# The `smoothcontour3` module for Asymptote

Charles Staats III

May 22, 2015

**Abstract**

The `smoothcontour3` module uses an adaptive algorithm to draw implicitly defined surfaces with smooth appearance. This documentation explains its use.

## Contents



## 1 Usage

The API for this module consists of the single function `implicitsurface()`:

```
surface implicitsurface(real f(triple) = null,
                        real ff(real,real,real) = null,
                        triple a,
                        triple b,
                        int n = nmesh,
                        bool keyword overlapedges = false,
                        int keyword nx=n,
                        int keyword ny=n,
                        int keyword nz=n,
```

```
                                int keyword maxdepth = 8);
```

The function has three required[1] parameters: a function $f$ (of type either `real(triple)` or `real(real,real,real)`) and two triples `a` and !b!. It returns a `surface` that approximates the zero locus of $f$ within the rectangular solid with opposite corners `a` and `b`.

Here are the explanations of the optional parameters:

- `int n`—the number of initial segments in each of the $x$, $y$, $z$ directions. Defaults to `nmesh`, which is usually 10.
- `bool overlapedges`—if `true`, the patches of the surface are slightly enlarged to compensate for an artifact in which the viewer can see through the boundary between patches. (Some of this may actually be a result of edges not lining up perfectly, but I'm fairly sure a lot of it arises purely as a rendering artifact.) Defaults to `false`. Keyword required—this parameter can only be used in key-value format.
- `int nx`—overrides `n` in the $x$ direction. Keyword required.
- `int ny`—overrides `n` in the $y$ direction. Keyword required.
- `int nz`—overrides `n` in the $z$ direction. Keyword required.
- `int maxdepth`—the maximum depth to which the adaptive algorithm will subdivide in an effort to find patches that closely approximate the true surface. Keyword required.

## 2 When should I use this module?

The algorithm is designed to deal with a differentiable function with a smooth zero locus. Simple point singularities in the zero locus are usually not a problem, but complex singularities (e.g. two tangent spheres) or one-dimensional singularities will take a long time to compute with no guarantee of a good result.

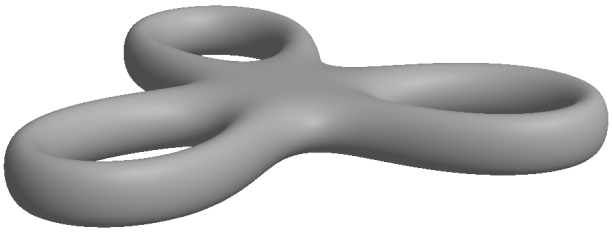
On one occasion I have seen it cope with a continuous, piecewise differentiable function, although it took a while to compute. Whatever you do, *do not* pass it a piecewise constant function—it will take forever and return nothing good.

[1] The syntax seems to say that `f` and `ff` are both optional parameters. However, the function will throw a runtime error unless exactly one of `f`, `ff` is set. This somewhat peculiar arrangement was chosen so that the module works in all three of the following scenarios:

- The user wants to graph the zero set of a function `f(real,real,real)`.
- The user wants to graph the zero set of a function `f(triple)`.
- The user has overloaded a function name `f`, defining both `f(triple)` and `f(real,real,real)`.

While the first two cases could be covered by overloading `implicitsurface`, that would cause an ambiguous call in the third case.

(a)
`overlapedges=true`



(b) without setting
`overlapedges`

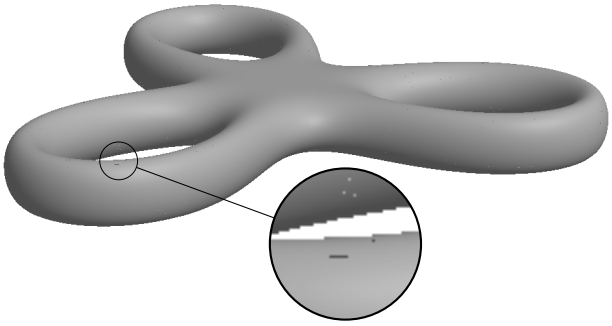


Figure 1: A smooth surface of genus three.

# 3 Examples

## 3.1 Genus three surface

The first example is code for a smooth surface of genus three; the output is Figure 1. Note that because of the adaptive nature of the algorithm, we can get a good result without having to set `nx`, `ny`, or `nz` manually.

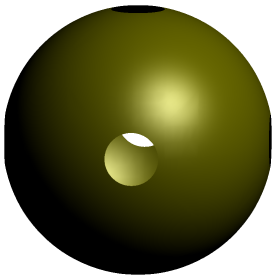
```
settings.outformat = "png";
settings.render = 4;
size(8cm);

import smoothcontour3;


// Erdos lemniscate of order n:
real erdos(pair z, int n) { return abs(z^n-1)^2 - 1; }

real h = 0.12;

// Erdos lemniscate of order 3:
real lemn3(real x, real y) { return erdos((x,y), 3); }

// "Inflate" the order-3 lemniscate into a smooth surface:
real f(real x, real y, real z) {
```

```
  return lemn3(x,y)^2 + (16*abs((x,y))^4 + 1) * (z^2 - h^2);
}

draw(implicitsurface(f, (-3,-3,-3), (3,3,3),
                     overlapedges=true),
     surfacepen=material(diffusepen=gray(0.5),
                                 emissivepen=gray(0.3),
                                 specularpen=gray(0.1)) );
```

Also shown is the result obtained by omitting the parameter `overlapedges=true`, with the rendering artifacts emphasized. The figure on top is not wholly devoid of such artifacts, but they are fewer and less prominent. Increasing `settings.render` can also reduce the appearance of such artifacts.

## 3.2 Cropped sphere

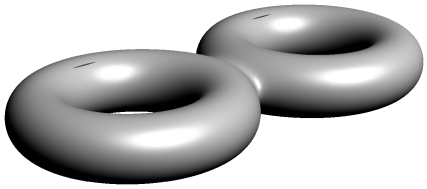
Next, we see how the module can be used to produce a cropping effect: If a surface can be described as the zero locus of a smooth function, then the plotted zero locus is automatically cropped to the specified rectangular solid.



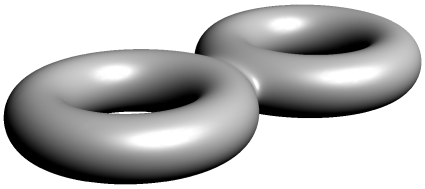
```
settings.outformat="png";
settings.render=8;
size(3.5cm,0);
import smoothcontour3;
currentprojection = orthographic(20,1,3);

real f(triple w) { return w.x^2 + w.y^2 +
    w.z^2 - 1; }

draw(implicitsurface(f, (-0.98,-0.98,-0.98), (0.98,0.98,0.98),
             n=2, overlapedges=true),
     surfacepen=olive);
```

The parameter `n=2` is specified to produce a more efficient surface. This version of the cropped sphere has "only" 408 Bézier patches. This is in part because the algorithm decides that `n=2` is actually to small and subdivides everything, so actually `n=2` and `n=4` give identical results. But by comparison, the default `n=10` gives 1152 patches, with no apparent change in the quality of the rendering.

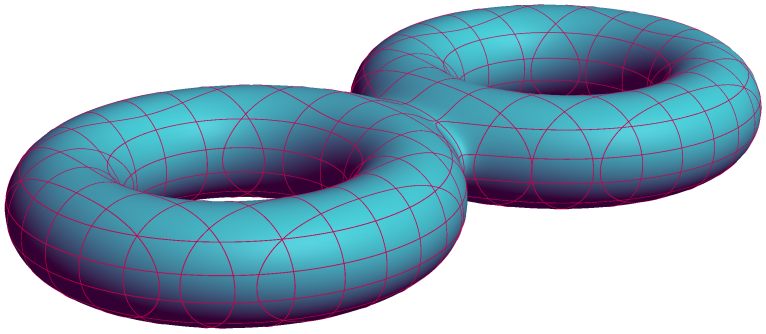
Another thing to note here is that the algorithm would not "notice" the need to crop the sphere if the vertices were not positioned exactly right: in this case, `n=2` produces better results than `n=3`. We do get good results for `n=5`. If you change the cropping to 0.99 instead of 0.98, then `n=2` still works great, but `n=9` is the first *odd* value of `n` at which the cropping shows up.



(a) default parameters



(b) `nx=20`, `nz=5`



(c) gridlines drawn using the `contour` module

Figure 2: A surface of genus two.

### 3.3 Genus two surface

Here's an attempt to draw a surface of genus two (with the result shown in Figure 2a):

```
settings.outformat="png";
settings.render=8;
size(0.95*165.59853pt, 0);
import smoothcontour3;
import contour;
currentprojection=perspective((18,20,10));

real tuberadius = 0.69;

// Convert to cylindrical coordinates to draw
// a circle revolved about the z axis.
real toruscontour(real x, real y, real z) {
  real r = sqrt(x^2 + y^2);
  return (r-2)^2 + z^2 - tuberadius^2;
}

// Take the union of the two tangent tori (by taking
// the product of the functions defining them). Then
```

```
// add (or subtract) a bit of noise to smooth things
// out.
real f(real x, real y, real z) {
  real f1 = toruscontour(x - 2 - tuberadius, y, z);
  real f2 = toruscontour(x + 2 + tuberadius, y, z);
  return f1 * f2 - 0.1;
}

// The noisy function extends a bit farther than the union of
// the two tori, so include a bit of extra space in the box.
triple max = (2*(2+tuberadius), 2+tuberadius, tuberadius) +
    (0.1, 0.1, 0.1);

// Draw the implicit surface.
draw(implicitsurface(f, -max, max, overlapedges=true),
     surfacepen=white);
```

The algorithm, as you can see, is not always smart enough to identify all the things it should draw. In this case, the gaps can be eliminated by changing the $10 \times 10 \times 10$ grid used for initial computations to a $20 \times 10 \times 5$ grid. so that the individual cells are closer to being cubes. To do this, replace the final line by

```
draw(implicitsurface(f, -max, max, overlapedges=true,
                     nx=20, nz=5),
     surfacepen=white);
```

One more variant, shown in Figure 2c, involves using the pre-existing `contour` module to compute gridlines. Note that this is highly memory-intensive.

```
⋮
size(10cm,0);

⋮
triple max = (2*(2+tuberadius), 2+tuberadius, tuberadius) +
    (0.1, 0.1, 0.1);
triple min = -max;

// Draw the implicit surface.
draw(implicitsurface(f, min, max, overlapedges=true,
                     nx=20, nz=5),
     surfacepen=material(green+0.8blue+0.1red,
             emissivepen=0.2blue+0.2red,
             specularpen=gray(0.1)));

/****************** WARNING *************************/
```

```
// The following code (for drawing gridlines) is highly
// memory-intensive. On my computer, it used over
// 1.7 GB of RAM.
/*****************************************************/

// To draw the gridlines, we need to compute the implicit curves
// in the planes defined by fixing one of the three variables.
typedef real function2(pair);
function2 xfixed(real x) {
  return new real(pair p) { return f(x, p.x, p.y); };
}
function2 yfixed(real y) {
  return new real(pair p) { return f(p.x, y, p.y); };
}
function2 zfixed(real z) {
  return new real(pair p) { return f(p.x, p.y, z); };
}

// These planes are used to lift a path to a path3.
typedef triple plane(pair);
plane liftx(real x) {
  return new triple(pair p) { return (x, p.x, p.y); };
}
plane lifty(real y) {
  return new triple(pair p) { return (p.x, y, p.y); };
}
plane liftz(real z) {
  return new triple(pair p) { return (p.x, p.y, z); };
}

pen gridpen = 0.8 red + 0.3 blue + linewidth(0.3pt);

int nx=20, ny=10, nz=5;

for (int i = 0; i <= nx; ++i) {
  real x = (i/nx) * (max.x - min.x) + min.x;
  guide[] wholepath = contour(xfixed(x), (min.y,min.z),
                              (max.y,max.z),
                              new real[]{0}, 200)[0];
  plane currentplane = liftx(x);
  for (guide pathpiece : wholepath) {
    draw(path3(pathpiece, currentplane), gridpen);
  }
}

for (int j = 0; j <= ny; ++j) {
```

```
  real y = (j/ny) * (max.y - min.y) + min.y;
  guide[] wholepath = contour(yfixed(y), (min.x,min.z),
                              (max.x,max.z),
                              new real[]{0}, 200)[0];
  plane currentplane = lifty(y);
  for (guide pathpiece : wholepath) {
    draw(path3(pathpiece, currentplane), gridpen);
  }
}

for (int k = 0; k <= nz; ++k) {
  real z = (k/nz) * (max.z - min.z) + min.z;
  guide[] wholepath = contour(zfixed(z), (min.x,min.y),
                              (max.x,max.y),
                              new real[]{0}, 200)[0];
  plane currentplane = liftz(z);
  for (guide pathpiece : wholepath) {
    draw(path3(pathpiece, currentplane), gridpen);
  }
}
```

# 4 Troubleshooting

In most graphing algorithms in Asymptote, the user specifies the number of subdivisions, and many artifacts can be corrected by increasing this number. For drawing smooth implicit surfaces, that approach proved impractical. Instead, the algorithm attempts to detect artifacts itself, and subdivides into smaller cells when it does. This works surprisingly well, but not perfectly.

Most of the artifacts below can be mitigated by increasing the number of subdivisions using the parameter `n`. However, simply increasing this parameter should be regarded as a "nuclear option." In addition to significantly increasing the computational resources required, it actually makes the first and most common artifact *worse*.

## 4.1 Random missing pixels

Perhaps the most common artifact to show up when using this package is that scattered pixels will randomly show up the wrong color; see Figure 1b, p. 3. These pixels are actually showing up at the edges between patches, where the renderer believes (usually mistakenly) that there is a gap it can "see through." There are several ways to mitigate the problem; they can be used together except for the last.

1. Set `overlapedges = true` in the call to `implicitsurface()`. This option artificially inflates the individual patches by 1% in an effort to convince

the renderer that all the gaps between edges are filled. It usually helps a lot.

2. Increase `settings.render`. As a general rule, asking the renderer to work harder will mitigate the random missing pixels.

3. Decrease the parameters `n` and/or `nx`, `ny`, `nz`. If you can decrease the number of patches (which is not guaranteed), then you correspondingly get fewer edges to produce problems. Also, if the patches are larger (since fewer), then `overlapedges=true` becomes more effective since a 1% increase in the size of the patches produces a bigger increase in absolute terms.

4. When drawing the surface, use a `meshpen`. This is not recommended—the mesh you get will not be one you want to see—but it will fill the gaps. Do *not* use this with `overlapedges=true`.

## 4.2 Gaps in the surface

Sometimes, the algorithm will fail to notice a piece of the surface that should be drawn, leaving a gap as in Figure 2a.

1. **Jiggle it a little.** Change small things to see if you can remove the coincidence that stumped the algorithm. For instance, add or subtract a small amount from the bounds, or change `n` by one. In the case of Figure 2a, subtracting 0.05 from the upper $z$ bound (and adding 0.05 to the lower $z$ bound) is sufficient to eliminate the problem.

2. **Judiciously change `nx, ny, nz`.** For instance, if $z$ can be described as a function of $(x, y)$, try setting `nz=1`. In the case of Figure 2a, the gaps are long in the $x$ direction, so you might try doubling `nx` to 20 from the default of 10. Since the gaps have little or no extent in the $z$ direction, and since the figure itself is not high in the $z$ direction, you might simultaneously decrease `nz` from 10 to 5. In this way, you have solved the issue (producing Figure 2b). Furthermore, since $20 \cdot 10 \cdot 5 = 10 \cdot 10 \cdot 10$, you have not actually increased the computation time.

3. **Significantly increase `n`.** Theoretically, if the surface is smooth, `n` is sufficiently large, and a small random amount of random noise is added to the bounds, errors of this sort should occur only with probability zero.[2]

## 4.3 Important features drawn wrong

For instance, a "cropped sphere" might fail to be cropped, or a very tiny sphere might go missing from the diagram. Or bad things could happen if the surface is supposed to have multiple "sheets" in close proximity.

[2] This is true for a suitably high level of abstraction. At a lower level of abstraction, a computer is a finite-state machine, so there's no such thing as "possible but with probability zero."

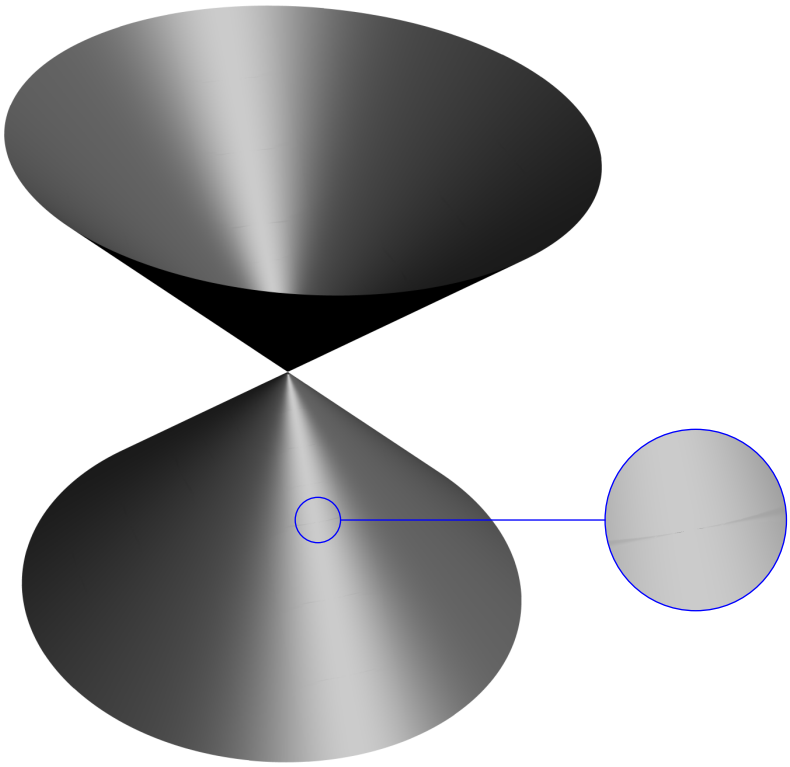


Figure 3: The graph of $x^2 + y^2 - z^2 = 0$ over the domain $[-1, 1]^3$. Note the minor wrinkle.

The advice here is the same as in 4.2; however, this time the problem is too vaguely worded for me to offer any "probability zero" guarantees.

### 4.4 Strange smoothness artifacts

Sometimes strange artifacts show up, such as the wrinkle in Figure 3. To deal with this, you can try the same 4.2, but with absolutely no assurance of success. Another thing you can try is to play with the coloring, lighting, and point of view to make the artifact less obvious. (In truth, it required some playing with coloring, lighting, and point of view to make the artifact visible in Figure 3.)

One particular piece of advice here: a strong `specularpen` can highlight any such artifacts if they actually exist. So if you are trying to de-emphasize an artifact, decreasing the `specularpen` and increasing the other pens in the `material` may be wise.