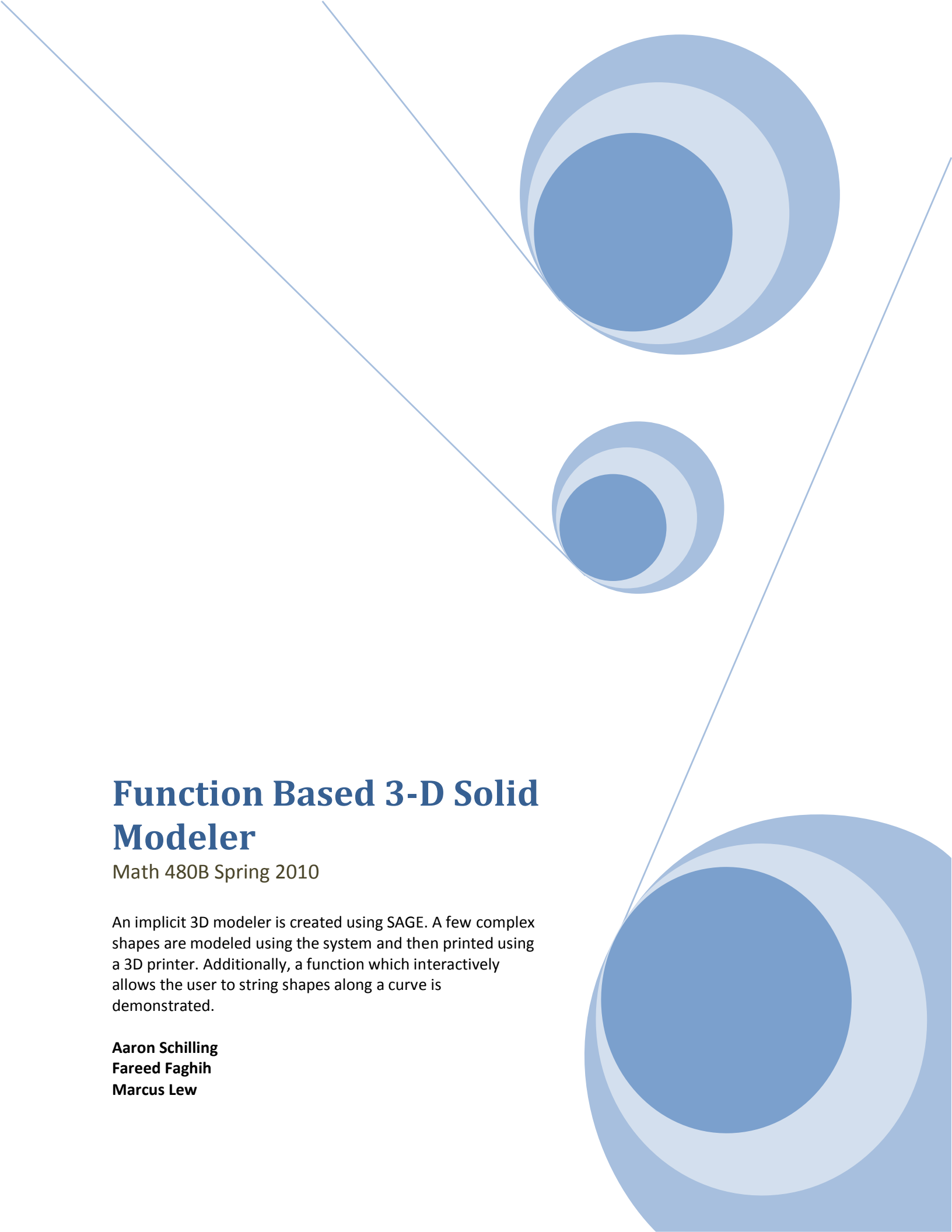


Function Based 3-D Solid Modeler

Math 480B Spring 2010

An implicit 3D modeler is created using SAGE. A few complex shapes are modeled using the system and then printed using a 3D printer. Additionally, a function which interactively allows the user to string shapes along a curve is demonstrated.

Aaron Schilling
Fareed Faghih
Marcus Lew

Background

Recent development of rapid prototyping technology, in conjunction with solid 3-D modeling, has given the engineering world the ability to turn an idea into a tangible object in a very short time period. The first step of this process is to design the concept. Originally done on paper, even computer-aided design (CAD) has had difficulties making adaptations from two to three dimensions. The modeling program outputs a universal file format from stereolithography called STL. After electronic design, the part is input in 3-D printing software that breaks the solid into a stack of layers for construction. Three dimensional printers have the ability to print in many different materials, ranging from plastics to powders and resin, and even glass. The object has then become a workable object that can be used for some limited applications.

All solid modeling programs use mathematical principles to generate three dimensional geometries. However, they do not all utilize the same methods. Common types of solid modelers use the methods boundary representation (B-Reps) or constructive solid geometry (CSG). B-Reps are based on the idea that the boundaries are represented by a collection of implicitly defined patches based on parameters. CSG uses primitive solids and Boolean commands to form their objects. The solid modeler that we have developed is a combination of the two. It can be described as an implicit representation or function based modeler. The shapes and Booleans are based on functions that implicitly define the objects' x-, y-, and z-coordinates. After developing objects, the three dimensional representations were then displayed with implicit_plot_3d, converted to STL using the function surface_to_stl, developed by Christopher Olah, and printed in University of Washington's Solheim RP Lab.

Methods

The code to achieve effective functional modeling is not very complex but provides a powerful tool. To begin, three variables, x, y, and z, are defined as variables which represent the three axes on which the modeling will be done. This provides an easy and convenient method to both produce the models as well as to allow for various transformations to create complex models.

The primitives and other basic shapes are defined as surfaces. This allows them to be defined as mathematical equations using the previously mentioned variables. The primitive shapes that were chosen were the ellipsoid, the prism, the cylinder, and the cone. By using various transforms these shapes can be modified into almost any imaginable shape. Additional shapes were also included such as the sinus and the drope. These shapes could be built from the primitives but would be time consuming for the designer. Instead, including the various unique shapes in a library allows the designer to pick from a more exact shape rather than starting

from scratch. Any shape defined by a closed surface with an implicit function can easily be added to the shape library improving the usefulness of our modeler.

The translation functions are the first of the transformations. Each function moves the surface which it is passed in a specific direction. This is achieved by substituting out each variable value with one that has had a constant subtracted from it resulting in a translation. The scaling functions work similarly in which all the values in one direction are divided by a constant resulting in the growth or shrinking of the surface in that direction. An additional function which scales in all directions also exists which scales the surface equally in all directions maintaining the aspect ratio of the surface. Another useful group of transformations are reflections. Three functions exist which will reflect the surface across the XY plane, the XZ plane, or the YZ plane. This is done by replacing the variable in the direction perpendicular to the plane with its opposite, remapping the point to the opposite side of the plane. The modeler also allows rotations in a specific direction. The rotations are achieved by using trigonometric ratios to determine how the points not in the plane of rotation are shifted. Both radian and degree values are accepted and specific functions exist for each as well as a conversion function to switch between the two. These four basic types of transformations allow the designer to create a near infinite amount of shapes from a simple starting shape.

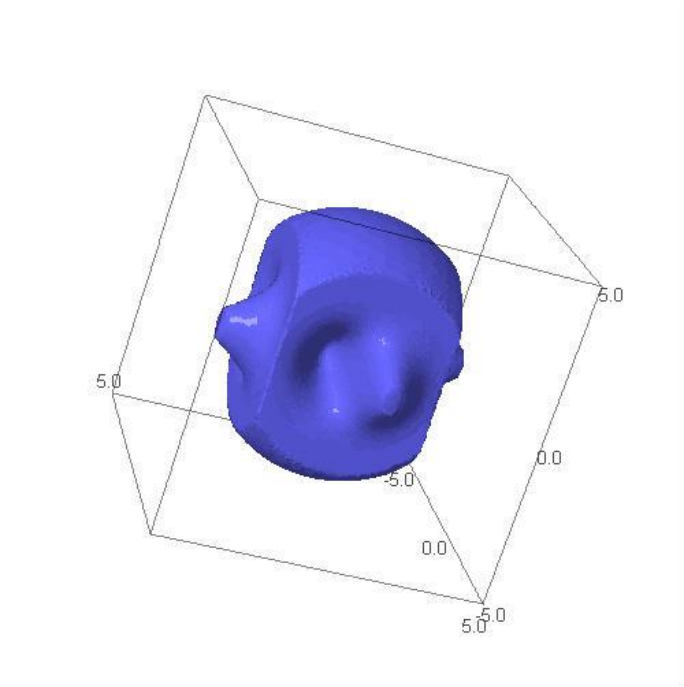
Once the desired shape has been created the designer may want to add or remove a section to the shape. For these operations various Boolean operators are needed. In this modeler the first Boolean operation is **Union** which joins two shapes together. The function takes the two shapes and adds them together in space very simply. If the designer wanted to only retain the shape formed from the intersection of two shapes then **Intersect** should be used. This method removes any material that is not shared between the two starting shapes. To remove one shape from another the **Subtract** method is useful. This method takes a shape and removes any section of that shape which the second shape overlaps. These three basic functions allow the designer the versatility to piece together various shapes to form a complex object. In addition to these three shape interaction functions, three more advanced shape interaction function are defined, **Morph**, **Blend** and **Combine.** Each provides its own unique functionality to further empower the designer.

To allow the designer to quickly and easily see the model which is being produced **Plot3d** was defined. It allows the user to quickly call **implicit_plot3d** without needing to specific the variables and the max and min of each axis. Instead, an isometric view is produced with axis length being specified by the user. The functionality in specifying the number of points to plot remains.

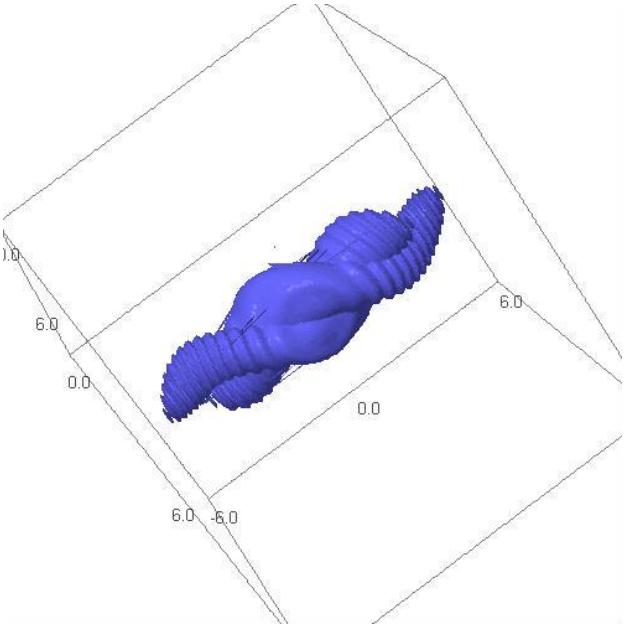
The function that converts from an image to a three dimensional STL representation was originally developed by Christopher Olah. An STL file is a collection of triangles represented by

a normal and three vertexes. We adapted the original function's output to return a file of type *.stl* without rapidly displaying all of its contents. The function now accepts both an implicit three-dimensional plot of an object and the filename.

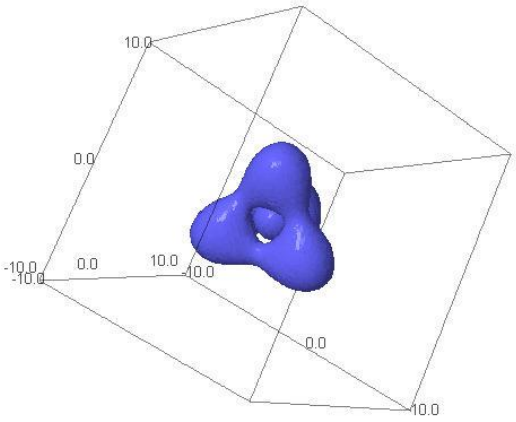
Using our function based 3D modeler various objects were created. The first complex object is the drope block. Its interlocking pin and divot shape make it useful as an interlocking piece. To create the piece first a sphere was created acting as the base for the shape. Then the drope shape was created and shifted away from the origin. The drope shape was then rotated and duplicated so that 4 copies forming a tube drope shape created. This was done using both the rotate and the intersect functions. To complete the figure the drope tube and the original sphere were intersected.



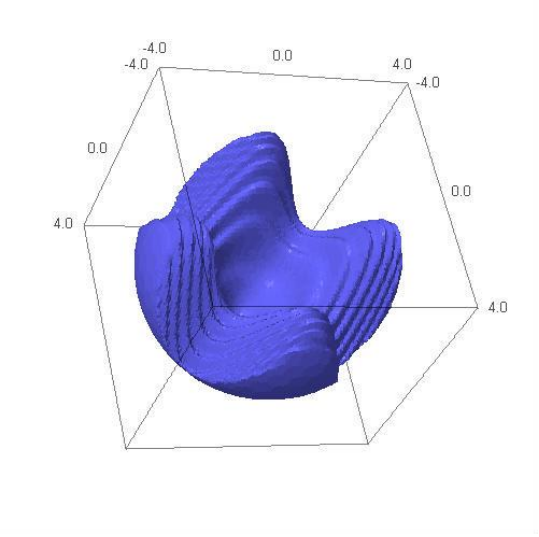
The sinus bug was created using the bug shape which resembles a coiled worm and the sinus which is periodic. Two interlocking bugs were created and then joined using **Union**. To incorporate the periodic features of the sinus the **Combine** function was used to merge the two bugs and the sinus.



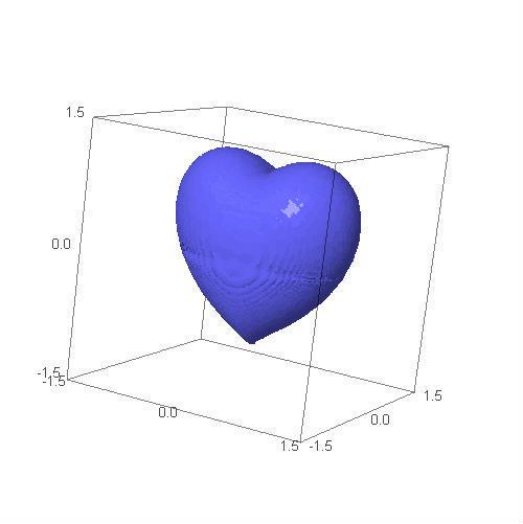
The super wiffle consists of a sphere trapped inside of a tetrahedron and showcases the powers of additive printing. Traditional fabrication makes it difficult to trap a sphere inside the larger shape. To make the model, a large tetrahedron is made and is joined to a smaller sphere using **Union**. It is interesting to note that the sphere and tetrahedron don't intersect at all.



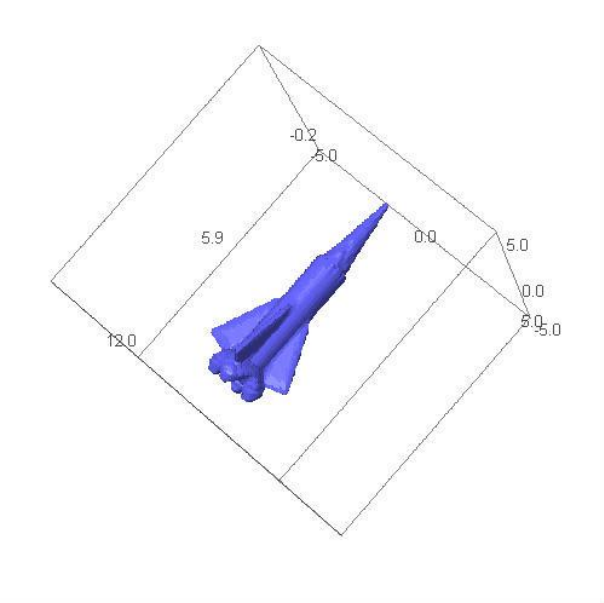
The sinus sphere is created using a sinus which is modified with a periodic function. This creates the characteristics of a step function in one direction with the periodic functions nature in the other. Intersecting the modified sinus function with a sphere creates a unique shape which can be used to create interlocking half spheres.



The heart is created very simply by using the heart shape in the shape library.



The spaceship is created using a combination of the prism, cone, cylinder, and the sinus sphere. First the fins are created. A single fin is formed by creating a prism, then shifting it away from the origin in the x direction and then rotating it along the x axis. To create the other fins the first fin is rotated pi around z axis and then joined to itself. This double fin is rotated pi/2 in the z and joined to itself creating the 4 fin figure. The cone, the cylinder, the four fins and the sinus sphere and joined together with **Union** after being shifted in the z direction so that they align properly. The design of the spaceship shows that our 3d modeler has the capability to create complex objects in fairly straightforward methods.



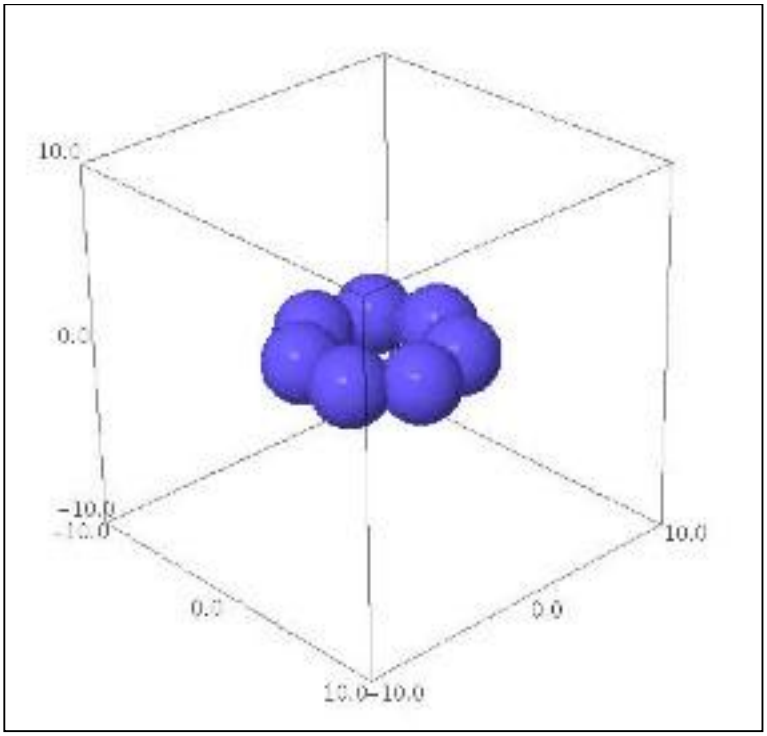
When building composite shapes by rotating, translating, and joining several shapes, the amount of time and thought that is necessary to build them can become very great. Thus, it is beneficial to create an automated system that can help build these composite shapes after only entering a few parameters. The first function that was developed was the **curve_fol_sphere** function, which takes a parametric function (f(t) = [x(t),y(t),z(t)]), the number of spheres desired (n), the first and last t values (a,b), and the radius of the spheres (radius) as inputs and returns a composite shape that consists of n spheres, equally spaced along the curve defined by f , between f(a) and f(b). This is done using the simple **trans_point** function (see below) that was created to evaluate the parametric function at t = s and determine the translation distances.

```
def trans_point(r,s): return f[r](t=s)
```

Then using the translation functions and the **Union** function, **curve_fol_sphere** builds the composite shape along the parametric curve. This composite shape can then be plotted. For example:

```
t = var('t')
f = [4*sin(t),  4*cos(t), 0*t]
circ=curve_fol_sphere(f,7,(0,2*pi),2)
Plot3d(circ,10,100)
```

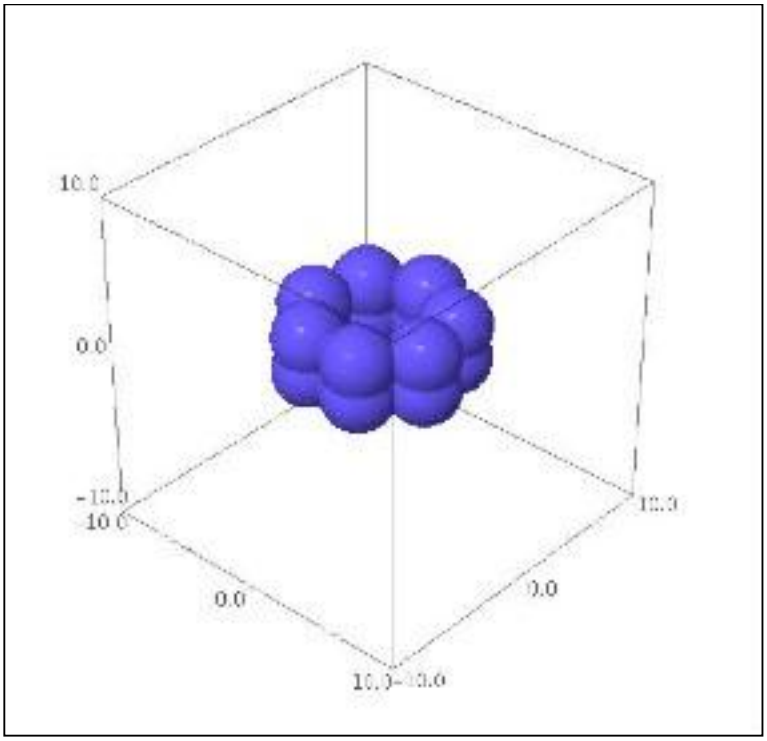
Returns:



The shape that is returned by the **curve_fol_sphere** function can then be used as building block for another composite shape. For example:

```
t = var('t')
f = [4*sin(t),  4*cos(t), 2*t/t]
circ2=curve_fol_sphere(f,7,(0,2*pi),2)
Plot3d(Union(circ,circ2),10,100)
```

Returns:

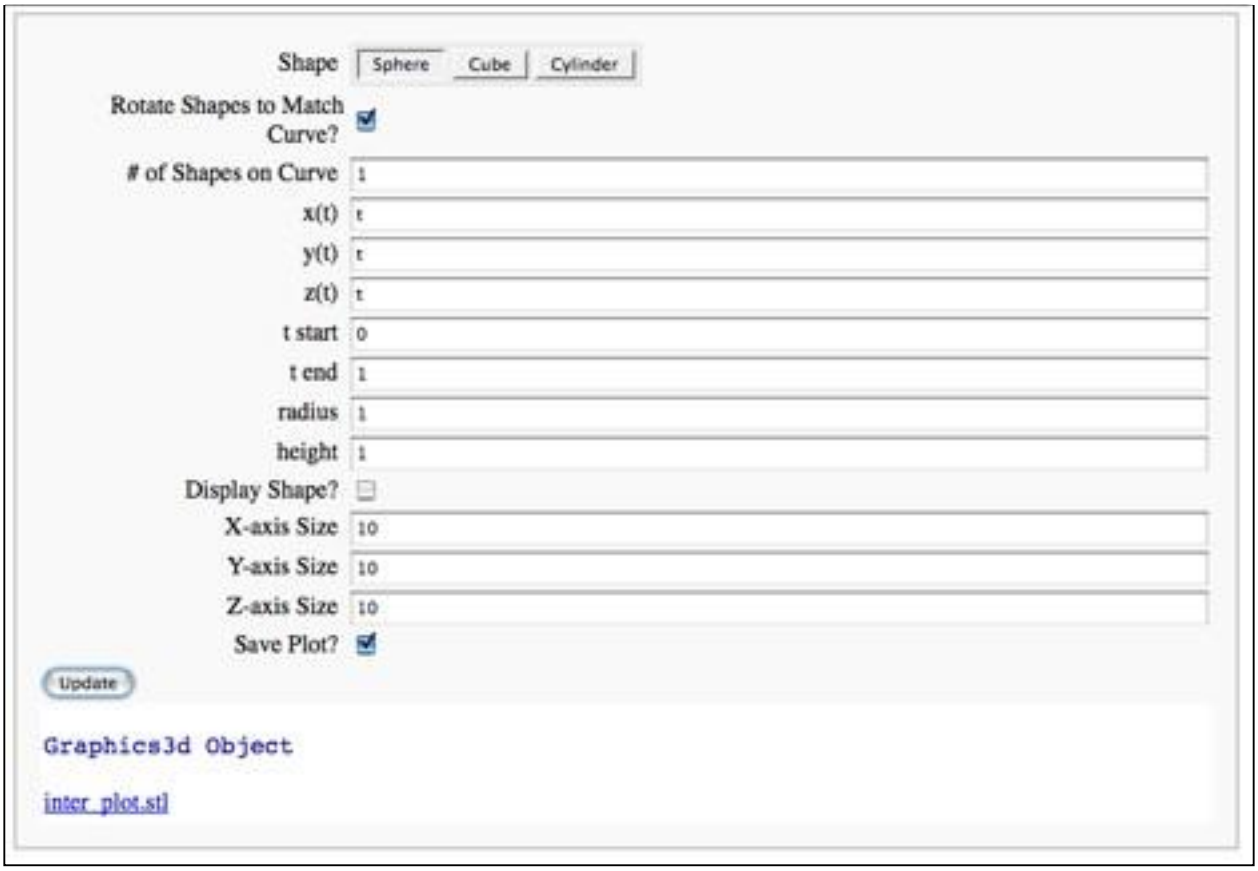


The code for the **curve_fol_sphere** function can be easily adapted to place any shape along the parametric curve. Specifically, we have developed **curve_fol_cyyl** and **curve_fol_box**, which place cylinders and cubes, respectively, on the parametric curve. These take the same inputs as **curve_fol_sphere**, with the exceptions that in **curve_fol_cyyl**, instead of radius it takes (c,d,e) where c and d correspond to the radii of the base and e corresponds to the height of the cylinders, and in **curve_fol_box**, size is used instead of radius and corresponds to the edge length of the cubes. The most significant difference between **curve_fol_sphere** and **curve_fol_cyyl** and **curve_fol_box** is that the **curve_fol_cyyl** and **curve_fol_box** functions not only move the shapes to the different points on the curve, but also rotate them to match the

tangent to the curve at each point. The rotation angles are calculated by the **angle** function (see below) that was created. This function calculates the tangent vector at the point f(t = v) and projects it onto the xy, yz, and xz planes when u = 0, 1, or 2 respectively. Then it finds the angle between the projection and the respective x, y and z unit vectors using arctangent of the slope.

```
def angle(u,v):
   if diff(f[u])(t = v) == 0:
   return pi/2
else:
   return arctan(diff(f[(u+1)%3])(t = v)/diff(f[u])(t = v))
```

In order to facilitate generating composite shapes quickly and easily, an interact was created (see below) using the Sage Notebook. This interact has many options and fields to fill. The first option is to select the shape to be used in the curve follower. Next, the user can select to have the shapes rotated to match the curve or not. Then the number of shapes and the functions for x(t), y(t) and z(t) are entered, as well as the minimum and maximum t values. The radius and height fields determine the size of the shapes. Checking the box labeled “Display Shape” will display the **implicit_plot3d** of the composite shape and the final three fields determine the scale of the plot. Finally, checking the box labeled “Save Plot” will save the plot as a *.stl* file. In order to prevent the interact from performing unnecessary evaluations, an update button is utilized so that all of the desired parameters can be entered and the interact only evaluates when the user clicks the update button.



Conclusion

The work presented in this report shows both the power of sage as well as the power of 3D printing. A function based 3D solid modeler was created which was then used to draw various parts. From these models solid objects were later printed. The current limitation of this project is its user interface. Currently to create an object the transforms must be coded. It would greatly benefit the user to create an interactive front that could be used to create the models. All in all, the project was useful and allowed us to design and produce our own objects.

Appendix

Sage Code

The following is published at **http://480.sagenb.org/home/pub/22** .

```
<p style="text-align: center;"><span style="text-decoration: underline;"><span style="font-size: xx-large;">Function Based 3-D Solid Modeller</span></span></p>

<p style="text-align: center;"><span style="font-size: large;">Aaron Schilling, Fareed Faghih, Marcus Lew</span></p>

<p style="text-align: center;"><span style="font-size: medium;">Math480B Final Project -- 6/4/2010</span></p>
```

```
<p style="text-align: center;"><span style="font-size: medium;"><img
src="http://upload.wikimedia.org/wikipedia/commons/0/0f/Jack-in-
cube_solid_model%2C_light_background.gif" alt="" width="180" height="274" /></span></p>

<p style="text-align: center;"><span style="font-size: xx-large;"><span style="font-family:
verdana, geneva;">3D Model of a Jack in a Cube</span></span></p>

<p style="text-align: center;"><span style="font-size: xx-large;"><span style="font-family:
verdana, geneva;"><img src="http://upload.wikimedia.org/wikipedia/commons/7/71/Cad-fm01s.gif"
alt="" width="338" height="308" /></span></span></p>

<p style="text-align: center;"><span style="font-size: xx-large;"><span style="font-family:
verdana, geneva;">3D Modelling Software in action</span></span></p>

<p style="text-align: center;"><img src="http://reprap.org/mediawiki/images/1/1f/Mendel.jpg"
alt="" width="850" height="550" /></p>

<p style="text-align: center;"><span style="color: #0000ff;"><span style="font-size: xx-
large;"><span style="color: #3366ff;"><span style="color: #000000;"><span style="font-family:
verdana, geneva;">Self-Replicating RepRap 3D Printer</span></span></span></span></span></p>

<p style="text-align: center;"> </p>

<p style="text-align: center;"><img
src="http://www.itg.uiuc.edu/printing/3D/3d%20printer%20open.jpg" alt="" width="500" height="375"
/></p>

<p style="text-align: center;"> </p>

<p style="text-align: center;"><span style="font-size: xx-large;"><span style="font-family:
verdana, geneva;">Commercial Z406 3D printer</span></span></p>

<p style="text-align: center;"><span style="font-size: medium;"><img
src="http://upload.wikimedia.org/wikipedia/commons/6/60/3D_scanning_and_printing.jpg" alt=""
width="800" height="754" /></span></p>

<p style="text-align: center;"><span style="font-size: xx-large;"><span style="font-family:
verdana, geneva;">Example 3D scan and 3D print</span></span></p>

<p style="text-align: center;"><span style="font-size: medium;"><br /></span></p>

sage: %auto

sage: #Define Variables

sage: x,y,z = var('x, y, z')

sage: #Transformations: Mathematical functions to change the geometrical size or location of
solids. Includes translations, scaling, reflection and rotation.

sage: #Translations: Movement in the specific direction

sage: def TransX(a,tx) : return a.substitute(x=(x-tx))

sage: def TransY(a,ty) : return a.substitute(y=(y-ty))

sage: def TransZ(a,tz) : return a.substitute(z=(z-tz))

sage: #Scaling: Size factor change in specific direction

sage: def ScaleX(a,sx) : return a.substitute(x=(x/sx))

sage: def ScaleY(a,sy) : return a.substitute(y=(y/sy))

sage: def ScaleZ(a,sz) : return a.substitute(z=(z/sz))

sage: def ScaleXYZ(a,s) : return a.substitute(x=(x/s),y=(y/s),z=(z/s))

sage: #Reflection: Reflects across specific plane

sage: def ReflectXY(a) : return a.substitute(z=(-z))

sage: def ReflectZX(a) : return a.substitute(y=(-y))

sage: def ReflectYZ(a) : return a.substitute(x=(-x))

sage: #Rotation: Turning in specific plane

sage: def DegToRad(a) : return (pi*a/180)

sage: def RotateRX(a,r) : return a.substitute(y=(y*cos(r) - z*sin(r)), z=(y*sin(r) + z*cos(r)))
```

```
sage: def RotateRY(a,r) : return a.substitute(x=(x*cos(r) + z*sin(r)), z=(-x*sin(r) + z*cos(r)))

sage: def RotateRZ(a,r) : return a.substitute(x=(x*cos(r) + y*sin(r)), y=(-x*sin(r) + y*cos(r)))

sage: def RotateX(a,d) : return RotateRX(a, DegToRad(d))

sage: def RotateY(a,d) : return RotateRY(a, DegToRad(d))

sage: def RotateZ(a,d) : return RotateRZ(a, DegToRad(d))

sage: # Simpler Plot Function: Reduces implicit_plot3d to a simpler more practical plotter

sage: def Plot3d (sld,size,pts) : return implicit_plot3d(sld, (x, -size, size), (y, -size, size),
(z, -size, size), plot_points=pts)

sage: # Boolean Operators: For shape interactions

sage: def Union(a,b) : return Min(a,b)

sage: def Intersect(a,b) : return Max(a,b)

sage: def Subtract(a,b) : return Max(a,-b)

sage: def Morph (a,b,t) : return ((b-a)*t+a)

sage: def Blend (a,b,l) : return (a*b)-l

sage: def Combine (a,b,k) : return k*(a+b)

sage: Max(a,b) = (a + b + abs(a - b)) / 2

sage: Min(a,b) = (a + b - abs(a - b)) / 2

sage: # Primitives and Shapes: Defines the specific building blocks which can be pieced together
or transformed to form the desired object. Adapted from w3dsurf and wolfram mathworld

sage: def Sphere (a) : return (x^2+y^2+z^2-a^2)

sage: def Cube(h) : return (Max(Max(abs(x/(h/2)), abs(y/(h/2))), abs(z/(h/2))) - 1)

sage: def Prism(a,b,c) : return (Max(Max(abs(x/(a/2)), abs(y/(b/2))), abs(z/(c/2))) - 1)

sage: def Torus(a,b) : return ((a-sqrt(x^2+y^2)^2+z^2-b^2))

sage: def Cyyl(a,b,c) : return Intersect(((x^2/a)+(y^2/b)-1),Close(z,c))

sage: def Cone(a,b,c) : return Intersect(((c*sqrt((x^2/a^2)+(y^2/b^2)))-z),Close(z,c))

sage: def Ellip(a,b,c) : return ((x^2/a^2)+(y^2/b^2)+(z^2/c^2)-1)

sage: def Para (a,b,c) : return (Max((x^2/a^2+y^2/b^2-c*z),z)-1)

sage: def Close(a,b) : return (abs(a/(b/2))-1)

sage: def HSpace (h) : return z-h

sage: def Tetrahedral (a,b,c) : return ((x^2 + y^2 + z^2)^2 + a*x*y*z - b*(x^2 + y^2 + z^2) + c)

sage: def Sinus (r,h) : return (sin(pi*(x^2+y^2-r^2))/2+h*z)

sage: def Bug (a,b,c) : return ((x*cos(b*y)-z*sin(b*y))+a)^2+(y/c)^2+(x*sin(b*y)+z*cos(b*y))^2 -
1

sage: def Drope (h) : return (z - h*x*exp(-x^2-y^2))

sage: def Heart (a,b,c,d,e) : return (a*(x^2)+b*(y^2)+c*(z^2)-1)^3-d*(x^2)*(z^3)-e*(y^2)*(z^3)

sage: def surface to stl filesave(surface,filename):

sage: #    Return an STL representation of the surface to a file.

sage: #    Original work by Christopher Olah and later modified to write to a

sage: #    file.

sage: #

sage: #    INPUT:

sage: #        - `surface` -- any surface, eg. output of a 3d plot function.

sage: #

sage: #    OUTPUT:
```

```
sage: #        A string that represents the surface in the STL format.
sage: #
sage: #    COMMENTS:
sage: #        (1) You must view the surface before plotting it.
sage: #            Otherwise, this will not work.
sage: #        (2) In order to do 3d printing with this, you will need to
sage: #            convert it into gcode. Skeinforge is an open-source
sage: #            program that can do this.
sage: #        (3) The size of the surface is not normalized in export.
sage: #            Sage's units will become the units of the STL
sage: #            description. These seem to be ~0.05 cm (at least when
sage: #            printed using skeinforge -> replicatorg -> hacklab.to's
sage: #            cupcake).
sage: #        (4) Be aware of the overhang limits of your 3d printer;
sage: #            most printers can only handle an overhang of Pi/2 (45*)
sage: #            before your model will start drooping.
sage: #
sage: #    EXAMPLES:
sage: #        sage: x,y,z = var('x,y,z')
sage: #        sage: a = implicit_plot3d(x^2+y^2+z^2-9, [x,-5,5], [y,-5,5],[z,
sage: #              -5,5])
sage: #        sage: surface_to_stl_filesave(a,'file')
sage: #
...     stl=open(filename + ".stl",'w');
...     stl.write('solid mathsurface\n');
...     for i in surface.face list():
...         n = ( i[1][1]*i[2][2] - i[2][1]*i[1][2],
...               i[1][2]*i[2][0] - i[1][0]*i[2][2],
...               i[1][0]*i[2][1] - i[2][0]*i[1][1] )
...         abs = (n[0]^2+n[1]^2+n[2]^2)^0.5
...         n= (n[0]/abs,n[1]/abs,n[2]/abs)
...         stl.write(' facet normal ' + repr(n[0])  + ' ' + repr(n[1])    + ' ' + repr(n[2])
+ '\n');
...         stl.write('  outer loop\n');
...         stl.write('   vertex ' + repr(i[0][0]) + ' ' + repr(i[0][1]) + ' ' + repr(i[0][2]) +
'\n');
...         stl.write('   vertex ' + repr(i[1][0]) + ' ' + repr(i[1][1]) + ' ' + repr(i[1][2]) +
'\n');
...         stl.write('   vertex ' + repr(i[2][0]) + ' ' + repr(i[2][1]) + ' ' + repr(i[2][2]) +
'\n');
...         stl.write('  endloop\n');
...         stl.write(' endfacet\n');
...     stl.write('endsolid surface\n');
...     stl.close();
```

```
...     return
...
...
sage: # Functions for Curve Followers
sage: # Translation Distances - evaluates the parametric function at t = s to determine the
translation distance.
sage: def trans_point(r,s): return f[r](t=s)
sage: # Rotation Angles - takes the tangent line at time t = v and projects it onto the xy, yz,
and xz planes when u = 0, 1, or 2 respectively. Then it finds the angle between the projection
and the respective x, y and z unit vectors using arctangent of the slope.
...
...
sage: def angle(u,v):
...       if diff(f[u])(t = v) == 0:
...           return pi/2
...       else:
...           return arctan(diff(f[(u+1)%3])(t = v)/diff(f[u])(t = v))
...
...
sage: # Sphere Curve Follower - f is a parametric function for a curve such that f =
[x(t),y(t),z(t)], n is the number of shapes desired on the curve, (a,b) is the interval for t,
and radius is the radius of each sphere.
sage: def curve_fol_sphere(f,n,(a,b),radius):
...       step = (b - a)/n
...       x0 = trans_point(0,a)
...       y0 = trans_point(1,a)
...       z0 = trans point(2,a)
...       combo = TransZ(TransY(TransX(Sphere(radius),x0),y0),z0)
...       for i in range(1,n):
...           tnew = a + i*step
...           x1 = trans_point(0,tnew)
...           y1 = trans_point(1,tnew)
...           z1 = trans_point(2,tnew)
...           next_sphere = TransZ(TransY(TransX(Sphere(radius),x1),y1),z1)
...           combo = Union(combo, next_sphere)
...       return combo
...
sage: # Cylinder Curve Follower - f is a parametric function for a curve such that f =
[x(t),y(t),z(t)], n is the number of shapes desired on the curve, (a,b) is the interval for t,
and """"size is the size of each cube."""""""
sage: def curve_fol_cyyl(f,n,(a,b),(c,d,e)):
...       step = (b - a)/n
...       x0 = trans point(0,a)
...       y0 = trans_point(1,a)
...       z0 = trans_point(2,a)
...       angxy0 = angle(0,a)
```

```
...       angyz0 = angle(1,a)+pi/2
...       angxz0 = angle(2,a)
...       cyll = RotateRZ(RotateRY(RotateRX(Cyyl(c,d,e),angxy0),angyz0),angxz0)
...       combo = TransZ(TransY(TransX(cyll,x0),y0),z0)
...       for i in range(1,n):
...           tnew = a + i*step
...           x1 = trans_point(0,tnew)
...           y1 = trans_point(1,tnew)
...           z1 = trans_point(2,tnew)
...           angxy1 = angle(0,tnew)
...           angyz1 = angle(1,tnew)+pi/2
...           angxz1 = angle(2,tnew)
...           next cyyl = RotateRZ(RotateRY(RotateRX(Cyyl(c,d,e),angxy1),angyz1),angxz1)
...           move next cyyl = TransZ(TransY(TransX(next cyyl,x1),y1),z1)
...           combo = Union(combo, move next cyyl)
...       return combo
...
...
sage: # Box Curve Follower - f is a parametric function for a curve such that f =
[x(t),y(t),z(t)], n is the number of shapes desired on the curve, (a,b) is the interval for t,
and size is the size of each cube.
sage: def curve_fol_box(f,n,(a,b),size):
...       step = (b - a)/n
...       x0 = trans_point(0,a)
...       y0 = trans_point(1,a)
...       z0 = trans_point(2,a)
...       angxy0 = angle(0,a)
...       angyz0 = angle(1,a)
...       angxz0 = angle(2,a)
...       box = RotateRZ(RotateRY(RotateRX(Cube(size),angxy0),angyz0),angxz0)
...       combo = TransZ(TransY(TransX(box,x0),y0),z0)
...       for i in range(1,n):
...           tnew = a + i*step
...           x1 = trans_point(0,tnew)
...           y1 = trans_point(1,tnew)
...           z1 = trans point(2,tnew)
...           angxy1 = angle(0,tnew)
...           angyz1 = angle(1,tnew)
...           angxz1 = angle(2,tnew)
...           next_box = RotateRZ(RotateRY(RotateRX(Cube(size),angxy1),angyz1),angxz1)
...           move_next_box = TransZ(TransY(TransX(next_box,x1),y1),z1)
...           combo = Union(combo,move next box)
...       return combo
```

```
sage: # "Drope Block" a sphere with the drope function on 4 of its 6 "sides" - Adapted from
Nicholas Lewis
sage: sp = Sphere(4)
sage: dr = TransZ(Drope (4),2.5)
sage: top = dr
sage: side1 = RotateY(top,90)
sage: ts1 = Intersect(top ,side1)
sage: bs2 = RotateY(ts1,180)
sage: tube =Intersect(ts1 ,bs2)
sage: sld = Intersect(tube,sp)
sage: dbplt = Plot3d(sld,5,100)
sage: dbplt
sage: # "Sinus Bugs" two interlocking bugs patterned with a sinus - Adapted from Nicholas Lewis
sage: t1 = 5*Bug (-.5,1, 4)
sage: t2 = 5*Bug (.5,1, 4)
sage: s = Sinus (0,5)
sage: sld = Combine(Union(t1,t2),s,1)
sage: plt = Plot3d(sld,6,200)
sage: plt
sage: # "Super Wiffle" a ball locked inside a tetrahedron
sage: c = Tetrahedral (18,18,60)
sage: sp = Sphere(1.5)
sage: sld = Union(c,sp)
sage: swplt = Plot3d(sld,10,100)
sage: swplt
sage: # "Sinus Sphere" a sphere patterned with a sinus - Adapted from Nicholas Lewis
sage: a = Sinus(0,4)
sage: b =  (cos(x) * cos(y)) * 10
sage: c = Sphere(4)
sage: sld = Intersect(b-a+2,c)
sage: Splt = Plot3d(sld,4,100)
sage: Splt
sage: #Heart
sage: hearts=Plot3d(Heart(1,9/4,1,1,9/80),1.5,100)
sage: hearts
sage: # "Spaceship" a spaceship modeled using prisms, a cone, a cylinder and the sinus sphere
sage: q=TransZ(TransX(RotateRY(Prism(1,.25,3),-pi/9),.75),8)
sage: r=Union(q,RotateRZ(q,pi))
sage: r2=Union(r,RotateRZ(r,pi/2))
sage:
spaceship=implicit_plot3d(Union(r2,Union(TransZ(RotateRX(ScaleXYZ(sld,.25),pi),10),Union(TransZ(C
yyl(.35,.35,6),7),Cone(1,1,9)))),(x, -5, 5), (y, -5, 5), (z, -.25, 12), plot points=100)
sage: spaceship
sage: # "Sphere Circle"
```

```
sage: t = var('t')
sage: f = [4*sin(t),  4*cos(t), 0*t]
sage: circ=curve_fol_sphere(f,7,(0,2*pi),2)
sage: Plot3d(circ,10,100)
sage: # "Stacked Sphere Circle"
sage: t = var('t')
sage: f = [4*sin(t),  4*cos(t), 2*t/t]
sage: circ2=curve_fol_sphere(f,7,(0,2*pi),2)
sage: criclplt=Plot3d(Union(circ,circ2),10,100)
sage: criclplt
sage: #stl_outputs
sage: surface_to_stl_filesave(spaceship,'spaceship')
sage: surface to stl filesave(criclplt,'criclplt')
sage: surface to stl filesave(hearts,'hearts')
sage: surface to stl filesave(Splt,'Splt')
sage: surface_to_stl_filesave(swplt,'swplt')
sage: surface_to_stl_filesave(plt,'plt')
sage: surface_to_stl_filesave(dbplt,'dbplt')
sage: %auto
sage: # Interactive Curve Follower: Various parameters controlling the creation of a parametric
curve on which 3D object will be placed are presented. Once the proper parameters have been
selected a 3d model of the figure is displayed.
sage: var('t x y z')
sage: inter_plot = 0
sage: @interact
sage: def curve_fol(shape_type = ("Shape",['Sphere','Cube','Cylinder']), rotshapes =
checkbox(True, "Rotate Shapes to Match Curve?"), n = ("# of Shapes on Curve",1), xfun = ("x(t)",
t), yfun = ("y(t)", t), zfun = ("z(t)", t), a = ("t start", 0), b = ("t end",1), radius =
("radius", 1), height = ("height", 1), showplot = checkbox(True, "Display Shape?"), xaxes = ("X-
axis Size", 10), yaxes = ("Y-axis Size", 10), zaxes = ("Z-axis Size", 10), saveplot  =
checkbox(False, "Save Plot?"), auto_update = False):
...       var('t x y z')
...       g = (xfun,yfun,zfun)
...
...       if shape_type == 'Cylinder':
...           shape_fun = Cyyl
...       if shape type == 'Sphere':
...           shape_fun = Sphere
...       if shape_type == 'Cube':
...           shape_fun = Cube
...       def trans_point2(r,s): return g[r](t=s)
...       def angle(u,v):
...           if diff(g[u])(t = v) == 0:
...               return pi/2
...           else:
...               return arctan(diff(g[(u+1)%3])(t = v)/diff(g[u])(t = v))
```

```
...       if shape_type == 'Cylinder':
...           step = (b - a)/n
...           x0 = trans_point2(0,a)
...           y0 = trans_point2(1,a)
...           z0 = trans_point2(2,a)
...           angxy0 = angle(0,a)
...           angyz0 = angle(1,a)+pi/2
...           angxz0 = angle(2,a)
...           if rotshapes:
...               shape1 =
RotateRZ(RotateRY(RotateRX(Cyyl(radius,radius,height),angxy0),angyz0),angxz0)
...           else:
...                   shape1 = Cyyl(radius,radius,height)
...           combo = TransZ(TransY(TransX(shape1,x0),y0),z0)
...           for i in range(1,n):
...               tnew = a + i*step
...               x1 = trans_point2(0,tnew)
...               y1 = trans_point2(1,tnew)
...               z1 = trans_point2(2,tnew)
...               angxy1 = angle(0,tnew)
...               angyz1 = angle(1,tnew)+pi/2
...               angxz1 = angle(2,tnew)
...               if rotshapes:
...                   shape_next =
RotateRZ(RotateRY(RotateRX(Cyyl(radius,radius,height),angxy0),angyz0),angxz0)
...               else:
...                   shape next = Cyyl(radius,radius,height)
...               next_sphere = TransZ(TransY(TransX(shape_next,x1),y1),z1)
...               combo = Union(combo, next_sphere)
...       elif shape_type == 'Cube':
...           step = (b - a)/n
...           x0 = trans_point2(0,a)
...           y0 = trans_point2(1,a)
...           z0 = trans_point2(2,a)
...           angxy0 = angle(0,a)
...           angyz0 = angle(1,a)
...           angxz0 = angle(2,a)
...           if rotshapes:
...               shape1 = RotateRZ(RotateRY(RotateRX(Cube(radius),angxy0),angyz0),angxz0)
...           else:
...                   shape1 = Cube(radius)
...           combo = TransZ(TransY(TransX(shape1,x0),y0),z0)
...           for i in range(1,n):
...               tnew = a + i*step
```

```
...                 x1 = trans_point2(0,tnew)
...                 y1 = trans_point2(1,tnew)
...                 z1 = trans_point2(2,tnew)
...                 angxy1 = angle(0,tnew)
...                 angyz1 = angle(1,tnew)
...                 angxz1 = angle(2,tnew)
...                 if rotshapes:
...                     shape_next =
RotateRZ(RotateRY(RotateRX(Cube(radius),angxy0),angyz0),angxz0)
...                 else:
...                     shape_next = Cube(radius)
...                 next_sphere = TransZ(TransY(TransX(shape_next,x1),y1),z1)
...                 combo = Union(combo, next sphere)
...         else:
...             step = (b - a)/n
...             x0 = trans_point2(0,a)
...             y0 = trans_point2(1,a)
...             z0 = trans_point2(2,a)
...             combo = TransZ(TransY(TransX(Sphere(radius),x0),y0),z0)
...             for i in range(1,n):
...                 tnew = a + i*step
...                 x1 = trans_point2(0,tnew)
...                 y1 = trans_point2(1,tnew)
...                 z1 = trans_point2(2,tnew)
...
...                 next_sphere = TransZ(TransY(TransX(Sphere(radius),x1),y1),z1)
...                 combo = Union(combo, next sphere)
...         inter_plot = implicit_plot3d(combo,(x,-xaxes,xaxes),(y,-yaxes,yaxes),(z,-zaxes,zaxes))
...         if showplot:
...             show(inter_plot)
...         if saveplot:
...             surface_to_stl_filesave(inter_plot,'inter_plot')
...         return inter_plot
<html><!--notruncate--><div padding=6 id="div-interact-111"> <table width=800px height=20px
bgcolor="#c5c5c5"
                cellpadding=15><tr><td bgcolor="#f9f9f9" valign=top align=left><table><tr><td
align=right><font color="black">Shape </font></td><td><table style="border:1px solid
#dfdfdf; background-color:#efefef;">
<tr><td><button style="border-style:inset;" value="0" onclick="$('BUTTON',
this.parentNode).css('border-style', 'outset'); $(this).css('border-style', 'inset');
interact(111, '_interact_.update(\'111\', \'shape_type\', 1,
 interact .standard b64decode(\''+encode64(this.value)+'\'), globals())')">Sphere</button>
<button style="border-style:outset;" value="1" onclick="$('BUTTON', this.parentNode).css('border-
style', 'outset'); $(this).css('border-style', 'inset'); interact(111,
'_interact_.update(\'111\', \'shape_type\', 1,
 interact .standard b64decode(\''+encode64(this.value)+'\'), globals())')">Cube</button>
```

```
<button style="border-style:outset;" value="2" onclick="$('BUTTON', this.parentNode).css('border-
style', 'outset'); $(this).css('border-style', 'inset'); interact(111,
'_interact_.update(\'111\', \'shape_type\', 1,
_interact_.standard_b64decode(\''+encode64(this.value)+'\'), globals())')">Cylinder</button>

</td></tr></table></td></tr>

<tr><td align=right><font color="black">Rotate Shapes to Match Curve? </font></td><td><input
type="checkbox" checked width=200px onchange="interact(111, '_interact_.update(\'111\',
\'rotshapes\', 2, _interact_.standard_b64decode(\''+encode64(this.checked)+'\'),
globals())')"></input></td></tr>

<tr><td align=right><font color="black"># of Shapes on Curve </font></td><td><input
type="text" value="1" size=80 onchange="interact(111, '_interact_.update(\'111\', \'n\', 3,
_interact_.standard_b64decode(\''+encode64(this.value)+'\'), globals())')"></input></td></tr>

<tr><td align=right><font color="black">x(t) </font></td><td><input type="text" value="t"
size=80 onchange="interact(111, ' interact .update(\'111\', \'xfun\', 4,
 interact .standard b64decode(\''+encode64(this.value)+'\'), globals())')"></input></td></tr>

<tr><td align=right><font color="black">y(t) </font></td><td><input type="text" value="t"
size=80 onchange="interact(111, ' interact .update(\'111\', \'yfun\', 5,
_interact_.standard_b64decode(\''+encode64(this.value)+'\'), globals())')"></input></td></tr>

<tr><td align=right><font color="black">z(t) </font></td><td><input type="text" value="t"
size=80 onchange="interact(111, '_interact_.update(\'111\', \'zfun\', 6,
 interact .standard b64decode(\''+encode64(this.value)+'\'), globals())')"></input></td></tr>

<tr><td align=right><font color="black">t start </font></td><td><input type="text" value="0"
size=80 onchange="interact(111, '_interact_.update(\'111\', \'a\', 7,
_interact_.standard_b64decode(\''+encode64(this.value)+'\'), globals())')"></input></td></tr>

<tr><td align=right><font color="black">t end </font></td><td><input type="text" value="1"
size=80 onchange="interact(111, '_interact_.update(\'111\', \'b\', 8,
 interact .standard b64decode(\''+encode64(this.value)+'\'), globals())')"></input></td></tr>

<tr><td align=right><font color="black">radius </font></td><td><input type="text" value="1"
size=80 onchange="interact(111, '_interact_.update(\'111\', \'radius\', 9,
_interact_.standard_b64decode(\''+encode64(this.value)+'\'), globals())')"></input></td></tr>

<tr><td align=right><font color="black">height </font></td><td><input type="text" value="1"
size=80 onchange="interact(111, '_interact_.update(\'111\', \'height\', 10,
_interact_.standard_b64decode(\''+encode64(this.value)+'\'), globals())')"></input></td></tr>

<tr><td align=right><font color="black">Display Shape? </font></td><td><input
type="checkbox" checked width=200px onchange="interact(111, '_interact_.update(\'111\',
\'showplot\', 11, _interact_.standard_b64decode(\''+encode64(this.checked)+'\'),
globals())')"></input></td></tr>

<tr><td align=right><font color="black">X-axis Size </font></td><td><input type="text"
value="10" size=80 onchange="interact(111, '_interact_.update(\'111\', \'xaxes\', 12,
_interact_.standard_b64decode(\''+encode64(this.value)+'\'), globals())')"></input></td></tr>

<tr><td align=right><font color="black">Y-axis Size </font></td><td><input type="text"
value="10" size=80 onchange="interact(111, ' interact .update(\'111\', \'yaxes\', 13,
 interact .standard b64decode(\''+encode64(this.value)+'\'), globals())')"></input></td></tr>

<tr><td align=right><font color="black">Z-axis Size </font></td><td><input type="text"
value="10" size=80 onchange="interact(111, '_interact_.update(\'111\', \'zaxes\', 14,
 interact .standard b64decode(\''+encode64(this.value)+'\'), globals())')"></input></td></tr>

<tr><td align=right><font color="black">Save Plot? </font></td><td><input type="checkbox"
width=200px onchange="interact(111, '_interact_.update(\'111\', \'saveplot\', 15,
_interact_.standard_b64decode(\''+encode64(this.checked)+'\'), globals())')"></input></td></tr>

<tr><td colspan=2><input type="button" value="Update" onclick="interact(111,
'_interact_.recompute(\'111\')')">

</td></tr>

</table><div id="cell-interact-111"><?__SAGE__START>

        <table border=0 bgcolor="white" width=100% height=100%>

        <tr><td bgcolor="white" align=left valign=top><pre>

Graphics3d Object
```

```
</pre></td></tr>

        <tr><td  align=left valign=top><div><script>jmol_applet(500,
"/home/marcuslew/5/cells/111/sage0-size500.jmol?1275713609");</script></div></td></tr>

        </table><?__SAGE__END></div></td>

                 </tr></table></div>

                 </html>

sage: var('t')

sage: f = [t,t*t,t]

sage: v = curve_fol_cyyl(f,3,(0,1),(1,1,1))

sage: Plot3d(v,6,100)
```