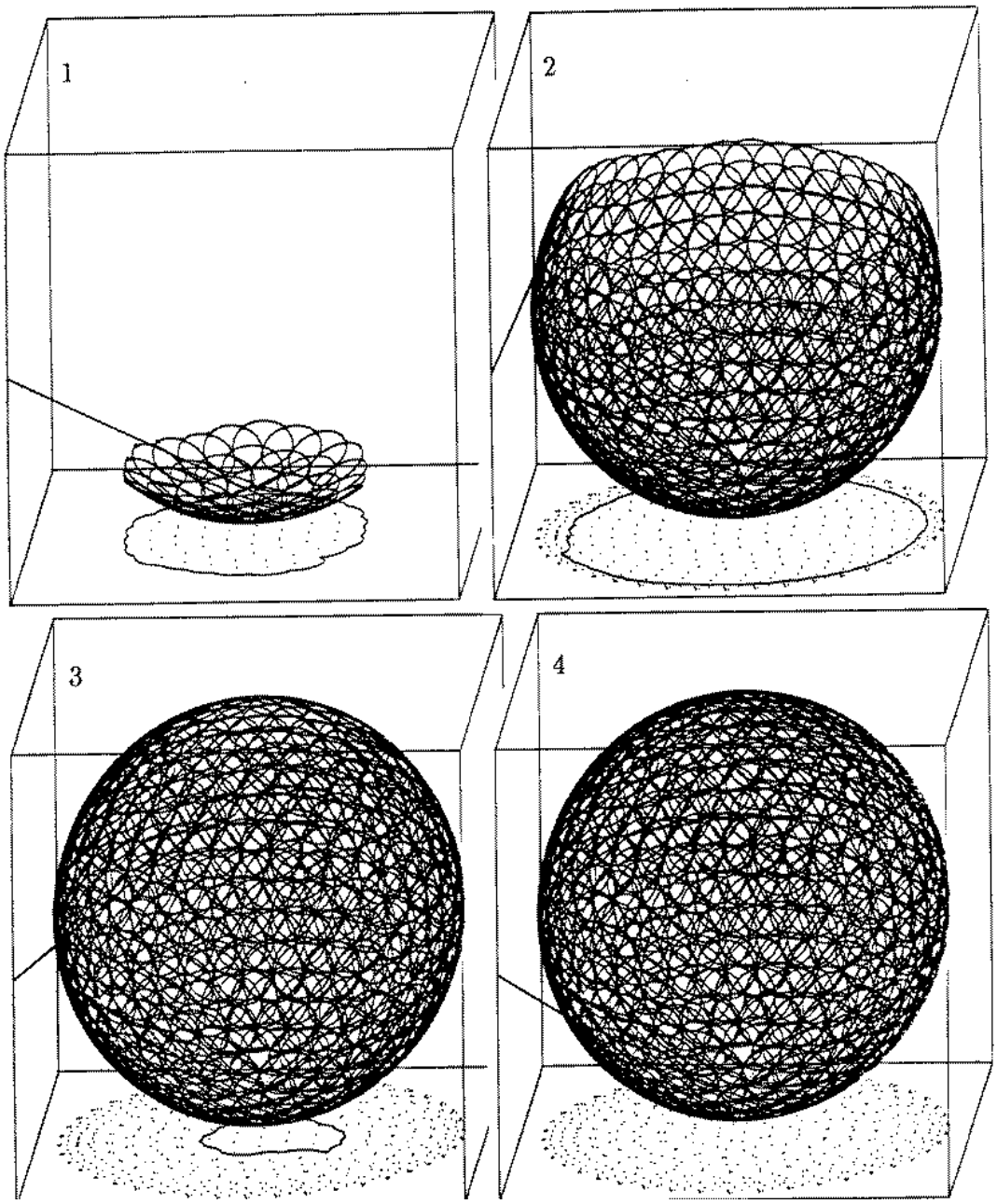

Figure 7: Continuation of a sphere. The centers of the `Disk`'s are shown projected onto the bottom of the cube, along with the projection of the boundary `Arc`'s.

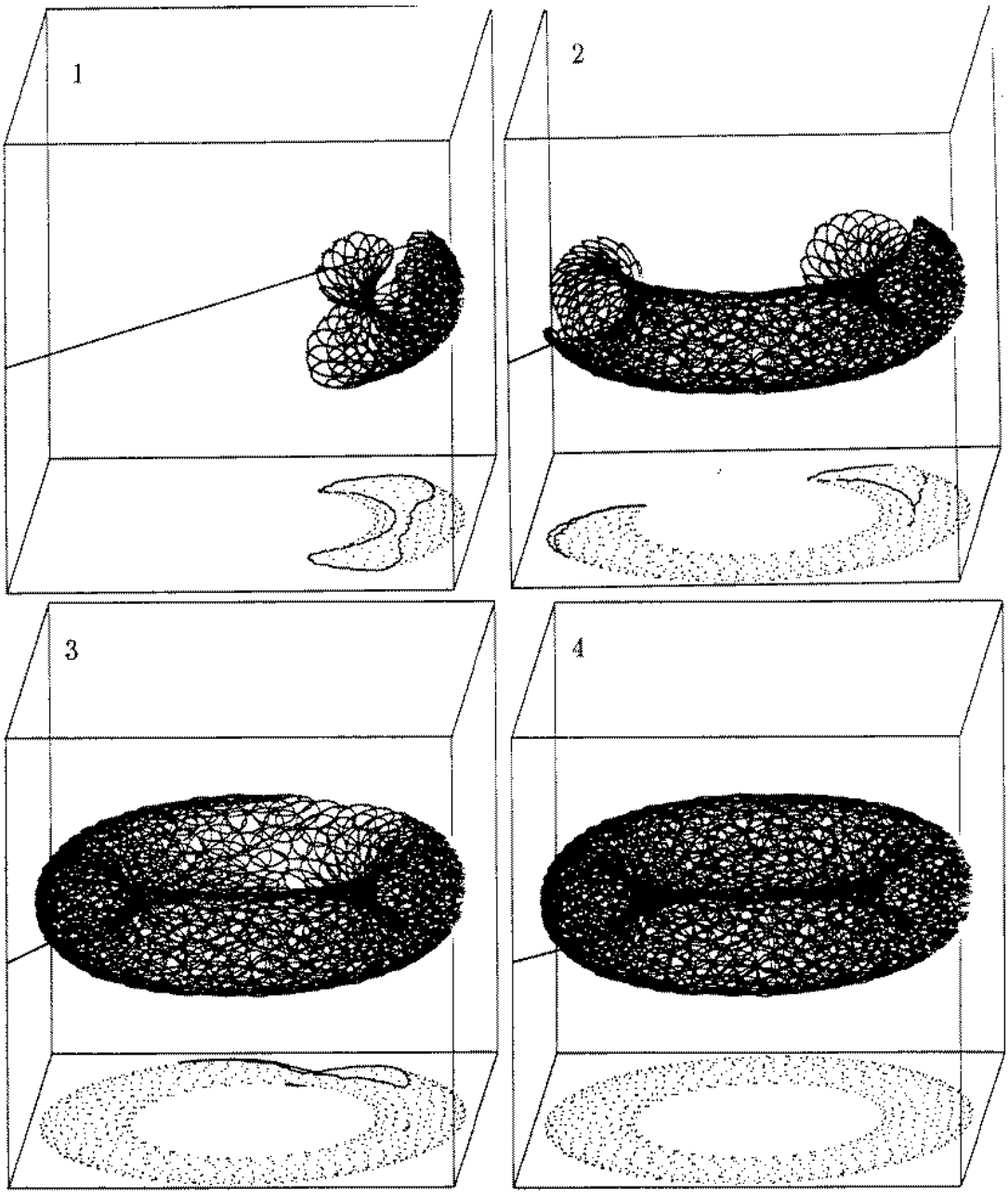

Figure 8: Continuation of a torus. The centers of the `Disk`'s are shown projected onto the bottom of the cube, along with the projection of the boundary `Arc`'s.

[7] E. Doedel and J. Kernévez, *Auto: Software for continuation and bifurcation problems in ordinary differential equations*, applied mathematics report, California Institute of Technology, 1986.

[8] M. Hall and J. Warren, *Adaptive polygonalization of implicitly defined surfaces*, IEEE Computer Graphics and Applications, (1990), pp. 33–42.

[9] H. B. Keller, *Numerical solutions of bifurcation and nonlinear eigenvalue problems*, in Applications of Bifurcation Theory, P. Rabinowitz, ed., New York, 1977, Academic Press, pp. 359–384.

[10] H. B. Keller, *Lectures on Numerical Methods in Bifurcation Theory*, Tata Institute and Springer-Verlag, Berlin, 1987.

[11] W. Lorensen and H. Cline, *Marching cubes: a high resolution 3D surface construction algorithm*, Computer Graphics, 21 (1987), pp. 163–170.

[12] A. Morgan, *Solving polynomial systems using continuation for engineering and scientific problems*, Prentice-Hall, Englewood Cliffs, N.J., 1987.

[13] F. P. Preparata and M. I. Shamos, *Computational Geometry*, Springer-Verlag, New York, 1985.

[14] W. C. Rheinboldt, *On the computation of multi-dimensional solution manifolds of parameterized equations*, Numer. Math., 53 (?), pp. 165–181.

[15] ——, *On a moving-frame algorithm and the triangulation of equilibrium manifolds*, in ISNM79: Bifurcation: Analysis, Algorithms, Applications, T. Kupper, R. Seydel, and H. Troger, eds., Basel, 1987, The University of Dortmund, Birkhäuser Verlag, pp. 256–267.

[16] W. C. Rheinboldt and J. V. Burkardt, *A program for a locally parameterized continuation process*, ACM Trans. Math. Software, 17 (1983), pp. 215–235.

[17] H. Schwetlick, *On the choice of steplength in path following methods*, Z. Angew. Math. u. Mech., 9 (1984), pp. 391–396.