Isosurface Tutorial

Notes concerning PDF adapted from archived Printable Version – Bill Walker "Bald Eagle" 3/2021

Mike Williams' excellent Isosurface Tutorial at http://www.econym.demon.co.uk/isotut/ has served an an invaluable resource and indispensible reference for everyone in the POV-Ray community experimenting with isosurfaces and functions for the past 17 years. In the process of converting the 175-page hypertext markup language (html) version to a static Adobe portable document (PDF) format, defunct hypertext links and duplicate or rendundant notes and comments were deleted, the typeface size was decreased, some minor text and layout formatting was performed for aesthetics, some typographical errors were corrected and changes were made to reflect updated language syntax and deprecated keywords.

Additions: colored keyword highlighting

Originally, links were provided to 28 separate zip files containing the POV source files for all the images that appeared on individual isosurface example webpages. These have been condensed into a single zip file.

Introduction

Isosurfaces are a feature of version 3.5 of POV-Ray. (*At the time of editing, POV-Ray is in version 3.7, with a 3.8 beta*) This tutorial assumes that you have some basic knowledge of isosurfaces.

General Points

Because of my mathematical background, I prefer to express my surfaces with zero threshold.
e.g. I think of a sphere as $x^2 + y^2 + z^2 - R^2 = 0$ rather than $x^2 + y^2 + z^2 = R^2$.
So I express my isosurfaces like:

```
function {x*x + y*y + z*z - 1}
threshold 0
```

But some people might prefer to write the same isosurface as:

```
function {x*x + y*y + z*z}
threshold 1
```

I've given almost all the images in this tutorial a visible indication of the object that the isosurface is contained_by. This is particularly useful in those images where the isosurface touches the container, so you can clearly see which features are caused by intersections with the container and which are a natural feature of the isosurface.

Hint I find it useful to add such visualisations of the container when developing surfaces. It helps avoid confusion that may arise if the isosurface accidentally touches the container. It also helps me spot situations in which the container is excessively large, wasting lots of rendering time. I comment out the

```
sphere {0, R pigment {rgbt <1, 0, 0, 0.9>}}
```

once I'm happy with the behaviour of the isosurface.

The source files for the examples on each page are available as ZIP files.

The tutorial is split into the following sections:

Isobars

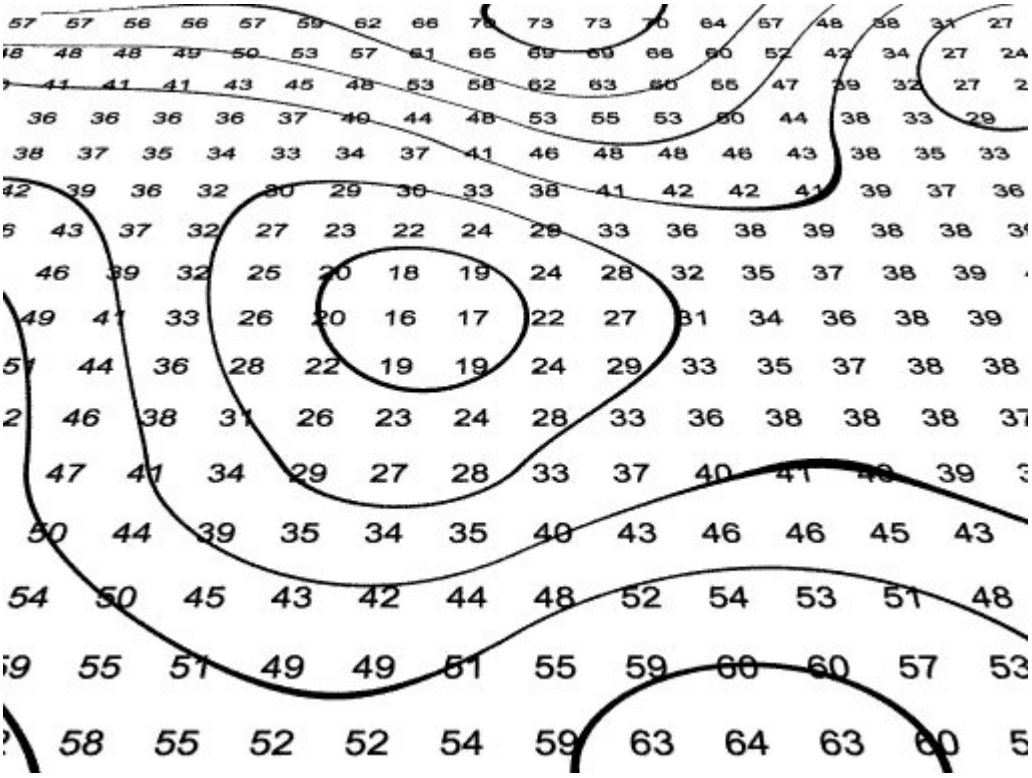
Isobar Analogy

In order to get a feel for what isosurfaces are, it might be useful to think about something that you might be more familiar with, the isobars that are drawn on weather maps.

Weather stations gather information about the air pressure at different points on the ground, and then the meteorologists draw lines through the points where the numbers are the same.

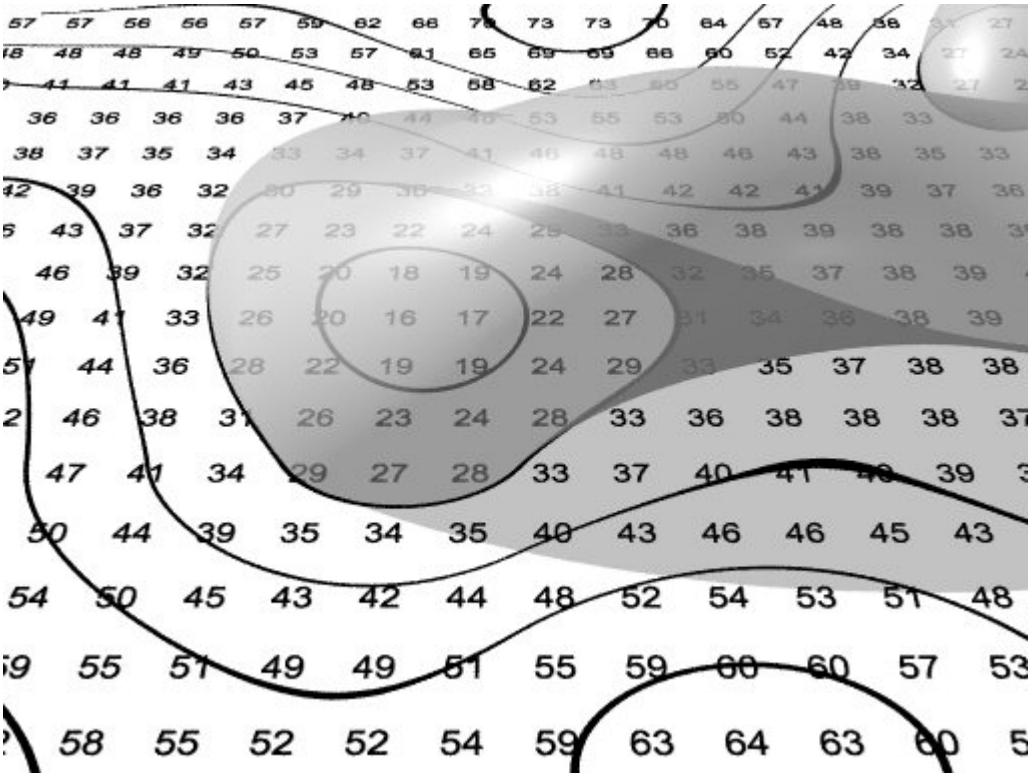
As well as isobars, which connect points of equal pressure, there are isotherms for temperature, isoclines for inclination, isogonics for magnetic declination etc. They are all curves that join points where some numeric value is the same.

The first image shows a ground level with numerical values at various points.



In second image there are lines joining points where the numerical values are 20, 30, 40, 50 and 60.

In the third image, we remove the limitation that the numerical values are only on the ground. Suppose there are numerical values (not shown) at every point in space. We can now join points that have the same numerical value with 3d surfaces. In this case the surface shown connects all the points where the numerical value is 30.



Rather than using external numbers that represent something real, like air pressure, the numerical values that are used to generate POVRay isosurfaces come from mathematical functions. A function is simply a formula that generates a numerical value for points in 3d space. For example if we take function { x*x + y*y + z*z - 4 }, then POV-Ray can calculate the value at a point like <1, 2, 0> by plugging those x, y, z values into the formula 1*1 + 2*2 + 0*0 - 4 giving a value of 1.

By carefully choosing the mathematical formula, we can make the associated isosurface take up a vast range of shapes that would be difficult to achieve otherwise.

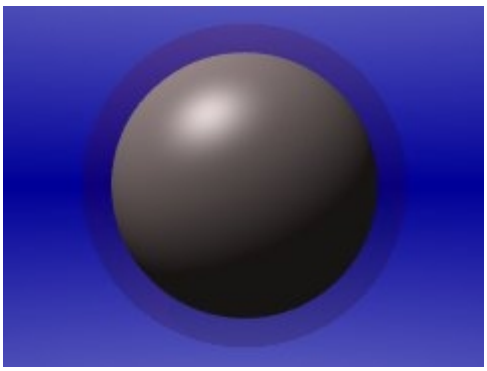
Simple Isosurfaces

It's possible to build most, if not all, of the basic POV shapes with isosurfaces. In practice, you'd probably not want to bother with an isosurface if there's an equivalent basic shape available, but understanding these simple shapes gives us insight into how to develop more complex surfaces.

The syntax used for the isosurfaces used on this page is like

```
isosurface {
    function {x*2 + y*2 + z*2 - R*R}
    accuracy 0.001
```

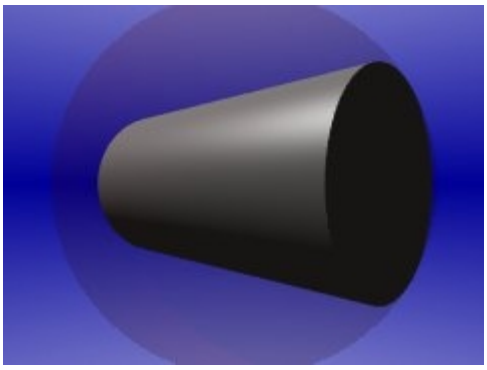
```
    max_gradient 4
    contained_by {sphere {0, 1.2}}
    pigment {rgb .9}
    finish {phong 0.5 phong_size 10}
}

sphere {0, 1.2 pigment {rgbt <1, 0, 0, 0.9>}}
```



Sphere

function { x*x + y*y + z*z - R*R }

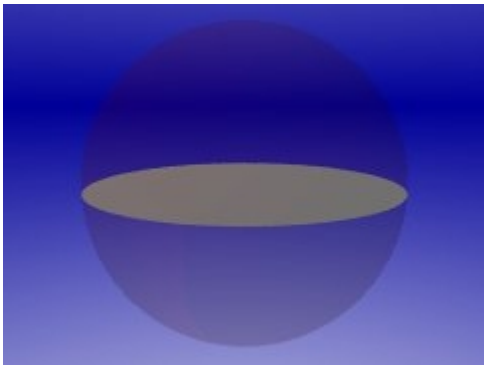
Maths: Take a look at what happens on the x, y, and z planes. On the x plane, x = 0, the equation reduces to y*y + z*z - R*R = 0 the two dimensional equation of a circle radius R. Similarly the intersection with the y and z planes are also circles. This probably doesn't come as much of a surprise, but if you get a feel for how the 2D equations come together to make the 3D equation you can sometimes get a feel for how to build other 3D surfaces.



Cylinder

function { y*y + z*z - R*R }

Maths: The cross section of this object on any x plane is clearly always the circle y*y + z*z - R*R = 0.



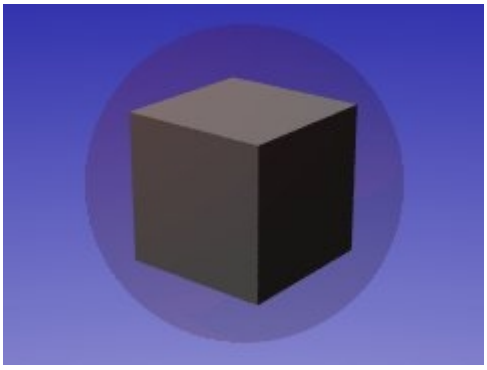
Plane

function {y}

Maths: Well, functions don't come much simpler than that. The isosurface exists wherever y takes the value zero.

This is the y plane. The x plane and z plane are simply function {x} and function {z}.

The plane through the point <0,1,0> is given mathematically by y = 1 so the function (with threshold zero) is therefore function { y - 1 }.

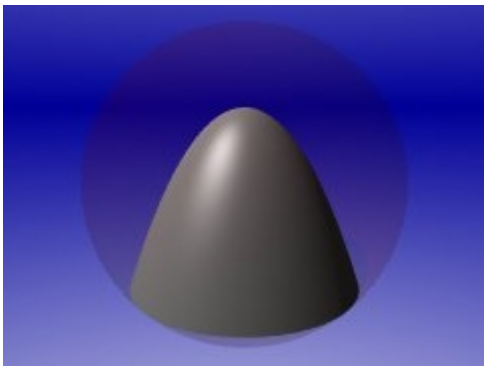


Box

function {max ((y*y-1), (x*x-1), (z*z-1) - R*R}

Maths: What I've done here is to take the double-plane **function {y*y-1}** and intersect it with similar x and z double planes. max() performs intersections of isosurfaces.

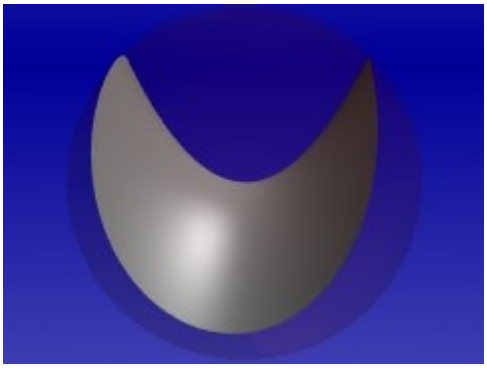
function {y*y-1} is mathematically y^2 -1 = 0 which has the two planar solutions y = +1 and y = -1, so the function describes the two planes.



Paraboloid

function {x*x + y + z*z - 1}

Maths: Observe that the intersection with the y plane is a circle x^2 + z^2 - 1 = 0 and the intersections with the x and z planes are parabolas y = x^2 + 1.



Hyperbolic Paraboloid

function {x*x + y - z*z}

Maths: Observe that the intersection with the y plane is a hyperbola x^2 - z^2 = 0 and the intersections with the x and z planes are parabolas y = -x^2.

Octahedron

function { abs (x) + abs(y) + abs(z) - 1 }

Syntax Subtleties

Max_gradient

POV-Ray 3.5 almost always needs to be told the maximum gradient of the function that you are asking it to evaluate. If you don't specify a max_gradient, or you specify a value that is too low then things will usually go horribly wrong. If you specify a value that is too high, the surface will render correctly but more slowly than necessary.

Here's a very simple example where nasty holes appear in the surface:-

```
isosurface {
    function {x*x + y*y + z*z - 1}
    accuracy 0.0001
    contained_by {sphere {0, 1.2}}
    pigment {rgb 0.9}
    finish {phong 0.5 phong_size 10}
}
```

That's just the Sphere from the Simple Surfaces page, but with the max_gradient 2.4 missing.

What happens is that the function evaluator guesses where each ray hits the surface and then improves the guess by walking up and down the ray. If max_gradient is set higher then the evaluator will take a larger number of small steps. If max_gradient is set too low, then the evaluator will take fewer large steps and may step right past the intersection without noticing.

If you're mathematically inclined, then it is possible to calculate the actual maximum gradient. In the case of this sphere, we happen to know that the function is symmetrical so we can go looking for the maximum gradient in any direction. Let's choose to look along the x axis. Along the x axis the value of y and z are always zero, so the function simplifies to x*x -1. If we differentiate that we obtain the gradient as 2*x. Within contained_by{sphere{0,1.2}}, 2*x ranges from -2.4 to +2.4, so we should add max_gradient 2.4.

This mathematical method is rather tedious, and we'll soon be meeting isosurface functions that we don't know how to differentiate, so we need to find another way to find the max_gradient. We could just guess, and keep trying higher values until the nasty holes disappear, or we could use the evaluate keyword.

In practice, I usually guess something like max_gradient 2 and render a small image and look at the Messages pane. There will be a warning in the Messages if POV-Ray considers the max_gradient to be incorrect, e.g.

Warning: The maximum gradient found was 4.734, but max_gradient of the isosurface was set to 7.000. Adjust max_gradient to get a faster rendering of the isosurface.

Some functions contain singularities: points where the gradient becomes infinite. The message will say that the maximum gradient found was some large number like 518238.857. In most such cases you can get away with using a considerably lower value if you are prepared to accept that the regions extremely close to the singularity may not render correctly.

Differences

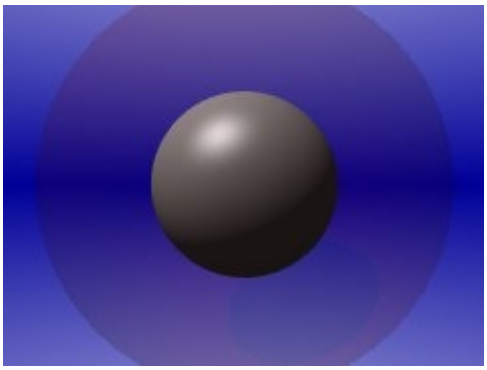
This section examines the differences in syntax between MegaPOV and POV-Ray v3.5 isosurfaces, and subtle differences in the ways that they work.

contained_by The contained_by keyword is mandatory. The braces after the type of container are optional - e.g. **contained_by {sphere 0,1}**	The contained_by keyword is optional. If absent it defaults to contained_by {box {-1,1}} The braces after the type of container are mandatory - e.g. **contained_by {sphere {0,1}}**
method There are two methods for evaluating functions, selected by **method 1** or **method 2**	All functions are evaluated with the same method. There is no **method** keyword.
eval **eval** can be used if you don't know the max_gradient of the function. It is possible to specify three parameters for **eval**, but these parameters are not explained sufficiently for them to be useful.	*evaluate* Three parameters are required. This feature is currently in development, and its current operation does not match the documentation. POV 3.5 will display a suggested value for the max_gradient if it considers that the value you have used is unsuitable, whether you use evaluate or not.
sign An isosurface can be turned inside out by using **sign -1**. This is particularly useful with certain built-in functions which are written in such a way that the inside is the opposite of what would normally be expected. e.g. **function {"Bicorn" <1,1>}** **threshold 0.1** **sign -1**	There is no **sign** keyword. To turn an isosurface inside out you must negate the function and negate the threshold. e.g. **function { - f_bicorn(x,y,z,1,1)}** **threshold - 0.1**
unions and intersections The **|** and **&** operators can be used to create unions and intersections of functions	Those operators now perform logical OR and logical AND operations. Unions and intersections are now performed by using **min()** and **max()**.
referencing #declared functions If you #declare a function "F", you can reference it as just "F", in which case "(x, y, z)" is assumed.	If you #declare a function "F", you must always include the three parameters, e.g. "F(x,y,z)".
built-in functions The built-in functions are always available. They look like **function {"Sphere" <4>}** The names of the functions are not case specific. To use a built in function in an expression you must #declare it first.	To use built-in functions, you must use **#include "functions.inc"** They look like **function {f_sphere(x, y, z, 1)}** The names of the functions are case specific. They're all in lower case. The functions can be used in expressions
operators with restricted ranges You can use operators like square roots and logarithms outside their normal ranges. E.g. you can use **sqrt(x)** and **ln(x)** without worrying whether x might be negative at some points. The results are therefore technically mathematically incorrect, but reasonably intuitive.	You can only use operators within the ranges where they are mathematically meaningful. **ln(x)** and **log(x)** will cause the render to terminate with a floating point exception if x becomes zero or negative at any point. **sqrt(x)** will cause the space to be filled with blackness wherever x is negative. **asin(x)** will cause the space to be filled with blackness wherever x is greater than 1.
coincident surfaces If part of an isosurface is coincident with another surface, it becomes invisible.	If part of an isosurface is coincident with another surface, the traditional coincident surface artefacts appear.

I use this quite often in this tutorial, adding a visible, partially transparent, copy of the contained_by object so that you can see where the container is.	In this tutorial I have made the contained_by surfaces **open** where this was problematical. Fortunately, the **open** keyword causes much less of a speed penalty than it does in MegaPOV.
evaluating pigments When you use a pigment as an isosurface function, the value is calculated from the red component and the green component. The red component being more significant.	When you use a pigment function in an isosurface, you must specify which component of the colour is to be used, e.g. **MyPigmentFn(x,y,z).red**. The possible components are .red, .blue, .green, .filter, .transmit, .grey and .hf. The .hf component is the same as MegaPOV, but is considered an experimental feature.
power operator MegaPOV supports the **"^"** operator.	In 3.5 you can either replace **x^2** by **x*x** or by **pow(x,2)**.

Hints
If you're converting stuff between MegaPOV and POV 3.5 you might find it handy to keep both programs running. By default this doesn't happen, but there is a way to achieve this.
In both programs, from the options menu, switch off the "keep single instance" flag.
Start MegaPOV before starting POV 3.5, not the other way round.

Variable Substitution



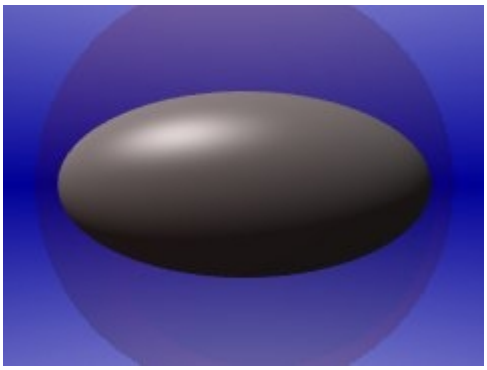
#declaring functions

It's possible to #declare a function for later use in an isosurface, but the x, y and z values used in the declaration are formal parameters and it's possible to substitute different actual parameters when the function is invoked.

```
#declare  S = function {x*x + y*y + z*z - 1}

isosurface {
  function {S(x,y,z)}
        accuracy 0.001
        contained_by {sphere {0, R}}
        pigment {rgb 0.9}
}
```

In this case, we use the same actual parameters as the formal parameters, and the result is a sphere, exactly as if we had said

```
isosurface {
  function {x*x + y*y + z*z - 1}
  ...
}
```



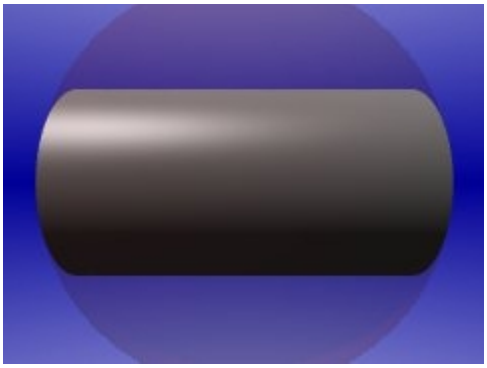
Scale

But we don't have to use the same parameters when we invoke the function. We could use **function { S(x/2, y, z) }** instead of **function { S(x, y, z) }**. This has the effect of performing a substitution of x/2 in the equation wherever the variable x occurred, giving us a surface equivalent to

```
isosurface {
    function {(x/2)*(x/2) + y*y + z*z - 1}
  ...
}
```

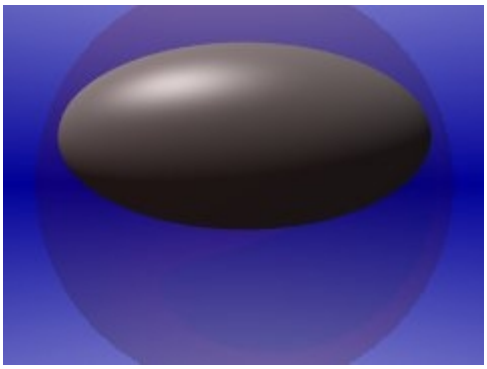
We could have written that in the first place, or we could have applied a conventional **scale <2,1,1>** transformation to the whole thing. *Note:* Applying **scale** would also scale the contained_by surface.

Note that to scale the x dimension by a factor of 2, we substituted x/2. This is a common feature of substitutions - things change in an inverse way to the change made to the variable. If the variable is halved then the scale doubles.



function {S(x, 0, z)}

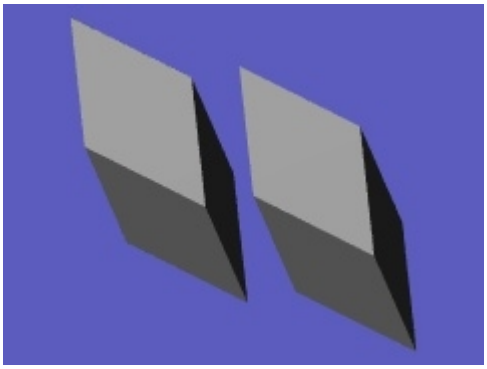
Substituting 0 for y causes the sphere to degenerate into a cylinder. You might consider this to be equivalent of an infinite scaling in the y direction.



Translation

function { S(x/2, y-0.5, z) }

Substituting y-0.5 for y causes the surface to be translated +0.5 in the y direction. We could have achieved this effect by translate <0, 0.5, 0>.



Shear

It's a bit more dificult to produce a particular shear transformation, because the values that we need to use in the variable substitution need to be the **inverse** of those normally used to generate the shear. We could calculate the inverse of the transformation matrix by hand, but it is possible to get POV-Ray to do it for us.

We first create a shear transformation, then declare a transformation function using its inverse.

```
#include "transforms.inc"
#declare TR = Shear_Trans (<-0.5, 0.5, 0.5>, <1.1, 0, -0.3>, <-0.3, 0.5, 0>)
#declare TRFI = function {transform {TR inverse}}
```

Now we can apply that transformation function to the unit vectors which gives us the values we need for the substitution. However, we can't use vectors from inside an isosurface function, so we have to extract the x, y and z values of each of those vectors outside the isosurface.

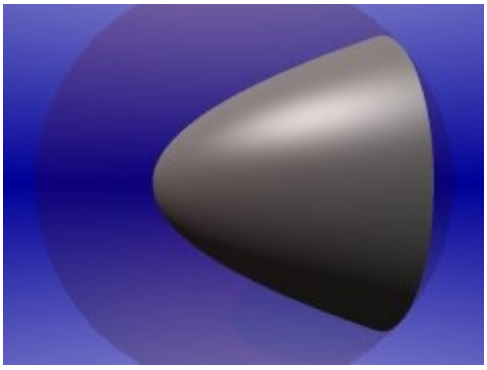
```
#declare A = TRFI (1, 0, 0);
#declare B = TRFI (0, 1, 0);
#declare C = TRFI (0, 0, 1);
#declare Ax = A.x; #declare Bx = B.x; #declare Cx = C.x;
#declare Ay = A.y; #declare By = B.y; #declare Cy = C.y;
#declare Az = A.z; #declare Bz = B.z; #declare Cz = C.z;
```

We can now use these scalar values in the variable substitution.

```
function {F(Ax*x+Bx*y+Cx*z, Ay*x+By*y+Cy*z, Az*x+Bz*y+Cz*z)}
```

The image on the left shows two boxes. One of them is sheared by variable substitution and the other is sheared with the equivalent conventional shear transform.

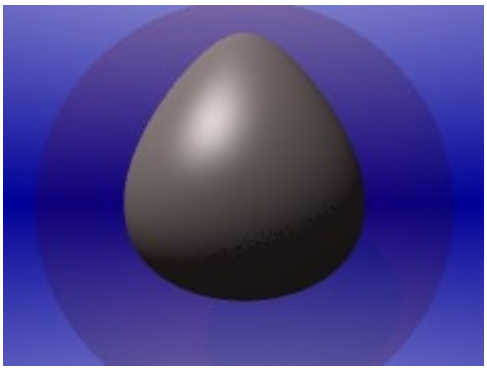
The same method can be used for combinations of shear, rotation and scale operations. It doesn't seem to work for transformations that involve translation, but we already know how to do that.



Flip

```
#declare  P = function {x*x + y + z*z - 1}

isosurface {
    function {P (y,-x, z)}
  ...
```

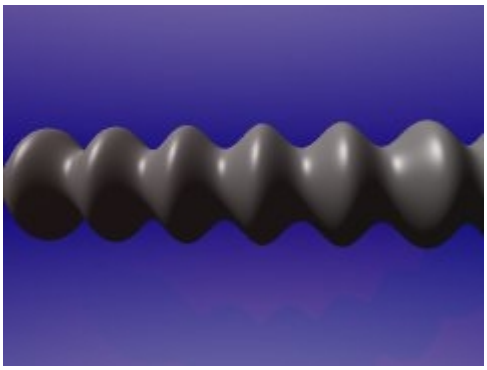
Substituting y for x and substituting -x for y causes this paraboloid to be flipped round to face in the -x direction instead of the +y direction.



Non-linear scale

```
#declare  P = function {x*x + y*y + z*z - 1}

isosurface {
    function {P(x, y*(1.05-y/5), z)}
  ...
```

A non-linear stretch has turned this sphere into something like a hen's egg. The sphere is stretched more as y becomes larger, and compressed more as y becomes more negative.

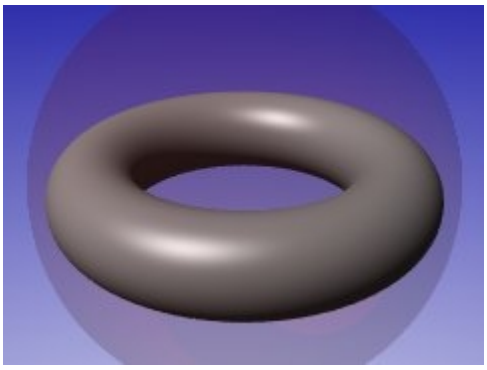


Surface of revolution

This is a particularly interesting variable substitution. It generates a surface of revolution from the positive part of an equation of two variables.

In this case the function F describes the sin wave "y = sin(x) + 4", the "+4" is there just to make the value of y always positive.

```
#declare  F = function {y-sin (x)-4}
isosurface {
    function {F(x, sqrt(y*y + z*z), z)}
  ...
```

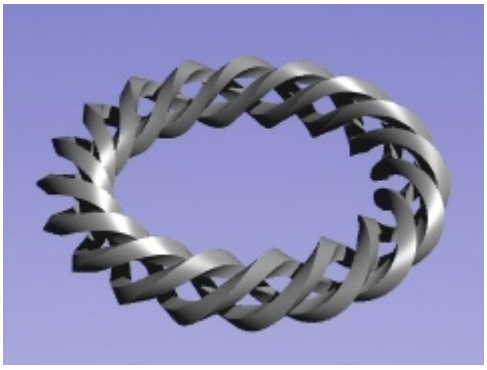


You can create a surface of revolution from the equation of any 2d curve in the same way.

A particular instance of a surface of revolution is the torus. The parameters r1 and r2 are the major and minor radii.

The original function {x*x + y*y - r2*r2} creates a 2d circle of radius r2, and the substitution shifts that circle a distance r1 along the x axis, then creates a surface of revolution from it.

```
#declare  F = function {x*x + y*y - r2*r2}

isosurface {
    function {F (sqrt (x*x + z*z)-r1, y, z)}
  ...
```

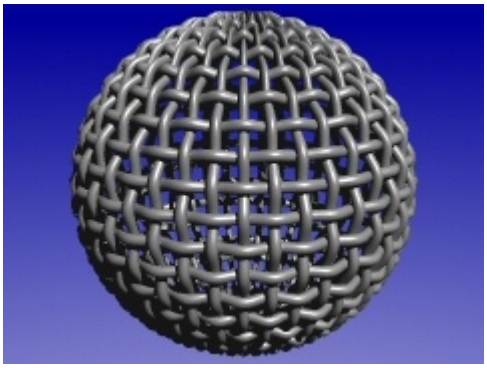


By transforming from cartesian to cylindrical polar coordinates, it's possible to bend any isosurface into a circle around the origin.

The cylindical polar coordinates are made available through the f_th() and f_r() functions, so you don't have to work them out for yourself.

```
#declare F = function {f_helix1 (x, z, y, Strands, Turns, R1, R2, 0.6, 2, 0)}
isosurface {
   function {F (f_r (x,y,z)-R3, y, f_th(x, y, z))}
```

```
...
```

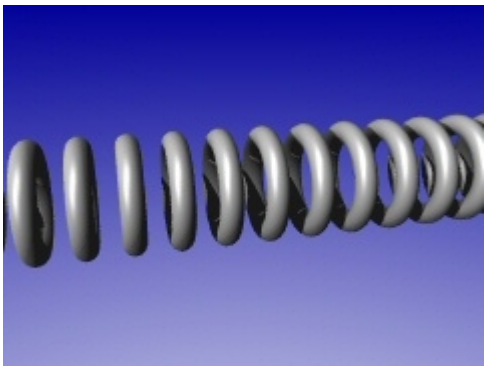


Similarly, it's possible to wrap a sheetlike isosurface around a sphere by transforming from cartesian to spherical polar coordinates.

The spherical polar coordinates are made available through the f_th(), f_r() and f_ph() functions, so you don't have to work them out for yourself.

The image on the left is created from a f_mesh1() function which has been converted to polar coordinates. The threads that previously lay in the x and z directions now lie in the longitude and latitude directions of the sphere. What was previously the height in the y direction is now altitude in the radial direction, with the "-1" lifting the whole thing radially away from the origin.

```
#declare F = function {f_mesh1 (x, y, z, 0.15, 0.15, 1, 0.02, 1) - 0.03}
isosurface {
    function {F (f_ph (x, y, z), f_r (x, y, z)-1, f_th (x, y, z))}
    ...
```



It's possible to use a single isosurface to produce multiple copies of a shape by using the **mod** operator in a substitution.

Using mod(x,2) will cause the shape to be repeated in the x direction every 2 units. If your original isosurface created a shape centred at the origin, then there's a problem with the way that it chooses which bits to repeat. The left half of the object gets repeated in one direction and the right half of the object gets repeated in the other direction. To fix this, we'd like to substitute mod(x,2)+1 where x is negative and mod(x,2)-1 where x is positive. To fix both directions at once, for a symmetrical object, we can use abs(x) like this mod(abs(x),2)-1.

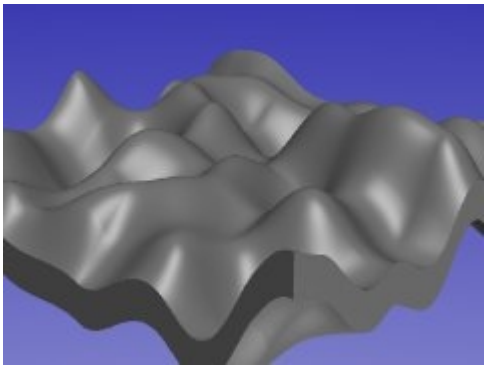
To change the length of the repeat unit we can do this mod(abs(x),Step)-Step/2.

We can repeat things in more than one direction, making a sheet or filling a volume with repeated units by performing similar substitutions in the x, y and z directions.

This trick can be extremely useful in situations where awkward CSG operations cause POV to apply very inefficient bounding. If we had created thousands of separate badly bounded objects, then POV would have to perform thousands of intersection tests on each ray. If we use this trick to use a single isosurface to generate thousands of objects, then POV only has to perform a single intersection test on each ray. That intersection test may well be somewhat slower than that for a non repeating surface, but it's likely to be much faster than performing a thousand such tests.

You need to be particularly careful with your max_gradient setting for this trick. Slight changes to the step size sometimes require large changes in max_gradient.

```
#declare  F = function {f_torus(y, x, z, 0.8, 0.19)}
#declare Step = 0.75;
isosurface {
    function {F ( mod(abs (x), Step)-Step/2, y, z)}
```

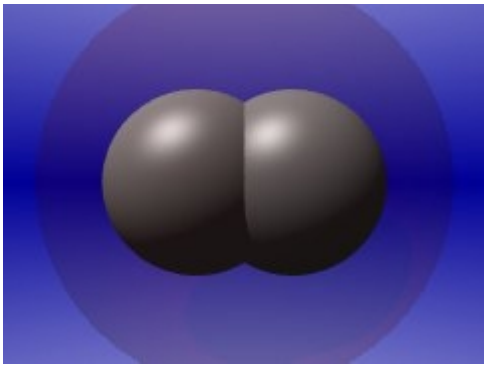


Thickening

If we have a function F(x,y,z) we can turn it into two parallel surfaces by using **abs(F(x,y,z))-C** where C is some small value. The original function should be one that works with zero threshold. The two resulting surfaces are what you would get by rendering the original function with threshold +C and -C, but combined into a single image. The space between the two surfaces becomes the "inside" of the object. In this way we can construct things like glasses and cups that have walls of non-zero thickness.

```
#declare F = function {y + f_noise3d (x*2, 0, z*2)}
isosurface {
    function {abs (F(x, y, z))-0.1}
    ...
```

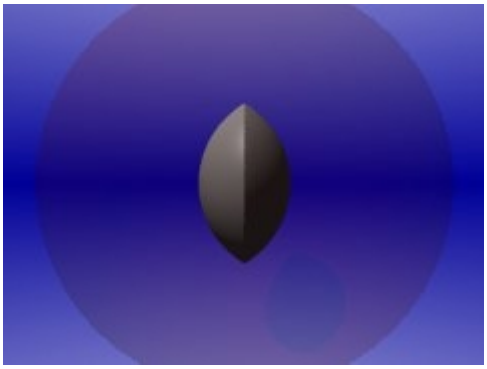
Combining Functions



Union

```
#declare  S = function {x*x + y*y + z*z - 1}

isosurface {
    function {min(S (x+0.5, y, z), S (x-0.5, y, z))}
    max_gradient 2
    contained_by {sphere {0, R}}
    pigment {rgb 0.9}
}
```

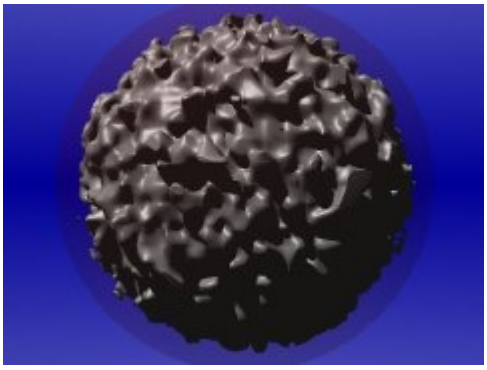
min() can be used to produce the union of two or more functions. In this case the union of two spheres, one translated -0.5 in the x direction and one translated +0.5 in the x direction.



Intersection

```
#declare S = function {x*x + y*y + z*z - 1}

isosurface {
    function {max(S (x+0.5, y, z), S (x-0.5, y, z))}
    max_gradient 5
    contained_by {sphere {0, R}}
    pigment {rgb 0.9}
}
```

max() can be used to produce the intersection of two or more functions. In this case the same two spheres as above.



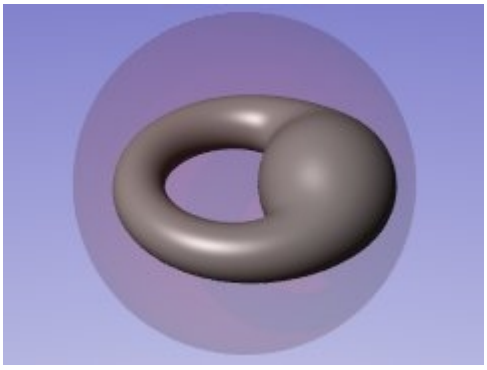
Addition and Subtraction

```
#include "functions.inc"
#declare S = function {x*x + y*y + z*z - 1}
isosurface {
    function { S(x, y, z) + f_noise3d (x*10, y*10, z*10)*0.3}
    max_gradient 7
    contained_by {sphere {0, R}}
    pigment {rgb 0.9}
}
```

Adding two functions together produces a sort of blend between the two functions. A particular use of such a blend is to add one or more noise or noise-like functions to a surface.

In this case I've added a bit of f_noise3d to a sphere. The result of the addition is a surface with a generally spherical shape but with a noisy surface. I've used variable substitution to control the frequency of the noise.

Adding noise creates a surface that is slightly smaller than the original sphere - the noise pokes inwards from the surface. Subtracting noise creates a surface that is slightly larger than the original sphere - the noise stands slightly proud of the surface.



Blob-like combinations

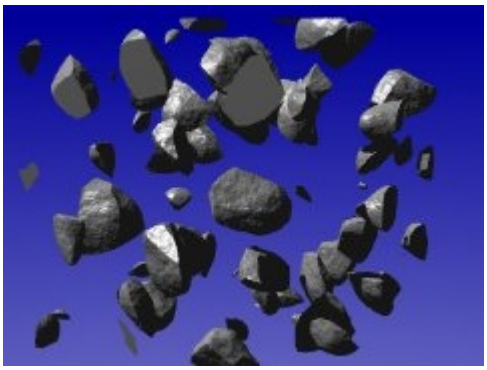
```
#include "functions.inc"
#declare  S = function {f_sphere (x, y, z, 0.5)}
#declare  T = function {f_torus (x, y, z, 1, 0.2)}
isosurface {
    function {S (x-0.7, y, z) * T (x, y, z) - 0.05}
    max_gradient 2
     0.001
    contained_by {sphere {0, R}}
    pigment {rgb 0.9}
}
```

Multiplying surfaces then adding a small constant produces an effect that's similar to using blobs. However, this technique can be used to blob together any kind of isosurface, not just spheres and cylinders.

Pigments as Functions

At the fundamental level, a pigment pattern is something that supplies a colour value to every point in space. When we use the pattern to paint a pigment onto a surface, POV-Ray evaluates a numerical value at each point on the surface of the object and looks up the result in a colour_map to produce the colour of that point.

It's possible to use the components of these colours as an isosurface function. If we choose the "red" component, then the surface is considered to exist wherever the red component of the pigment is equal to the isosurface threshold.



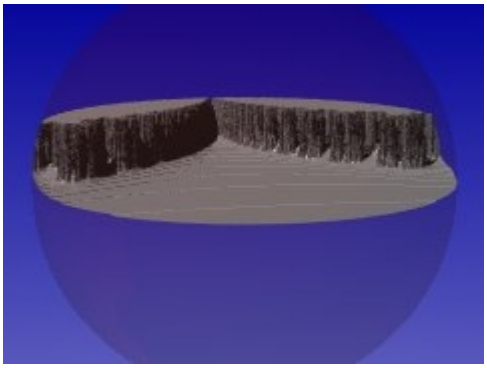
```
#declare F = function {
    pigment {
        crackle
        turbulence 0.1
        color_map {[0 rgb 1] [1 rgb 0]}
        scale 0.5
    }
}

isosurface {
    function {F (x, y, z).red - 0.5}
    max_gradient 5.5
    contained_by {box {-1, 1}}
```

```
    pigment {rgb 0.9}
}
```

On its own a pigment function doesn't usually look like much of anything.

```
#declare F = function {
    pigment {
        crackle
        turbulence 0.1
        color_map {[0 rgb 1] [1 rgb 0]}
        scale 0.5
    }
}
isosurface {
         function {x*x + y*y + z*z - 1
            + F (x, y, z).grey*0.3}
         max_gradient 3.5
         contained_by {sphere {0, R}}
         pigment {rgb 0.9}
}
```

But when we add or subtract a pigment function to a shape like a sphere it's possible to produce interesting effects. This is the same pigment function as above, but this time 0.3 of it is added to a sphere instead of just the pigment being allowed to fill the contained_by object.



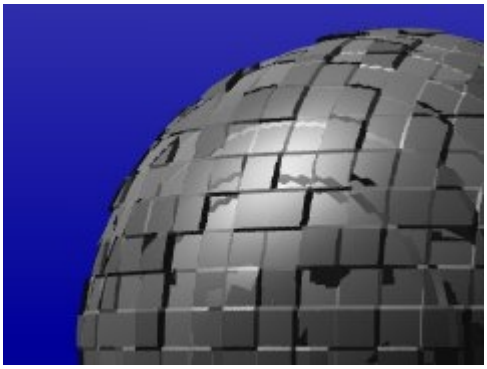
```
#declare F = function {
```

```
    pigment {
    mandel 50
    color_map {[0 rgb 0] [1 rgb 1]}
    scale 8
    translate <-3, 0, 0>
    }
}

isosurface {
        function {y - F (x, y, z).grey*0.3}
        contained_by {sphere {0, R}} open
        pigment {rgb 0.9}
}
```

Pigment isosurfaces can be useful for landscapes. This canyon is made from a Mandelbrot pigment added to the y plane.

Because the Mandelbrot pigment faces the z direction, I flipped the axes by using F(x,z,y) instead of F(x,y,z).

The Mandelbrot pigment has, by default, a rather strange colour map which isn't suitable for our purposes, so color_map { [0 rgb 0] [1 rgb 1] } is used to disable the default colour map.



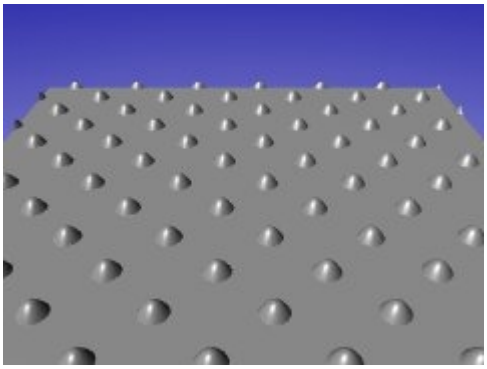
These pigment isosurfaces can be used to give something like "greebles" and might make a nice surface for a spaceship hull.

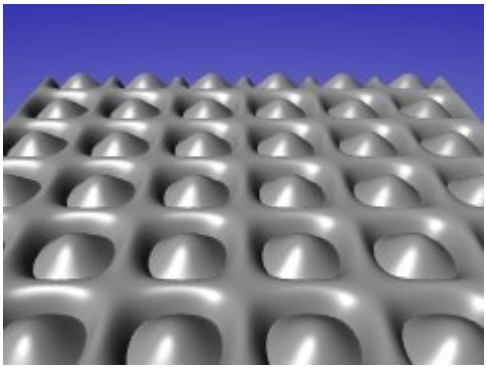
This version uses the crackle pigment with metric 0 and solid modifiers.

```
#declare P = pigment {
  crackle
  metric 0
  solid
  color_map {[0 rgb 1] [1 rgb 0.5]}
  scale 0.1
}
#declare F = function {pigment {P}}
function {x*x + y*y + z*z -1 + F(x, y, z).red*0.2}
```

```
#declare P = pigment {
  cells
  color_map {[0 rgb 1] [1 rgb 0.5]}
  scale 0.1
}
#declare F = function {pigment {P}}
function {x*x + y*y + z*z -1 + F (x, y, z).red*0.1}
```

This version produces a similar effect using the **cells** pattern, and runs very much faster.





You don't always have to stick with simple colour maps like colour_map {[0 rgb 0] [1 rgb 1]}.

These two surfaces were made with the leopard pigment by changing the colour map.

For the first one, the insertion of [0.3 rgb 0] holds the pigment value at zero for a while, creating a flat zone from which the bumps rise sharply.

I've used P(x, 0, z) instead of P(x, y, z) to prevent changes in the function values above the plane which would otherwise cause blobs of surface to become detached.

```
#declare P = function {
    pigment {
        leopard
        colour_map {
            [0.0 rgb 0.0]
            [0.3 rgb 0.0]
            [1.0 rgb 1.0]
        }
        scale 0.1
    }
}
function { y - P (x, 0, z).red*0.4}
```

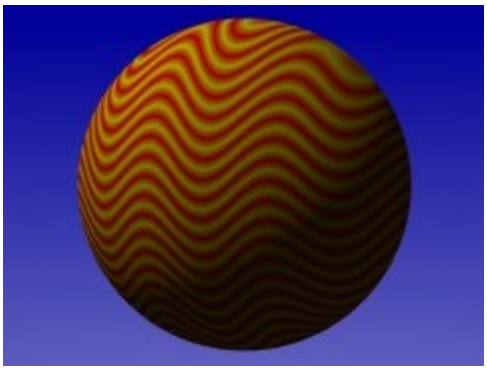
In the second one, I've changed the zero entry of the colour map to push up what would normally be the lowest part of the surface.

```
#declare P = function {
    pigment {
        leopard
        colour_map {
            [0.0  rgb 0.3]
            [0.1  rgb 0.0]
            [1.0  rgb 1.0]
    }
    scale 0.15
    rotate y*45
    }
}
function {y - P (x, 0, z).red*0.4}
```

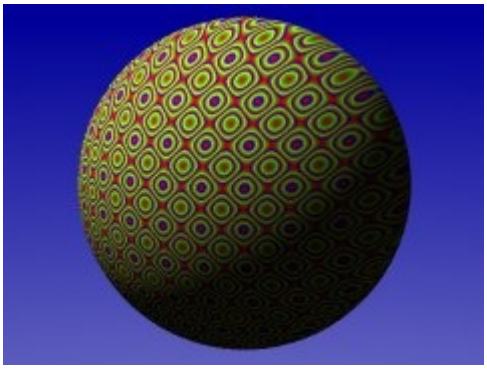
Functions as Pigments

As we saw on the previous page, pigment patterns and isosurface functions are fundamentally just things that supply numerical values to all points in space.

It turns out that we are allowed to create patterns from functions and use them for pigments or normals. In this way it's possible to create a pigment that obeys any mathematical equation we like.

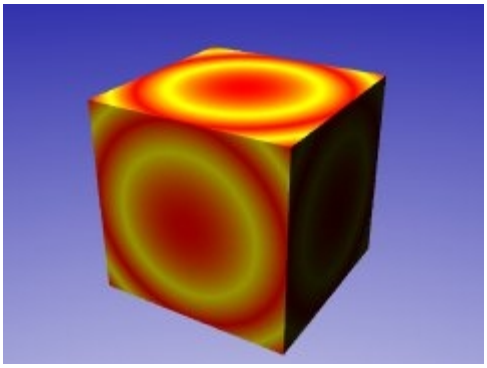


```
sphere {0, 1
    pigment {
        function {sin (x*10) + 10*y}
        color_map {
            [0.0 rgb <1,0,0>]
            [0.5 rgb <1,1,0>]
            [1.0 rgb <1,0,0>]
    }
  }
}
```

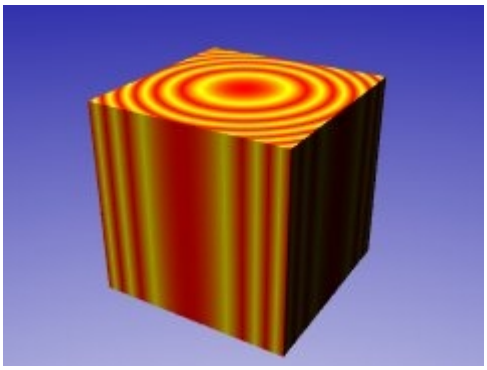


```
sphere {0,1
    pigment {
        function {sin (x*30) + sin (y*30)}
        color_map {
            [0.0 rgb <1, 0, 0>]
            [0.2 rgb <0, 0, 1>]
            [0.5 rgb <1, 1, 0>]
            [0.7 rgb <0, 1, 0>]
            [1.0 rgb <1, 0, 0>]
        }
```

```
    }
}
```



```
box {-1, 1
    pigment {
        function {x*x + y*y + z*z}
        color_map {
            [0.0 rgb <1, 0, 0>]
            [0.5 rgb <1, 1, 0>]
            [1.0 rgb <1, 0, 0>]
        }
    }
}
```



```
box {-1, 1
    pigment {
        function {(x*x + z*z)*3}
        color_map {
            [0.0 rgb <1, 0, 0>]
            [0.5 rgb <1, 1, 0>]
            [1.0 rgb <1, 0, 0>]
        }
```

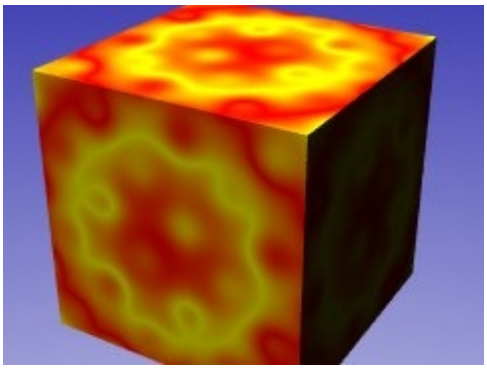
```
    }
}
```

```
sphere {0, 1
    pigment {
        function {sin (x*20)/sin (y*20)*0.7}
        color_map {
            [0.0 rgb <1, 0, 0>]
            [0.5 rgb <1, 1, 0>]
            [1.0 rgb <1, 0, 0>]
        }
    }
}
```

```
sphere {0, 1
    pigment {
        function {atan2 (z, x)*2}
        color_map {
            [0.0 rgb <1, 0, 0>]
            [0.5 rgb <1, 1, 0>]
            [1.0 rgb <1, 0, 0>]
        }
```

```
    }
}
```

It's even possible to start out with pigment patterns, convert them into functions, perform mathematical operations on those functions that are not possible with pigments, then convert them back into pigments again.



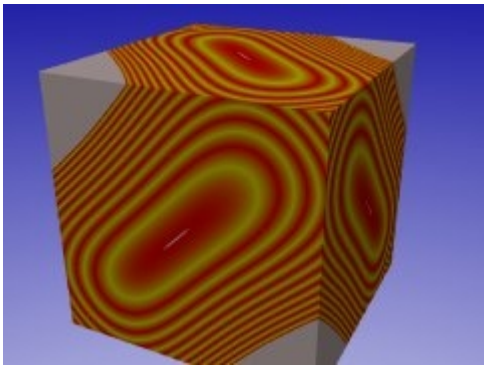
```
#declare F1 = function {pigment {onion scale 0.5}}
#declare F2 = function {pigment {leopard scale 0.1}}

box {-1, 1
    pigment {
        function {F1 (x, y, z).grey + F2 (x, y, z).grey*0.5}
        color_map {
            [0.0 rgb <1, 0, 0>]
            [0.5 rgb <1, 1, 0>]
            [1.0 rgb <1, 0, 0>]
        }
    }
}
```

In this case, I've made functions F1 and F2 from the onion and leopard pigment patterns and simply added them together. If you use certain built in functions to generate pigments you may find some areas (like alternate corners of the following cube) where there are patches of plain colour. This isn't a natural feature of the function, but an artefact of the library code.

These functions set a bound on the maximum numerical value that can be returned. This value is usually 10.0 but a few surfaces allow the value to be passed as a parameter. If there's a point where the function calculates a value greater than 10.0, the library returns the value 10.0, which for the purposes of calculating a pigment is equivalent to 0.0

I've made this more apparent in this image by mapping values that are very close to 0.0 to white. Where the function values vary the white stripe is so thin that it gets anti-aliassed into invisibility, but in areas where the function would exceed 10.0 there is plain whiteness. There are 10 yellow bands visible in this image,

corresponding to regions where the Steiner's Roman function takes the values 0.5 to 9.5.



```
#include "functions.inc"
box {-0.5, 0.5
    pigment {
        function {f_steiners_roman (x, y, z, 90)}
        color_map {
            [0.000 rgb <1, 1, 1>]
            [0.001 rgb <1, 0, 0>]
            [0.500 rgb <1, 1, 0>]
            [0.999 rgb <1, 0, 0>]
            [1.000 rgb <1, 1, 1>]
        }
    }
}
```

Parametric Equations

Parametric surfaces render very slowly in POV 3.5.

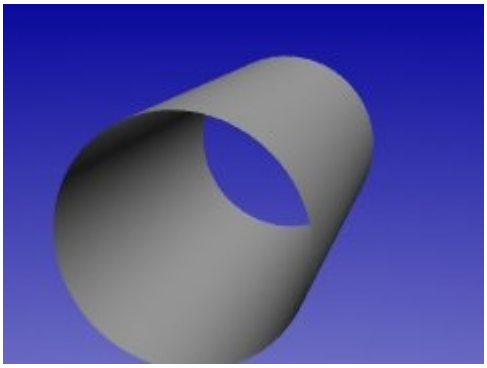
Sometimes it is difficult, or even impossible, to specify a single equation in x, y and z that specifies the surface but a set of parametric equations might be available.

In the 2d case, we know that x*x + y*y - r*r = 0 is the equation of a circle, but we can also describe it by the two parametric equations x = cos(theta), y = sin(theta). Each value of theta gives a value for x and y, i.e. a 2d point. As theta varies from 0 to 2pi the point goes round the unit circle.

In the 3d case we need three parametric equations, one each for x, y and z; and we need two parameters which we will call u and v. If we consider the equations

```
x = sin (u)
y = cos (u)
z = v
```

we can see that each pair of values for u and v gives a single xyz point in 3d space. As u varies from 0 to 2pi, the point goes round a circle. As v varies from -2 to 2 the point moves parallel to the z axis. It turns out that these are the parametric equations for a cylinder.



This is the cylinder generated by these parametric equations.

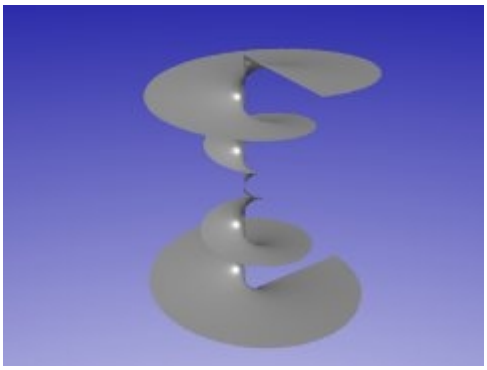
The three parametric functions are listed; then the u,v bounds; Then the contained_by object.

The u,v bounds used here specify that {u,v} varies from {0,-2} to {2*pi, 2} i.e. u varies from 0 to 2*pi and v varies from -2 to 2.

The "contained_by" object can be a sphere or a box.

The "isosurface", "evaluate", "max_trace", "threshold", "open" and "closed" keywords are not used.

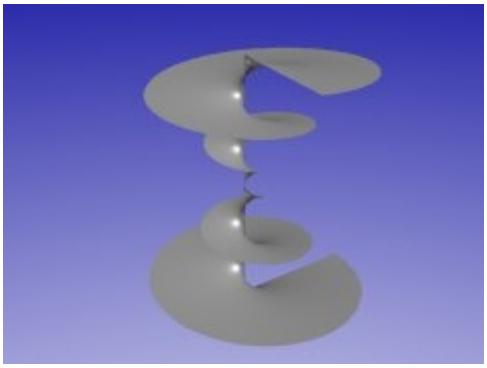
```
parametric {
    function {sin (u)}
    function {cos (u)}
    function {v}
    <0, -2>,<2*pi, 2>
    contained_by {box {<-2, -2, -2>, <2, 2, 2>}}
    pigment {rgb 0.9}
    finish {phong 0.5 phong_size 10}
}
```



Here's a sort of conical spiral that would be very difficult to specify without using parametric equations.

```
parametric {
  function {u*v*sin (15*v)}
  function {v}
  function {u*v*cos (15*v)}
      <0, -1>, <1, 1>
```

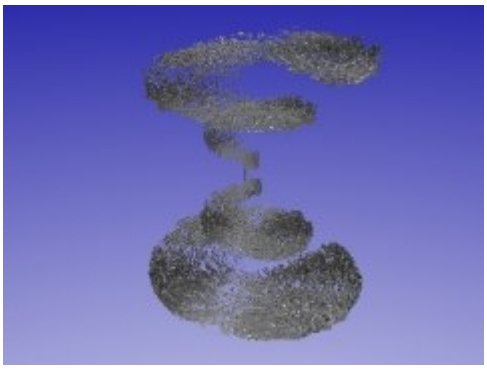
```
  max_gradient 4
  contained_by {box {<-R, -R, -R>, <R, R, R>}}
```



This is the same surface as last time, but in this case I've declared two of the equations beforehand. It is possible to perform variable substitution here.

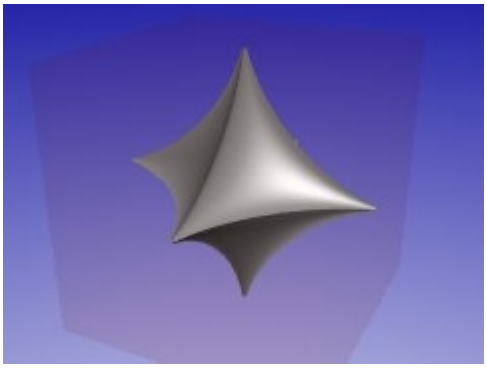
The "precompute" keyword can speed up the rendering by telling the renderer to store some calculations in an array, thus trading memory and parsing time against rendering time. I find that "precompute 18, x,y,z" tends to give a reasonable speed improvement on my machine. Higher values cause the parse time to become rather long. In this case, there's no speed gain from precomputing y, but there's not much of a penalty either.

```
#declare F1 = function {u*v*sin (15*v)}
#declare F2 = function {u*v*cos (15*v)}
parametric {
    function {F1 (u, v, 0)}
    function {v}
    function {F2 (u, v, 0)}
    <0, -1>, <1, 1>
    contained_by {box {<-R, -R, -R>, <R, R, R>}}
    precompute 18, x, y, z
    accuracy 0.003
    pigment {rgb 0.9}
    finish {phong 0.5 phong_size 10}
}
```



I get the distinct impression that things like pigment functions aren't really supported with parametric surfaces. The parser seems to be trying to prevent me from using "pigment" in the same function as "u" or "v", but the following syntax is accepted.

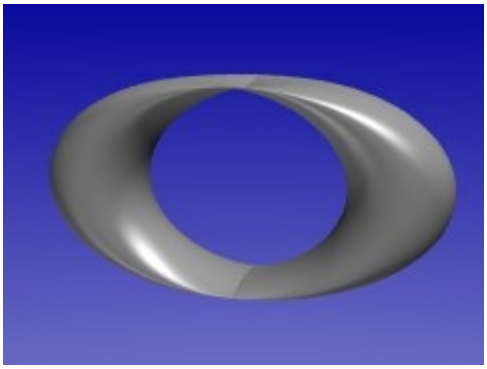
```
#declare Fx = function {u*v*sin (15*v)}
#declare Fy = function {v}
#declare Fz = function {u*v*cos (15*v)}
#declare Fp = function {pigment {granite scale 0.1}}
parametric {
    function {Fx (u, v, 0)}
    function {Fy (u, v, 0) + Fp (u, v, 0).grey*0.2}
    function {Fz (u, v, 0)}
    <0, -1>, <1, 1>
    contained_by {box {<-R, -R, -R>, <R, R, R>}}
    precompute 18, x, y, z
    pigment {rgb 0.9}
    finish {phong 0.5 phong_size 10}
    no_shadow
}
```



This is the Astroidal Ellipse. The surface goes in where an ordinary ellipse goes out.

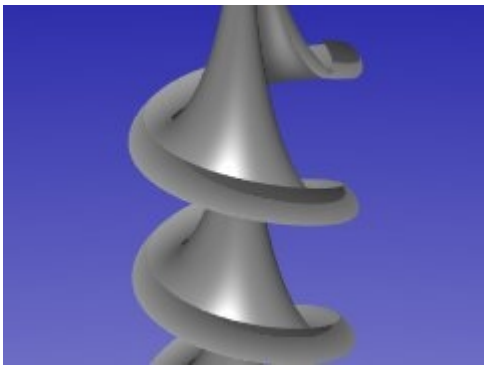
```
#declare A = 1;
```

```
#declare B = 1;
#declare C = 1;
#declare Fx = function {pow (A*cos (u)*cos (v), 3)}
#declare Fy = function {pow (B*sin (u)*cos (v), 3)}
#declare Fz = function {pow (C*sin (v), 3)}
parametric {
    function {Fx (u, v, 0)}
    function {Fy (u, v, 0)}
    function {Fz (u, v, 0)}
    <0, 0>,<2*pi, 2*pi>
    contained_by {box {<-R, -R, -R>, <R, R, R>}}
    precompute 18, x, y, z
```



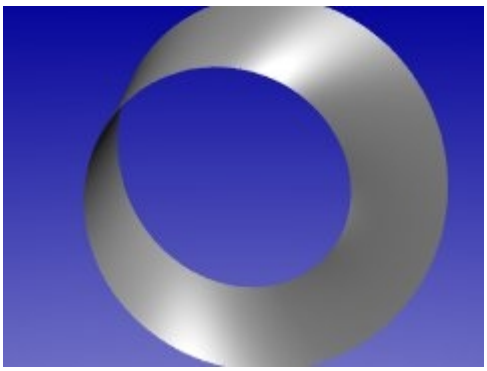
This surface is called the Bohemian Dome.

```
#declare A = 0.5;
#declare B = 1.5;
#declare C = 1.0;
#declare Fx = function {A*cos (u)}
#declare Fy = function {B*cos (v) + A*sin (u)}
#declare Fz = function {C*sin (v)}
parametric {
    function {Fx (u, v, 0)}
    function {Fy (u, v, 0)}
    function {Fz (u, v, 0)}
    <0, 0>, <2*pi, 2*pi>
    contained_by {box {<-R, -R, -R>, <R, R, R>}}
    precompute 18, x, y, z
```



This is part of Dini's Surface of constant negative curvature.

```
#declare A = 1;
#declare B = 0.2;
#declare Fx = function {A*cos (u)*sin (v)}
#declare Fy = function {A*sin (u)*sin (v)}
#declare Fz = function {A*(cos (v) + ln (tan (v/2))) + B*u}
parametric {
  function {Fx (u, v, 0)}
  function {Fy (u, v, 0)}
  function {Fz (u, v, 0)}
      <2*pi, 0>, <4*pi, 2.1*pi>
  contained_by {box {<-R, -R, -R>, <R, R, R>}}
  precompute 18, x, y, z
```

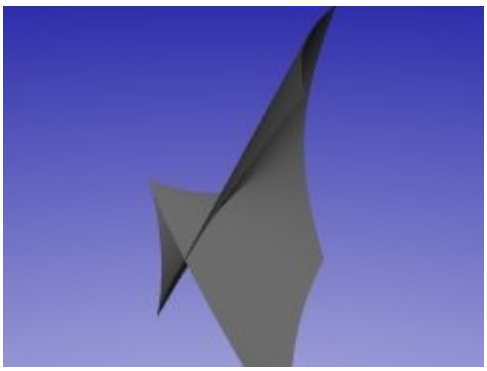


This is a Moebius Strip.

The range of "u" values controls how far round the circle we go (0 to 2*pi is a complete circle) and the range of "v" values controls the width of the strip.

```
#declare Fx = function {cos(u) + v*cos(u/2)*cos (u)}
#declare Fy = function {sin (u) + v*cos (u/2)*sin (u)}
#declare Fz = function {v*sin (u/2)}
```

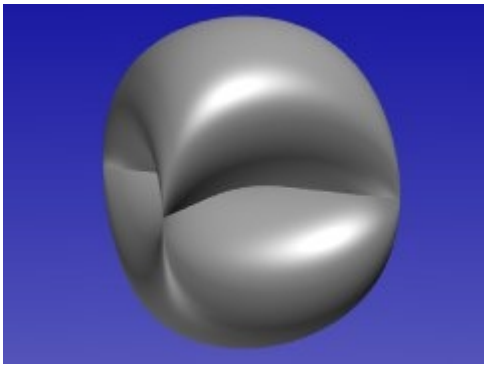
```
#declare U1 =  0;
#declare U2 =  2*pi;
#declare V1 = -0.3;
#declare V2 =  0.3;
parametric {
  function {Fx (u, v, 0)}
  function {Fy (u, v, 0)}
  function {Fz (u, v, 0)}
      <U1, V1>, <U2, V2>
  contained_by {box {<-R, -R, -R>, <R, R, R>}}
```



The swallowtail is the second simplest catastrophe surface.

```
#declare Fx = function {u*v*v + 3*pow (v, 4)}
#declare Fy = function {-2*u*v – 4*pow(v, 3)}
#declare Fz = function {u}

#declare U1 = -2;
#declare U2 =  2;
#declare V1 = -0.8;
#declare V2 =  0.8;
parametric {
  function {Fx (u, v, 0)}
  function {Fy (u, v, 0)}
  function {Fz (u, v, 0)}
      <U1, V1>, <U2, V2>
  contained_by {box {<-R, -R, -R>, <R, R, R>}}
```

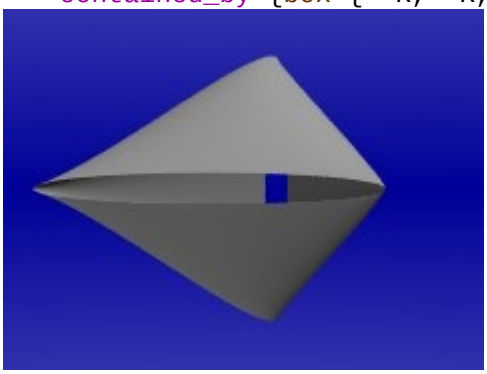
Several of these surfaces have been shamelessly nicked from http://www.uib.no/People/nfytn/mathgal.htm. (Dead link – use: https://web.archive.org/web/20051025061902/http://www.uib.no/People/nfytn/mathgal.htm) You might like to take a look at the Pov 3.1 code on that site that was used to generate some of the images there. Now that we have parametric isosurfaces, we can specify a surface in a few lines that used to take yards of mathematics to specify with smooth triangles.



It's quite easy to invent new parametric surfaces, and they're more likely to look interesting than new conventional isosurfaces are.

If you use combinations of cos and sin functions you can ensure that your surface always remains within a certain distance of the origin, or that it only goes off to infinity in one dimension. For this surface that I just invented Fx, Fy and Fz are all trig functions that are never outside the range -1 to +1, so the complete surface must lie inside the unit cube.

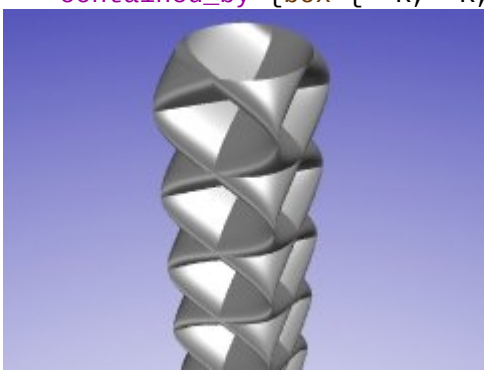
```
#declare Fx = function {sin (u)*sin (v)}
#declare Fy = function {cos (u)*sin (v)}
#declare Fz = function {cos (u)*cos (v)}

#declare U1 = -1*pi;
#declare U2 =  1*pi;
#declare V1 = -1*pi;
#declare V2 =  1*pi;
parametric {
    function {Fx (u, v, 0)}
    function {Fy (u, v, 0)}
    function {Fz (u, v, 0)}
    <U1, V1>, <U2, V2>
    contained_by {box {<-R, -R, -R>, <R, R, R>}}
```



For this surface, I've allowed Fz to go off to go off to infinity, but I've clipped the surface by limiting v to the range -1.4 to +1.4.

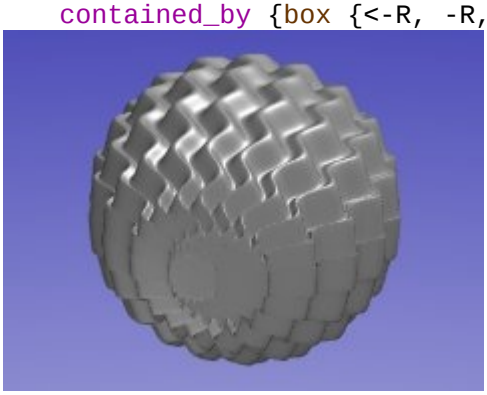
```
#declare Fx = function {sin (u)}
#declare Fy = function {cos (u + v)}
#declare Fz = function {v}

#declare U1 = -pi;
```

```
#declare U2 =  pi;
#declare V1 = -1.4;
#declare V2 =  1.4;
parametric {
    function {Fx (u, v, 0)}
    function {Fy (u, v, 0)}
    function {Fz (u, v, 0)}
    <U1, V1>, <U2, V2>
    contained_by {box {<-R, -R, -R>, <R, R, R>}}
```



It's possible to arrange for the surface to go off to infinity in only one direction (in this case the negative y direction) by using the abs() function or by squaring the equation.

```
#declare Fx = function {sin (u)}
#declare Fz = function {cos (u + v)}
#declare Fy = function {-abs (v)/2}

#declare U1 = -pi;
#declare U2 =  pi;
#declare V1 = -40;
#declare V2 =  40;
parametric {
    function {Fx (u, v, 0)}
    function {Fy (u, v, 0)}
    function {Fz (u, v, 0)}
    <U1, V1>, <U2, V2>
    contained_by {box {<-R, -R, -R>, <R, R, R>}}
```



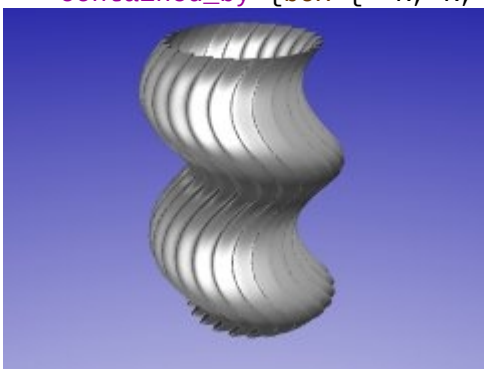
The parametric form of the sphere is:

```
x = sin (u)*sin (v)
y = cos (u)*sin (v)
z = cos (v)
```

To these equations I've added "+cos(20*v)*0.05" to the x and y equations. The "20*v" controls the frequency of the ripples, and the "*0.05" factor controls their amplitude.

Numerous different ripple effects can be achieved by using "sin" instead of "cos", "20*u" instead of "20*v", and applying perturbations to different combinations of Fx, Fy Fz.

```
#declare Fx = function {sin (u)*sin (v) + cos (20*v)*0.05}
#declare Fy = function {cos (u)*sin (v) + cos (20*u)*0.05}
#declare Fz = function {cos (v)}

#declare U1 = -1*pi;
#declare U2 =  1*pi;
#declare V1 = -1*pi;
#declare V2 =  1*pi;
parametric {
    function {Fx (u, v, 0)}
    function {Fy (u, v, 0)}
    function {Fz (u, v, 0)}
    <U1, V1>, <U2, V2>
    contained_by {box {<-R,-R,-R>,<R,R,R>}}
```



Here's a similar trick performed with the parametric form of Helix2.

The first term of Fx and Fy makes a circle of radius r2. The second term offsets that circle by a distance r1 in a direction that spirals round as u varies by 1/Turns. The third term creates the fluting effect.

```
#declare Turns = 3;
#declare r1 = 0.3;
#declare r2 = 1.0;
#declare Flute = 0.04;
#declare Freq = 24;
#declare Fx = function {r2*cos (v) + r1*sin (u*Turns)
              + Flute*sin (Freq*v)}
#declare Fy = function {u}
#declare Fz = function {r2*sin (v) + r1*cos (u*Turns)
              + Flute*sin (Freq*v)}

#declare U1 = -1*pi;
#declare U2 =  1*pi;
#declare V1 = -1*pi;
#declare V2 =  1*pi;
parametric {
  function {Fx (u, v, 0)}
```

```
  function {Fy (u, v, 0)}
  function {Fz (u, v, 0)}
      <U1, V1>, <U2, V2>
  contained_by {box {<-R, -R, -R>, <R, R, R>}}
```

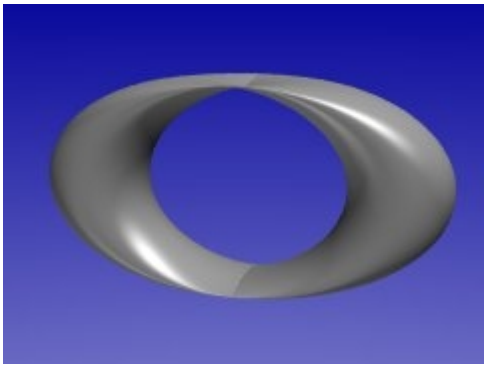
Param.inc

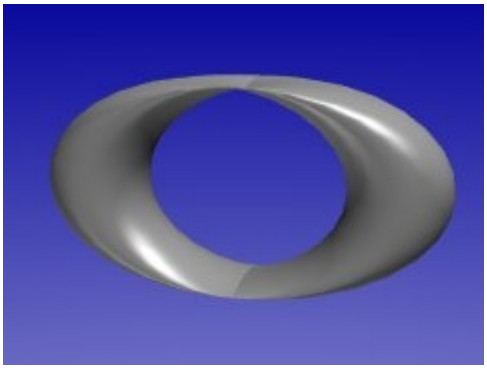
Mesh2 vs parametrics

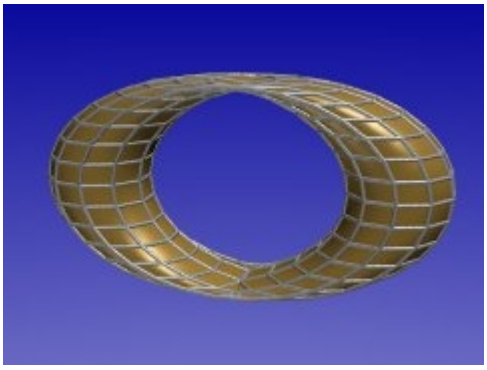
Parametric equations are very slow in POV 3.5, but Ingo Janssen has produced an include file which accepts a syntax quite similar to that of parametric equations and produces a mesh2 object. Ingo's technique can usually produce images that are indistinguishable from real parametric surfaces in a fraction of the time.

Param.inc and associated files can be found on Ingo's webpage. There are several advantages to this technique:

- It's very much faster.
- The mesh2 data can be written to a file, so it need only be parsed once making it even faster to re-render the scene or to have more than one copy of the surface in a scene.
- The mesh2 data is prepared with UV mapping information, so a UV mapped texture can be applied. This is not the case for parametric surfaces or isosurfaces.
- param.inc can be passed macros instead of functions. This allows the use of the float functions and vector functions that are not available in used defined functions.







The top image was created as a POV 3.5 parametric surface. On my machine it took over 12 minutes to render.

The second image was created with the "param.inc" file as a mesh2 object. On my machine it took 5 seconds to render. I can't tell the difference between the resulting images.

I've added a grid to the third image to show the edges of the mesh.

The syntax of the POV 3.5 parametric isosurface is:

```
#declare Fx = function {A*cos (u)}
#declare Fy = function {B*cos (v) + A*sin(u)}
#declare Fz = function {C*sin (v)}
parametric {
    function {Fx (u, v, 0)}
    function {Fy (u, v, 0)}
    function {Fz (u, v, 0)}
    <0, 0>, <2*pi, 2*pi>
    contained_by {box {<-R, -R, -R>, <R, R, R>}}
    precompute 18, x, y, z}
```

The syntax for the mesh2 object is:

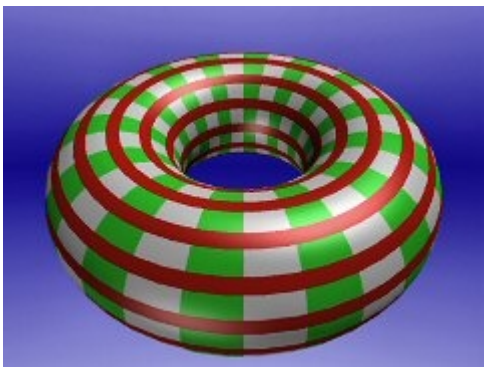
```
#declare Fx = function (u, v) {A*cos (u)}
#declare Fy = function (u, v) {B*cos (v) + A*sin (u)}
```

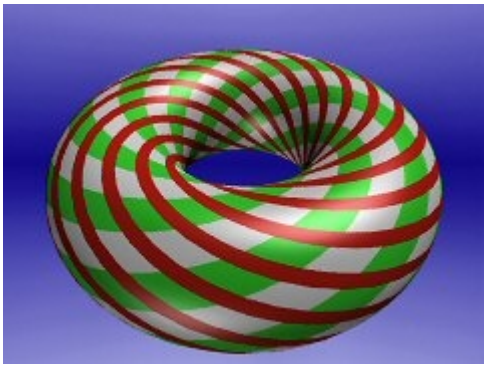
```
#declare Fz = function (u, v) {C*sin (v)}
#include "param.inc"
object {
    Parametric (Fx, Fy, Fz, <0, 0>, <2*pi, 2*pi>, 30, 30, "")
}
```

When used this way, param.inc requires

- The three parametric functions Fx(), Fy() and Fz(), each of which must be functions of two variables.
- The min and max values of u and v.
- The number of steps into which the u and v ranges are divided.
- Optionally a filename for saving/loading the mesh data.

For details of the options, read the comments inside Ingo's param.inc file.





Here's an example of one way of using uv mapping on a mesh2 created with param.inc.

The first surface is a simple torus

```
#declare Fx = function (u, v) {cos (u)*(R1 + R2*cos (v))}
#declare Fy = function (u, v) {sin (u)*(R1 + R2*cos (v))}
#declare Fz = function (u, v) {R2*sin (v)}
```

I've used a layered texture. The bottom layer is green and white stripes in the u direction, and the top layer is red and white stripes in the v direction. These are actually the same u and v that occur in the functions that specify the surface.

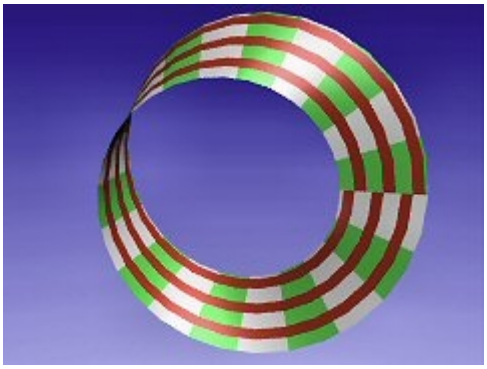
In this case, u and v are both in the range 0 to 2*pi, but I want an integral number of stripes (otherwise I get one stripe with a different width) so I've multiplied the texture parameters by 7/pi to get 14 stripes instead of 2*pi stripes.

```
texture {
    pigment {
        uv_mapping
        function {u*7/pi}
        colour_map {
            [0.5 rgb y]   // y = <0, 1, 0> [BW]
            [0.5 rgb 1]}
    }
  }
texture {
    pigment {
        uv_mapping
        function {v*7/pi}
        colour_map {
            [0.4 rgb x]   // x = <1, 0, 0> [BW]
            [0.4 rgbt 1]}
    }
    finish {
        phong 0.5
        phong_size 10
    }
}
```

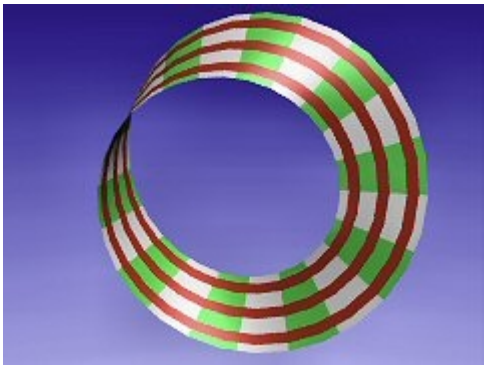
In the second example, the textures are still in the u and v directions. It's the actual surface that's twisted.

```
#declare Fx = function (u, v) {cos (u - v) * (R1 + R2*cos (v + u))}
#declare Fy = function (u, v) {sin (u - v) * (R1 + R2*cos (v + u))}
#declare Fz = function (u, v) {R2*sin (v + u)}
```

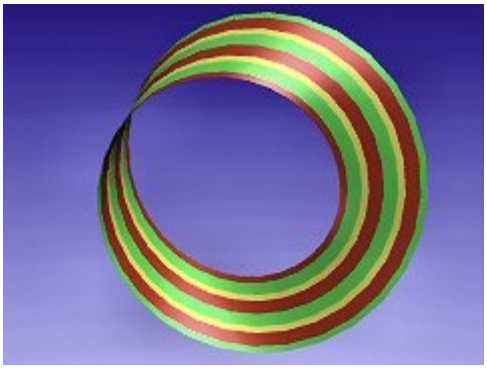
Without the uv mapped textures, these two surfaces look identical.



Moebius strip with discontinuous pigment



Continuous symmetrical pigment



Continuous asymmetric pigment If the object you are attempting to uvmap has a place where the inside meets the outside, like in a Moebius strip, then you may see a discontinuity in the mapping. To fix this, you could use a texture that is symmetrical, like this:

```
pigment {
    uv_mapping
    function {v*3/0.6}
    colour_map {
        [0.2 rgb x] [0.2 rgbt 1]
        [0.8 rgbt 1][0.8 rgb x]
    }
}
```

Or, if you really want an asymmetric texture you could paint the interior_texture with a copy of the same texture that is inverted. The easiest way to do this is to apply **scale -1** to the interior_texture. In some cases it may be necessary to use **scale <-1,1,1>** or **scale <1,-1,1>**

Approximation Macro

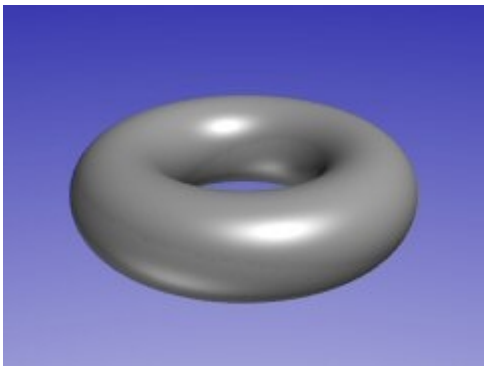
Kevin Loney's Approximation Macro

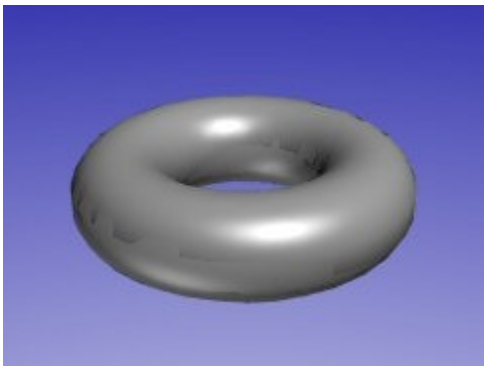
Kevin Loney has produced an include file which produces a mesh object that approximates an isosurface. This technique produces object which renders faster than real isosurfaces, but usually takes much longer to parse.

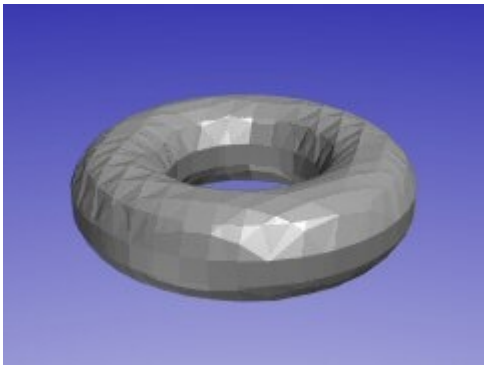
Kevin's isosurface.inc can be found on Kevin's webpage.

The mesh data can be written to a file, so it need only be parsed once making it faster to re-render the scene or to have more than one copy of the surface in a scene.

In the vast majority of cases, the total time required to parse and render the approximation is greater than the time taken to render a real isosurface, but there could well be situations where using the approximation could be faster, for example if the same isosurface were to appear in each frame of an animation it would only need to be parsed once and then the data could be read from the file for all the subsequent frames. This might be useful for Mechsim animations - the real isosurface would be used for the simulation calculations, but the approximation could be used for the actual rendering.







The top image was created as a POV 3.5 isosurface.

The second image was created as an approximation.

The third image shows how the approximation is constructed from a mesh of triangles.

The last image shows the same approximation but with a mesh of flat triangles instead of smooth_triangles

The syntax of the POV 3.5 isosurface is:

```
#declare f = function {f_torus (x, y, z, 1, 0.4)}
#declare isoMin = <-R, -R, -R>
#declare isoMax = <R, R, R>
    isosurface {
    function { f (x, y, z)}
    max_gradient 1.1
    contained_by {box {isoMin,isoMax}} open
  }
```

The syntax for the approximation is:

```
#declare f = function {f_torus (x, y, z, 1, 0.4)}
#declare isoMin = <-R, -R, -R>
#declare isoMax = <R, R, R>
```

```
#declare isoSmooth = yes;
#declare isoSegs = <15, 15, 15>
#declare isoFileOption = 1;
#declare isoFile = "KL01.iso";
#declare isoName = "Surface";
#include "isosurface.inc"
object {Surface}
```

The possible parameters for the macro are:-

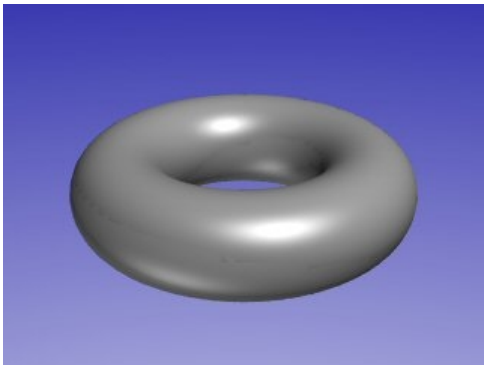
- f() The isosurface function.
- isoMin and isoMax The corners of the bounding box. If not supplied, the unit cube <-1,-1,-1><1,1,1> is used.
- isoSegs The number of segments to be used in the x, y and z directions. If not supplied, <10,10,10> is used. Higher values give better approximations, but take longer to parse.
- isoFileOption Set this to 1 to generate the approximation and store the mesh in a file. Set it to 0 to read the data from a previously generated file. Set it to 2 to generate the approximation but not store the data in a file, in which case the syntax for using the mesh is slightly different as the result is an inline mesh rather than being #declared.
- isoFile The name of the file for the mesh data.
- isoName The name to be used for the generated mesh.
- isoSmooth Set this true for a smooth_triangle mesh. Set it false for unsmooth triangles. If not supplied, the triangles will be unsmoothed, as shown in the fourth image.
- isoThreshold The value for the "threshold" parameter of the isosurface. If not supplied, the threshold is set to 0.
- isoNormal This controls the Gradient Function used for vector analysis operations called from "math.inc" which are used for calculating the normals for smooth_triangles. In most cases, this value has no noticeable effect on the quality of the image or on the time taken.

The result is returned as a mesh which is #declared with the name that you specified in isoName, except when isoFileOption is set to 2, in which case the mesh is actioned immediately rather than being #declared.

The syntax when using isoFileOption = 2 is like this:

```
#declare f = function {f_torus (x, y, z, 1, 0.4)}
#declare isoMin = <-R, -R, -R>
#declare isoMax = <R, R, R>
#declare isoSmooth = yes;
#declare isoSegs = <15, 15, 15>
#declare isoFileOption = 2;
object {
    #include "isosurface.inc"
    pigment {rgb 1}
}
```

There is now also a modified version of this macro by Jaap Frank, which runs nearly twice as fast.



The parameters for the Jaap Frank version of the macro are slightly different in that the isosurface function is called **Fn()** instead of **f()**.

In the lastest version of the macro there's an additional Depth parameter. The macro performs surface subdivision on cells which contain part of the surface, and skips cells which don't.

For example, rendering this example file without subdivision, like this

```
#declare isoSegs = <32, 32, 32>
#declare Depth = 0;
```

causes the macro to examine all 32768 cells (32*32*32), but only 2664 of those cells contain parts of the surface.

Rendering the example with one level of subdivision, like this

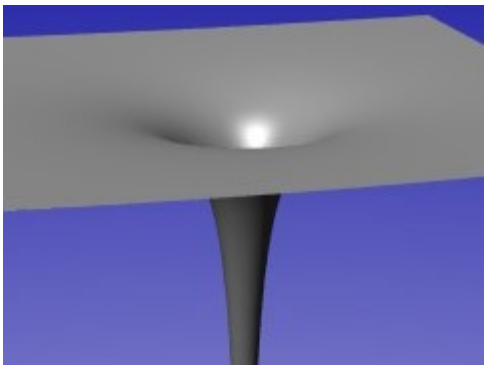
```
#declare isoSegs = <16, 16, 16>
#declare Depth = 1;
```

causes the macro to examine 4096 cells at the initial depth, and find that only 720 of them contain parts of the surface. It then subdivides each of these 720 cells into 8 subcells and examines them and finds that 2664 of the subcells contain parts of the surface. It renders the same 2664 surface fragments as before, but it only needed to examine 9856 cells in order to find them (4096 + 720*8) instead of 32768.

Don't forget to change your isoSegs to smaller values when using subdivision, and be aware that the default Depth value is 2.

All the other parameters are the same as the Kevin Loney version.

New Stuff

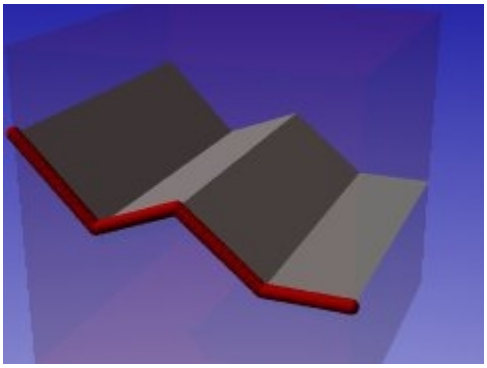


User functions with parameters

It is now possible to declare user functions that have extra parameters.

```
#declare Gravity_Well = function (x, y, z, Strength) {x*x - Strength/y*y + z*z}
```
declares a function that takes an extra "Strength" parameter. We don't tell the function what the value of "Strength" is when we declare it, but only when we reference it

```
isosurface {function {Gravity_Well (x, y, z, 0.002) }
```



```
#declare S = function {
    spline {
    linear_spline
    -1.00, < 0.5, 0, 0>,
    -0.50, < 0.0, 0, 0>,
     0.01, < 0.2, 0, 0>,
     0.50, <-0.2, 0, 0>,
     1.00, <-0.2, 0, 0>
    }
}
isosurface {function {y - S (x).x}
```

Spline Functions

It is now possible to use a spline function to generate an isosurface.

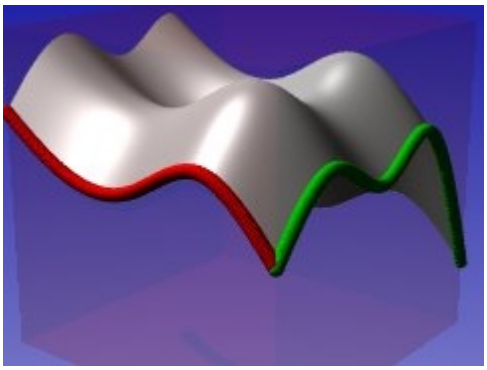
In this case I'm using the first column of the spline only. As x goes from -1 to +1 S(x) goes from <0.5,0,0> to <-0.2,0,0> along the spline.
For an isosurface, we don't want a vector function, so we choose the S(x).x component of the function.

We can see that the isosurface follows the x co-ordinate of the spline, which is shown in red in the attached image.

Warning: There are known bugs in spline functions in POV 3.5b1. For example, if we replace the 0.01 point in the above example by

```
    0.0, < 0.2, 0, 0>,
```

the spline behaves in an unexpected manner.



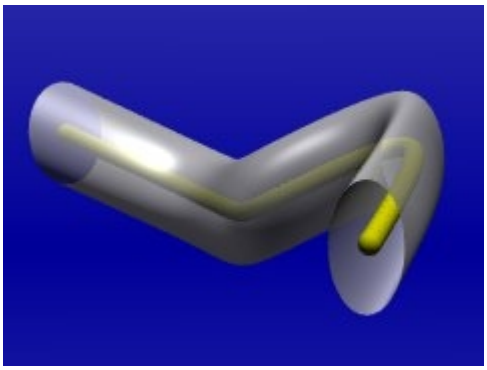
```
#declare S = function {
    spline {
    natural_spline
    -1.00, < 0.5, 0, 0.0>,
    -0.50, < 0.2, 0, 0.4>,
     0.01, < 0.2, 0, 0.2>,
     0.50, < 0.4, 0, 0.4>,
     1.00, < 0.0, 0,-0.6>
    }
 }
isosurface {function {y - S (x).x - S (z).z}
```

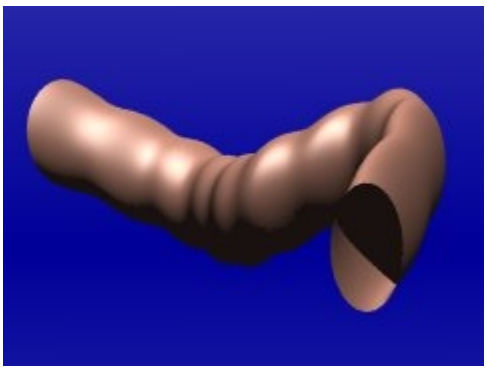
But we're not restricted to using just one dimension of the spline, or to using linear splines.

Here's a natural spline function. We can see that the surface follows both the x and z co-ordinates of the spline, which are shown in red and green respectively in the attached image.

In this way, it's possible to generate a 3d isosurface sheet from two 2d splines.

The new POV 3.5 "cubic_spline" is not very suitable for isosurface work because it can be undefined at the endpoints, which causes strange effects in the surface.





Another thing that we can do with spline functions is to sweep along a 3d spline, in a similar manner to a sphere_sweep.

```
#declare S = function {
    spline {
    natural_spline
    -1.00, < 0, 0.5, 0.0>,
    -0.50, < 0, 0.2, 0.4>,
     0.01, < 0, 0.2, 0.2>,
     0.50, < 0, 0.4, 0.4>,
     1.00, < 0, 0.0,-0.6>
    }
}
isosurface {function {pow (y - S (x).y), 2) + pow(z - S (x).z, 2) - 0.05}
```

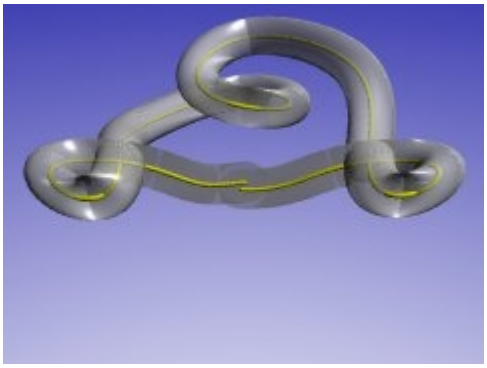
What you get isn't exactly the same as a sphere_sweep, it's more like a "hoop sweep" with the hoop always oriented in the same direction rather than turning as the spline turns.

It's also possible to write it in a way that more clearly separates the "hoop" function from the spline function, making it easier to replace the circular hoop with some other shape.

```
#declare Hoop = function (x, y, z, r) {y*y + z*z - r*r}
isosurface {function {Hoop (x, y-S (x).y, z-S (x).z, 0.223)}
```

In the upper image, I've made the isosurface slightly transparent and indicated the path of the spline itself in yellow.

In the lower image I've added a ripple to the isosurface, so now you can drape intestines along a 3d spline path.



```
parametric {
    function {S (u).x + sin (v)}
    function {S (u).y + cos (v)}
    function {S (u).z}
    <0, -pi> <17, pi>
    contained_by {box
        {min_extent (The_Path) - <1, 1, 0.1>
         max_extent (The_Path) + <1, 1, 0.1>}
    }
```

But that previous technique doesn't work with splines that have loops in them. The problem is that we were using the spline control points as our x co-ordinate and expressing y and z in terms of that x. The sequence of control points can't loop back on itself.

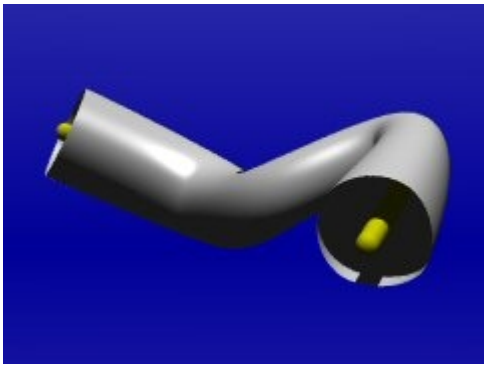
The way to follow 3d splines that contain loops is to use all three of the spline dimensions to specify the path of the spline, and express x, y, and z in terms of a fourth variable, call it "u", which follows the control points.

So what we need is a parametric isosurface.

In this case "u" follows the spline control points, and x, y, z are expressed in terms of u like S(u).x, S(u).y, S(u).z. On its own that would give an infinitely thin string that follows the spline path. We can fatten it out by adding some terms in a second variable "v" which describe how far the surface is from the path.

Once again I have made the isosurface transparent and drawn the actual spline path inside it in yellow.

For this image, I didn't know how large to make the contained_by box, so I've calculated it during the parsing of the scene by taking the min_extent() and max_extent() vectors of the yellow spline path and added something to allow for the thickness of the wrapping.

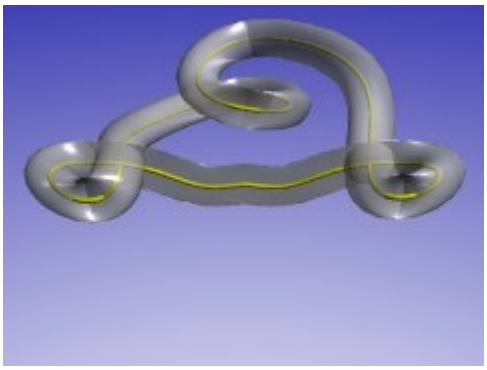


You may have noticed that in the previous examples, the swept shape always faces in the same direction. In the attached zip file I've included the SweepSpline macro that sweeps a shape in such a way that it turns to always be perpendicular to the spline that it is being swept along.

It's not actually an isosurface at all, but a mesh. I just stuck it here since it's doing a similar job to some of the isosurface examples on this page.

Parametric spline functions

I've created all the images on this page both with Ingo's param.inc macro. (See param.inc for details), and more slowly, by using real parametric isosurfaces. The zip file at the end of the page contains both sets of source files.



```
#declare Fx = function (x, y) {S (u).x + sin (v)}
#declare Fy = function (x, y) {S (u).y + cos (v)}
#declare Fz = function (x, y) {S (u).z}
#include "param.inc"
object {Parametric(Fx, Fy, Fz, <0, -pi>, <17, pi>, 100, 20, "")
    pigment {rgbt <1, 1, 1, 0.4>}
    finish {phong 0.5 phong_size 10}
    no_shadow
}
```

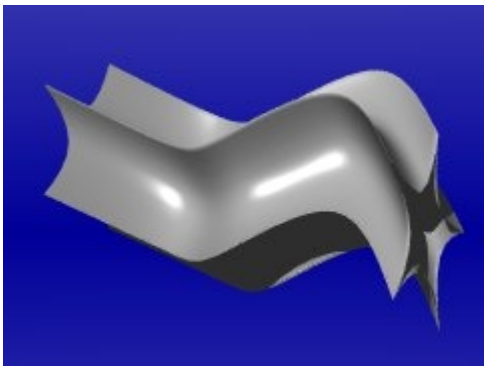
The spline technique shown on the previous page doesn't work with splines that have loops in them. The problem is that we were using the spline control points as our x co-ordinate and expressing y and z in terms of that x. The sequence of control points can't loop back on itself.

The way to follow 3d splines that contain loops is to use all three of the spline dimensions to specify the path of the spline, and express x, y, and z in terms of a fourth variable, call it "u", which follows the control points.

So what we need is a parametric isosurface.

In this case "u" follows the spline control points, and x, y, z are expressed in terms of u like S(u).x, S(u).y, S(u).z. On its own that would give an infinitely thin string that follows the spline path. We can fatten it out by adding some terms in a second variable "v" which describe how far the surface is from the path.

Once again I have made the isosurface transparent and drawn the actual spline path inside it in yellow.

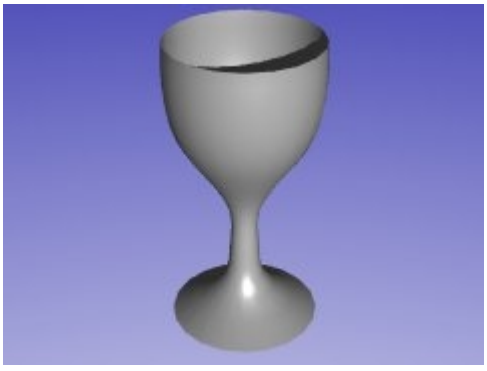


```
#declare Fx = function (x, y) {u}
#declare Fy = function (x, y) {S (u).y + S2 (v).y}
#declare Fz = function (x, y) {S (u).z + S2 (v).z}
```

Bent prism

This parametric isosurface is equivalent to taking a prism object and bending it to follow an open spline path.

In this case, the prism is a five pointed star with cubic interpolation, given by the spline function S2(). The curved path along which the prism is bent is given by the spline function S().



```
#declare Fx = function (x, y) {(S (u).x * sin (v)/2)}
#declare Fy = function (x, y) {u}
#declare Fz = function (x, y) {(S (u).x * cos (v)/2)}
```

Lathe

The following examples will be based on this goblet.

I could have created this goblet as a lathe object, as a sor object or I could have used a non-parametric isosurface. However, I've used a parametric isosurface so that I can perform the alterations shown in the later examples.

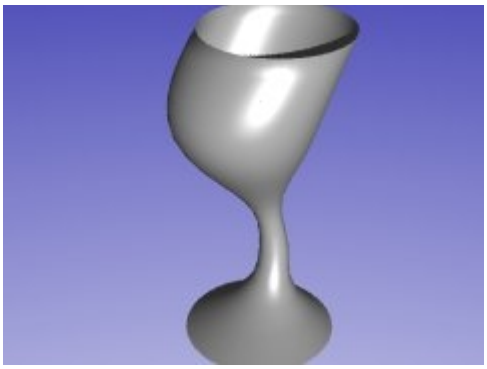
The spline function S() is a one-dimensional spline which gives the profile of the goblet.

```
#declare Fx = function (x, y) {u}
#declare Fy = function (x, y) {S (u).y * S2 (v).y}
#declare Fz = function (x, y) {S (u).y * S2 (v).z}
```

Prismatic lathe

This example is a combination of a prism and a lathe.

The profile of the goblet is given by the one-dimensional spline function S(), and the star-shaped cross section is given by the two-dimensional spline function S2().

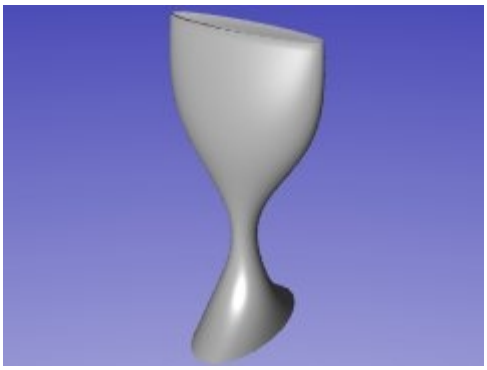


```
#declare Fx = function (x, y) {(S (u).x * sin (v)/2) + S2 (u).x}
#declare Fy = function (x, y) {u}
#declare Fz = function (x, y) {(S (u).x * cos (v)/2) + S2 (u).z}
```

Bent lathe

In the same way that a prism can be bent along a curved spline, it's possible to bend a lathe.

The profile of the goblet is given by the one-dimensional spline function S(), and it is then bent along the two-dimensional spline S2().

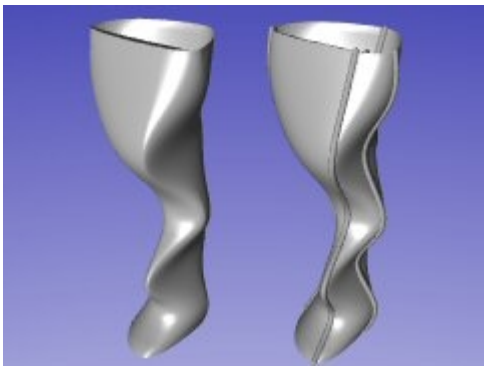


```
#declare Fx = function (x, y) {(S (u).x * sin (v)/2)}
#declare Fy = function (x, y) {u}
#declare Fz = function (x, y) {(S (u).z * cos (v)/2)}
```

Two-way lathe

In this scene, the object almost looks like a lathe, but it has a different profile in the x direction from the z direction.

The x profile is given by the x component of the two-dimensional spline function S(), and the z profile is given by the z component of the same spline function.



Four-way lathe

It's possible to extend this idea to a shape that uses four different splines to control the profile in the +x, -x, +z and -z directions.

This is achieved by mutiplying one x spline function by (v<=pi) and the other by (v>pi). The comparison operators return 0 for false and 1 for true, so the sum switches from being one spline to the other when v becomes pi.

Similarly the z spline functions are multiplied by (cos(v)<=0) and (cos(v)>0), which switch when v=pi/2 and switch back when v=3*pi/2.

```
#declare Fx = function (x,y) {sin(v)/2 *(S(u).x*-(v<=pi) + S2(u).x*-(v>pi)) }
#declare Fy = function (x,y) {u}
#declare Fz = function (x,y) {cos(v)/2 *(S(u).z*(cos(v)<=0) + S2(u).z*(cos(v)>0))}
```
I've drawn in the four splines on one copy of the surface.

Patterns and Noise

```
function {y-f_wood (x*2, 0, z*2)*0.2}
```

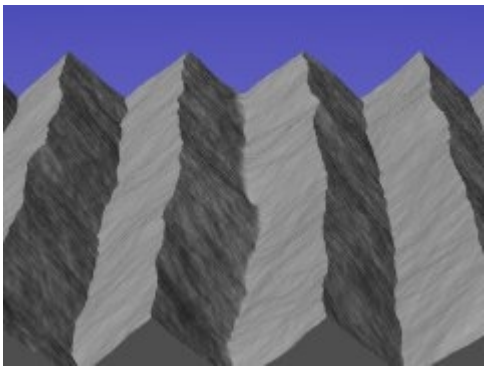
Pattern Functions

It's now possible to generate isosurfaces from patterns in a similar way to the way they can be generated from pigments. Generating directly from patterns is more efficient, since the pattern only generates one float value for each point in space, and that's all we need. When we use a pigment, the underlying pattern is used together with a colour_map to generate a colour vector and then we only use one component of that vector to generate the isosurface. By generating the isosurface directly from the pattern we cut out the middle step.

```
#declare P =
function {
    pigment {wood
        colour_map {
            [0 rgb 0]
```

```
        [1 rgb 1]
        }
    }
}
isosurface {function {y - P(x*2, 0, z*2).grey*0.2}
```

This is how we would generate the same surface from a pigment. The result is the same, but the processing is slightly less efficient. The difference in speed is only about 5% and there's no difference in the memory usage, so there's no real need to use pattern functions if you're happy with pigment functions.

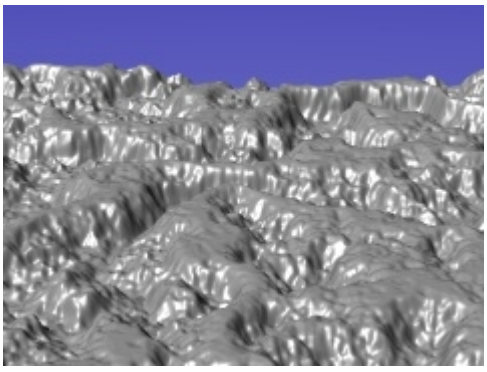


```
#declare P =
function {pattern {wood turbulence 0.02}}
isosurface {function {y - P(x*2, 0, z*2)*0.2}}
```

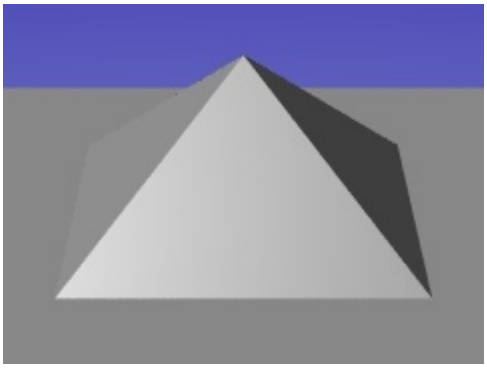
Some pattern functions are built into "functions.inc", as above, but we could just as well define our own pattern functions directly. This allows us to add some pattern modifiers such as turbulence and warps.

Available patterns

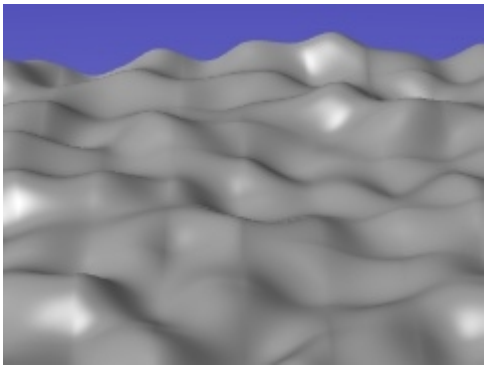
Other patterns, like brick, gradient, mandel, onion, planar and radial contain discontinuities.
At a discontinuity the gradient becomes infinite and the pattern does not work very well as an isossurface.



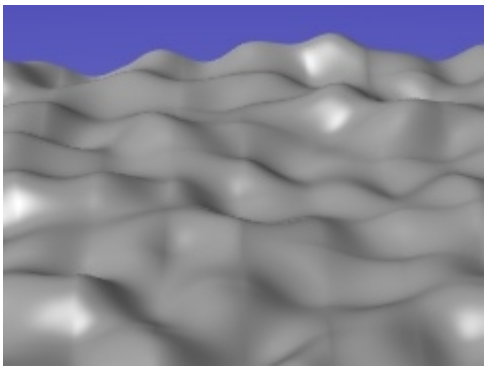
f_agate



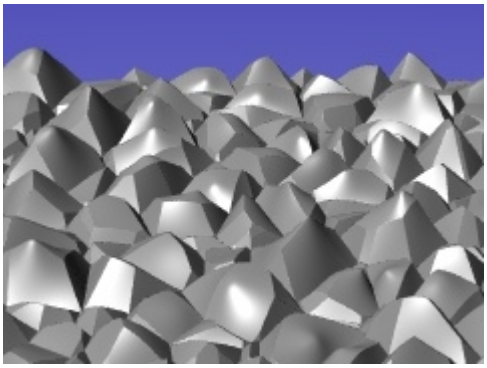
f_boxed



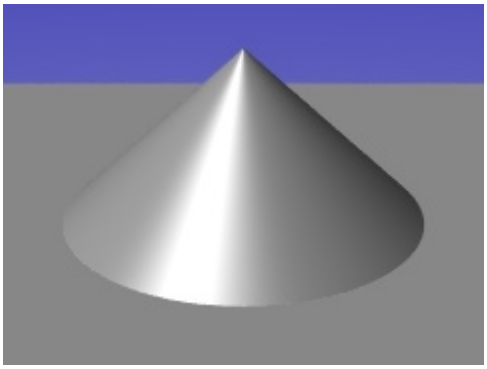
f_bozo



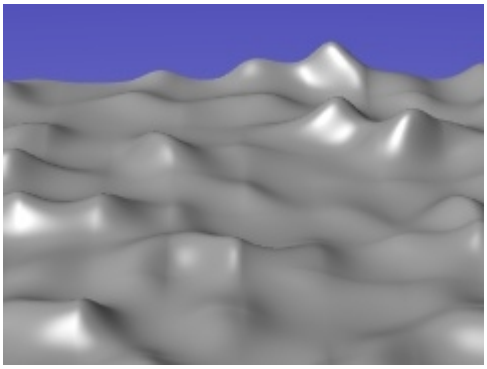
f_bumps



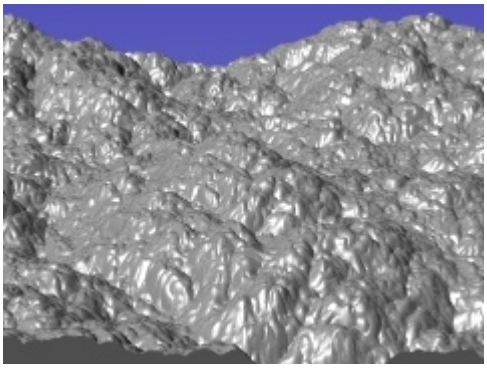
f_crackle



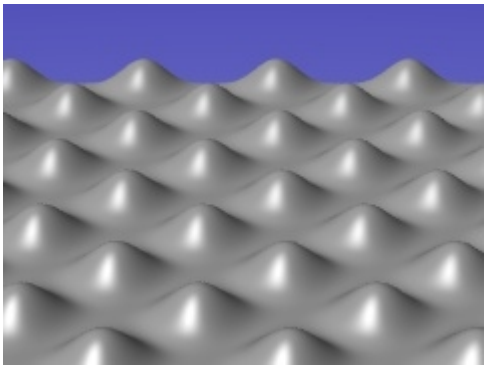
f_cylindrical



f_dents

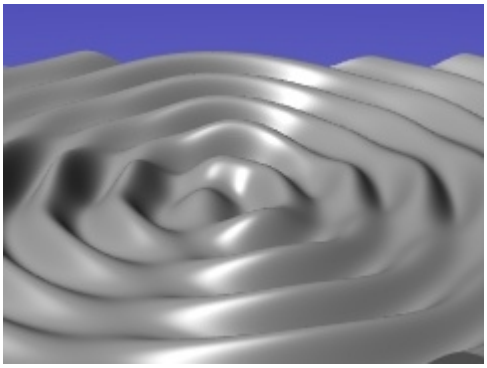


f_granite

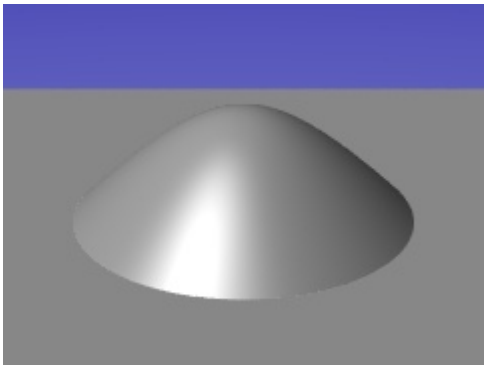


f_leopard

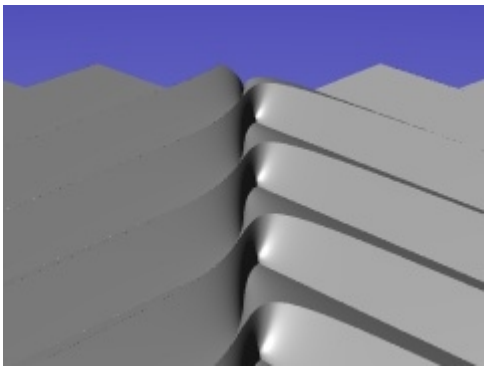
f_marble



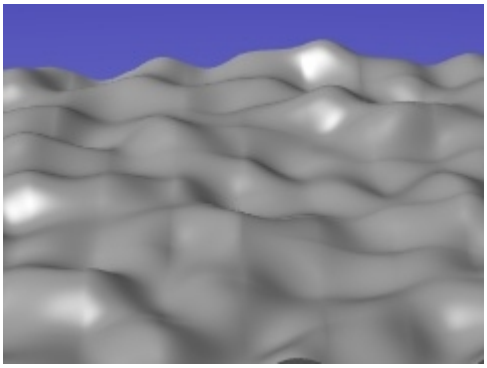
f_ripples



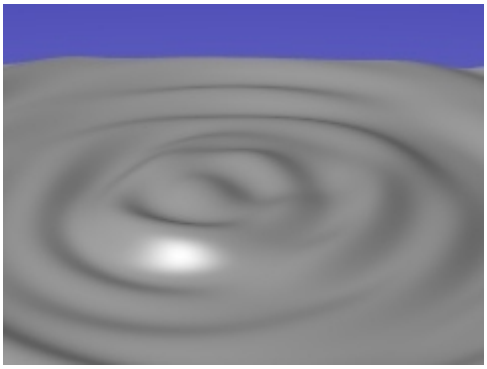
f_spherical



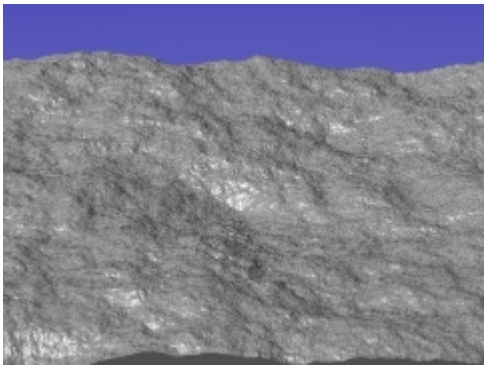
f_spiral1



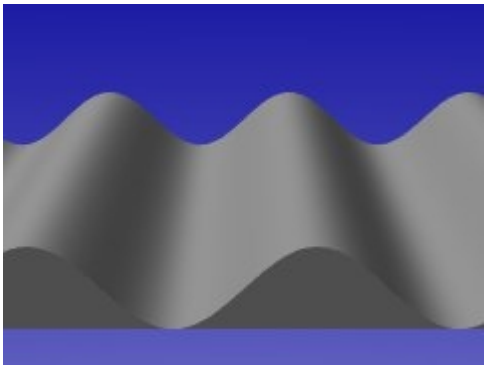
f_spotted



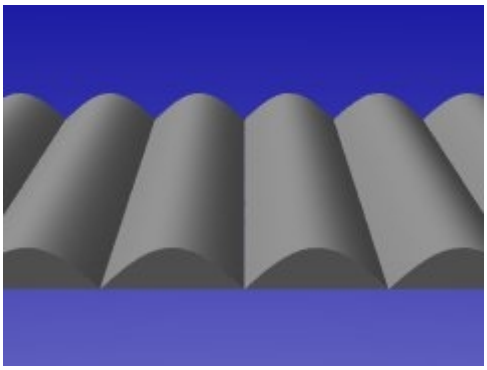
f_waves



f_wrinkles



```
function {y - f_sine_wave (x, 0.1, 3)}
```



```
function {y - f_scallop_wave (x, 0.1, 3)}
```

f_sine_wave and f_scallop_wave

These new functions behave differently to the other built in functions, in that their parameters aren't x,y,z but value,amplitude,frequency.

[If I'd have been creating these functions I would have defined them to be called like "my_sine_wave(x,y,z,amplitude,frequency)" even if the y and z values are not used, for consistency with all the other functions]

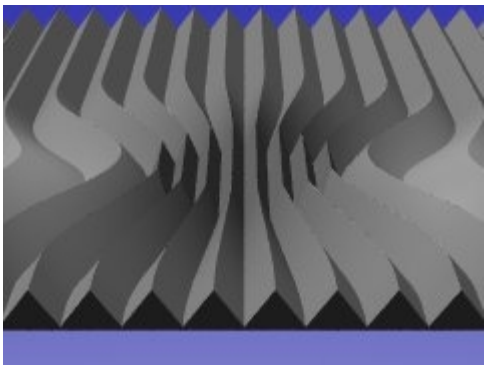
The "value" parameter is expected to be something that increases linearly across the space that you're working in. For example "x" or "z" or "3*x + y". The "amplitude" and "frequency" parameters are expected to be constants.

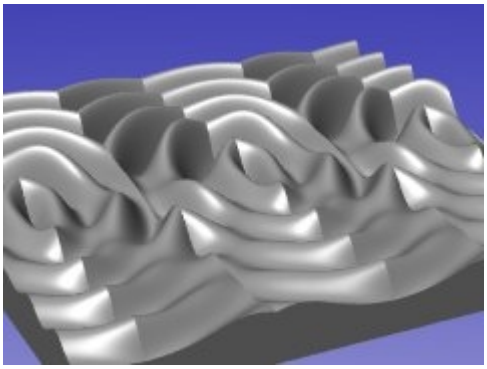
[You don't have to use the expected types of parameters, you could use variables for "amplitude" and "frequency" and a constant for "value" but the result probably won't be a sine_wave or scallop_wave any longer. You can use unexpected things for the parameters of any of the built in functions, e.g. function { f_sphere(x, y, z,

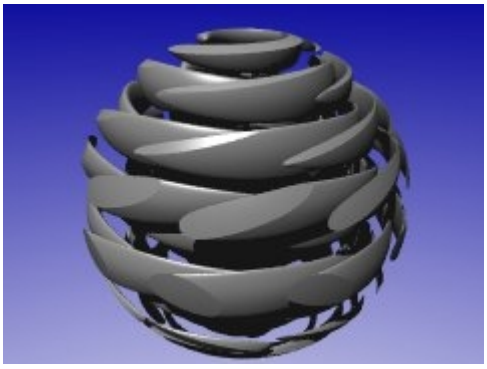
abs(sin(x-y)))} misuses the f_sphere function by specifying a radius that isn't constant, but the result isn't much like a sphere].

f_hexagon

Although f_hexagon is documented as if it were a pattern function, it is actually a pigment function, so the syntax is slightly different. See Pigments as functions.







Warp with Patterns

It's also possible to use warps with patterns.

The top image shows a black hole warp of a wood pattern.

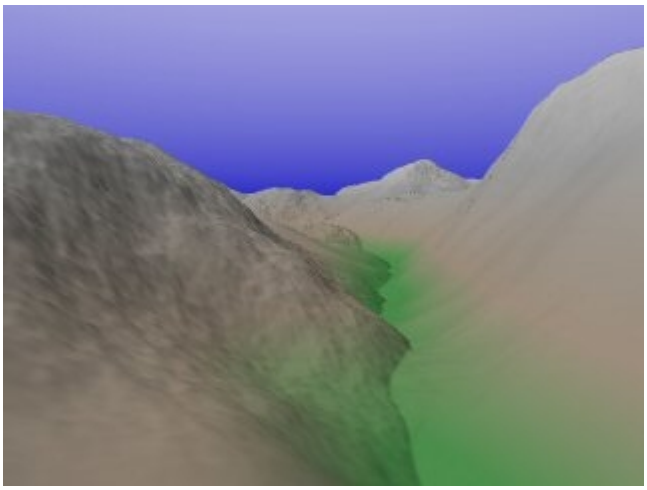
```
#declare F =
function {
    pattern {
        wood scale 0.15
        warp {black_hole 0, 1 strength 2}
    }
}
```

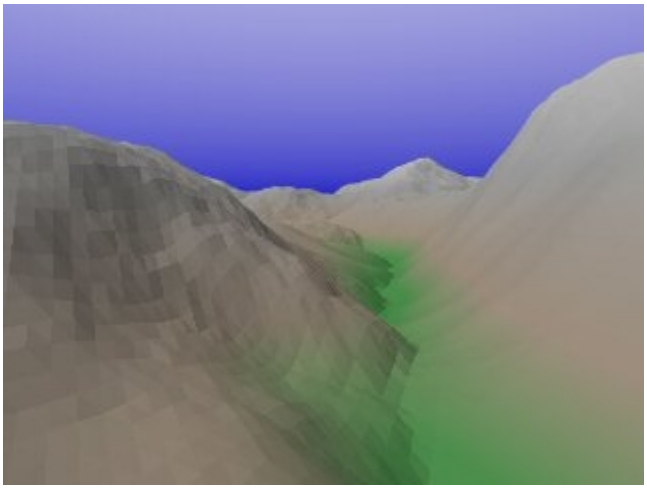
The second image shows a repeat warp of a ripple pattern.

```
#declare F =
function {
    pattern {
        ripples scale 0.2
        rotate y*30
        warp {repeat x*0.4 flip x}
    }
}
```

The third image shows a cylindrical warp of a leopard pattern.

```
#declare F =
function {
    pattern {
        leopard
        warp {
            cylindrical
            orientation y
            dist_exp 1.5
        }
    scale 0.1
    }
}
```





Isosurfaces vs Height Field Patterns

You may have noticed that it's now possible to use functions in height fields. It may well be preferable to use such height fields instead of isosurfaces when creating certain sorts of scenes, particularly landscapes.

The upper image on the left was created with an isosurface

```
#declare P1 = function {pattern {leopard turbulence 0.3 scale 0.05}}
#declare P2 = function {pattern {crackle turbulence 0.3 scale 0.70}}
#declare P  = function {P1 (x,0,z)*0.3 + P2 (x, 0, z)}
    isosurface {
    function {y – P (x, 0, 1-z)}
    contained_by {box {<0, 0, 0>, <1, 1, 1>}}
    translate <-0.5, 0, -0.5>
```

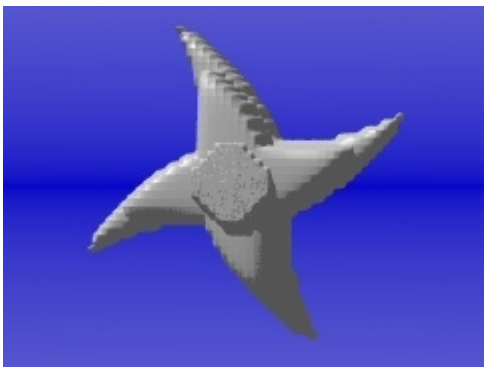
The lower image was created with a height field using the same pattern.

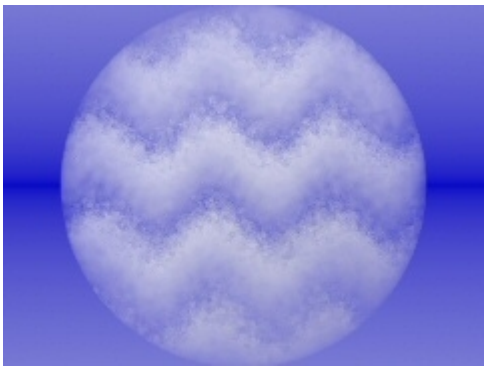
```
height_field {function 300, 300 {P (x, 0, y)} translate <-0.5, 0, -0.5>}
```

The odd looking **P(x,0,1-z)** and **P(x,0,y)** adjust for the different ways that isosurface and height field data is oriented, so that the same parts of the function are used in the same places. This allows you to develop your scene using the fast height field code and then produce a final render using the isosurface code which may be 20 times slower.

The significant differences are

- The height_field renders very much faster
- The resolution of the height_field is defined by the source file. If you render a larger image there will be more fine detail in the isosurface unless you increase the height field resolution.
- The height field uses more memory because the entire height field pattern is rendered and stored in memory.





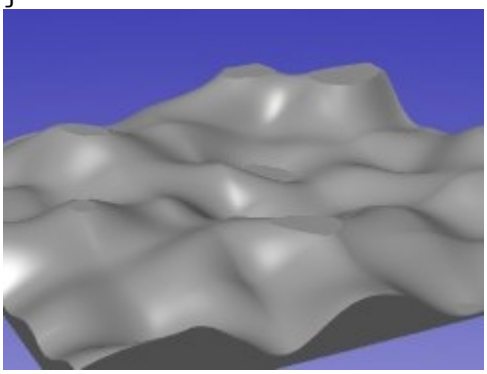
Density patterns

Anything that can be used as a density pattern for media can now be used as an isosurface function. That includes DF3 format files. The upper image on the left uses the spiral.df3 example file that comes with the POVRay distribution.

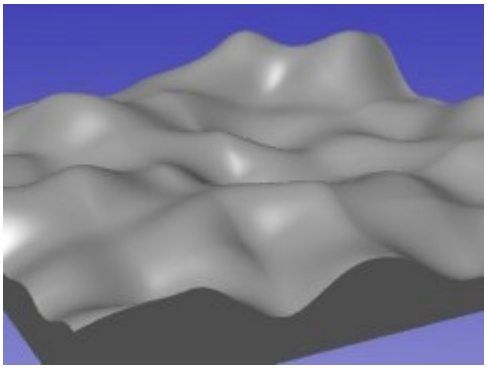
```
#declare F = function {pattern {density_file df3 "spiral.df3" interpolate 1}}
isosurface {function {0.05 - F(x, y, z)}
```

Similarly, any function can be used as a media density pattern.

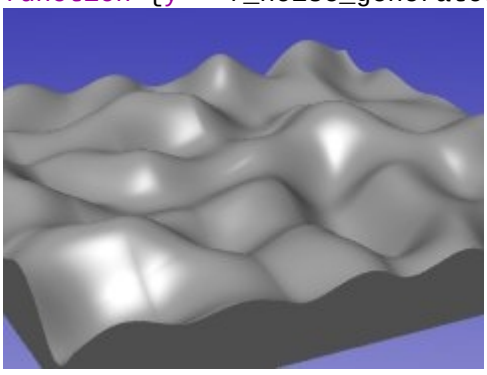
```
media {
    scattering {1, 0.7}
    density {
        function {(sin (x*10)*0.2 + y*2) -  f_noise3d (x*20, y*20, z*20)}
    }
}
```



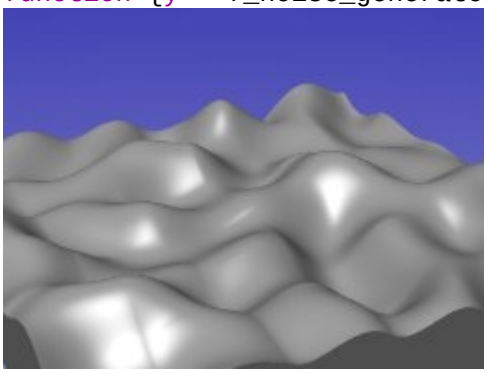
```
function {y – f_noise_generator (x*3, 0, z*3, 1)*0.2}
```



```
function {y – f_noise_generator (x*3, 0, z*3, 2)*0.4}
```



```
function {y - f_noise_generator (x*3, 0, z*3, 3)*0.4}
```



```
function {y - f_snoise3d (x*3, 0, z*3)*0.2}
```

Types of noise

POV-Ray 3.5 has three types of noise.

The old noise algorithm looked OK when used for generating bozo pigments, and bump and dent normals, but when used in an isosurface it becomes clear that there are ugly plateau regions. This is now known as noise type 1. The plateau artefacts are generated when the noise function evaluates to a value outside the region 0 to 1, and the result gets clipped to keep it within that region.

Noise type 2 is the same algorithm as type 1, except that the values are scaled down so that they don't need to be clipped.

Noise type 3 is an entirely new perlin noise algorithm.

Type 3 is default, and it's probably a good idea to stick with it unless you have an image created with a previous version where you want to reproduce the behaviour of the old noise. In which case use type 1 if you want to reproduce the plateau artefacts or type 2 if you just need the bumps in the same places and don't want the plateaux.

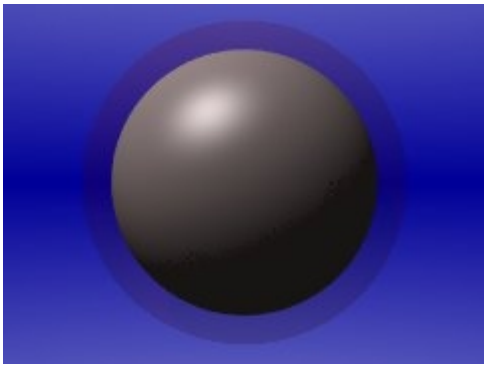
Noise type 1 produces a stronger effect than the other two types (because it is effectively scaled over a range greater than [0,1]) so I've scaled it down in the top image on the left.

The type of noise can either be chosen by using

- f_noise_generator(x, y, z, n)
- noise_generator n as a pattern modifier
- global_settings {noise_generator n} in which case it affects everything that uses the noise algorithm, such as bozo, bumps, dents and f_noise3d().

The f_snoise3d() function returns a noise value in the range [-1,+1] instead of the range [0,+1]. I've reduced the scale to get about the same amount of bumpiness as the other noise functions.

Built In Functions

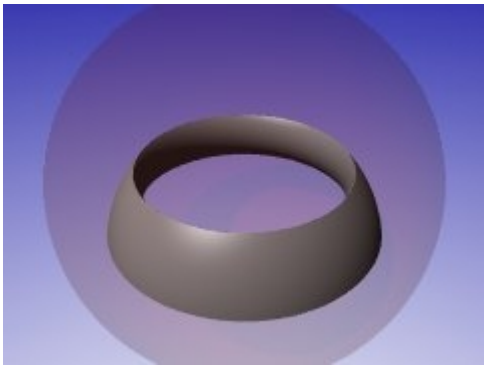


There is a considerable collection of built in functions. The simplest such function is the sphere, which can be used like this:

```
#include "functions.inc"
isosurface {
    function {f_sphere (x, y, z, 1)}
    accuracy 0.001
    contained_by {sphere {0, 1.2}}
    pigment {rgb 0.9}
}
```

Which gives exactly the same results as you would get by using a conventional **sphere {0, 1}** object or using the mathematical isosurface **function { x*x + y*y + z*z - 1 }** .

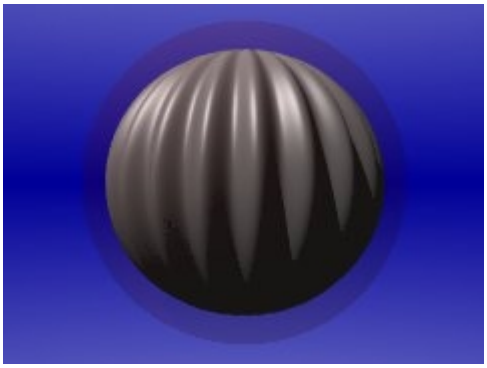
The ,1 is a parameter that gets applied to the built in function. In the case of a sphere, the parameter sets the radius.



To perform the intersection of two built in isosurfaces, we take the max() of the two function, like this

```
function {max (f_torus (x, y, z, 1, 0.3), f_sphere (x, y+0.5, z, 1))}
```

Notice that it also possible to perform variable substitution at the same time. In this case the sphere has been shifted down by using "y+0.5" instead of "y".

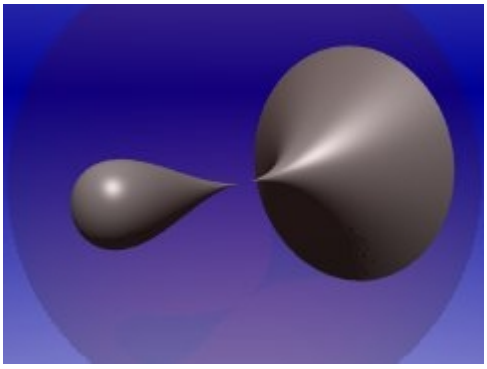


Some built in functions are not surfaces when used alone, and are only useful when combined with other surfaces.
f_th() is one such function. **f_th()** produces a value which is equal to the "theta" angle, in radians, at any point. The theta angle is like the longitude co-ordinate on the Earth. It stays the same as you move north or south, but varies from east to west.

The functions f_r(), f_th() and f_ph() are equivalent to the standard 3d polar co-ordinates Radius, Theta and Phi.

This surface is a sphere to which has been added a height that's proportional to sin(theta), but which also decreases as you get near the poles (otherwise the ridges look out of proportion when the come close together).

```
#declare Theta  = function {f_th (x, y, z)}
#declare Sphere = function {f_sphere (x, y, z, 1)}
isosurface {
    function {Sphere (x, y, z) + sin (Theta (x, y, z)*20)*0.05*(1-y*y) }
}
```



Here's a nice teardrop shape, called "Glob".

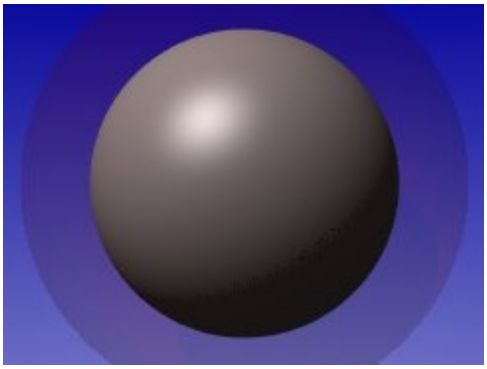
```
isosurface {
  function {-f_glob (x, y, z, 0.1)}
        max_gradient 2
        contained_by {sphere {0, 1.2}}
        pigment {rgb 0.9}
        finish {phong 0.5 phong_size 10}
}
```

One part of this surface would actually go off to infinity if it were not restricted by the contained_by shape. It's possible to select just the teardrop part by choosing the contained_by shape appropriately.

Standard Built In Functions

This page looks at what used to be called the "standard" built in functions. POV-Ray 3.5 does not distinguish "standard" functions from "library" functions

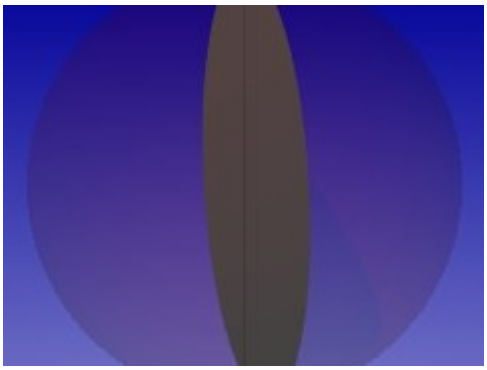
The first three functions are more useful when used in combination with other functions, or for expressing a surface in terms of 3d polar co-ordinates, but these images show them working alone. In the helixes and spiral functions, the 6th parameter is the cross section type. The following values are possible:-
0: square
1: circle
2: diamond
3: concave diamond
Fractional values produce results that are intermediate between these shapes, i.e.
0.0 to 1.0: rounded squares
1.0 to 2.0: rounded diamonds
2.0 to 3.0: partially concave diamonds



```
function {f_r (x, y, z) - 0.7}
```

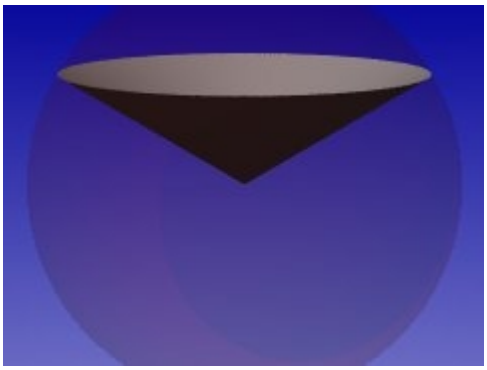
When used alone, the **f_r()** function gives a surface that consists of all the points that are a specific distance from the origin, i.e. a sphere. If you use a threshold of zero (the default) this gives a sphere of size zero, which is invisible.

In this image I've subtracted 0.7 from the function, which is identical to setting the threshold to 0.7. (My mathematical background causes me to prefer to think of the surface as "R - 0.7 = 0" rather than "R = 0.7".



```
function {f_th (x, y, z)}
```

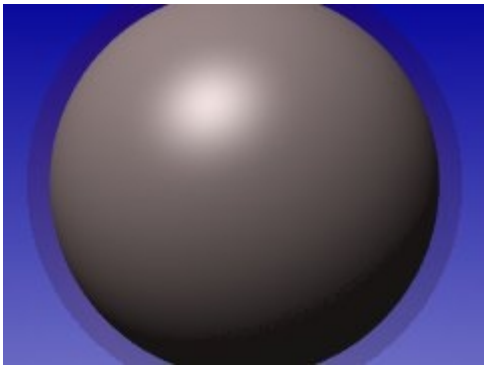
When used alone, the **f_th()** function gives a surface which consists of all points that have a longitude of zero or 180 degrees. I.e. a plane through the origin.



```
function {f_ph (x, y, z)}
threshold 1
```

When used alone, the **f_ph()** function gives a surface that consists of all points that are at a particular latitude, i.e. a cone. If you use a threshold of zero (the default) this gives a cone of width zero, which is invisible.

For this image, I've set the threshold to 1. The cone consists of all points that have Phi equal to 1 radian.



```
function {f_sphere(x, y, z, 0.9)}
```

The **f_sphere()** function creates a sphere.

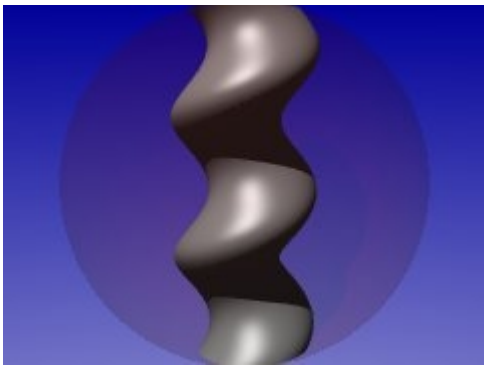
There is one parameter:

1.Radius of sphere

```
function {f_helix1 (x, y, z, 2, 5, 0.1, 0.3, 1, 2, 45)}
```

At last, an interesting shape. **f_helix1()** is intended for use with helixes where the major radius is greater than the minor radius. There are seven parameters:-

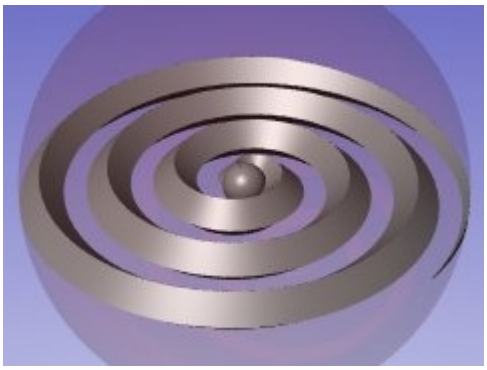
1.Number of helixes - e.g. 2 for a double helix
2.Period - is related to the number of turns per unit length.
3.Minor radius
4.Major radius
5.Shape parameter. If this is greater than 1 then the tube becomes fatter in the y direction.
6.Cross section type.
7.Cross section rotation angle (degrees). E.g. if you choose a square cross section and rotate it by 45 degrees you get a diamond cross section.



```
function {f_helix2 (x, y, z, 0, 8, 0.3, 0.1, 1, 1, 0) }
```

f_helix2() is intended for use with helixes where the minor radius is greater than the major radius. I.e. for situations like twisty table legs.
The parameters are:-

1.Not used
2.Period - is related to the number of turns per unit length.
3.Minor radius
4.Major radius
5.Not used.
6.Cross section type.
7.Cross section rotation angle (degrees). E.g. if you choose a square cross section and rotate it by 45 degrees you get a diamond cross section.

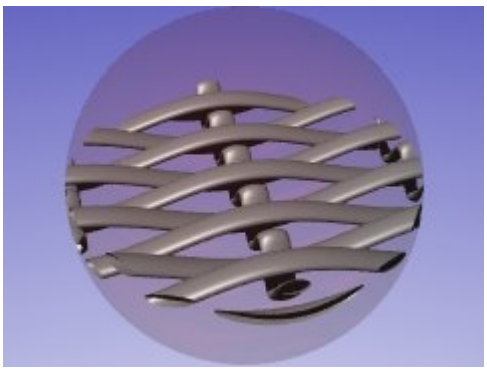


```
function {f_spiral (x, y, z, 0.3, 0.1, 1, 0, 0, 2) }
```

The parameters of the **f_spiral()** function are:-

1.Distance between windings: setting this to 0.3 means that the spiral is 0.3 pov units further from the origin each time it completes one whole turn.
2.Thickness

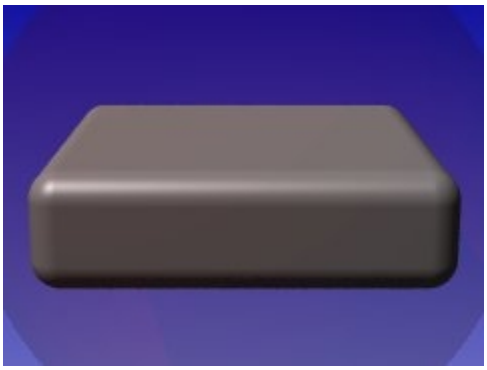
3.Outer diameter of the spiral. The surface behaves as if it is contained_by a sphere of this diameter.
4.not used
5.not used
6.cross-section shape



```
function {f_mesh1 (x, y, z, 1, 0.2, 1, 0.1, 2)}
threshold 0.08}
```

f_mesh1() gives a set of threads that weave up and down through each other in a rectangular pattern.
Note: The overall thickness of the threads is controlled by the isosurface threshold, not by a parameter. If you render a mesh1 with zero threshold, the threads have zero thickness and are therefore invisible. Parameters 3 and 5 control the shape of the thread relative to this threshold parameter.
The parameters are:

1.Number of threads per unit in the x direction
2.Number of threads per unit in the z direction
3.Relative thickness in the x and z directions.
4.Amplitude of the weaving effect. Set to zero for a flat grid.
5.Relative thickness in the y direction.



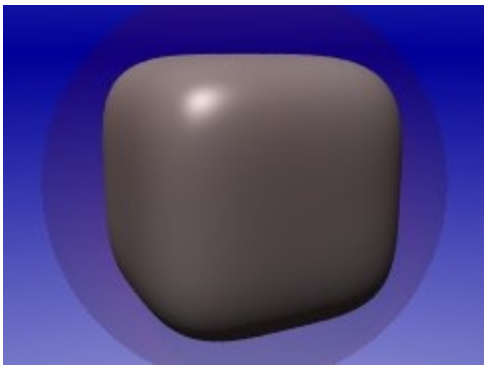
```
function {f_rounded_box (x, y, z, 0.1, 0.7, 0.2, 0.7)}
```
The **f_rounded_box()** takes 4 parameters:

1. Radius of curvature. Zero gives square corners, 0.1 gives corners that match "sphere {0,0.1}".
2. X dimension.
3. Y dimension.
4. Z dimension.

```
function {f_torus (x, y, z, 0.8, 0.1)}
```
The **f_torus()** function takes 2 parameters:

1. Major radius.
2. Minor radius.

Thus the isosurface **function {f_torus(x,y,z,0.8,0.1)}** is identical to the conventional **torus {0.8,0.1}**.

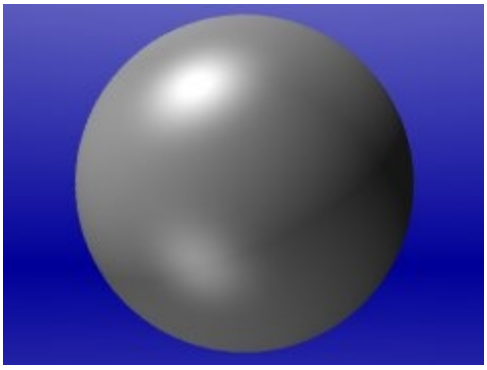


```
function {-f_superellipsoid (x, y, z, 0.5, 0.5) }
```
The **f_superellipsoid()** isosurface function creates a surface that's the same as the conventional superellipsoid object, and the two parameters have the same effects.

It happens that the algorithm used for this function is inside out, so if you render function {f_superellipsoid(x,y,z,0.5,0.5) } all you see is the outside of your contained_by surface, and you may not be aware that there's a superellipsoidal hole buried inside it.

To turn it right side out, a minus sign must be applied. If using a non-zero threshold, it needs to be negated also.

Inside Out



Sometimes, when you render a surface, you may find that you get nothing like what you expect. This is supposed to be a "Bicorn" surface, but it looks absolutely nothing like it.

What you're actually seeing is the "contained_by" surface.

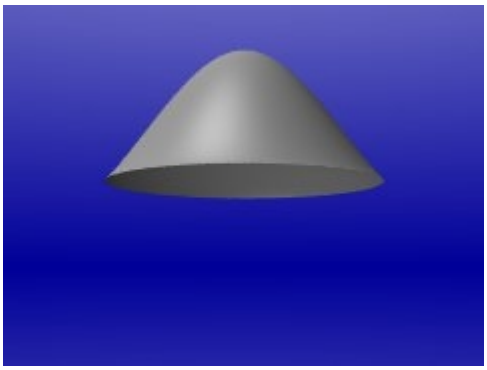
```
isosurface {
    function {f_bicorn(x, y, z, 1, 1)}
         max_gradient 2
         contained_by {sphere {0, R}}
         pigment {rgb 0.9}
         finish {phong 0.5 phong_size 10}
}
```

To see what's going on, we can slice the object in half. (Note, in passing that when we perform CSG operations on an isosurface we need to specify a max_trace parameter, otherwise some parts of the surface may be missing. In this case max_trace 3 is just sufficient to cause the full surface to appear.

We can see that the surface that we wanted is buried inside, and it's hollow.

```
intersection {
```

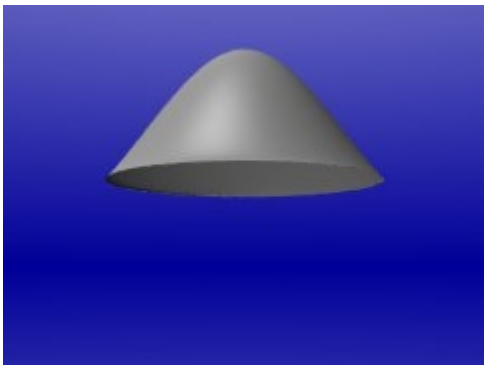
```
  plane {-z, 0 }
  isosurface {
    function {f_bicorn(x, y, z, 1, 1)}
        max_gradient 20
        contained_by{sphere {0, R}}
        max_trace 3
  }
  pigment {rgb 0.9}
  finish {phong 0.5 phong_size 10}
}
```



This effect is caused by the fact that the inside of the surface is defined as the region of space where the function is less than the threshold (i.e. in this case the region where the function evaluates to a negative value). For many functions, the inside is where you would expect it to be, but some complicated functions are defined inside out.

We can invert the isosurface by making the function negative. (Note that we can't use the "inverse" command because that inverts the contained_by surface as well and the result looks the same apart from the "Camera is inside a non-hollow object" warning.) If we were using a threshold value we would have to make that negative also.

```
isosurface {
  function {0 - f_bicorn (x, y, z, 1, 1)}
        max_gradient 5
        contained_by {sphere {0, R}}
        pigment {rgb 0.9}
        finish {phong 0.5 phong_size 10}
}
```



Another way of solving the problem is to make the contained_by surface "open".

When you specify "open", the contained_by surface becomes invisible.

```
isosurface {
  function {f_bicorn (x, y, z, 1, 1)}
        max_gradient 2
        contained_by {sphere {0, R}} open
        pigment {rgb 0.9}
        finish {phong 0.5 phong_size 10}
}
```

i_algbr Library part 1

On this page, I examine the built in functions that came from the i_algbr library. These functions were originally based on a comp.graphics.rendering.raytracing article by Tore Nordstrand, and was originally implemented as an external library.

These functions are now included in the program code, so they can be used without using external library calls. There are some special parameters with complicated effects that are found in several of these functions. Rather than describe them each time they occur, I'll describe them here, and refer to them later as "Field Strength" and "Field Limit" etc.

Field Strength The numerical value at a point in space generated by the function is multiplied by the Field Strength. The set of points where the function evaluates to zero are unaffected by any positive value of this parameter, so if you're just using the function on its own with threshold = 0, the generated surface is still the same.

In some cases, the field strength has a considerable effect on the speed and of rendering the surface. In general, increasing the field strength speeds up the rendering, but if you set the value too high the surface starts to break up and may disappear completely.

The surface generated with threshold = 0.1 and field strength 1 is the same as the surface generated with threshold = 0.2 and field strength 2. Doubling the field strength has doubled the numeric value at a point in space, but doubling the threshold tells the program to look for points where the numeric value is double.

Setting the field strength to a negative value produces the inverse of the surface, like setting sign -1.

Field Limit The numerical value at a point in space generated by the function is limited to plus or minus the field limit. E.g. if you set the field limit to 2, then the value returned for a point that is a large distance from the surface will be +2 or -2.

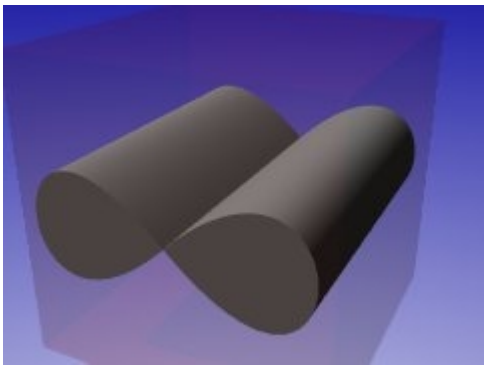
This won't make any difference to the generated surface if you're using threshold that's within the field limit (and will kill the surface completely if the threshold is greater than the field limit). However, it may make a huge difference to the rendering times.

If you use the function to generate a pigment, then all points that are a long way from the surface will have the same colour, the colour that corresponds to the numerical value of the field limit.

SOR Switch If greater than zero, the curve is swept out as a surface of revolution. If the value is zero or negative, the curve is extruded linearly in the Z direction.

SOR Offset If the SOR switch is on, then the curve is shifted this distance in the X direction before being swept out.

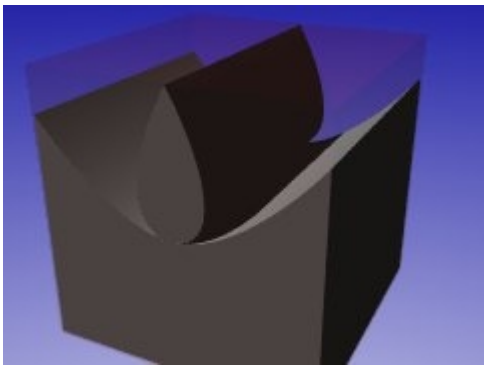
SOR Angle If the SOR switch is on, then the curve is rotated this number of degrees about the Z axis before being swept out.



An algebraic cylinder is what you get if you take any 2d curve and plot it in 3d. The 2d curve is simply extruded along the third axis, in this case the z axis.

In the article that started all this, Tore Nordstrand just happened to mention four particular 2d curves that could be used in this way, and these have become the Algbr_Cyl functions that are implemented.

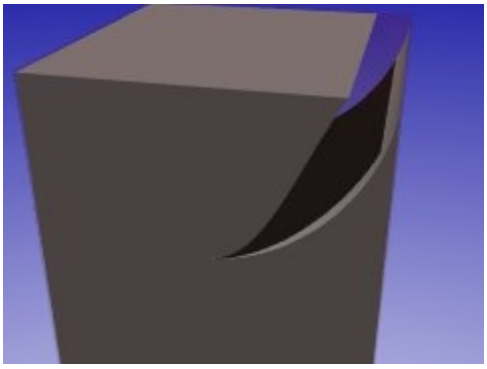
```
function {f_algbr_cyl1 (x, y, z, 1, 1, 0, 0, 0)}
```



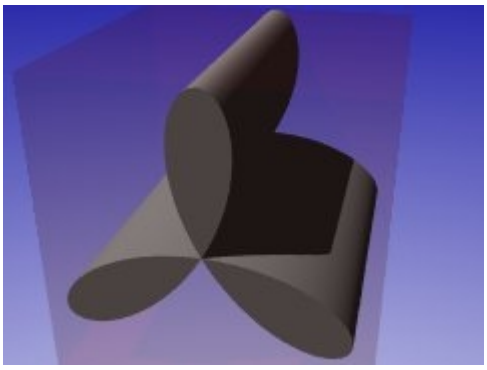
The parameters of all the algebraic cylinders are:

1. Field Strength.
2. Field Limit.
3. SOR Switch.
4. SOR Offset.
5. SOR Angle.

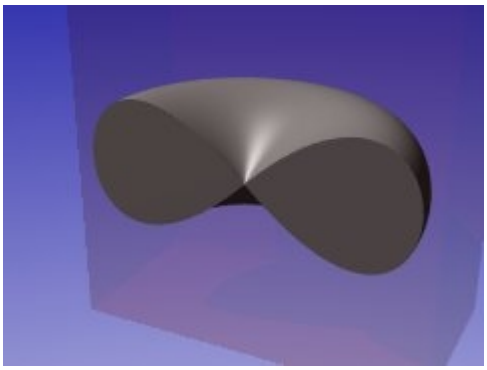
```
function {f_algbr_cyl2 (x, y, z, 1, 1, 0, 0, 0)}
```



```
function {f_algbr_cyl3 (x, y, z, 1, 1, 0, 0, 0)}
```



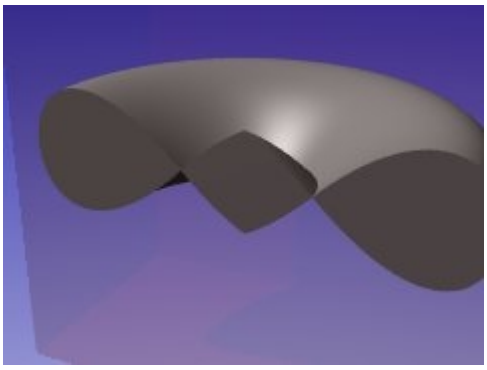
```
function {0 – f_algbr_cyl4 (x, y, z, 1, 1, 0, 0, 0)}
```



This is Algebraic Cylinder #1 again, but this time with the *SOR Switch* switched on. The figure-of-eight curve is now rotated around the Y axis instead of being extruded along the Z axis.

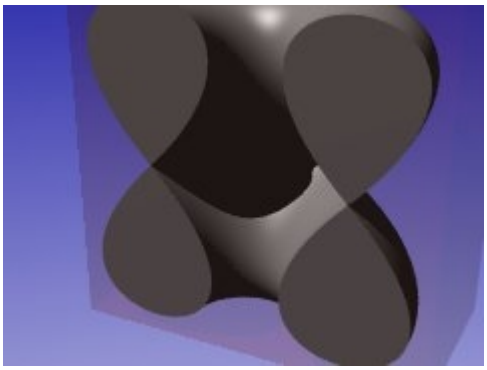
I've sliced these three images in half so that you can see the 2d figure-of-eight curve that generates them more easily.

```
function {f_algbr_cyl1 (x, y, z, 1, 1, 1, 0, 0)}
```



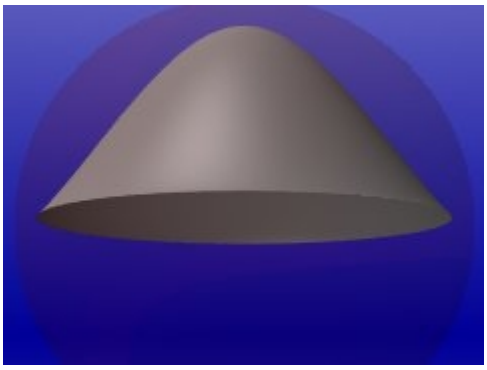
This time the figure-of-eight curve has been shifted 0.4 units to the right before being swept out by setting *SOR Offset* to 0.4.

```
function {f_algbr_cyl1 (x, y, z, 1 , 1, 1, 0.4, 0)}
```



This time the figure-of-eight curve has been rotated by 90 degrees and shifted 0.6 units to the right before being swept out by setting *SOR Angle* to 90.

```
function {f_algbr_cyl1 (x, y, z, 1, 1, 1, 0.6, 90)}
```

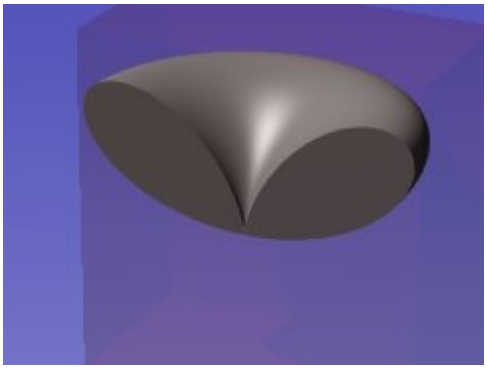


The bicorn surface looks something like the top part of a paraboloid bounded below by another paraboloid.

The parameters are:

1. Field Strength.
2. Scale. The surface is always the same shape. Changing this parameter has the same effect as adding a scale modifier. Setting the scale to 1 gives a surface with a radius of about 1 unit.

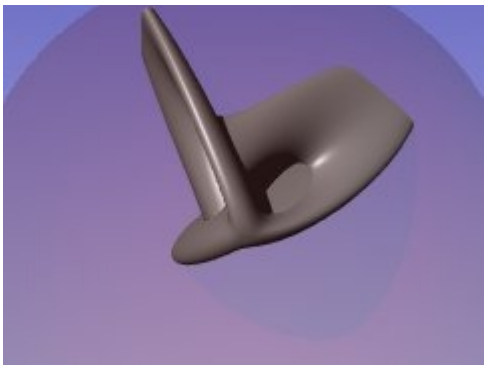
```
function {-f_bicorn (x, y, z, 1, 1)}
```



I've cut this bifolia in half to make the structure clearer. The surface is a surface of revolution. The parameters are:

1. Field Strength.
2. Scale. The mathematics of this surface suggest that the shape should be different for different values of this parameter. In practice the difference in shape is hard to spot. Setting the scale to 3 gives a surface with a radius of about 1 unit.

```
function {-f_bifolia (x, y, z, 1, 3)}
```



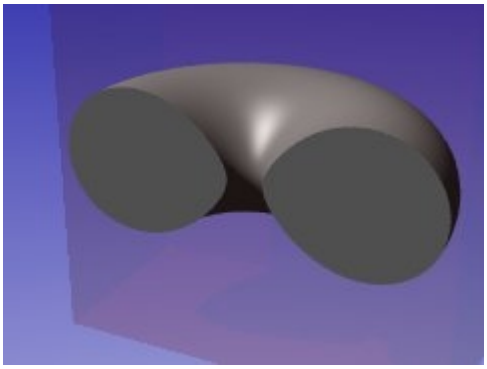
The "Boy_surface" is a model of the projective plane without singularities. Found by Werner Boy on assignment from David Hilbert.

For this surface, it helps if the field strength is set extremely low, otherwise the surface has a tendency to break up or disappear entirely. This has the side effect of making the rendering times extremely long - this image took 16 times longer to render than any other surface on this page.

The parameters are:

1. Field Strength.
2. Scale. The surface is always the same shape. Changing this parameter has the same effect as adding a scale modifier.

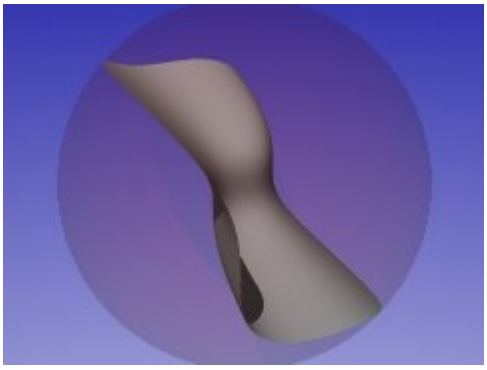
```
function {-f_boy_surface (x, y, z, 0.0000001, 1)}
```



The Ovals of Cassini are a generalisation of the torus shape. The parameters are:

1. Field Strength.
2. Major radius - like the major radius of a torus.
3. Filling. Set this to zero, and you get a torus. Set this to a higher value and the hole in the middle starts to heal up. Set it even higher and you get an ellipsoid with a dimple.
4. Thickness. The higher you set this value, the plumper is the result.

```
function {-f_ovals_of_cassini (x, y, z, 1, 0.5, 0.24, 5)}
```



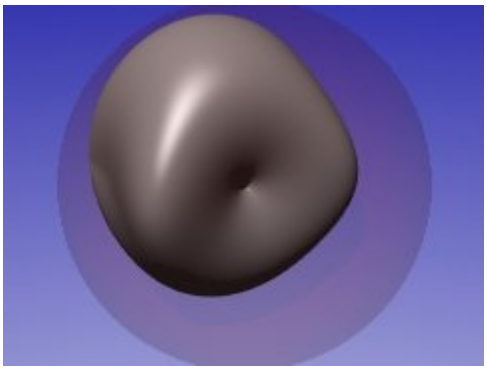
This is a cubic saddle.

The parameters are:

1. Field Strength.

For this surface, it helps if the field strength is set quite low, otherwise the surface has a tendency to break up or disappear entirely.

```
function {f_cubic_saddle (x, y, z, 0.1)}
```

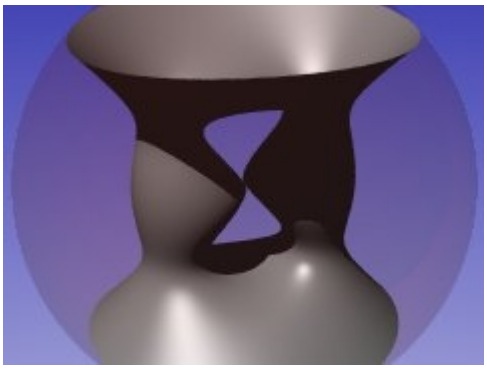


This is the "Cushion" surface.

The parameters are:

1. Field Strength.

```
function {-f_cushion (x, y, z, 1)}
```

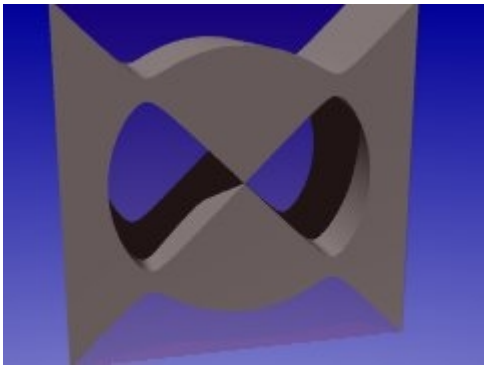


This is the "Devil's Curve" surface.

The parameters are:

1. Field Strength.

```
function {-f_devils_curve (x, y, z, 1)}
```



The 2d devil's curve can be extruded along the z axis, like this, or using the SOR parameters it can be made into a surface of revolution.

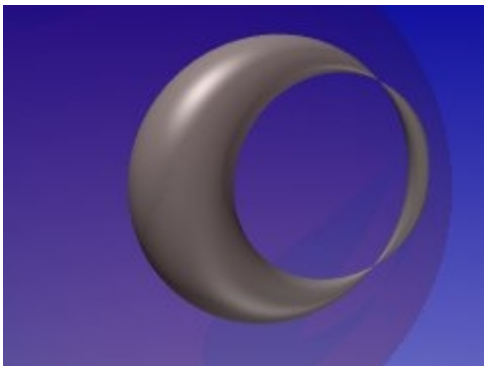
The parameters are:

1. Field Strength.
2. X factor.
3. Y factor.
4. SOR Switch.
5. SOR Offset.
6. SOR Angle.

The X and Y factors control the size of the central feature. If the X factor is slightly stronger than the Y factor, then the side pieces are linked to the central piece by a horizontal bridge at each corner. If the Y factor is slightly

greater than the X factor, then there is a vertical gap between the side pieces and the central piece at each corner. If the X and Y factors are equal each of the four corners meets at a point.

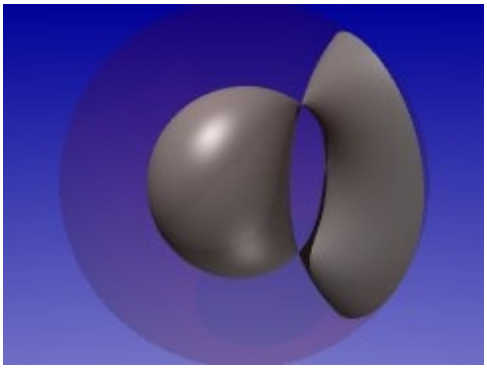
```
function {f_devils_curve_2d (x, y, z, 1, 1.5, 1.52, 0, 0, 0) }
```



The Dupin Cyclid can take several forms depending on the parameters. This particular configuration is known as a "double crescent".

A Dupin Cyclid can be considered to be the envelope of all the possible spheres that just kiss three other spheres. It can also be considered to be the envelope of all spheres on a conic section that just touch a given sphere. And it can also be considered to be the "inversion" of a torus.

```
function {f_dupin_cyclid (x, y, z, 0.0001, 4.9, 5, 2, 0, 3)}
```



The parameters of the Dupin Cyclid consider it to be generated by the "inversion" of a torus. (Inversion here has some weird mathematical meaning, don't confuse it with the POV concept of "inverse").

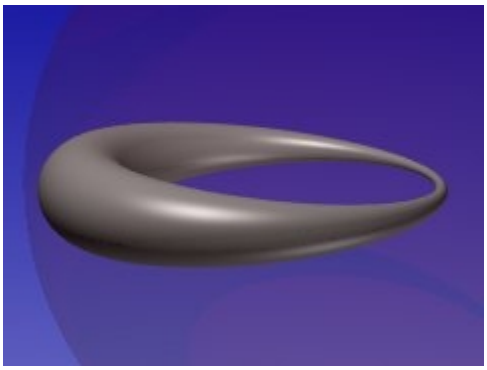
The parameters are:

1. Field Strength.
2. Major radius of torus.
3. Minor radius of torus.
4. X displacement of torus.
5. Y displacement of torus.

6.radius of inversion.

This particular Dupin Cyclid is called a "degenerate w. arch".

```
function {f_dupin_cyclid (x, y, z, 0.000001, 3, 5, 3, 0, 9)}
```

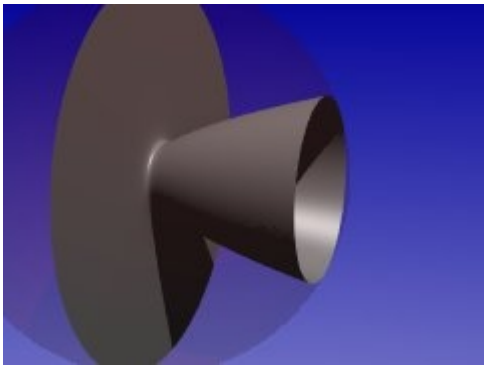


This is a "plain" Dupin Cyclid.

```
function {-f_dupin_cyclid (x, y, z, 0.0000001, 6, 0.5, 3, 0, 12)}
```

i_algbr Library part 2

See the previous page for details of "*Field Strength*", "*Field Limit*", "*SOR switch*", "*SOR offset*" and "*SOR angle*" parameters

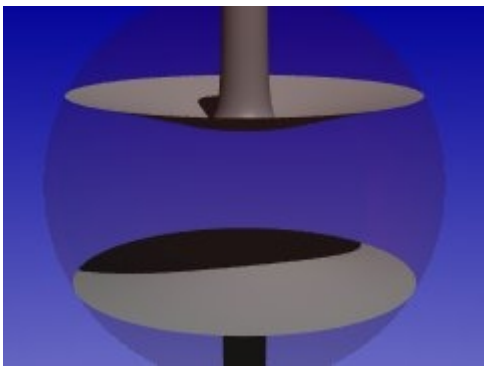


A Folium Surface looks something like a paraboloid glued to a plane.

The parameters are:

1. Field Strength
2. Neck width factor - the larger you set this, the narrower the neck where the paraboloid meets the plane.
3. Divergence - the higher you set this value, the wider the paraboloid gets.

```
function {-f_folium_surface (x, y, z, 0.01, 3, 5)}
```

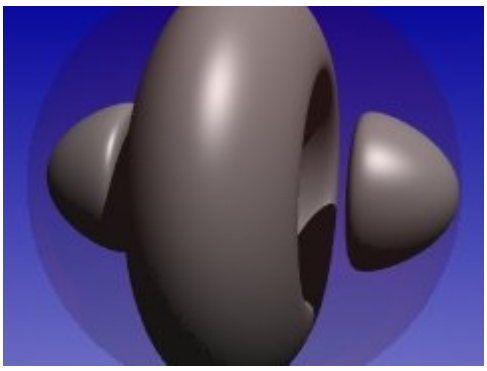


The 2d folium curve can be rotated around the X axis to generate the same 3d surface as the above, or it can be extruded in the Z direction (by switching the *SOR switch* off), or it can be rotated around the Y axis, like this.

The parameters are:

1. Field Strength
2. Neck width factor - same as the 3d surface if you're revolving it around the Y axis.
3. Divergence - same as the 3d surface if you're revolving it around the Y axis.
4. SOF switch
5. SOF offset
6. SOF angle

```
function {f_folium_surface_2d (x, y, z, 0.01, 1, 1, 1, 0, 0)}
```

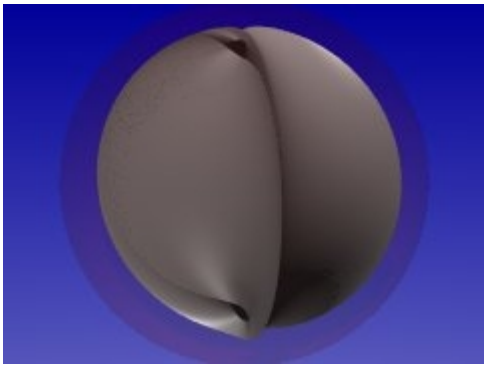


The "Torus Gumdrop" surface us something like a torus with a couple of gumdrops hanging off the end.

The parameters are:

1. Field Strength

```
function {-f_torus_gumdrop (x, y, z, 0.01)}
```

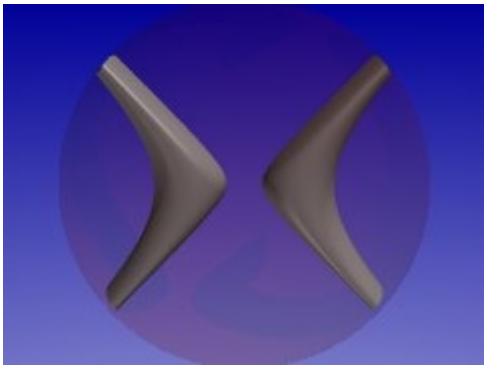


This is the "Hunt Surface".

The parameters are:

1. Field Strength

```
function {-f_hunt_surface (x, y, z, 0.1)}
```

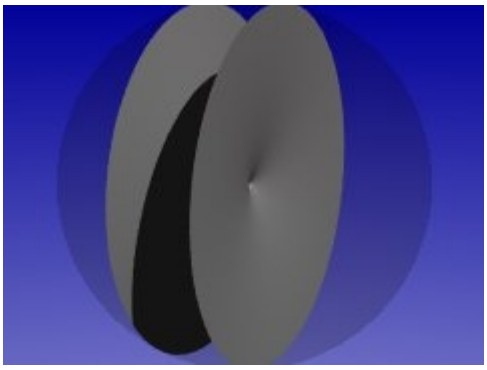


This is a "Hyperbolic Torus".

The parameters are:

1. Field Strength
2. Major radius: separation between the centres of the tubes at the closest point.
3. Minor radius: thickness of the tubes at the closest point.

```
function {-f_hyperbolic_torus (x, y, z, 1, 0.6, 0.4)}
```

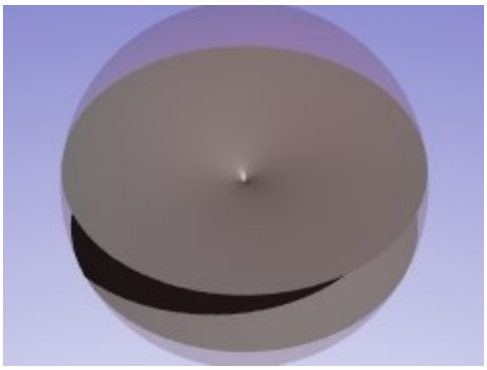


The Kampyle of Eudoxus" is like two infinite planes with a dimple at the centre.

The parameters are:

1. Field Strength
2. Dimple: When zero, the two dimples punch right through and meet at the centre. Non-zero values give less dimpling.
3. Closeness: Higher values make the two planes become closer.

```
function {f_kampyle_of_eudoxus (x, y, z, 1, 0, 1)}
```



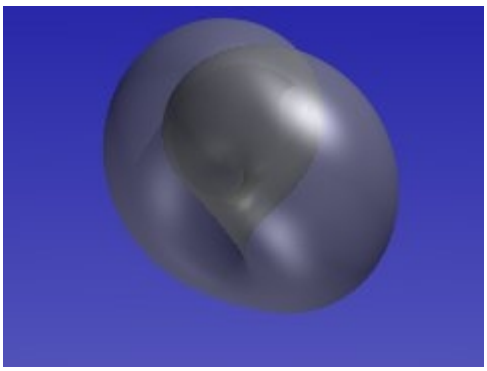
The 2d curve that generates the above surface can be extruded in the Z direction or rotated about various axes by using the SOR parameters. In this case I've created the same shape as in the 3d version.

The parameters are:

1. Field Strength
2. Dimple: When zero, the two dimples punch right through and meet at the centre. Non-zero values give less dimpling.
3. Closeness: Higher values make the two planes become closer.
4. SOR switch
5. SOR offset

6. SOR angle

```
function {-f_kampyle_of_eudoxus_2d (x, y, z, 1, 0, 1, 1, 0, 90)}
```

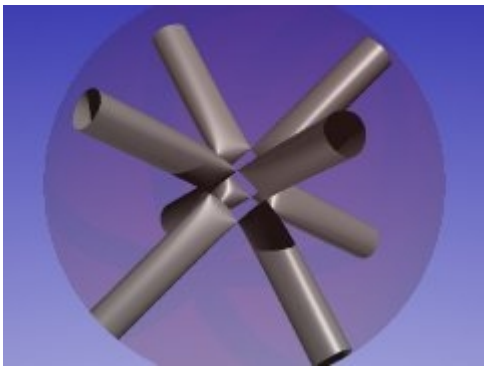


The Klein Bottle is the 3d equivalent of the Moebius Strip. It's a surface with only one side. It's hard to see what's going on from one side, so I've made the surface partially transparent so that those of you with good imaginations can get a hint of how the whole shape works.

The parameters are:

1. Field Strength

```
function {f_klein_bottle (x, y, z, -1)}
```



The Kummer surface consists of a collection of radiating rods.

The parameters are:

1. Field Strength

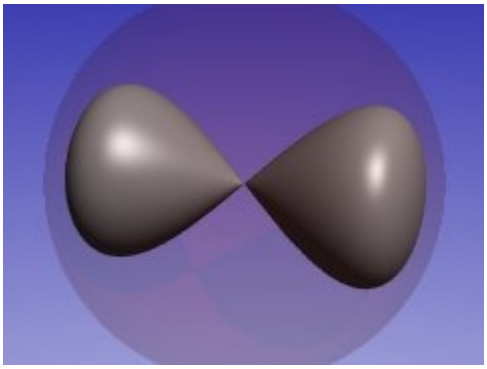
```
function {-f_kummer_surface_v1 (x, y, z, 0.01)}
```

Version 2 of the Kummer Surface only looks like radiating rods when the parameters are set to particular negative values. For positive values it tends to look rather like a superellipsoid.

The parameters are:

1. Field Strength
2. Rod width (negative): Setting this parameter to larger negative values increases the diameter of the rods
3. Divergence (negative): Setting this number to -1 causes the rods to become approximately cylindrical. Larger negative values cause the rods to become fatter further from the origin. Smaller negative numbers cause the rods to become narrower away from the origin, and have a finite length.
4. Influences the length of half of the rods. Changing the sign affects the other half of the rods. 0 has no effect.

```
function {f_kummer_surface_v2 (x, y, z, 0.001, -2, -0.94, 0.4)}
```

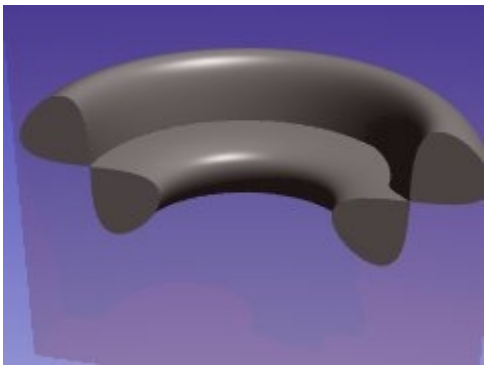


The "Lemniscate_of_Gerono" surface is an hourglass shape. Two teardrops with their ends connected.

The parameters are:

1. Field Strength

```
function {f_lemniscate_of_gerono (x, y, z, 1)}
```



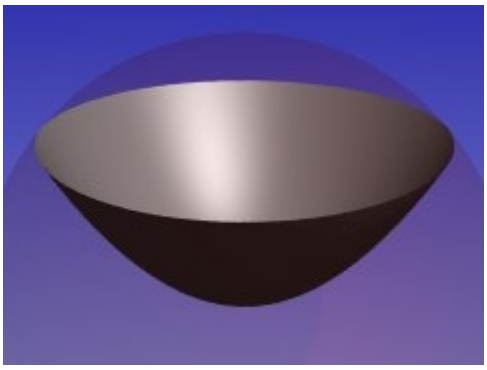
The 2d version of the Lemniscate can be extruded in the Z direction, or used as a surface of revolution to generate the equivalent of the 3d version, or revolved in different ways. To produce the 3d Lemniscate, switch SOR on and set the SOR offset to zero and the SOR angle to 90.

I've cut the surface in half so that you can see the figure-of-eight curve that sweeps round the Y axis to generate this surface of revolution.

The parameters are:

1. Field Strength
2. Size: increasing this makes the 2d curve larger and less rounded.
3. Width: increasing this makes the 2d curve fatter.
4. SOR switch
5. SOR offset
6. SOR angle

```
function {f_lemniscate_of_gerono_2d (x, y, z, -0.1, 1, 1, 1, 2, -45)}
```



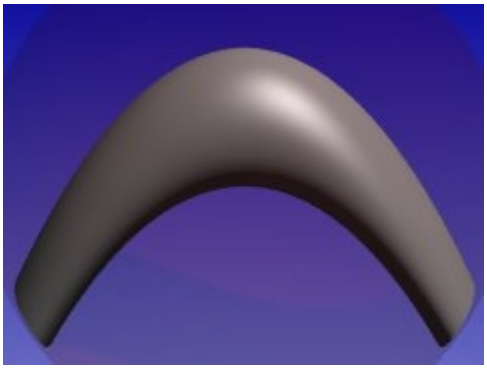
This paraboloid is the surface of revolution that you get if you rotate a parabola about the Y axis. To do this without using the built in function declare a function to represent the parabola y - x*x = 0, then use substitution of variables to perform the SOR operation:

```
#declare F = function {y - x*x}
```

```
isosurface {function {F (sqrt(x*x + z*z), y, z)}}
```
The parameters are:

1. Field Strength

```
function {-f_paraboloid (x, y, z, 1)}
```

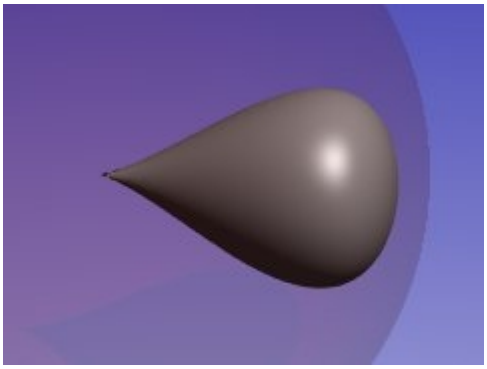


This is a parabolic torus.

The parameters are:

1. Field Strength
2. Major radius.
3. Minor radius.

```
function {-f_parabolic_torus (x, y, z, 0.1, 0.6, 0.5)}
```



The piriform surface looks rather like half a lemniscate.

The parameters are:

1. Field Strength

```
function {f_piriform (x, y, z, 1)}
```

The 2d version of the Piriform can be extruded in the Z direction, or used as a surface of revolution to generate the equivalent of the 3d version, or revolved in different ways.

This might be a useful shape for making hot air balloons - reduce the fatness parameter to make a weather balloon.

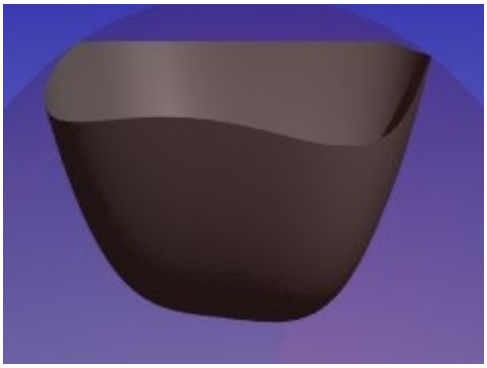
The parameters are:

1. Field Strength
2. Size factor 1: increasing this makes the curve larger.
3. Size factor 2: making this less negative makes the curve larger but also thinner.
4. Fatness: increasing this makes the curve fatter.
5. SOR switch
6. SOR offset
7. SOR angle

```
function {f_piriform_2d (x, y, z, -1, 1, -1, 1, 1, 0, -90)}
```

i_algbr Library part 3

See here for details of "*Field Strength*", "*Field Limit*", "*SOR switch*", "*SOR offset*" and "*SOR angle*" parameters

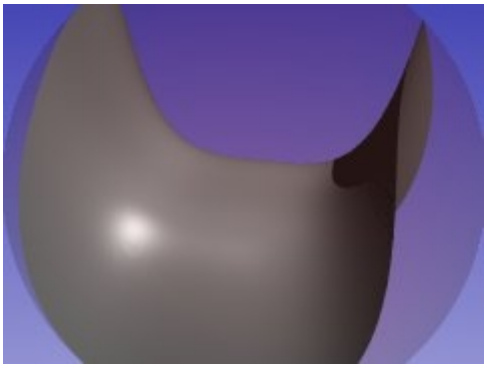


The Quartic paraboloid is similar to a paraboloid, but has a squarer shape.

The parameters are:

1. Field Strength

```
function {-f_quartic_paraboloid (x, y, z, 1)}
```

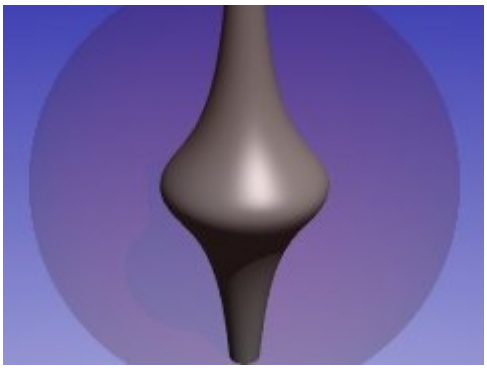


The Quartic saddle is similar to a saddle, but has a squarer shape.

The parameters are:

1. Field Strength

```
function {f_quartic_saddle (x, y, z, 1)}
```

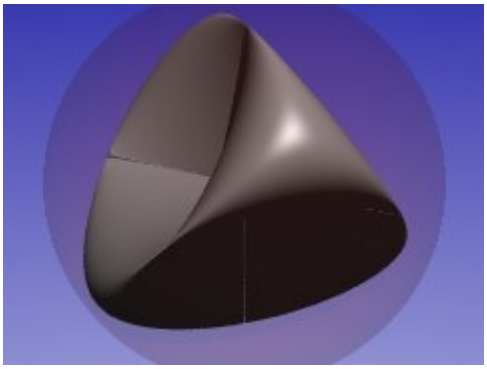


The Quartic cylinder looks a bit like a cylinder that's swallowed an egg.

The parameters are:

1. Field Strength
2. Diameter of the "egg".
3. Controls the width of the tube and the vertical scale of the "egg".

```
function {-f_quartic_cylinder (x, y, z, 1, 0.6, 0.3)}
```



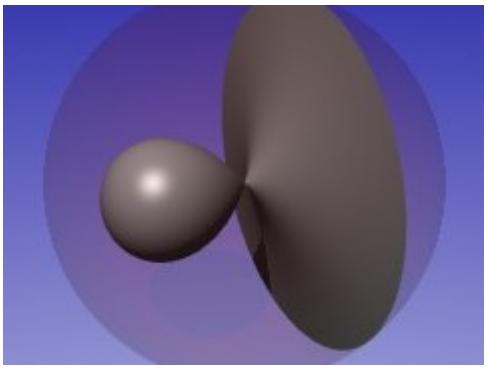
The "Steiners Roman" is composed of four identical triangular pads which together make up a sort of rounded tetrahedron. There are creases along the X, Y and Z axes where the pads meet.

It is a model of the projective plane.

The parameters are:

1. Field Strength

```
function {f_steiners_roman (x, y, z, -1) + 0}
```



The Strophoid is like an infinite plane with a bulb sticking out of it.

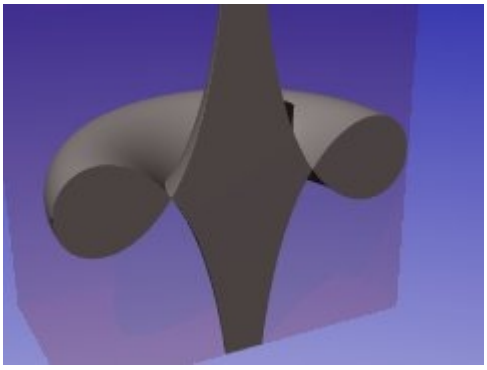
The parameters are:

1. Field Strength
2. Size of bulb. Larger values give larger bulbs. Negative values give a bulb on the other side of the plane.
3. Sharpness. When zero, the bulb is like a sphere that just touches the plane. When positive, there is a crossover point. When negative the bulb simply bulges out of the plane like a pimple.
4. Fatness. Higher values make the top end of the bulb fatter.

When the Size and Sharpness parameters are equal, the surface is given the special name "right strophid".

When the Size parameter is 3 times the Sharpness, the surface is given the special name "trisectrix of Maclaurin".

```
function {f_strophoid (x, y, z, 1, 1.5, 1, 1.2)}
```



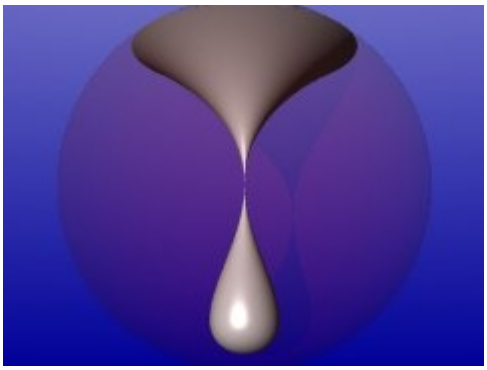
The 2d strophid curve can be extruded in the Z direction or rotated about various axes by using the SOR parameters.

The parameters are:

1. Field Strength

2.Size of bulb. Larger values give larger bulbs. Negative values give a bulb on the other side of the plane.
3.Sharpness. When zero, the bulb is like a sphere that just touches the plane. When positive, there is a crossover point. When negative the bulb simply bulges out of the plane like a pimple.
4.Fatness. Higher values make the top end of the bulb fatter.
5.SOR switch
6.SOR offset
7.SOR angle

```
function {f_strophoid_2d (x, y, z, -1, 1.5, 1, 1.2, 1, 1, 180)}
```

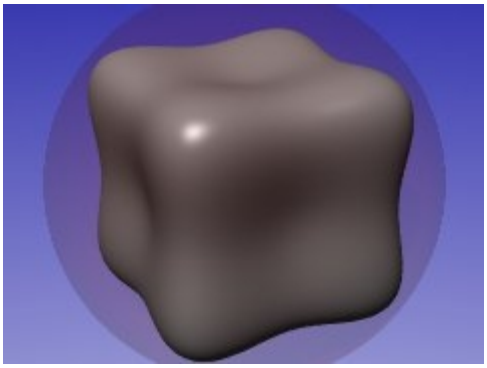


I've rotated this Glob function through z*90 so that it looks like a falling drip of thick liquid.

The parameters are:

1.Field Strength

```
function {f_glob (x, y, z, -1)}
```



This "pillow" surface apparently featured on the back cover of the 1992 Siggraph proceedings.

The parameters are:

1. Field Strength

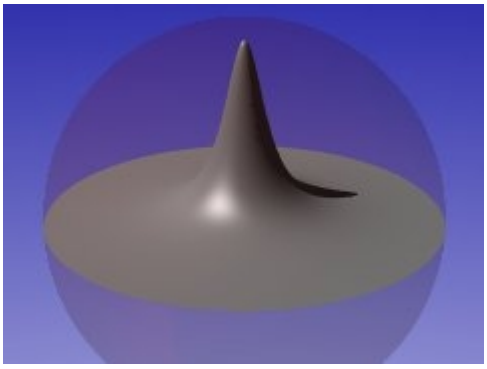
```
function {f_pillow (x, y, z, 1)}
```

This is the "Crossed Trough" surface.

The parameters are:

1. Field Strength

```
function {-f_crossed_trough (x, y, z, 1) + 0}
```

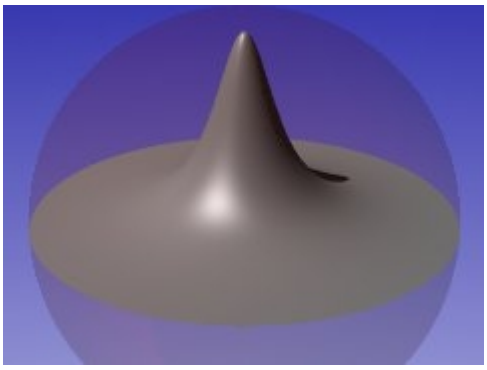


The "Witch of Agnesi" surface looks something like a witches hat.

The parameters are:

1. Field Strength
2. Controls the width of the spike. The height of the spike is always about 1 unit.

```
function {-f_witch_of_agnesi (x, y, z, 1, 0.02)}
```



The 2d version of the Witch of Agnesi curve can be extruded in the Z direction or rotated about various axes by use of the SOR parameters.

The parameters are:

1. Field Strength
2. Controls the size of the spike.
3. Controls the height of the spike.

To produce a surface that matches the 3d version, the height parameter needs to be the square of the size parameter.

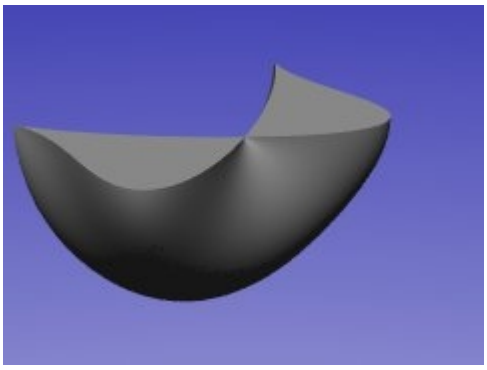
```
function {-f_witch_of_agnesi_2d (x, y, z, 1, 0.2, 0.04, 1, 0, 0)}
```

The "Mitre" surface looks a bit like an ellipsoid which has been nipped at each end with a pair of sharp nosed pliers.

The parameters are:

1. Field Strength

```
function {f_mitre(x, y, z, -1)}
```



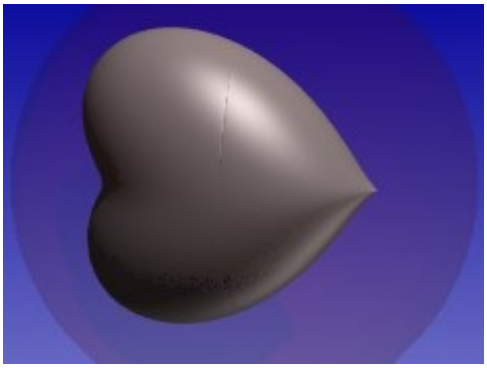
One odd thing about the "Odd" surface is that it's identical to the "Cushion" surface.

This time I've cut it in half so that you can see the way that the rear face puckers forward to meet the dimple from the front face.

The parameters are:

1. Field Strength

```
function {f_odd (x, y, z, -1)}
```

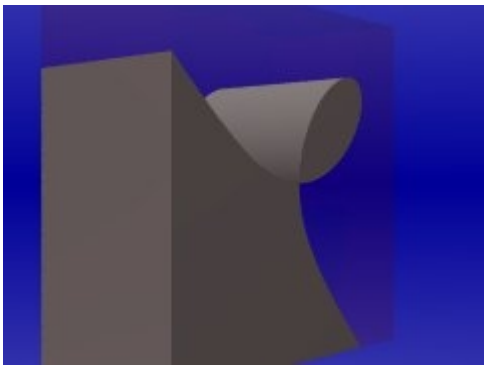


The "Heart" - a surface for Valentine's Day.

The parameters are:

1. Field Strength

```
function {f_heart(x, y, z, -1)}
```

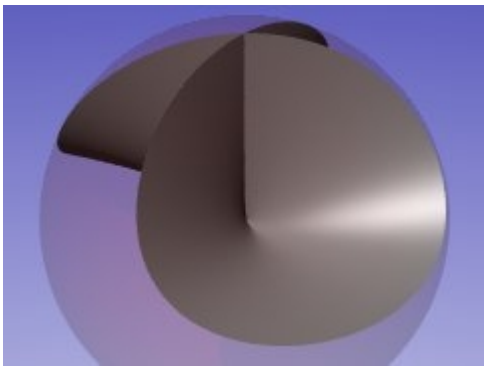


The Nodal Cubic is something like what you'd get if you were to extrude the Stophid2D curve along the X axis and then lean it over.

The parameters are:

1. Field Strength

```
function {f_nodal_cubic (x, y, z, -0.1)}
```

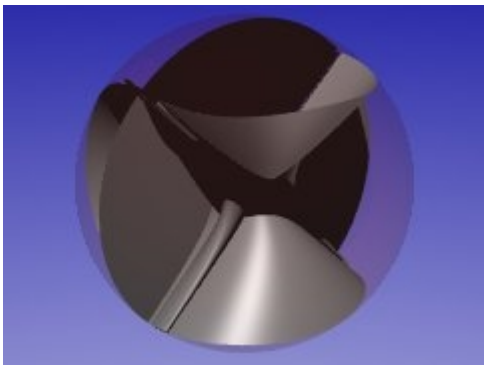


This is the "Umbrella" surface

The parameters are:

1. Field Strength

```
function {f_umbrella (x, y, z, 1)}
```



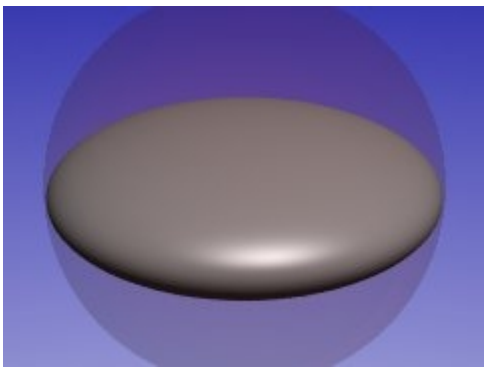
This is the Enneper surface.

The parameters are:

1. Field Strength

```
function {f_enneper (x, y, z, -0.1)}
```

i_nfunc Library part 1



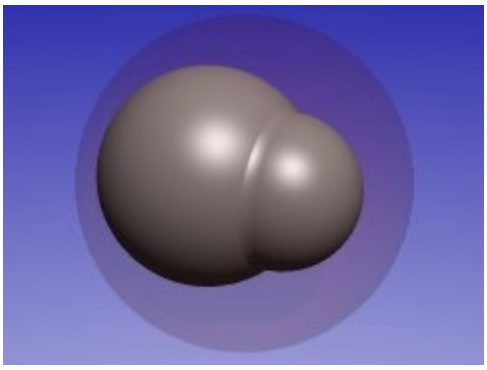
f_ellipsoid generates spheres and ellipsoids.

The parameters are:

1. X scale (inverse)
2. Y scale (inverse)
3. Z scale (inverse)

Setting these scaling parameters to 1/n gives exactly the same effect as performing a scale operation to increase the scaling by n in the corresponding direction.

```
function {f_ellipsoid (x, y, z, 1, 3, 1)}
threshold 1
```



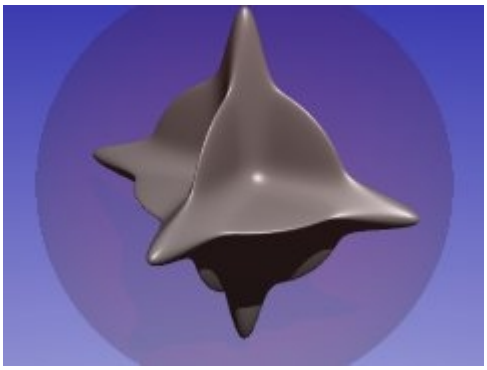
f_blob generates blobs that are similar to a CSG blob with two spherical components.

For some unknown reason, this function only seems to work with negative threshold settings. (It took me lots of attempts with blank images before I thought of trying that.)

The parameters are:

1. X distance between the two components
2. Blob strength of component 1
3. Inverse blob radius of component 1
4. Blob strength of component 2
5. Inverse blob radius of component 2

```
function {f_blob (x, y, z, 1, 1, 0.7, 1, 1)}
threshold -0.01
```



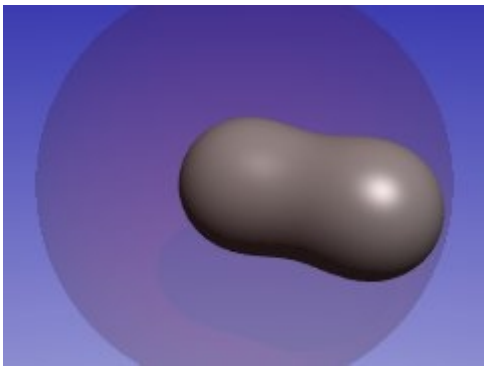
This is f_flange_cover

The parameters are:

1. Spikiness. Set this to very low values to increase the spikes. Set it to 1 and you get a sphere.
2. Inverse size. Increase this to decrease the size of the surface. (The other parameters also drastically affect the size, but this parameter has no other effects).
3. Flange. Increase this to increase the flanges that appear between the spikes. Set it to 1 for no flanges.

4.Threshold. Setting this parameter to 1 and the threshold to zero has exactly the same effect as setting this parameter to zero and the threshold to -1.

```
function {f_flange_cover (x, y, z, 0.01, 35, 1.5, 1)}
```

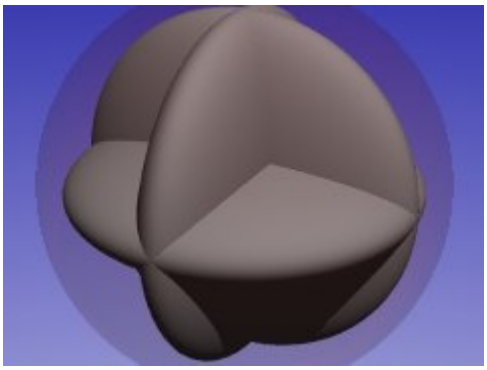


The f_blob2 surface is similar to a CSG blob with two spherical components.

The parameters are:

1.Separation. One blob component is at the origin, and the other is this distance away on the X axis.
2.Inverse size. Increase this to decrease the size of the surface.
3.Blob strength.
4.Threshold. Setting this parameter to 1 and the threshold to zero has exactly the same effect as setting this parameter to zero and the threshold to -1.

```
function {f_blob2 (x, y, z, 1, 3, 2, 1)}
```



The f_cross_ellipsoids surface is like the union of three crossed ellipsoids, one oriented along each axis.

The parameters are:

1.Eccentricity. When less than 1, the ellipsoids are oblate, when greater than 1 the ellipsoids are prolate, when zero the ellipsoids are spherical (and hence the whole surface is a sphere).

2.Inverse size. Increase this to decrease the size of the surface.
3.Diameter. Increase this to increase the size of the ellipsoids.
4.Threshold. Setting this parameter to 1 and the threshold to zero has exactly the same effect as setting this parameter to zero and the threshold to -1.

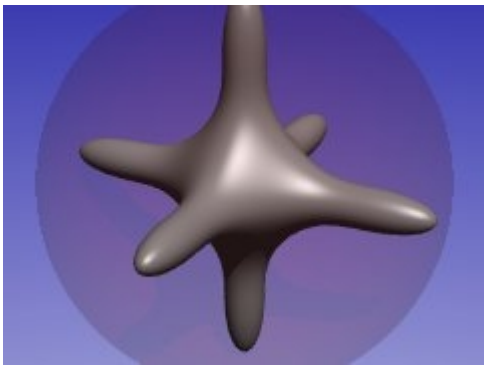
```
function {f_cross_ellipsoids (x, y, z, 0.1, 8, 4, 1)}
```

The f_isect_ellipsoids surface is like the **intersection** of three crossed ellipsoids, one oriented along each axis.

The parameters are:

1.Eccentricity. When less than 1, the ellipsoids are oblate, when greater than 1 the ellipsoids are prolate, when zero the ellipsoids are spherical (and hence the whole surface is a sphere).
2.Inverse size. Increase this to decrease the size of the surface.
3.Diameter. Increase this to increase the size of the ellipsoids.
4.Threshold. Setting this parameter to 1 and the threshold to zero has exactly the same effect as setting this parameter to zero and the threshold to -1.

```
f_isect_ellipsoids (x, y, z, 3, 1, 4, 1)
```

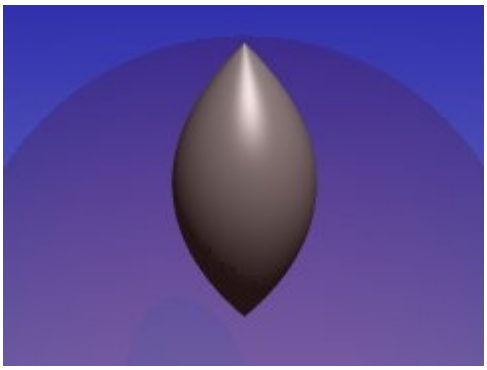


This is the f_spikes surface.

The parameters are:

1. Spikiness. Set this to very low values to increase the spikes. Set it to 1 and you get a sphere.
2. Hollowness. Increasing this causes the sides to bend in more.
3. Size. Increasing this increases the size of the object.
4. Roundness. This parameter has a subtle effect on the roundness of the spikes.
5. Fatness. Increasing this makes the spikes fatter.

```
function {-f_spikes (x, y, z, 0.04, 8, 1, 1, 1)}
threshold -1
```



This function can be used to generate the surface of revolution of any polynomial up to degree 4.

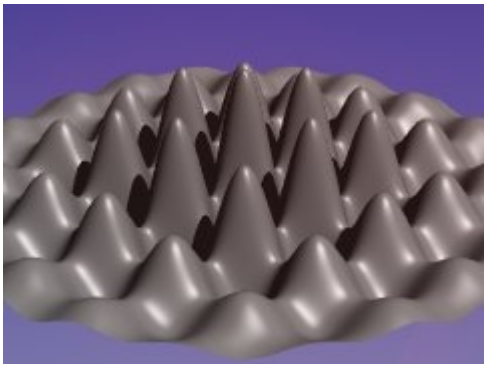
The parameters are:

1. Constant
2. Y coefficient.
3. Y^2 coefficient.
4. Y^3 coefficient.
5. Y^4 coefficient.

To put it another way: If we call the parameters A, B, C, D, E; then this function generates the surface of revolution formed by revolving "x = A + By + Cy2 + Dy3 + Ey4" around the Y axis.

```
function {f_poly4 (x, y, z, 0, 1, -1, 0, 0)}
```

i_nfunc Library part 2



This is the f_spikes_2d surface.

The parameters are:

1. Height of central spike.
2. Frequency of spikes in the X direction.
3. Frequency of spikes in the Z direction.
4. Rate at which the spikes reduce as you move away from the centre.

```
f_spikes_2d (x, y, z, 0.4, 15, 15, 2.5)
```

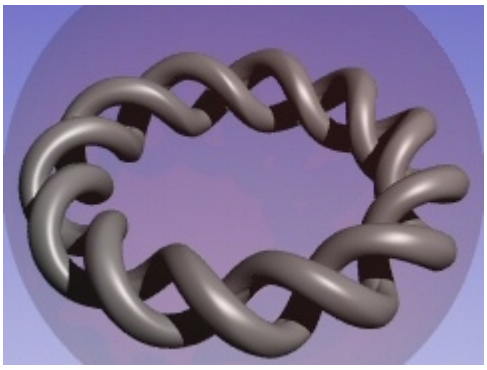
There's a comment in the source code that says "well known function in quantum mechanics". I don't know quantum mechanics well enough to recognise it.

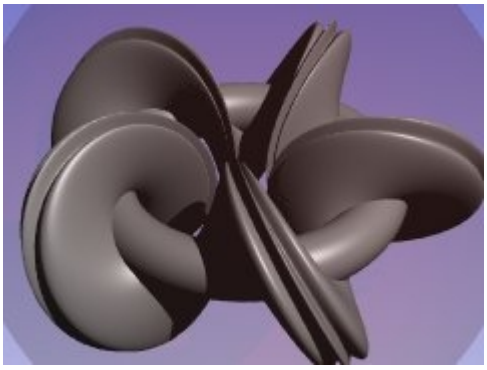
The parameters are:

1. Not used.

This function doesn't use the passed parameter, but the old syntax used to require that at least one parameter be present, and the existence of the parameter has been retained in the new code.

```
f_quantum (x, y, z, 0)
```





This is the f_helical_torus surface. With some sets of parameters, it looks like a helix bent into a circle with, optionally, a torus through the middle. The helix optionally has grooves around the outside.

I'm not completely sure about how the parameters work for this surface.

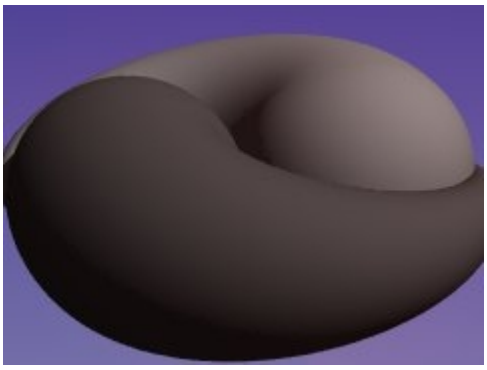
The parameters are something like:

1. Major radius
2. Number of winding loops.
3. Twistiness of winding. When zero, each winding loop is separate.
   When set to one, each loop twists into the next one. When set to two, each loop twists into the one after next.
4. Fatness of winding?
5. Threshold. Setting this parameter to 1 and the threshold to zero has a similar effect as setting this parameter to zero and the threshold to 1.
6. Negative minor radius? Reducing this parameter increases the minor radius of the central torus.
   Increasing it can make the torus disappear and be replaced by a vertical column.
   The value at which the surface switches from one form to the other depends on several other parameters.
7. Another fatness of winding control?
8. Groove period. Increase this for more grooves.
9. Groove amplitude. Increase this for deeper grooves.
10. Groove phase. Set this to zero for symmetrical grooves.

```
function {f_helical_torus (x, y, z, 6, 12, 2, 0.1, .5, 1, 0.1, 1, 1.0, 0)}
```

```
function {f_helical_torus (x, y, z, 2, 5, 1, 0.1, 1, 0.5, 1, 6, 3, 0)}
```

A more controlable helical torus can be created by starting with an ordinary helix and then transforming the coordinate system from cartesian to cylindrical polar coordinates.



The f_comma surface is very much like half a Yin/Yang.

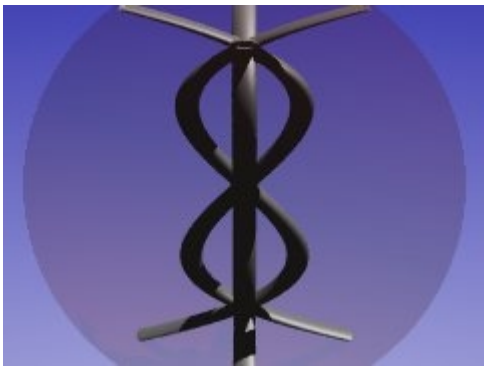
I've included two of them in the image, and chosen the viewing angle so that you can't see that they don't quite fit together properly in the middle.

Actually, they fit reasonably well if you set the threshold to about 0.05.

The parameters are:

1. Scale.

```
function {f_comma (x, y, z, 1)
```



The f_polytubes surface consists of a number of tubes. Each tube follows a 2d curve which is specified by a polynomial of degree 4 or less.

The parameters are:

1. Number of tubes
2. Constant
3. Y coefficient.

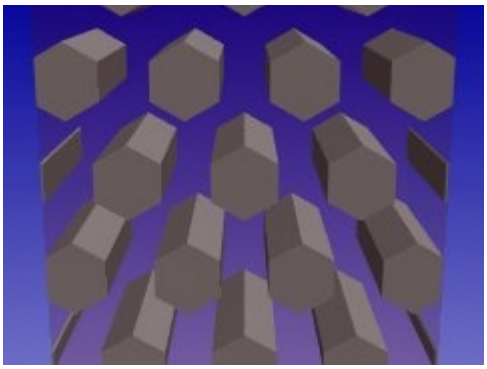
4. Y^2 coefficient.
5. Y^3 coefficient.
6. Y^4 coefficient.

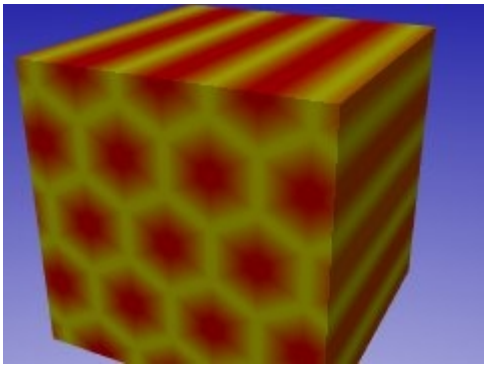
To put it another way: If we call the parameters A, B, C, D, E, F; then this function generates "A" tubes which all follow the equation "x = B + Cy + Dy2 + Ey3 + Fy4" arranged around the Y axis.

```
function {f_polytubes (x, y, z, 4, 0, -1, 0, 1, 0)}
```

Other built-in functions

I guess that these surfaces are mainly intended for use as pigment functions or for modifying other surfaces as you might use a height field.



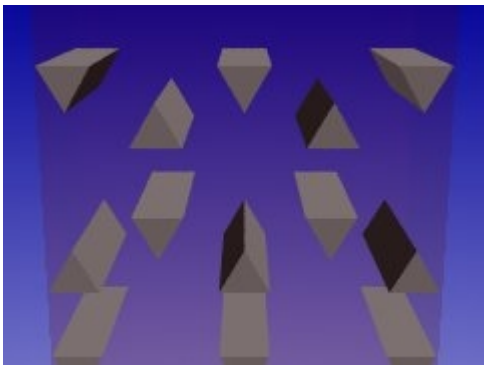


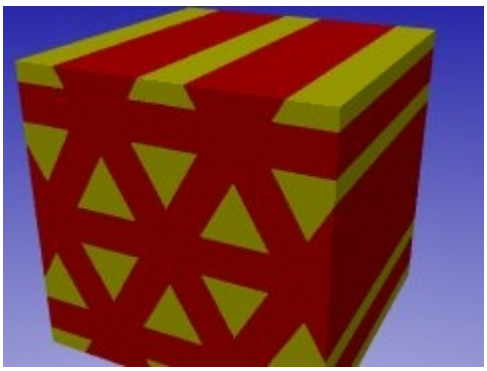
This is the "hex_x" function.

The parameters are:

1. No effect (but the old syntax required at least one parameter)

```
function {f_hex_x (x, y, z, 0)}
```



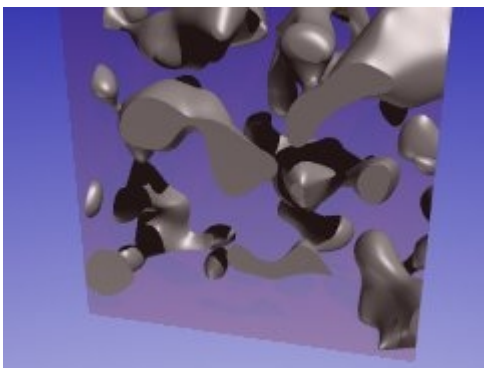


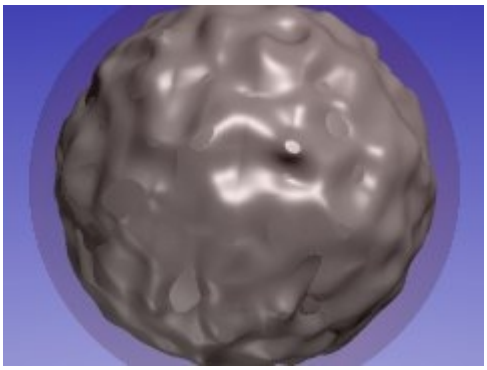
This is the "hex_y" function.

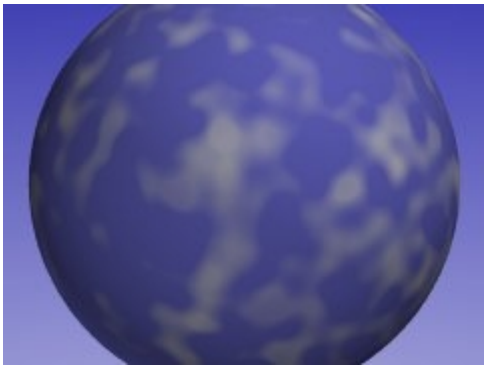
The parameters are:

1. No effect (but the old syntax required at least one parameter)

```
function {-f_hex_x (x, y, z, 0)}
```







This is the "ridge" function.

The three images show

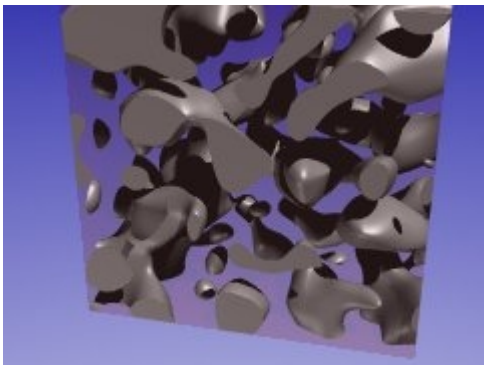
- •the function used alone as a surface
- •the function used as a height field on a sphere
- •the function used as a pigment on a sphere

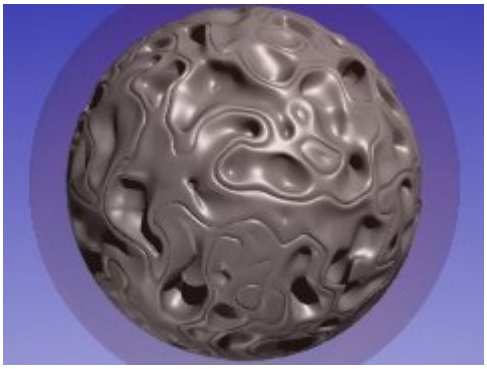
The parameters are:

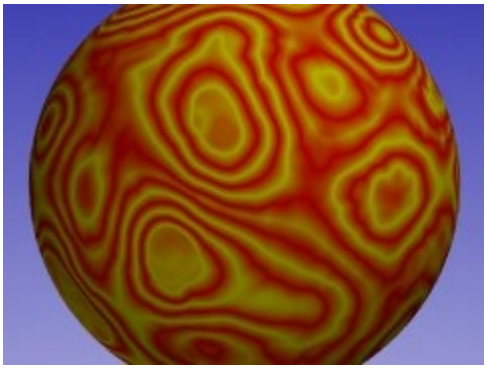
1. Lambda
2. Octaves
3. Omega
4. Offset
5. Ridge
6. Noise Type

The effects of these parameters are fairly subtle.

```
function {f_ridge (x, y, z, 1, 3, 1, 0.2, 0, 0)}
```







This is the Ridged Multifractal surface

The three images show

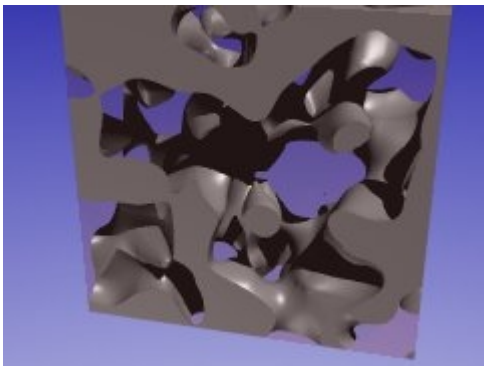
- •the function used alone as a surface
- •the function used as a height field on a sphere
- •the function used as a pigment on a sphere

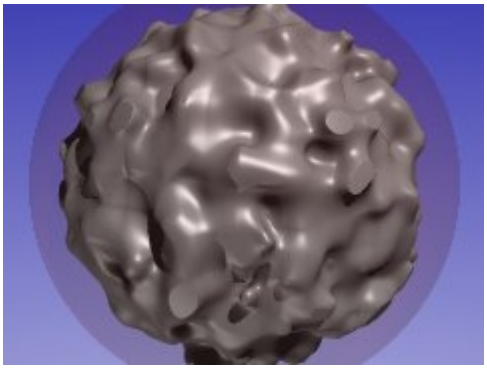
The parameters are:

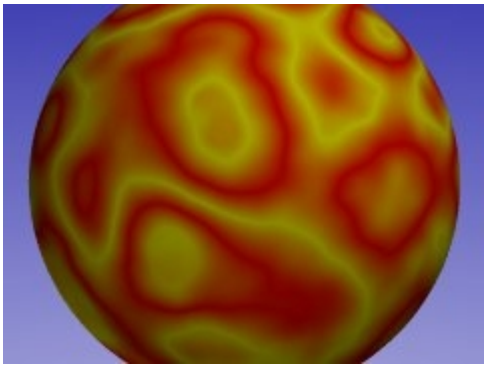
1.H
2.Lacunarity
3.Octaves
4.Offset
5.Gain
6.Noise Type

The effects of these parameters are fairly subtle, and are comprehensively described in the help file.

```
function {-f_ridged_mf (x, y, z, 2, 3, 1, 0.1, 1, 2)}
```







This is the Hetero Multifractal surface

The three images show

- •the function used alone as a surface
- •the function used as a height field on a sphere
- •the function used as a pigment on a sphere

The parameters are:

1. H
2. Lacunarity
3. Octaves
4. Offset
5. Heterogeneity
6. Noise Type

The effects of these parameters are fairly subtle, and are comprehensively described in the help file.

```
#declare F = function {f_hetero_mf (x, y, z, 2, 3, 1, 0.1, 1, 0)}
```

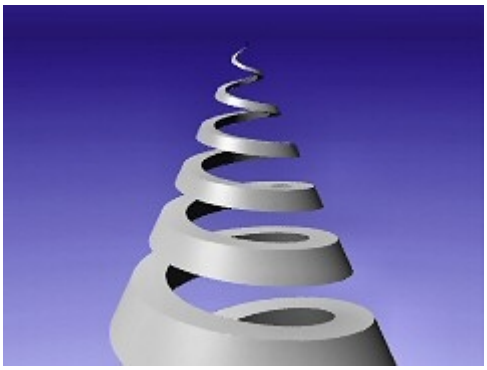
Variable Parameters

The parameters of the built-in functions don't necessarily have to be constants. It is possible to use variables, to cause the value of the parameter to vary across the surface.

```
function {f_helix1 (x, y, z, 1, 8*pi, 0.07, 0.4*(1-y), 1, 1, 0) }
```

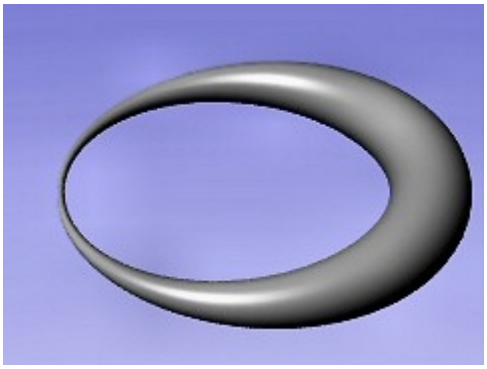
This is the built-in function f_helix1, but instead of using a constant for the major radius (like 0.4) I've used the expression 0.4*(1-y). This causes the major radius to vary from 0 to 0.8 as y goes from 1 to -1.

I've deliberately chosen the expression 0.4*(1-y) so that y doesn't become negative anywhere in the evaluated region.



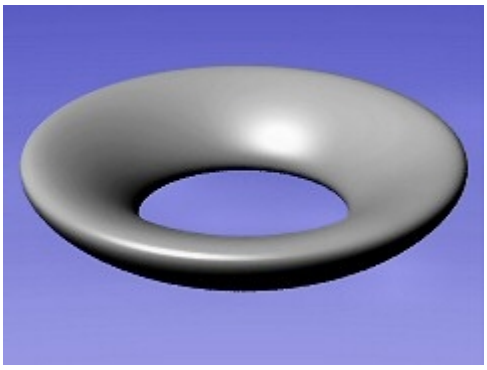
```
function {f_helix1 (x, y, z, 1, 10*(2+y), 0.07*(1-y), 0.4*(1-y), 1, 0, 0)}
```

In this scene the period, minor radius and major radius all vary as y changes.



```
function {f_torus (x, y, z, 0.5, 0.1*(0.6+x)) }
```

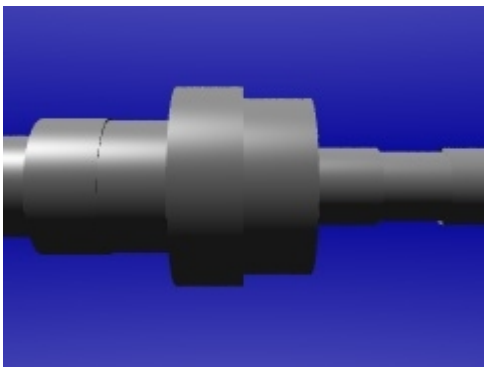
Changing the minor radius of a torus as a function of x can give an effect similar to the Dupin cyclid



```
function {f_torus (x, y, z, 1*(y+0.4), 0.1)}
```

Changing the major radius of a torus as a function of y can give an effect similar to an elliptical torus.

Using Arrays and Macros



Using Arrays

You can't index an array inside an isosurface function.

You can use individual fixed elements of an array, like A[3] but you can't use the function variables to index the array like A[floor(x)].

If you've got a reasonably small number of elements in your array, you can use nested select() operations to pick out the individual elements of the array. In this example the array has eight elements.

```
#declare Index = function (i) {
    select (i-4,
        select (i-2,
            select (i-1, A[0], A[1]),
            select (i-3, A[2], A[3])
        ),
```

```
        select (i-6,
            select (i-5, A[4], A[5]),
            select (i-7, A[6], A[7])
        )
    )
}
isosurface {function {F_cylinder (x, y, z, Index(x))}
```

This Index() function behaves rather like an array lookup, but it's going to be rather cumbersome to use this technique with large arrays. You're not restricted to using float arrays. If you wish, you can create arrays of functions, like this:

```
#declare A[0] = function {f_sphere (x, y, z, 1.0)}
```

And reference them like

```
function {A[0] (x, y, z)}
```

Using Macros

You can't pass function variables to a macro.

You can call a macro from inside an isosurface function, but it gets evaluated once, at parse time.

However, it is possible to use macros when you approximate a parametric surface, because Ingo's "param.inc" includes the ability to work with macros instead of functions. You can't use Ingo's Parametric macro, because it only works with functions, you have to call the underlying Paramcalc macro directly, which is a bit trickier.

When using Paramcalc you have to name your macros (or functions) __Fx, __Fy, __Fz, rather than passing them as parameters to the macro. The remaining parameters of Paramcalc are the same as those for Parametric.

The code inside your macros has access to the #local variables within Ingo's include file. This means that you have to be careful to avoid variable name conflicts if you want the macro to work with variables that are declared in your own code. E.g. if you #declare variables in your own code called 'A', 'B', 'C', 'D', 'P', 'U' or 'V' then you can't use them within the passed macros because Paramcalc has #local variables with the same names.

You can't use variables u, v, x, y or z inside your macros, even as their formal parameters, because POV interprets them as shorthand for the unit vectors when parsing macros.

You can use arrays and vectors inside your macros.

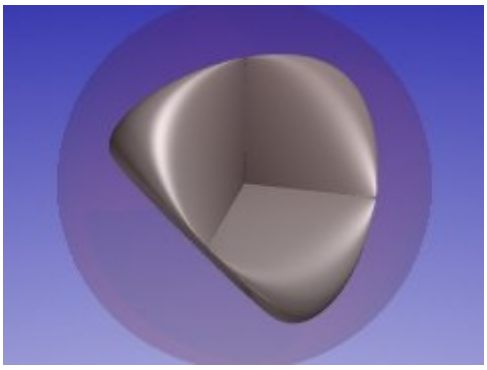
Your macros can call functions and other macros.

```
#macro __Fx (U,V)
    My_Function (U) * cos (My_B*U) + My_Array[U]
#end
#include "param.inc"
object{Paramcalc (<0, 0>, <2*pi, 2*pi>, 50, 40, "")}
```

Steiner Surfaces

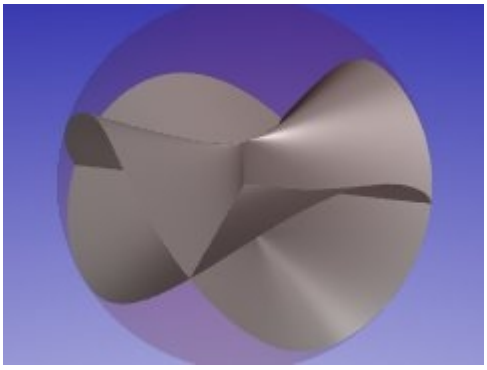
Steiner surfaces are representations of the projective plane, discovered by J. Steiner. For in-depth mathematical information about Steiner surfaces see http://www.ipfw.edu/math/Coffman/steinersurface.html.

All Steiner surfaces contain singularities of some form or other.



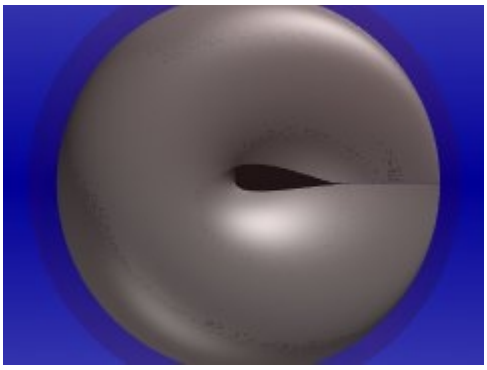
This is Steiner's Roman Surface. It has three double lines, six pinch points, and a triple point.

```
function {x*x*y*y + x*x*z*z + y*y*z*z - x*y*z}
```



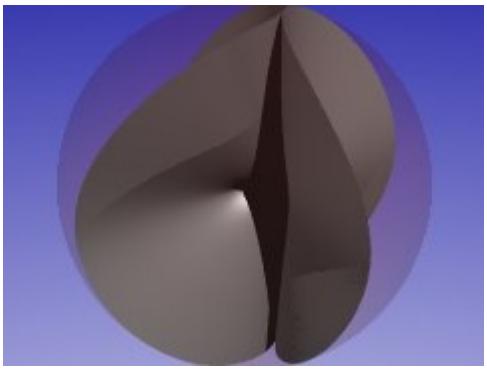
This surface has two pinch points, three double lines and a triple point.

```
function {x*x*y*y + x*x*z*z + y*y*z*z - x*y*z}   // TYPO?
```



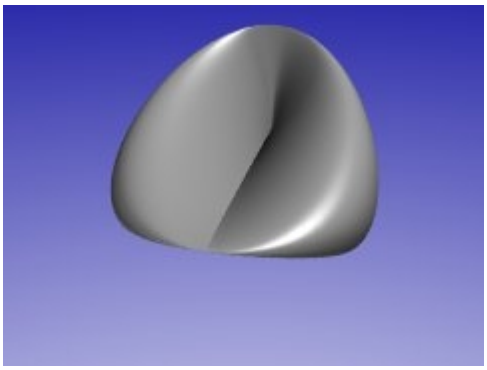
This is Steiner's Cross-Cap Surface. It has one double line and two pinch points. The pinch points are known as Whitney singularities.

```
function {4*x*x*(x*x + y*y + z*z + z) + y*y*(y*y + z*z - 1)}
```



This surface has a "tachnodal" line, a double line and two pinch points.

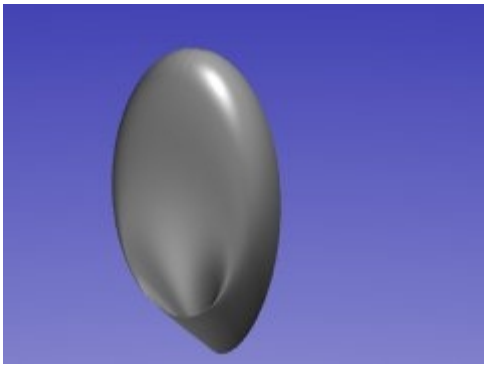
```
function {y*y - 2*x*y*y - x*z*z + x*x*y*y + x*x*z*z - z*z*z*z}
```



This surface has a "tachnodal" line, a double line and four pinch points.

For some reason, this surface didn't look anything remotely like what it was supposed to when I used function x*x*(z-1)*(z-1) +y*y*(y*y+z*z-1) so I've actually used a parametric isosurface to generate it.

```
#declare Fx = function {2*u*cos (v)*sqrt(1-u*u)}
#declare Fy = function {2*u*sin (v)*sqrt(1-u*u)}
#declare Fz = function {1-2*u*u*cos (v)*cos (v)}
parametric {
    function {Fx (u, v, 0)}
    function {Fy (u, v, 0)}
    function {Fz (u, v, 0)}
    <0, 0>, <1, 2*pi>
    contained_by {box {<-R, -R, -R>, <R, R, R>}}
    precompute 18, x, y, z
}
```

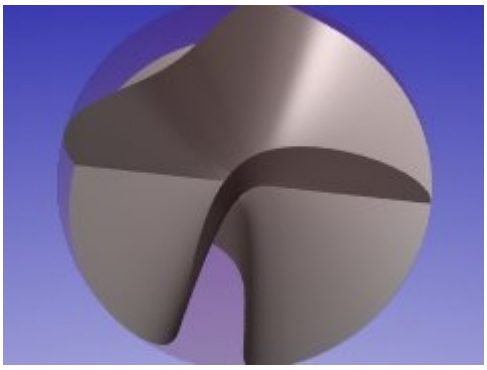


This surface has an "oscnodal" line. It has been called the "Cross Cup" due to its similarity to the cross cap.

For some reason, this surface didn't look anything remotely like what it was supposed to when I used a conventional isosurface so I've actually used a parametric isosurface to generate it.

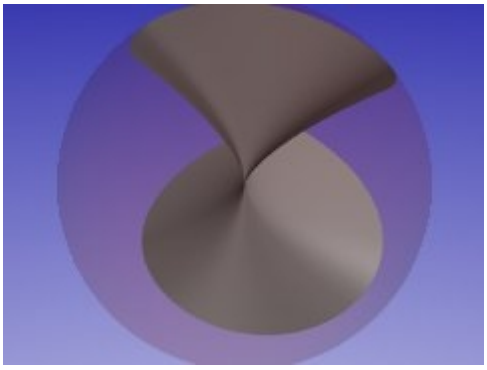
```
#declare Fx = function {1 - u*u + u*u*sin (v)*sin (v)}
#declare Fy = function {u*u*sin (v)*sin (v)
        + 2*u*u*sin(v)*cos (v)}
#declare Fz = function {sqrt ((1-u*u)/2)*u*(sin (v) + cos (v))}
parametric {
    function {Fx (u, v, 0)}
    function {Fy (u, v, 0)}
    function {Fz (u, v, 0)}
      <0, 0>, <1, 2*pi>
    contained_by {box {V1, V2}}
    max_gradient 10
    accuracy 0.00001
    precompute 20, x, y, z
```

Warning: This surface takes a ridiculously long time to render.



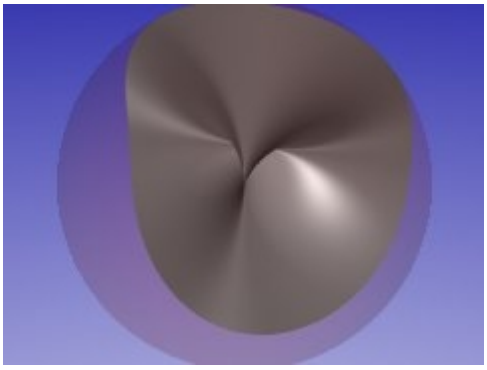
This surface has a double line and no pinch points.

```
function {x*y*y - y*z - x*z*z }
```



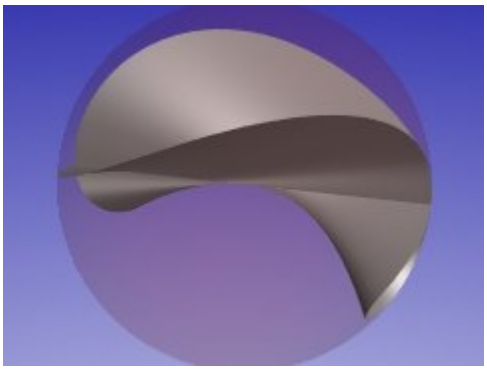
This is Whitney's Umbrella. It has two pinch points, one of which is at infinity.

```
function {x*x - y*y*z}
```



This surface is isomorphic to Whitney's Umbrella, but has the two pinch points close to the origin. It's called Plücker's Conoid.

```
function {x*x - x*x*z - y*y*z }
```

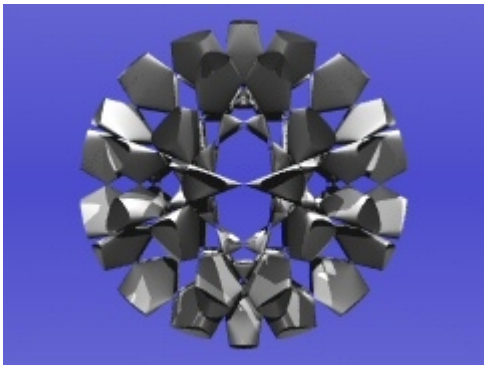


This is Cayley's Ruled Cubic. It has one double line and a singularity called a "unode," which is not a pinch point.

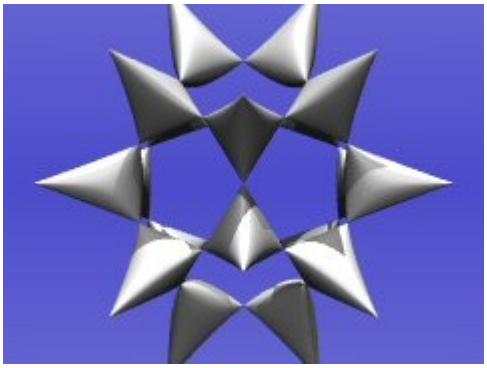
```
function {y*y*y + x*y*z - z*z}
```

Mathematical Zoo

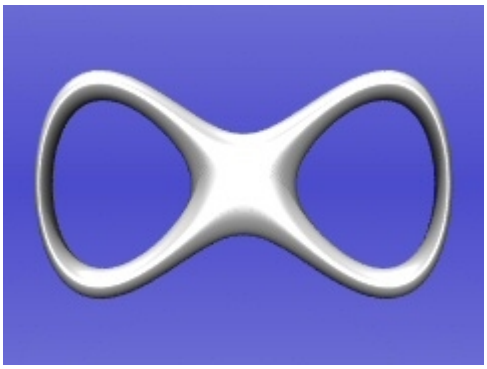
This is a collection of surfaces that mathematicians have found interesting for various reasons.
There are no new techniques here, and I don't have anything specific to say about any of the surfaces.
The source code is in the zip file at the end of the page.



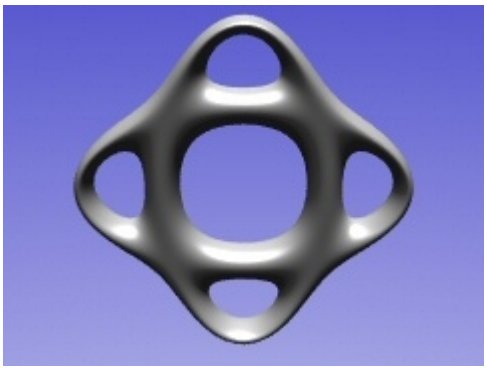
Barth's Decic



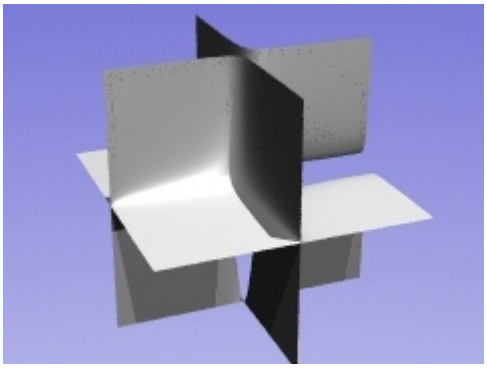
Barth's Sextic



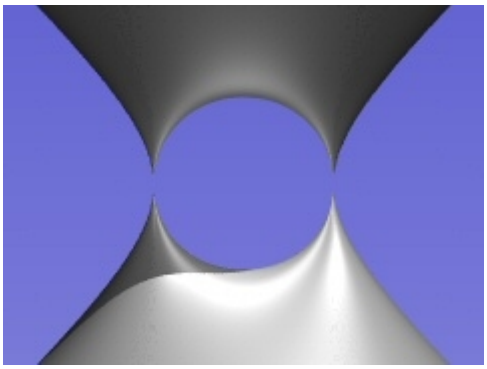
2-Hole Bretzel



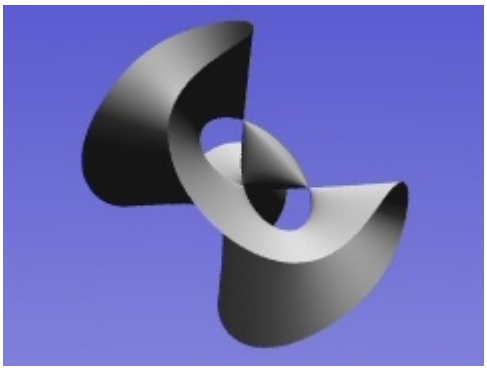
5-Hole Bretzel



Burkhardt's Quartic



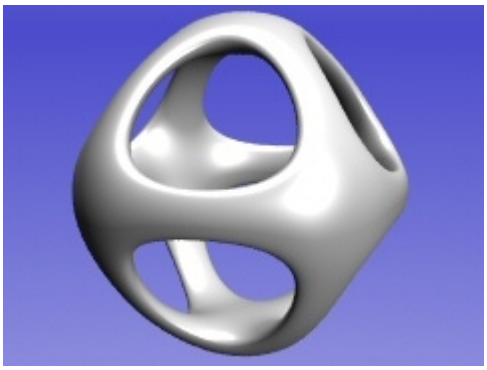
Cassini's Surface



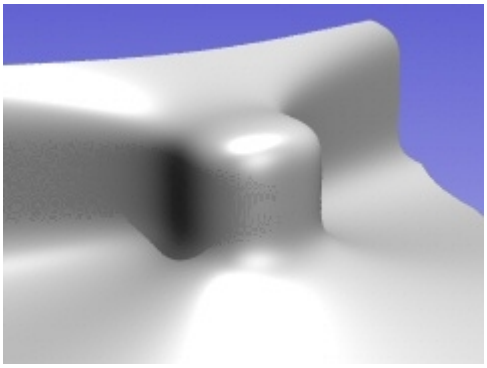
Cayley's Surface

DecoCube

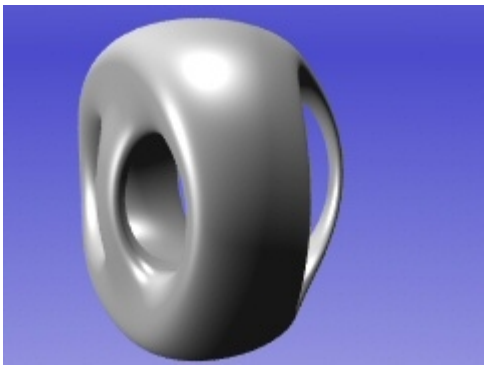
Goursat's Surface



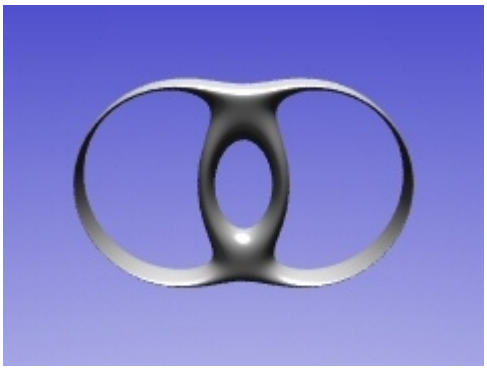
OrthoCircles



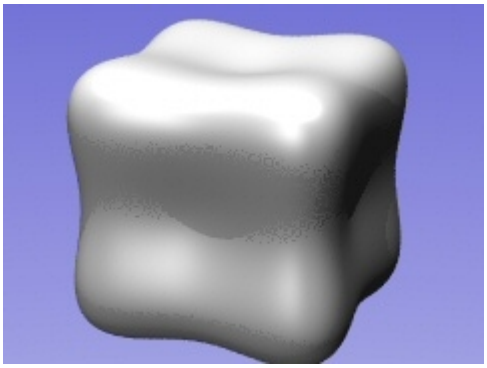
Peninsula Surface



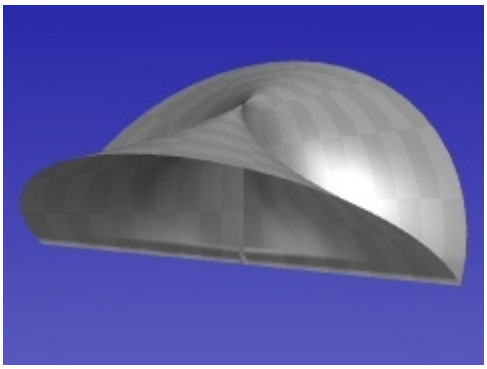
Pilz Surface



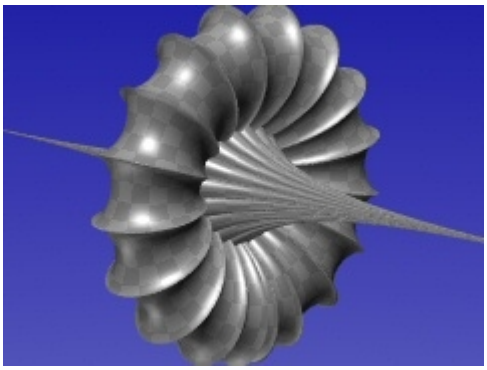
Pretzel



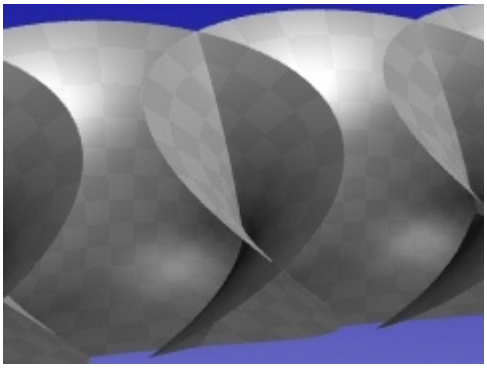
Tooth Surface



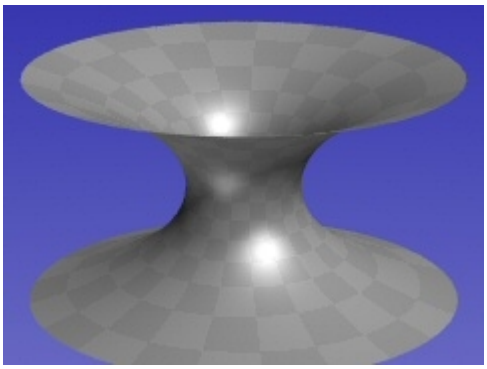
Wallis's Conical Wedge



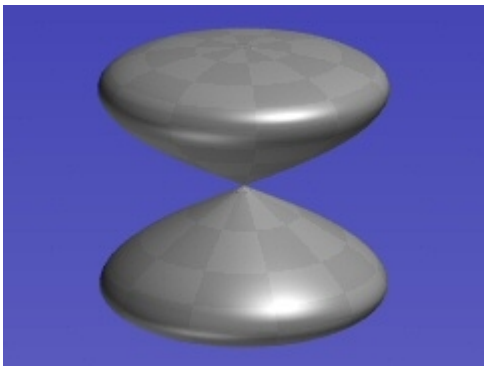
Breather Surface



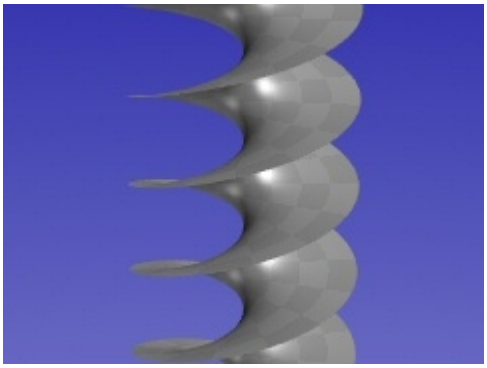
Catalan Surface



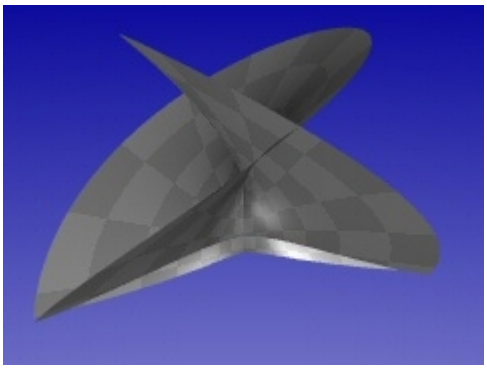
Catenoid - soap bubble surface



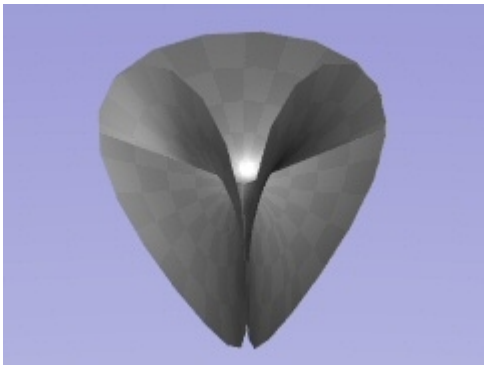
Eight Surface



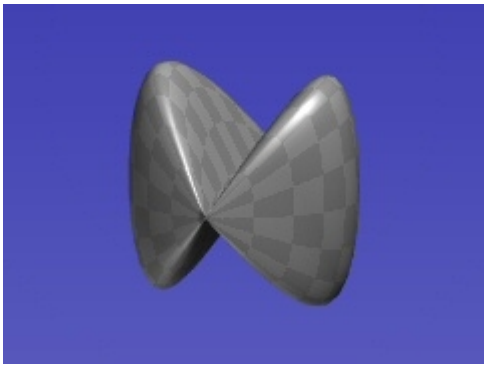
Helicoid - soap bubble surface for a helix



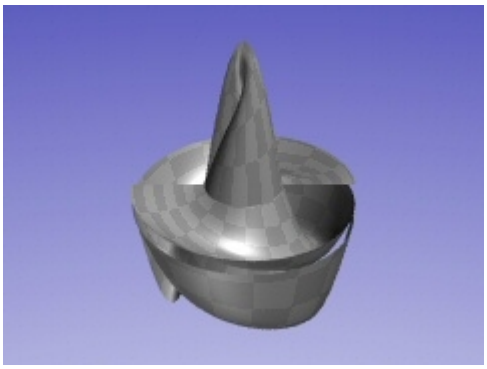
Henneberg's Minimal Surface



Maeder's Owl Minimal Surface



Pseudo Cross Cap



Steinbach Screw

"Realistic" Shapes

Well, they're slightly more realistic than most of the images in this tutorial.

Euphonium

This is a parametric isosurface using the functions:

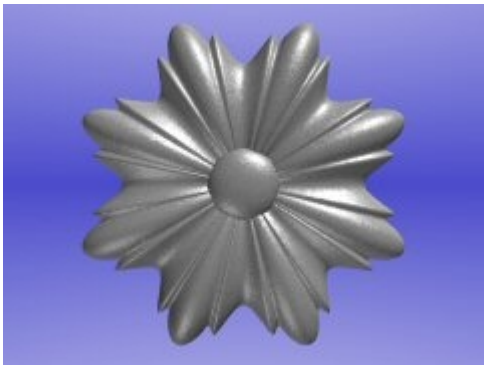
```
#declare Flare = function (u) {u/(2*pi) + pow (u/(2*pi), 20)*2}
#declare Fx = function (u, v) {(2 + u/(2*pi)*cos (v))*sin (u) - 0.1*u}
```

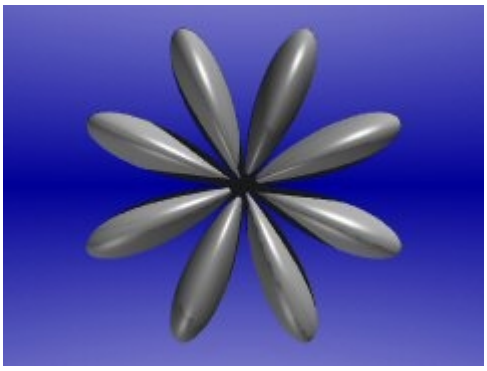
```
#declare Fy = function (u, v) {(2 + Flare (u)*cos (v))*cos (u) + 0.5*u}
#declare Fz = function (u, v) {Flare (u)*sin (v)}
```
Once again I started with a parametric torus. In this case I happened to start with

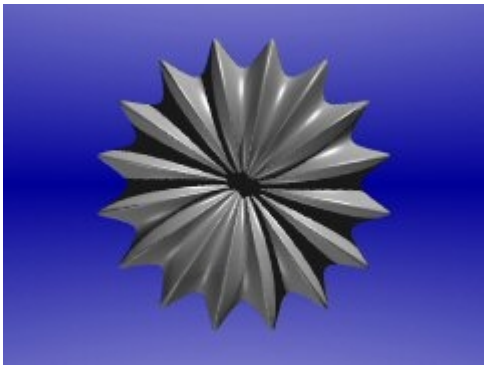
```
#declare Fx = function (u, v) {(2 + cos (v))*sin (u)}
#declare Fy = function (u, v) {(2 + cos (v))*cos (u)}
#declare Fz = function (u, v) {sin (v)}
```

I initially added **u/(2*pi)** factors to all three components, but then replaced these by **Flare(u)** in Fy and Fz. The **Flare(u)** function is designed so that it looks very similar to **u/(2*pi)** when u lies between 0 and about 1.8*pi, but increases rapidly between u=1.8*pi and u=2*pi. Thus causing the final widening at the top of the instrument in the Z and Y directions.

The final - 0.1*u and + 0.5*u factors fine tune the position of the top of the instrument.







Tudor Rose

This surface was inspired by a stylised rose decoration on a wrought iron gate.

I've cheated slightly by adding a flattened sphere to the centre. That sphere is not part of the actual parametric surface.

The functions are:

```
#declare R = function (u, v) {
    cos (v)*cos (v)
    * max (abs (sin (4*u)), 0.9 – 0.2*abs (cos (8*u)))
}
#declare Fx = function (u, v) {R (u, v)*cos (u)*cos (v)}
#declare Fy = function (u, v) {R (u, v)*sin (u)*cos (v)}
#declare Fz = function (u, v) {R (u, v)*sin (v)*0.5}
sphere {-0.2*z, 0.2 scale <1, 1, 0.5>}
```

This shape renders rather slowly as a real parametric isosurface, so I suggest using Ingo's param.inc macro to approximate it.

I started from the parametric functions for a sphere:

```
#declare Fx = function (u, v) {R*cos (u)*cos (v)}
#declare Fy = function (u, v) {R*sin (u)*cos (v)}
#declare Fz = function (u, v) {R*sin (v)}
```

Then replaced the constant **R** with the function **R(u,v)**.

There are three main parts to the function R(u,v). This bit

```
#declare R = function (u, v) {cos (v)*cos (v)}
```

controls the 2d shape we see if we take a vertical cross section. It's a sort of figure-8. By using this figure-8 to modify the radius of the sphere, we get something like a torus but with the central hole just filled in.

The abs(sin(4*u)) varies the radius as we go round the longitude of the sphere. So

```
#declare R = function (u, v) {cos (v)*cos (v) * abs (sin (4*u))}
```

gives the shape shown in the second image on the left, and

```
#declare R = function (u, v) {cos (v)*cos (v) * 0.9-0.2*abs (cos (8*u))}
```
gives the shape shown in the third image.

The max() operation takes the union of those two shapes.

Another Heart

The f_heart() shape tends to render with a subtle crease across the middle because the numbers involved in the calculation get rather extreme. Here's a different heart shape that's not quite as true to the conventional Valentine's shape, but doesn't have a crease.

```
#declare R = function (u, v)
   {4*cos (v)*pow (sin (abs (u)), abs (u)) }
#declare Fx = function (u, v) {R (u, v)*cos (u)*cos (v)*1.6}
#declare Fy = function (u, v) {R (u, v)*sin (u)*cos (v)}
#declare Fz = function (u, v) {2*sin (v)}
```

The functions are based on the parametric form of the sphere, but with the radius varying under the control of the **R()** function rather then being constant.

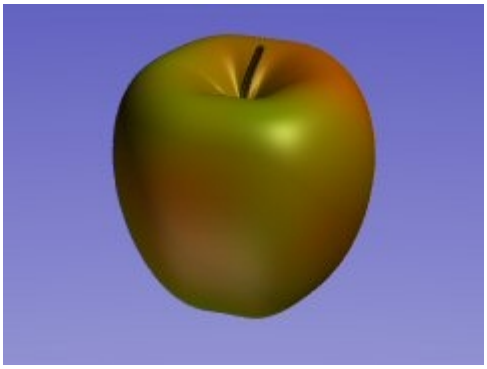
The *1.6 in Fx and the *2 in Fz are simply there to perform scaling. I could just as easily left them out of the functions and performed scale <1.6, 1, 2> on the completed surface.

Yet Another Heart

Here's another heart, and a heart hoop, that I discovered recently. They render extremely quickly and do look quite like the conventional Valentine shape.

These shapes are created by performing variable transformations on the standard f_torus() and f_sphere() functions. The y variable is replaced by y-pow(abs(x),0.8)*0.5 which causes the shape to be bent upwards.The z variable is replaced by z*2 in the f_sphere() to perform simple scaling.

```
function { f_torus  (y-pow (abs (x), 0.8)*0.5, z, x, 0.8, 0.1) }
function { f_sphere (y-pow (abs (x), 0.8)*0.5, z*2, x, 0.6) }
```



Apple

This surface is generated by the functions

```
#declare Fx = function (u, v) {cos (u)*(R1 + R2*cos (v))
                    + pow((v/pi),100)}
#declare Fy = function u, v) {sin (u)*(R1 + R2*cos (v))
                    + 0.25*cos (5*u)}
#declare Fz = function (u, v) {-2.3*ln (1 - v*0.3157)
                    + 6*sin(v) + 2*cos (v)}
```

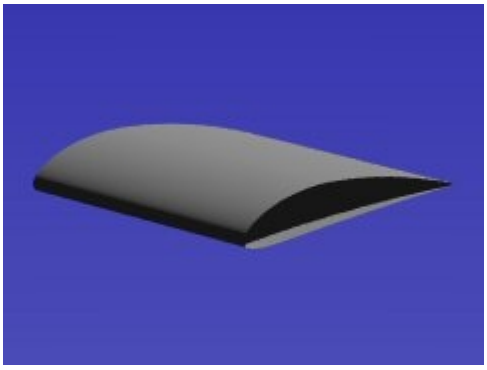
The stalk of the apple really is part of the surface.

The shape is based on a fat torus, but with the tube part of the torus made slightly elliptical and tilted by means of the +2*cos(v) at the end of the Fz function.

The ln(1 - v*0.3157) bit in Fz becomes significant when v approaches pi (0.3157 is just a little lower than 1/pi). It creates the stalk.

The + pow((v/pi),100) bit in Fx adds a curve to the stalk. The effect only happens when v is very close to pi, so when using Ingo's macro to approximate the parametric it is necessary to use a large number of segments in the v direction.

The +0.25*cos(5*u) bit in Fy adds a gentle ripple to the shape.



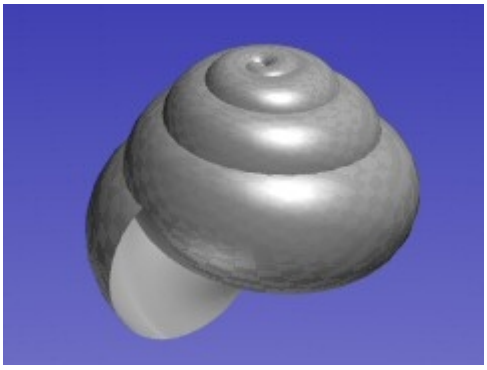
NACA 4-digit airfoil section

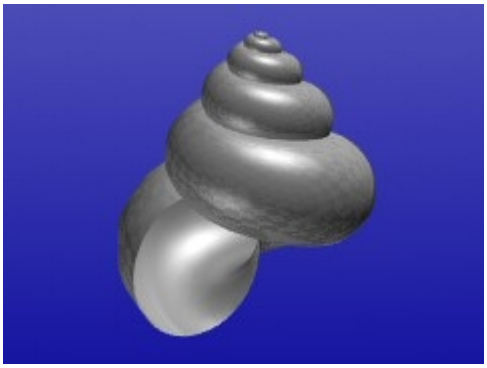
The National Advisory Committee for Aeronautics published a system of standard airfoils designated by 4-digit and 5-digit codes. This parametric isosurface is an approximation to the NACA5415 aerofoil.

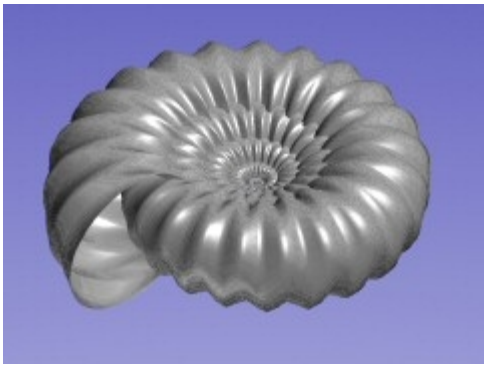
The code can be used to generate any NACA 4-digit aerofoil by changing the settings of "M", "P" and "T".

A description of the system can be found at www.aerospaceweb.org/question/airfoils/q0100.shtml.

Seashells







Seashell surfaces

These are parametric isosurfaces using variations of functions like:

```
#declare W  = function (u) {u/(2*pi)}
#declare Fx = function (u, v) {W (u)*cos (N*u)*(1+cos (v))}
#declare Fy = function (u, v) {W (u)*sin (N*u)*(1+cos (v))}
#declare Fz = function (u, v) {W (u)*sin (v)  + H*pow (W (u), 2)}
```

Where

- H controls the height of the shell
- N is the number of turns

u/(2*pi) simply gives a value that goes linearly from 0 to 1 as u goes from 0 to 2*pi. It turns up several times in the functions, so I've made it into a separate sub-function W(u)

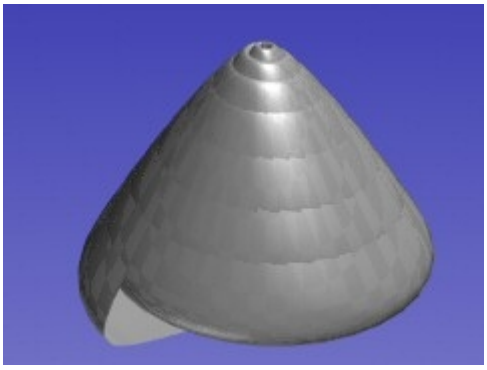
I started with the parametric functions for a torus:

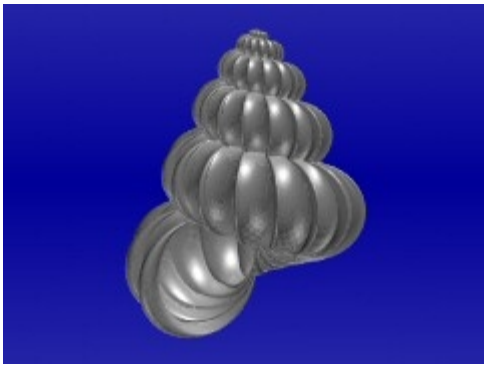
```
#declare Fx = function (u, v) {cos (u)*(1+cos (v))}
#declare Fy = function (u, v) {sin (u)*(1+cos (v))}
#declare Fz = function (u, v) {sin (v)}
```

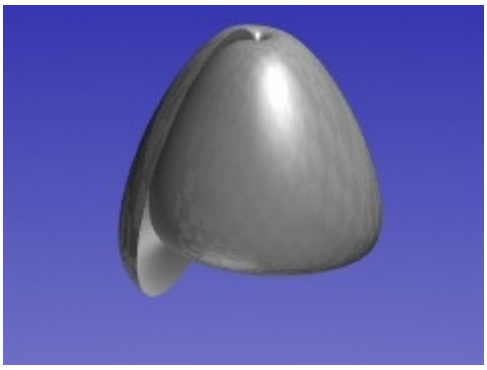
The first parts of Fx, Fy and Fz are multiplied by **R(u)** so that the radius of the tube increases linearly as u increases.

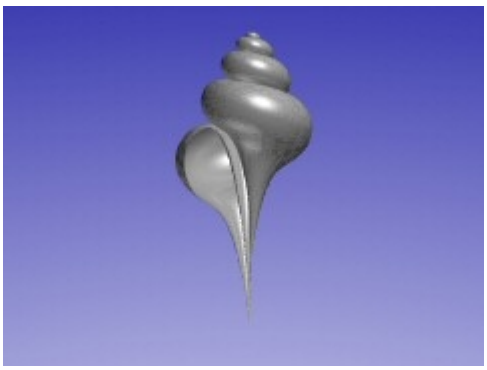
The N*u factors that occur in Fx and Fy cause the tube to go round the origin N times.

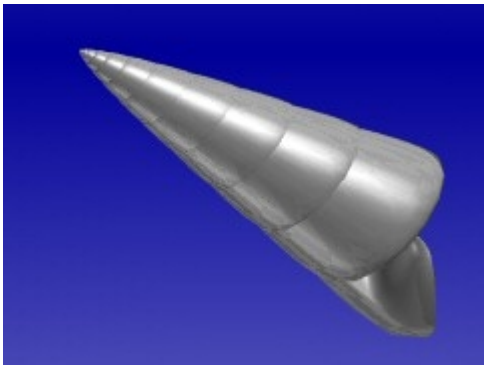
Then I added +H*pow(W(u),2) to Fz so that the turns of the spiral are offset in the z direction by a distance that varies from 0 to H. I found that I needed to square the u/(2*pi) term, otherwise the offset was too rapid at the start.



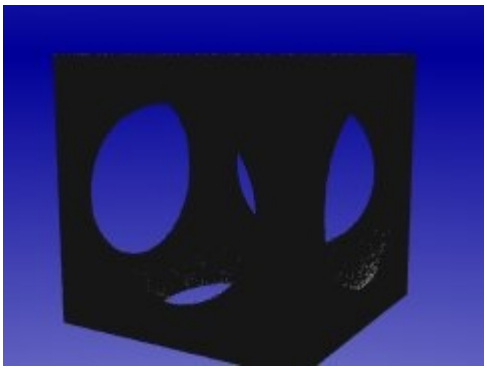








Things That Don't Work



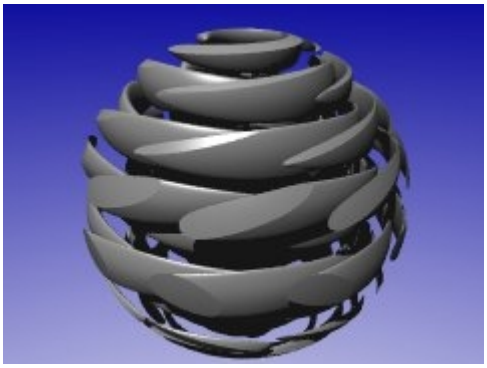
```
#declare Thing =
  difference {box {-1, 1}
      sphere {0, 1.2}
  }
#declare P = function {pattern {object {Thing}}}
#include "functions.inc"
isosurface {
  function {P (x, y, z) - 0.999}
  max_gradient 100
  contained_by {box {-1.1, 1.1}} open
```

It might seem like there would be all sorts of interesting effects that could be achieved by using the "object" pattern.

function {pattern {object {My_Object}}}
is a function that returns the value 1 for all points that are inside My_Object and the value 0 for all points that are outside My_Object. So we might think that we could make an isosurface from that function and then modify it in interesting ways, for example by adding a noise function.

Unfortunately, any path through the isosurface encounters a point where the value jumps instantly from 0 to 1, so the actual max_gradient of every point on the surface is infinite. Instead of getting a few nasty little holes in our nice isosurface, we get a few tiny patches of surface and the rest is hole, unless very extreme values of max_gradient are used (and then it takes ages).

Also, the idea of adding noise to the surface doesn't work quite like what you'd intuitively expect. Adding a noise function to the object pattern function gives one region of space where the function evaluates to noise+1.0 and in the rest of space it evaluates to noise+0.0. One thing we could do is to apply a turbulence warp to the object pattern, but that's not so easy to control and still suffers from infinite max_gradient.



If you think about max_gradient for a while you might get the idea that something with a low max_gradient will always render faster than something with a higher max_gradient. If this were the case, then we could speed things up by artificially reducing the gradient of the function.

For example, if we consider

```
isosurface {
  function {0.3 - F (x, y, z)}
        max_gradient 10
```

we might notice that we can obtain exactly the same surface by using

```
isosurface {
  function {(0.3 - F (x, y, z))*0.1}
        max_gradient 1
```

or even

```
isosurface {
  function {(0.3 - F (x, y, z))*0.01}
        max_gradient 0.1
```

These do in fact produce the same image when rendered. However, although the max_gradient is very different, the rendering times are exactly the same. You can't use vectors in an isosurface function.

You can't even extract float values from a vector by using things like V.red

---

You can't index an array inside an isosurface function.

You can use individual fixed elements of an array, like A[3] but you can't use the function variables to index the array like A[floor(x)].

However, you can do this.

You can't pass function variables to a macro.

You can call a macro from inside an isosurface function, but it gets evaluated once, at parse time.

Don't get confused by the fact that you can use "x", "y" and "z" in your macro. "x", "y" and "z" have two different meanings in POV-Ray, and inside a macro they are always interpreted as having the other meaning: shortcuts for the unit vectors.

You can, however, use macros with Ingo's "param.inc" file, like this

Alphabetical Index

Back to the Isosurface Index

Back to the Isosurface Index

Back to the Non-Printer-Friendly Isosurface Index