Cornell CS 322 Introduction to Scientific Computing *out: Wed Feb 6, 2008*
Project #1: Rendering Implicit Surfaces using Root Finding *due: Mon Feb 25, 2008*
By Jeffrey Chadwick (TA) and Prof. Doug James

1 Introduction

The task of finding roots of a proverbial $f(x)$ can arise in so very many ways. In this assignment you will apply the root finding methods learned in class (Bisection, Newton, Secant, etc.) to the very tangible task of rendering images of three-dimensional objects represented by implicit surfaces. Although the techniques presented in class may have appeared simple, you will see that, in practice, a fair amount of care is required to ensure that the methods are used effectively.

To find roots efficiently and robustly, it is important to understand and exploit the structure of the function at hand. In this problem, you will want to find the left-most "first root," with a typical function $f(t)$ shown in Figure 1. The small red circles below the plot correspond to "hints" (particle locations) that we will provide you to aid in the root finding process (more details on this below). To render a large image, you may need to perform millions of such root finding problems, each with a different $f(t)$.

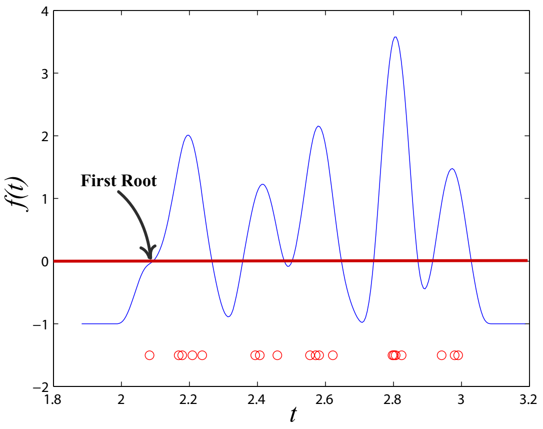


Figure 1: *Plot of a "typical" $f(t)=0$ problem.*

We now provide a little background on implicit surfaces (§1.1), and how the $f(t)$ functions arise during ray tracing (§1.3), before going on to describe exactly what you need to do in this assignment.

1.1 Implicit Surfaces

An implicit surface (sometimes referred to as an "isosurface") in $\mathbb{R}^3$ is a set of points S defined as follows:

$$S = \left\{(x, y, z) \in \mathbb{R}^3 : f(x, y, z) = 0\right\},$$

where f is some function. That is, S is the set of all points (x, y, z) at which f evaluates to 0. Various well-known surfaces can be defined in this way. For example, a sphere of radius R, centred at the origin, is defined implicitly by

$$x^2 + y^2 + z^2 - R^2 = 0.$$

In this case, $f(x, y, z) = x^2 + y^2 + z^2 - R^2$. One can determine whether a point (x, y, z) lies on the surface by simply evaluating f at that point.

1.2 Metaball Surfaces

This assignment will deal with a particular type of implicit surface known as a "metaball surface" or "blobby surface". These surfaces consist of a collection of "metaballs". On its own, a metaball can only take the form of a simple sphere, but together they can merge to form complex shapes. Each metaball is characterized by its position (x_0, y_0, z_0) in $\mathbb{R}^3$ and a positive "support radius," $r_{support}$. The support (or bounding) radius is essentially the maximum extent of the metaball's possible interaction with other metaballs. Each metaball has a "weight" function that establishes its value at any point in space. For this assignment, we will use the weight function

$$w(x,y,z) = \begin{cases} \left(1 - \left(\frac{r}{r_{support}}\right)^2\right)^3 & \text{if } r \le r_{support} \\ 0 & \text{otherwise} \end{cases}$$

where r is the Euclidean radius,

$$r = \sqrt{(x-x_0)^2 + (y-y_0)^2 + (z-z_0)^2}.$$

A graph of this function (for $r_{support} = 1$) is shown in Figure 2.

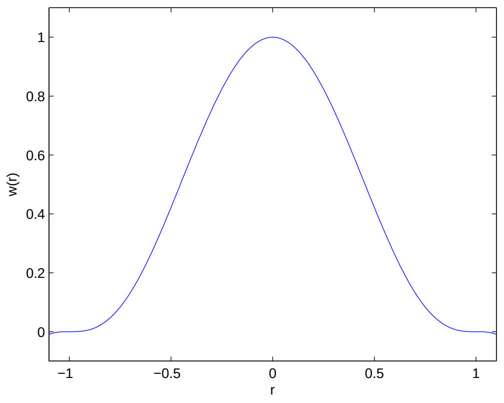


Figure 2: *A plot of the function $w(r)$ for $r_{support} = 1$*

If we have multiple metaballs defined at the points

$$(x_1, y_1, z_1), (x_2, y_2, z_2) \ldots (x_n, y_n, z_n)$$

then we refer to the weighting function for metaball i as $w_i(x,y,z)$. The total weight for the metaball field at any point in space is the sum of all metaball weights,

$$W(x,y,z) = \sum_{i=1}^{n} w_i(x,y,z).$$

We can then use W to define an implicit surface by choosing an isosurface, or "level set," with level ℓ,

$$f(x,y,z) = W(x,y,z) - \ell = 0.$$

Summing metaball weights to form $W(x,y,z)$ causes isosurfaces to merge together smoothly as metaballs approach each other (see Figure 3).

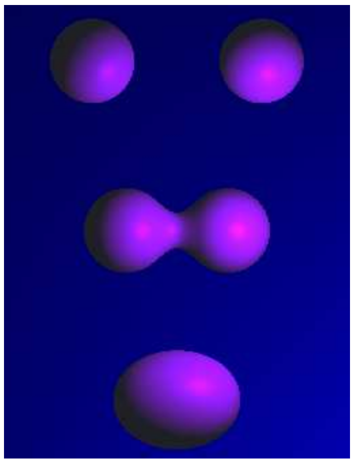


Figure 3: *Two metaballs merging*

1.3 Ray Tracing

Ray tracing is a commonly used technique for rendering images in computer graphics. Roughly speaking, this process involves setting up a virtual camera in a scene which may include any number of objects to be displayed. Rays are cast from the camera, and intersections are computed between the rays and objects in the scene. Computing these intersections involves root finding. These intersections are then shaded to provide a rendering of the scene from the viewpoint of the virtual camera. An example of the kind of ray-traced image you can produce upon completion of this assignment is shown in Figure 4.

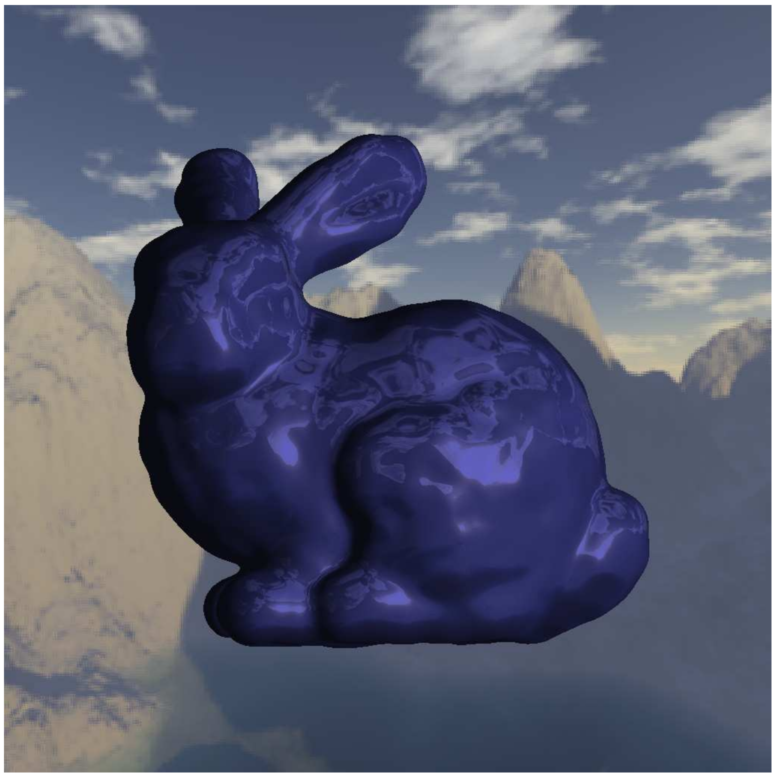


Figure 4: *A ray-traced image of an implicit surface modeled using thousands of metaballs.*

A ray is characterized by an origin, $\vec{\mathbf{o}} \in \mathbb{R}^3$, and a direction vector, $\vec{\mathbf{d}} \in \mathbb{R}^3$. Positions along the ray are then given by the function,

$$\vec{\mathbf{r}}(t) = \vec{\mathbf{o}} + t\,\vec{\mathbf{d}} = \begin{pmatrix} x(t) \\ y(t) \\ z(t) \end{pmatrix}, \qquad t \geq 0.$$

Ray tracing opaque objects involves finding the smallest value of $t_0 \geq 0$ such that $\vec{\mathbf{r}}(t_0)$ intersects an object in the scene. Once this intersection is found, a "surface shader" uses information such as the object's surface normal and the positions of lights in the scene to determine the observed surface color. If no intersection is found, the color of the background may be used.

For implicit surfaces, this intersection criteria has a particularly nice definition. Given an implicit surface, $f(x, y, z) = 0$, we simply substitute in our parametric ray, $\vec{\mathbf{r}}(t)$, to obtain

$$f(t) = f(x(t), y(t), z(t)) = 0.$$

Thus, the problem has been translated in to that of finding roots of a univariate function, $f(t)$–precisely the problem the root finding methods you learned about in class are good for!

1.4 Rendering Fluids

An important and fun application is rendering fluids, and we will provide you with some datasets to get you started. Certain fluid simulation techniques involve discretizing a fluid volume into many tiny particles; these particles are moved around (advected) by internal fluid forces, and external forces acting on the fluid. Although the motion of these particles can effectively capture the behavior of fluid, this volumetric representation is not always ideal for rendering fluid interfaces. Fortunately, implicit representations, such as metaball surfaces (§1.2), provide a way to support ray tracing directly. In this assignment you will have the opportunity to render particle fields from fluid simulations using metaballs. If your ray tracer is fast enough, you may even have the time to create some animations from the rendered images.

2 Matlab Program Structure

For this assignment, you will be required to implement (in Matlab) a root-finding solution to the ray tracing problem described in §1.3. In short, we will provide you with a sequence of different $f(t)=0$ problems (with helpful hints) corresponding to the ray-intersection problems for every pixel in the image. This is because we provide an implementation of the implicit surface function, $f(x, y, z)$, which can be evaluated at any point in space. We also provide ray origins $\vec{\mathbf{o}}$ and directions $\vec{\mathbf{d}}$ to used to evaluate the implicit function at points along the given rays. Your task is as follows: for each ray you must find the smallest t such that

$$f(t) = f(x(t), y(t), z(t)) = 0, \quad t \geq 0,$$

where $(x(t), y(t), z(t))$ denotes a point along the ray. In the end, this boils down to finishing the implementation of this Matlab function

```
function root = CastRay(fluid, origin, direction, level, hints, tolerance)
```

where `tolerance` specifies the absolute error tolerance to be used. The return value `root` should contain the first point along the ray given by `origin` and `direction` at which the ray intersects the implicit function. `root` should be a single positive real number. If the value t_0 is returned by `root`, then this implies that if $(x(t), y(t), z(t))$ describes the ray then t_0 is the smallest value at which $f(x(t_0), y(t_0), z(t_0)) = 0$ (where f represents the implicit function).

To make your life easy, we also provide the functions

```
function y = f(t, fluid, origin, direction, level)
function d = df_dt(t, fluid, origin, direction, level)
```

so that the function

$$f(t) = f(x(t), y(t), z(t))$$

and its derivative (<u>if</u> you want it)

$$\frac{d}{dt}f(t) = \frac{d}{dt}f(x(t), y(t), z(t))$$

can be evaluated easily. The parameters `fluid`, `origin`, `direction` and `level` should not be changed and should be provided directly to the functions `f` and `df_dt`. You should only need to vary the parameter `t`. You should think of `f` as a function $f(t)$ that you are trying to find the first root of. In Matlab terms, `root` should contain the smallest non-negative value `t` such that

```
f(t, fluid, origin, direction, level) == 0
```

If no root can be found for the ray given by `origin` and `direction` then `root` should be set to -1.

The input `hints` is an array of ray t-values corresponding to the projected positions of nearby particles whose weight functions $w(r)$ contribute to $f(t)$ along the ray. These positions are sorted in increasing order of t. These can be useful in finding the general location of the root, before refining using your own root-finding method, e.g., based on Bisection, Newton, Secant, etc. The red circles in figure 1 correspond to these positions along a ray. Notice that all roots of the function occur fairly close to one of these positions. The field `fluid.radius` contains the value $r_{support}$ discussed in 1.2. You will notice that roots always occur within $r_{support}$ of at least one of the positions along the ray.

2.1 Running the Program

In Matlab, navigate to the directory where you put the assignment scripts (`RayTracer.m`, `FindIntersections.m`, `CastRay.m`, etc.). Type `RayTracer` at the Matlab prompt to run the assignment. You will be prompted for an input file, as while as an output image and the size of that image. The `RayTracer` script returns an array `intersections` containing all ray intersection points found using your implementation of `CastRay`. It also returns a structure `fluid`. This can be used in the function `TestSingleIntersection` to test your code.

Passing the parameter 1 to the `RayTracer` script (ie. `RayTracer(1)`) will cause the program to run using the provided brute-force solution (see 3.2).

2.2 Provided Data

When running the program you will be prompted to input a `*.dat` file. These files contain lists of particles defining the implicit surface you will be ray tracing, as well as some other data. Four data files can be found in `raytracer/data`, and are shown in Table 1. Sample images are also provided for all four of these data sets. We will try to make some additional interesting data sets available for you to test with during the course of the assignment.

2.3 Environment Maps (Cubemaps)

If you wish to change the environment map (and background) image used, you can do so by changing the images in

```
<raytracer root directory>/environment/cubemap_terrain
```

For example, you can visit `http://www.codemonsters.de/home/content.php?show=cubemaps` to download additional cubemap textures for this purpose. You must rename the files accordingly so that the directory `environment/cubemap_terrain` contains the files

```
terrain_negative_x.png
terrain_negative_y.png
terrain_negative_z.png
terrain_positive_x.png
terrain_positive_y.png
terrain_positive_z.png
```

3 Implementing Your Solution

As described in section 2, you will need to implement a root finder in the Matlab function `CastRay`. The following section details the general approach you will need to take.

3.1 Root Isolation and Refinement

Where to start? If you consider the root finding methods described in class, such as Bisection or Newton's method, you will realize that both of these methods have certain requirements that must be met before they can be applied to find the root of a function. Namely, Bisection requires that you possess some interval $[t_1, t_2]$ such that $f(t_1)f(t_2) < 0$, whereas Newton requires that you have some starting point t_0 such that $f'(t_0) \neq 0$ and t_0 is "close enough" to the desired root so that Newton's method will converge. Initially, neither of these criteria are satisfied. Since `CastRays` is passed only the ray origin (eg. $t = 0$), you do not know an interval bracketing a root. Furthermore, as you can see in Figure 1, $f'(t) = 0$ far enough away from the particle positions in the `hints` array, and therefore $f'(0) = 0$. So, the problem at hand is not as simple as just running a standard root finding algorithm with the given function $f(t)$. To make matters worse, the function $f(t)$ may not have <u>any</u> roots, or it may have multiple roots, in which case you are required to return the <u>first</u> one.

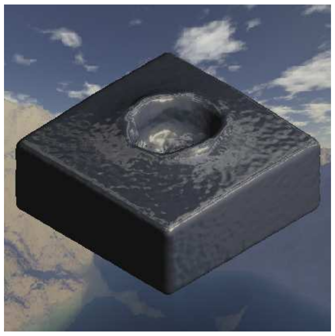
	test.dat This is a simple 3 particle scene and should be fairly easy and fast to render.
	random_particles.dat This is a 1500 particle scene in which particles are randomly distributed in a unit box centred at the origin.
	water-bunny.dat This scene contains roughly 21 000 particles and generates image 4.
	**splash.dat** This is a complicated scene with about 500 000 particles. It shows fluid splashing in a tank. Note that rendering an image from this file will tend to take much longer than rendering images from the other three provided files.

Table 1: *Example datasets provided in this assignment.*

Root Isolation and Refinement: Roughly speaking, your root-finding strategy will be broken down into two steps: root isolation and root refinement. Root isolation is the process of finding an interval $[t_0, t_1]$ such that the first root of the function $f(t)$ is contained within this interval. As you know, for continuous functions you can guarantee that one root (and maybe more!) lies in this interval if $f(t_0)f(t_1) < 0$. Finding this interval is not a straightforward task in general. A few facts that may help you in this process are as follows:

(i) Notice that $f(t) = -\ell$ for all t values sufficiently far from the particle positions indicated in the provided `hints` array (recall the definition of ℓ from 1.2). This implies that $f(0) < 0$. Therefore, a necessary (but not sufficient (why?)) condition for an interval $[t_0, t_1]$ with $f(t_0)f(t_1) < 0$ to contain the first root is that $f(t_0) < 0$.

(ii) The `hints` vector provided in `CastRay` contains the positions t of all particles that can affect the value of $f(t)$. Referring back to Figure 1, notice that the function $f(t)$ tends to have "lumps" around these particle positions (indicated by the red circles in this figure). The provided particle positions are meant to help you in finding an interval containing the first root. Of note is that all roots of $f(t)$ occur within $r_{support}$ of one of these positions ($r_{support}$ is stored in `fluid.radius`), so these provide a good starting point when looking for an interval containing the desired root.

A root isolation strategy you will likely wish to make use of, is to walk along the ray in increasing values of t in relatively coarse steps until you find the needed interval (or decide that the function has no roots). The length of these steps will affect both the speed and accuracy of your solution. Namely, smaller steps will result in more function evaluations and a slower solution but potentially more accurate solution, while larger steps increase speed at the cost of accuracy. A few hints on this:

(i) The step size you choose should depend on `fluid.radius`. For example, if the radius of particles along the ray is 0.5 and you step along the ray in increments of $\Delta t = 1$, you are very likely to miss roots of the function and identify either an incorrect root or no root at all.

(ii) You may wish to use an adaptive step size in this process; that is, a step size that changes as you march along the ray depending on properties of $f(t)$. For instance, if you are currently at a position t along the ray at which $f(t) < 0$ and $f'(t) < 0$ then t is decreasing at that point, meaning you may be able to get away with a larger step size than you would if t was increasing. Your step size may also depend on the current difference between $f(t)$ and 0. That is, as $f(t)$ becomes closer to 0, a smaller step size may be necessary to ensure that roots are not missed.

This part of the assignment is fairly open-ended. Feel free to use any strategy that you think works well. You will almost certainly have to spend some time experimenting with this to find out what works and what doesn't. You can compare your results to the brute force solution (described later).

Root refinement is the process of taking the interval $[t_0, t_1]$ from the root isolation stage, and narrowing it down to an actual root, t^*. For this, you will want to use a method with superlinear convergence, such as Newton's method, or the Secant method, or modified Regula Falsi method. Be aware, however, that methods such as Newton or Secant may diverge, or converge to a different root outside of the interval you start with. Your code should handle cases like this by falling back on the more reliable (but slower) bracketing method, such as Bisection, to find the root. Note however, that Bisection is not something you'll want your code to do very often, as its linear convergence will slow down your root finding process. Again, you will have to do some experimenting here. For example, if you find that Newton's method is rarely converging on a root, this may mean that your starting points are too far from the desired roots, and you may need to find a smaller bounding interval in the root isolation phase, or use another approach to find a better starting point.

NOTE: `CastRay`'s `tolerance` parameter specifies the root finding tolerance to use in your implementation.

3.2 Provided Brute-Force Solution

To help you evaluate your roots, we have coded a simple, brute-force solution to the root finding problem (`CastRayBruteForce.m`) that simply steps along the ray in very small steps until the first root is found. Although this method is very slow, it will provide you with a useful reference solution, i.e., you can compare the ray intersections found by your solution to the ones produced by the brute-force solution.

4 Evaluation

Your solution to this problem will be evaluated based on both accuracy and efficiency. Your goal is to minimize the number of function evaluations needed for `CastRay` to find a solution accurate to the specified tolerance (a bound on absolute error). Note that $f(t)$ and $\frac{df}{dt}$ both count as a separate function evaluation, i.e., one step of Newton costs 2, and one step of Secant costs 1. Please also note the following:

> **IMPORTANT:** At no point in your code should you modify the GLOBAL variable `FUNCTIONEVALS`. We use this variable to count the number of function evaluations ($f(t)$ and $\frac{df}{dt}$) performed by your program.

4.1 Testing

The `RayTracer` script provides a pair of return values; `fluid` and `intersections`. After running the `RayTracer` program, this data will be useful for testing your implementation. We have provided two scripts for testing:

- **The `TestSingleIntersection` script** will run a single ray intersection (read, "root finding attempt") using your solution and the brute-force solution provided in `CastRayBruteForce`. To run this test script type

  ```
  [intersection, bruteForceIntersection, diff, evals, bfEvals]
        = TestSingleIntersection(fluid, x, y, tolerance)
  ```

  in the Matlab command prompt. This function casts a single ray through the implicit surface and uses both your implementation of `CastRay` and `CastRayBruteForce` to find an intersection with the surface. Here `fluid` is the structure returned by `RayTracer`; the parameters `x` and `y` specify the coordinates from which to cast the ray–look in the fields `fluid.minPoint` and `fluid.maxPoint` to find the minimum and maximum bounds of the implicit surface, respectively; `tolerance` specifies the absolute error used by both root finding methods when looking for a root. This script returns three values: (1) `intersection` is the intersection value t returned by your root finder, (2) `bruteForceIntersection` is the intersection value t returned by the brute-force root finder, and (3) `diff` is the absolute difference between the two. If your method is working properly, then you should observe that `diff` $\leq$ `tolerance`. You may see cases in which `diff` is very large (ie. 10^5). This indicates that one root finder discovers a root and the other does not for the given inputs. `evals` and `bfEvals` return the number of function evaluations performed by your root finder and the brute force solution, respectively.

- **The `TestCrossSection` script** tests several root intersections against the brute force solution along a cross section of the data stored in `fluid`. It can be called as follows:

  ```
  [numMisses, percentError, evals, bfEvals]
  = TestCrossSection(fluid, y, xMin, xMax, stepSize, tolerance)
  ```

  This function will start at the coordinate $(xMin, y)$ and step along the x direction in increments of `stepSize` until `xMax` is reached, testing each ray intersection along the way. It returns the number of times your solution disagrees with the brute force solution, as well as the percentage of times the solutions disagree. These values are stored in `numMisses` and `percentError`, respectively.

4.2 What to Submit?

You need to submit the following:

1. **A single Matlab file (`CastRay.m`)** containing your root finding implementation. If you need to add any additional functions you may do so below the provided functions in `CastRay.m`.

2. **Code Documentation:** Your program should be documented thoroughly. Numerical code without good documentation can be very hard to read, so it is important that you explain to us what you are doing via documentation in your program. If there is any point in your solution at which is is not completely clear what your are doing or why you are doing it, explain it using comments. If your approach has limitations or potential shortcomings, explain it using comments. If you do something clever in your program to speed it up, or make it more accurate, or make it more robust, explain it using comments. Part of your mark is based on how easy your code is to understand. A more detailed description of your algorithm should be included in the following...

3. **A Design Document:** You will also be required to submit a short document ($\frac{1}{2}$ to 1 page long) containing a high-level description of your algorithm. This should explain details such as how you went about isolating and refining roots in your solution (see 3.1).