3D Graphics

Release 10.1

The Sage Development Team

Jul 05, 2024

CONTENTS

CHAPTER

ONE

INTRODUCTION

Sage has a wide support for 3D graphics, from basic shapes to implicit and parametric plots.

The following graphics functions are supported:

- `plot3d()` - plot a 3d function
- `parametric_plot3d()` - a parametric three-dimensional space curve or surface
- `revolution_plot3d()` - a plot of a revolved curve
- `plot_vector_field3d()` - a plot of a 3d vector field
- `implicit_plot3d()` - a plot of an isosurface of a function
- `list_plot3d()`- a 3-dimensional plot of a surface defined by a list of points in 3-dimensional space
- `list_plot3d_matrix()` - a 3-dimensional plot of a surface defined by a matrix defining points in 3-dimensional space
- `list_plot3d_array_of_arrays()`- A 3-dimensional plot of a surface defined by a list of lists defining points in 3-dimensional space
- `list_plot3d_tuples()` - a 3-dimensional plot of a surface defined by a list of points in 3-dimensional space

The following classes for basic shapes are supported:

- `Box` - a box given its three magnitudes
- `Cone` - a cone, with base in the xy-plane pointing up the z-axis
- `Cylinder` - a cylinder, with base in the xy-plane pointing up the z-axis
- `Line` - a 3d line joining a sequence of points
- `Sphere` - a sphere centered at the origin
- `Text` - a text label attached to a point in 3d space
- `Torus` - a 3d torus
- `Point` - a position in 3d, represented by a sphere of fixed size

The following plotting functions for basic shapes are supported

- `ColorCube()` - a cube with given size and sides with given colors
- `LineSegment()` - a line segment, which is drawn as a cylinder from start to end with given radius
- `line3d()` - a 3d line joining a sequence of points
- `arrow3d()` - a 3d arrow
- `point3d()` - a point or list of points in 3d space

- *bezier3d()* - a 3d bezier path
- *frame3d()* - a frame in 3d
- *frame_labels()* - labels for a given frame in 3d
- *polygon3d()* - draw a polygon in 3d
- *polygons3d()* - draw the union of several polygons in 3d
- *ruler()* - draw a ruler in 3d, with major and minor ticks
- *ruler_frame()* - draw a frame made of 3d rulers, with major and minor ticks
- *sphere()* - plot of a sphere given center and radius
- *text3d()* - 3d text

Sage also supports platonic solids with the following functions:

- *tetrahedron()*
- *cube()*
- *octahedron()*
- *dodecahedron()*
- *icosahedron()*

Different viewers are supported: a web-based interactive viewer using the Three.js JavaScript library (the default), Jmol, and the Tachyon ray tracer. The viewer is invoked by adding the keyword argument `viewer='threejs'` (respectively `'jmol'` or `'tachyon'`) to the command `show()` on any three-dimensional graphic.

- *Tachyon* - create a scene the can be rendered using the Tachyon ray tracer
- *Axis_aligned_box* - box with axis-aligned edges with the given min and max coordinates
- *Cylinder* - an infinite cylinder
- *FCylinder* - a finite cylinder
- *FractalLandscape*- axis-aligned fractal landscape
- *Light* - represents lighting objects
- *ParametricPlot* - parametric plot routines
- *Plane* - an infinite plane
- *Ring* - an annulus of zero thickness
- *Sphere*- a sphere
- *TachyonSmoothTriangle* - a triangle along with a normal vector, which is used for smoothing
- *TachyonTriangle* - basic triangle class
- *TachyonTriangleFactory* - class to produce triangles of various rendering types
- *Texfunc* - creates a texture function
- *Texture* - stores texture information
- *tostr()* - converts vector information to a space-separated string

CHAPTER

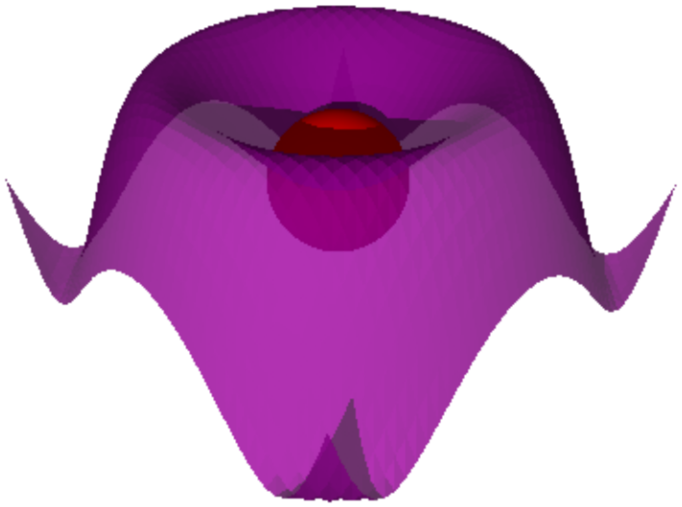
TWO

FUNCTION AND DATA PLOTS

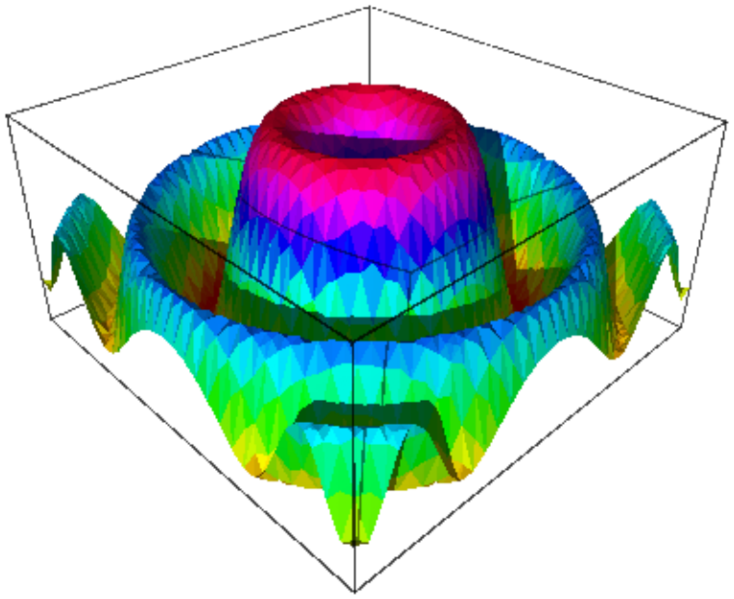
2.1 Plotting functions

EXAMPLES:

```
sage: x, y = var('x y')
sage: W = plot3d(sin(pi*((x)^2 + (y)^2))/2, (x, -1, 1), (y, -1, 1),
....:            frame=False, color='purple', opacity=0.8)
sage: S = sphere((0, 0, 0), size=0.3, color='red', aspect_ratio=[1,1,1])
sage: show(W + S, figsize=8)
```



```
sage: def f(x,y):
....:     return math.sin(y^2 + x^2)/math.sqrt(x^2 + y^2 + 0.0001)
sage: P = plot3d(f, (-3, 3),(-3, 3), adaptive=True,
....:            color=rainbow(60, 'rgbtuple'), max_bend=.1, max_depth=15)
sage: P.show()
```

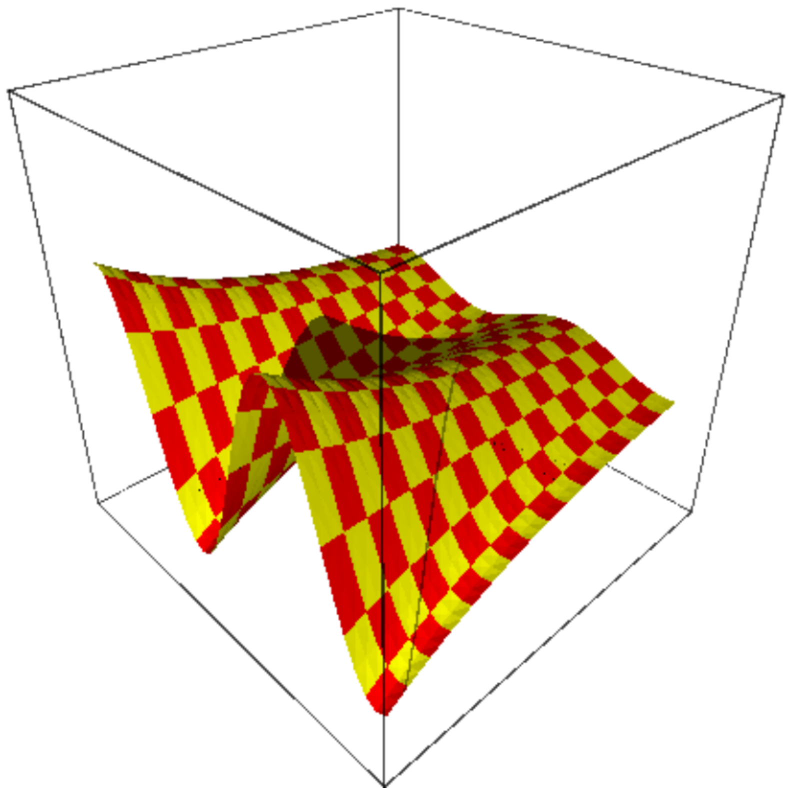


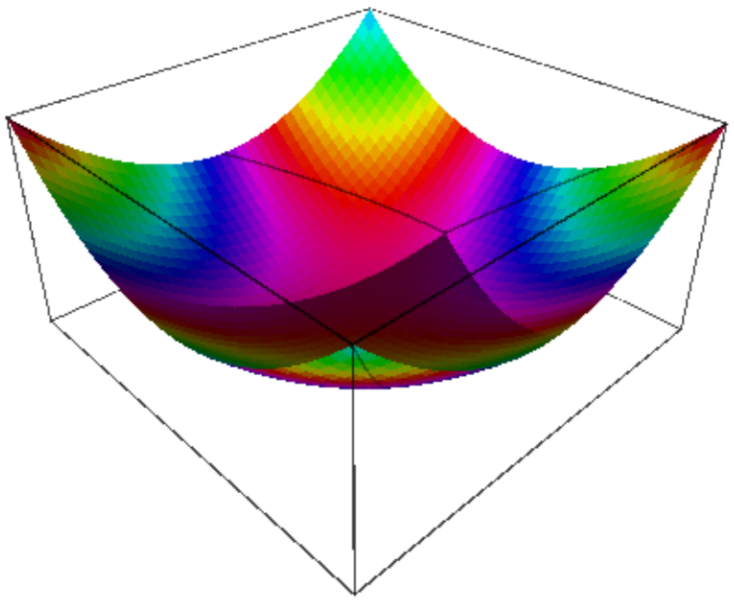
```
sage: def f(x,y):
....:     return math.exp(x/5)*math.sin(y)
....:
sage: P = plot3d(f, (-5, 5), (-5, 5), adaptive=True, color=['red', 'yellow'])
sage: from sage.plot.plot3d.plot3d import axes
sage: S = P + axes(6, color='black')
sage: S.show()
```

Here is an example using a colormap and a color function c:

```
sage: x, y = var('x y')
sage: cm = colormaps.hsv
sage: def c(x, y): return float((x + y + x*y)/15) % 1
sage: plot3d(x*x + y*y, (x, -4, 4), (y, -4, 4), color=(c, cm))
Graphics3d Object
```

Beware that the color function must take values between 0 and 1.



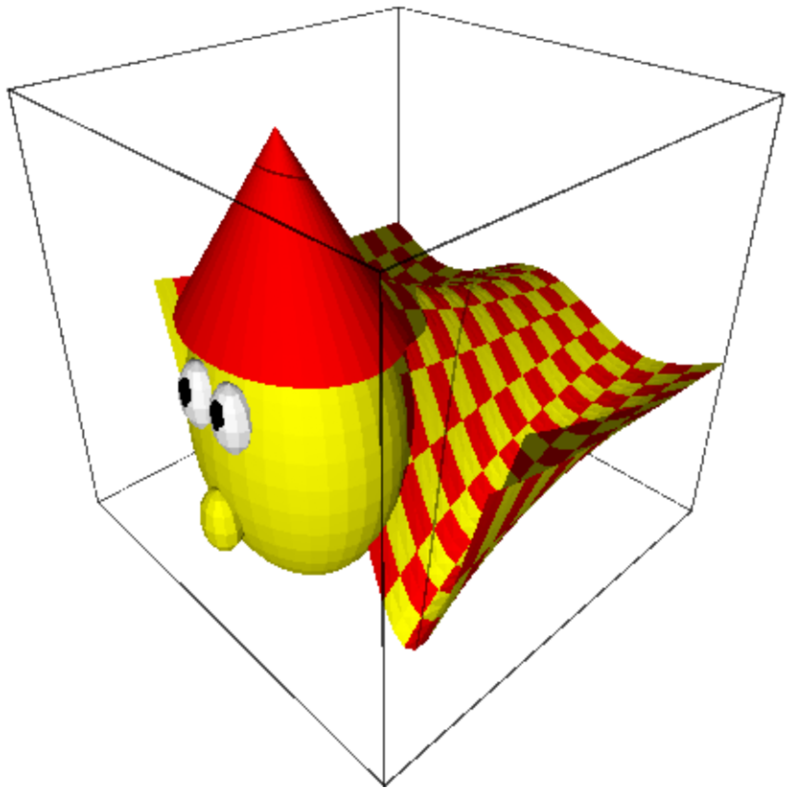


We plot “cape man”:

```
sage: S = sphere(size=.5, color='yellow')
```

```
sage: from sage.plot.plot3d.shapes import Cone
sage: S += Cone(.5, .5, color='red').translate(0,0,.3)
```

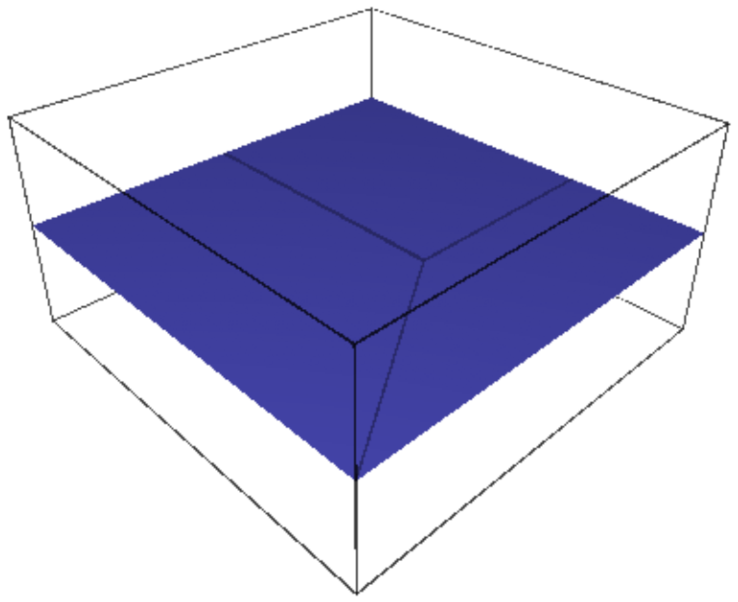
```
sage: S += sphere((.45, -.1, .15), size=.1, color='white')
sage: S += sphere((.51,-.1,.17), size=.05, color='black')
sage: S += sphere((.45, .1, .15), size=.1, color='white')
sage: S += sphere((.51, .1,.17), size=.05, color='black')
sage: S += sphere((.5, 0, -.2), size=.1, color='yellow')
sage: def f(x,y): return math.exp(x/5)*math.cos(y)
sage: P = plot3d(f, (-5, 5), (-5, 5), adaptive=True,
....:            color=['red','yellow'], max_depth=10)
sage: cape_man = P.scale(.2) + S.translate(1, 0, 0)
sage: cape_man.show(aspect_ratio=[1, 1, 1])
```



Or, we plot a very simple function indeed:

```
sage: plot3d(pi, (-1,1), (-1,1))
Graphics3d Object
```

Transparent with fractional opacity value:



```
sage: plot3d(lambda x, y: x^2 + y^2, (-2,2), (-2,2), opacity=8/10)
Graphics3d Object
```

Todo: Add support for smooth triangles.

AUTHORS:

- Tom Boothby: adaptive refinement triangles
- Josh Kantor: adaptive refinement triangles
- Robert Bradshaw (2007-08): initial version of this file
- William Stein (2007-12, 2008-01): improving 3d plotting
- Oscar Lazo, William Cauchois, Jason Grout (2009-2010): Adding coordinate transformations

class `sage.plot.plot3d.plot3d.`**`Cylindrical`**(*dep_var*, *indep_vars*)

Bases: `_Coordinates`

A cylindrical coordinate system for use with `plot3d(transformation=...)` where the position of a point is specified by three numbers:

- the *radial distance* (`radius`) from the z-axis
- the *azimuth angle* (`azimuth`) from the positive x-axis
- the *height* or *altitude* (`height`) above the xy-plane

These three variables must be specified in the constructor.

EXAMPLES:

Construct a cylindrical transformation for a function for `height` in terms of `radius` and `azimuth`:

```
sage: T = Cylindrical('height', ['radius', 'azimuth'])
```

If we construct some concrete variables, we can get a transformation:

```
sage: r, theta, z = var('r theta z')
sage: T.transform(radius=r, azimuth=theta, height=z)
(r*cos(theta), r*sin(theta), z)
```

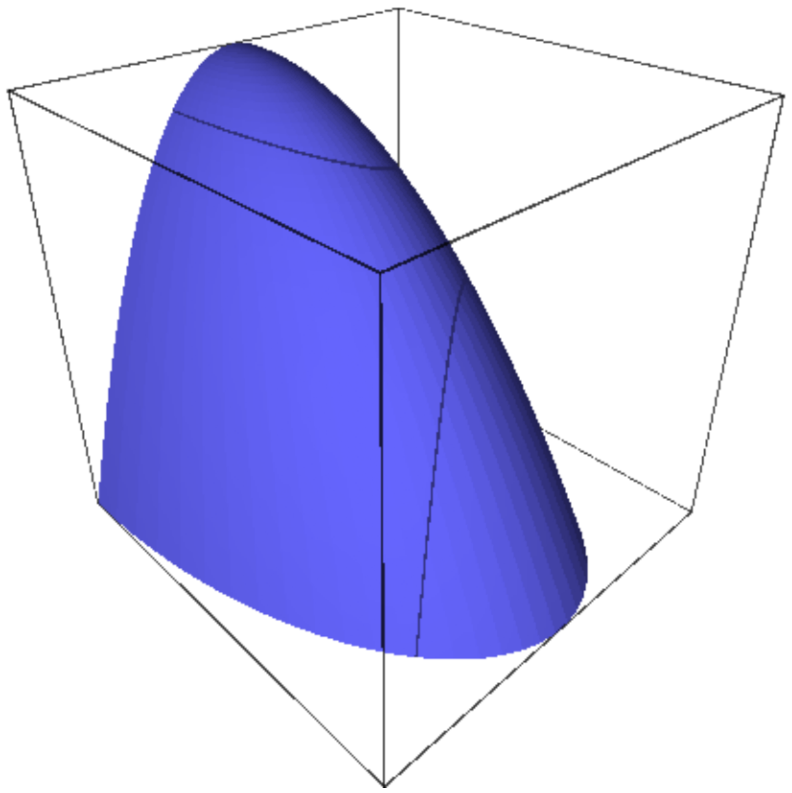
We can plot with this transform. Remember that the dependent variable is the height, and the independent variables are the radius and the azimuth (in that order):

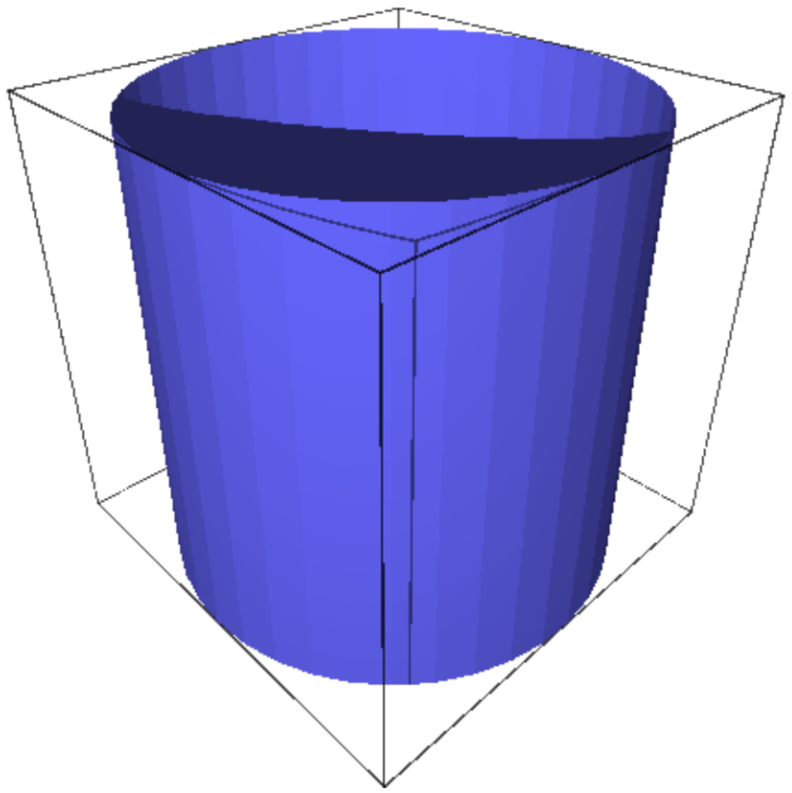
```
sage: plot3d(9-r^2, (r, 0, 3), (theta, 0, pi), transformation=T)
Graphics3d Object
```

We next graph the function where the radius is constant:

```
sage: S = Cylindrical('radius', ['azimuth', 'height'])
sage: theta, z = var('theta, z')
sage: plot3d(3, (theta, 0, 2*pi), (z, -2, 2), transformation=S)
Graphics3d Object
```

See also *`cylindrical_plot3d()`* for more examples of plotting in cylindrical coordinates.





transform(*radius=None*, *azimuth=None*, *height=None*)

A cylindrical coordinates transform.

EXAMPLES:

```
sage: T = Cylindrical('height', ['azimuth', 'radius'])
sage: T.transform(radius=var('r'), azimuth=var('theta'), height=var('z'))
(r*cos(theta), r*sin(theta), z)
```

class `sage.plot.plot3d.plot3d.`**Spherical**(*dep_var*, *indep_vars*)

Bases: `_Coordinates`

A spherical coordinate system for use with `plot3d(transformation=...)` where the position of a point is specified by three numbers:

- the *radial distance* (`radius`) from the origin
- the *azimuth angle* (`azimuth`) from the positive x-axis
- the *inclination angle* (`inclination`) from the positive z-axis

These three variables must be specified in the constructor.

EXAMPLES:

Construct a spherical transformation for a function for the radius in terms of the azimuth and inclination:

```
sage: T = Spherical('radius', ['azimuth', 'inclination'])
```

If we construct some concrete variables, we can get a transformation in terms of those variables:

```
sage: r, phi, theta = var('r phi theta')
sage: T.transform(radius=r, azimuth=theta, inclination=phi)
(r*cos(theta)*sin(phi), r*sin(phi)*sin(theta), r*cos(phi))
```

We can plot with this transform. Remember that the dependent variable is the radius, and the independent variables are the azimuth and the inclination (in that order):

```
sage: plot3d(phi * theta, (theta, 0, pi), (phi, 0, 1), transformation=T)
Graphics3d Object
```

We next graph the function where the inclination angle is constant:

```
sage: S = Spherical('inclination', ['radius', 'azimuth'])
sage: r, theta = var('r,theta')
sage: plot3d(3, (r,0,3), (theta, 0, 2*pi), transformation=S)
Graphics3d Object
```

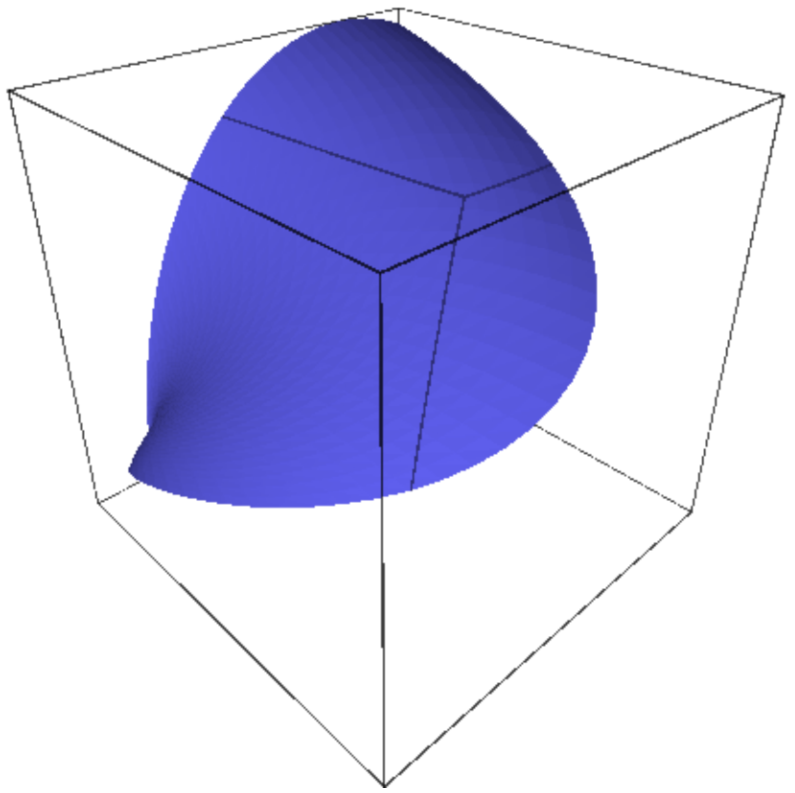
See also *`spherical_plot3d()`* for more examples of plotting in spherical coordinates.

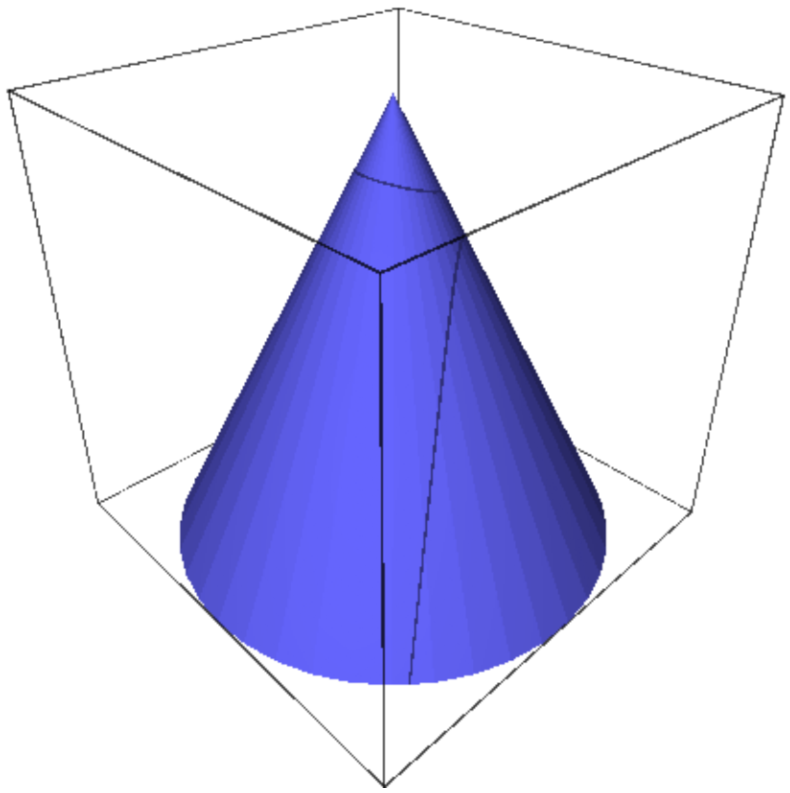
transform(*radius=None*, *azimuth=None*, *inclination=None*)

A spherical coordinates transform.

EXAMPLES:

```
sage: T = Spherical('radius', ['azimuth', 'inclination'])
sage: T.transform(radius=var('r'), azimuth=var('theta'), inclination=var('phi'))
(r*cos(theta)*sin(phi), r*sin(phi)*sin(theta), r*cos(phi))
```





class `sage.plot.plot3d.plot3d.`**`SphericalElevation`**(*dep_var*, *indep_vars*)

Bases: `_Coordinates`

A spherical coordinate system for use with `plot3d(transformation=...)` where the position of a point is specified by three numbers:

- the *radial distance* (`radius`) from the origin
- the *azimuth angle* (`azimuth`) from the positive x-axis
- the *elevation angle* (`elevation`) from the xy-plane toward the positive z-axis

These three variables must be specified in the constructor.

EXAMPLES:

Construct a spherical transformation for the radius in terms of the azimuth and elevation. Then, get a transformation in terms of those variables:

```
sage: T = SphericalElevation('radius', ['azimuth', 'elevation'])
sage: r, theta, phi = var('r theta phi')
sage: T.transform(radius=r, azimuth=theta, elevation=phi)
(r*cos(phi)*cos(theta), r*cos(phi)*sin(theta), r*sin(phi))
```

We can plot with this transform. Remember that the dependent variable is the radius, and the independent variables are the azimuth and the elevation (in that order):

```
sage: plot3d(phi * theta, (theta, 0, pi), (phi, 0, 1), transformation=T)
Graphics3d Object
```

We next graph the function where the elevation angle is constant. This should be compared to the similar example for the `Spherical` coordinate system:

```
sage: SE = SphericalElevation('elevation', ['radius', 'azimuth'])
sage: r, theta = var('r,theta')
sage: plot3d(3, (r,0,3), (theta, 0, 2*pi), transformation=SE)
Graphics3d Object
```

`transform`(*radius=None*, *azimuth=None*, *elevation=None*)

A spherical elevation coordinates transform.

EXAMPLES:

```
sage: T = SphericalElevation('radius', ['azimuth', 'elevation'])
sage: T.transform(radius=var('r'), azimuth=var('theta'), elevation=var('phi'))
(r*cos(phi)*cos(theta), r*cos(phi)*sin(theta), r*sin(phi))
```

class `sage.plot.plot3d.plot3d.`**`TrivialTriangleFactory`**

Bases: `object`

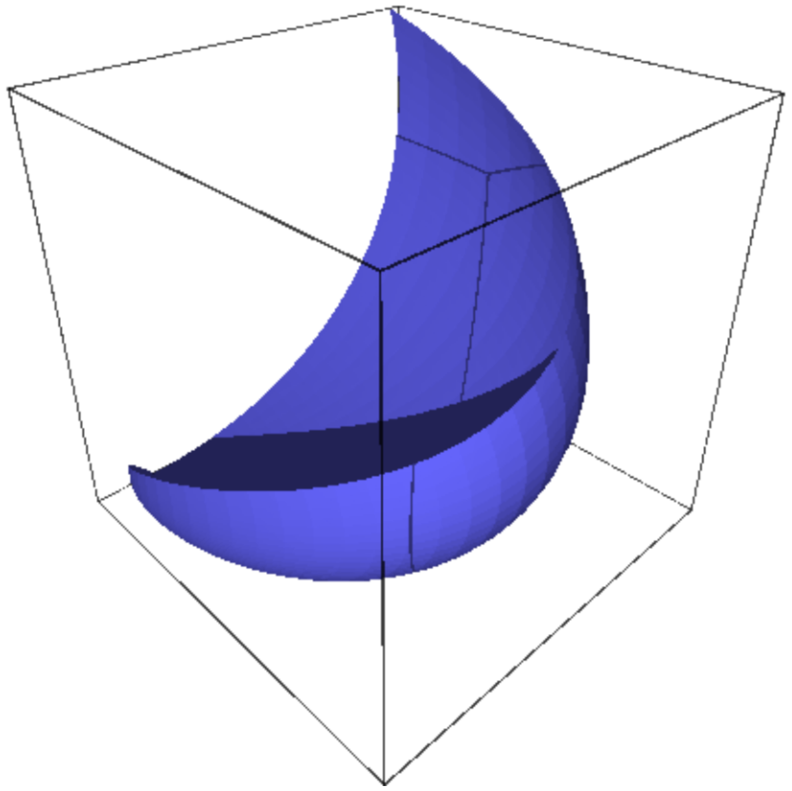
Class emulating behavior of *`TriangleFactory`* but simply returning a list of vertices for both regular and smooth triangles.

`smooth_triangle`(*a*, *b*, *c*, *da*, *db*, *dc*, *color=None*)

Function emulating behavior of *`smooth_triangle()`* but simply returning a list of vertices.

INPUT:

- a, b, c – triples (x,y,z) representing corners on a triangle in 3-space



- da, db, dc – ignored
- color – ignored

OUTPUT:

- the list [a,b,c]

triangle(*a*, *b*, *c*, *color=None*)

Function emulating behavior of *triangle()* but simply returning a list of vertices.

INPUT:

- a, b, c – triples (x,y,z) representing corners on a triangle in 3-space
- color – ignored

OUTPUT:

- the list [a,b,c]

sage.plot.plot3d.plot3d.**axes**(*scale=1*, *radius=None*, ***kwds*)

Creates basic axes in three dimensions. Each axis is a three dimensional arrow object.

INPUT:

- scale – (default: 1) The length of the axes (all three will be the same).
- radius – (default: .01) The radius of the axes as arrows.

EXAMPLES:

```
sage: from sage.plot.plot3d.plot3d import axes
sage: S = axes(6, color='black'); S
Graphics3d Object
```

```
sage: T = axes(2, .5); T
Graphics3d Object
```

sage.plot.plot3d.plot3d.**cylindrical_plot3d**(*f*, *urange*, *vrange*, ***kwds*)

Plots a function in cylindrical coordinates. This function is equivalent to:

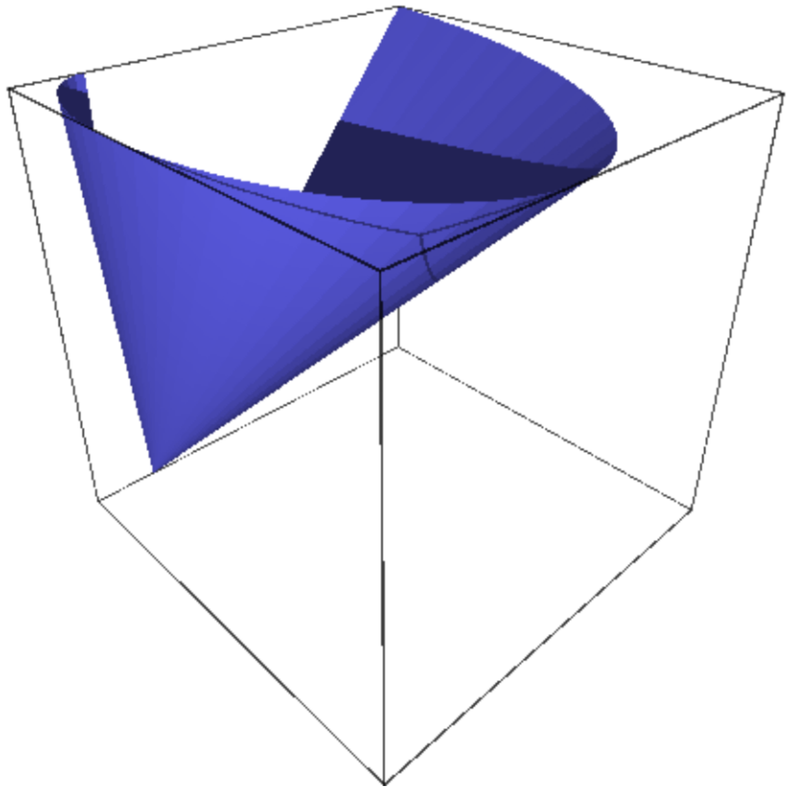
```
sage: r, u, v = var('r,u,v')
sage: f = u*v; urange = (u, 0, pi); vrange = (v, 0, pi)
sage: T = (r*cos(u), r*sin(u), v, [u, v])
sage: plot3d(f, urange, vrange, transformation=T)
Graphics3d Object
```

or equivalently:

```
sage: T = Cylindrical('radius', ['azimuth', 'height'])
sage: f=lambda u,v: u*v; urange=(u,0,pi); vrange=(v,0,pi)
sage: plot3d(f, urange, vrange, transformation=T)
Graphics3d Object
```

INPUT:

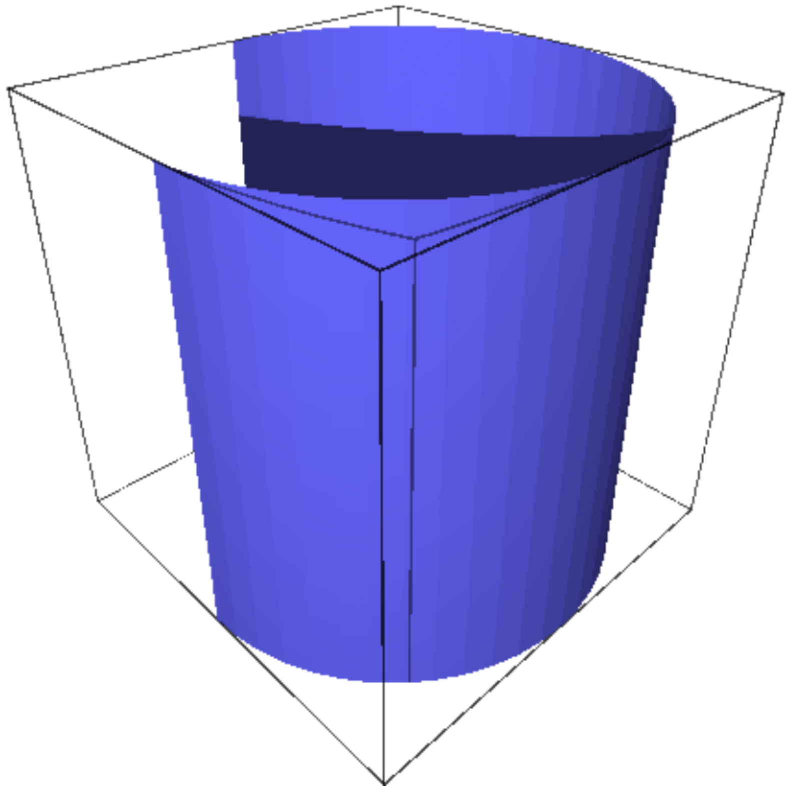
- f – a symbolic expression or function of two variables, representing the radius from the z-axis
- urange – a 3-tuple (u, u_min, u_max), the domain of the azimuth variable
- vrange – a 3-tuple (v, v_min, v_max), the domain of the elevation (z) variable



EXAMPLES:

A portion of a cylinder of radius 2:

```
sage: theta, z = var('theta,z')
sage: cylindrical_plot3d(2, (theta, 0, 3*pi/2), (z, -2, 2))
Graphics3d Object
```



Some random figures:

```
sage: cylindrical_plot3d(cosh(z), (theta, 0, 2*pi), (z, -2, 2))
Graphics3d Object
```

```
sage: cylindrical_plot3d(e^(-z^2)*(cos(4*theta) + 2) + 1, (theta, 0, 2*pi), (z, -2,␣
↪2), plot_points=[80, 80]).show(aspect_ratio=(1, 1, 1))
```

`sage.plot.plot3d.plot3d.`**`plot3d`**(*f*, *urange*, *vrange*, *adaptive=False*, *transformation=None*, ***kwds*)

Plots a function in 3d.

INPUT:

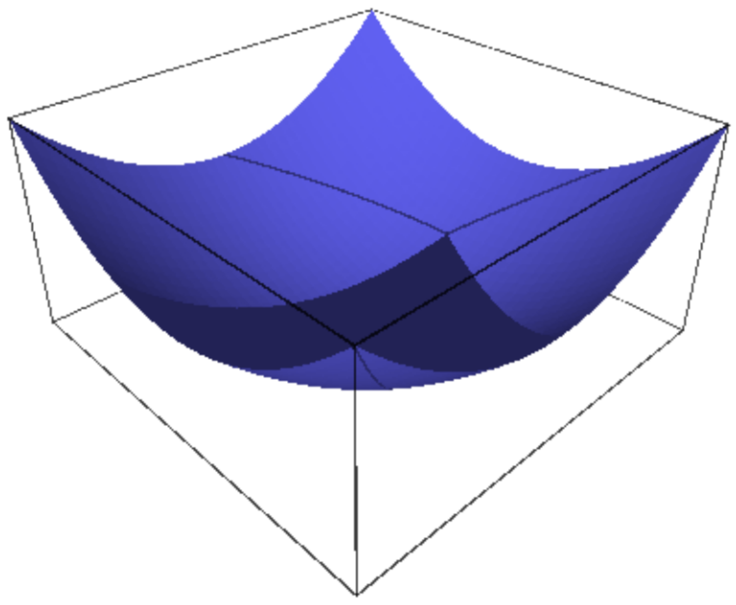
- `f` – a symbolic expression or function of 2 variables
- `urange` – a 2-tuple (u_min, u_max) or a 3-tuple (u, u_min, u_max)
- `vrange` – a 2-tuple (v_min, v_max) or a 3-tuple (v, v_min, v_max)

- `adaptive` – (default: False) whether to use adaptive refinement to draw the plot (slower, but may look better). This option does NOT work in conjunction with a transformation (see below).
- `mesh` – bool (default: False) whether to display mesh grid lines
- `dots` – bool (default: False) whether to display dots at mesh grid points
- `plot_points` – (default: "automatic") initial number of sample points in each direction; an integer or a pair of integers
- `transformation` – (default: None) a transformation to apply. May be a 3 or 4-tuple (x_func, y_func, z_func, independent_vars) where the first 3 items indicate a transformation to Cartesian coordinates (from your coordinate system) in terms of u, v, and the function variable fvar (for which the value of f will be substituted). If a 3-tuple is specified, the independent variables are chosen from the range variables. If a 4-tuple is specified, the 4th element is a list of independent variables. `transformation` may also be a predefined coordinate system transformation like Spherical or Cylindrical.

Note: `mesh` and `dots` are not supported when using the Tachyon raytracer renderer.

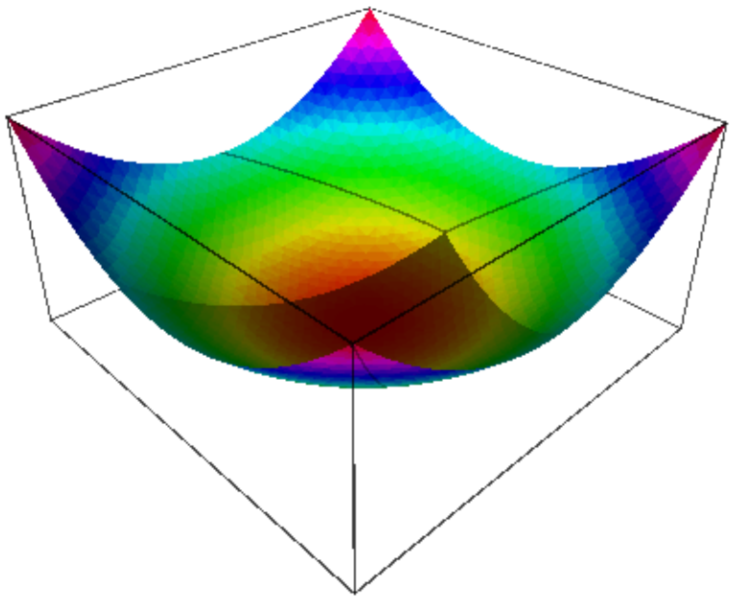
EXAMPLES: We plot a 3d function defined as a Python function:

```
sage: plot3d(lambda x, y: x^2 + y^2, (-2,2), (-2,2))
Graphics3d Object
```



We plot the same 3d function but using adaptive refinement:

```
sage: plot3d(lambda x, y: x^2 + y^2, (-2,2), (-2,2), adaptive=True)
Graphics3d Object
```



Adaptive refinement but with more points:

```
sage: plot3d(lambda x, y: x^2 + y^2, (-2,2), (-2,2), adaptive=True, initial_depth=5)
Graphics3d Object
```

We plot some 3d symbolic functions:

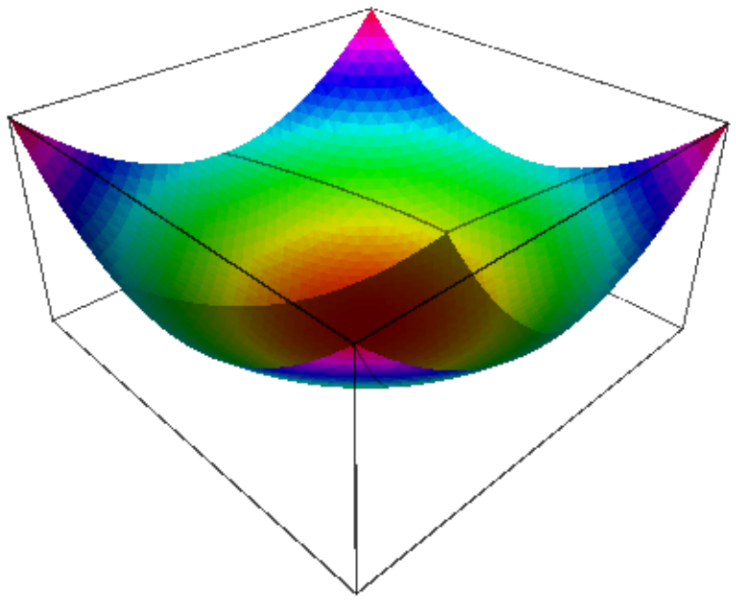
```
sage: var('x,y')
(x, y)
sage: plot3d(x^2 + y^2, (x,-2,2), (y,-2,2))
Graphics3d Object
```

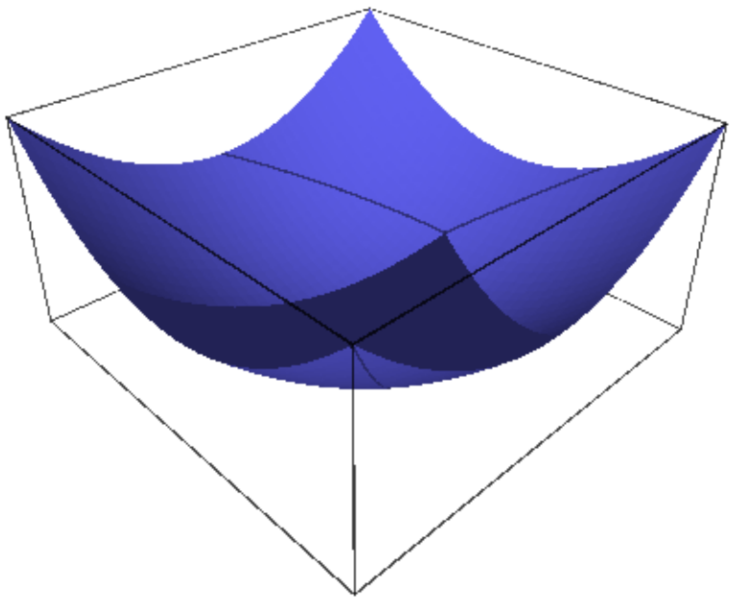
```
sage: plot3d(sin(x*y), (x, -pi, pi), (y, -pi, pi))
Graphics3d Object
```

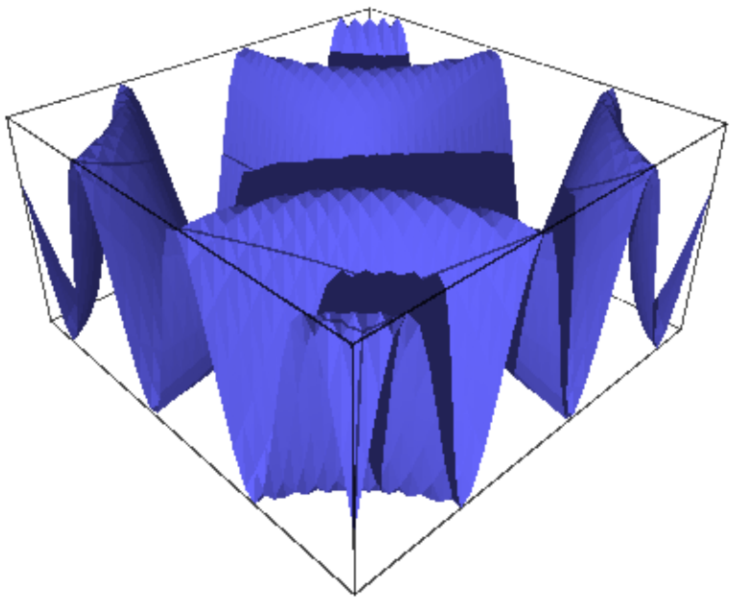
We give a plot with extra sample points:

```
sage: var('x,y')
(x, y)
```

(continues on next page)

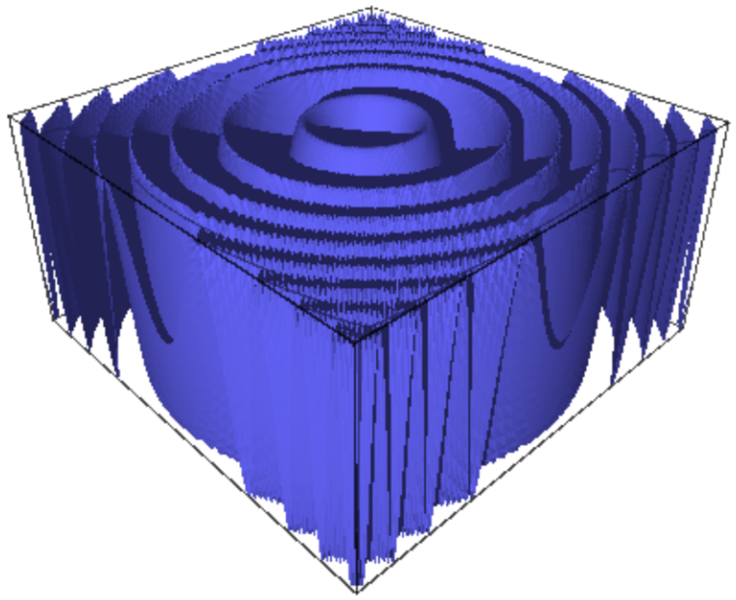






(continued from previous page)

```
sage: plot3d(sin(x^2 + y^2), (x,-5,5), (y,-5,5), plot_points=200)
Graphics3d Object
```



```
sage: plot3d(sin(x^2 + y^2), (x, -5, 5), (y, -5, 5), plot_points=[10, 100])
Graphics3d Object
```

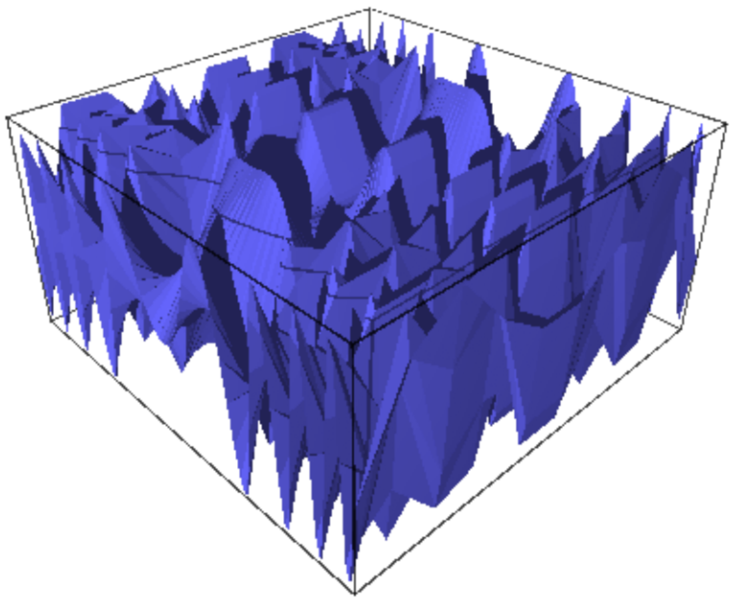
A 3d plot with a mesh:

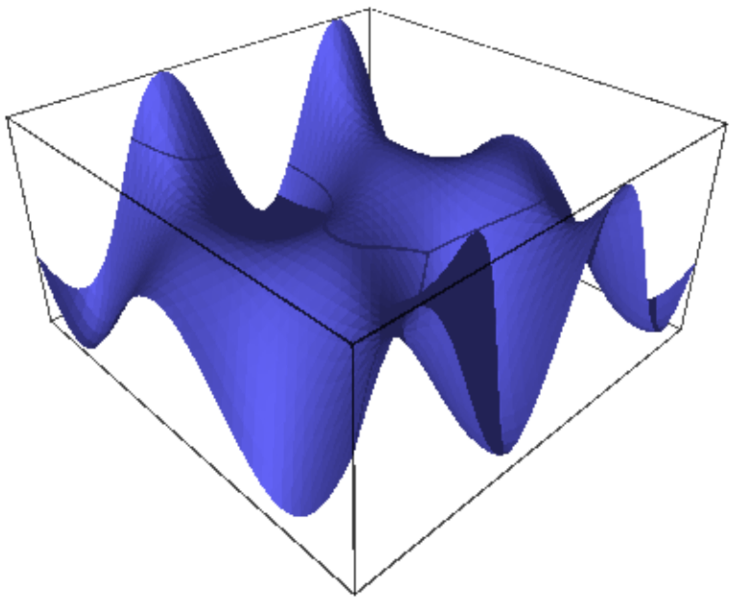
```
sage: var('x,y')
(x, y)
sage: plot3d(sin(x - y)*y*cos(x), (x, -3, 3), (y, -3, 3), mesh=True)
Graphics3d Object
```

The same with thicker mesh lines (not supported in all viewers):

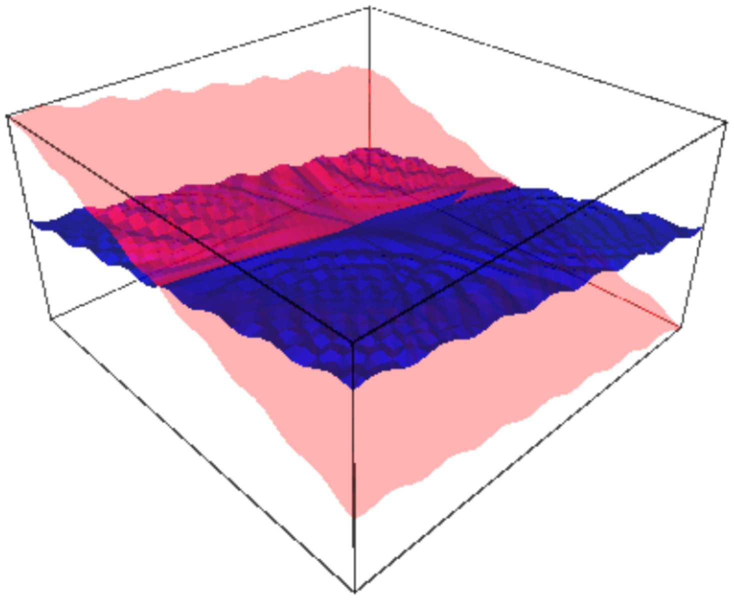
```
sage: var('x,y')
(x, y)
sage: plot3d(sin(x - y)*y*cos(x), (x, -3, 3), (y, -3, 3), mesh=True,
....:        thickness=2, viewer='threejs')
Graphics3d Object
```

Two wobby translucent planes:





```
sage: x,y = var('x,y')
sage: P = plot3d(x + y + sin(x*y), (x, -10, 10), (y, -10, 10),
....:            opacity=0.87, color='blue')
sage: Q = plot3d(x - 2*y - cos(x*y),(x, -10, 10), (y, -10, 10),
....:            opacity=0.3, color='red')
sage: P + Q
Graphics3d Object
```



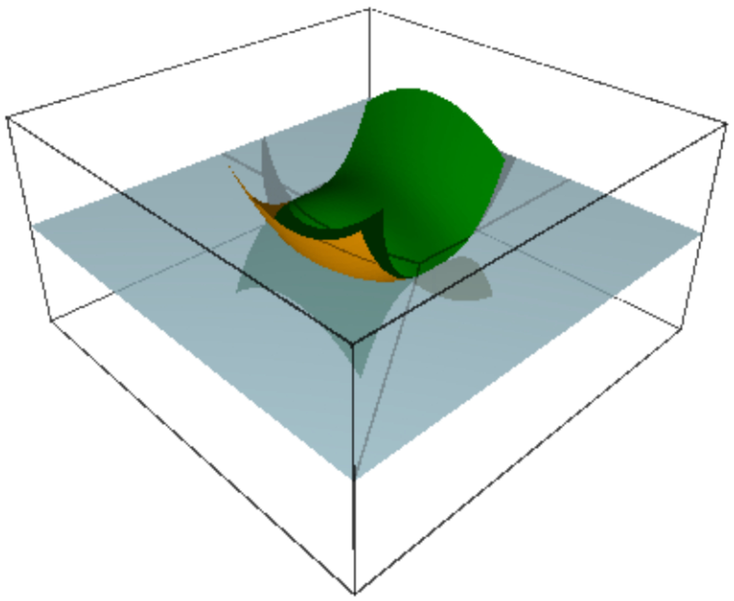
We draw two parametric surfaces and a transparent plane:

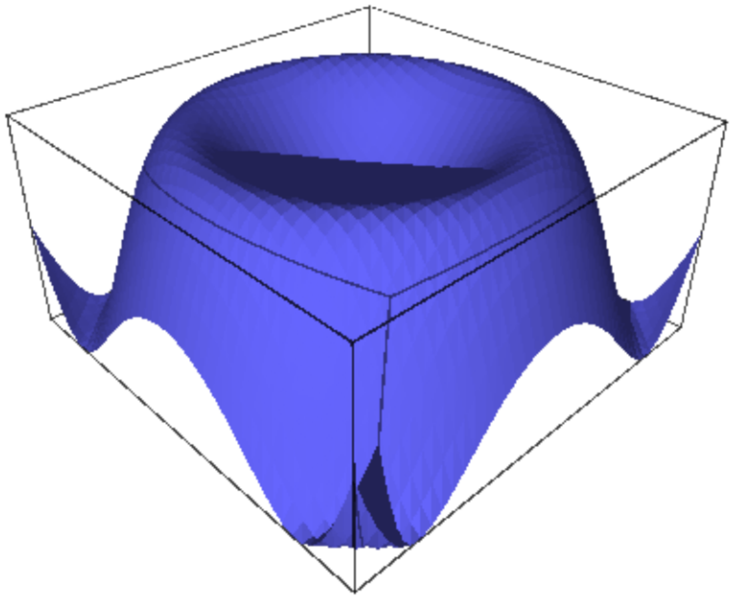
```
sage: L = plot3d(lambda x,y: 0, (-5,5), (-5,5), color="lightblue", opacity=0.8)
sage: P = plot3d(lambda x,y: 4 - x^3 - y^2, (-2,2), (-2,2), color='green')
sage: Q = plot3d(lambda x,y: x^3 + y^2 - 4, (-2,2), (-2,2), color='orange')
sage: L + P + Q
Graphics3d Object
```

We draw the “Sinus” function (water ripple-like surface):

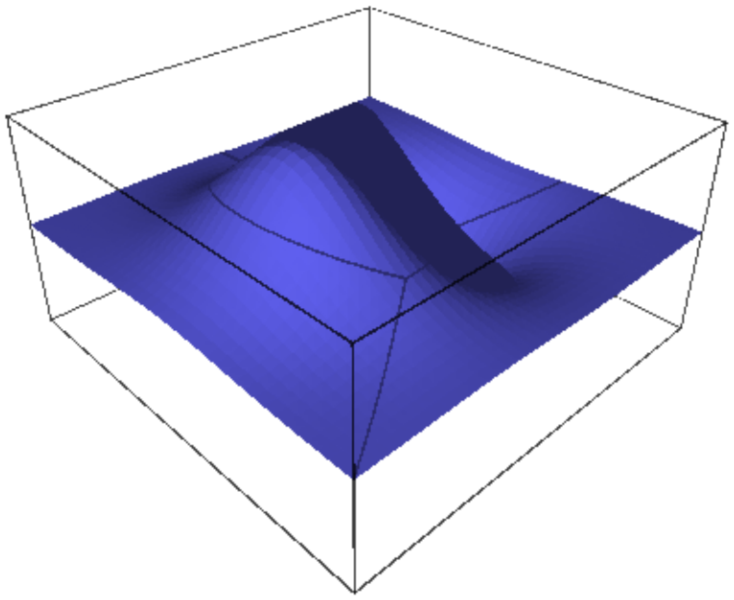
```
sage: x, y = var('x y')
sage: plot3d(sin(pi*(x^2 + y^2))/2, (x, -1, 1), (y, -1, 1))
Graphics3d Object
```

Hill and valley (flat surface with a bump and a dent):





```
sage: x, y = var('x y')
sage: plot3d(4*x*exp(-x^2 - y^2), (x, -2, 2), (y, -2, 2))
Graphics3d Object
```



An example of a transformation:

```
sage: r, phi, z = var('r phi z')
sage: trans = (r*cos(phi), r*sin(phi), z)
sage: plot3d(cos(r), (r, 0, 17*pi/2), (phi, 0, 2*pi), transformation=trans,␣
↪opacity=0.87).show(aspect_ratio=(1,1,2), frame=False)
```

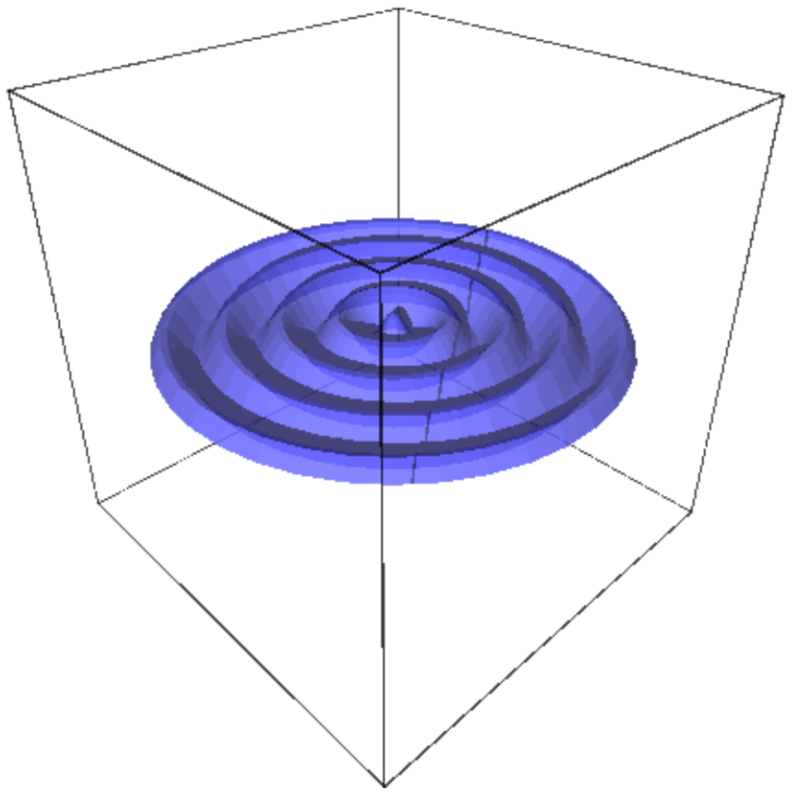
An example of a transformation with symbolic vector:

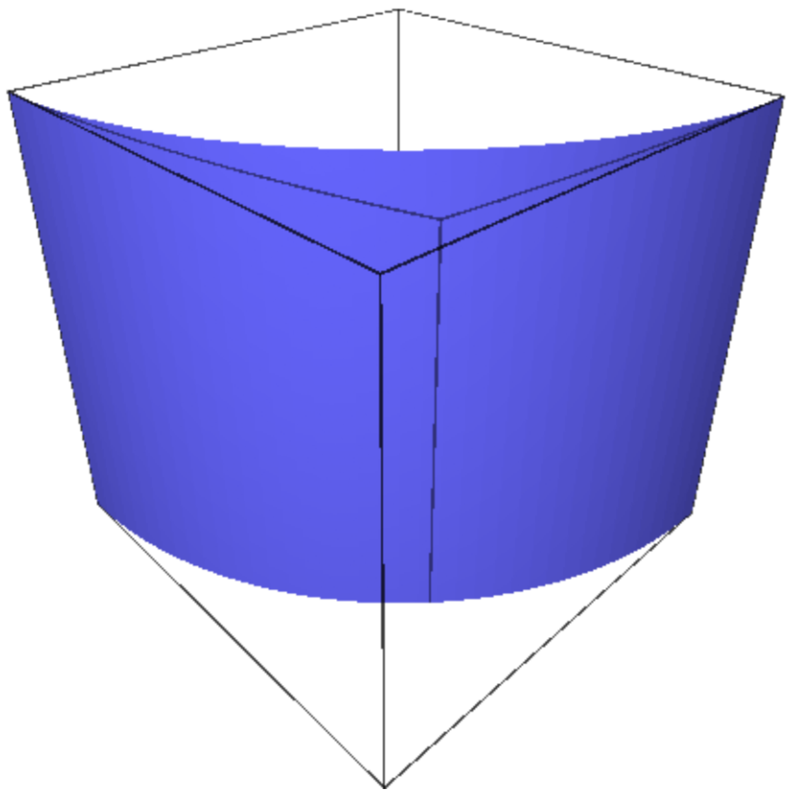
```
sage: cylindrical(r, theta, z) = [r*cos(theta), r*sin(theta), z]
sage: plot3d(3, (theta, 0, pi/2), (z, 0, pi/2), transformation=cylindrical)
Graphics3d Object
```

Many more examples of transformations:

```
sage: u, v, w = var('u v w')
sage: rectangular=(u,v,w)
sage: spherical=(w*cos(u)*sin(v),w*sin(u)*sin(v),w*cos(v))
sage: cylindric_radial=(w*cos(u),w*sin(u),v)
```

(continues on next page)





(continued from previous page)

```
sage: cylindric_axial=(v*cos(u),v*sin(u),w)
sage: parabolic_cylindrical=(w*v,(v^2-w^2)/2,u)
```

Plot a constant function of each of these to get an idea of what it does:

```
sage: A = plot3d(2,(u,-pi,pi),(v,0,pi),transformation=rectangular,plot_points=[100,
↪100])
sage: B = plot3d(2,(u,-pi,pi),(v,0,pi),transformation=spherical,plot_points=[100,
↪100])
sage: C = plot3d(2,(u,-pi,pi),(v,0,pi),transformation=cylindric_radial,plot_
↪points=[100,100])
sage: D = plot3d(2,(u,-pi,pi),(v,0,pi),transformation=cylindric_axial,plot_
↪points=[100,100])
sage: E = plot3d(2,(u,-pi,pi),(v,-pi,pi),transformation=parabolic_cylindrical,plot_
↪points=[100,100])
sage: @interact
....: def _(which_plot=[A,B,C,D,E]):
....:     show(which_plot)
...Interactive function <function _ at ...> with 1 widget
  which_plot: Dropdown(description='which_plot', options=(Graphics3d Object,␣
↪Graphics3d Object, Graphics3d Object, Graphics3d Object, Graphics3d Object),␣
↪value=Graphics3d Object)
```

Now plot a function:

```
sage: g=3+sin(4*u)/2+cos(4*v)/2
sage: F = plot3d(g,(u,-pi,pi),(v,0,pi),transformation=rectangular,plot_points=[100,
↪100])
sage: G = plot3d(g,(u,-pi,pi),(v,0,pi),transformation=spherical,plot_points=[100,
↪100])
sage: H = plot3d(g,(u,-pi,pi),(v,0,pi),transformation=cylindric_radial,plot_
↪points=[100,100])
sage: I = plot3d(g,(u,-pi,pi),(v,0,pi),transformation=cylindric_axial,plot_
↪points=[100,100])
sage: J = plot3d(g,(u,-pi,pi),(v,0,pi),transformation=parabolic_cylindrical,plot_
↪points=[100,100])
sage: @interact
....: def _(which_plot=[F, G, H, I, J]):
....:     show(which_plot)
...Interactive function <function _ at ...> with 1 widget
  which_plot: Dropdown(description='which_plot', options=(Graphics3d Object,␣
↪Graphics3d Object, Graphics3d Object, Graphics3d Object, Graphics3d Object),␣
↪value=Graphics3d Object)
```

sage.plot.plot3d.plot3d.**plot3d_adaptive**(*f*, *x_range*, *y_range*, *color='automatic'*, *grad_f=None*, *max_bend=0.5*, *max_depth=5*, *initial_depth=4*, *num_colors=128*, ***kwds*)

Adaptive 3d plotting of a function of two variables.

This is used internally by the plot3d command when the option `adaptive=True` is given.

INPUT:

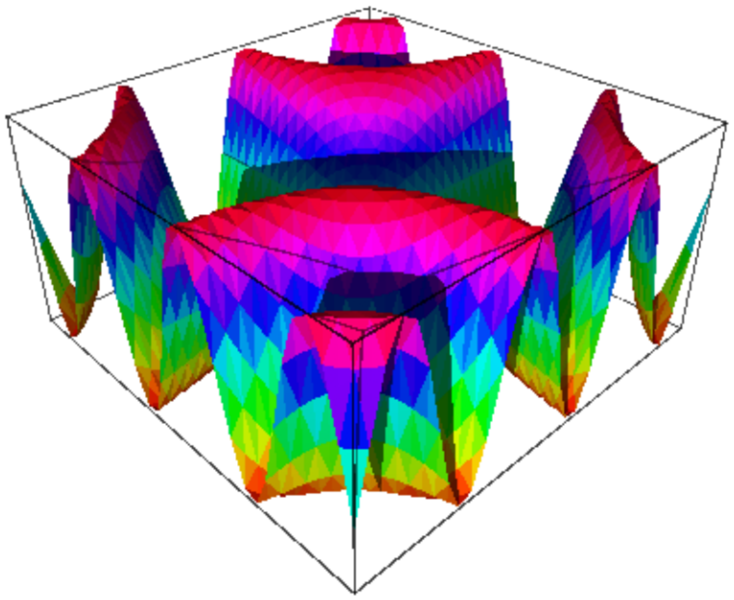
- `f` – a symbolic function or a Python function of 3 variables.

- `x_range` – x range of values: 2-tuple (xmin, xmax) or 3-tuple (x,xmin,xmax)
- `y_range` – y range of values: 2-tuple (ymin, ymax) or 3-tuple (y,ymin,ymax)
- `grad_f` – gradient of f as a Python function
- `color` – "automatic" - a rainbow of num_colors colors
- `num_colors` – (default: 128) number of colors to use with default color
- `max_bend` – (default: 0.5)
- `max_depth` – (default: 5)
- `initial_depth` – (default: 4)
- `**kwds` – standard graphics parameters

EXAMPLES:

We plot $\sin(xy)$:

```
sage: from sage.plot.plot3d.plot3d import plot3d_adaptive
sage: x, y = var('x,y')
sage: plot3d_adaptive(sin(x*y), (x, -pi, pi), (y, -pi, pi), initial_depth=5)
Graphics3d Object
```



`sage.plot.plot3d.plot3d.`**`spherical_plot3d`**(*f*, *urange*, *vrange*, ***kwds*)

Plots a function in spherical coordinates. This function is equivalent to:

```
sage: r,u,v=var('r,u,v')
sage: f=u*v; urange=(u,0,pi); vrange=(v,0,pi)
sage: T = (r*cos(u)*sin(v), r*sin(u)*sin(v), r*cos(v), [u,v])
sage: plot3d(f, urange, vrange, transformation=T)
Graphics3d Object
```

or equivalently:

```
sage: T = Spherical('radius', ['azimuth', 'inclination'])
sage: f=lambda u,v: u*v; urange=(u,0,pi); vrange=(v,0,pi)
sage: plot3d(f, urange, vrange, transformation=T)
Graphics3d Object
```

INPUT:

- `f` – a symbolic expression or function of two variables
- `urange` – a 3-tuple (u, u_min, u_max), the domain of the azimuth variable
- `vrange` – a 3-tuple (v, v_min, v_max), the domain of the inclination variable

EXAMPLES:

A sphere of radius 2:

```
sage: x,y = var('x,y')
sage: spherical_plot3d(2, (x, 0, 2*pi), (y, 0, pi))
Graphics3d Object
```

The real and imaginary parts of a spherical harmonic with $l = 2$ and $m = 1$:

```
sage: phi, theta = var('phi, theta')
sage: Y = spherical_harmonic(2, 1, theta, phi)
sage: rea = spherical_plot3d(abs(real(Y)), (phi, 0, 2*pi), (theta, 0, pi), color=
↪'blue', opacity=0.6)
sage: ima = spherical_plot3d(abs(imag(Y)), (phi, 0, 2*pi), (theta, 0, pi), color=
↪'red', opacity=0.6)
sage: (rea + ima).show(aspect_ratio=1)  # long time (4s on sage.math, 2011)
```

A drop of water:

```
sage: x,y = var('x,y')
sage: spherical_plot3d(e^-y, (x, 0, 2*pi), (y, 0, pi), opacity=0.5).
↪show(frame=False)
```

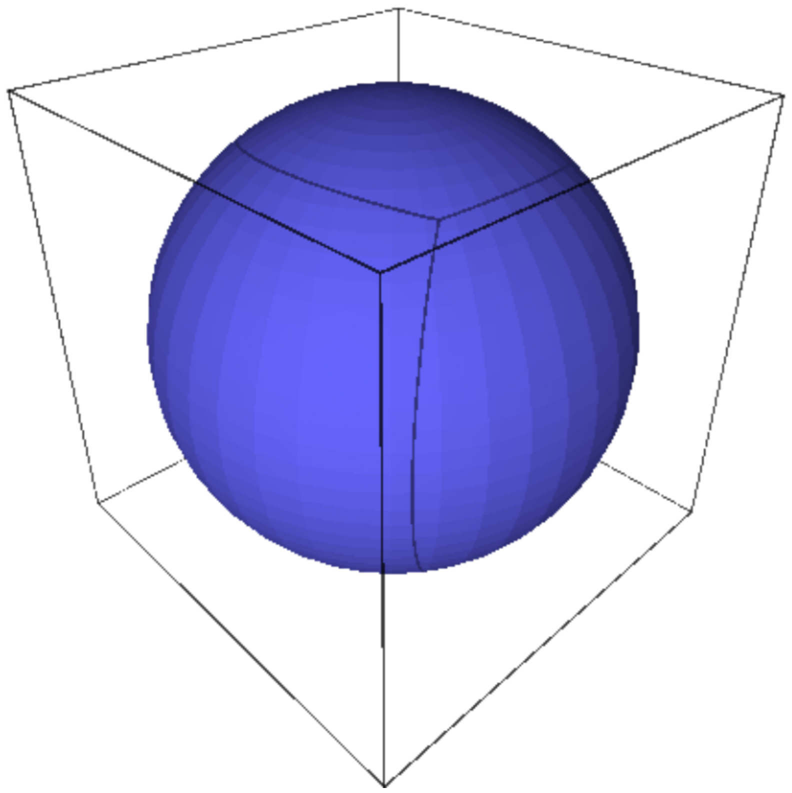
An object similar to a heart:

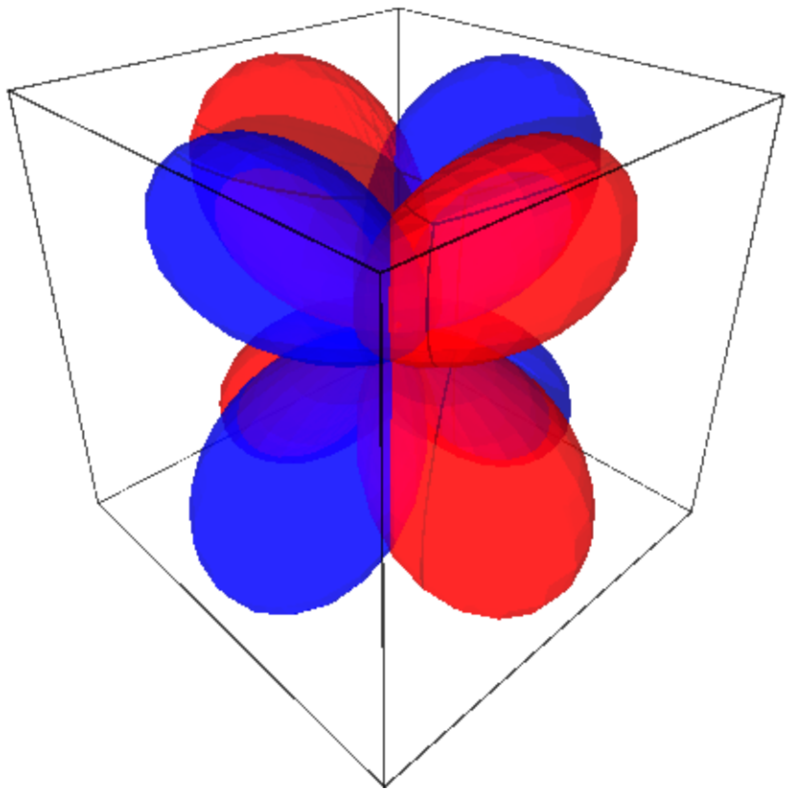
```
sage: x,y = var('x,y')
sage: spherical_plot3d((2 + cos(2*x))*(y + 1), (x, 0, 2*pi), (y, 0, pi),␣
↪rgbcolor=(1, .1, .1))
Graphics3d Object
```

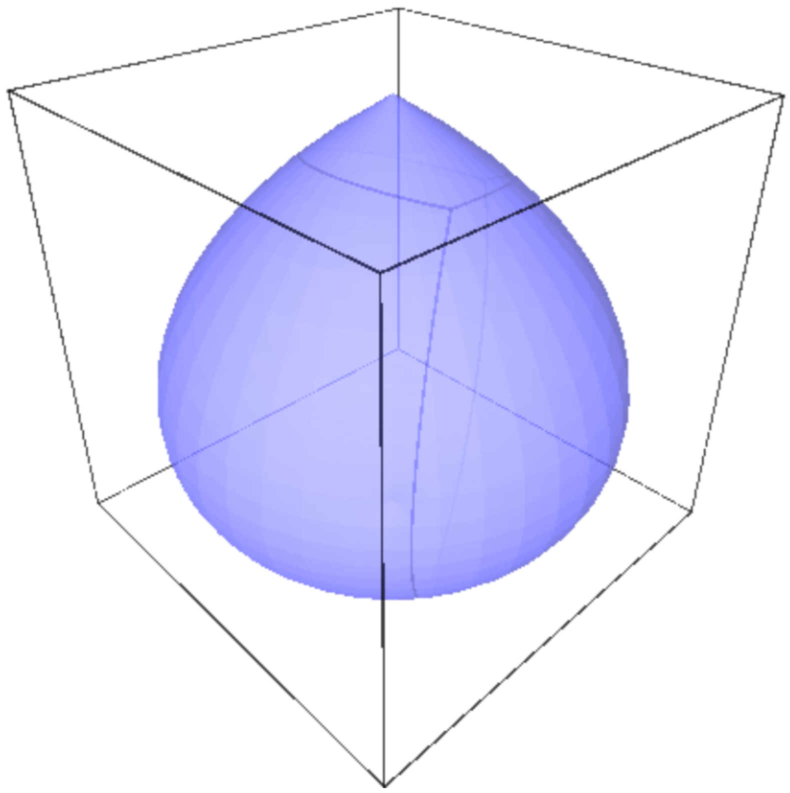
Some random figures:

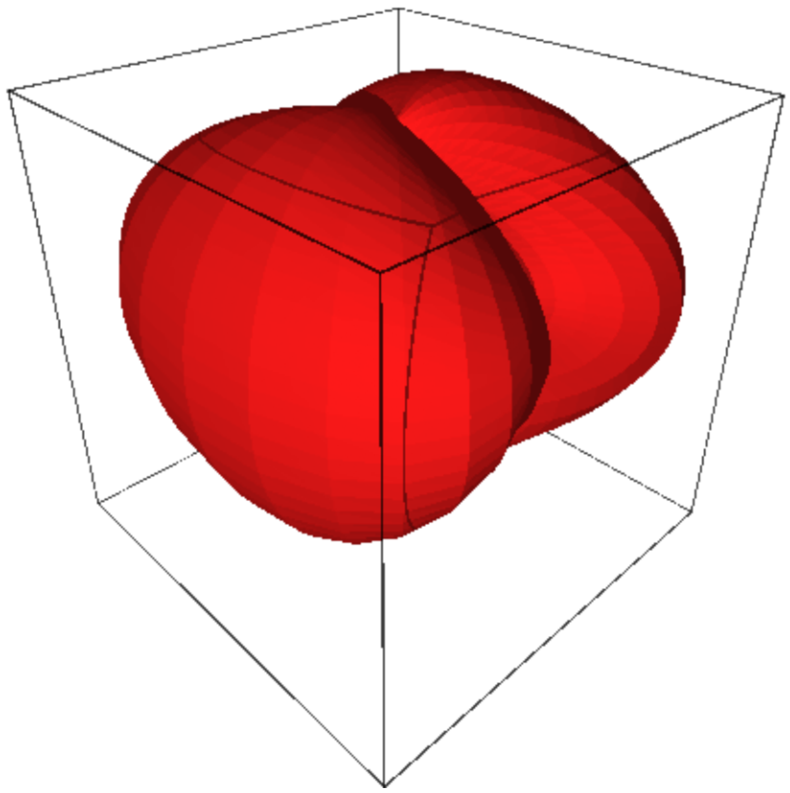
```
sage: x,y = var('x,y')
sage: spherical_plot3d(1 + sin(5*x)/5, (x, 0, 2*pi), (y, 0, pi), rgbcolor=(1, 0.5,␣
```

(continues on next page)



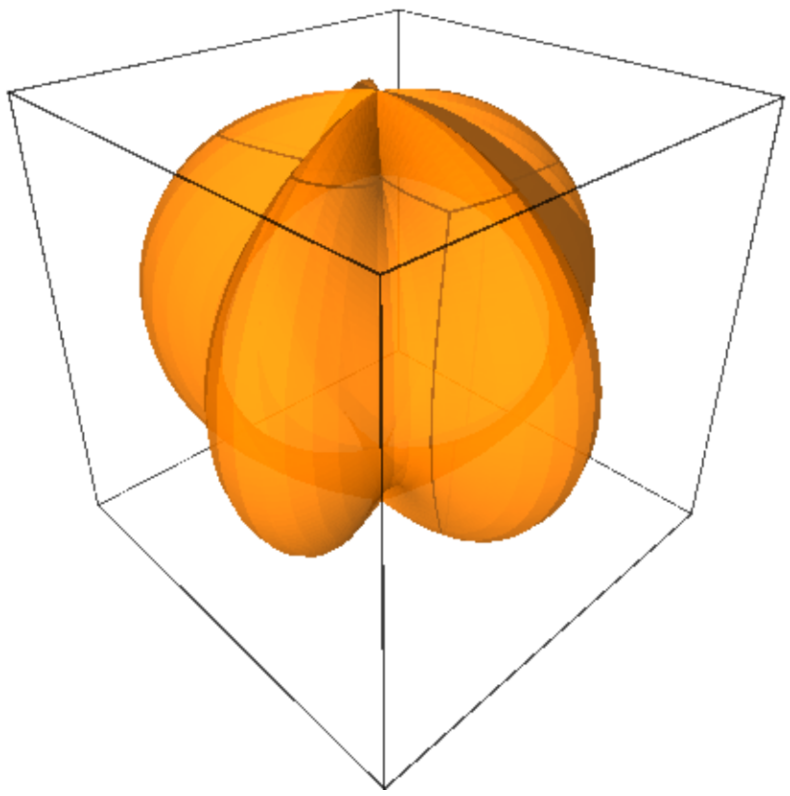






(continued from previous page)

```
↪0), plot_points=(80, 80), opacity=0.7)
Graphics3d Object
```



```
sage: x, y = var('x,y')
sage: spherical_plot3d(1 + 2*cos(2*y), (x, 0, 3*pi/2), (y, 0, pi)).show(aspect_
↪ratio=(1, 1, 1))
```

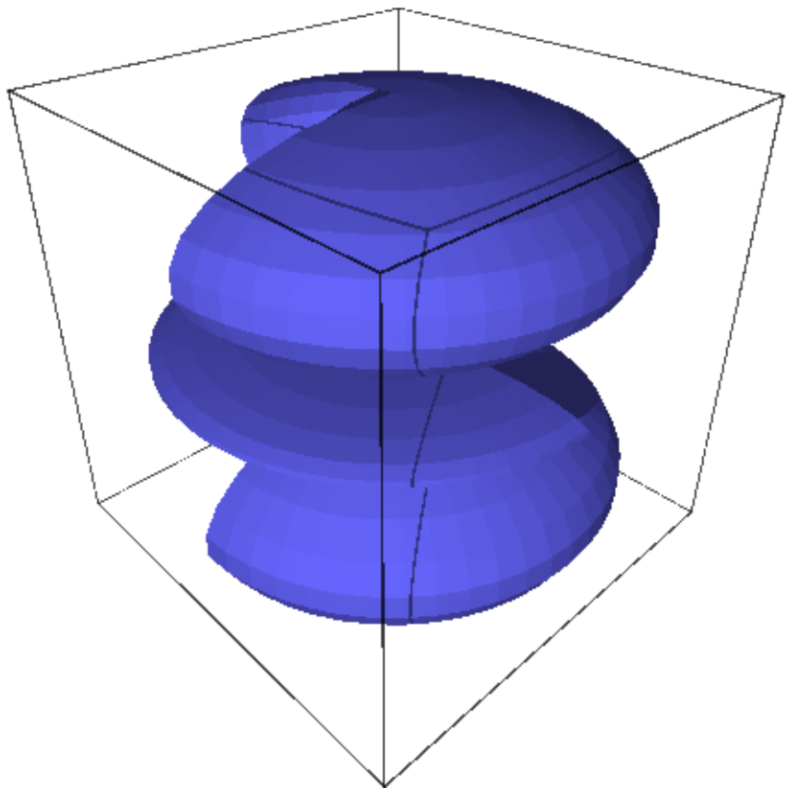
2.2 Parametric plots

sage.plot.plot3d.parametric_plot3d.**parametric_plot3d**(*f*, *urange*, *vrange=None*, *plot_points='automatic'*, *boundary_style=None*, ***kwds*)

Return a parametric three-dimensional space curve or surface.

There are four ways to call this function:

- `parametric_plot3d([f_x, f_y, f_z], (u_min, u_max))`: f_x, f_y, f_z are three functions and $u_{\min}$ and $u_{\max}$ are real numbers
- `parametric_plot3d([f_x, f_y, f_z], (u, u_min, u_max))`: f_x, f_y, f_z can be viewed as functions of u



- parametric_plot3d([f_x, f_y, f_z], (u_min, u_max), (v_min, v_max)): f_x, f_y, f_z are each functions of two variables
- parametric_plot3d([f_x, f_y, f_z], (u, u_min, u_max), (v, v_min, v_max)): f_x, f_y, f_z can be viewed as functions of u and v

INPUT:

- f - a 3-tuple of functions or expressions, or vector of size 3
- urange - a 2-tuple (u_min, u_max) or a 3-tuple (u, u_min, u_max)
- vrange - (optional - only used for surfaces) a 2-tuple (v_min, v_max) or a 3-tuple (v, v_min, v_max)
- plot_points - (default: "automatic", which is 75 for curves and [40,40] for surfaces) initial number of sample points in each parameter; an integer for a curve, and a pair of integers for a surface.
- boundary_style - (default: None, no boundary) a dict that describes how to draw the boundaries of regions by giving options that are passed to the line3d command.
- mesh - bool (default: False) whether to display mesh grid lines
- dots - bool (default: False) whether to display dots at mesh grid points

Note:

1. By default for a curve any points where f_x, f_y, or f_z do not evaluate to a real number are skipped.
2. Currently for a surface f_x, f_y, and f_z have to be defined everywhere. This will change.
3. mesh and dots are not supported when using the Tachyon ray tracer renderer.

EXAMPLES: We demonstrate each of the four ways to call this function.

1. A space curve defined by three functions of 1 variable:

```
sage: parametric_plot3d((sin, cos, lambda u: u/10), (0,20))
Graphics3d Object
```

   Note above the lambda function, which creates a callable Python function that sends u to $u/10$.

2. Next we draw the same plot as above, but using symbolic functions:

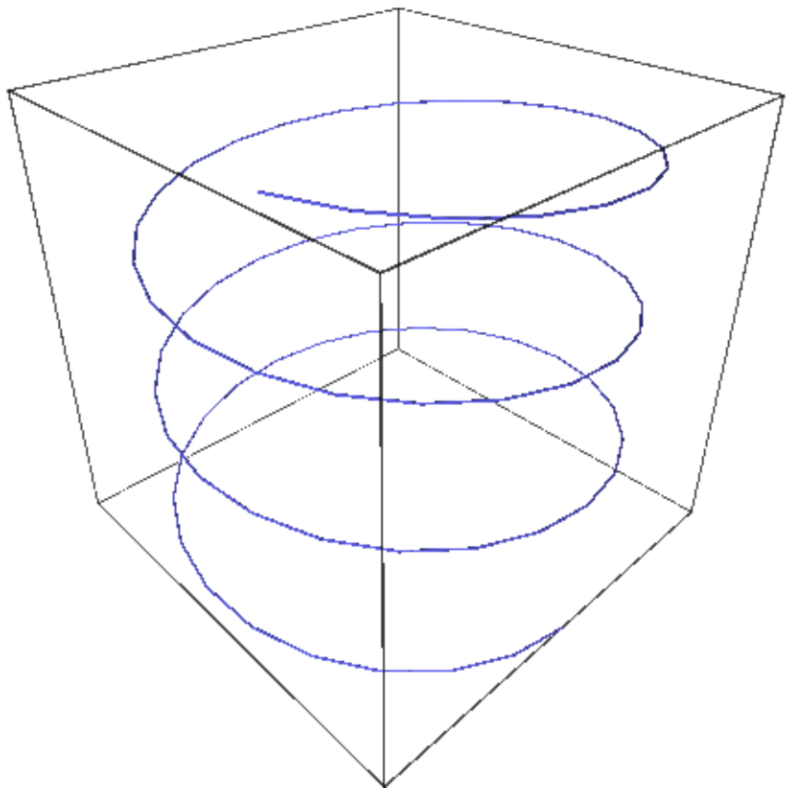
```
sage: u = var('u')
sage: parametric_plot3d((sin(u), cos(u), u/10), (u,0,20))
Graphics3d Object
```

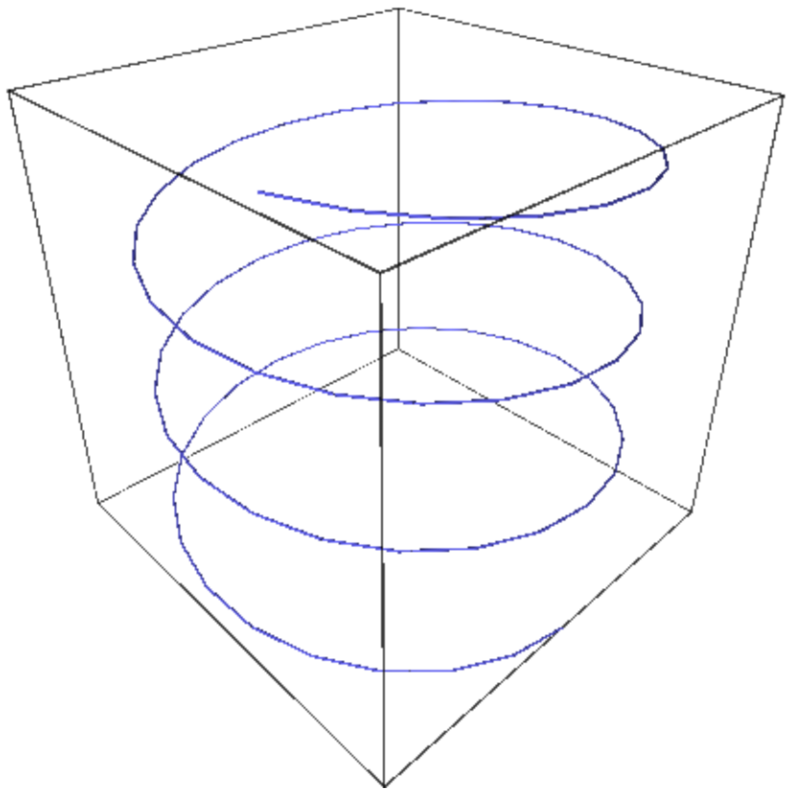
3. We draw a parametric surface using 3 Python functions (defined using lambda):

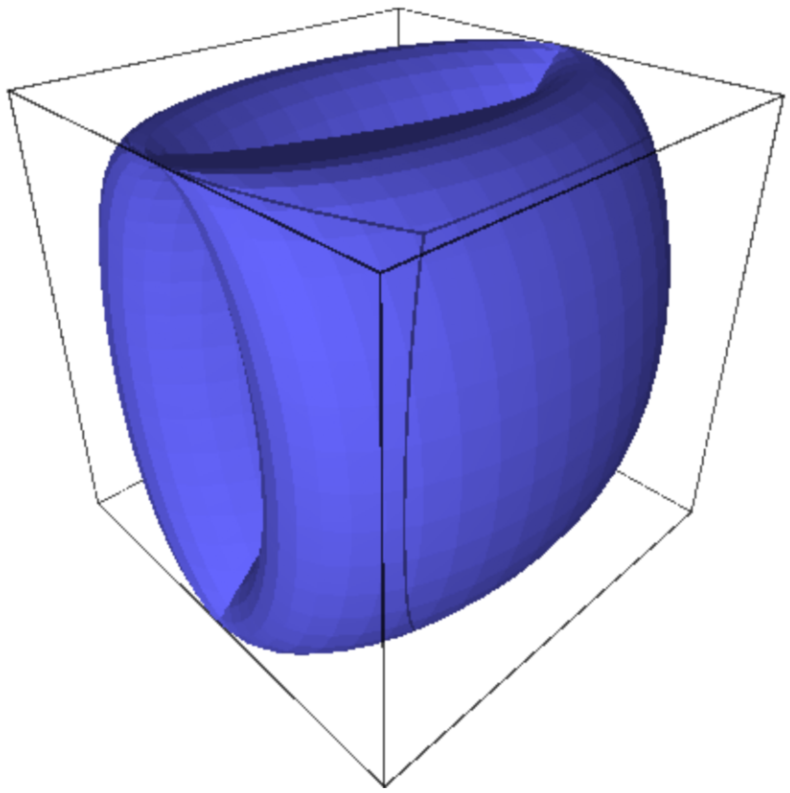
```
sage: f = (lambda u,v: cos(u), lambda u,v: sin(u)+cos(v), lambda u,v:␣
↪sin(v))
sage: parametric_plot3d(f, (0,2*pi), (-pi,pi))
Graphics3d Object
```

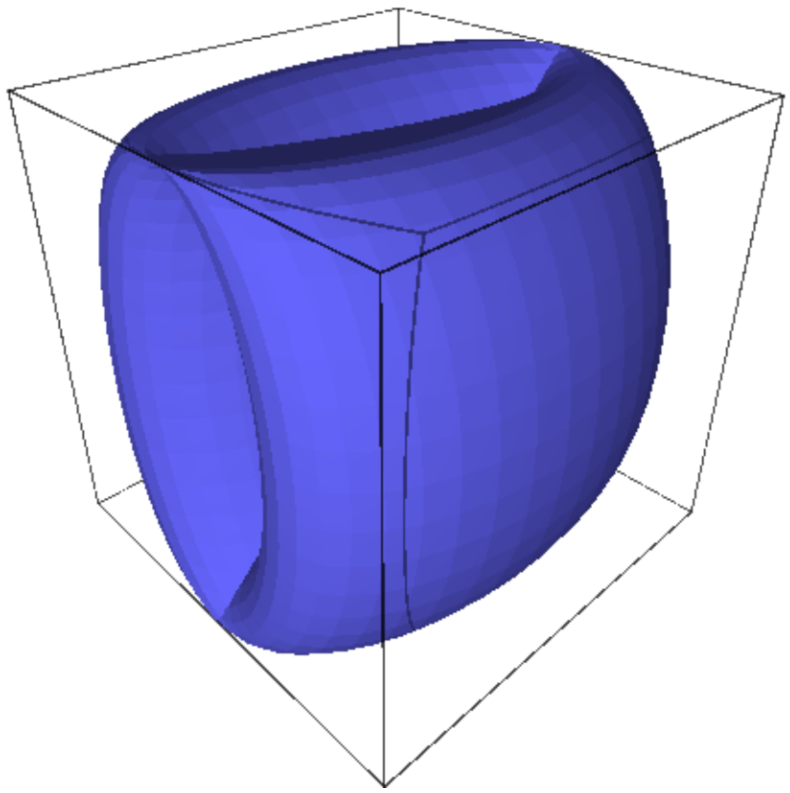
4. The same surface, but where the defining functions are symbolic:

```
sage: u, v = var('u,v')
sage: parametric_plot3d((cos(u), sin(u)+cos(v), sin(v)), (u,0,2*pi), (v,-
↪pi,pi))
Graphics3d Object
```



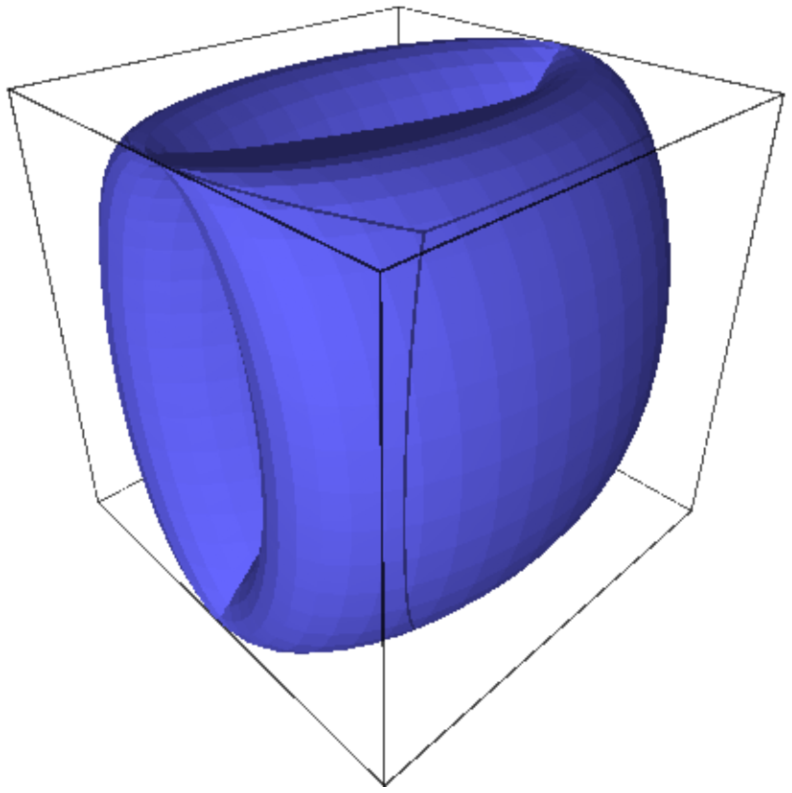






The surface, but with a mesh:

```
sage: u, v = var('u,v')
sage: parametric_plot3d((cos(u), sin(u)+cos(v), sin(v)), (u,0,2*pi), (v,-pi,pi),␣
↪mesh=True)
Graphics3d Object
```



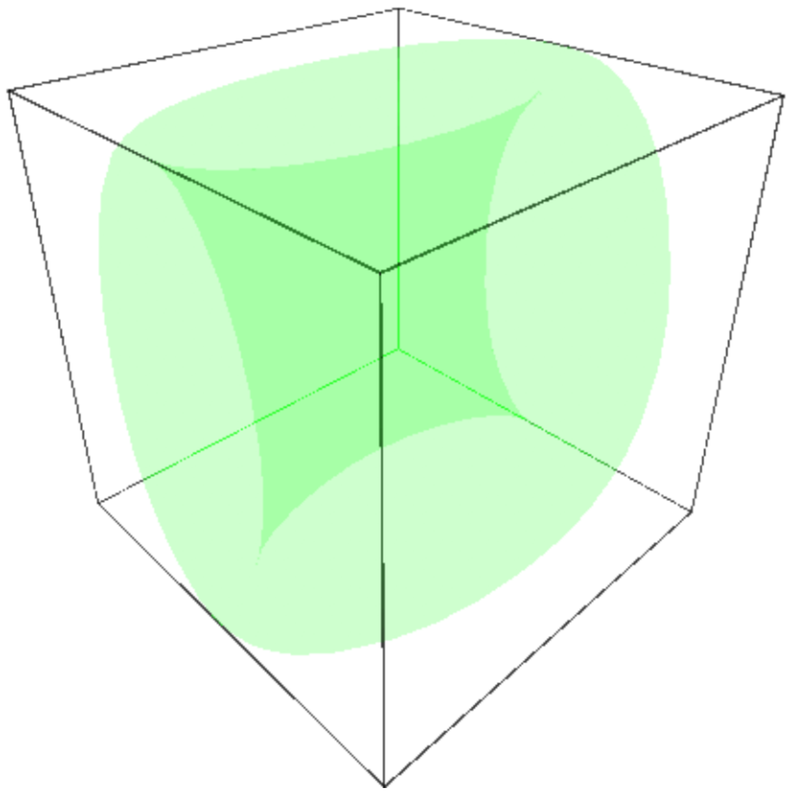
We increase the number of plot points, and make the surface green and transparent:

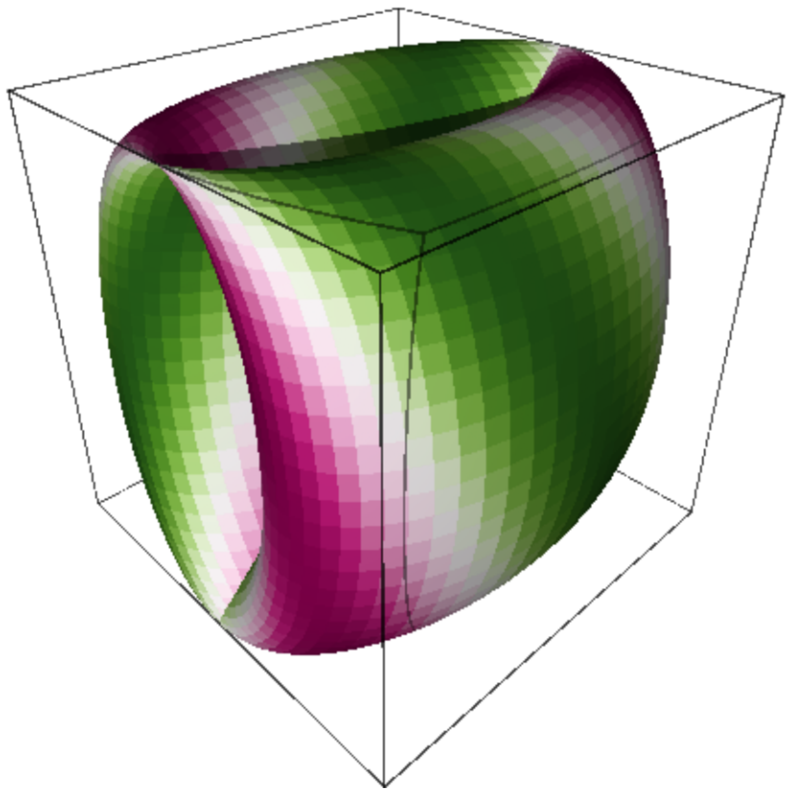
```
sage: parametric_plot3d((cos(u), sin(u)+cos(v), sin(v)), (u,0,2*pi), (v,-pi,pi),
....: color='green', opacity=0.1, plot_points=[30,30])
Graphics3d Object
```

One can also color the surface using a coloring function and a colormap as follows. Note that the coloring function must take values in the interval [0,1].

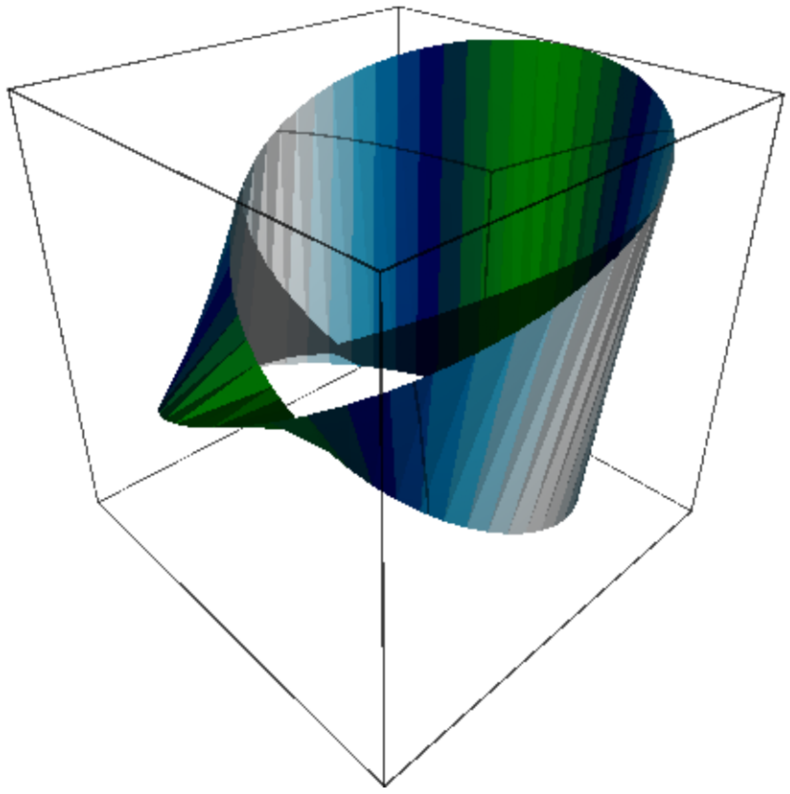
```
sage: u,v = var('u,v')
sage: def cf(u,v): return sin(u+v/2)**2
sage: P = parametric_plot3d((cos(u), sin(u)+cos(v), sin(v)),
....:   (u,0,2*pi), (v,-pi,pi), color=(cf,colormaps.PiYG), plot_points=[60,60])
sage: P.show(viewer='tachyon')
```

Another example, a colored Möbius band:





```
sage: cm = colormaps.ocean
sage: def c(x,y): return sin(x*y)**2
sage: from sage.plot.plot3d.parametric_surface import MoebiusStrip
sage: MoebiusStrip(5, 1, plot_points=200, color=(c,cm))
Graphics3d Object
```



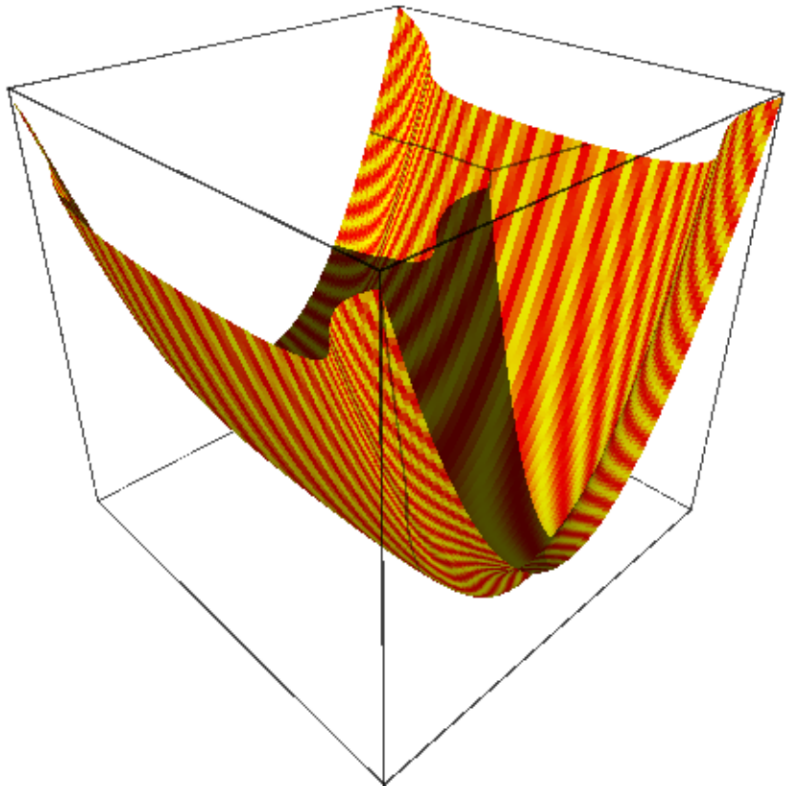
Yet another colored example:

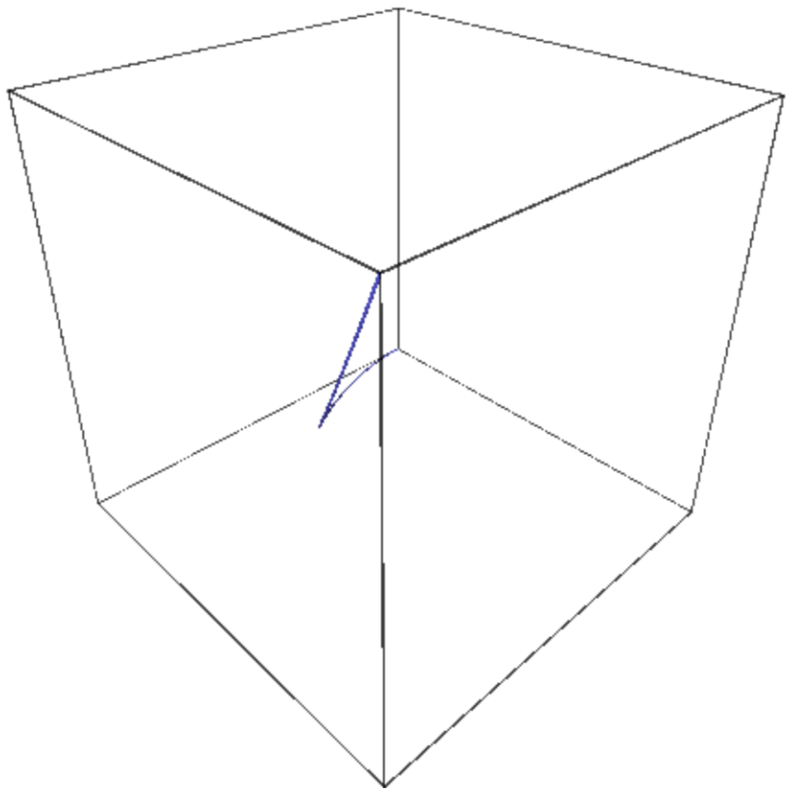
```
sage: from sage.plot.plot3d.parametric_surface import ParametricSurface
sage: cm = colormaps.autumn
sage: def c(x,y): return sin(x*y)**2
sage: def g(x,y): return x, y+sin(y), x**2 + y**2
sage: ParametricSurface(g, (srange(-10,10,0.1), srange(-5,5.0,0.1)), color=(c,cm))
Graphics3d Object
```

We call the space curve function but with polynomials instead of symbolic variables.

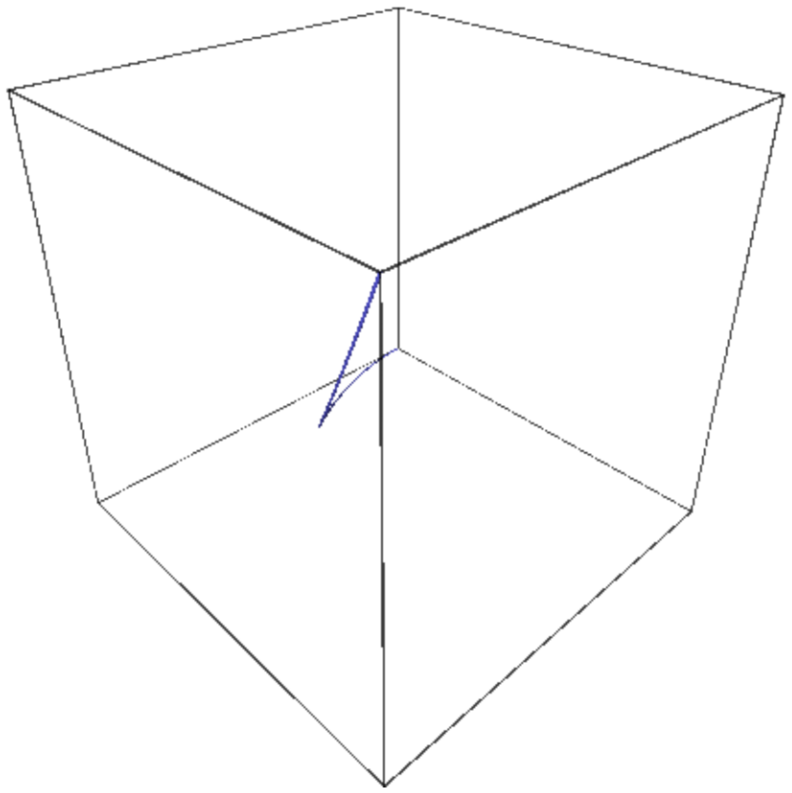
```
sage: R.<t> = RDF[]
sage: parametric_plot3d((t, t^2, t^3), (t,0,3))
Graphics3d Object
```

Next we plot the same curve, but because we use (0, 3) instead of (t, 0, 3), each polynomial is viewed as a callable function of one variable:





```
sage: parametric_plot3d((t, t^2, t^3), (0,3))
Graphics3d Object
```



We do a plot but mix a symbolic input, and an integer:

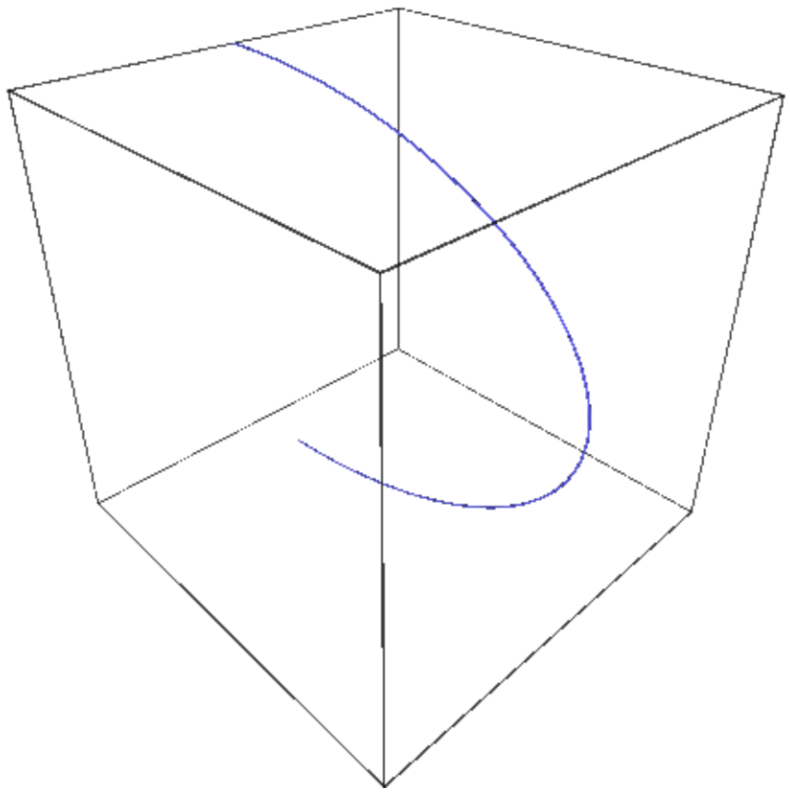
```
sage: t = var('t')
sage: parametric_plot3d((1, sin(t), cos(t)), (t,0,3))
Graphics3d Object
```

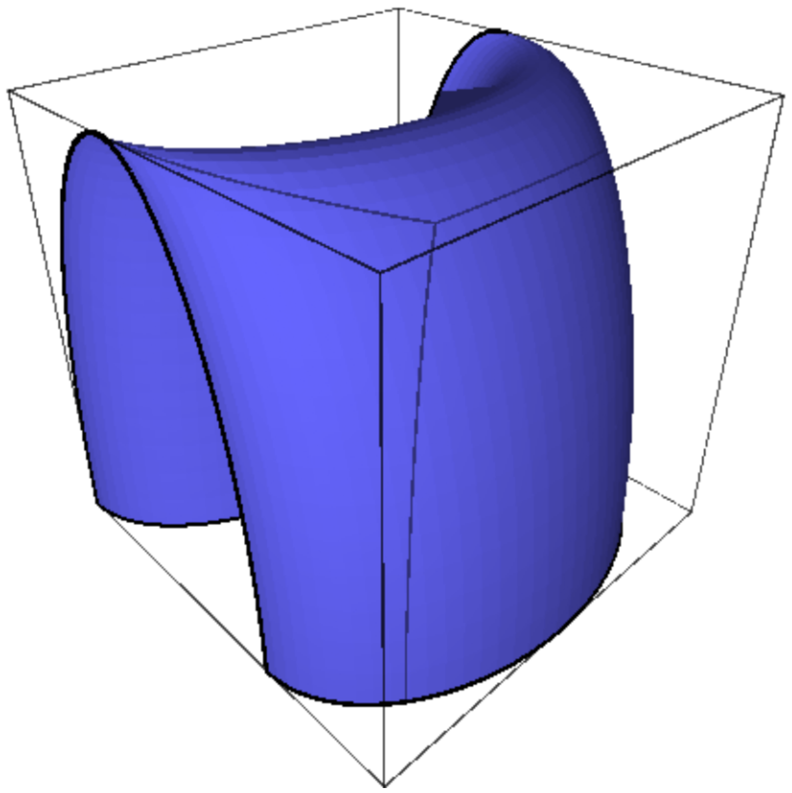
We specify a boundary style to show us the values of the function at its extrema:

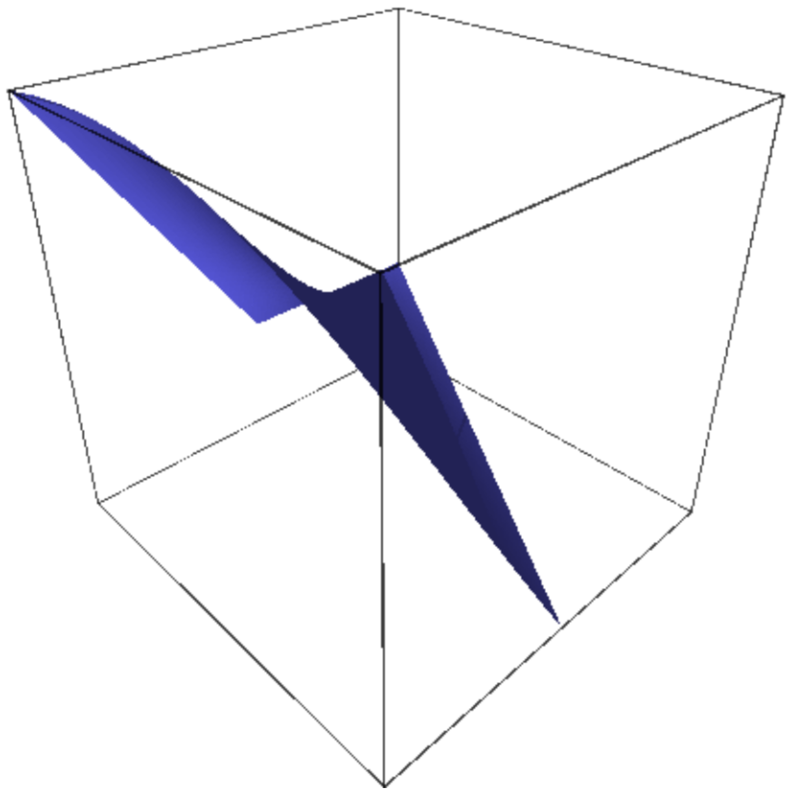
```
sage: u, v = var('u,v')
sage: parametric_plot3d((cos(u), sin(u)+cos(v), sin(v)), (u,0,pi), (v,0,pi),
....:                   boundary_style={"color": "black", "thickness": 2})
Graphics3d Object
```

We can plot vectors:

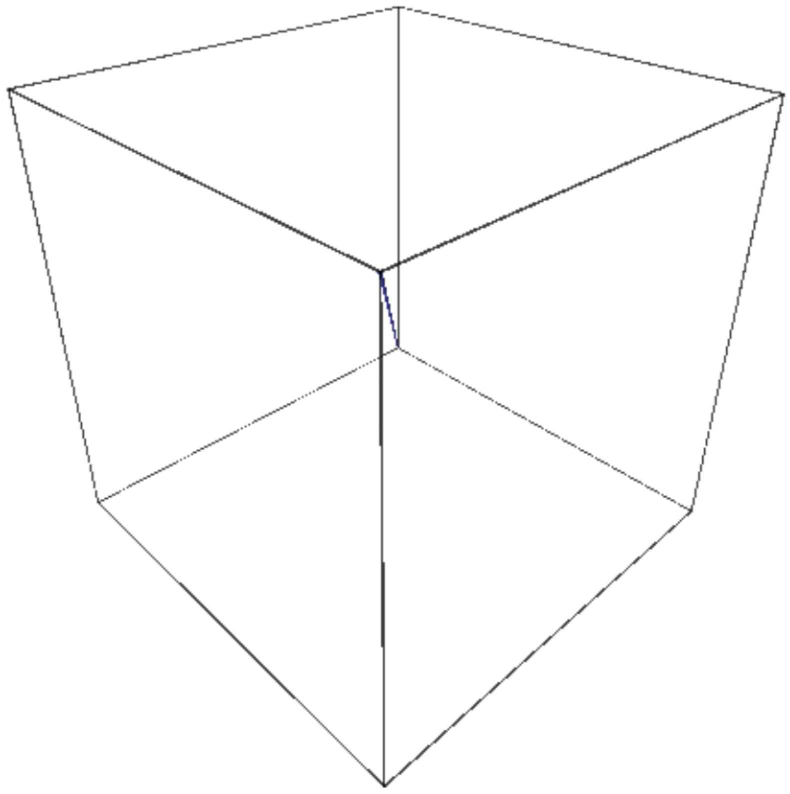
```
sage: x,y = var('x,y')
sage: parametric_plot3d(vector([x-y, x*y, x*cos(y)]), (x,0,2), (y,0,2))
Graphics3d Object
```







```
sage: t = var('t')
sage: p = vector([1,2,3])
sage: q = vector([2,-1,2])
sage: parametric_plot3d(p*t+q, (t,0,2))
Graphics3d Object
```



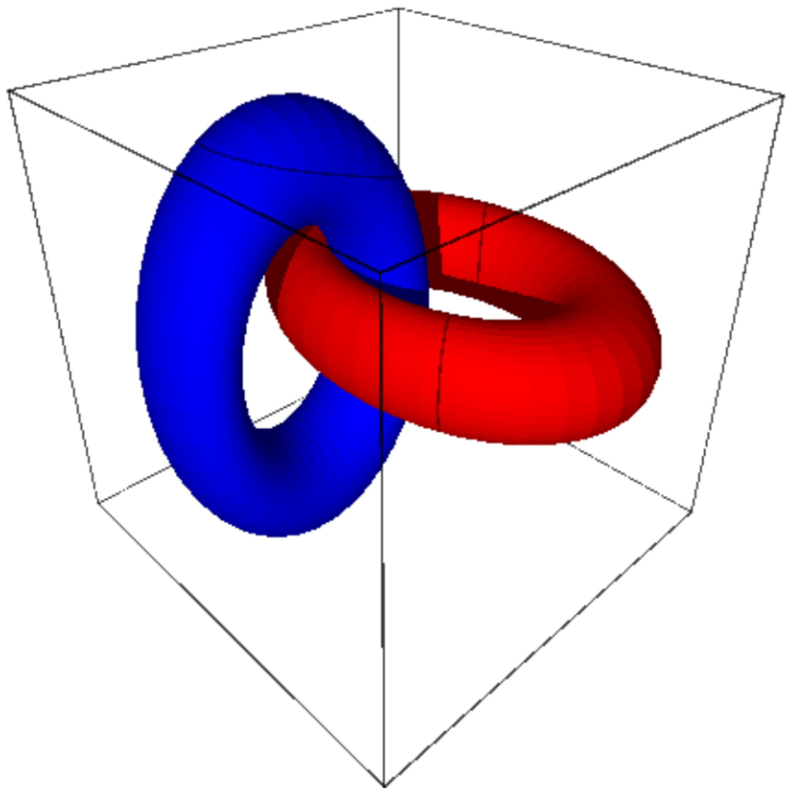
Any options you would normally use to specify the appearance of a curve are valid as entries in the `boundary_style` dict.

MANY MORE EXAMPLES:

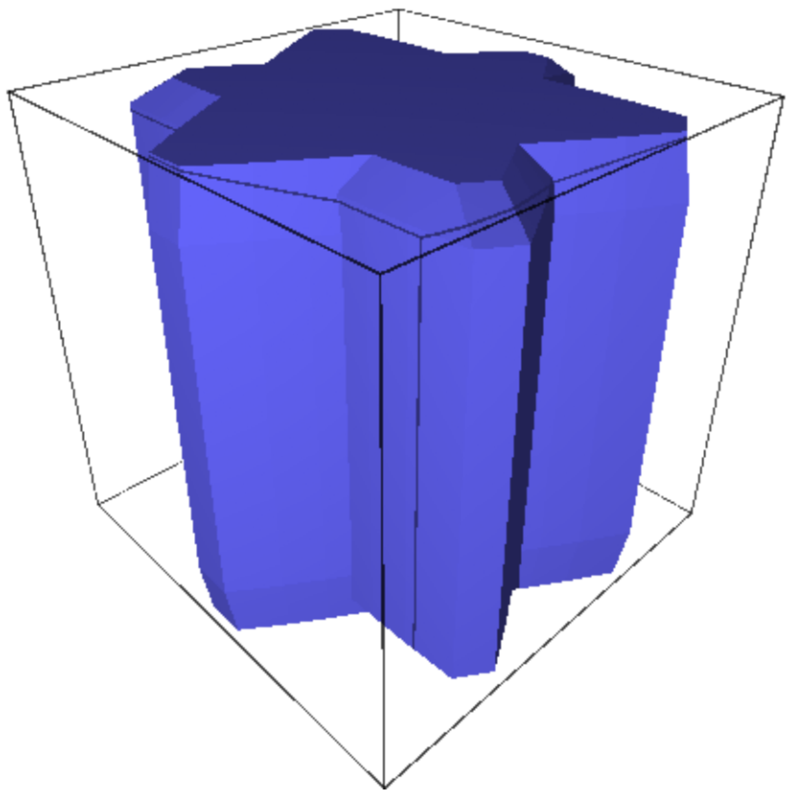
We plot two interlinked tori:

```
sage: u, v = var('u,v')
sage: f1 = (4+(3+cos(v))*sin(u), 4+(3+cos(v))*cos(u), 4+sin(v))
sage: f2 = (8+(3+cos(v))*cos(u), 3+sin(v), 4+(3+cos(v))*sin(u))
sage: p1 = parametric_plot3d(f1, (u,0,2*pi), (v,0,2*pi), texture="red")
sage: p2 = parametric_plot3d(f2, (u,0,2*pi), (v,0,2*pi), texture="blue")
sage: p1 + p2
Graphics3d Object
```

A cylindrical Star of David:



```
sage: u,v = var('u v')
sage: K = (abs(cos(u))^200+abs(sin(u))^200)^(-1.0/200)
sage: f_x = cos(u) * cos(v) * (abs(cos(3*v/4))^500+abs(sin(3*v/4))^500)^(-1/260) * K
sage: f_y = cos(u) * sin(v) * (abs(cos(3*v/4))^500+abs(sin(3*v/4))^500)^(-1/260) * K
sage: f_z = sin(u) * K
sage: parametric_plot3d([f_x, f_y, f_z], (u, -pi, pi), (v, 0, 2*pi))
Graphics3d Object
```



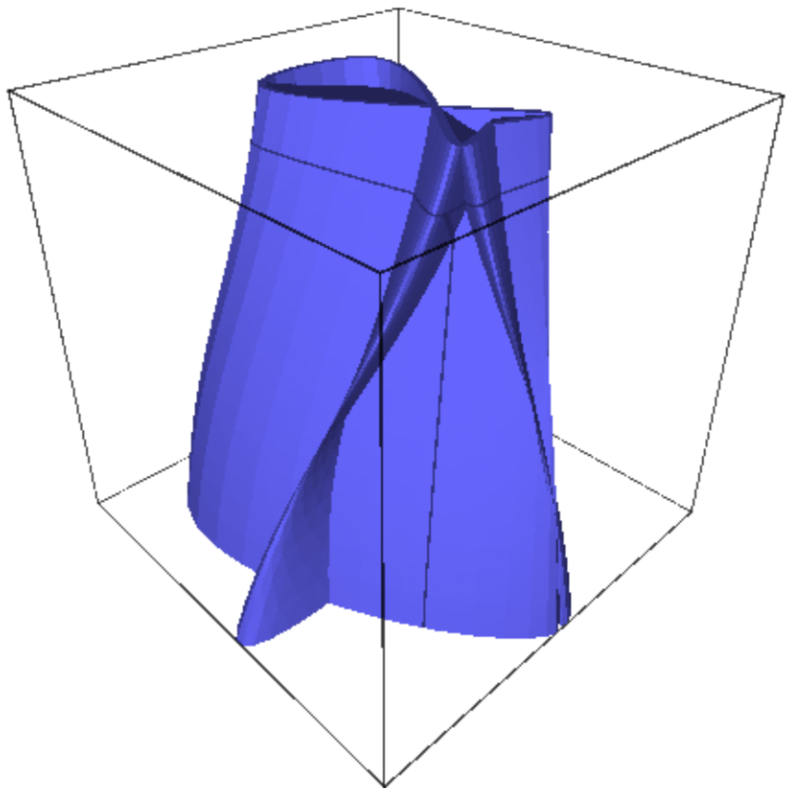
Double heart:

```
sage: u, v = var('u,v')
sage: G1 = abs(sqrt(2)*tanh((u/sqrt(2))))
sage: G2 = abs(sqrt(2)*tanh((v/sqrt(2))))
sage: f_x = (abs(v) - abs(u) - G1 + G2)*sin(v)
sage: f_y = (abs(v) - abs(u) - G1 - G2)*cos(v)
sage: f_z = sin(u)*(abs(cos(u)) + abs(sin(u)))^(-1)
sage: parametric_plot3d([f_x, f_y, f_z], (u,0,pi), (v,-pi,pi))
Graphics3d Object
```

Heart:

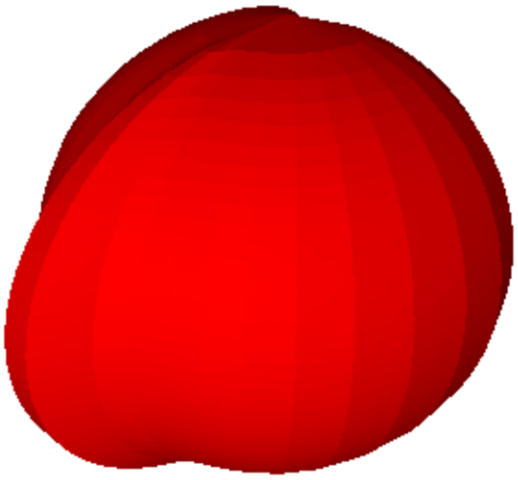
```
sage: u, v = var('u,v')
sage: f_x = cos(u)*(4*sqrt(1-v^2)*sin(abs(u))^abs(u))
```

(continues on next page)



(continued from previous page)

```
sage: f_y = sin(u)*(4*sqrt(1-v^2)*sin(abs(u))^abs(u))
sage: f_z = v
sage: parametric_plot3d([f_x, f_y, f_z], (u,-pi,pi), (v,-1,1), frame=False, color=
↪"red")
Graphics3d Object
```



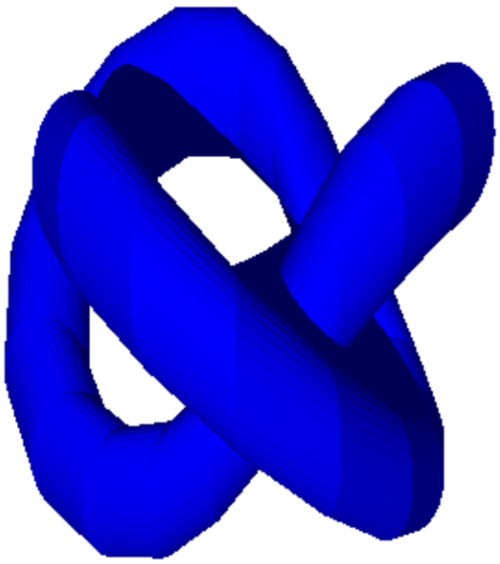
A Trefoil knot (Wikipedia article Trefoil_knot):

```
sage: u, v = var('u,v')
sage: f_x = (4*(1+0.25*sin(3*v))+cos(u))*cos(2*v)
sage: f_y = (4*(1+0.25*sin(3*v))+cos(u))*sin(2*v)
sage: f_z = sin(u)+2*cos(3*v)
sage: parametric_plot3d([f_x, f_y, f_z], (u,-pi,pi), (v,-pi,pi), frame=False, color=
↪"blue")
Graphics3d Object
```

Green bowtie:

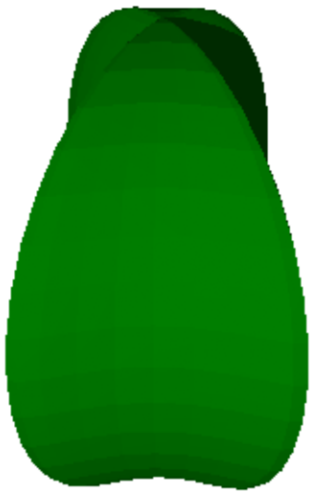
```
sage: u, v = var('u,v')
sage: f_x = sin(u) / (sqrt(2) + sin(v))
sage: f_y = sin(u) / (sqrt(2) + cos(v))
sage: f_z = cos(u) / (1 + sqrt(2))
```

(continues on next page)



(continued from previous page)

```
sage: parametric_plot3d([f_x, f_y, f_z], (u,-pi,pi), (v,-pi,pi), frame=False, color=
↪"green")
Graphics3d Object
```



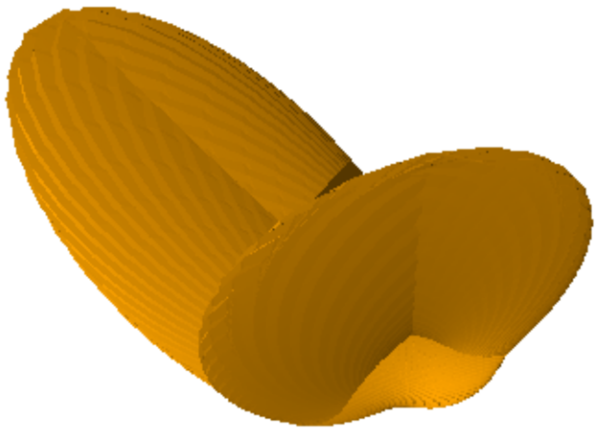
Boy's surface (Wikipedia article Boy%27s_surface and https://mathcurve.com/surfaces/boy/boy.shtml):

```
sage: u, v = var('u,v')
sage: K = cos(u) / (sqrt(2) - cos(2*u)*sin(3*v))
sage: f_x = K * (cos(u)*cos(2*v)+sqrt(2)*sin(u)*cos(v))
sage: f_y = K * (cos(u)*sin(2*v)-sqrt(2)*sin(u)*sin(v))
sage: f_z = 3 * K * cos(u)
sage: parametric_plot3d([f_x, f_y, f_z],      # long time
....:                   (u,-2*pi,2*pi),
....:                   (v,0,pi),
....:                   plot_points=[90,90],
....:                   frame=False,
....:                   color="orange")
Graphics3d Object
```

Maeder's Owl also known as Bour's minimal surface (Wikipedia article Bour%27s_minimal_surface):

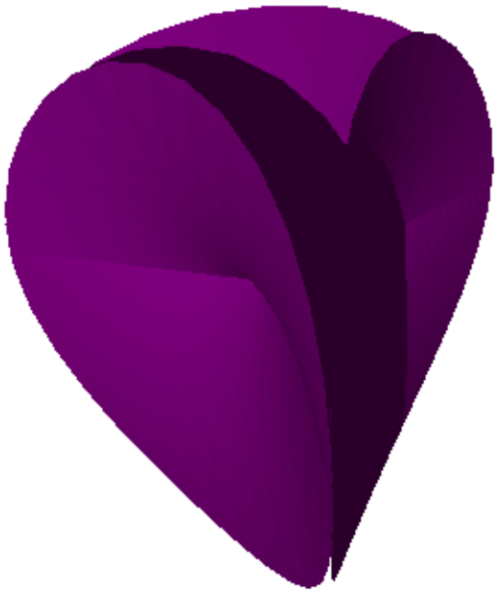
```
sage: u, v = var('u,v')
```

(continues on next page)



(continued from previous page)

```
sage: f_x = v*cos(u) - 0.5*v^2*cos(2*u)
sage: f_y = -v*sin(u) - 0.5*v^2*sin(2*u)
sage: f_z = 4 * v^1.5 * cos(3*u/2) / 3
sage: parametric_plot3d([f_x, f_y, f_z], (u,-2*pi,2*pi), (v,0,1),
....:                   plot_points=[90,90], frame=False, color="purple")
Graphics3d Object
```



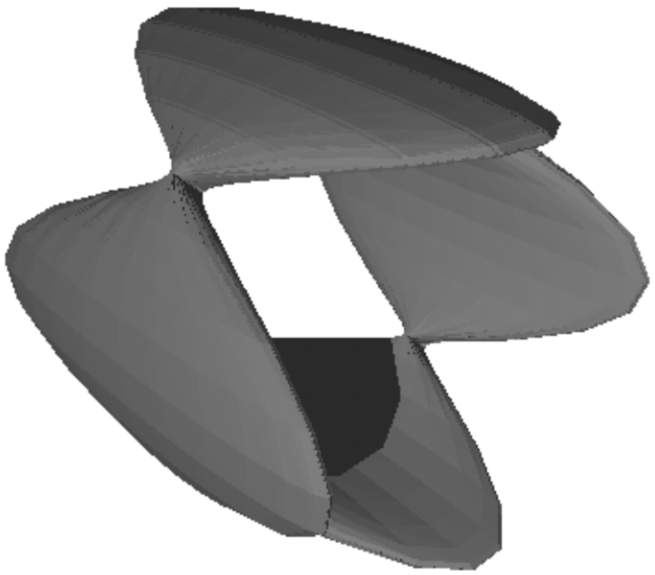
Bracelet:

```
sage: u, v = var('u,v')
sage: f_x = (2 + 0.2*sin(2*pi*u))*sin(pi*v)
sage: f_y = 0.2 * cos(2*pi*u) * 3 * cos(2*pi*v)
sage: f_z = (2 + 0.2*sin(2*pi*u))*cos(pi*v)
sage: parametric_plot3d([f_x, f_y, f_z], (u,0,pi/2), (v,0,3*pi/4), frame=False,␣
↪color="gray")
Graphics3d Object
```

Green goblet:

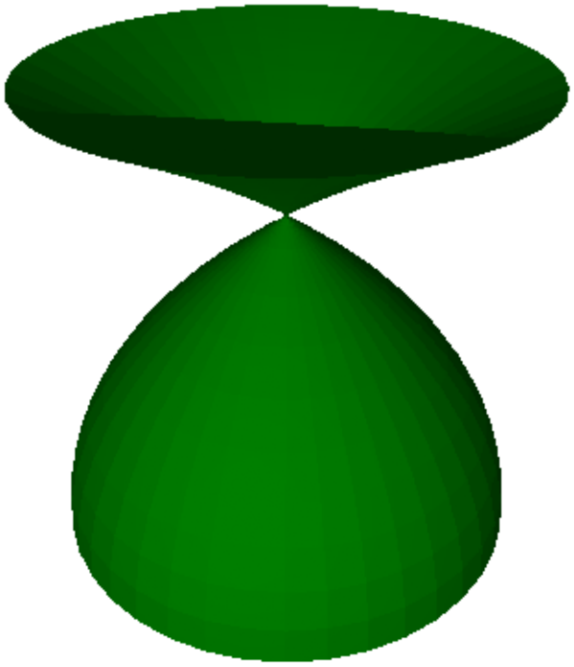
```
sage: u, v = var('u,v')
sage: f_x = cos(u) * cos(2*v)
sage: f_y = sin(u) * cos(2*v)
```

(continues on next page)



(continued from previous page)

```
sage: f_z = sin(v)
sage: parametric_plot3d([f_x, f_y, f_z], (u,0,2*pi), (v,0,pi), frame=False, color=
↪"green")
Graphics3d Object
```



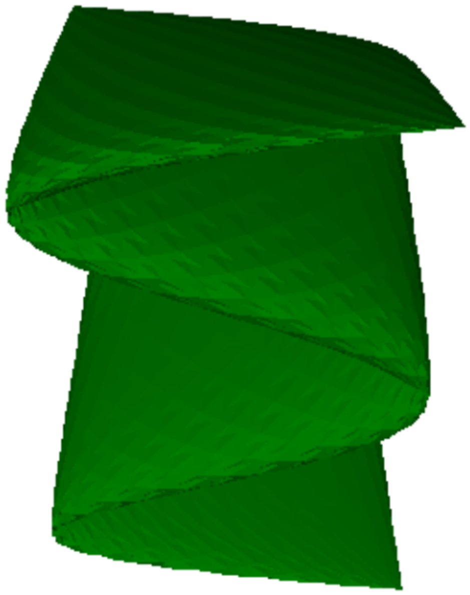
Funny folded surface - with square projection:

```
sage: u, v = var('u,v')
sage: f_x = cos(u) * sin(2*v)
sage: f_y = sin(u) * cos(2*v)
sage: f_z = sin(v)
sage: parametric_plot3d([f_x, f_y, f_z], (u,0,2*pi), (v,0,2*pi), frame=False, color=
↪"green")
Graphics3d Object
```

Surface of revolution of figure 8:

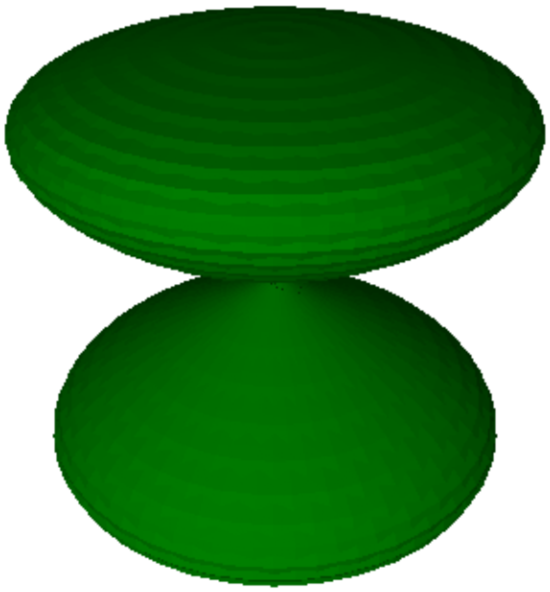
```
sage: u, v = var('u,v')
sage: f_x = cos(u) * sin(2*v)
sage: f_y = sin(u) * sin(2*v)
sage: f_z = sin(v)
sage: parametric_plot3d([f_x, f_y, f_z], (u,0,2*pi), (v,0,2*pi), frame=False, color=
```

(continues on next page)



(continued from previous page)

```
↪"green")
Graphics3d Object
```

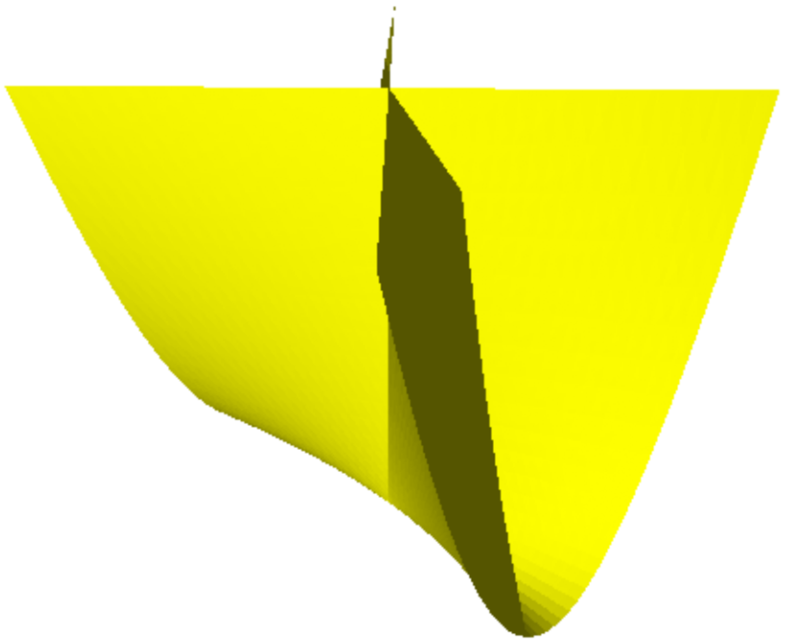


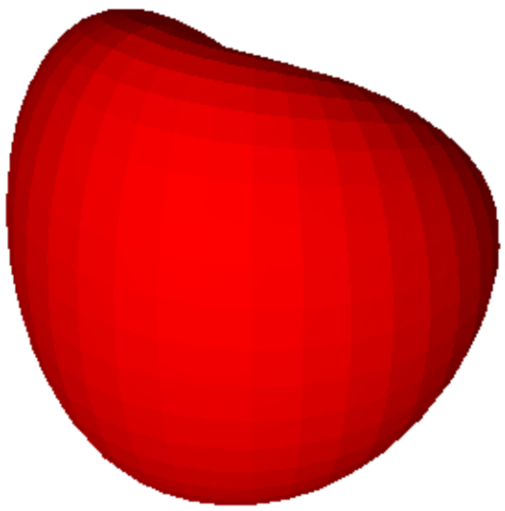
Yellow Whitney's umbrella (Wikipedia article Whitney_umbrella):

```
sage: u, v = var('u,v')
sage: f_x = u*v
sage: f_y = u
sage: f_z = v^2
sage: parametric_plot3d([f_x, f_y, f_z], (u,-1,1), (v,-1,1), frame=False, color=
↪"yellow")
Graphics3d Object
```

Cross cap (Wikipedia article Cross-cap):

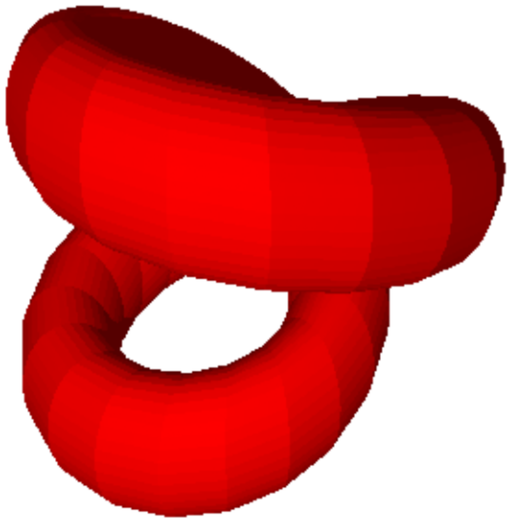
```
sage: u, v = var('u,v')
sage: f_x = (1+cos(v)) * cos(u)
sage: f_y = (1+cos(v)) * sin(u)
sage: f_z = -tanh((2/3)*(u-pi)) * sin(v)
sage: parametric_plot3d([f_x, f_y, f_z], (u,0,2*pi), (v,0,2*pi), frame=False, color=
↪"red")
Graphics3d Object
```





Twisted torus:

```
sage: u, v = var('u,v')
sage: f_x = (3+sin(v)+cos(u)) * cos(2*v)
sage: f_y = (3+sin(v)+cos(u)) * sin(2*v)
sage: f_z = sin(u) + 2*cos(v)
sage: parametric_plot3d([f_x, f_y, f_z], (u,0,2*pi), (v,0,2*pi), frame=False, color=
↪"red")
Graphics3d Object
```



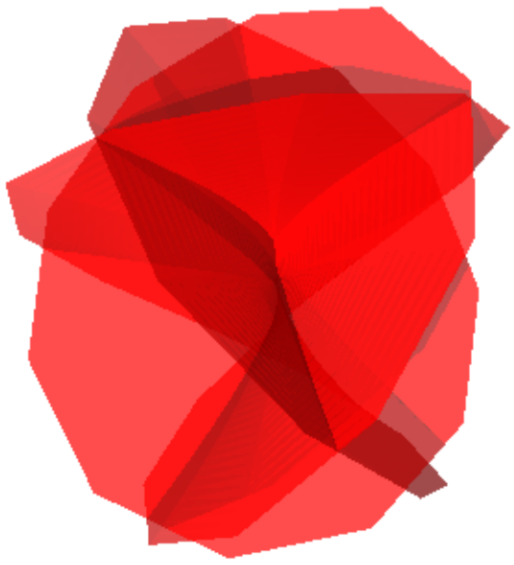
Four intersecting discs:

```
sage: u, v = var('u,v')
sage: f_x = v*cos(u) - 0.5*v^2*cos(2*u)
sage: f_y = -v*sin(u) - 0.5*v^2*sin(2*u)
sage: f_z = 4 * v^1.5 * cos(3*u/2) / 3
sage: parametric_plot3d([f_x, f_y, f_z], (u,0,4*pi), (v,0,2*pi), frame=False, color=
↪"red", opacity=0.7)
Graphics3d Object
```

Steiner surface/Roman's surface (see Wikipedia article Roman_surface and Wikipedia article Steiner_surface):

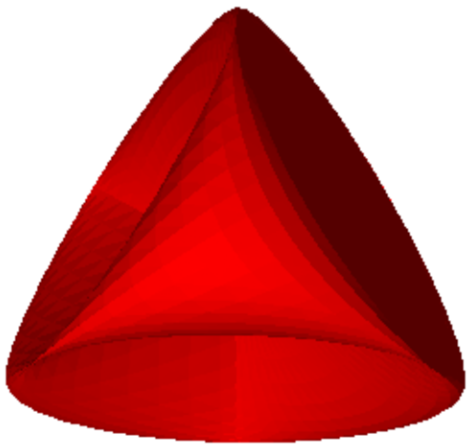
```
sage: u, v = var('u,v')
sage: f_x = (sin(2*u) * cos(v) * cos(v))
```

(continues on next page)



(continued from previous page)

```
sage: f_y = (sin(u) * sin(2*v))
sage: f_z = (cos(u) * sin(2*v))
sage: parametric_plot3d([f_x, f_y, f_z], (u,-pi/2,pi/2), (v,-pi/2,pi/2),␣
↪frame=False, color="red")
Graphics3d Object
```



Klein bottle? (see Wikipedia article Klein_bottle):

```
sage: u, v = var('u,v')
sage: f_x = (3*(1+sin(v)) + 2*(1-cos(v)/2)*cos(u)) * cos(v)
sage: f_y = (4+2*(1-cos(v)/2)*cos(u)) * sin(v)
sage: f_z = -2 * (1-cos(v)/2) * sin(u)
sage: parametric_plot3d([f_x, f_y, f_z], (u,0,2*pi), (v,0,2*pi), frame=False, color=
↪"green")
Graphics3d Object
```

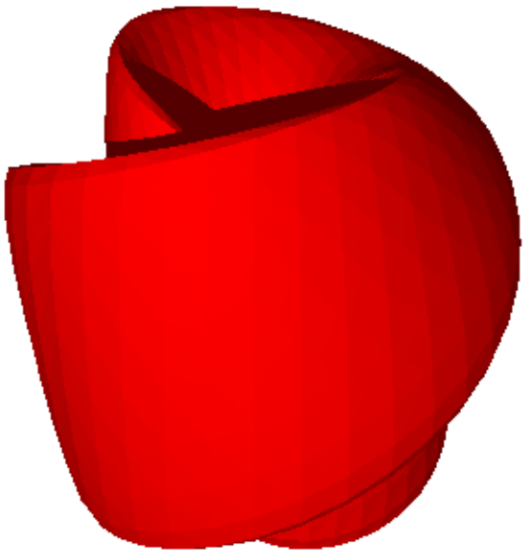
A Figure 8 embedding of the Klein bottle (see Wikipedia article Klein_bottle):

```
sage: u, v = var('u,v')
sage: f_x = (2+cos(v/2)*sin(u)-sin(v/2)*sin(2*u)) * cos(v)
sage: f_y = (2+cos(v/2)*sin(u)-sin(v/2)*sin(2*u)) * sin(v)
sage: f_z = sin(v/2)*sin(u) + cos(v/2)*sin(2*u)
```

(continues on next page)

(continued from previous page)

```
sage: parametric_plot3d([f_x, f_y, f_z], (u,0,2*pi), (v,0,2*pi), frame=False, color=
↪"red")
Graphics3d Object
```



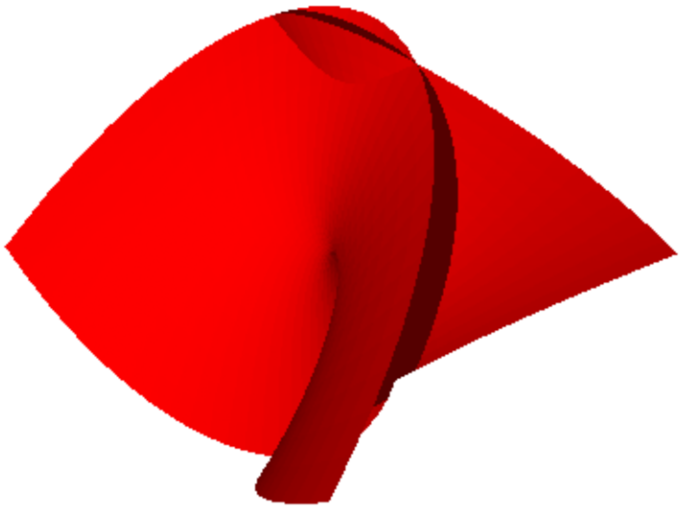
Enneper's surface (see Wikipedia article Enneper_surface):

```
sage: u, v = var('u,v')
sage: f_x = u - u^3/3 + u*v^2
sage: f_y = v - v^3/3 + v*u^2
sage: f_z = u^2 - v^2
sage: parametric_plot3d([f_x, f_y, f_z], (u,-2,2), (v,-2,2), frame=False, color="red
↪")
Graphics3d Object
```

Henneberg's surface (see http://xahlee.org/surface/gallery_m.html):

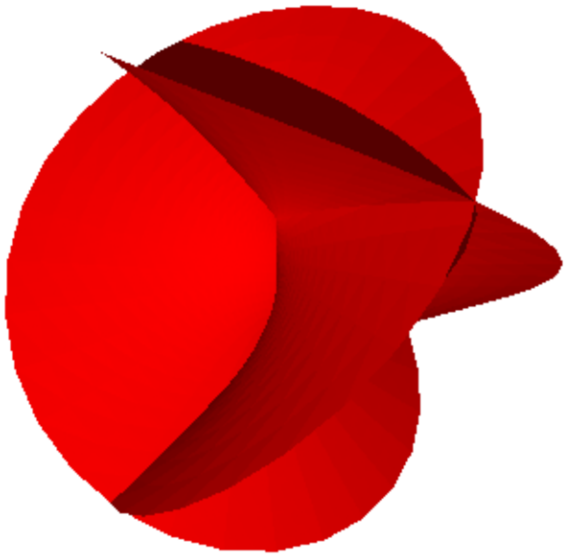
```
sage: u, v = var('u,v')
sage: f_x = 2*sinh(u)*cos(v) - (2/3)*sinh(3*u)*cos(3*v)
sage: f_y = 2*sinh(u)*sin(v) + (2/3)*sinh(3*u)*sin(3*v)
sage: f_z = 2 * cosh(2*u) * cos(2*v)
sage: parametric_plot3d([f_x, f_y, f_z], (u,-1,1), (v,-pi/2,pi/2), frame=False,␣
↪color="red")
```

(continues on next page)



(continued from previous page)

```
Graphics3d Object
```



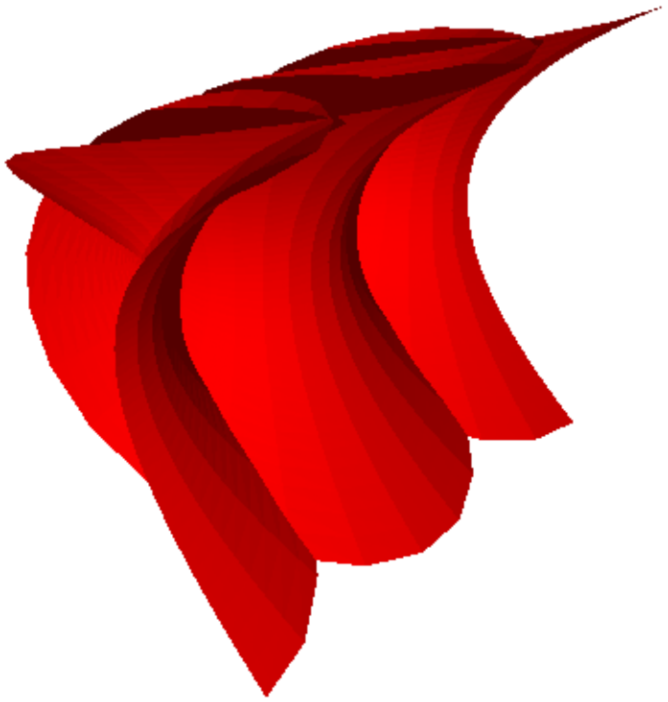
Dini's spiral:

```
sage: u, v = var('u,v')
sage: f_x = cos(u) * sin(v)
sage: f_y = sin(u) * sin(v)
sage: f_z = (cos(v)+log(tan(v/2))) + 0.2*u
sage: parametric_plot3d([f_x, f_y, f_z], (u,0,12.4), (v,0.1,2), frame=False, color=
↪"red")
Graphics3d Object
```

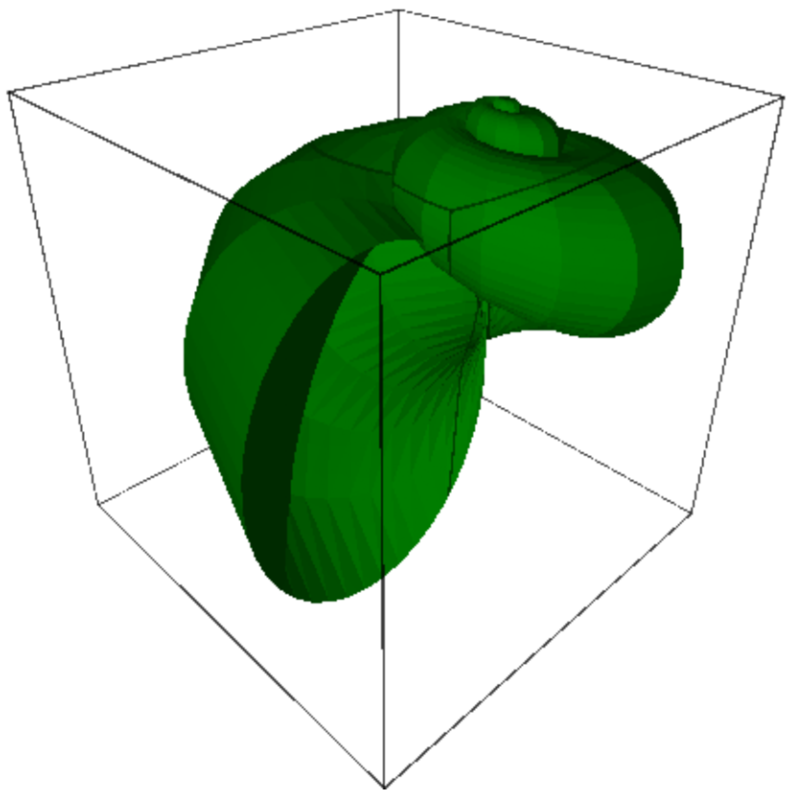
Catalan's surface (see http://xahlee.org/surface/catalan/catalan.html):

```
sage: u, v = var('u,v')
sage: f_x = u - sin(u)*cosh(v)
sage: f_y = 1 - cos(u)*cosh(v)
sage: f_z = 4 * sin(1/2*u) * sinh(v/2)
sage: parametric_plot3d([f_x, f_y, f_z], (u,-pi,3*pi), (v,-2,2), frame=False, color=
↪"red")
Graphics3d Object
```

A Conchoid:



```
sage: u, v = var('u,v')
sage: k = 1.2; k_2 = 1.2; a = 1.5
sage: f = (k^u*(1+cos(v))*cos(u), k^u*(1+cos(v))*sin(u), k^u*sin(v)-a*k_2^u)
sage: parametric_plot3d(f, (u,0,6*pi), (v,0,2*pi), plot_points=[40,40], texture=(0,
↪0.5,0))
Graphics3d Object
```



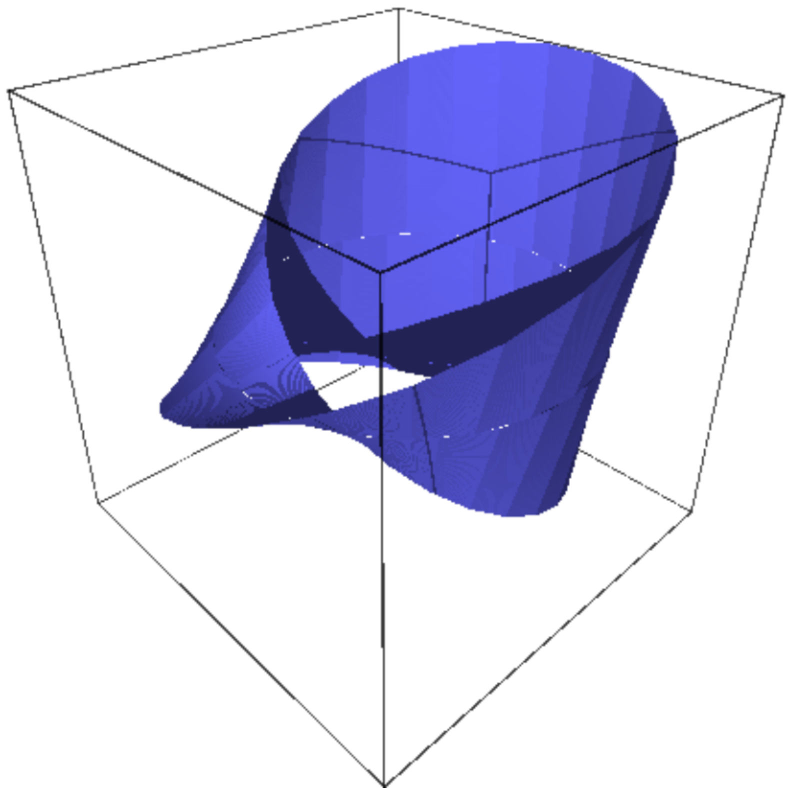
A Möbius strip:

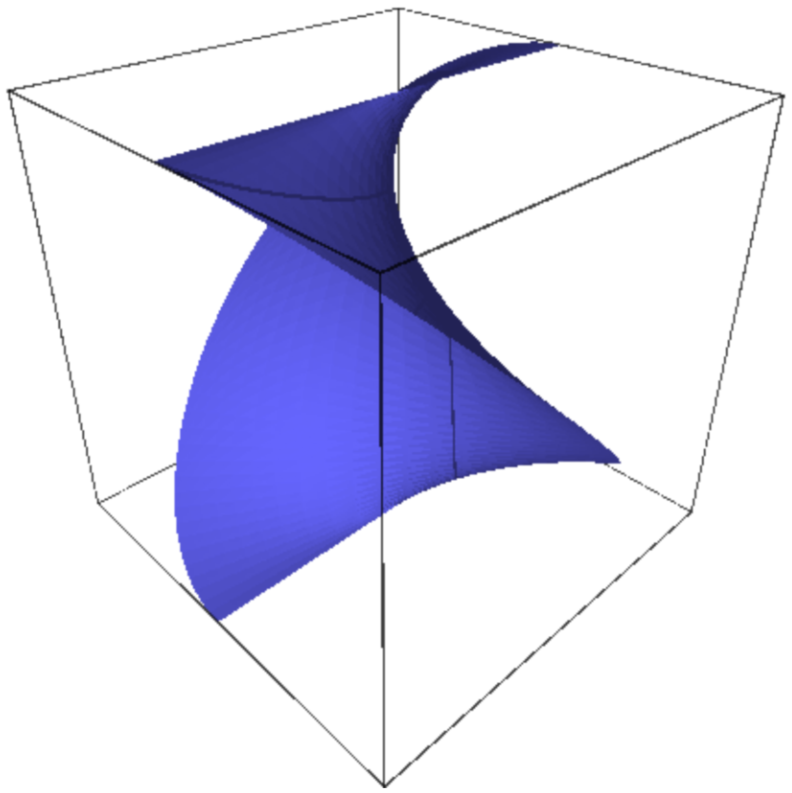
```
sage: u,v = var("u,v")
sage: parametric_plot3d([cos(u)*(1+v*cos(u/2)), sin(u)*(1+v*cos(u/2)), 0.2*v*sin(u/
↪2)],
....:                   (u,0, 4*pi+0.5), (v,0, 0.3), plot_points=[50,50])
Graphics3d Object
```

A Twisted Ribbon:

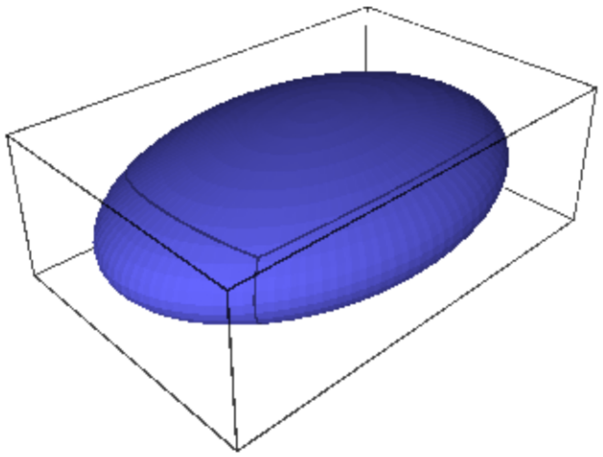
```
sage: u, v = var('u,v')
sage: parametric_plot3d([3*sin(u)*cos(v), 3*sin(u)*sin(v), cos(v)],
....:                   (u,0,2*pi), (v,0,pi), plot_points=[50,50])
Graphics3d Object
```

An Ellipsoid:





```
sage: u, v = var('u,v')
sage: parametric_plot3d([3*sin(u)*cos(v), 2*sin(u)*sin(v), cos(u)],
....:                   (u,0, 2*pi), (v, 0, 2*pi), plot_points=[50,50], aspect_
↪ratio=[1,1,1])
Graphics3d Object
```



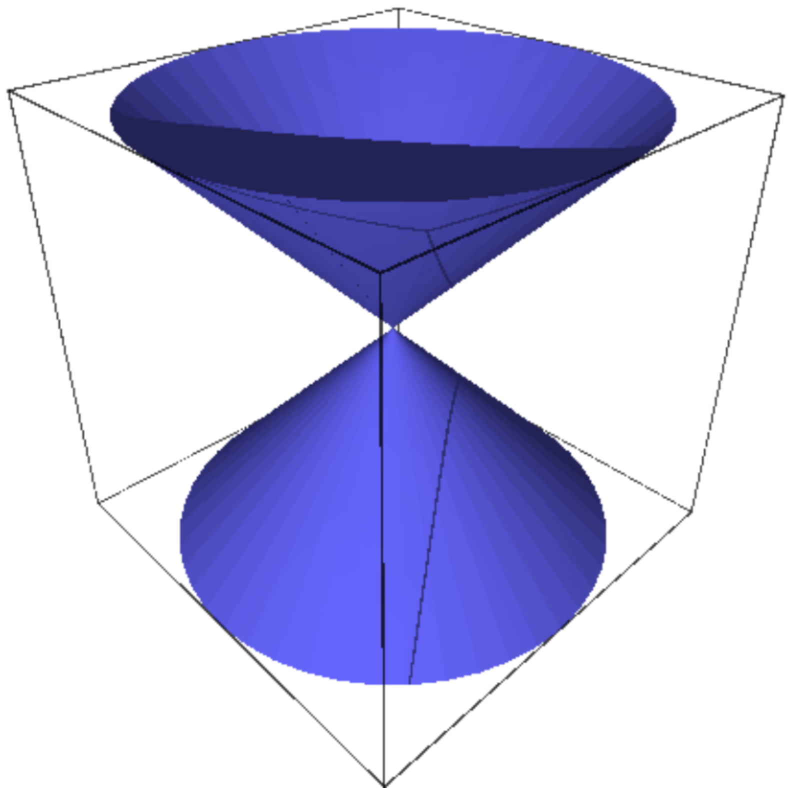
A Cone:

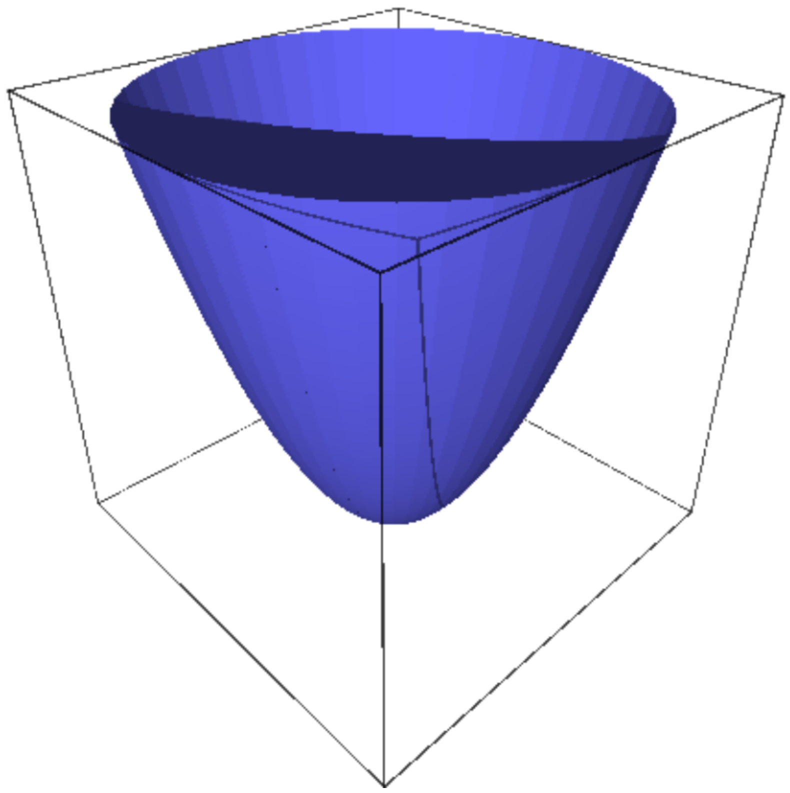
```
sage: u, v = var('u,v')
sage: parametric_plot3d([u*cos(v), u*sin(v), u], (u,-1,1), (v,0,2*pi+0.5), plot_
↪points=[50,50])
Graphics3d Object
```

A Paraboloid:

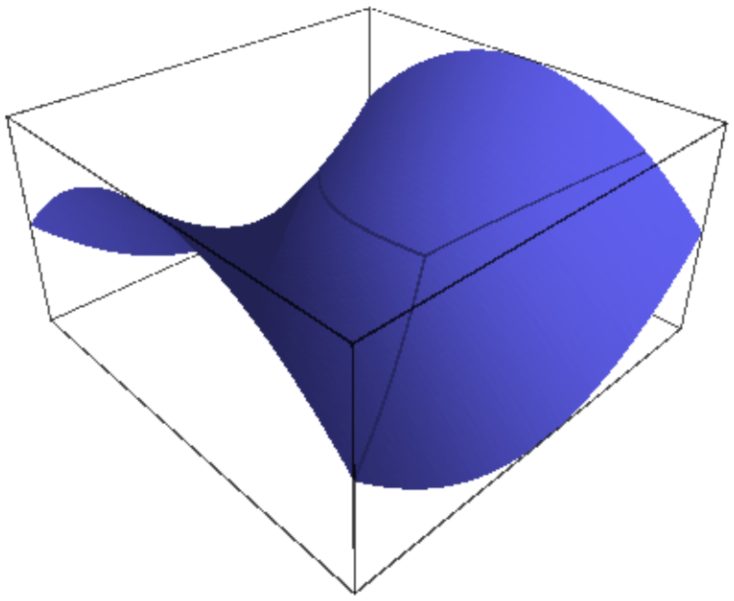
```
sage: u, v = var('u,v')
sage: parametric_plot3d([u*cos(v), u*sin(v), u^2], (u,0,1), (v,0,2*pi+0.4), plot_
↪points=[50,50])
Graphics3d Object
```

A Hyperboloid:





```
sage: u, v = var('u,v')
sage: plot3d(u^2-v^2, (u,-1,1), (v,-1,1), plot_points=[50,50])
Graphics3d Object
```

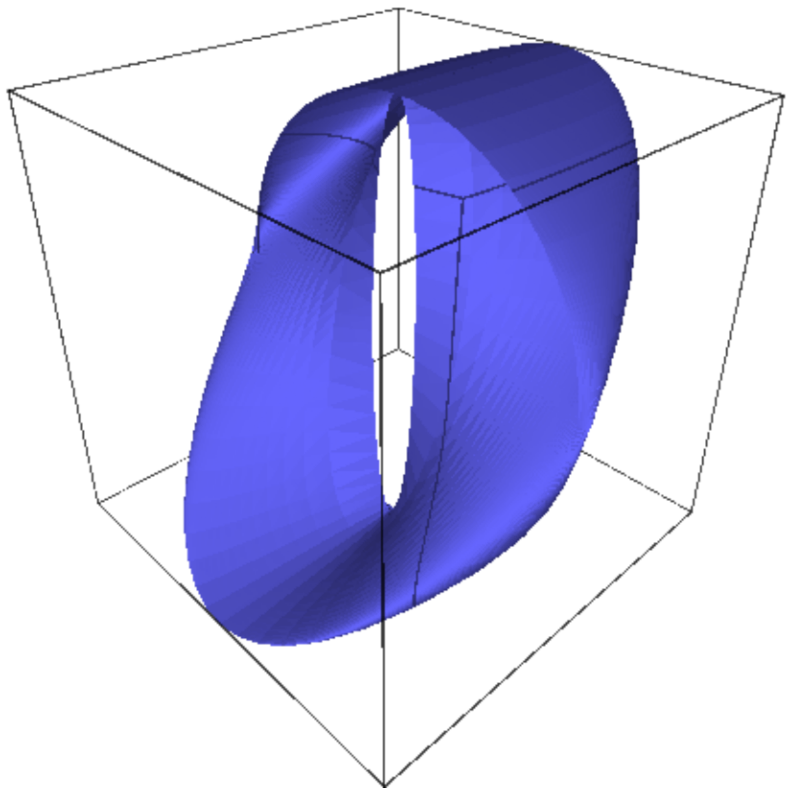


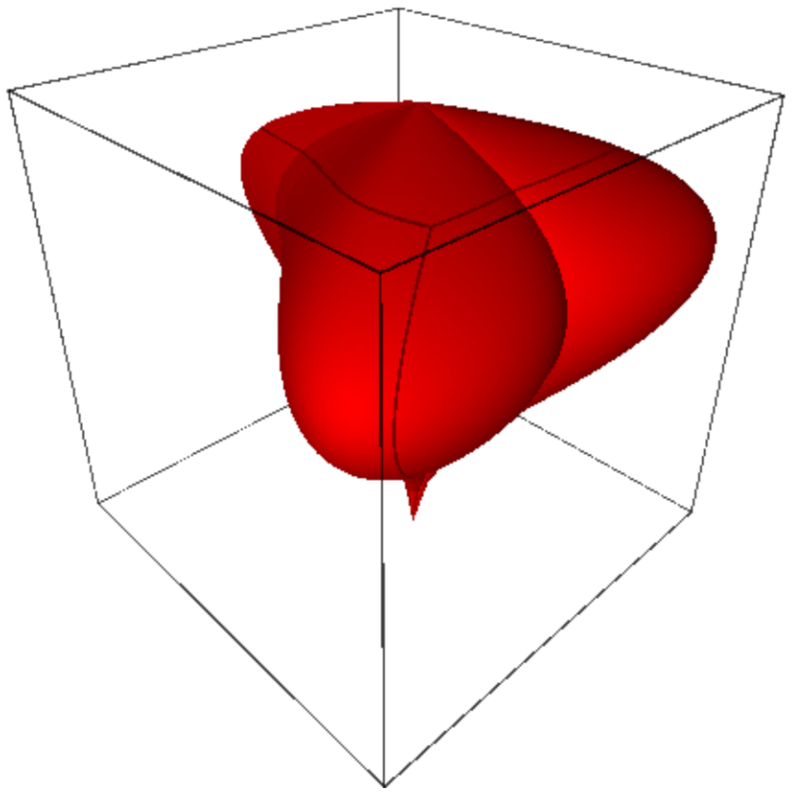
A weird looking surface - like a Möbius band but also an O:

```
sage: u, v = var('u,v')
sage: parametric_plot3d([sin(u)*cos(u)*log(u^2)*sin(v), (u^2)^(1/6)*(cos(u)^2)^(1/
↪4)*cos(v), sin(v)],
....:                   (u,0.001,1), (v,-pi,pi+0.2), plot_points=[50,50])
Graphics3d Object
```

A heart, but not a cardioid (for my wife):

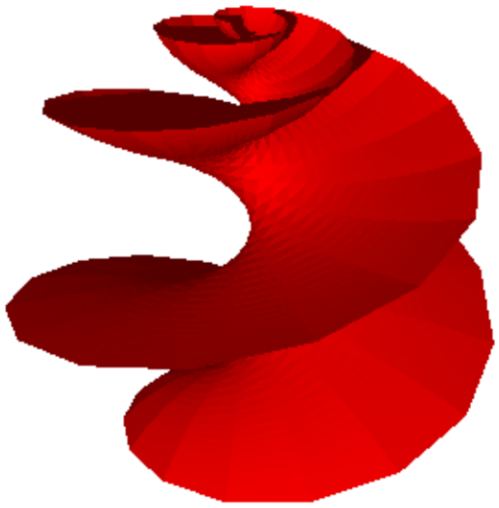
```
sage: u, v = var('u,v')
sage: p1 = parametric_plot3d([sin(u)*cos(u)*log(u^2)*v*(1-v)/2, ((u^6)^(1/
↪20)*(cos(u)^2)^(1/4)-1/2)*v*(1-v), v^(0.5)],
....:                         (u,0.001,1), (v,0,1), plot_points=[70,70], color='red')
sage: p2 = parametric_plot3d([-sin(u)*cos(u)*log(u^2)*v*(1-v)/2, ((u^6)^(1/
↪20)*(cos(u)^2)^(1/4)-1/2)*v*(1-v), v^(0.5)],
....:                         (u, 0.001,1), (v,0,1), plot_points=[70,70], color='red
↪')
sage: show(p1+p2)
```





A Hyperhelicoidal:

```
sage: u = var("u")
sage: v = var("v")
sage: f_x = (sinh(v)*cos(3*u)) / (1+cosh(u)*cosh(v))
sage: f_y = (sinh(v)*sin(3*u)) / (1+cosh(u)*cosh(v))
sage: f_z = (cosh(v)*sinh(u)) / (1+cosh(u)*cosh(v))
sage: parametric_plot3d([f_x, f_y, f_z], (u,-pi,pi), (v,-pi,pi), plot_points=[50,
↪50], frame=False, color="red")
Graphics3d Object
```

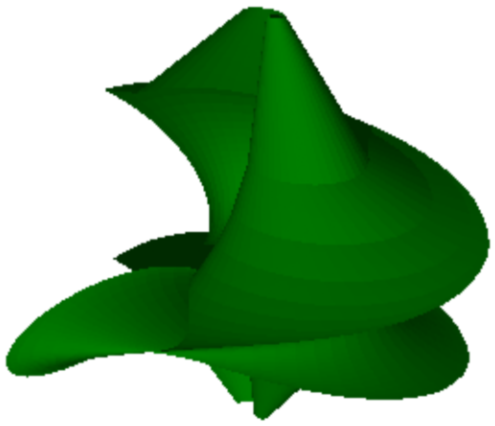


A Helicoid (lines through a helix, Wikipedia article Helix):

```
sage: u, v = var('u,v')
sage: f_x = sinh(v) * sin(u)
sage: f_y = -sinh(v) * cos(u)
sage: f_z = 3 * u
sage: parametric_plot3d([f_x, f_y, f_z], (u,-pi,pi), (v,-pi,pi), plot_points=[50,
↪50], frame=False, color="red")
Graphics3d Object
```

Kuen’s surface (http://virtualmathmuseum.org/Surface/kuen/kuen.html):

```
sage: f_x = (2*(cos(u) + u*sin(u))*sin(v))/(1+ u^2*sin(v)^2)
sage: f_y = (2*(sin(u) - u*cos(u))*sin(v))/(1+ u^2*sin(v)^2)
sage: f_z = log(tan(1/2 *v)) + (2*cos(v))/(1+ u^2*sin(v)^2)
sage: parametric_plot3d([f_x, f_y, f_z], (u,0,2*pi), (v,0.01,pi-0.01), plot_
↪points=[50,50], frame=False, color="green")
Graphics3d Object
```



A 5-pointed star:

```
sage: G1 = (abs(cos(u/4))^0.5+abs(sin(u/4))^0.5)^(-1/0.3)
sage: G2 = (abs(cos(5*v/4))^1.7+abs(sin(5*v/4))^1.7)^(-1/0.1)
sage: f_x = cos(u) * cos(v) * G1 * G2
sage: f_y = cos(u) * sin(v) * G1 * G2
sage: f_z = sin(u) * G1
sage: parametric_plot3d([f_x, f_y, f_z], (u,-pi/2,pi/2), (v,0,2*pi), plot_
↪points=[50,50], frame=False, color="green")
Graphics3d Object
```

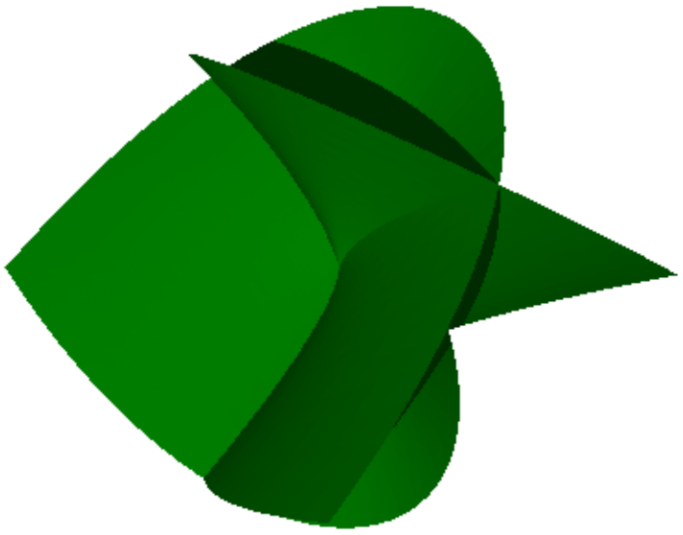
A cool self-intersecting surface (Eppener surface?):

```
sage: f_x = u - u^3/3 + u*v^2
sage: f_y = v - v^3/3 + v*u^2
sage: f_z = u^2 - v^2
```

(continues on next page)

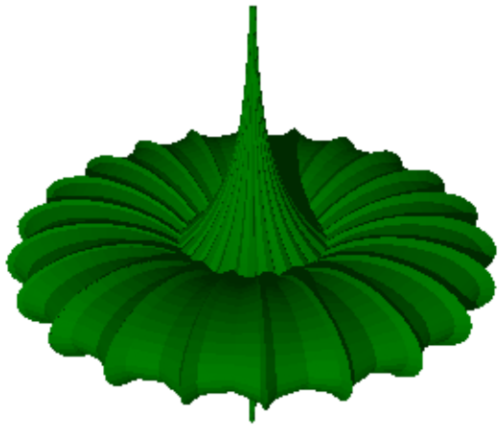
(continued from previous page)

```
sage: parametric_plot3d([f_x, f_y, f_z], (u,-25,25), (v,-25,25), plot_points=[50,
↪50], frame=False, color="green")
Graphics3d Object
```



The breather surface (Wikipedia article Breather_surface):

```
sage: K = sqrt(0.84)
sage: G = (0.4*((K*cosh(0.4*u))^2 + (0.4*sin(K*v))^2))
sage: f_x = (2*K*cosh(0.4*u)*(-(K*cos(v)*cos(K*v)) - sin(v)*sin(K*v)))/G
sage: f_y = (2*K*cosh(0.4*u)*(-(K*sin(v)*cos(K*v)) + cos(v)*sin(K*v)))/G
sage: f_z = -u + (2*0.84*cosh(0.4*u)*sinh(0.4*u))/G
sage: parametric_plot3d([f_x, f_y, f_z], (u,-13.2,13.2), (v,-37.4,37.4), plot_
↪points=[90,90], frame=False, color="green")
Graphics3d Object
```



2.3 Surfaces of revolution

AUTHORS:

- Oscar Gerardo Lazo Arjona (2010): initial version.

`sage.plot.plot3d.revolution_plot3d.`**`revolution_plot3d`**(*curve*, *trange*, *phirange=None*, *parallel_axis='z'*, *axis=(0, 0)*, *print_vector=False*, *show_curve=False*, ***kwds*)

Return a plot of a revolved curve.

There are three ways to call this function:

- `revolution_plot3d(f,trange)` where f is a function located in the xz plane.
- `revolution_plot3d((f_x,f_z),trange)` where (f_x, f_z) is a parametric curve on the xz plane.
- `revolution_plot3d((f_x,f_y,f_z),trange)` where (f_x, f_y, f_z) can be any parametric curve.

INPUT:

- `curve` – A curve to be revolved, specified as a function, a 2-tuple or a 3-tuple.
- `trange` – A 3-tuple $(t, t_{\min}, t_{\max})$ where t is the independent variable of the curve.
- `phirange` – A 2-tuple of the form $(\phi_{\min}, \phi_{\max})$, (default $(0, \pi)$) that specifies the angle in which the curve is to be revolved.
- `parallel_axis` – A string (one of `'x'`, `'y'`, `'z'`) that specifies the coordinate axis parallel to the revolution axis.
- `axis` – A 2-tuple that specifies the position of the revolution axis. If `parallel_axis` is:
  - `'z'` – then `axis` is the point in which the revolution axis intersects the x y plane.
  - `'x'` – then `axis` is the point in which the revolution axis intersects the y z plane.
  - `'y'` – then `axis` is the point in which the revolution axis intersects the x z plane.
- `print_vector` – If `True`, the parametrization of the surface of revolution will be printed.
- `show_curve` – If `True`, the curve will be displayed.

EXAMPLES:

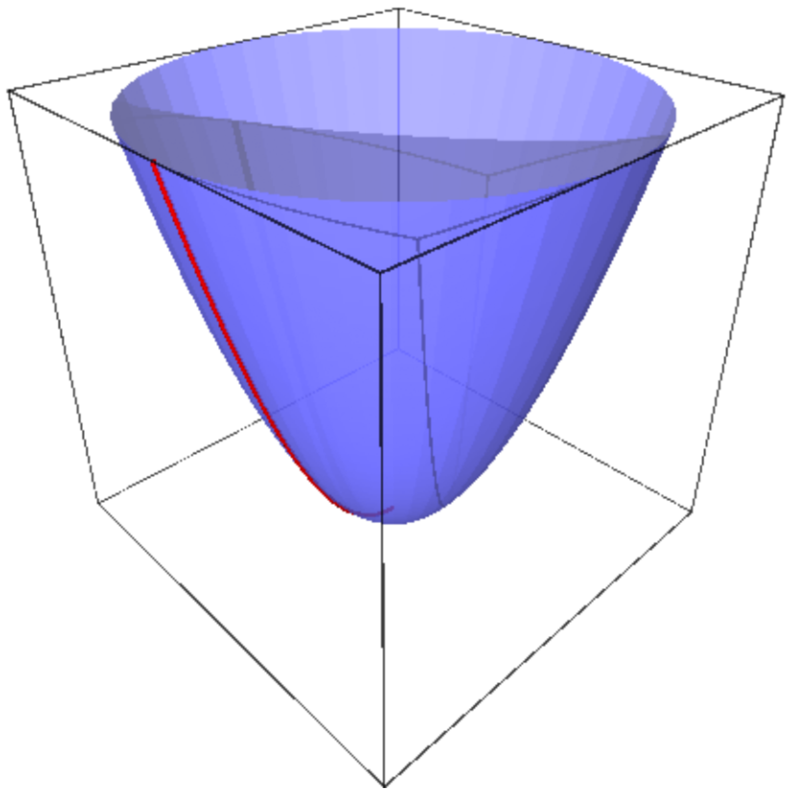
Let's revolve a simple function around different axes:

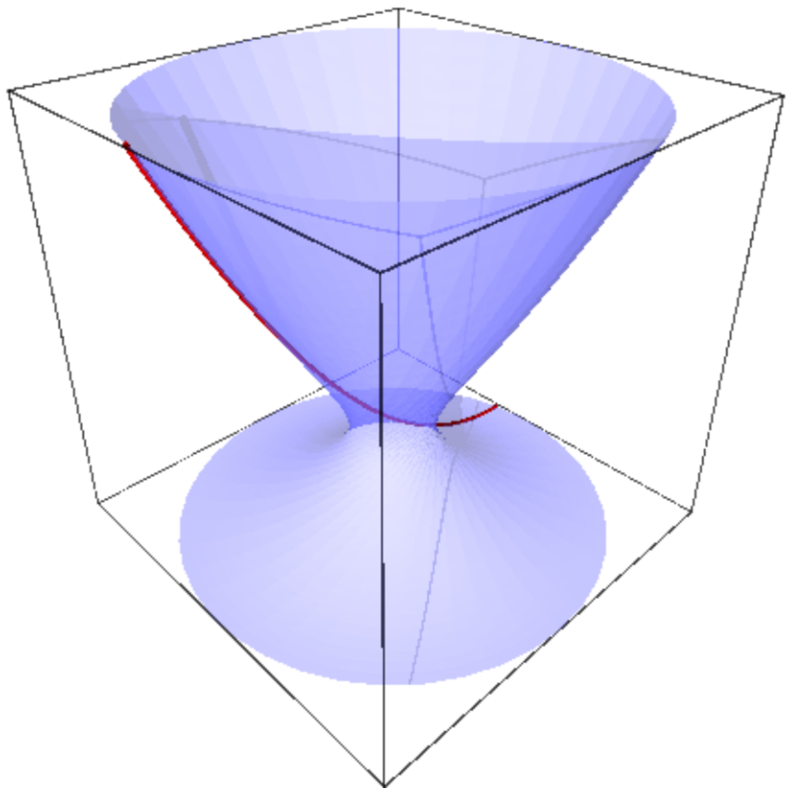
```
sage: u = var('u')
sage: f = u^2
sage: revolution_plot3d(f, (u,0,2),
....:                   show_curve=True, opacity=0.7).show(aspect_ratio=(1,1,1))
```

If we move slightly the axis, we get a goblet-like surface:

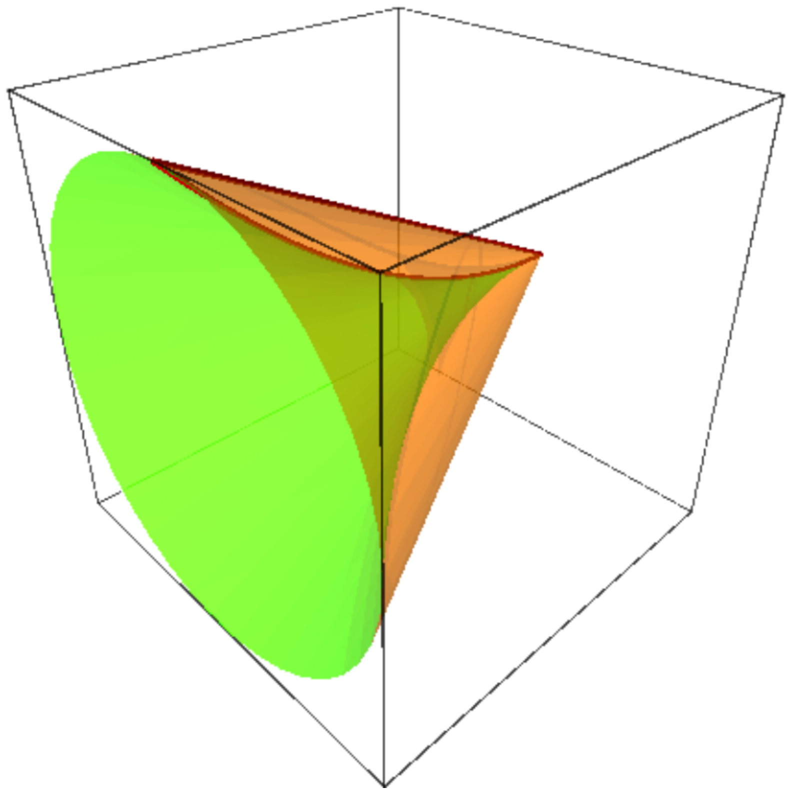
```
sage: revolution_plot3d(f, (u,0,2), axis=(1,0.2),
....:                   show_curve=True, opacity=0.5).show(aspect_ratio=(1,1,1))
```

A common problem in calculus books, find the volume within the following revolution solid:





```
sage: line = u
sage: parabola = u^2
sage: sur1 = revolution_plot3d(line, (u,0,1), opacity=0.5, rgbcolor=(1,0.5,0),
....:                          show_curve=True, parallel_axis='x')
sage: sur2 = revolution_plot3d(parabola, (u,0,1), opacity=0.5, rgbcolor=(0,1,0),
....:                          show_curve=True, parallel_axis='x')
sage: (sur1 + sur2).show()
```



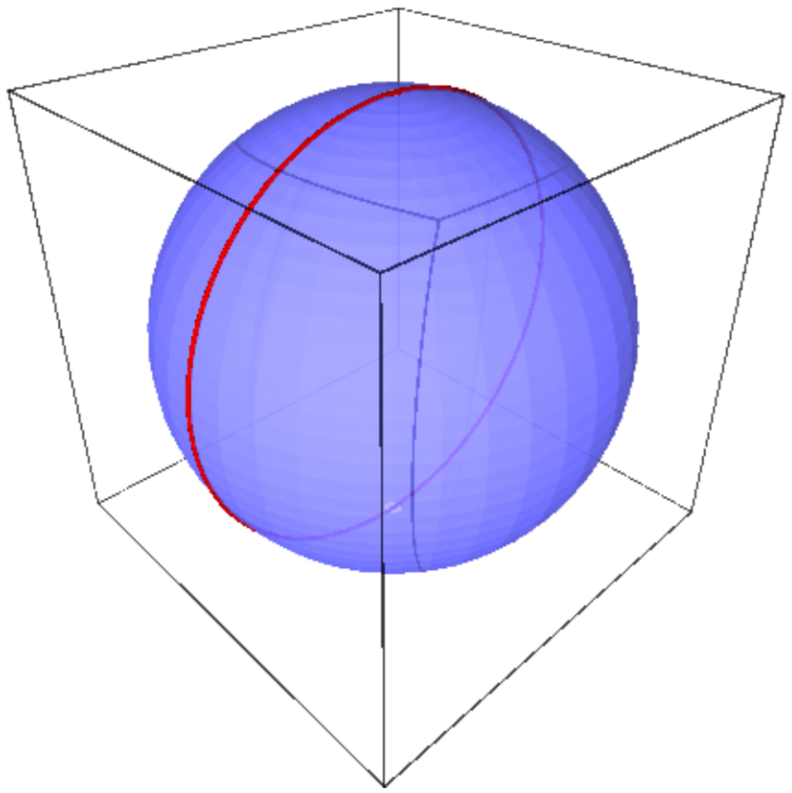
Now let's revolve a parametrically defined circle. We can play with the topology of the surface by changing the axis, an axis in $(0, 0)$ (as the previous one) will produce a sphere-like surface:

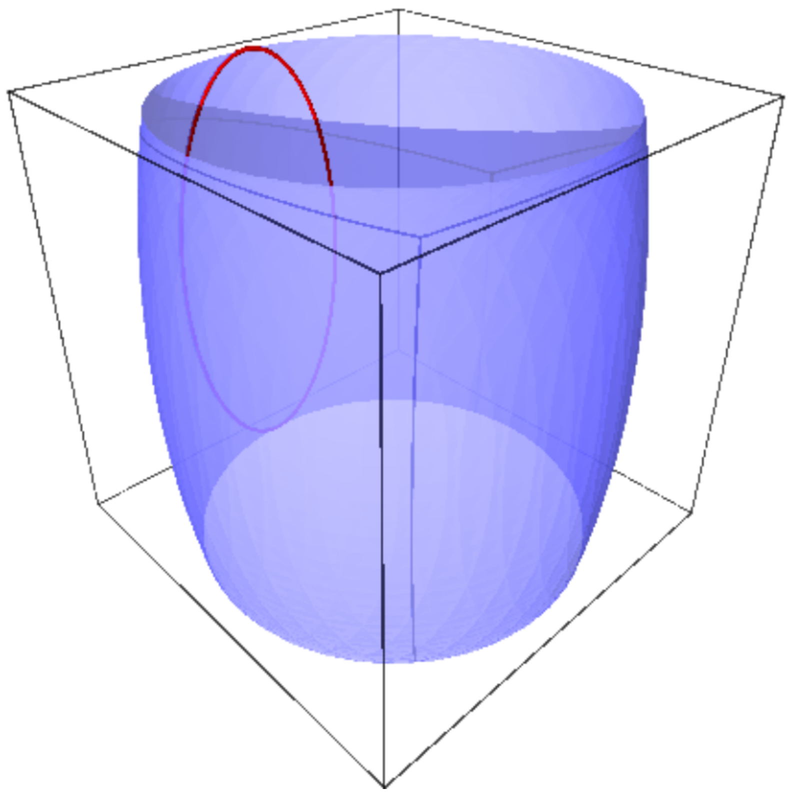
```
sage: u = var('u')
sage: circle = (cos(u), sin(u))
sage: revolution_plot3d(circle, (u,0,2*pi), axis=(0,0),
....:                   show_curve=True, opacity=0.5).show(aspect_ratio=(1,1,1))
```

An axis on $(0, y)$ will produce a cylinder-like surface:

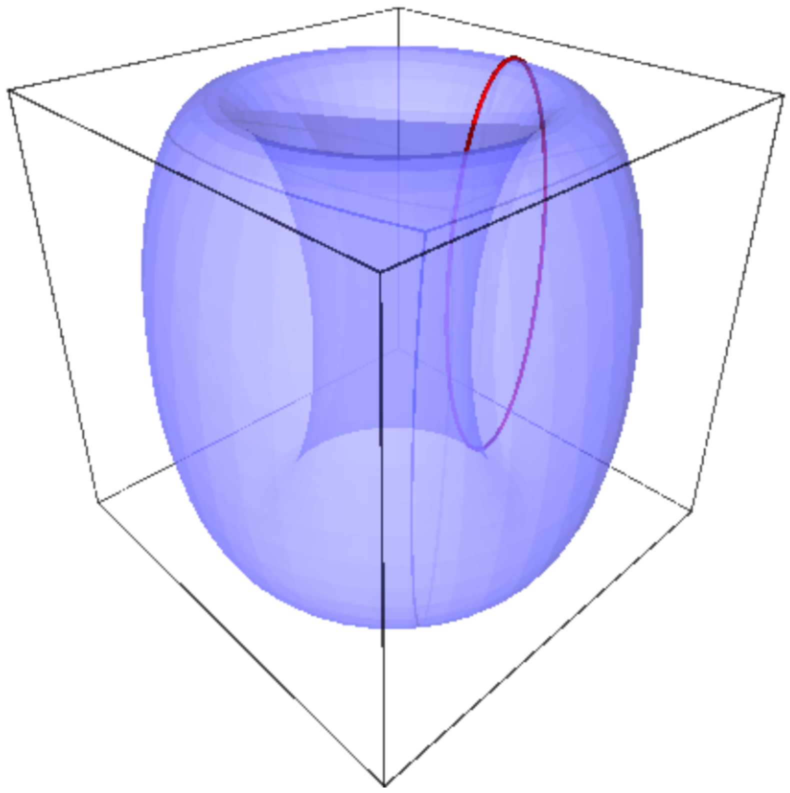
```
sage: revolution_plot3d(circle, (u,0,2*pi), axis=(0,2),
....:                   show_curve=True, opacity=0.5).show(aspect_ratio=(1,1,1))
```

And any other axis will produce a torus-like surface:





```
sage: revolution_plot3d(circle, (u,0,2*pi), axis=(2,0),
....:                   show_curve=True, opacity=0.5).show(aspect_ratio=(1,1,1))
```



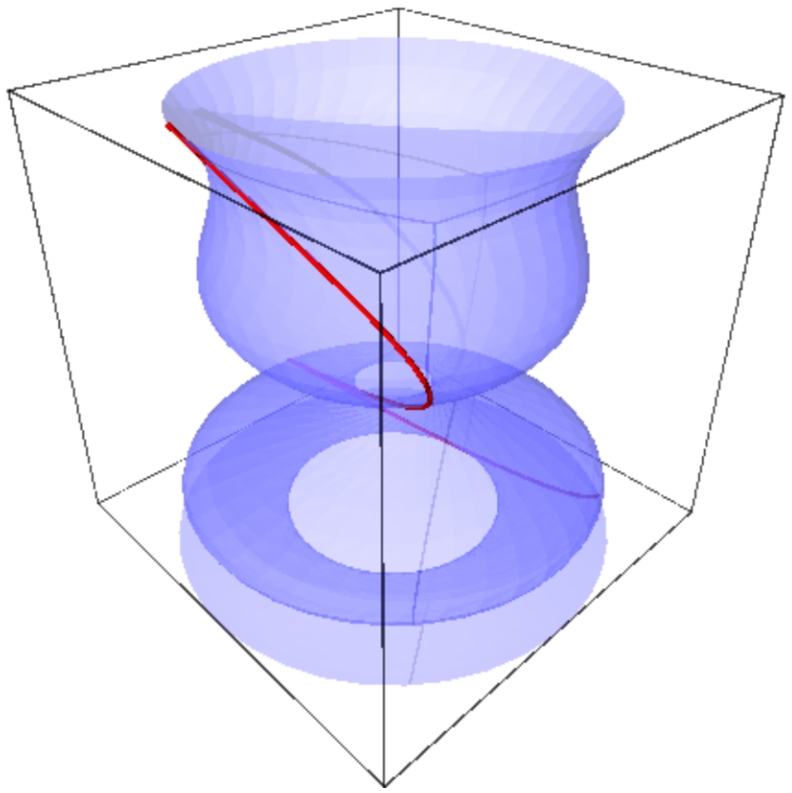
Now, we can get another goblet-like surface by revolving a curve in 3d:

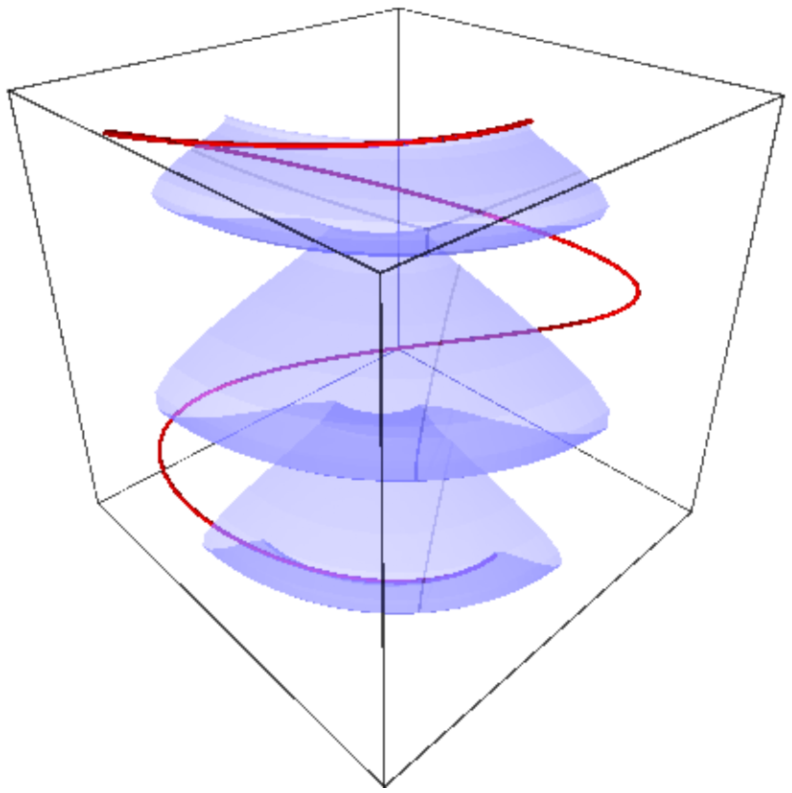
```
sage: u = var('u')
sage: curve = (u, cos(4*u), u^2)
sage: P = revolution_plot3d(curve, (u,0,2), parallel_axis='z', axis=(1,.2),
....:                       show_curve=True, opacity=0.5)
sage: P.show(aspect_ratio=(1,1,1))
```

A curvy curve with only a quarter turn:

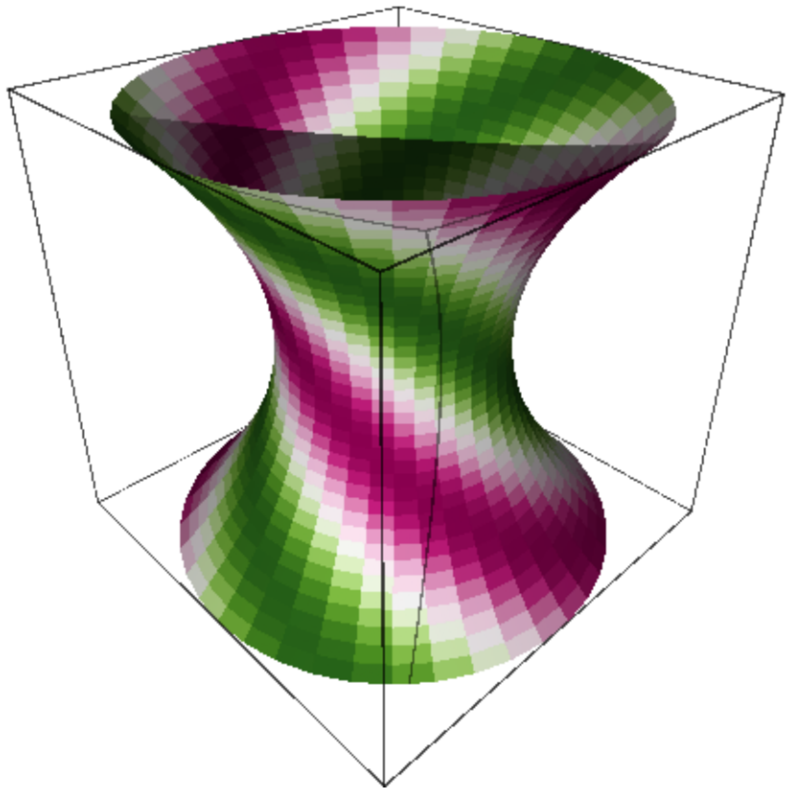
```
sage: u = var('u')
sage: curve = (sin(3*u), .8*cos(4*u), cos(u))
sage: revolution_plot3d(curve, (u,0,pi), (0,pi/2), parallel_axis='z',
....:                   show_curve=True, opacity=0.5).show(aspect_ratio=(1,1,1),
....:                                                      frame=False)
```

One can also color the surface using a coloring function of two parameters and a colormap as follows. Note that the coloring function must take values in the interval $[0, 1]$.





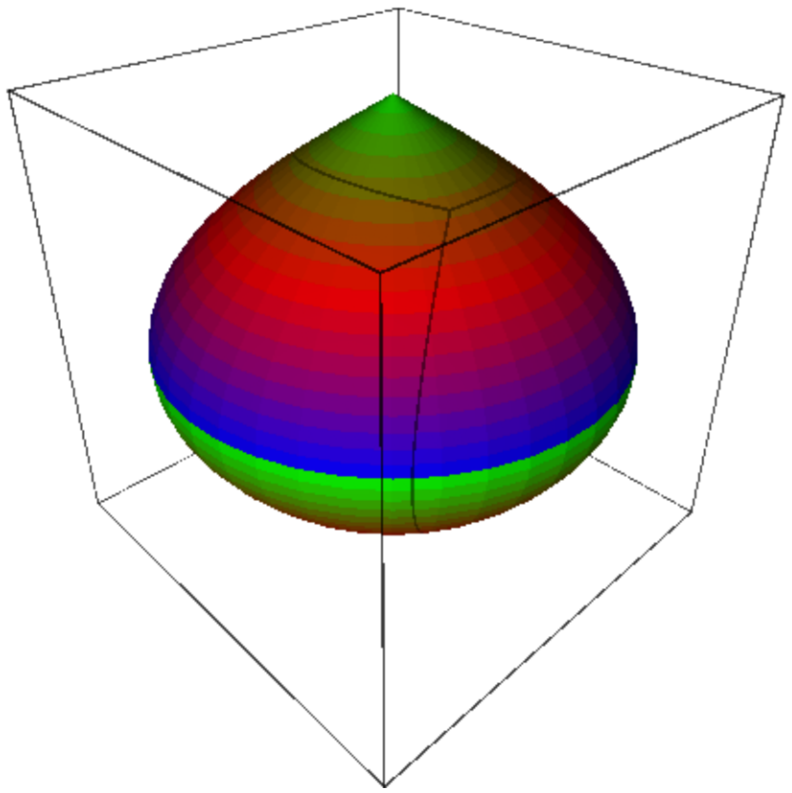
```
sage: u, phi = var('u,phi')
sage: def cf(u,phi): return sin(phi+u) ^ 2
sage: curve = (1+u^2/4, 0, u)
sage: revolution_plot3d(curve, (u,-2,2), (0,2*pi), parallel_axis='z',
....:                   color=(cf, colormaps.PiYG)).show(aspect_ratio=(1,1,1))
```



The first parameter of the coloring function will be identified with the parameter of the curve, and the second with the angle parameter.

Another colored example, illustrating that one can use (colormap, color function) instead of (color function, colormap):

```
sage: u, phi = var('u,phi')
sage: def cf(u, phi): return float(2 * u / pi) % 1
sage: curve = (sin(u), 0, u)
sage: revolution_plot3d(curve, (u,0,pi), (0,2*pi), parallel_axis='z',
....:                   color=(colormaps.brg, cf)).show(aspect_ratio=1)
```



2.4 Plotting 3D fields

sage.plot.plot3d.plot_field3d.**plot_vector_field3d**(*functions*, *xrange*, *yrange*, *zrange*, *plot_points=5*, *colors='jet'*, *center_arrows=False*, ***kwds*)

Plot a 3d vector field

INPUT:

- `functions` – a list of three functions, representing the x-, y-, and z-coordinates of a vector
- `xrange`, `yrange`, and `zrange` – three tuples of the form (var, start, stop), giving the variables and ranges for each axis
- `plot_points` – (default 5) either a number or list of three numbers, specifying how many points to plot for each axis
- `colors` – (default `'jet'`) a color, list of colors (which are interpolated between), or matplotlib colormap name, giving the coloring of the arrows. If a list of colors or a colormap is given, coloring is done as a function of length of the vector
- `center_arrows` – (default `False`) If `True`, draw the arrows centered on the points; otherwise, draw the arrows with the tail at the point
- any other keywords are passed on to the `plot()` command for each arrow

EXAMPLES:

A 3d vector field:

```
sage: x,y,z = var('x y z')
sage: plot_vector_field3d((x*cos(z), -y*cos(z), sin(z)),
....:                     (x,0,pi), (y,0,pi), (z,0,pi))
Graphics3d Object
```

same example with only a list of colors:

```
sage: plot_vector_field3d((x*cos(z), -y*cos(z), sin(z)),
....:                     (x,0,pi), (y,0,pi), (z,0,pi),
....:                     colors=['red','green','blue'])
Graphics3d Object
```

same example with only one color:

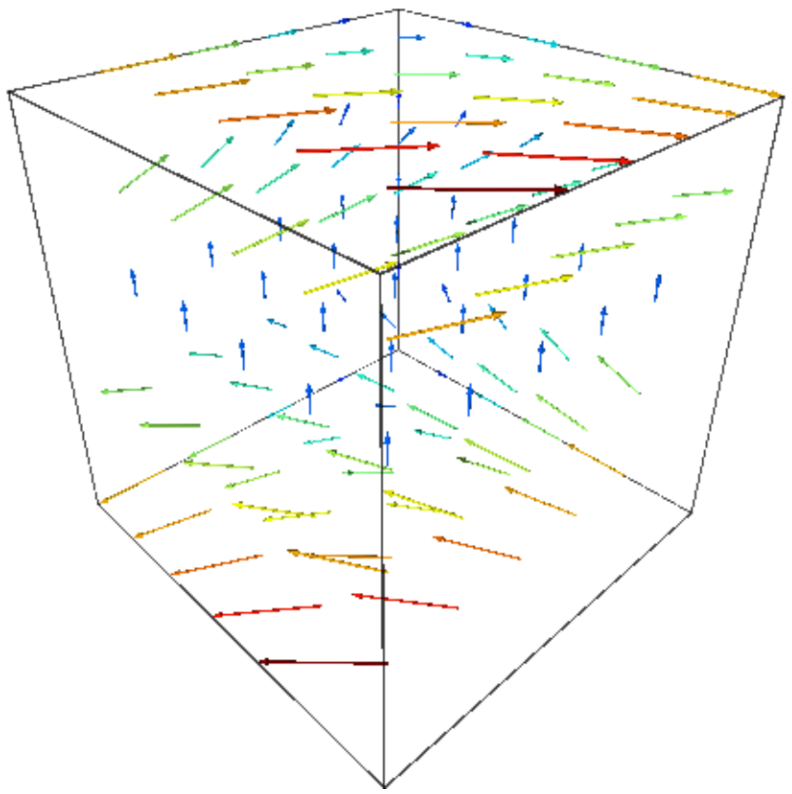
```
sage: plot_vector_field3d((x*cos(z), -y*cos(z), sin(z)),
....:                     (x,0,pi), (y,0,pi), (z,0,pi), colors='red')
Graphics3d Object
```

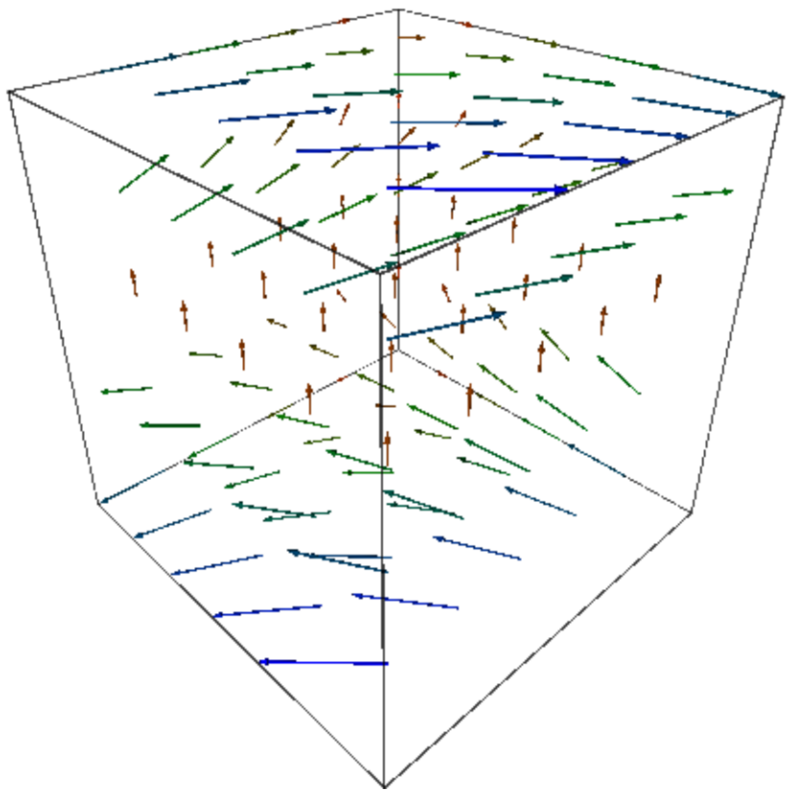
same example with the same plot points for the three axes:

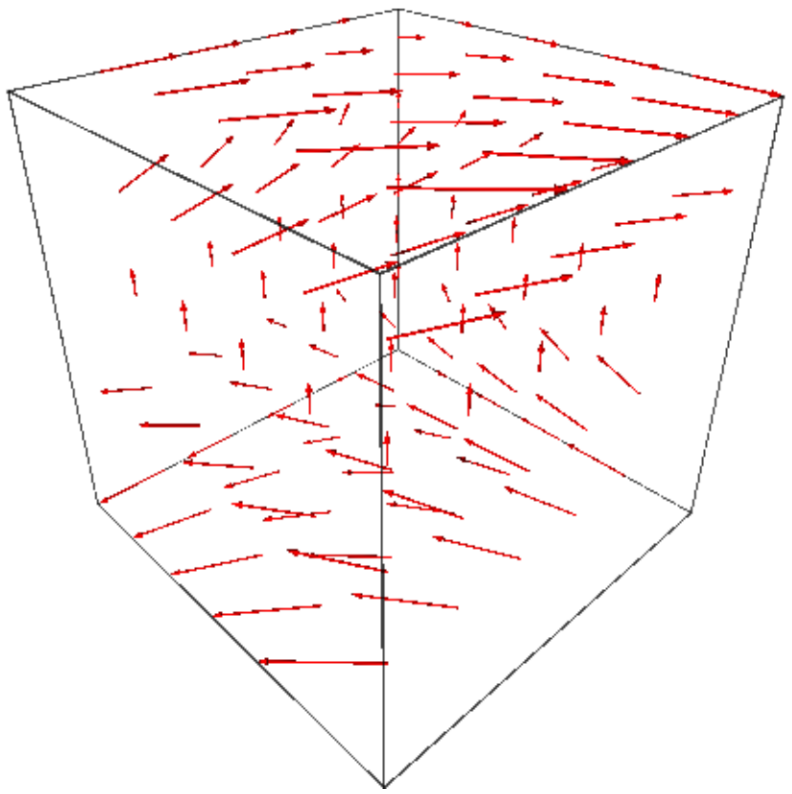
```
sage: plot_vector_field3d((x*cos(z), -y*cos(z), sin(z)),
....:                     (x,0,pi), (y,0,pi), (z,0,pi), plot_points=4)
Graphics3d Object
```

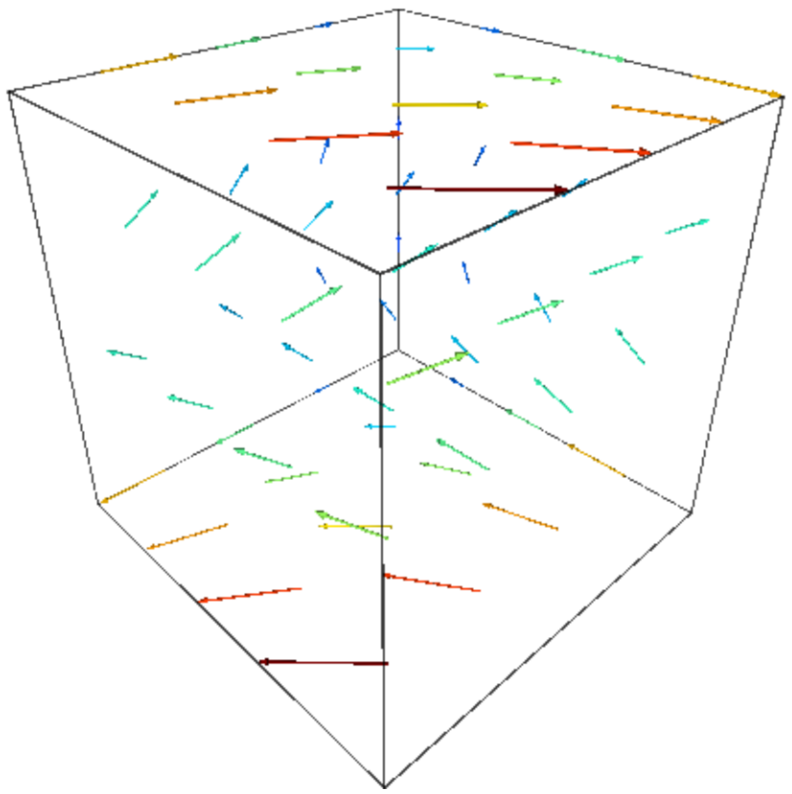
same example with different number of plot points for each axis:

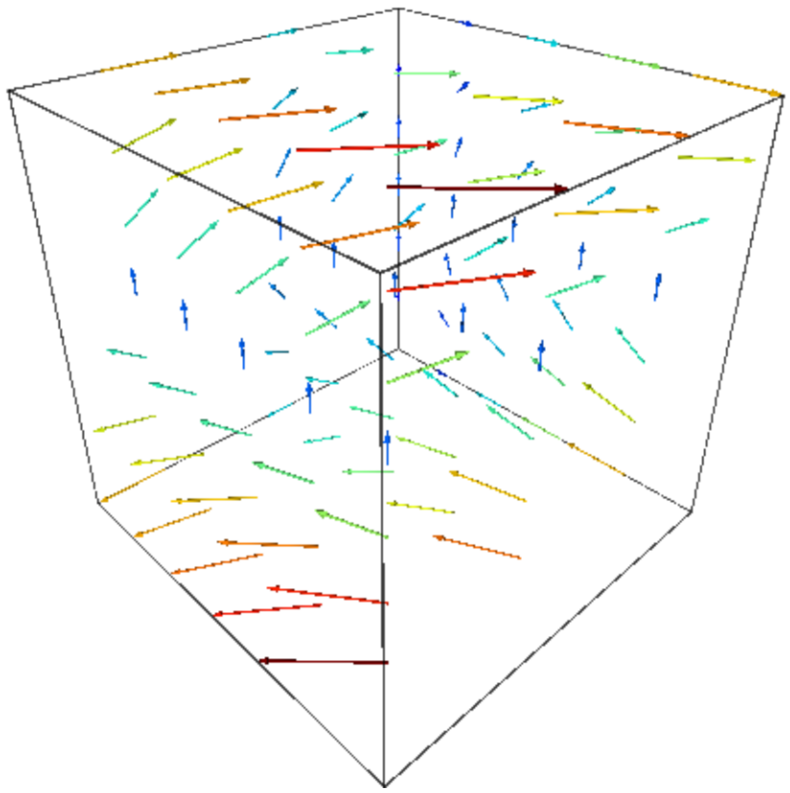
```
sage: plot_vector_field3d((x*cos(z), -y*cos(z), sin(z)),
....:                     (x,0,pi), (y,0,pi), (z,0,pi), plot_points=[3,5,7])
Graphics3d Object
```





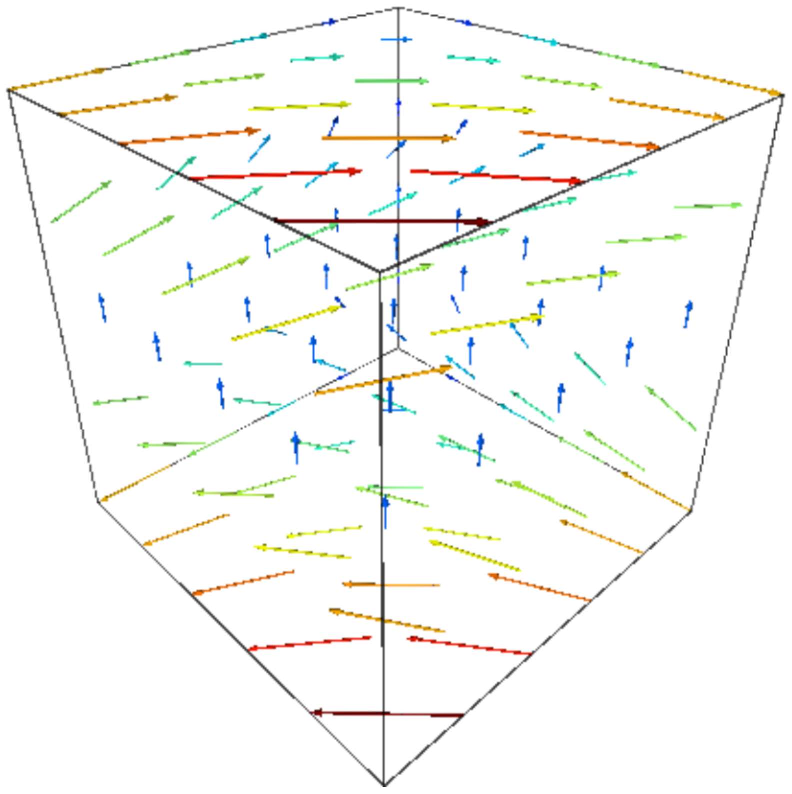






same example with the arrows centered on the points:

```
sage: plot_vector_field3d((x*cos(z), -y*cos(z), sin(z)),
....:                     (x,0,pi), (y,0,pi), (z,0,pi), center_arrows=True)
Graphics3d Object
```



2.5 Implicit plots

`sage.plot.plot3d.implicit_plot3d.implicit_plot3d(`*f*, *xrange*, *yrange*, *zrange*, ***kwds*`)`

Plot an isosurface of a function.

INPUT:

- `f` – function
- `xrange` – a 2-tuple (x_min, x_max) or a 3-tuple (x, x_min, x_max)
- `yrange` – a 2-tuple (y_min, y_max) or a 3-tuple (y, y_min, y_max)
- `zrange` – a 2-tuple (z_min, z_max) or a 3-tuple (z, z_min, z_max)
- `plot_points` – (default: "automatic", which is 40) the number of function evaluations in each direction. (The number of cubes in the marching cubes algorithm will be one less than this). Can be a triple of integers, to specify a different resolution in each of x,y,z.

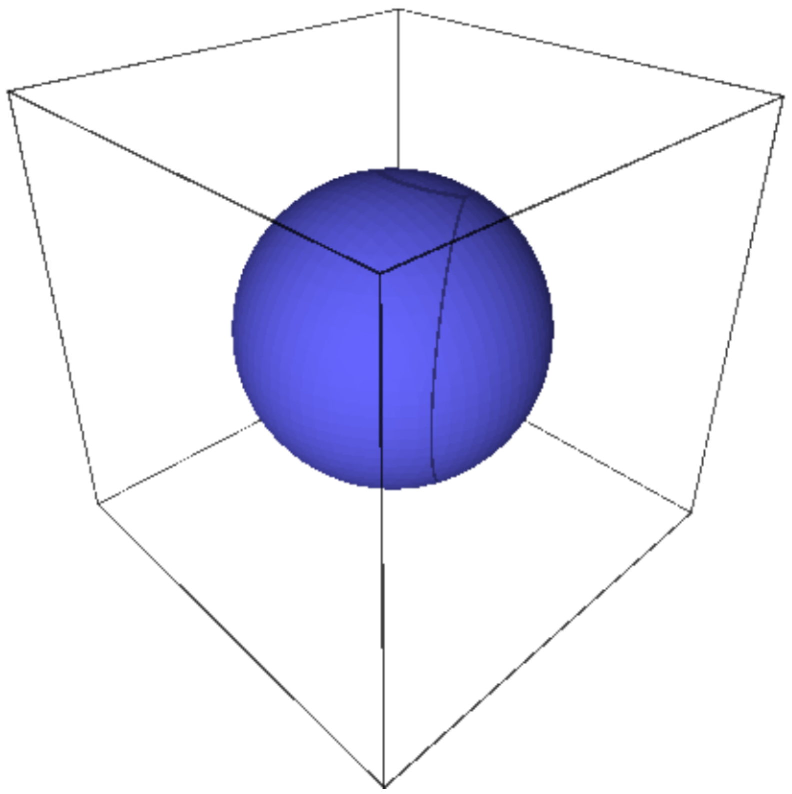
- contour – (default: 0) plot the isosurface `f(x,y,z) == contour`. Can be a list, in which case multiple contours are plotted.
- region – (default: None) If region is given, it must be a Python callable. Only segments of the surface where `region(x,y,z)` returns a number > 0 will be included in the plot. (Note that returning a Python boolean is acceptable, since `True == 1` and `False == 0`).

EXAMPLES:

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

A simple sphere:

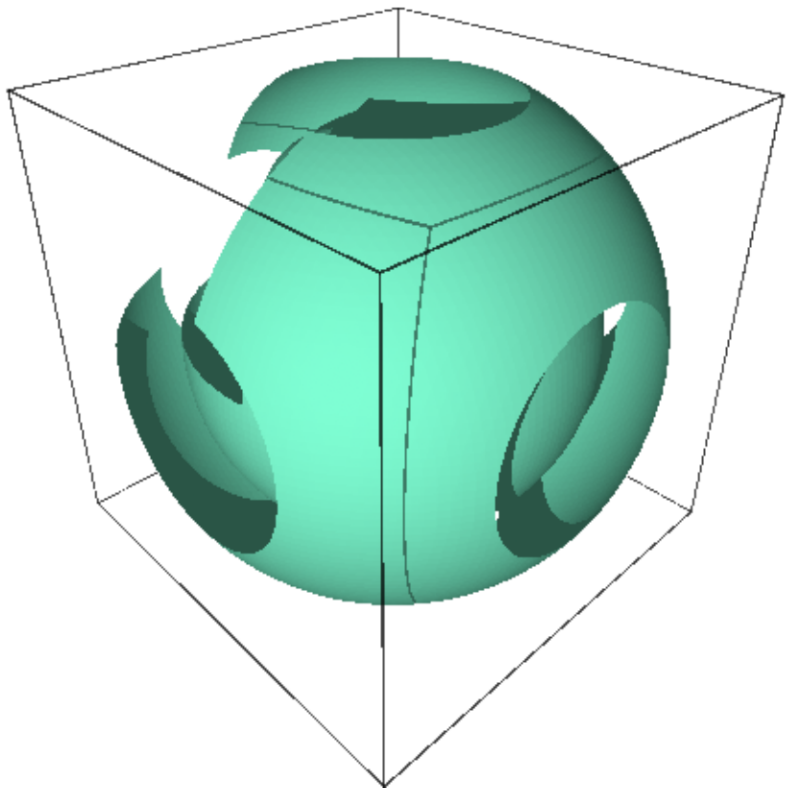
```
sage: implicit_plot3d(x^2+y^2+z^2==4, (x,-3,3), (y,-3,3), (z,-3,3))
Graphics3d Object
```



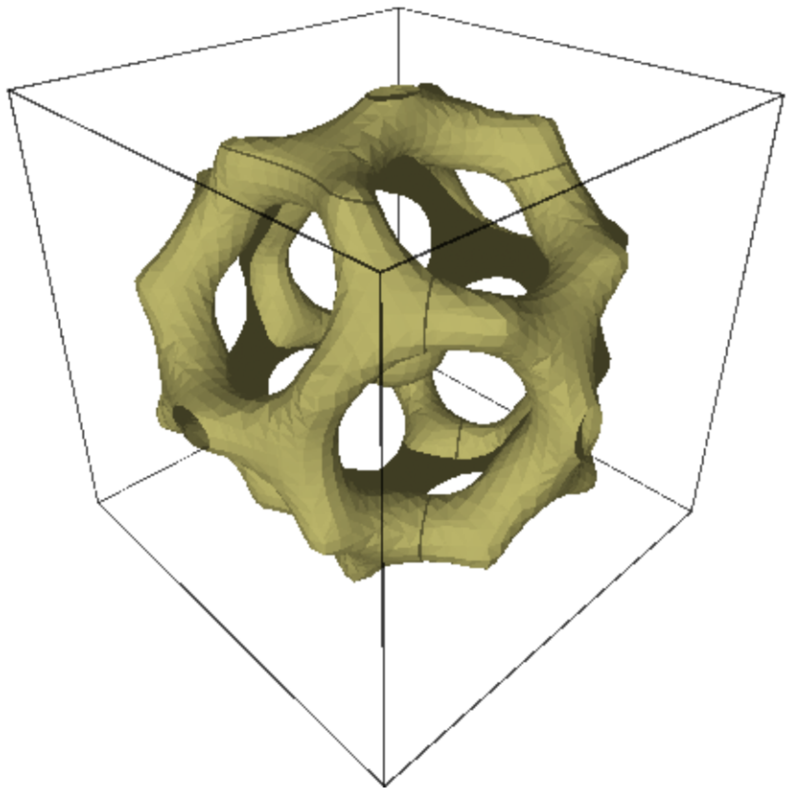
A nested set of spheres with a hole cut out:

```
sage: F = x^2 + y^2 + z^2
sage: P = implicit_plot3d(F, (x,-2,2), (y,-2,2), (z,-2,2), plot_points=60,
....:                     contour=[1,3,5], color='aquamarine',
....:                     region=lambda x,y,z: x<=0.2 or y>=0.2 or z<=0.2)
sage: P.show(viewer='tachyon')  # long time
```

A very pretty example, attributed to Douglas Summers-Stay (archived page):



```
sage: T = RDF(golden_ratio)
sage: F = 2 - (cos(x+T*y) + cos(x-T*y) + cos(y+T*z)
....:          + cos(y-T*z) + cos(z-T*x) + cos(z+T*x))
sage: r = 4.77
sage: implicit_plot3d(F, (x,-r,r), (y,-r,r), (z,-r,r),
....:                 plot_points=40, color='darkkhaki').show(viewer='tachyon')
```



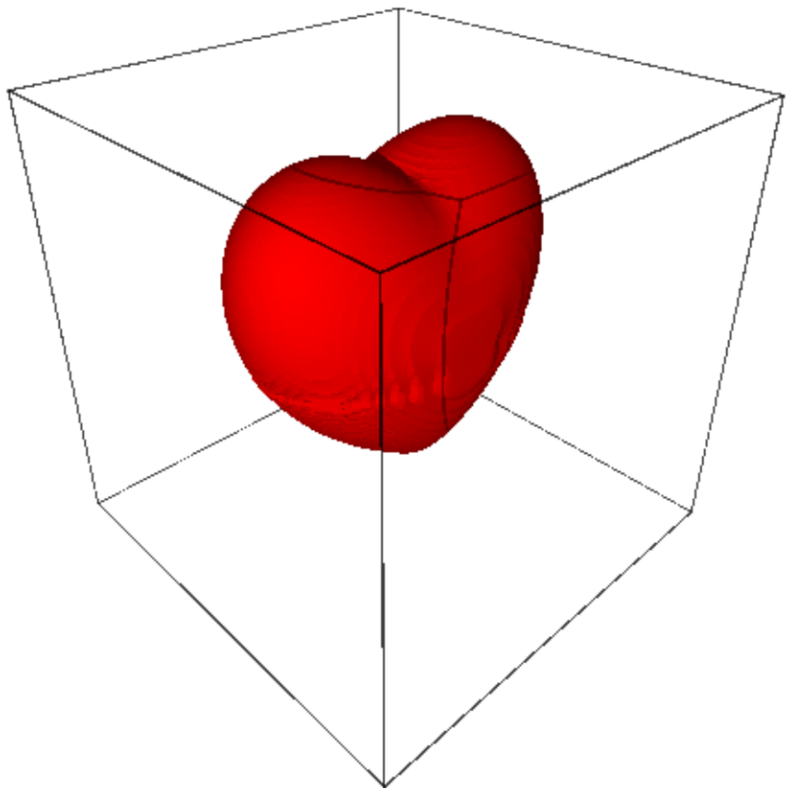
As I write this (but probably not as you read it), it's almost Valentine's day, so let's try a heart (from http://mathworld.wolfram.com/HeartSurface.html)

```
sage: F = (x^2+9/4*y^2+z^2-1)^3 - x^2*z^3 - 9/(80)*y^2*z^3
sage: r = 1.5
sage: V = implicit_plot3d(F, (x,-r,r), (y,-r,r), (z,-r,r),
....:                     plot_points=80, color='red', smooth=False)
sage: V.show(viewer='tachyon')  # long time
```

The same examples also work with the default Jmol viewer; for example:

```
sage: T = RDF(golden_ratio)
sage: F = 2 - (cos(x + T*y) + cos(x - T*y) + cos(y + T*z)
....:          + cos(y - T*z) + cos(z - T*x) + cos(z + T*x))
sage: r = 4.77
sage: implicit_plot3d(F, (x,-r,r), (y,-r,r), (z,-r,r),
```

(continues on next page)

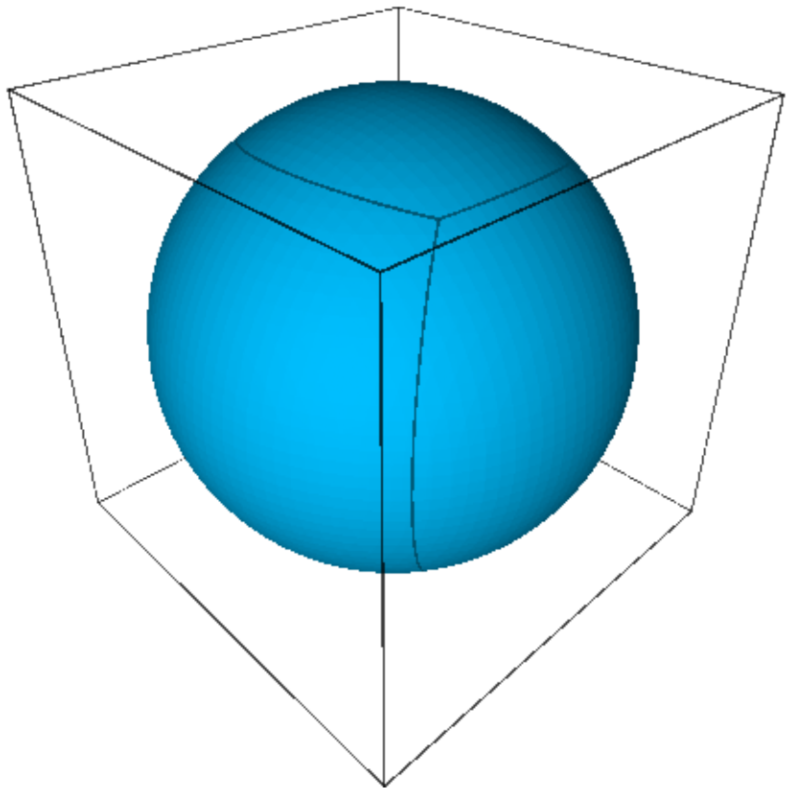


(continued from previous page)

```
....:                     plot_points=40, color='deepskyblue').show()
```

Here we use smooth=True with a Tachyon graph:

```
sage: implicit_plot3d(x^2 + y^2 + z^2, (x,-2,2), (y,-2,2), (z,-2,2),
....:                     contour=4, color='deepskyblue', smooth=True)
Graphics3d Object
```



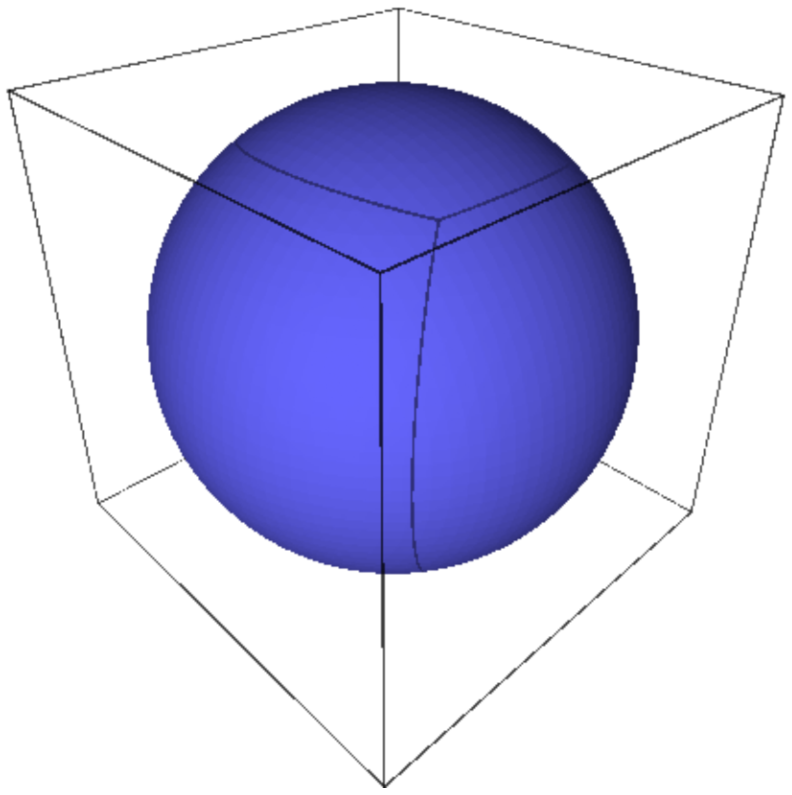
We explicitly specify a gradient function (in conjunction with smooth=True) and invert the normals:

```
sage: gx = lambda x, y, z: -(2*x + y^2 + z^2)
sage: gy = lambda x, y, z: -(x^2 + 2*y + z^2)
sage: gz = lambda x, y, z: -(x^2 + y^2 + 2*z)
sage: implicit_plot3d(x^2+y^2+z^2, (x,-2,2), (y,-2,2), (z,-2,2),
....:                     contour=4, plot_points=40, smooth=True,
....:                     gradient=(gx, gy, gz)).show(viewer='tachyon')
```

A graph of two metaballs interacting with each other:

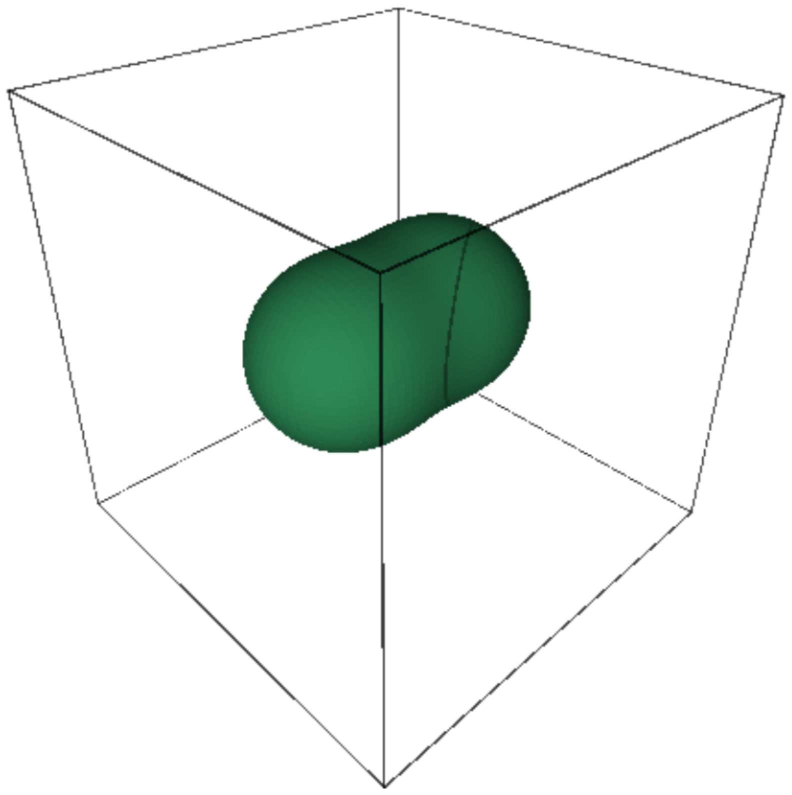
```
sage: def metaball(x0, y0, z0):
....:     return 1 / ((x-x0)^2+(y-y0)^2+(z-z0)^2)
sage: implicit_plot3d(metaball(-0.6,0,0) + metaball(0.6,0,0),
```

(continues on next page)



(continued from previous page)

```
....:                   (x,-2,2), (y,-2,2), (z,-2,2),
....:                   plot_points=60, contour=2, color='seagreen')
Graphics3d Object
```

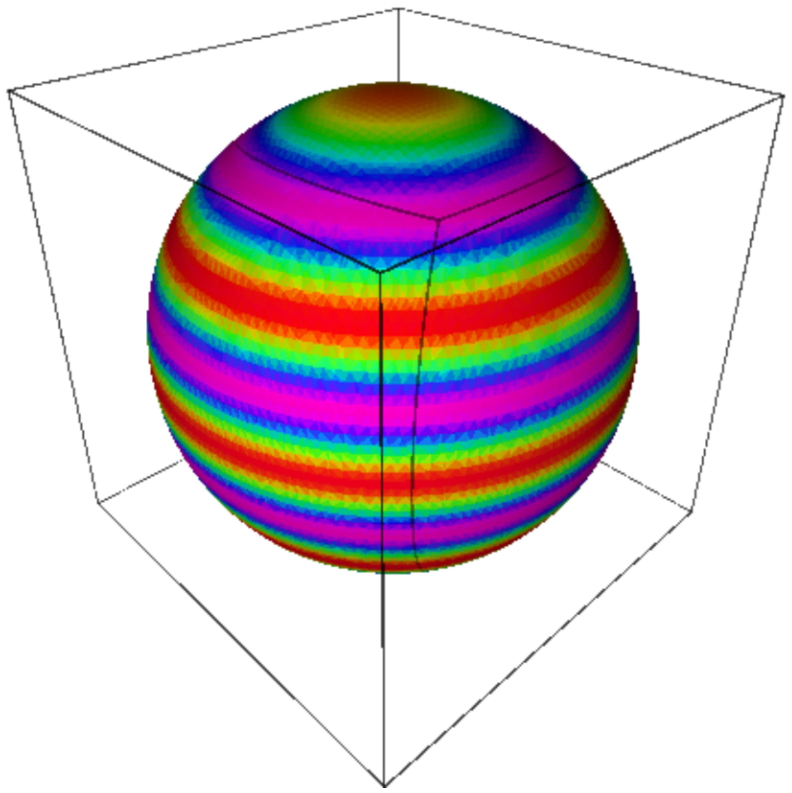


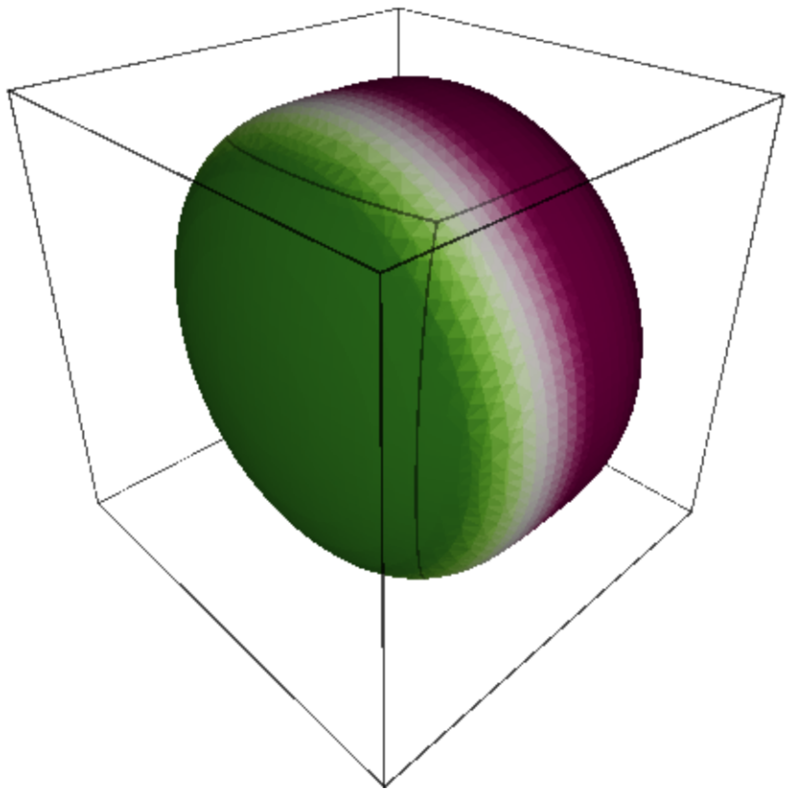
One can also color the surface using a coloring function and a colormap as follows. Note that the coloring function must take values in the interval $[0, 1]$.

```
sage: t = (sin(3*z)**2).function(x,y,z)
sage: cm = colormaps.gist_rainbow
sage: G = implicit_plot3d(x^2 + y^2 + z^2, (x,-2,2), (y,-2,2), (z,-2, 2),
....:                     contour=4, color=(t,cm), plot_points=100)
sage: G.show(viewer='tachyon')  # long time
```

Here is another colored example:

```
sage: x, y, z = var('x,y,z')
sage: t = (x).function(x,y,z)
sage: cm = colormaps.PiYG
sage: G = implicit_plot3d(x^4 + y^2 + z^2, (x,-2,2), (y,-2,2),(z,-2,2),
....:                     contour=4, color=(t,cm), plot_points=40)
sage: G
Graphics3d Object
```

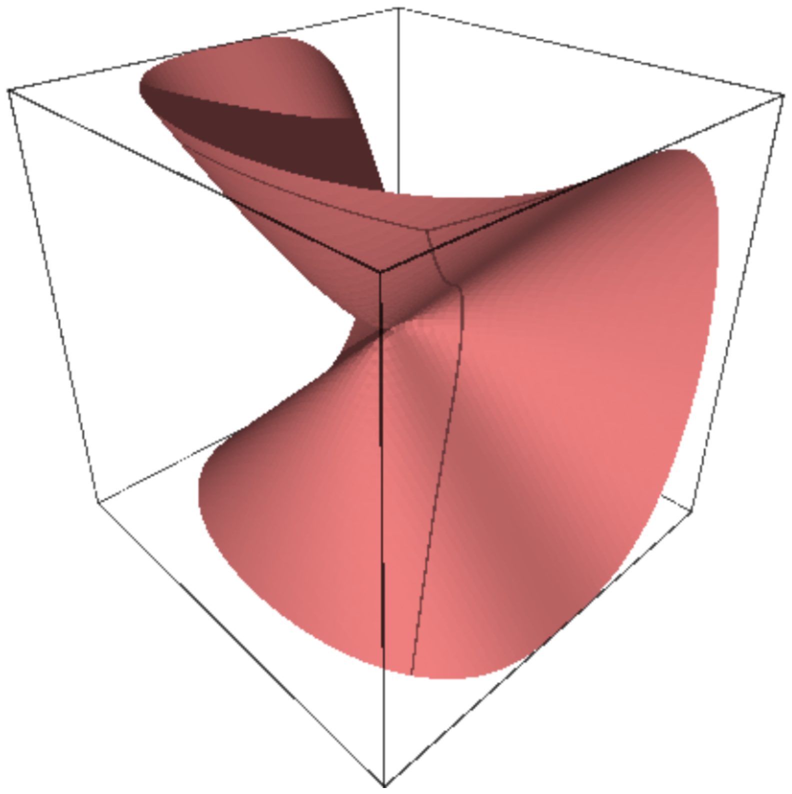




MANY MORE EXAMPLES:

A kind of saddle:

```
sage: implicit_plot3d(x^3 + y^2 - z^2, (x,-2,2), (y,-2,2), (z,-2,2),
....:                 plot_points=60, contour=0, color='lightcoral')
Graphics3d Object
```



A smooth surface with six radial openings:

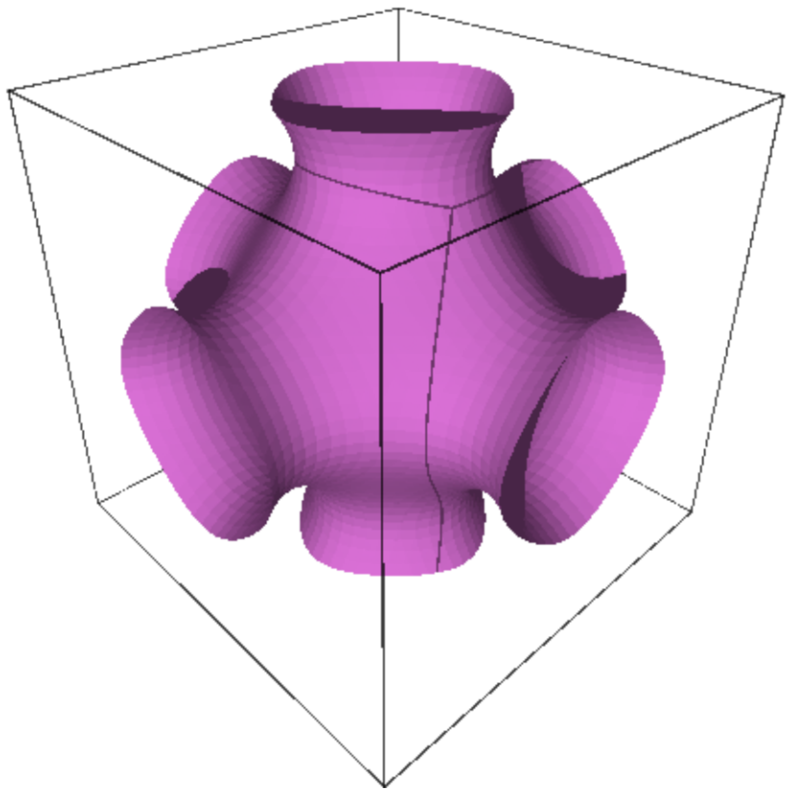
```
sage: implicit_plot3d(-(cos(x) + cos(y) + cos(z)),
....:                 (x,-4,4), (y,-4,4), (z,-4,4), color='orchid')
Graphics3d Object
```

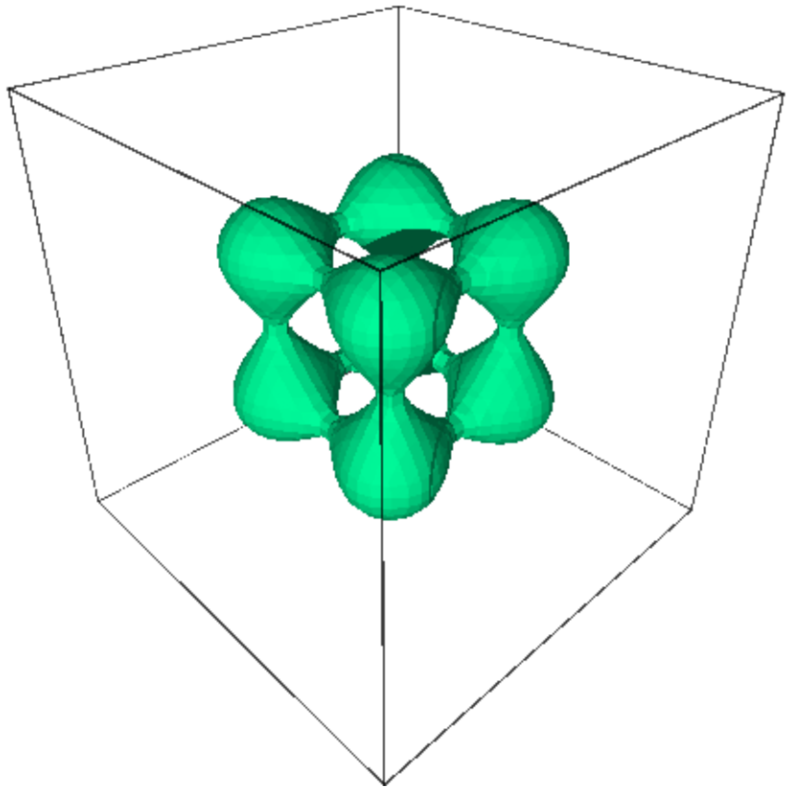
A cube composed of eight conjoined blobs:

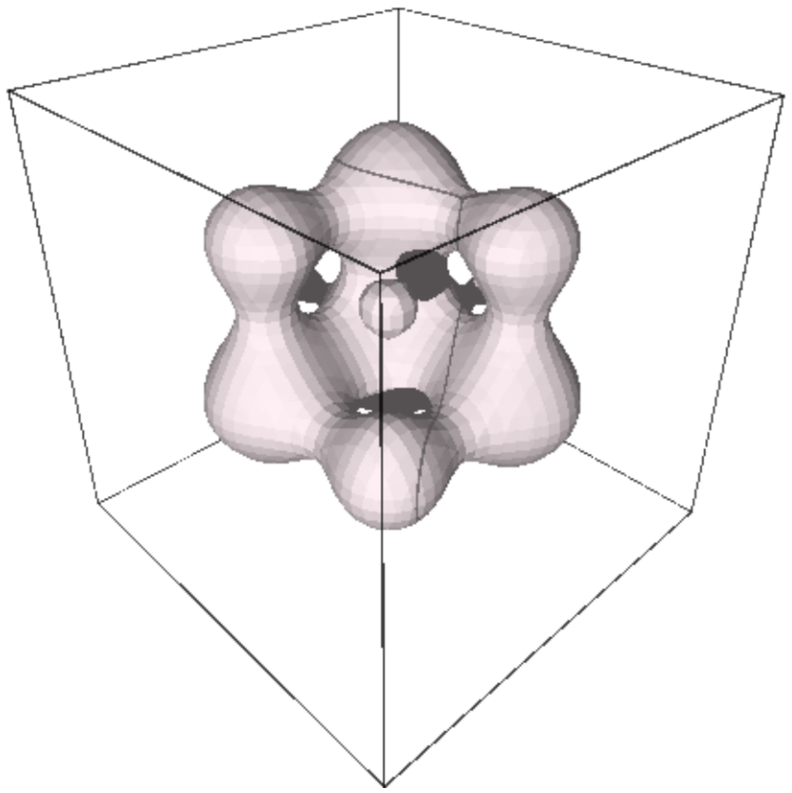
```
sage: F = x^2 + y^2 + z^2 + cos(4*x) + cos(4*y) + cos(4*z) - 0.2
sage: implicit_plot3d(F, (x,-2,2), (y,-2,2), (z,-2,2), color='mediumspringgreen')
Graphics3d Object
```

A variation of the blob cube featuring heterogeneously sized blobs:

```
sage: F = x^2 + y^2 + z^2 + sin(4*x) + sin(4*y) + sin(4*z) - 1
sage: implicit_plot3d(F, (x,-2,2), (y,-2,2), (z,-2,2), color='lavenderblush')
Graphics3d Object
```

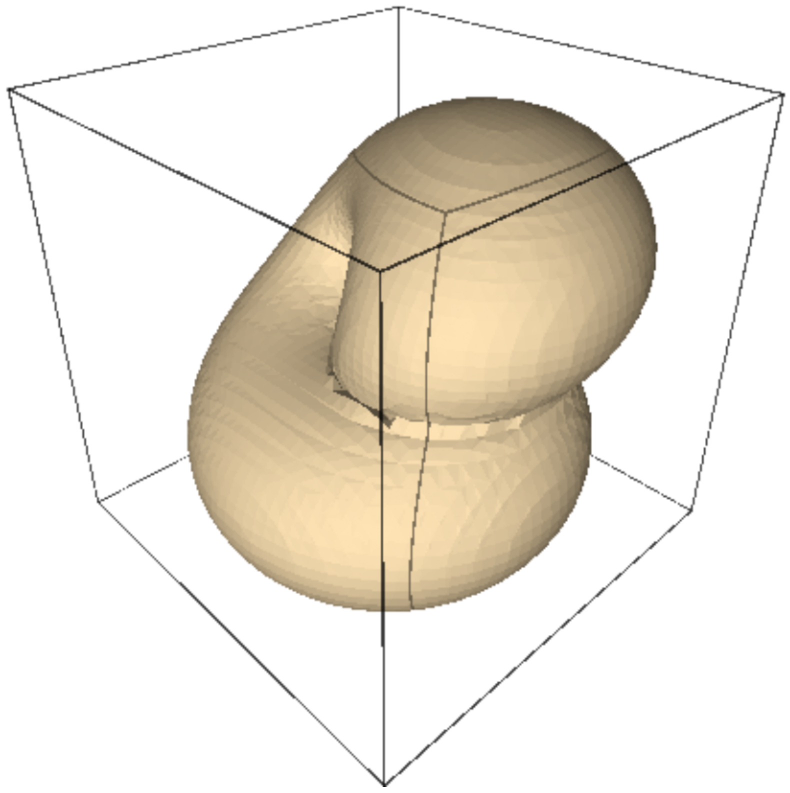






A Klein bottle:

```
sage: G = x^2 + y^2 + z^2
sage: F = (G+2*y-1)*((G-2*y-1)^2-8*z^2) + 16*x*z*(G-2*y-1)
sage: implicit_plot3d(F, (x,-3,3), (y,-3.1,3.1), (z,-4,4), color='moccasin')
Graphics3d Object
```



A lemniscate:

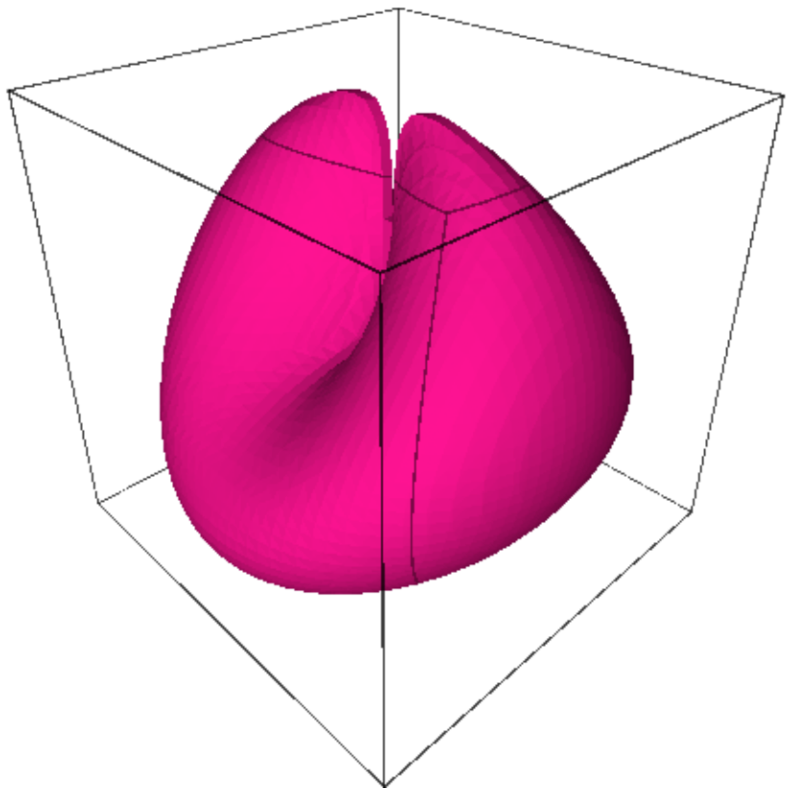
```
sage: F = 4*x^2*(x^2+y^2+z^2+z) + y^2*(y^2+z^2-1)
sage: implicit_plot3d(F, (x,-0.5,0.5), (y,-1,1), (z,-1,1), color='deeppink')
Graphics3d Object
```

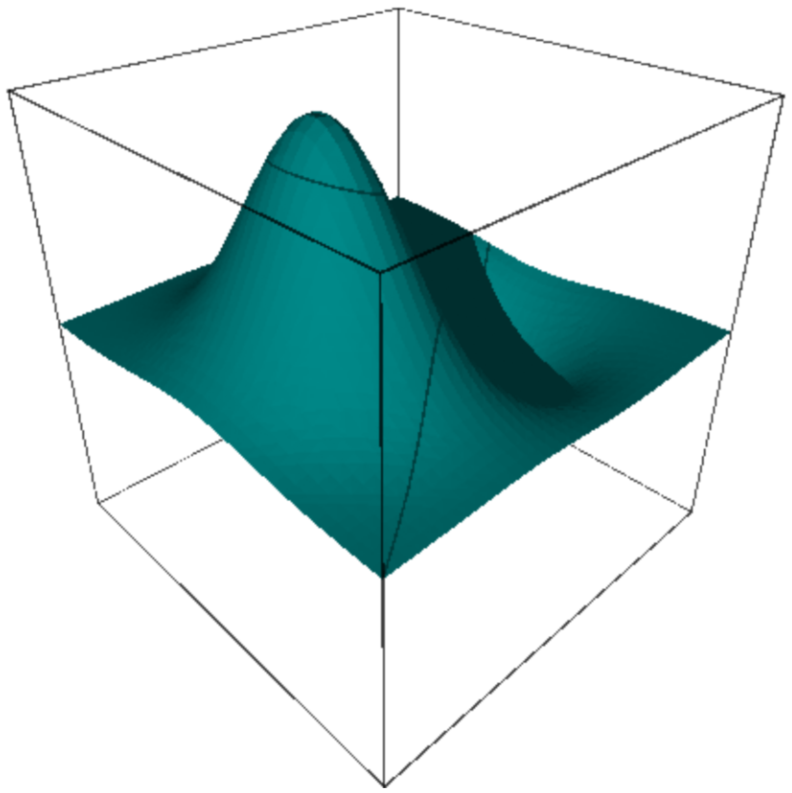
Drope:

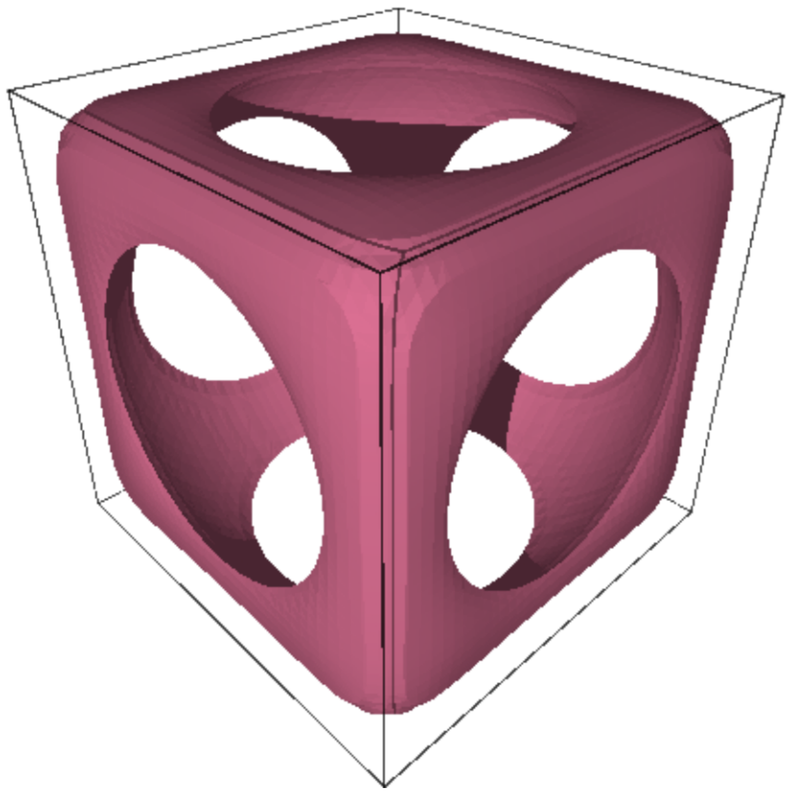
```
sage: implicit_plot3d(z - 4*x*exp(-x^2-y^2), (x,-2,2), (y,-2,2), (z,-1.7,1.7),
....:                 color='darkcyan')
Graphics3d Object
```

A cube with a circular aperture on each face:

```
sage: F = ((1/2.3)^2 * (x^2 + y^2 + z^2))^(-6) + ((1/2)^8 * (x^8 + y^8 + z^8))^6 - 1
sage: implicit_plot3d(F, (x,-2,2), (y,-2,2), (z,-2,2), color='palevioletred')
Graphics3d Object
```

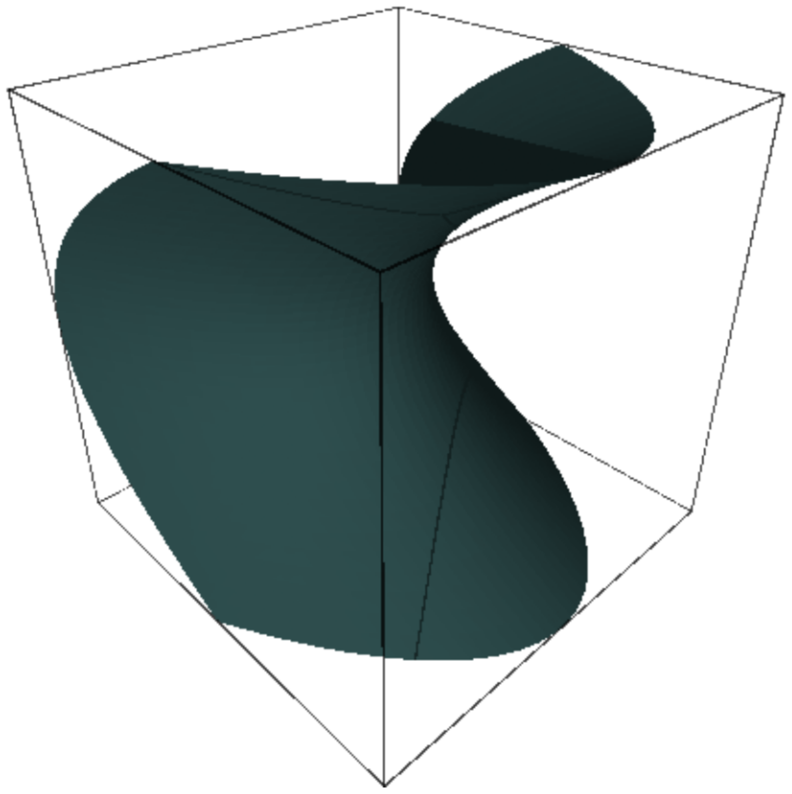






A simple hyperbolic surface:

```
sage: implicit_plot3d(x^2 + y - z^2, (x,-1,1), (y,-1,1), (z,-1,1),
....:                 color='darkslategray')
Graphics3d Object
```



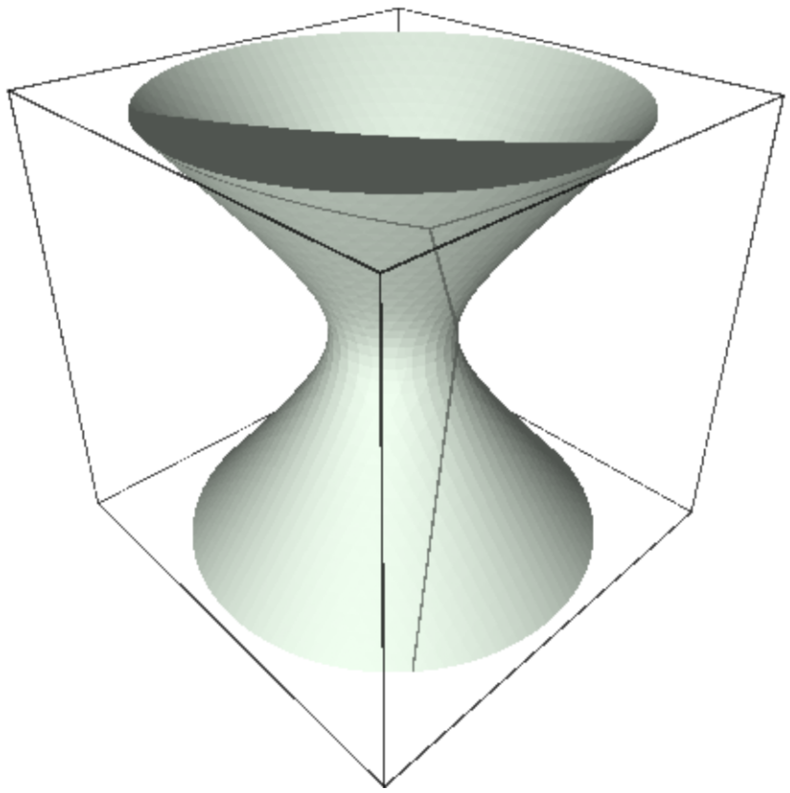
A hyperboloid:

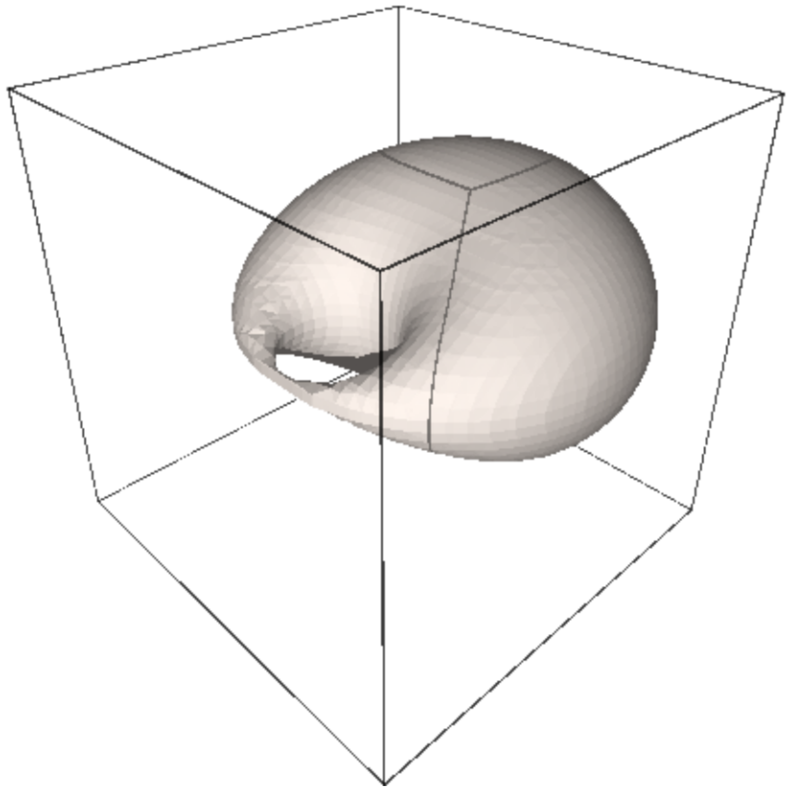
```
sage: implicit_plot3d(x^2 + y^2 - z^2 -0.3, (x,-2,2), (y,-2,2), (z,-1.8,1.8),
....:                 color='honeydew')
Graphics3d Object
```

Dupin cyclide (Wikipedia article Dupin_cyclide)

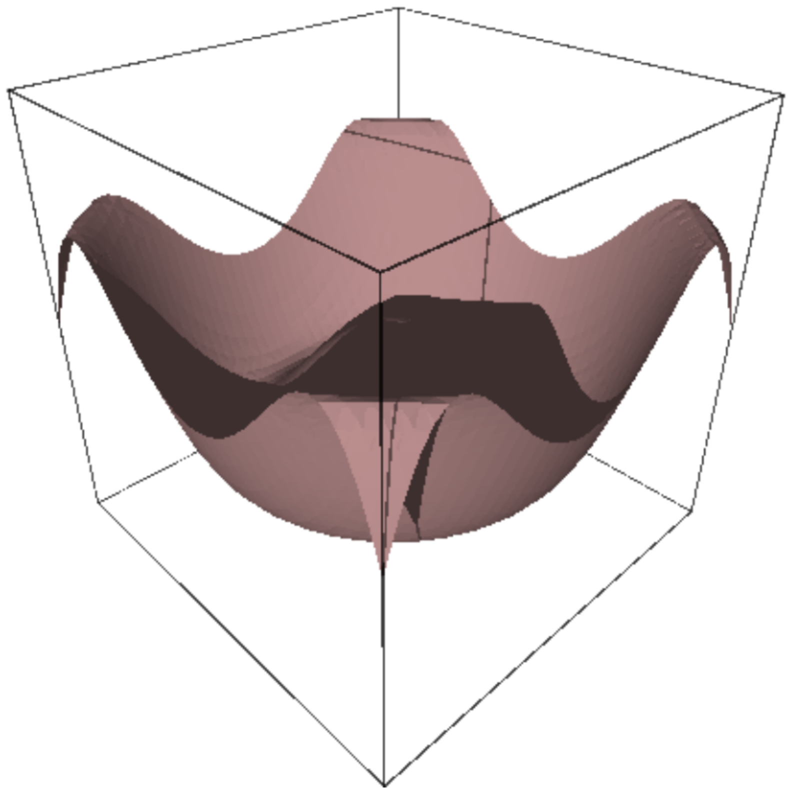
```
sage: x, y, z, a, b, c, d = var('x,y,z,a,b,c,d')
sage: a = 3.5
sage: b = 3
sage: c = sqrt(a^2 - b^2)
sage: d = 2
sage: F = (x^2 + y^2 + z^2 + b^2 - d^2)^2 - 4*(a*x-c*d)^2 - 4*b^2*y^2
sage: implicit_plot3d(F, (x,-6,6), (y,-6,6), (z,-6,6), color='seashell')
Graphics3d Object
```

Sinus:





```
sage: implicit_plot3d(sin(pi*((x)^2+(y)^2))/2 + z, (x,-1,1), (y,-1,1), (z,-1,1),
....:                 color='rosybrown')
Graphics3d Object
```



A torus:

```
sage: implicit_plot3d((sqrt(x*x+y*y)-3)^2 + z*z - 1, (x,-4,4), (y,-4,4), (z,-1,1),
....:                 color='indigo')
Graphics3d Object
```

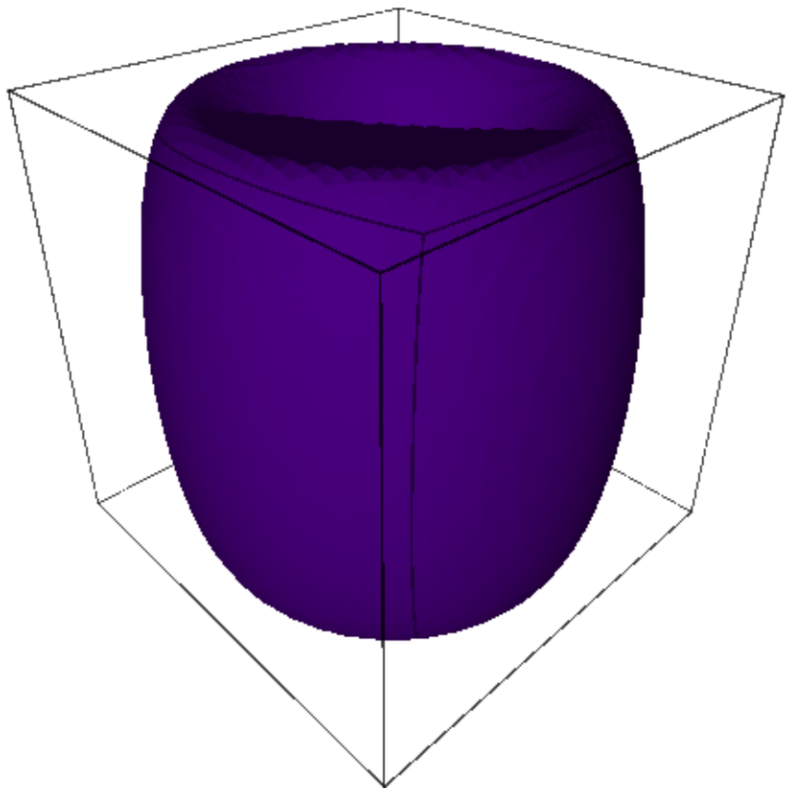
An octahedron:

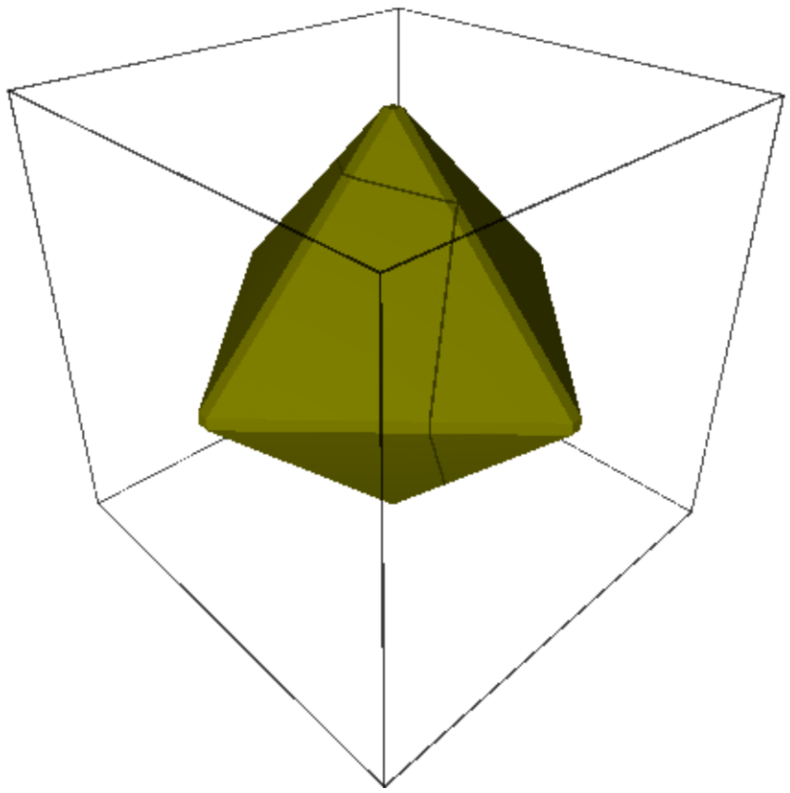
```
sage: implicit_plot3d(abs(x) + abs(y) + abs(z) - 1, (x,-1,1), (y,-1,1), (z,-1,1),
....:                 color='olive')
Graphics3d Object
```

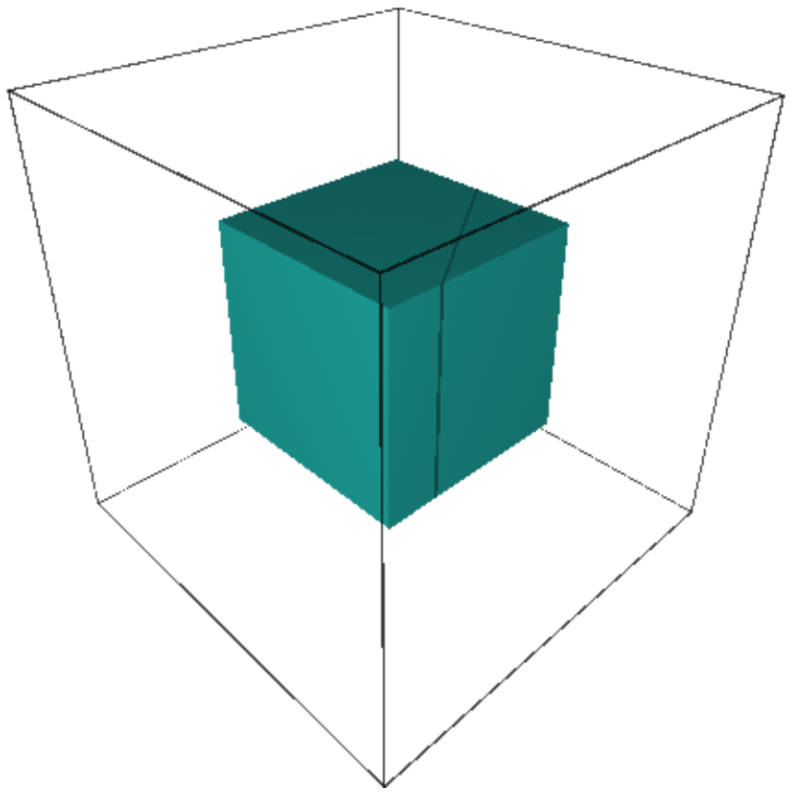
A cube:

```
sage: implicit_plot3d(x^100 + y^100 + z^100 - 1, (x,-2,2), (y,-2,2), (z,-2,2),
....:                 color='lightseagreen')
Graphics3d Object
```

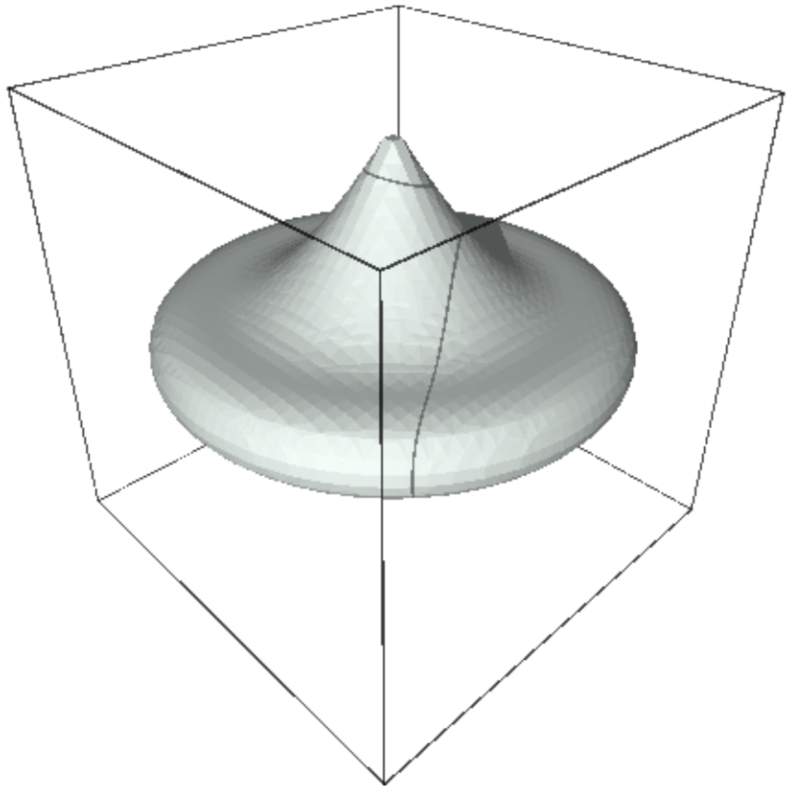
Toupie:







```
sage: implicit_plot3d((sqrt(x*x+y*y)-3)^3 + z*z - 1, (x,-4,4), (y,-4,4), (z,-6,6),
....:                 color='mintcream')
Graphics3d Object
```



A cube with rounded edges:

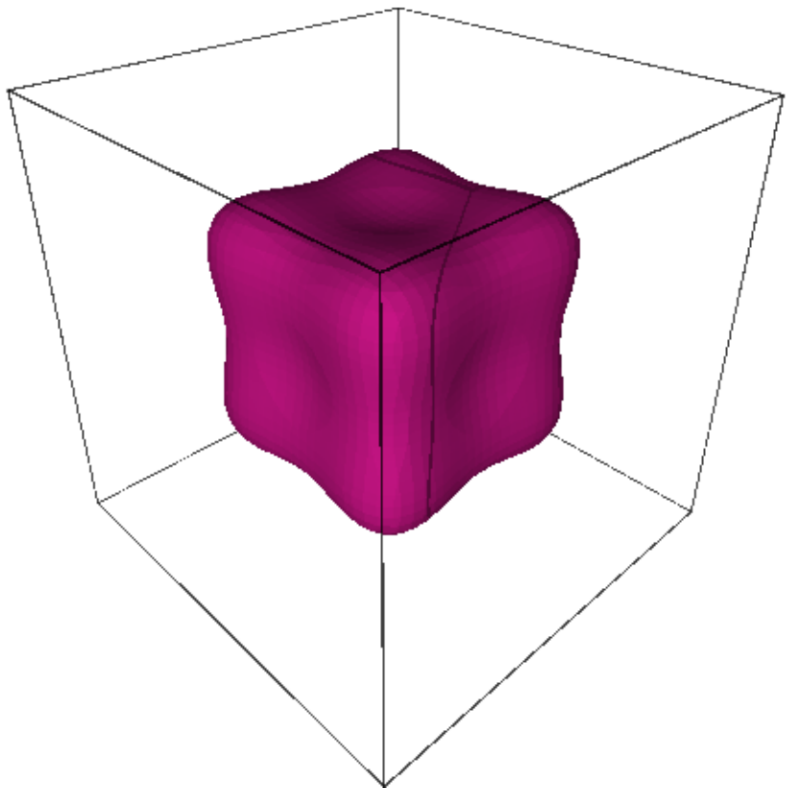
```
sage: F = x^4 + y^4 + z^4 - (x^2 + y^2 + z^2)
sage: implicit_plot3d(F, (x,-2,2), (y,-2,2), (z,-2,2), color='mediumvioletred')
Graphics3d Object
```

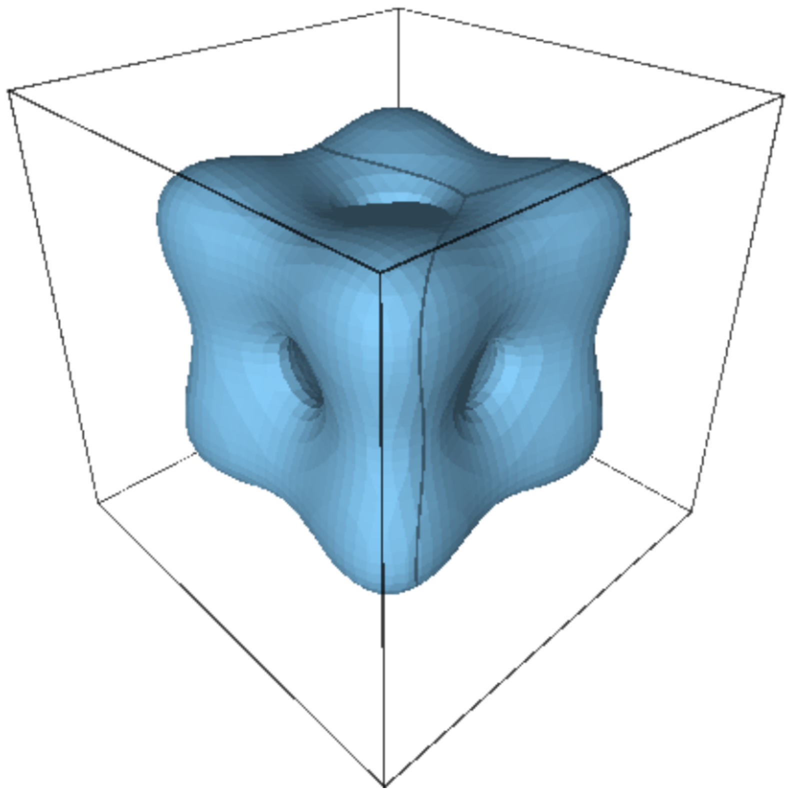
Chmutov:

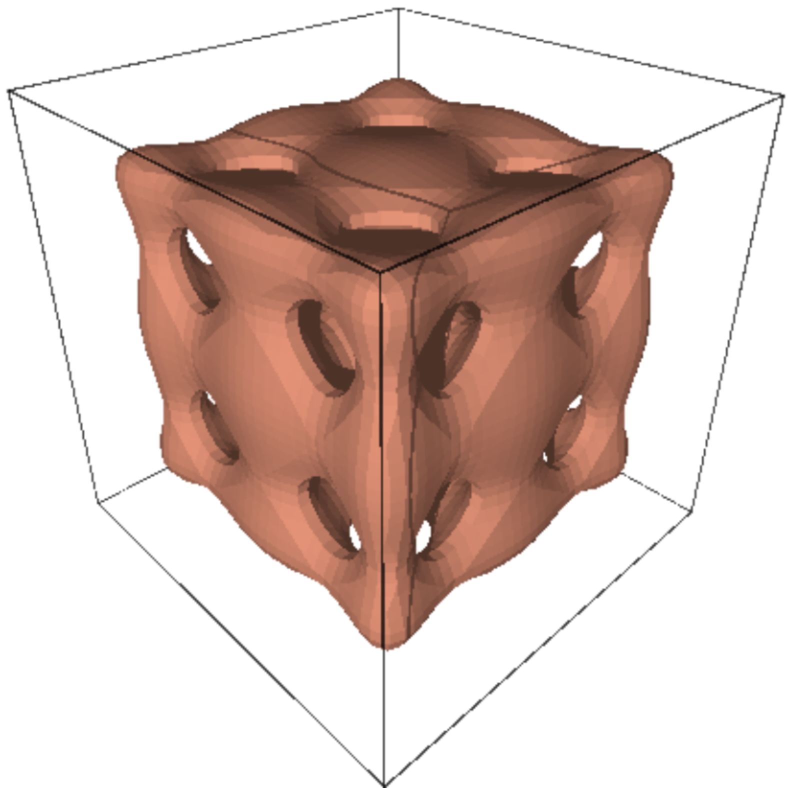
```
sage: F = x^4 + y^4 + z^4 - (x^2 + y^2 + z^2 - 0.3)
sage: implicit_plot3d(F, (x,-1.5,1.5), (y,-1.5,1.5), (z,-1.5,1.5),
....:                 color='lightskyblue')
Graphics3d Object
```

Further Chmutov:

```
sage: F = 2*(x^2*(3-4*x^2)^2+y^2*(3-4*y^2)^2+z^2*(3-4*z^2)^2) - 3
sage: implicit_plot3d(F, (x,-1.3,1.3), (y,-1.3,1.3), (z,-1.3,1.3),
....:                 color='darksalmon')
Graphics3d Object
```

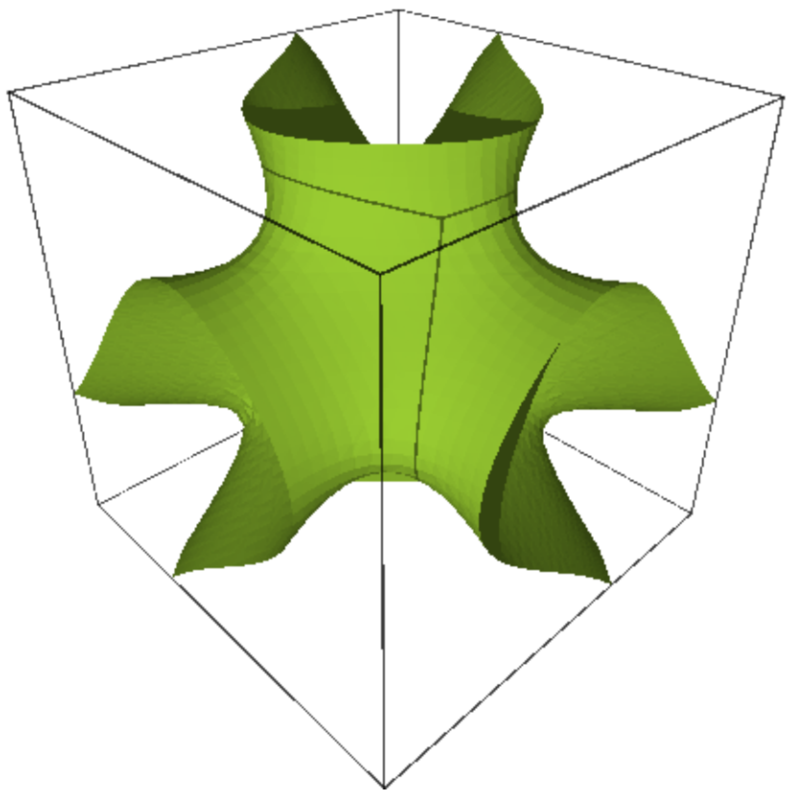






Clebsch surface:

```
sage: F_1 = 81 * (x^3+y^3+z^3)
sage: F_2 = 189 * (x^2*(y+z)+y^2*(x+z)+z^2*(x+y))
sage: F_3 = 54 * x * y * z
sage: F_4 = 126 * (x*y+x*z+y*z)
sage: F_5 = 9 * (x^2+y^2+z^2)
sage: F_6 = 9 * (x+y+z)
sage: F = F_1 - F_2 + F_3 + F_4 - F_5 + F_6 + 1
sage: implicit_plot3d(F, (x,-1,1), (y,-1,1), (z,-1,1), color='yellowgreen')
Graphics3d Object
```

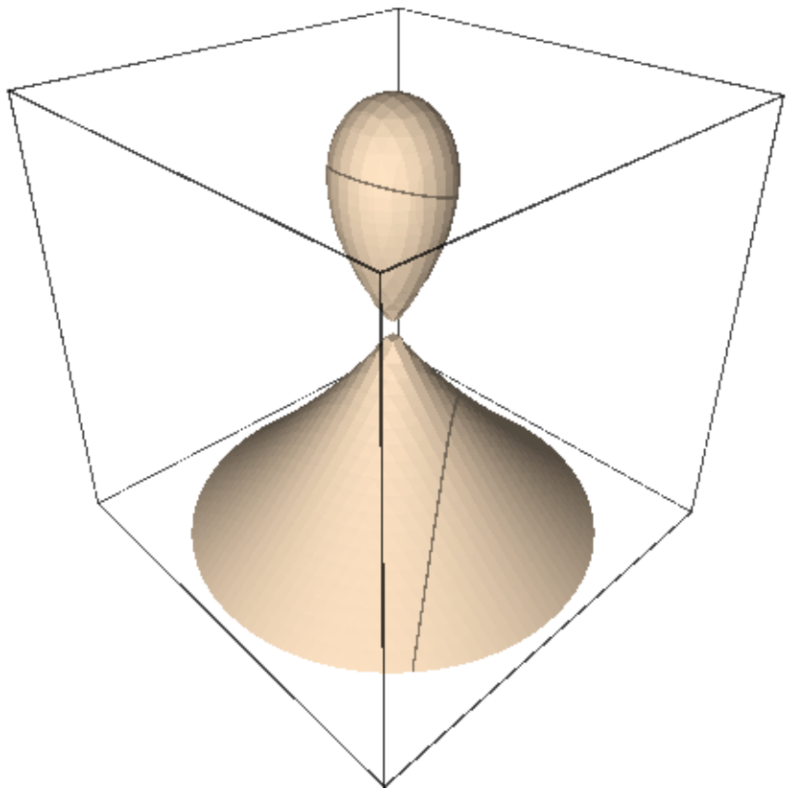


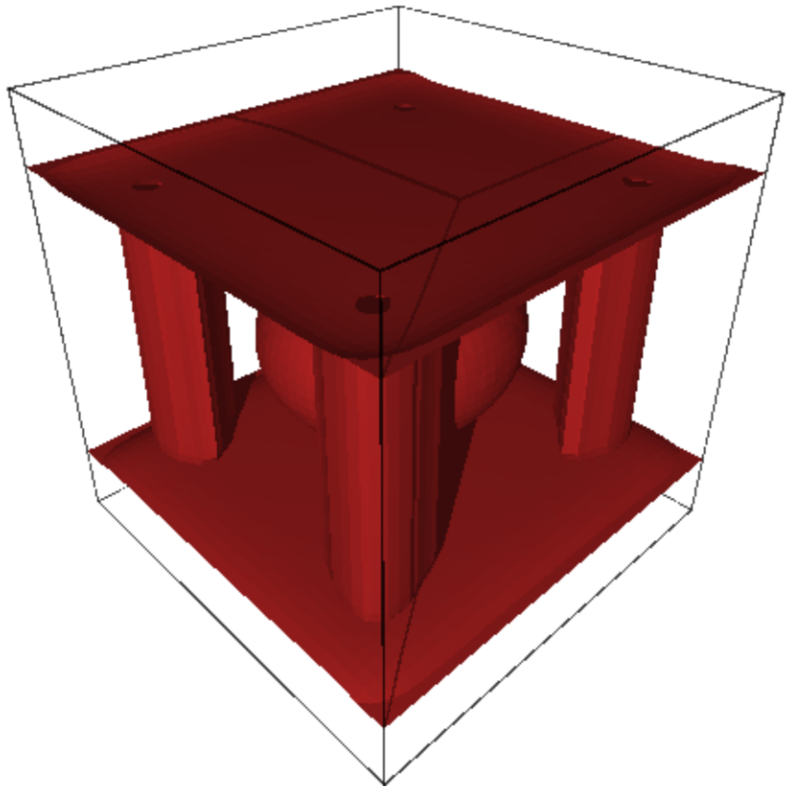
Looks like a water droplet:

```
sage: implicit_plot3d(x^2 + y^2 - (1-z)*z^2, (x,-1.5,1.5), (y,-1.5,1.5), (z,-1,1),
....:                 color='bisque')
Graphics3d Object
```

Sphere in a cage:

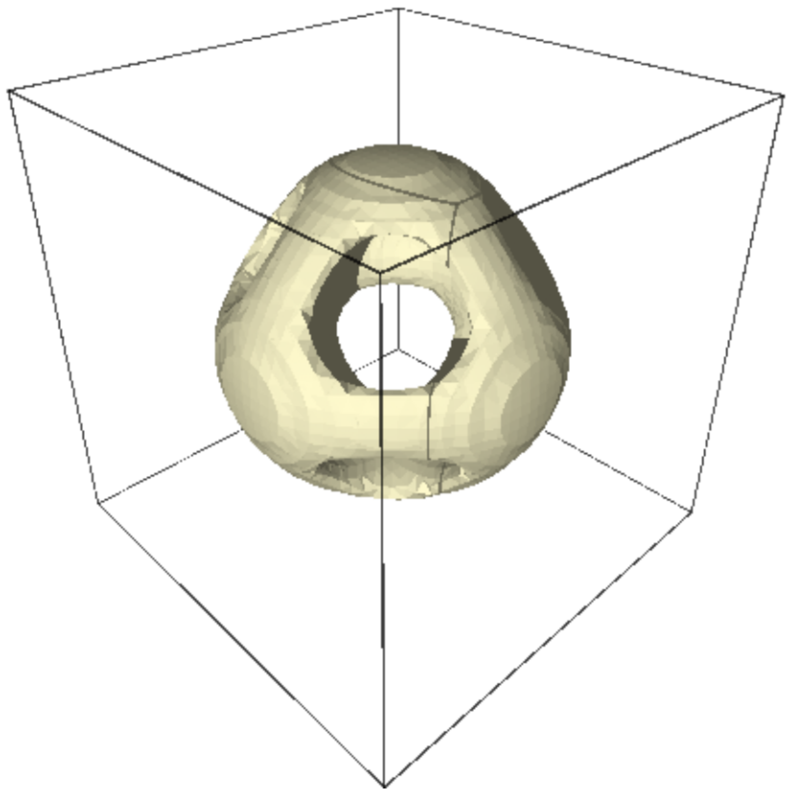
```
sage: F = (x^8+z^30+y^8-(x^4 + z^50 + y^4 -0.3)) * (x^2+y^2+z^2-0.5)
sage: implicit_plot3d(F, (x,-1.2,1.2), (y,-1.3,1.3), (z,-1.5,1.5),
....:                 color='firebrick')
Graphics3d Object
```





Ortho circle:

```
sage: F = (((x^2+y^2-1)^2+z^2) * ((y^2+z^2-1)^2+x^2) * ((z^2+x^2-1)^2+y^2)
....:      - 0.075^2 * (1+3*(x^2+y^2+z^2)))
sage: implicit_plot3d(F, (x,-1.5,1.5), (y,-1.5,1.5), (z,-1.5,1.5),
....:                 color='lemonchiffon')
Graphics3d Object
```



Cube sphere:

```
sage: F = 12 - ((1/2.3)^2 *(x^2 + y^2 + z^2))^-6 - ((1/2)^8 * (x^8 + y^8 + z^8))^6
sage: implicit_plot3d(F, (x,-2,2), (y,-2,2), (z,-2,2), color='rosybrown')
Graphics3d Object
```

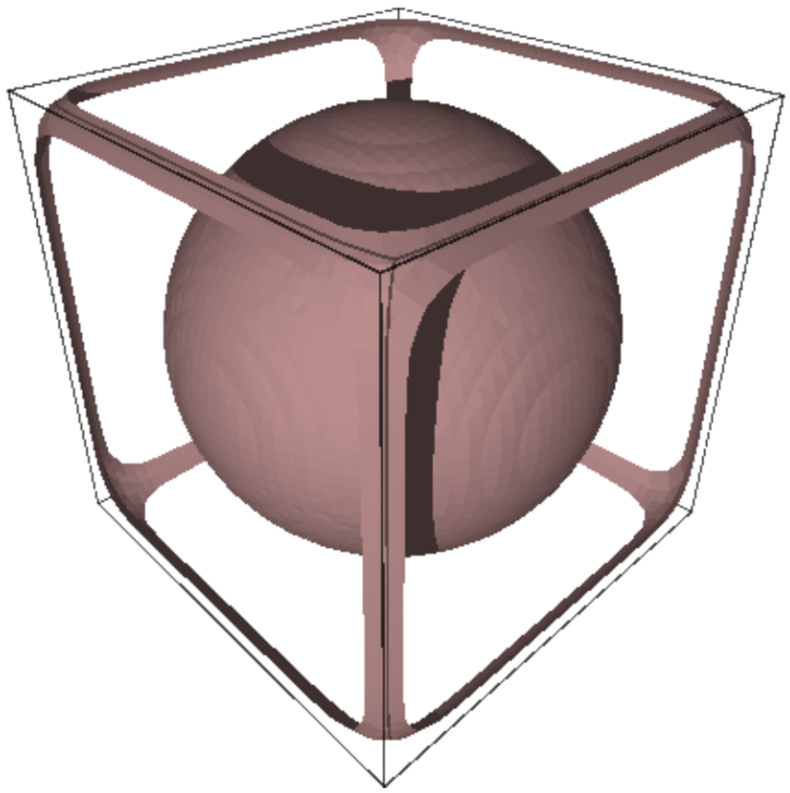
Two cylinders intersect to make a cross:

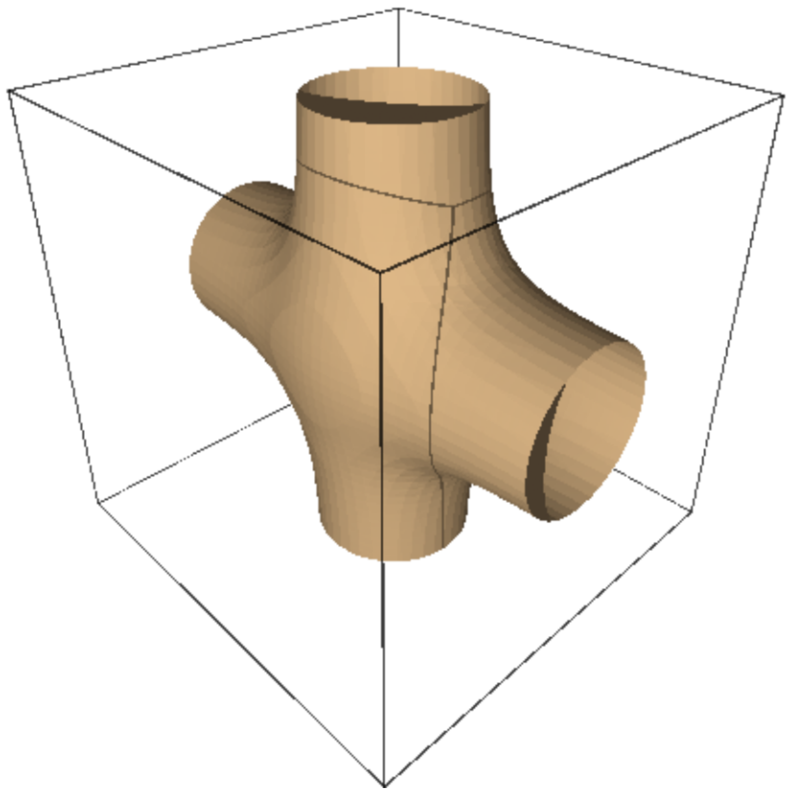
```
sage: implicit_plot3d((x^2+y^2-1) * (x^2+z^2-1) - 1, (x,-3,3), (y,-3,3), (z,-3,3),
....:                 color='burlywood')
Graphics3d Object
```

Three cylinders intersect in a similar fashion:

```
sage: implicit_plot3d((x^2+y^2-1) * (x^2+z^2-1) * (y^2+z^2-1) - 1,
....:                 (x,-3,3), (y,-3,3), (z,-3,3), color='aqua')
```

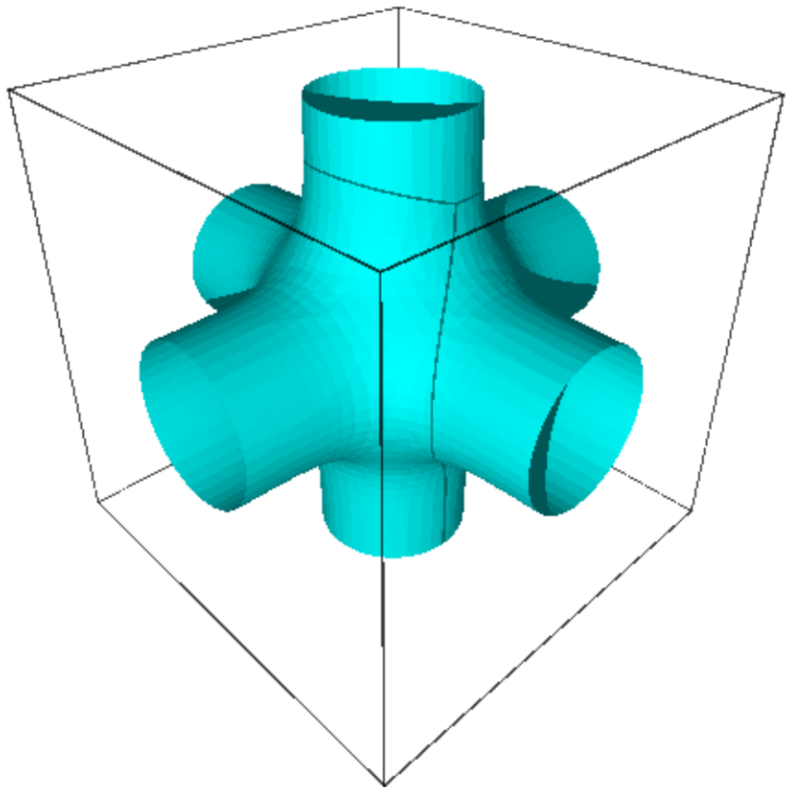
(continues on next page)





(continued from previous page)

```
Graphics3d Object
```



A sphere-ish object with twelve holes, four on each XYZ plane:

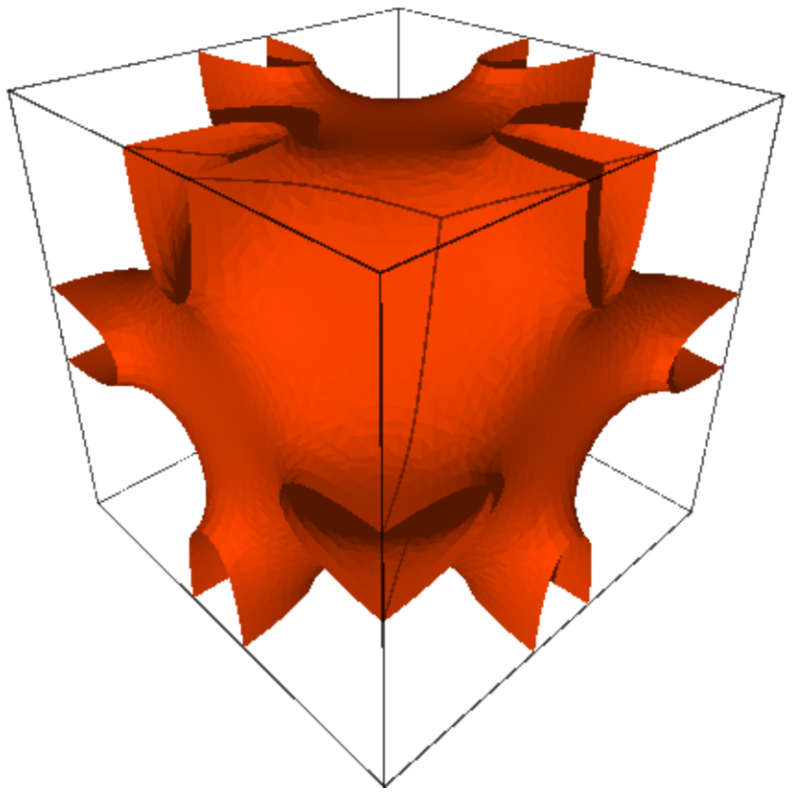
```
sage: implicit_plot3d(3*(cos(x)+cos(y)+cos(z)) + 4*cos(x)*cos(y)*cos(z),
....:                 (x,-3,3), (y,-3,3), (z,-3,3), color='orangered')
Graphics3d Object
```

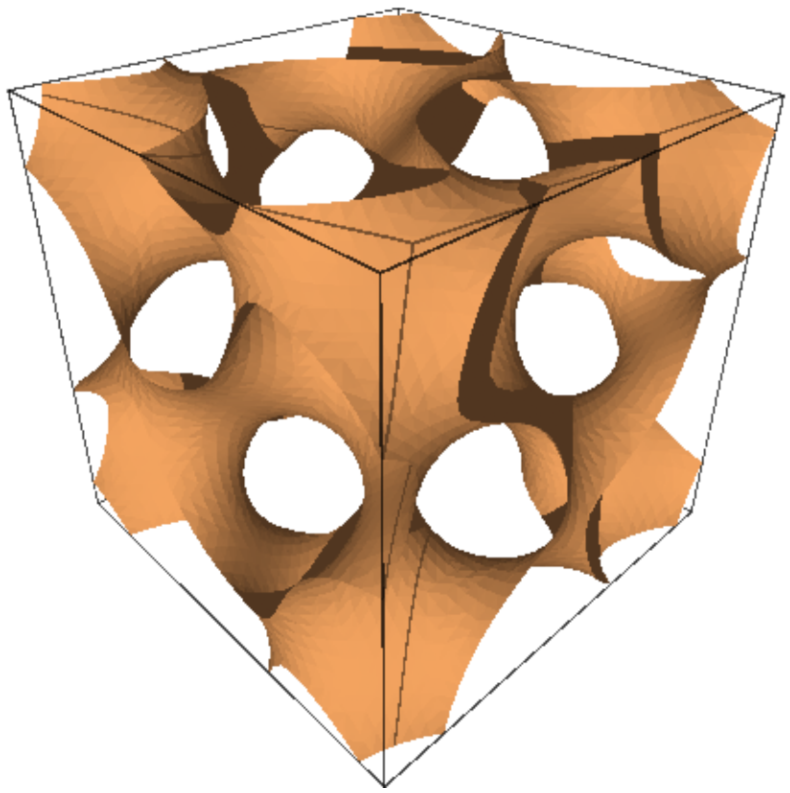
A gyroid:

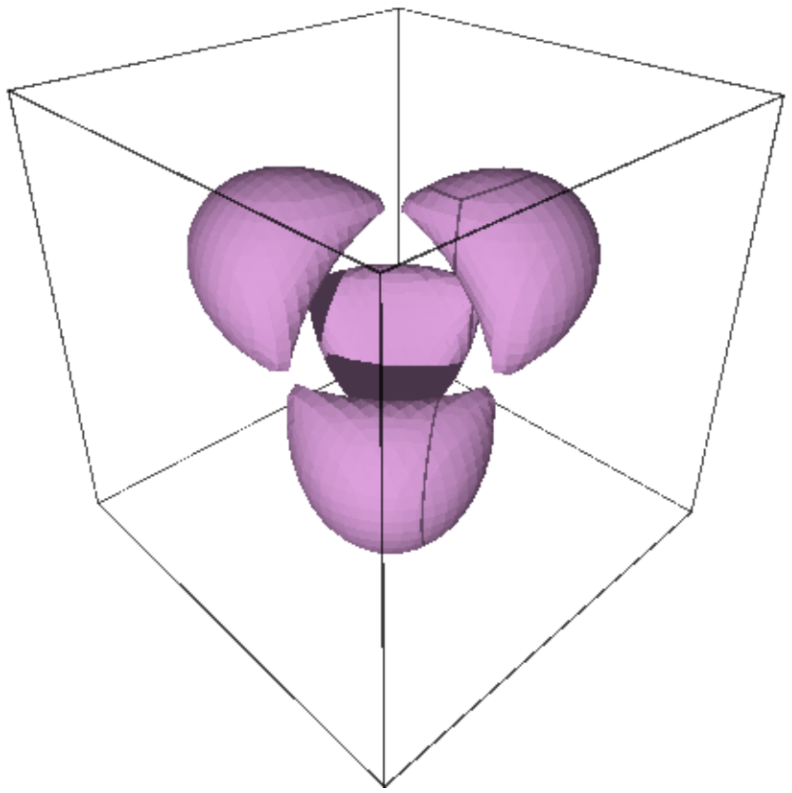
```
sage: implicit_plot3d(cos(x)*sin(y) + cos(y)*sin(z) + cos(z)*sin(x),
....:                 (x,-4,4), (y,-4,4), (z,-4,4), color='sandybrown')
Graphics3d Object
```

Tetrahedra:

```
sage: implicit_plot3d((x^2+y^2+z^2)^2 + 8*x*y*z - 10*(x^2+y^2+z^2) + 25,
....:                 (x,-4,4), (y,-4,4), (z,-4,4), color='plum')
Graphics3d Object
```







2.6 List plots

`sage.plot.plot3d.list_plot3d.`**`list_plot3d`**(*v*, *interpolation_type='default'*, *point_list=None*, ***kwds*)

A 3-dimensional plot of a surface defined by the list v of points in 3-dimensional space.

INPUT:

- `v` - something that defines a set of points in 3 space:
  - a matrix
  - a list of 3-tuples
  - a list of lists (all of the same length) - this is treated the same as a matrix.

OPTIONAL KEYWORDS:

- `interpolation_type` - 'linear', 'clough' (CloughTocher2D), 'spline'

  'linear' will perform linear interpolation

  The option 'clough' will interpolate by using a piecewise cubic interpolating Bezier polynomial on each triangle, using a Clough-Tocher scheme. The interpolant is guaranteed to be continuously differentiable. The gradients of the interpolant are chosen so that the curvature of the interpolating surface is approximatively minimized.

  The option 'spline' interpolates using a bivariate B-spline.

  When v is a matrix the default is to use linear interpolation, when v is a list of points the default is 'clough'.

- `degree` - an integer between 1 and 5, controls the degree of spline used for spline interpolation. For data that is highly oscillatory use higher values
- `point_list` - If point_list=True is passed, then if the array is a list of lists of length three, it will be treated as an array of points rather than a 3xn array.
- `num_points` - Number of points to sample interpolating function in each direction, when `interpolation_type` is not `default`. By default for an $n \times n$ array this is n.
- `**kwds` - all other arguments are passed to the surface function

OUTPUT: a 3d plot

EXAMPLES:

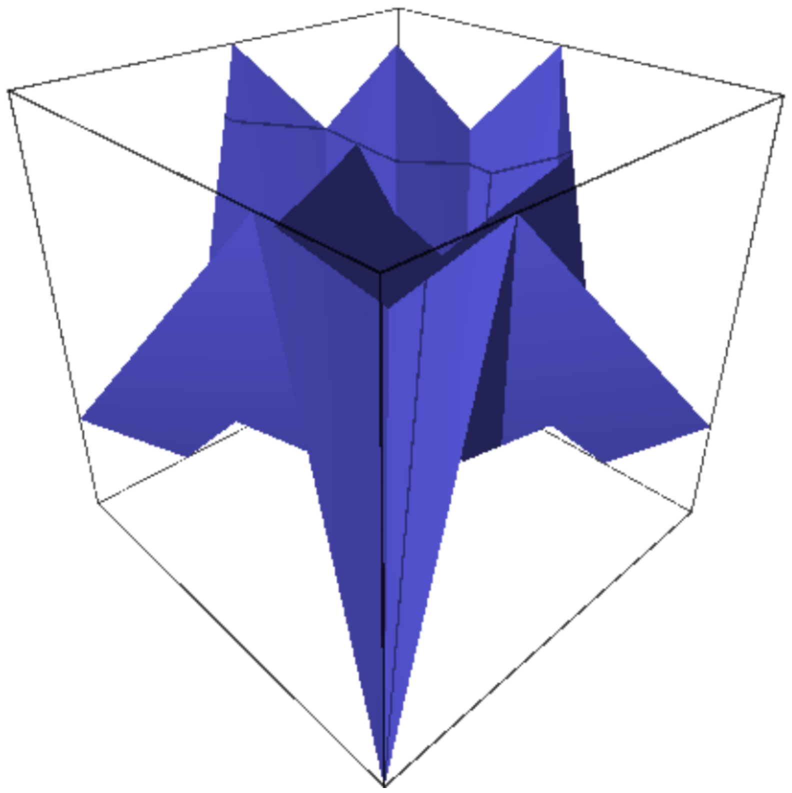
We plot a matrix that illustrates summation modulo n:

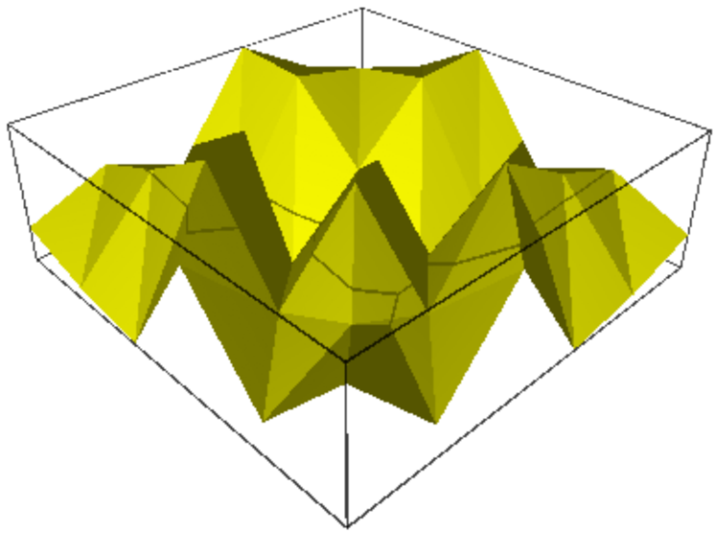
```
sage: n = 5
sage: list_plot3d(matrix(RDF, n, [(i+j)%n for i in [1..n] for j in [1..n]]))
Graphics3d Object
```

We plot a matrix of values of `sin`:

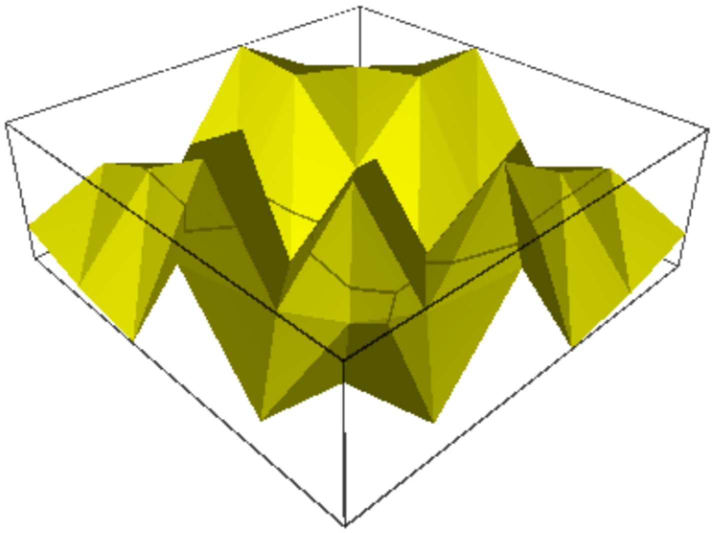
```
sage: from math import pi
sage: m = matrix(RDF, 6, [sin(i^2 + j^2)
....:                     for i in [0,pi/5,..,pi] for j in [0,pi/5,..,pi]])
sage: list_plot3d(m, color='yellow', frame_aspect_ratio=[1, 1, 1/3])
Graphics3d Object
```

Though it does not change the shape of the graph, increasing `num_points` can increase the clarity of the graph:





```
sage: list_plot3d(m, color='yellow', num_points=40,
....:             frame_aspect_ratio=[1, 1, 1/3])
Graphics3d Object
```



We can change the interpolation type:

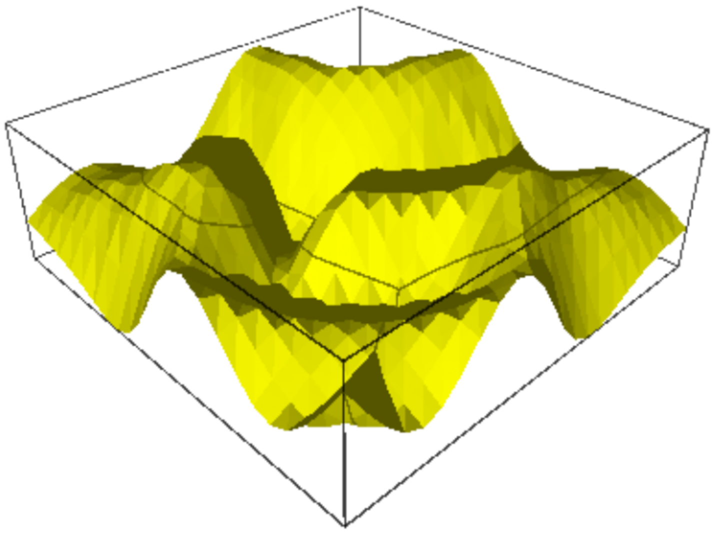
```
sage: import warnings
sage: warnings.simplefilter('ignore', UserWarning)
sage: list_plot3d(m, color='yellow', interpolation_type='clough',
....:             frame_aspect_ratio=[1, 1, 1/3])
Graphics3d Object
```

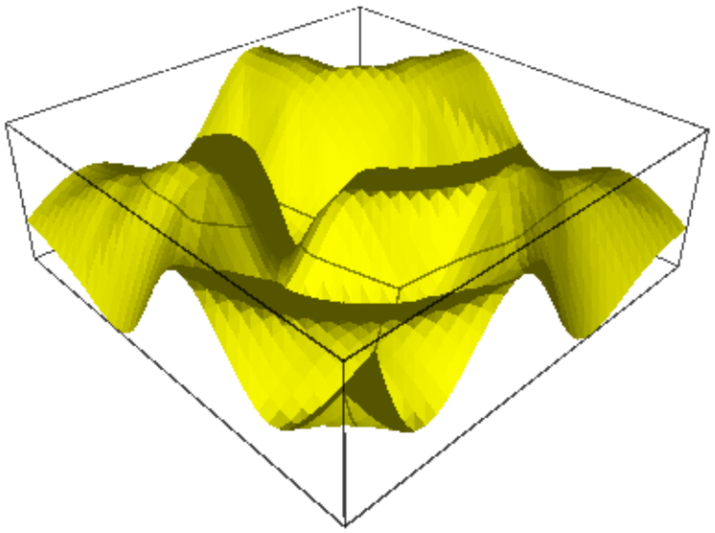
We can make this look better by increasing the number of samples:

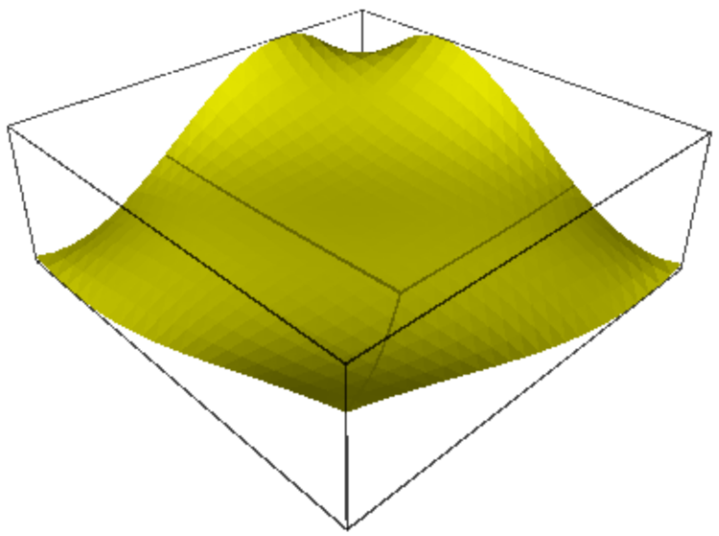
```
sage: list_plot3d(m, color='yellow', interpolation_type='clough',
....:             frame_aspect_ratio=[1, 1, 1/3], num_points=40)
Graphics3d Object
```

Let us try a spline:

```
sage: list_plot3d(m, color='yellow', interpolation_type='spline',
....:             frame_aspect_ratio=[1, 1, 1/3])
Graphics3d Object
```

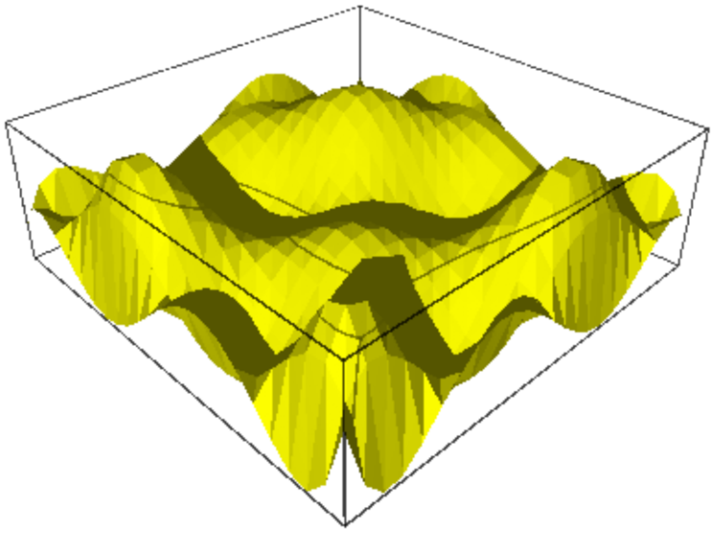






That spline does not capture the oscillation very well; let's try a higher degree spline:

```
sage: list_plot3d(m, color='yellow', interpolation_type='spline', degree=5,
....:             frame_aspect_ratio=[1, 1, 1/3])
Graphics3d Object
```



We plot a list of lists:

```
sage: show(list_plot3d([[1, 1, 1, 1], [1, 2, 1, 2], [1, 1, 3, 1], [1, 2, 1, 4]]))
```

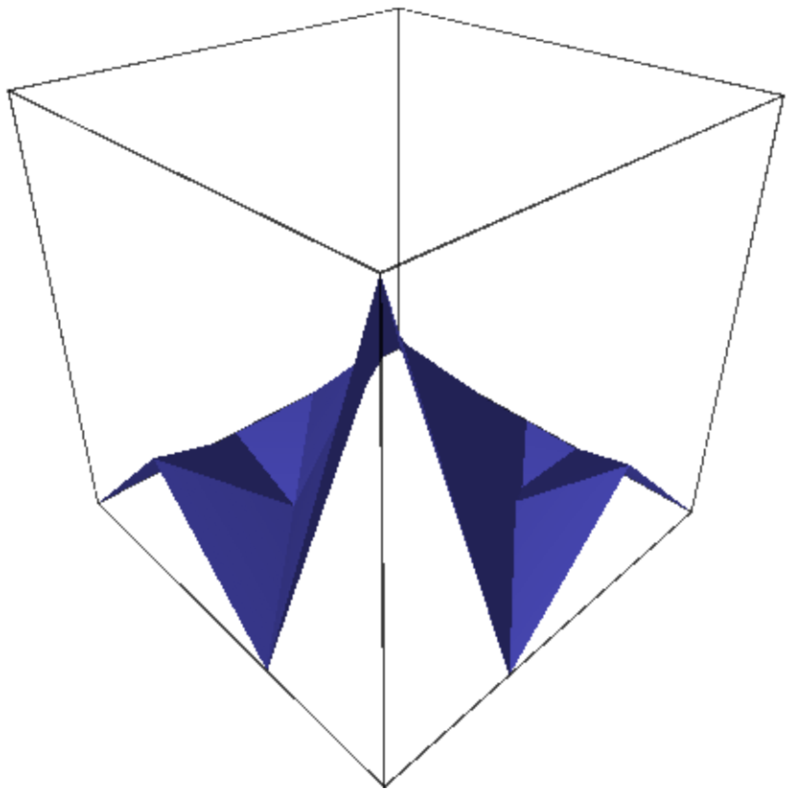
We plot a list of points. As a first example we can extract the (x,y,z) coordinates from the above example and make a list plot out of it. By default we do linear interpolation:

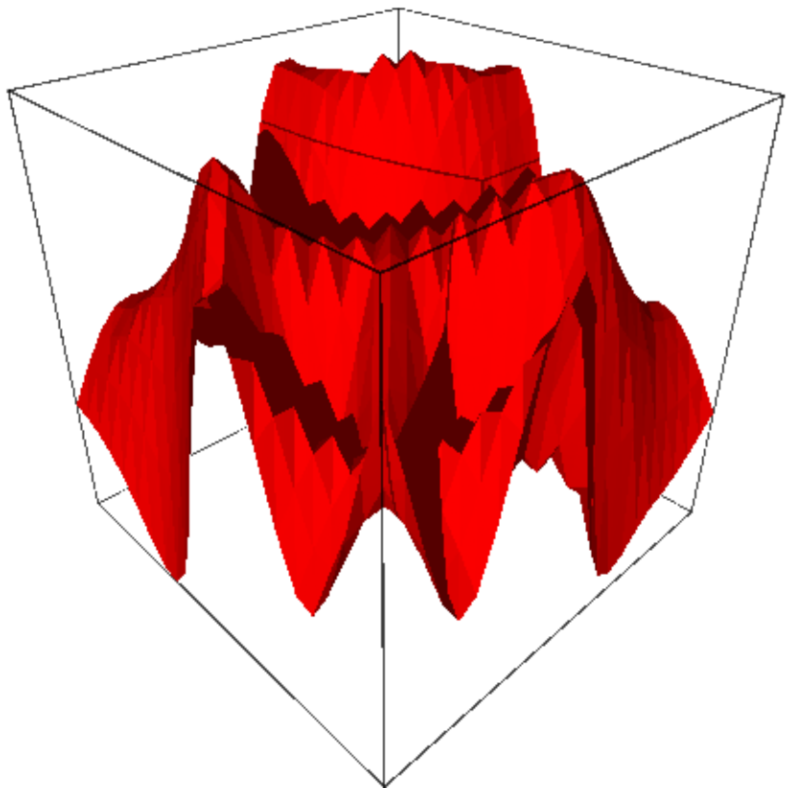
```
sage: l = []
sage: for i in range(6):
....:     for j in range(6):
....:         l.append((float(i*pi/5), float(j*pi/5), m[i, j]))
sage: list_plot3d(l, color='red')
Graphics3d Object
```

Note that the points do not have to be regularly sampled. For example:

```
sage: l = []
sage: for i in range(-5, 5):
```

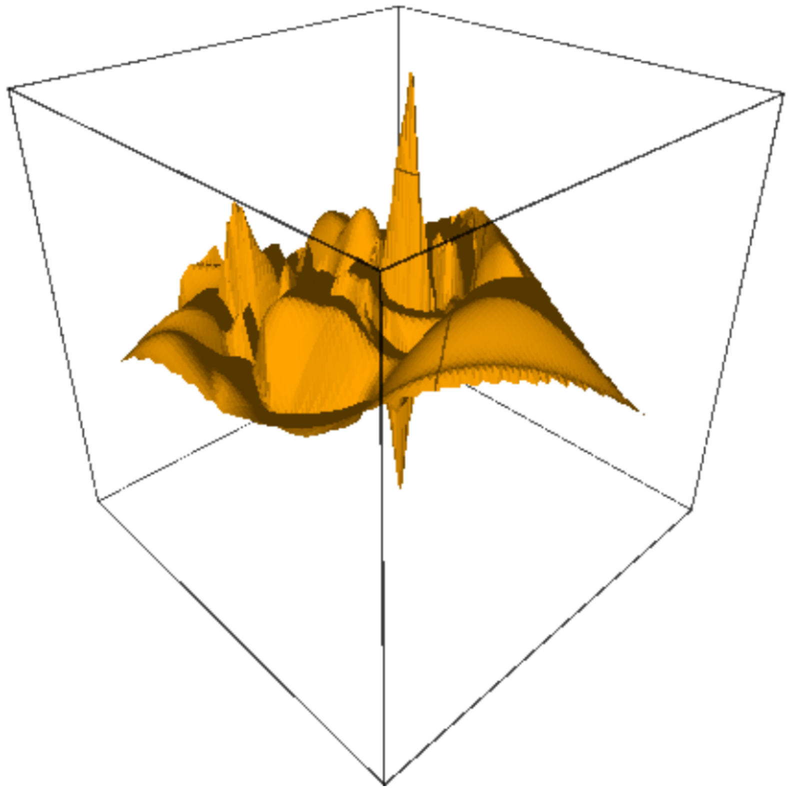
(continues on next page)





(continued from previous page)

```
....:     for j in range(-5, 5):
....:         l.append((normalvariate(0, 1),
....:                   normalvariate(0, 1),
....:                   normalvariate(0, 1)))
sage: L = list_plot3d(l, interpolation_type='clough',
....:                 color='orange', num_points=100)
sage: L
Graphics3d Object
```



Check that no NaNs are produced (see github issue #13135):

```
sage: any(math.isnan(c) for v in L.vertices() for c in v)
False
```

sage.plot.plot3d.list_plot3d.**list_plot3d_array_of_arrays**(*v*, *interpolation_type*, ***kwds*)

A 3-dimensional plot of a surface defined by a list of lists v defining points in 3-dimensional space.

This is done by making the list of lists into a matrix and passing back to *list_plot3d()*. See *list_plot3d()* for full details.

INPUT:

- v - a list of lists, all the same length
- interpolation_type - (default: 'linear')

OPTIONAL KEYWORDS:

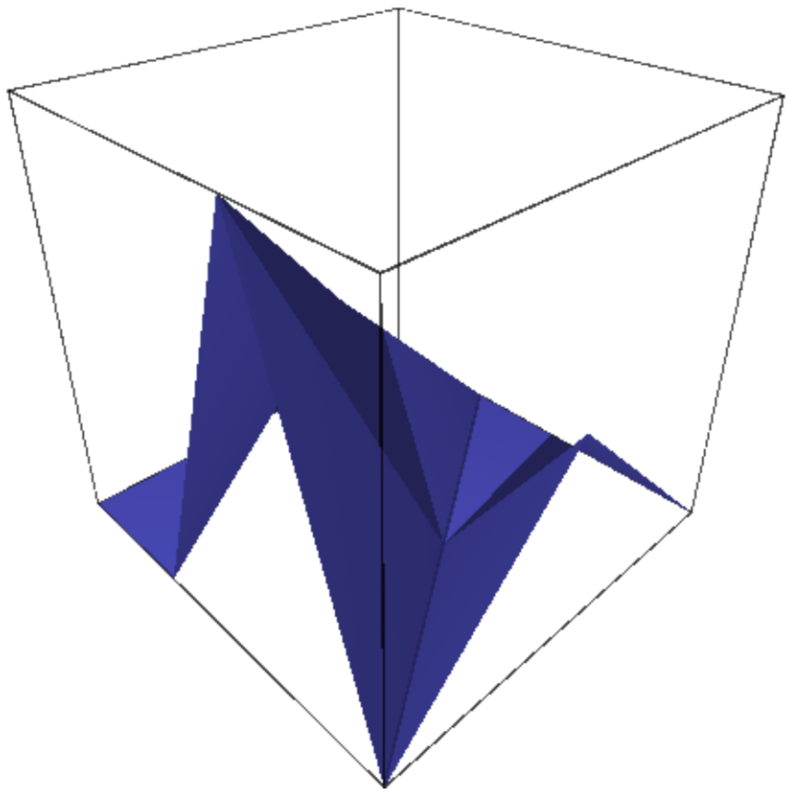
- `**kwds` - all other arguments are passed to the surface function

OUTPUT: a 3d plot

EXAMPLES:

The resulting matrix does not have to be square:

```
sage: show(list_plot3d([[1, 1, 1, 1], [1, 2, 1, 2], [1, 1, 3, 1]])) # indirect␣
↪doctest
```



The normal route is for the list of lists to be turned into a matrix and use *list_plot3d_matrix()*:

```
sage: show(list_plot3d([[1, 1, 1, 1], [1, 2, 1, 2], [1, 1, 3, 1], [1, 2, 1, 4]]))
```

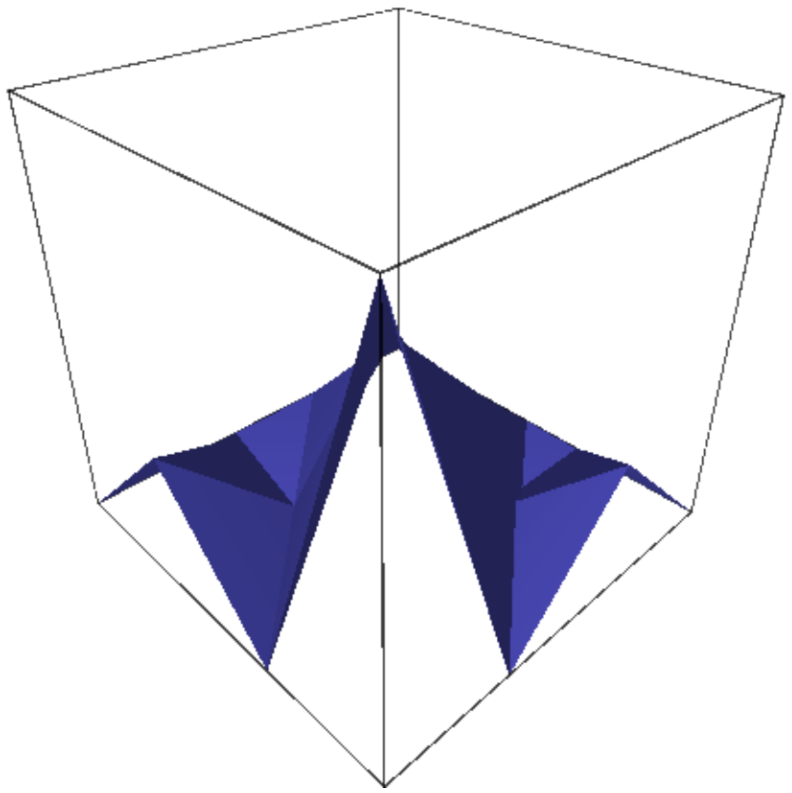
With certain extra keywords (see *list_plot3d_matrix()*), this function will end up using *list_plot3d_tuples()*:

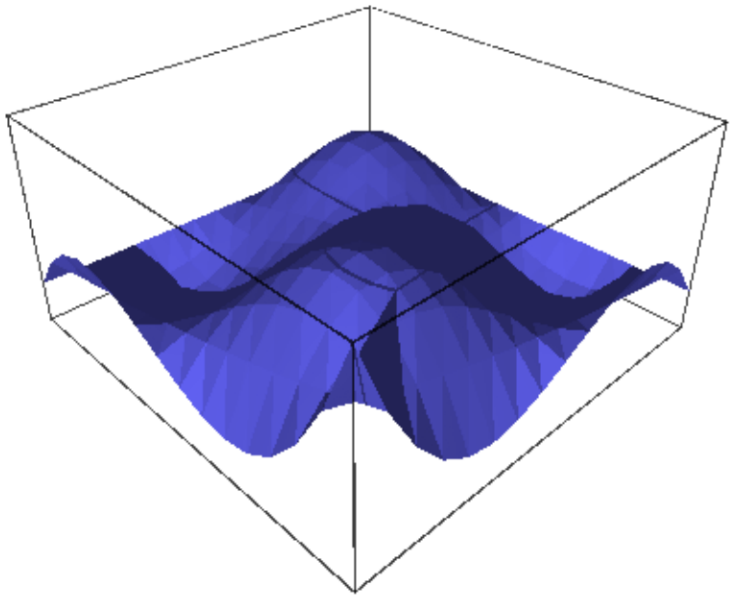
```
sage: show(list_plot3d([[1, 1, 1, 1], [1, 2, 1, 2], [1, 1, 3, 1], [1, 2, 1, 4]],
....:                  interpolation_type='spline'))
```

sage.plot.plot3d.list_plot3d.**list_plot3d_matrix**(*m*, ***kwds*)

A 3-dimensional plot of a surface defined by a matrix `M` defining points in 3-dimensional space.

See *list_plot3d()* for full details.





INPUT:

- `M` – a matrix

OPTIONAL KEYWORDS:

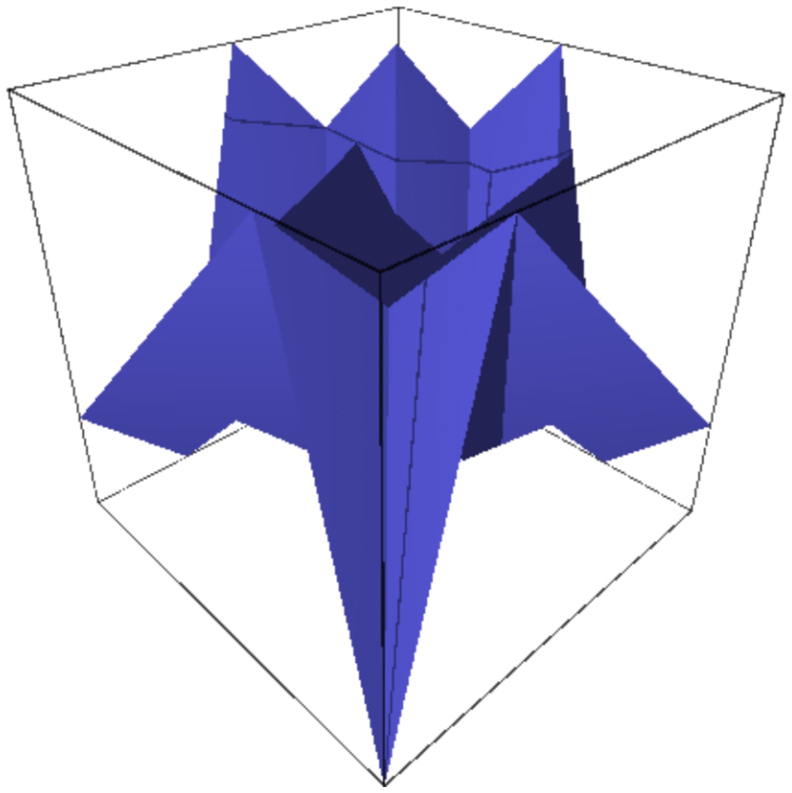
- `**kwds` - all other arguments are passed to the surface function

OUTPUT: a 3d plot

EXAMPLES:

We plot a matrix that illustrates summation modulo n:

```
sage: n = 5
sage: list_plot3d(matrix(RDF, n, [(i+j) % n         # indirect doctest
....:                             for i in [1..n] for j in [1..n]]))
Graphics3d Object
```



The interpolation type for matrices is 'linear'; for other types use other *list_plot3d()* input types.

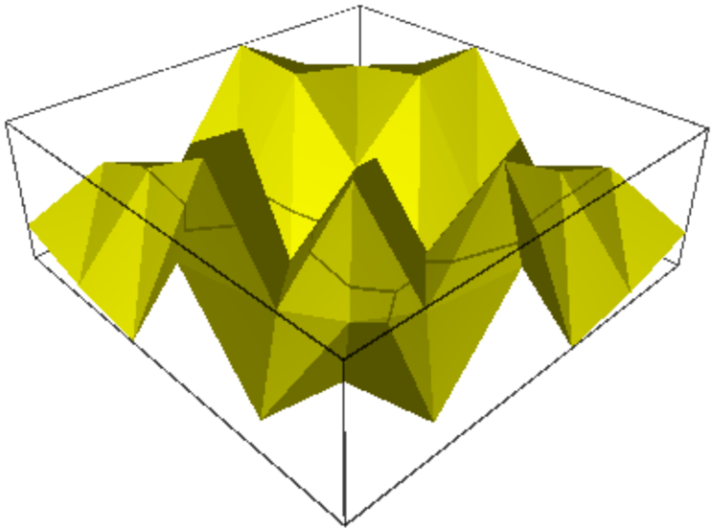
We plot a matrix of values of sin:

```
sage: from math import pi
sage: m = matrix(RDF, 6, [sin(i^2 + j^2)
....:                     for i in [0,pi/5,..,pi] for j in [0,pi/5,..,pi]])
sage: list_plot3d(m, color='yellow', frame_aspect_ratio=[1, 1, 1/3])  # indirect␣
```

(continues on next page)

(continued from previous page)

```
↪doctest
Graphics3d Object
```



::

sage: list_plot3d(m, color='yellow', interpolation_type='linear') # indirect doctest Graphics3d Object

Here is a colored example, using a colormap and a coloring function which must take values in (0, 1):

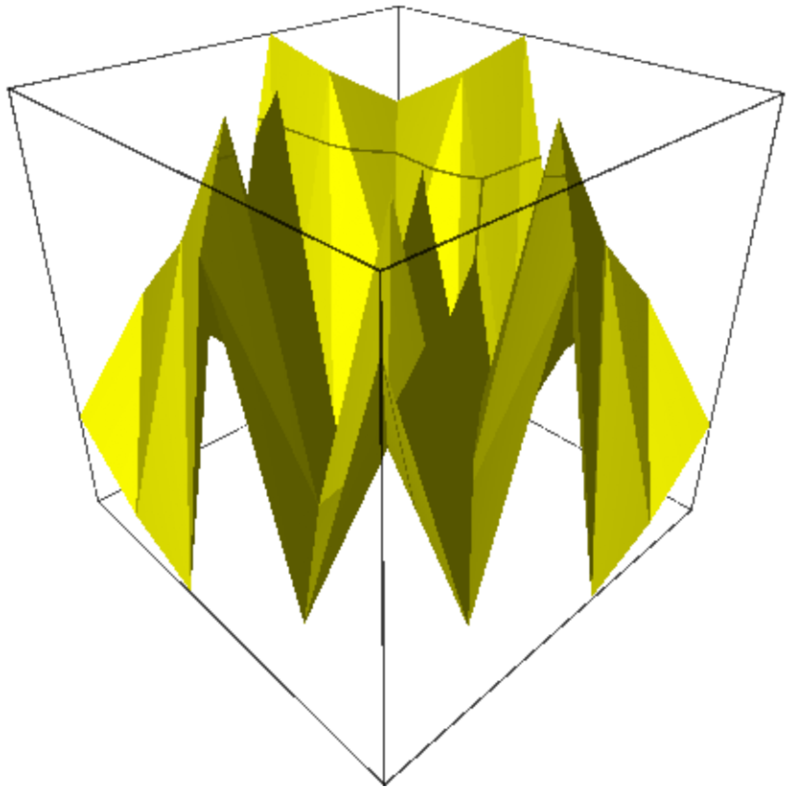
```
sage: cm = colormaps.rainbow
sage: n = 20
sage: cf = lambda x, y: ((2*(x-y)/n)**2) % 1
sage: list_plot3d(matrix(RDF, n, [cos(pi*(i+j)/n) for i in [1..n]
....:                             for j in [1..n]]), color=(cf,cm))
Graphics3d Object
```

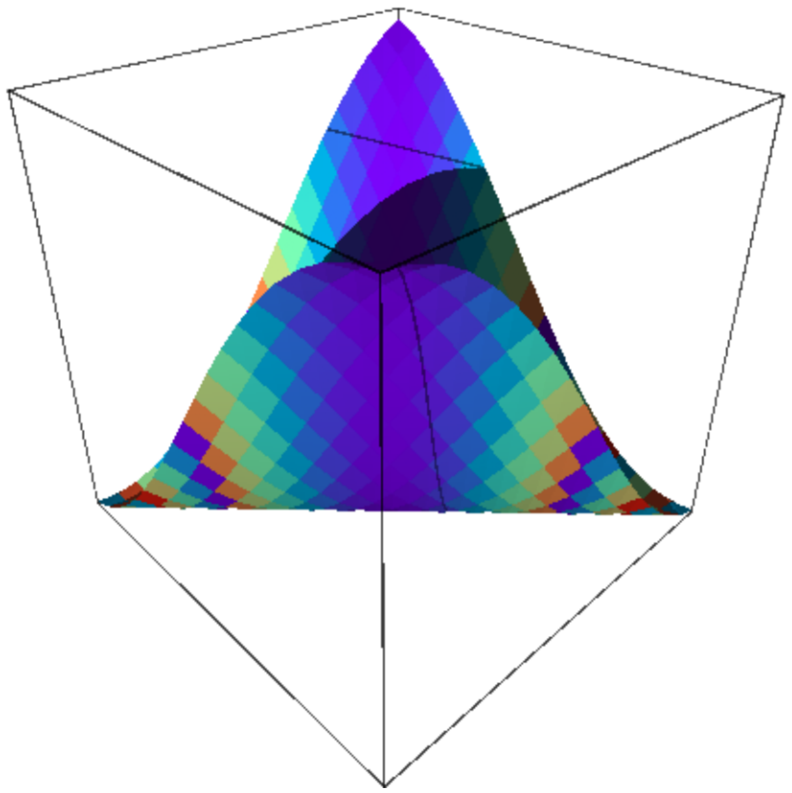
`sage.plot.plot3d.list_plot3d.`**`list_plot3d_tuples`**(*v*, *interpolation_type*, ***kwds*)

A 3-dimensional plot of a surface defined by the list v of points in 3-dimensional space.

INPUT:

- v – something that defines a set of points in 3 space, for example:
  - a matrix





This will be if using an `interpolation_type` other than `'linear'`, or if using `num_points` with `'linear'`; otherwise see *list_plot3d_matrix()*.

- a list of 3-tuples
- a list of lists (all of the same length, under same conditions as a matrix)

OPTIONAL KEYWORDS:

- `interpolation_type` – one of `'linear'`, `'clough'` (CloughTocher2D), `'spline'`

  `'linear'` will perform linear interpolation

  The option 'clough' will interpolate by using a piecewise cubic interpolating Bezier polynomial on each triangle, using a Clough-Tocher scheme. The interpolant is guaranteed to be continuously differentiable.

  The option `'spline'` interpolates using a bivariate B-spline.

  When `v` is a matrix the default is to use linear interpolation, when `v` is a list of points the default is `'clough'`.

- `degree` – an integer between 1 and 5, controls the degree of spline used for spline interpolation. For data that is highly oscillatory use higher values
- `point_list` – If `point_list=True` is passed, then if the array is a list of lists of length three, it will be treated as an array of points rather than a $3 \times n$ array.
- `num_points` – Number of points to sample interpolating function in each direction. By default for an $n \times n$ array this is n.
- `**kwds` – all other arguments are passed to the surface function

OUTPUT: a 3d plot

EXAMPLES:

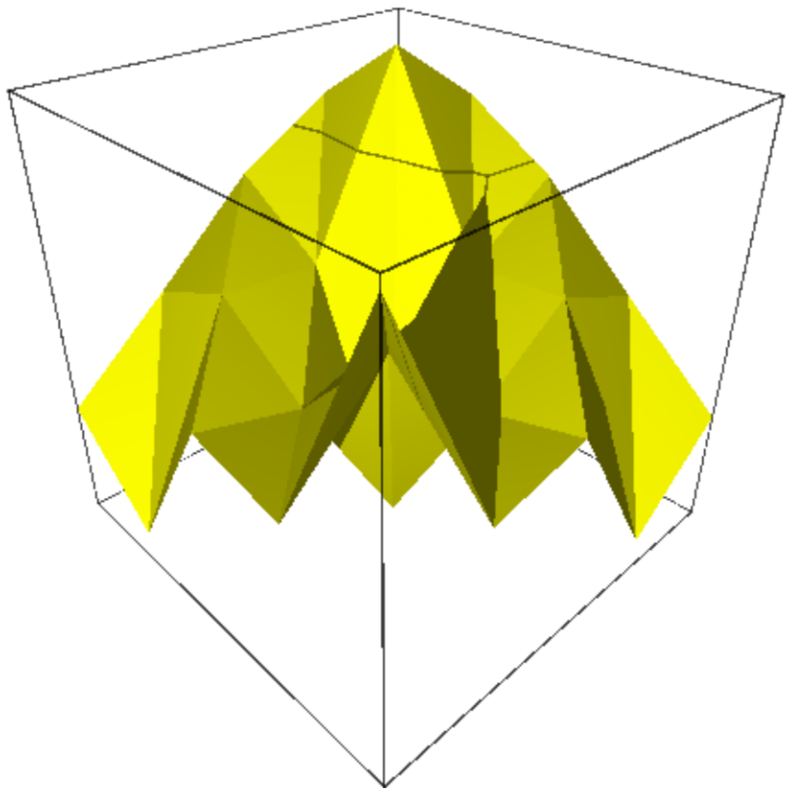
All of these use this function; see *list_plot3d()* for other list plots:

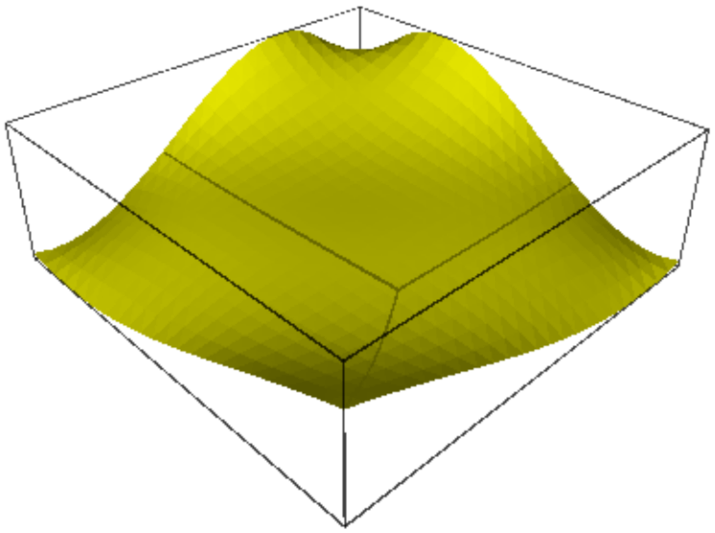
```
sage: from math import pi
sage: m = matrix(RDF, 6, [sin(i^2 + j^2)
....:                     for i in [0,pi/5,..,pi] for j in [0,pi/5,..,pi]])
sage: list_plot3d(m, color='yellow', interpolation_type='linear',  # indirect␣
↪doctest
....:             num_points=5)
Graphics3d Object
```

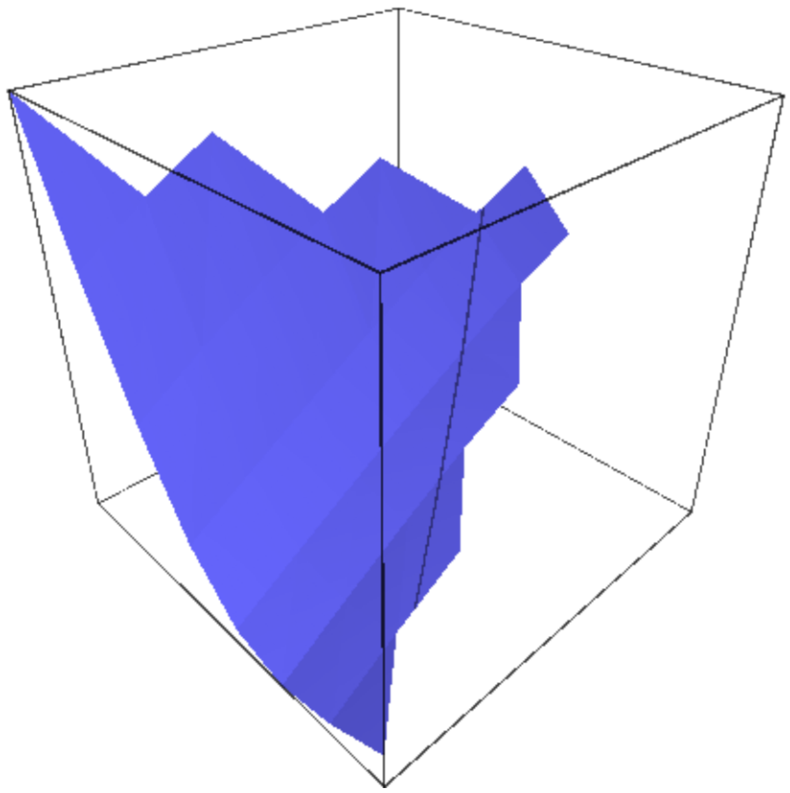
```
sage: list_plot3d(m, color='yellow', interpolation_type='spline',
....:             frame_aspect_ratio=[1, 1, 1/3])
Graphics3d Object
```

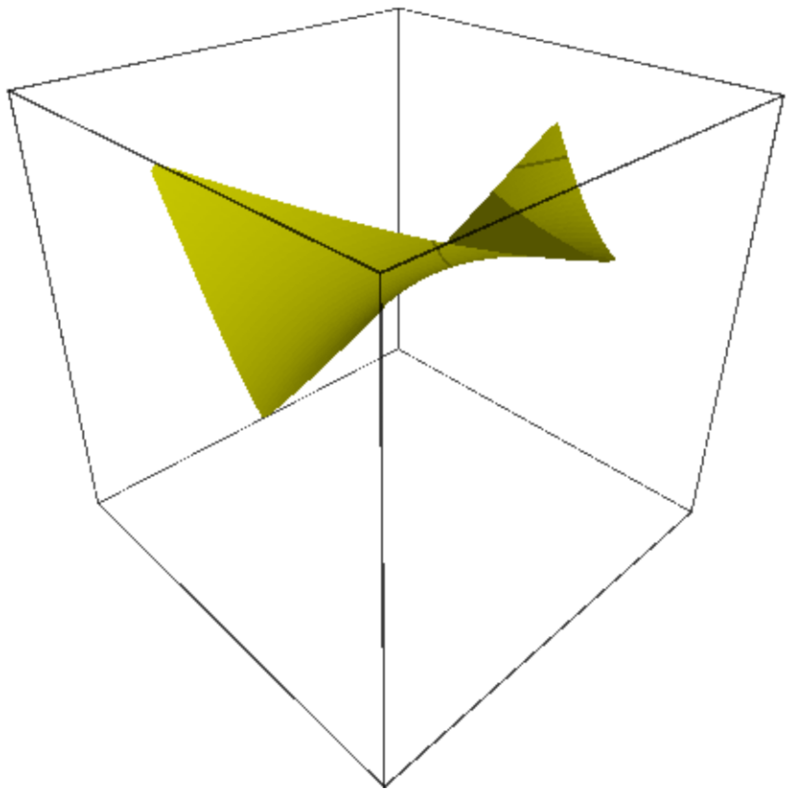
```
sage: show(list_plot3d([[1, 1, 1], [1, 2, 1], [0, 1, 3], [1, 0, 4]],
....:                  point_list=True))
```

```
sage: list_plot3d([(1, 2, 3), (0, 1, 3), (2, 1, 4), (1, 0, -2)],  # long time
....:             color='yellow', num_points=50)
Graphics3d Object
```









CHAPTER THREE

BASIC SHAPES AND PRIMITIVES

3.1 Base classes for 3D graphics objects and plotting

The most important facts about these classes are that you can simply add graphics objects together (G1+G2, see *Graphics3d.__add__()*), and the *Graphics3d.show()* method with its options for choosing a viewer and setting various parameters for displaying the graphics.

Most of the other methods of these classes are technical and for special usage.

AUTHORS:

- Robert Bradshaw (2007-02): initial version
- Robert Bradshaw (2007-08): Cythonization, much optimization
- William Stein (2008)
- Paul Masson (2016): Three.js support
- Joshua Campbell (2020): Three.js animation support
- Günter Rote (2021): camera and light parameters for tachyon

---

Todo: finish integrating tachyon – good default lights

full documentation of three.js viewer parameters

zoom by changing camera parameters instead of scaling objects

---

class sage.plot.plot3d.base.**BoundingSphere**(*cen*, *r*)

Bases: SageObject

A bounding sphere is like a bounding box, but is simpler to deal with and behaves better under rotations.

transform(*T*)

Return the bounding sphere of this sphere acted on by T. This always returns a new sphere, even if the resulting object is an ellipsoid.

EXAMPLES:

```
sage: from sage.plot.plot3d.transform import Transformation
sage: from sage.plot.plot3d.base import BoundingSphere
sage: BoundingSphere((0,0,0), 10).transform(Transformation(trans=(1,2,3)))
Center (1.0, 2.0, 3.0) radius 10.0
sage: BoundingSphere((0,0,0), 10).transform(Transformation(scale=(1/2, 1, 2)))
```

(continues on next page)

(continued from previous page)

```
Center (0.0, 0.0, 0.0) radius 20.0
sage: BoundingSphere((0,0,3), 10).transform(Transformation(scale=(2, 2, 2)))
Center (0.0, 0.0, 6.0) radius 20.0
```

class sage.plot.plot3d.base.**Graphics3d**

Bases: SageObject

This is the baseclass for all 3d graphics objects.

__add__(*left*, *right*)

Addition of objects adds them to the same scene.

EXAMPLES:

```
sage: A = sphere((0,0,0), 1, color='red')
sage: B = dodecahedron((2, 0, 0), color='yellow')
sage: A+B
Graphics3d Object
```

For convenience, we take 0 and `None` to be the additive identity:

```
sage: A + 0 is A
True
sage: A + None is A, 0 + A is A, None + A is A
(True, True, True)
```

In particular, this allows us to use the sum() function without having to provide an empty starting object:

```
sage: sum(point3d((cos(n), sin(n), n)) for n in [0..10, step=.1])
Graphics3d Object
```

A Graphics 3d object and a 2d object can also be added:

```
sage: A = sphere((0, 0, 0), 1) + circle((0, 0), 1.5)
sage: A.show(aspect_ratio=1)
```

_rich_repr_(*display_manager*, ***kwds*)

Rich Output Magic Method

See `sage.repl.rich_output` for details.

EXAMPLES:

```
sage: from sage.repl.rich_output import get_display_manager
sage: dm = get_display_manager()
sage: g = sphere()
sage: g._rich_repr_(dm)  # OutputSceneThreejs container outside doctest mode
OutputSceneJmol container
```

amf_ascii_string(*name='surface'*)

Return an AMF (Additive Manufacturing File Format) representation of the surface.

Warning: This only works for triangulated surfaces!

INPUT:

- `name` (string, default: "surface") – name of the surface.

OUTPUT:

A string that represents the surface in the AMF format.

See Wikipedia article Additive_Manufacturing_File_Format

Todo: This should rather be saved as a ZIP archive to save space.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: x,y,z = var('x,y,z')
sage: a = implicit_plot3d(x^2+y^2+z^2-9,[x,-5,5],[y,-5,5],[z,-5,5])
sage: a_amf = a.amf_ascii_string()
sage: a_amf[:160]
'<?xml version="1.0" encoding="utf-8"?><amf><object id="surface"><mesh>
↪<vertices><vertex><coordinates><x>2.948717948717948</x><y>-0.384615384615385</
↪y><z>-0.3935'

sage: p = polygon3d([[0,0,0], [1,2,3], [3,0,0]])
sage: print(p.amf_ascii_string(name='triangle'))
<?xml version="1.0" encoding="utf-8"?><amf><object id="triangle"><mesh>
↪<vertices><vertex><coordinates><x>0.0</x><y>0.0</y><z>0.0</z></coordinates></
↪vertex><vertex><coordinates><x>1.0</x><y>2.0</y><z>3.0</z></coordinates></
↪vertex><vertex><coordinates><x>3.0</x><y>0.0</y><z>0.0</z></coordinates></
↪vertex></vertices><volume><triangle><v1>0</v1><v2>1</v2><v3>2</v3></triangle>
↪</volume></mesh></object></amf>
```

aspect_ratio(*v=None*)

Set or get the preferred aspect ratio.

INPUT:

- `v` – (default: `None`) must be a list or tuple of length three, or the integer 1. If no arguments are provided then the default aspect ratio is returned.

EXAMPLES:

```
sage: D = dodecahedron()
sage: D.aspect_ratio()
[1.0, 1.0, 1.0]
sage: D.aspect_ratio([1,2,3])
sage: D.aspect_ratio()
[1.0, 2.0, 3.0]
sage: D.aspect_ratio(1)
sage: D.aspect_ratio()
[1.0, 1.0, 1.0]
```

bounding_box()

Return the lower and upper corners of a 3d bounding box.

This is used for rendering, and the scene should fit entirely within this box.

Specifically, the first point returned has x, y, and z coordinates that are the respective minimum over all points in the graphics, and the second point is the maximum.

The default return value is simply the box containing the origin.

EXAMPLES:

```
sage: sphere((1,1,1), 2).bounding_box()
((-1.0, -1.0, -1.0), (3.0, 3.0, 3.0))
sage: G = line3d([(1, 2, 3), (-1,-2,-3)])
sage: G.bounding_box()
((-1.0, -2.0, -3.0), (1.0, 2.0, 3.0))
```

default_render_params()

Return an instance of RenderParams suitable for plotting this object.

EXAMPLES:

```
sage: type(dodecahedron().default_render_params())
<class 'sage.plot.plot3d.base.RenderParams'>
```

export_jmol(*filename='jmol_shape.jmol'*, *force_reload=False*, *zoom=1*, *spin=False*, *background=(1, 1, 1)*, *stereo=False*, *mesh=False*, *dots=False*, *perspective_depth=True*, *orientation=(-764, -346, -545, 76.39)*, ***ignored_kwds*)

A jmol scene consists of a script which refers to external files. Fortunately, we are able to put all of them in a single zip archive, which is the output of this call.

EXAMPLES:

```
sage: out_file = tmp_filename(ext=".jmol")
sage: G = sphere((1, 2, 3), 5) + cube() + sage.plot.plot3d.shapes.Text("hi")
sage: G.export_jmol(out_file)
sage: import zipfile
sage: z = zipfile.ZipFile(out_file)
sage: z.namelist()
['obj_...pmesh', 'SCRIPT']

sage: print(z.read('SCRIPT').decode('ascii'))
data "model list"
2
empty
Xx 0 0 0
Xx 5.5 5.5 5.5
end "model list"; show data
select *
wireframe off; spacefill off
set labelOffset 0 0
background [255,255,255]
spin OFF
moveto 0 -764 -346 -545 76.39
centerAt absolute {0 0 0}
zoom 100
frank OFF
set perspectivedepth ON
isosurface sphere_1  center {1.0 2.0 3.0} sphere 5.0
```

(continues on next page)

(continued from previous page)

```
color isosurface  [102,102,255]
pmesh obj_... "obj_...pmesh"
color pmesh  [102,102,255]
select atomno = 1
color atom  [102,102,255]
label "hi"
isosurface fullylit; pmesh o* fullylit; set antialiasdisplay on;

sage: print(z.read(z.namelist()[0]).decode('ascii'))
24
0.5 0.5 0.5
-0.5 0.5 0.5
...
-0.5 -0.5 -0.5
6
5
0
1
...
```

flatten()

Try to reduce the depth of the scene tree by consolidating groups and transformations.

The generic Graphics3d object cannot be made flatter.

EXAMPLES:

```
sage: G = sage.plot.plot3d.base.Graphics3d()
sage: G.flatten() is G
True
```

frame_aspect_ratio(*v=None*)

Set or get the preferred frame aspect ratio.

INPUT:

- v – (default: None) must be a list or tuple of length three, or the integer 1. If no arguments are provided then the default frame aspect ratio is returned.

EXAMPLES:

```
sage: D = dodecahedron()
sage: D.frame_aspect_ratio()
[1.0, 1.0, 1.0]
sage: D.frame_aspect_ratio([2,2,1])
sage: D.frame_aspect_ratio()
[2.0, 2.0, 1.0]
sage: D.frame_aspect_ratio(1)
sage: D.frame_aspect_ratio()
[1.0, 1.0, 1.0]
```

jmol_repr(*render_params*)

A (possibly nested) list of strings which will be concatenated and used by jmol to render the object.

(Nested lists of strings are used because otherwise all the intermediate concatenations can kill performance). This may refer to several remove files, which are stored in render_parames.output_archive.

EXAMPLES:

```
sage: G = sage.plot.plot3d.base.Graphics3d()
sage: G.jmol_repr(G.default_render_params())
[]
sage: G = sphere((1, 2, 3))
sage: G.jmol_repr(G.default_render_params())
[['isosurface sphere_1  center {1.0 2.0 3.0} sphere 1.0\ncolor isosurface  [102,
↪102,255]']]
```

json_repr(*render_params*)

A (possibly nested) list of strings. Each entry is formatted as JSON, so that a JavaScript client could eval it and get an object. Each object has fields to encapsulate the faces and vertices of the object. This representation is intended to be consumed by the canvas3d viewer backend.

EXAMPLES:

```
sage: G = sage.plot.plot3d.base.Graphics3d()
sage: G.json_repr(G.default_render_params())
[]
```

mtl_str()

Return the contents of a .mtl file, to be used to provide coloring information for an .obj file.

EXAMPLES:

```
sage: G = tetrahedron(color='red') + tetrahedron(color='yellow', opacity=0.5)
sage: print(G.mtl_str())
newmtl ...
Ka 0.5 5e-06 5e-06
Kd 1.0 1e-05 1e-05
Ks 0.0 0.0 0.0
illum 1
Ns 1.0
d 1.0
newmtl ...
Ka 0.5 0.5 5e-06
Kd 1.0 1.0 1e-05
Ks 0.0 0.0 0.0
illum 1
Ns 1.0
d 0.5
```

obj()

An .obj scene file (as a string) containing the this object.

A .mtl file of the same name must also be produced for coloring.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import ColorCube
sage: print(ColorCube(1, ['red', 'yellow', 'blue']).obj())
g obj_1
usemtl ...
v 1 1 1
```

(continues on next page)

(continued from previous page)

```
v -1 1 1
v -1 -1 1
v 1 -1 1
f 1 2 3 4
...
g obj_6
usemtl ...
v -1 -1 1
v -1 1 1
v -1 1 -1
v -1 -1 -1
f 21 22 23 24
```

obj_repr(*render_params*)

A (possibly nested) list of strings which will be concatenated and used to construct an .obj file of the object.

(Nested lists of strings are used because otherwise all the intermediate concatenations can kill performance). This may include a reference to color information which is stored elsewhere.

EXAMPLES:

```
sage: G = sage.plot.plot3d.base.Graphics3d()
sage: G.obj_repr(G.default_render_params())
[]
sage: G = cube()
sage: G.obj_repr(G.default_render_params())
['g obj_1',
 'usemtl ...',
 ['v 0.5 0.5 0.5',
  'v -0.5 0.5 0.5',
  'v -0.5 -0.5 0.5',
  'v 0.5 -0.5 0.5',
  'v 0.5 0.5 -0.5',
  'v -0.5 0.5 -0.5',
  'v 0.5 -0.5 -0.5',
  'v -0.5 -0.5 -0.5'],
 ['f 1 2 3 4',
  'f 1 5 6 2',
  'f 1 4 7 5',
  'f 6 5 7 8',
  'f 7 4 3 8',
  'f 3 2 6 8'],
 []]
```

plot()

Draw a 3D plot of this graphics object, which just returns this object since this is already a 3D graphics object. Needed to support PLOT in docstrings, see github issue #17498

EXAMPLES:

```
sage: S = sphere((0,0,0), 2)
sage: S.plot() is S
True
```

ply_ascii_string(*name='surface'*)

Return a PLY (Polygon File Format) representation of the surface.

INPUT:

- name (string, default: "surface") – name of the surface.

OUTPUT:

A string that represents the surface in the PLY format.

See Wikipedia article PLY_(file_format)

EXAMPLES:

```
sage: # needs sage.symbolic
sage: x,y,z = var('x,y,z')
sage: a = implicit_plot3d(x^2+y^2+z^2-9,[x,-5,5],[y,-5,5],[z,-5,5])
sage: astl = a.ply_ascii_string()
sage: astl.splitlines()[:10]
['ply',
'format ascii 1.0',
'comment surface',
'element vertex 15540',
'property float x',
'property float y',
'property float z',
'element face 5180',
'property list uchar int vertex_indices',
'end_header']

sage: p = polygon3d([[0,0,0], [1,2,3], [3,0,0]])
sage: print(p.ply_ascii_string(name='triangle'))
ply
format ascii 1.0
comment triangle
element vertex 3
property float x
property float y
property float z
element face 1
property list uchar int vertex_indices
end_header
0.0 0.0 0.0
1.0 2.0 3.0
3.0 0.0 0.0
3 0 1 2
```

rotate(*v*, *theta*)

Return the object rotated about the vector v by θ radians.

EXAMPLES:

```
sage: from math import pi
sage: from sage.plot.plot3d.shapes import Cone
sage: v = (1,2,3)
sage: G = arrow3d((0, 0, 0), v)
```

(continues on next page)

(continued from previous page)

```
sage: G += Cone(1/5, 1).translate((0, 0, 2))
sage: C = Cone(1/5, 1, opacity=.25).translate((0, 0, 2))
sage: G += sum(C.rotate(v, pi*t/4) for t in [1..7])
sage: G.show(aspect_ratio=1)

sage: from sage.plot.plot3d.shapes import Box
sage: Box(1/3, 1/5, 1/7).rotate((1, 1, 1), pi/3).show(aspect_ratio=1)
```

rotateX(*theta*)

Return the object rotated about the x-axis by the given angle.

EXAMPLES:

```
sage: from math import pi
sage: from sage.plot.plot3d.shapes import Cone
sage: G = Cone(1/5, 1) + Cone(1/5, 1, opacity=.25).rotateX(pi/2)
sage: G.show(aspect_ratio=1)
```

rotateY(*theta*)

Return the object rotated about the y-axis by the given angle.

EXAMPLES:

```
sage: from math import pi
sage: from sage.plot.plot3d.shapes import Cone
sage: G = Cone(1/5, 1) + Cone(1/5, 1, opacity=.25).rotateY(pi/3)
sage: G.show(aspect_ratio=1)
```

rotateZ(*theta*)

Return the object rotated about the z-axis by the given angle.

EXAMPLES:

```
sage: from math import pi
sage: from sage.plot.plot3d.shapes import Box
sage: G = Box(1/2, 1/3, 1/5) + Box(1/2, 1/3, 1/5, opacity=.25).rotateZ(pi/5)
sage: G.show(aspect_ratio=1)
```

save(*filename*, ***kwds*)

Save the graphic in a file.

The file type depends on the file extension you give in the filename. This can be either:

- an image file (of type: PNG, BMP, GIF, PPM, or TIFF) rendered using Jmol (default) or Tachyon,
- a Sage object file (of type `.sobj`) that you can load back later (a pickle),
- an HTML file depicting the graphic using the Three.js viewer,
- a data file (of type: X3D, STL, AMF, PLY) for export and use in other software.

For data files, the support is only partial. For instance STL and AMF only works for triangulated surfaces. The prefered format is X3D.

INPUT:

- `filename` – string. Where to save the image or object.

- `**kwds` – When specifying an image file to be rendered by Tachyon or Jmol, any of the viewing options accepted by *show()* are valid as keyword arguments to this function and they will behave in the same way. Accepted keywords include: `viewer`, `verbosity`, `figsize`, `aspect_ratio`, `frame_aspect_ratio`, `zoom`, `frame`, and `axes`. Default values are provided.

EXAMPLES:

```
sage: f = tmp_filename(ext='.png')
sage: G = sphere()
sage: G.save(f)
```

We demonstrate using keyword arguments to control the appearance of the output image:

```
sage: G.save(f, zoom=2, figsize=[5, 10])
```

Using Tachyon instead of the default viewer (Jmol) to create the image:

```
sage: G.save(f, viewer='tachyon')
```

Since Tachyon only outputs PNG images, PIL will be used to convert to alternate formats:

```
sage: cube().save(tmp_filename(ext='.gif'), viewer='tachyon')
```

Here is how to save in one of the data formats:

```
sage: f = tmp_filename(ext='.x3d')
sage: cube().save(f)

sage: open(f).read().splitlines()[7]
"<Shape><Box size='0.5 0.5 0.5'/><Appearance><Material diffuseColor='0.4 0.4 1.0
↪' shininess='1.0' specularColor='0.0 0.0 0.0'/></Appearance></Shape>"
```

Producing a Three.js-based HTML file:

```
sage: f = tmp_filename(ext='.html')
sage: G.save(f, frame=False, online=True)
```

save_image(*filename*, ***kwds*)

Save a 2-D image rendering.

The image type is determined by the extension of the filename. For example, this could be `.png`, `.jpg`, `.gif`, `.pdf`, `.svg`.

INPUT:

- `filename` – string. The file name under which to save the image.

Any further keyword arguments are passed to the renderer.

EXAMPLES:

```
sage: G = sphere()
sage: png = tmp_filename(ext='.png')
sage: G.save_image(png)
sage: with open(png, 'rb') as fobj:
....:     assert fobj.read().startswith(b'\x89PNG')

```

(continues on next page)

(continued from previous page)

```
sage: gif = tmp_filename(ext='.gif')
sage: G.save_image(gif)
sage: with open(gif, 'rb') as fobj:
....:     assert fobj.read().startswith(b'GIF')
```

`scale`(**x*)

Return the object scaled in the x, y, and z directions.

EXAMPLES:

```
sage: G = dodecahedron() + dodecahedron(opacity=.5).scale(2)
sage: G.show(aspect_ratio=1)
sage: G = icosahedron() + icosahedron(opacity=.5).scale([1, 1/2, 2])
sage: G.show(aspect_ratio=1)
```

`show`(***kwds*)

Display graphics immediately

This method attempts to display the graphics immediately, without waiting for the currently running code (if any) to return to the command line. Be careful, calling it from within a loop will potentially launch a large number of external viewer programs.

INPUT:

- `viewer` – string (default: `'threejs'`), how to view the plot; admissible values are
  - `'threejs'`: interactive web-based 3D viewer using JavaScript and a WebGL renderer
  - `'jmol'`: interactive 3D viewer using Java
  - `'tachyon'`: ray tracer generating a static PNG image; can produce high-resolution graphics, but does not show any text labels
  - `'canvas3d'`: web-based 3D viewer using JavaScript and a canvas renderer (Sage notebook only)
- `verbosity` – display information about rendering the figure
- `figsize` – (default: 5); x or pair [x,y] for numbers, e.g., [5,5]; controls the size of the output figure. With 'tachyon', the resolution (in number of pixels) is 100 times `figsize`. This is ignored for the jmol embedded renderer.
- `aspect_ratio` – (default: `'automatic'`) – aspect ratio of the coordinate system itself. Give [1,1,1] or 1 to make spheres look round.
- `frame_aspect_ratio` – (default: `'automatic'`) aspect ratio of frame that contains the 3d scene.
- `zoom` – (default: 1) how zoomed in
- `frame` – (default: True) if True, draw a bounding frame with labels
- `axes` – (default: False) if True, draw coordinate axes
- `camera_position` (for tachyon) – (default: (2.3, 2.4, 2.0)) the viewpoint, with respect to the cube $[-1,1]\times[-1,1]\times[-1,1]$, into which the bounding box of the scene is scaled and centered. The default viewing direction is towards the origin.
- `viewdir` (for tachyon) – (default: None) three coordinates specifying the viewing direction.
- `updir` (for tachyon) – (default: (0,0,1)) the "upward" direction of the camera
- `light_position` (for tachyon) – (default: (4,3,2)) the position of the single light source in the scene (in addition to ambient light)

- antialiasing (for tachyon) – (default: False)
- raydepth (for tachyon) – (default: 8) see the *sage.plot.plot3d.tachyon.Tachyon* class
- shade (for tachyon) – string (default: 'full'); shading options. Admissible values are
  - 'full': best quality rendering (and slowest). Sets tachyon command line flag -fullshade.
  - 'medium: good quality rendering, but no shadows. Sets tachyon command line flag -mediumshade.
  - 'low': low quality rendering, preview (and fast). Sets tachyon command line flag -lowshade.
  - 'lowest': worst quality rendering, preview (and fastest). Sets tachyon command line flag -lowestshade.
- extra_opts (for tachyon) – string (default: empty string); extra options that will be appended to the tachyon command line.
- **kwds – other options, which make sense for particular rendering engines

OUTPUT:

This method does not return anything. Use *save()* if you want to save the figure as an image file.

Warning: By default, the jmol and tachyon viewers perform some non-uniform scaling of the axes.

If this is not desired, one can set aspect_ratio=1:

```
sage: p = plot3d(lambda u,v:(cos(u)-cos(v)), (-0.2,0.2),(-0.2,0.2))
sage: p.show(viewer="threejs")
sage: p.show(viewer="jmol")
sage: p.show(viewer="jmol",aspect_ratio=1)
sage: p.show(viewer="tachyon",camera_position=(4,0,0))
sage: p.show(viewer="tachyon",camera_position=(2,2,0.3),aspect_ratio=1)
```

CHANGING DEFAULTS: Defaults can be uniformly changed by importing a dictionary and changing it. For example, here we change the default so images display without a frame instead of with one:

```
sage: from sage.plot.plot3d.base import SHOW_DEFAULTS
sage: SHOW_DEFAULTS['frame'] = False
```

This sphere will not have a frame around it:

```
sage: sphere((0,0,0))
Graphics3d Object
```

We change the default back:

```
sage: SHOW_DEFAULTS['frame'] = True
```

Now this sphere is enclosed in a frame:

```
sage: sphere((0,0,0))
Graphics3d Object
```

EXAMPLES: We illustrate use of the aspect_ratio option:

```
sage: x, y = var('x,y')                                                    #␣
↪needs sage.symbolic
sage: p = plot3d(2*sin(x*y), (x, -pi, pi), (y, -pi, pi))                   #␣
↪needs sage.symbolic
sage: p.show(aspect_ratio=[1,1,1])                                         #␣
↪needs sage.symbolic
```

This looks flattened, but filled with the plot:

```
sage: p.show(frame_aspect_ratio=[1,1,1/16])                                #␣
↪needs sage.symbolic
```

This looks flattened, but the plot is square and smaller:

```
sage: p.show(aspect_ratio=[1,1,1], frame_aspect_ratio=[1,1,1/8])           #␣
↪needs sage.symbolic
```

This example shows indirectly that the defaults from `plot()` are dealt with properly:

```
sage: plot(vector([1,2,3]))
Graphics3d Object
```

We use the 'canvas3d' backend from inside the notebook to get a view of the plot rendered inline using HTML canvas:

```
sage: p.show(viewer='canvas3d')                                            #␣
↪needs sage.symbolic
```

Sometimes shadows in Tachyon-produced images can lead to confusing plots. To remove them:

```
sage: p.show(viewer="tachyon", shade="medium")                             #␣
↪needs sage.symbolic
```

One can also pass Tachyon command line flags directly. For example, the following line produces the same result as the previous one:

```
sage: p.show(viewer="tachyon", extra_opts="-mediumshade")                  #␣
↪needs sage.symbolic
```

`stl_ascii_string`(*name='surface'*)

Return an STL (STereoLithography) representation of the surface.

Warning: This only works for surfaces, not for general plot objects!

INPUT:

- `name` (string, default: "surface") – name of the surface.

OUTPUT:

A string that represents the surface in the STL format.

See Wikipedia article STL_(file_format)

See also:

stl_binary()

EXAMPLES:

```
sage: # needs sage.symbolic
sage: x,y,z = var('x,y,z')
sage: a = implicit_plot3d(x^2+y^2+z^2-9,[x,-5,5],[y,-5,5],[z,-5,5])
sage: astl = a.stl_ascii_string()
sage: astl.splitlines()[:7]  # abs tol 1e-10
['solid surface',
'facet normal 0.9733285267845754 -0.16222142113076257 -0.16222142113076257',
'    outer loop',
'        vertex 2.94871794872 -0.384615384615 -0.39358974359',
'        vertex 2.95021367521 -0.384615384615 -0.384615384615',
'        vertex 2.94871794872 -0.39358974359 -0.384615384615',
'    endloop']

sage: p = polygon3d([[0,0,0], [1,2,3], [3,0,0]])
sage: print(p.stl_ascii_string(name='triangle'))
solid triangle
facet normal 0.0 0.8320502943378436 -0.5547001962252291
    outer loop
        vertex 0.0 0.0 0.0
        vertex 1.0 2.0 3.0
        vertex 3.0 0.0 0.0
    endloop
endfacet
endsolid triangle
```

Now works when faces have more then 3 sides:

```
sage: P = polytopes.dodecahedron()                                      #␣
↪needs sage.geometry.polyhedron
sage: Q = P.plot().all[-1]                                              #␣
↪needs sage.geometry.polyhedron
sage: print(Q.stl_ascii_string().splitlines()[:7])                      #␣
↪needs sage.geometry.polyhedron
['solid surface',
 'facet normal 0.0 0.5257311121191338 0.8506508083520399',
 '    outer loop',
 '        vertex -0.7639320225002102 0.7639320225002102 0.7639320225002102',
 '        vertex -0.4721359549995796 0.0 1.2360679774997898',
 '        vertex 0.4721359549995796 0.0 1.2360679774997898',
 '    endloop']
```

stl_binary()

Return an STL (STereoLithography) binary representation of the surface.

Warning: This only works for surfaces, transforms and unions of surfaces, but not for general plot objects!

OUTPUT:

A binary string that represents the surface in the binary STL format.

See Wikipedia article STL_(file_format)

See also:

stl_ascii_string()

EXAMPLES:

```
sage: # needs sage.symbolic
sage: x,y,z = var('x,y,z')
sage: a = implicit_plot3d(x^2+y^2+z^2-9,[x,-5,5],[y,-5,5],[z,-5,5])
sage: astl = a.stl_binary()
sage: print(astl[:40].decode('ascii'))
STL binary file / made by SageMath / ###

sage: p = polygon3d([[0,0,0], [1,2,3], [3,0,0]])
sage: print(p.stl_binary()[:40].decode('ascii'))
STL binary file / made by SageMath / ###
```

This works when faces have more then 3 sides:

```
sage: P = polytopes.dodecahedron()                                      #␣
↪needs sage.geometry.polyhedron
sage: Q = P.plot().all[-1]                                              #␣
↪needs sage.geometry.polyhedron
sage: print(Q.stl_binary()[:40].decode('ascii'))                        #␣
↪needs sage.geometry.polyhedron
STL binary file / made by SageMath / ###
```

tachyon(*zoom=1.0*, *antialiasing=False*, *figsize=[5, 5]*, *raydepth=8*, *camera_position=[2.3, 2.4, 2.0]*, *updir=[0, 0, 1]*, *light_position=[4.0, 3.0, 2.0]*, *viewdir=None*)

A tachyon input file (as a string) containing the this object.

EXAMPLES:

```
sage: print(sphere((1, 2, 3), 5, color='yellow').tachyon())

begin_scene
resolution 500 500

         camera
        ...
      plane
        center -592.870151560437 618.647114671761 -515.539262226467
        normal -2.3 2.4 -2.0
        TEXTURE
            AMBIENT 1.0 DIFFUSE 0.0 SPECULAR 0.0 OPACITY 1.0
            COLOR 1.0 1.0 1.0
            TEXFUNC 0

    Texdef texture...
  Ambient 0.3333333333333333 Diffuse 0.6666666666666666 Specular 0.0 Opacity 1.0
  Color 1.0 1.0 0.0
  TexFunc 0
```

(continues on next page)

(continued from previous page)

```
    Sphere center 1.0 -2.0 3.0 Rad 5.0 texture...

end_scene

sage: G = icosahedron(color='red') + sphere((1,2,3), 0.5, color='yellow')
sage: G.show(viewer='tachyon', frame=false)
sage: print(G.tachyon())
begin_scene
...
Texdef texture...
  Ambient 0.3333333333333333 Diffuse 0.6666666666666666 Specular 0.0 Opacity 1.0
   Color 1.0 1.0 0.0
   TexFunc 0
TRI V0 ...
Sphere center 1.0 -2.0 3.0 Rad 0.5 texture...
end_scene
```

tachyon_keywords = ('antialiasing', 'zoom', 'raydepth', 'figsize', 'light_position', 'camera_position', 'updir', 'viewdir')

tachyon_repr(*render_params*)

A (possibly nested) list of strings which will be concatenated and used by tachyon to render the object.

(Nested lists of strings are used because otherwise all the intermediate concatenations can kill performance). This may include a reference to color information which is stored elsewhere.

EXAMPLES:

```
sage: G = sage.plot.plot3d.base.Graphics3d()
sage: G.tachyon_repr(G.default_render_params())
[]
sage: G = sphere((1, 2, 3))
sage: G.tachyon_repr(G.default_render_params())
['Sphere center 1.0 2.0 3.0 Rad 1.0 texture...']
```

testing_render_params()

Return an instance of RenderParams suitable for testing this object.

In particular, it opens up a temporary file as an auxiliary zip file for jmol.

EXAMPLES:

```
sage: type(dodecahedron().testing_render_params())
<class 'sage.plot.plot3d.base.RenderParams'>
```

texture

texture_set()

Often the textures of a 3d file format are kept separate from the objects themselves. This function returns the set of textures used, so they can be defined in a preamble or separate file.

EXAMPLES:

```
sage: sage.plot.plot3d.base.Graphics3d().texture_set()
set()

sage: G = tetrahedron(color='red') + tetrahedron(color='yellow') +␣
↪tetrahedron(color='red', opacity=0.5)
sage: [t for t in G.texture_set() if t.color == colors.red] # we should have␣
↪two red textures
[Texture(texture..., red, ff0000), Texture(texture..., red, ff0000)]
sage: [t for t in G.texture_set() if t.color == colors.yellow] # ...and one␣
↪yellow
[Texture(texture..., yellow, ffff00)]
```

threejs_repr(*render_params*)

A flat list of (kind, desc) tuples where kind is one of: 'point', 'line', 'text', or 'surface'; and where desc is a dictionary describing a point, line, text, or surface.

EXAMPLES:

```
sage: G = sage.plot.plot3d.base.Graphics3d()
sage: G.threejs_repr(G.default_render_params())
[]
```

transform(***kwds*)

Apply a transformation, where the inputs are passed onto a TransformGroup object.

Mostly for internal use; see the translate, scale, and rotate methods for more details.

EXAMPLES:

```
sage: sphere((0,0,0), 1).transform(trans=(1, 0, 0), scale=(2,3,4)).bounding_
↪box()
((-1.0, -3.0, -4.0), (3.0, 3.0, 4.0))
```

translate(**x*)

Return the object translated by the given vector (which can be given either as a 3-iterable or via positional arguments).

EXAMPLES:

```
sage: icosahedron() + sum(icosahedron(opacity=0.25).translate(2*n, 0, 0) for n␣
↪in [1..4])
Graphics3d Object
sage: icosahedron() + sum(icosahedron(opacity=0.25).translate([-2*n, n, n^2])␣
↪for n in [1..4])
Graphics3d Object
```

viewpoint()

Return the viewpoint of this plot.

Currently only a stub for x3d.

EXAMPLES:

```
sage: type(dodecahedron().viewpoint())
<class 'sage.plot.plot3d.base.Viewpoint'>
```

x3d()

An x3d scene file (as a string) containing the this object.

EXAMPLES:

```
sage: print(sphere((1, 2, 3), 5).x3d())
<X3D version='3.0' profile='Immersive' xmlns:xsd='http://www.w3.org/2001/
↪XMLSchema-instance' xsd:noNamespaceSchemaLocation=' http://www.web3d.org/
↪specifications/x3d-3.0.xsd '>
<head>
<meta name='title' content='sage3d'/>
</head>
<Scene>
<Viewpoint position='0 0 6'/>
<Transform translation='1 2 3'>
<Shape><Sphere radius='5.0'/><Appearance><Material diffuseColor='0.4 0.4 1.0'␣
↪shininess='1.0' specularColor='0.0 0.0 0.0'/></Appearance></Shape>
</Transform>
</Scene>
</X3D>

sage: G = icosahedron() + sphere((0,0,0), 0.5, color='red')
sage: print(G.x3d())
<X3D version='3.0' profile='Immersive' xmlns:xsd='http://www.w3.org/2001/
↪XMLSchema-instance' xsd:noNamespaceSchemaLocation=' http://www.web3d.org/
↪specifications/x3d-3.0.xsd '>
<head>
<meta name='title' content='sage3d'/>
</head>
<Scene>
<Viewpoint position='0 0 6'/>
<Shape>
<IndexedFaceSet coordIndex='...'>
  <Coordinate point='...'/>
</IndexedFaceSet>
<Appearance><Material diffuseColor='0.4 0.4 1.0' shininess='1.0' specularColor=
↪'0.0 0.0 0.0'/></Appearance></Shape>
<Transform translation='0 0 0'>
<Shape><Sphere radius='0.5'/><Appearance><Material diffuseColor='1.0 0.0 0.0'␣
↪shininess='1.0' specularColor='0.0 0.0 0.0'/></Appearance></Shape>
</Transform>
</Scene>
</X3D>
```

class sage.plot.plot3d.base.**Graphics3dGroup**(*all=()*, *rot=None*, *trans=None*, *scale=None*, *T=None*)

Bases: *Graphics3d*

This class represents a collection of 3d objects. Usually they are formed implicitly by summing.

bounding_box()

Box that contains the bounding boxes of the objects.

EXAMPLES:

```
sage: A = sphere((0,0,0), 5)
sage: B = sphere((1, 5, 10), 1)
sage: A.bounding_box()
((-5.0, -5.0, -5.0), (5.0, 5.0, 5.0))
sage: B.bounding_box()
((0.0, 4.0, 9.0), (2.0, 6.0, 11.0))
sage: (A+B).bounding_box()
((-5.0, -5.0, -5.0), (5.0, 6.0, 11.0))
sage: (A+B).show(aspect_ratio=1, frame=True)

sage: sage.plot.plot3d.base.Graphics3dGroup([]).bounding_box()
((0.0, 0.0, 0.0), (0.0, 0.0, 0.0))
```

flatten()

Try to reduce the depth of the scene tree by consolidating groups and transformations.

EXAMPLES:

```
sage: G = sum([circle((0, 0), t) for t in [1..10]], sphere()); G
Graphics3d Object
sage: G.flatten()
Graphics3d Object
sage: len(G.all)
2
sage: len(G.flatten().all)
11
```

jmol_repr(*render_params*)

The jmol representation of a group is simply the concatenation of the representation of its objects.

EXAMPLES:

```
sage: G = sphere() + sphere((1,2,3))
sage: G.jmol_repr(G.default_render_params())
[[['isosurface sphere_1  center {0.0 0.0 0.0} sphere 1.0\ncolor isosurface ␣
↪[102,102,255]']],
 [['isosurface sphere_2  center {1.0 2.0 3.0} sphere 1.0\ncolor isosurface ␣
↪[102,102,255]']]]
```

json_repr(*render_params*)

The JSON representation of a group is simply the concatenation of the representations of its objects.

EXAMPLES:

```
sage: G = sphere() + sphere((1, 2, 3))
sage: G.json_repr(G.default_render_params())
[[['{"vertices":...']], [['{"vertices":...']]]
```

obj_repr(*render_params*)

The obj representation of a group is simply the concatenation of the representation of its objects.

EXAMPLES:

```
sage: G = tetrahedron() + tetrahedron().translate(10, 10, 10)
sage: G.obj_repr(G.default_render_params())
[['g obj_1',
  'usemtl ...',
  ['v 0 0 1',
   'v 0.942809 0 -0.333333',
   'v -0.471405 0.816497 -0.333333',
   'v -0.471405 -0.816497 -0.333333'],
  ['f 1 2 3', 'f 2 4 3', 'f 1 3 4', 'f 1 4 2'],
  []],
 [['g obj_2',
   'usemtl ...',
   ['v 10 10 11',
    'v 10.9428 10 9.66667',
    'v 9.5286 10.8165 9.66667',
    'v 9.5286 9.1835 9.66667'],
   ['f 5 6 7', 'f 6 8 7', 'f 5 7 8', 'f 5 8 6'],
   []]]]
```

plot()

set_texture(***kwds*)

EXAMPLES:

```
sage: G = dodecahedron(color='red', opacity=.5) + icosahedron((3, 0, 0), color=
↪'blue')
sage: G
Graphics3d Object
sage: G.set_texture(color='yellow')
sage: G
Graphics3d Object
```

stl_binary_repr(*render_params*)

The stl binary representation of a group is simply the concatenation of the representation of its objects.

The STL binary representation is a list of binary strings, one for each triangle.

EXAMPLES:

```
sage: G = sphere() + sphere((1,2,3))
sage: len(G.stl_binary_repr(G.default_render_params()))
2736
```

tachyon_repr(*render_params*)

The tachyon representation of a group is simply the concatenation of the representations of its objects.

EXAMPLES:

```
sage: G = sphere() + sphere((1,2,3))
sage: G.tachyon_repr(G.default_render_params())
[['Sphere center 0.0 0.0 0.0 Rad 1.0 texture...'],
 ['Sphere center 1.0 2.0 3.0 Rad 1.0 texture...']]
```

texture_set()

The texture set of a group is simply the union of the textures of all its objects.

EXAMPLES:

```
sage: G = sphere(color='red') + sphere(color='yellow')
sage: [t for t in G.texture_set() if t.color == colors.red] # one red texture
[Texture(texture..., red, ff0000)]
sage: [t for t in G.texture_set() if t.color == colors.yellow] # one yellow␣
↪texture
[Texture(texture..., yellow, ffff00)]

sage: T = sage.plot.plot3d.texture.Texture('blue'); T
Texture(texture..., blue, 0000ff)
sage: G = sphere(texture=T) + sphere((1, 1, 1), texture=T)
sage: len(G.texture_set())
1
```

threejs_repr(*render_params*)

The three.js representation of a group is the concatenation of the representations of its objects.

EXAMPLES:

```
sage: G = point3d((1,2,3)) + point3d((4,5,6)) + line3d([(1,2,3), (4,5,6)])
sage: G.threejs_repr(G.default_render_params())
[('point',
  {'color': '#6666ff', 'opacity': 1.0, 'point': (1.0, 2.0, 3.0), 'size': 5.0}),
 ('point',
  {'color': '#6666ff', 'opacity': 1.0, 'point': (4.0, 5.0, 6.0), 'size': 5.0}),
 ('line',
  {'color': '#6666ff',
   'linewidth': 1.0,
   'opacity': 1.0,
   'points': [(1.0, 2.0, 3.0), (4.0, 5.0, 6.0)]})]
```

transform(***kwds*)

Transforming this entire group simply makes a transform group with the same contents.

EXAMPLES:

```
sage: G = dodecahedron(color='red', opacity=.5) + icosahedron(color='blue')
sage: G
Graphics3d Object
sage: G.transform(scale=(2,1/2,1))
Graphics3d Object
sage: G.transform(trans=(1,1,3))
Graphics3d Object
```

x3d_str()

The x3d representation of a group is simply the concatenation of the representation of its objects.

EXAMPLES:

```
sage: G = sphere() + sphere((1,2,3))
sage: print(G.x3d_str())
<Transform translation='0 0 0'>
<Shape><Sphere radius='1.0'/><Appearance><Material diffuseColor='0.4 0.4 1.0'␣
↪shininess='1.0' specularColor='0.0 0.0 0.0'/></Appearance></Shape>
```

(continues on next page)

(continued from previous page)

```
</Transform>
<Transform translation='1 2 3'>
<Shape><Sphere radius='1.0'/><Appearance><Material diffuseColor='0.4 0.4 1.0'␣
↪shininess='1.0' specularColor='0.0 0.0 0.0'/></Appearance></Shape>
</Transform>
```

class sage.plot.plot3d.base.**KeyframeAnimationGroup**(*all=()*, ***kwds*)

Bases: *Graphics3dGroup*

A group of objects, each depicting a single frame of animation

threejs_repr(*render_params*)

Adds keyframe information to the representations of the group's contents.

EXAMPLES:

```
sage: a = point3d((0, 0, 1))
sage: b = point3d((0, 1, 0))
sage: c = point3d((1, 0, 0))
sage: g = sage.plot.plot3d.base.KeyframeAnimationGroup([a, b, c])
sage: g.threejs_repr(g.default_render_params())
[('point', {..., 'keyframe': 0, ..., 'point': (0.0, 0.0, 1.0), ...}),
 ('point', {..., 'keyframe': 1, ..., 'point': (0.0, 1.0, 0.0), ...}),
 ('point', {..., 'keyframe': 2, ..., 'point': (1.0, 0.0, 0.0), ...})]
```

Only top-level objects get a unique keyframe. Nested objects share the same keyframe:

```
sage: g = sage.plot.plot3d.base.KeyframeAnimationGroup([a + b, c])
sage: g.threejs_repr(g.default_render_params())
[('point', {..., 'keyframe': 0, ..., 'point': (0.0, 0.0, 1.0), ...}),
 ('point', {..., 'keyframe': 0, ..., 'point': (0.0, 1.0, 0.0), ...}),
 ('point', {..., 'keyframe': 1, ..., 'point': (1.0, 0.0, 0.0), ...})]
```

class sage.plot.plot3d.base.**PrimitiveObject**

Bases: *Graphics3d*

This is the base class for the non-container 3d objects.

get_texture()

EXAMPLES:

```
sage: G = dodecahedron(color='red')
sage: G.get_texture()
Texture(texture..., red, ff0000)
```

jmol_repr(*render_params*)

Default behavior is to render the triangulation. The actual polygon data is stored in a separate file.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Torus
sage: G = Torus(1, .5)
sage: G.jmol_repr(G.testing_render_params())
['pmesh obj_1 "obj_1.pmesh"\ncolor pmesh  [102,102,255]']
```

obj_repr(*render_params*)

Default behavior is to render the triangulation.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Torus
sage: G = Torus(1, .5)
sage: G.obj_repr(G.default_render_params())
['g obj_1',
 'usemtl ...',
 ['v 0 1 0.5',
 ...
  'f ...'],
 []]
```

set_texture(*texture=None*, ***kwds*)

EXAMPLES:

```
sage: G = dodecahedron(color='red'); G
Graphics3d Object
sage: G.set_texture(color='yellow'); G
Graphics3d Object
```

tachyon_repr(*render_params*)

Default behavior is to render the triangulation.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Torus
sage: G = Torus(1, .5)
sage: G.tachyon_repr(G.default_render_params())
['TRI V0 0 1 0.5
...
'texture...']
```

texture_set()

EXAMPLES:

```
sage: G = dodecahedron(color='red')
sage: G.texture_set()
{Texture(texture..., red, ff0000)}
```

threejs_repr(*render_params*)

Default behavior is to render the triangulation.

EXAMPLES:

```
sage: from sage.plot.plot3d.base import PrimitiveObject
sage: class SimpleTriangle(PrimitiveObject):
....:     def triangulation(self):
....:         return polygon3d([(0,0,0), (1,0,0), (0,1,0)])
sage: G = SimpleTriangle()
sage: G.threejs_repr(G.default_render_params())
[('surface',
```

(continues on next page)

(continued from previous page)

```
  {'color': '#0000ff',
   'faces': [[0, 1, 2]],
   'opacity': 1.0,
   'vertices': [{'x': 0.0, 'y': 0.0, 'z': 0.0},
    {'x': 1.0, 'y': 0.0, 'z': 0.0},
    {'x': 0.0, 'y': 1.0, 'z': 0.0}]})]
```

x3d_str()

EXAMPLES:

```
sage: sphere().flatten().x3d_str()
"<Transform>\n<Shape><Sphere radius='1.0'/><Appearance><Material diffuseColor=
↪'0.4 0.4 1.0' shininess='1.0' specularColor='0.0 0.0 0.0'/></Appearance></
↪Shape>\n\n</Transform>"
```

class sage.plot.plot3d.base.**RenderParams**(***kwds*)

Bases: SageObject

This class is a container for all parameters that may be needed to render triangulate/render an object to a certain format. It can contain both cumulative and global parameters.

Of particular note is the transformation object, which holds the cumulative transformation from the root of the scene graph to this node in the tree.

antialiasing = 8

dots = False

force_reload = False

mesh = False

pop_transform()

Remove the last transformation off the stack, resetting self.transform to the previous value.

EXAMPLES:

```
sage: from sage.plot.plot3d.transform import Transformation
sage: params = sage.plot.plot3d.base.RenderParams()
sage: T = Transformation(trans=(100, 500, 0))
sage: params.push_transform(T)
sage: params.transform.get_matrix()
[  1.0   0.0   0.0 100.0]
[  0.0   1.0   0.0 500.0]
[  0.0   0.0   1.0   0.0]
[  0.0   0.0   0.0   1.0]
sage: params.push_transform(Transformation(trans=(-100, 500, 200)))
sage: params.transform.get_matrix()
[   1.0    0.0    0.0    0.0]
[   0.0    1.0    0.0 1000.0]
[   0.0    0.0    1.0  200.0]
[   0.0    0.0    0.0    1.0]
sage: params.pop_transform()
sage: params.transform.get_matrix()
[  1.0   0.0   0.0 100.0]
```

(continues on next page)

(continued from previous page)

```
[  0.0   1.0   0.0 500.0]
[  0.0   0.0   1.0   0.0]
[  0.0   0.0   0.0   1.0]
```

push_transform(*T*)

Push a transformation onto the stack, updating self.transform.

EXAMPLES:

```
sage: from sage.plot.plot3d.transform import Transformation
sage: params = sage.plot.plot3d.base.RenderParams()
sage: params.transform is None
True
sage: T = Transformation(scale=(10,20,30))
sage: params.push_transform(T)
sage: params.transform.get_matrix()
[10.0  0.0  0.0  0.0]
[ 0.0 20.0  0.0  0.0]
[ 0.0  0.0 30.0  0.0]
[ 0.0  0.0  0.0  1.0]
sage: params.push_transform(T)  # scale again
sage: params.transform.get_matrix()
[100.0   0.0   0.0   0.0]
[  0.0 400.0   0.0   0.0]
[  0.0   0.0 900.0   0.0]
[  0.0   0.0   0.0   1.0]
```

randomize_counter = 0

unique_name(*desc='name'*)

Return a unique identifier starting with `desc`.

INPUT:

- `desc` (string) – the prefix of the names (default 'name')

EXAMPLES:

```
sage: params = sage.plot.plot3d.base.RenderParams()
sage: params.unique_name()
'name_1'
sage: params.unique_name()
'name_2'
sage: params.unique_name('texture')
'texture_3'
```

class sage.plot.plot3d.base.**TransformGroup**(*all=[]*, *rot=None*, *trans=None*, *scale=None*, *T=None*)

Bases: *Graphics3dGroup*

This class is a container for a group of objects with a common transformation.

bounding_box()

Return the bounding box, i.e., the box containing the contents of the object after applying the transformation.

EXAMPLES:

```
sage: from math import pi
sage: G = cube()
sage: G.bounding_box()
((-0.5, -0.5, -0.5), (0.5, 0.5, 0.5))
sage: G.scale(4).bounding_box()
((-2.0, -2.0, -2.0), (2.0, 2.0, 2.0))
sage: G.rotateZ(pi/4).bounding_box()
((-0.7071067811865475, -0.7071067811865475, -0.5),
 (0.7071067811865475, 0.7071067811865475, 0.5))
```

flatten()

Try to reduce the depth of the scene tree by consolidating groups and transformations.

EXAMPLES:

```
sage: G = sphere((1,2,3)).scale(100)
sage: T = G.get_transformation()
sage: T.get_matrix()
[100.0   0.0   0.0   0.0]
[  0.0 100.0   0.0   0.0]
[  0.0   0.0 100.0   0.0]
[  0.0   0.0   0.0   1.0]

sage: G.flatten().get_transformation().get_matrix()
[100.0   0.0   0.0 100.0]
[  0.0 100.0   0.0 200.0]
[  0.0   0.0 100.0 300.0]
[  0.0   0.0   0.0   1.0]
```

get_transformation()

Return the current transformation object.

EXAMPLES:

```
sage: G = sphere().scale(100)
sage: T = G.get_transformation()
sage: T.get_matrix()
[100.0   0.0   0.0   0.0]
[  0.0 100.0   0.0   0.0]
[  0.0   0.0 100.0   0.0]
[  0.0   0.0   0.0   1.0]
```

jmol_repr(*render_params*)

Transformations for jmol are applied at the leaf nodes.

EXAMPLES:

```
sage: G = sphere((1,2,3)).scale(2)
sage: G.jmol_repr(G.default_render_params())
[[['isosurface sphere_1  center {2.0 4.0 6.0} sphere 2.0\ncolor isosurface ␣
↪[102,102,255]']]]
```

json_repr(*render_params*)

Transformations are applied at the leaf nodes.

EXAMPLES:

```
sage: G = cube().rotateX(0.2)
sage: G.json_repr(G.default_render_params())
[['{"vertices":[{"x":0.5,"y":0.589368,"z":0.390699},...']]
```

obj_repr(*render_params*)

Transformations for .obj files are applied at the leaf nodes.

EXAMPLES:

```
sage: G = cube().scale(4).translate(1, 2, 3)
sage: G.obj_repr(G.default_render_params())
[[['g obj_1',
   'usemtl ...',
   ['v 3 4 5',
    'v -1 4 5',
    'v -1 0 5',
    'v 3 0 5',
    'v 3 4 1',
    'v -1 4 1',
    'v 3 0 1',
    'v -1 0 1'],
   ['f 1 2 3 4',
    'f 1 5 6 2',
    'f 1 4 7 5',
    'f 6 5 7 8',
    'f 7 4 3 8',
    'f 3 2 6 8'],
   []]]]
```

stl_binary_repr(*render_params*)

Transformations are applied at the leaf nodes.

The STL binary representation is a list of binary strings, one for each triangle.

EXAMPLES:

```
sage: G = sphere().translate((1,2,0))
sage: len(G.stl_binary_repr(G.default_render_params()))
1368
```

tachyon_repr(*render_params*)

Transformations for Tachyon are applied at the leaf nodes.

EXAMPLES:

```
sage: G = sphere((1,2,3)).scale(2)
sage: G.tachyon_repr(G.default_render_params())
[['Sphere center 2.0 4.0 6.0 Rad 2.0 texture...']]
```

threejs_repr(*render_params*)

Transformations for three.js are applied at the leaf nodes.

EXAMPLES:

```
sage: G = point3d((1,2,3)) + point3d((4,5,6))
sage: G = G.translate(-1, -2, -3).scale(10)
sage: G.threejs_repr(G.default_render_params())
[('point',
  {'color': '#6666ff', 'opacity': 1.0, 'point': (0.0, 0.0, 0.0), 'size': 5.0}),
 ('point',
  {'color': '#6666ff',
   'opacity': 1.0,
   'point': (30.0, 30.0, 30.0),
   'size': 5.0})]
```

transform(***kwds*)

Transforming this entire group can be done by composing transformations.

EXAMPLES:

```
sage: G = dodecahedron(color='red', opacity=.5) + icosahedron(color='blue')
sage: G
Graphics3d Object
sage: G.transform(scale=(2,1/2,1))
Graphics3d Object
sage: G.transform(trans=(1,1,3))
Graphics3d Object
```

x3d_str()

To apply a transformation to a set of objects in x3d, simply make them all children of an x3d Transform node.

EXAMPLES:

```
sage: sphere((1,2,3)).x3d_str()
"<Transform translation='1 2 3'>\n<Shape><Sphere radius='1.0'/><Appearance>
↪<Material diffuseColor='0.4 0.4 1.0' shininess='1.0' specularColor='0.0 0.0 0.
↪0'/></Appearance></Shape>\n\n</Transform>"
```

class sage.plot.plot3d.base.**Viewpoint**(**x*)

Bases: *Graphics3d*

This class represents a viewpoint, necessary for x3d.

In the future, there could be multiple viewpoints, and they could have more properties. (Currently they only hold a position).

x3d_str()

EXAMPLES:

```
sage: sphere((0,0,0), 100).viewpoint().x3d_str()
"<Viewpoint position='0 0 6'/>"
```

sage.plot.plot3d.base.**flatten_list**(*L*)

This is an optimized routine to turn a list of lists (of lists …) into a single list. We generate data in a non-flat format to avoid multiple data copying, and then concatenate it all at the end.

This is NOT recursive, otherwise there would be a lot of redundant copying (which we are trying to avoid in the first place, though at least it would be just the pointers).

EXAMPLES:

```
sage: from sage.plot.plot3d.base import flatten_list
sage: flatten_list([])
[]
sage: flatten_list([[[[]]]])
[]
sage: flatten_list([['a', 'b'], 'c'])
['a', 'b', 'c']
sage: flatten_list([['a'], [[['b'], 'c'], ['d'], [[['e', 'f', 'g']]]]])
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

sage.plot.plot3d.base.**max3**(*v*)

Return the componentwise maximum of a list of 3-tuples.

EXAMPLES:

```
sage: from sage.plot.plot3d.base import min3, max3
sage: max3([(-1,2,5), (-3, 4, 2)])
(-1, 4, 5)
```

sage.plot.plot3d.base.**min3**(*v*)

Return the componentwise minimum of a list of 3-tuples.

EXAMPLES:

```
sage: from sage.plot.plot3d.base import min3, max3
sage: min3([(-1,2,5), (-3, 4, 2)])
(-3, 2, 2)
```

sage.plot.plot3d.base.**optimal_aspect_ratios**(*ratios*)

Average the aspect ratios. compute the elementwise maximum of triples.

sage.plot.plot3d.base.**optimal_extra_kwds**(*v*)

Merge a list v of dictionaries such that later dictionaries have precedence.

sage.plot.plot3d.base.**point_list_bounding_box**(*v*)

Return the bounding box of a list of points.

EXAMPLES:

```
sage: from sage.plot.plot3d.base import point_list_bounding_box
sage: point_list_bounding_box([(1,2,3),(4,5,6),(-10,0,10)])
((-10.0, 0.0, 3.0), (4.0, 5.0, 10.0))
sage: point_list_bounding_box([(float('nan'), float('inf'), float('-inf')), (10,0,
↪10)])
((10.0, 0.0, 10.0), (10.0, 0.0, 10.0))
```

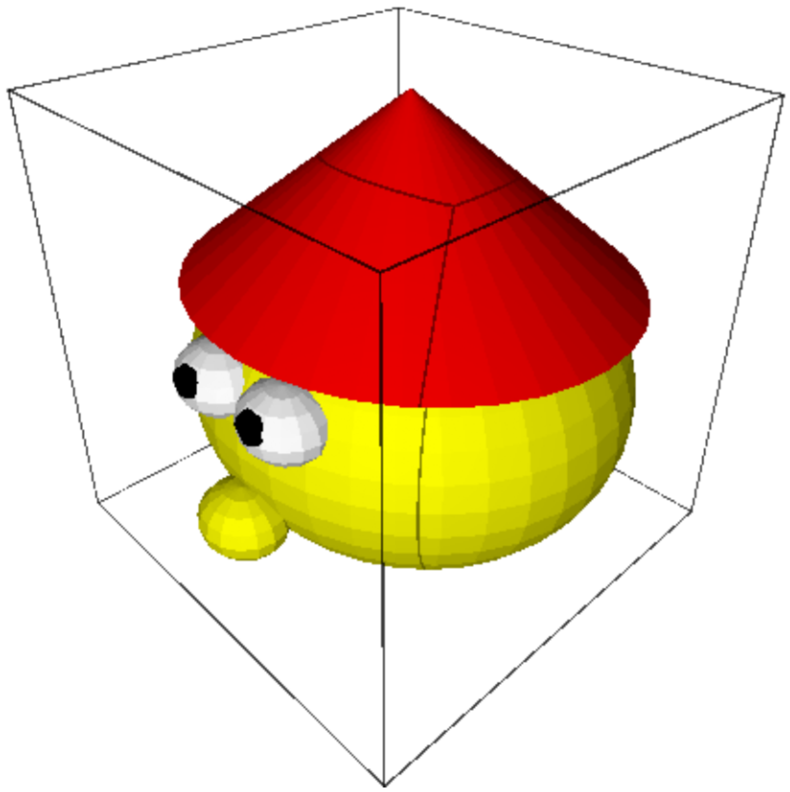
3.2 Basic objects such as Sphere, Box, Cone, etc.

AUTHORS:

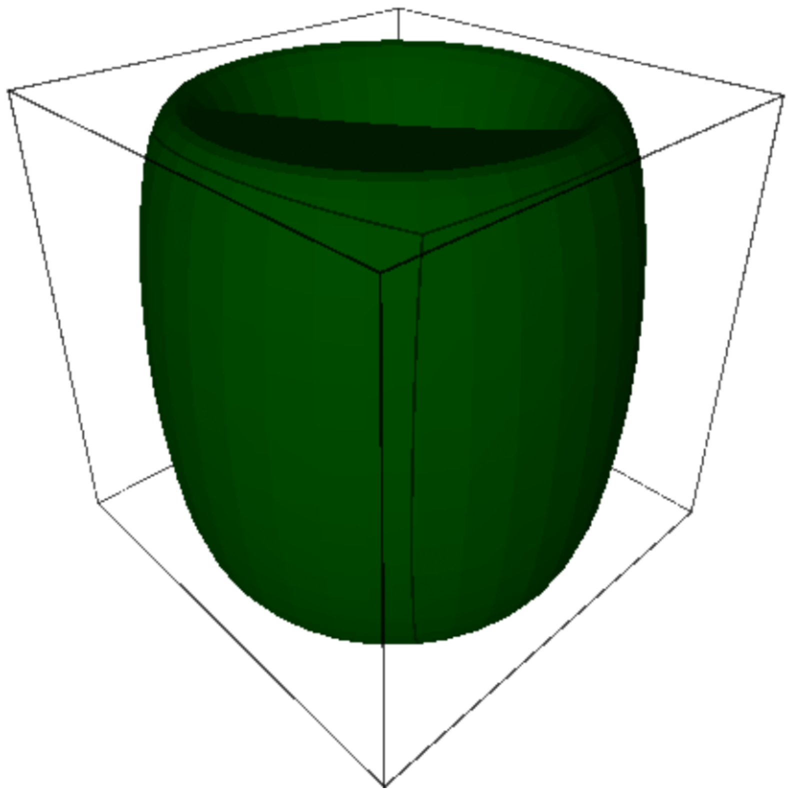
- Robert Bradshaw 2007-02: initial version
- Robert Bradshaw 2007-08: obj/tachyon rendering, much updating
- Robert Bradshaw 2007-08: cythonization

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import *
sage: S = Sphere(.5, color='yellow')
sage: S += Cone(.5, .5, color='red').translate(0,0,.3)
sage: S += Sphere(.1, color='white').translate(.45,-.1,.15)
sage: S += Sphere(.05, color='black').translate(.51,-.1,.17)
sage: S += Sphere(.1, color='white').translate(.45, .1,.15)
sage: S += Sphere(.05, color='black').translate(.51, .1,.17)
sage: S += Sphere(.1, color='yellow').translate(.5, 0, -.2)
sage: S.show()
sage: S.scale(1,1,2).show()
```



```
sage: from sage.plot.plot3d.shapes import *
sage: Torus(.7, .2, color=(0,.3,0)).show()
```



class sage.plot.plot3d.shapes.**Box**(**size*, ***kwds*)

Bases: *IndexFaceSet*

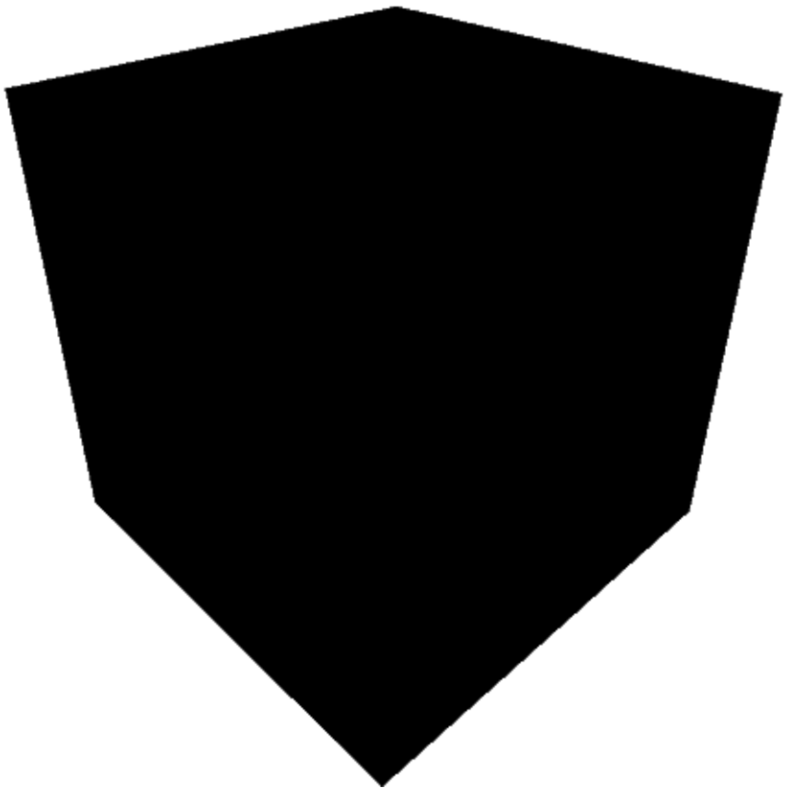
Return a box.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Box
```

A square black box:

```
sage: show(Box([1,1,1]), color='black')
```



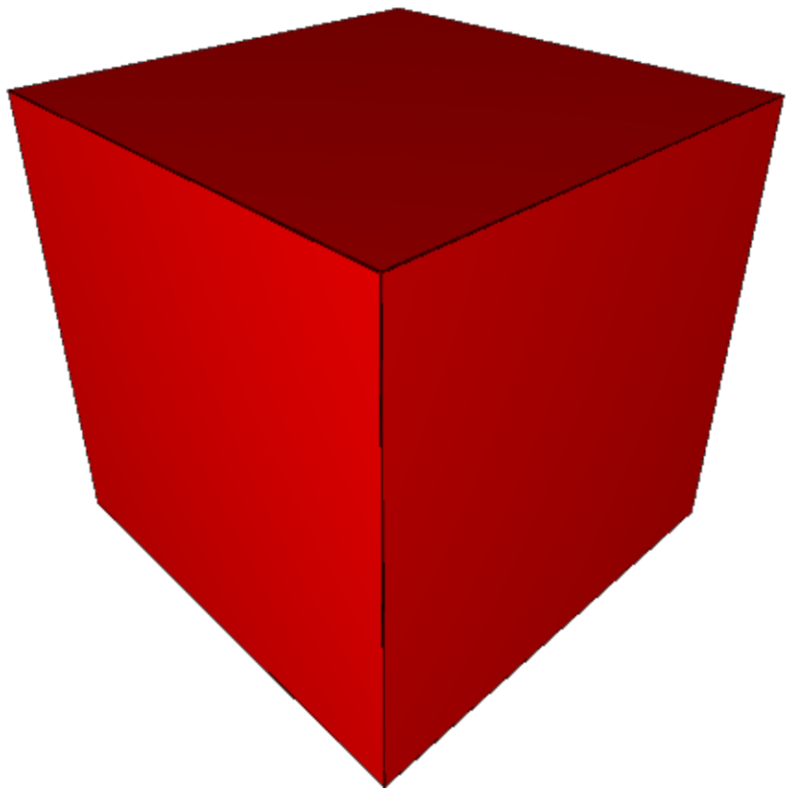
A red rectangular box:

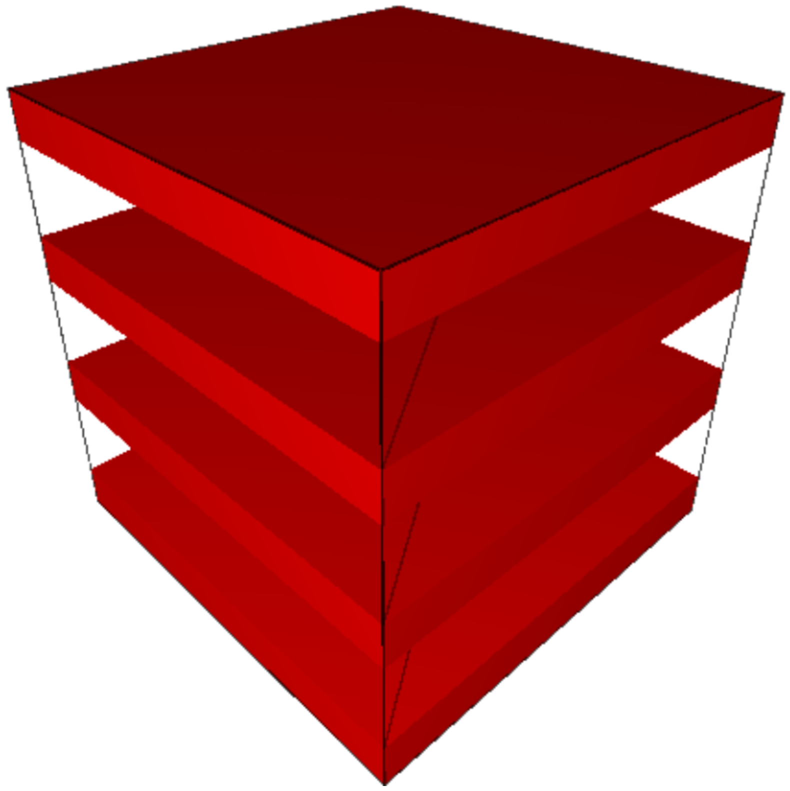
```
sage: show(Box([2,3,4], color="red"))
```

A stack of boxes:

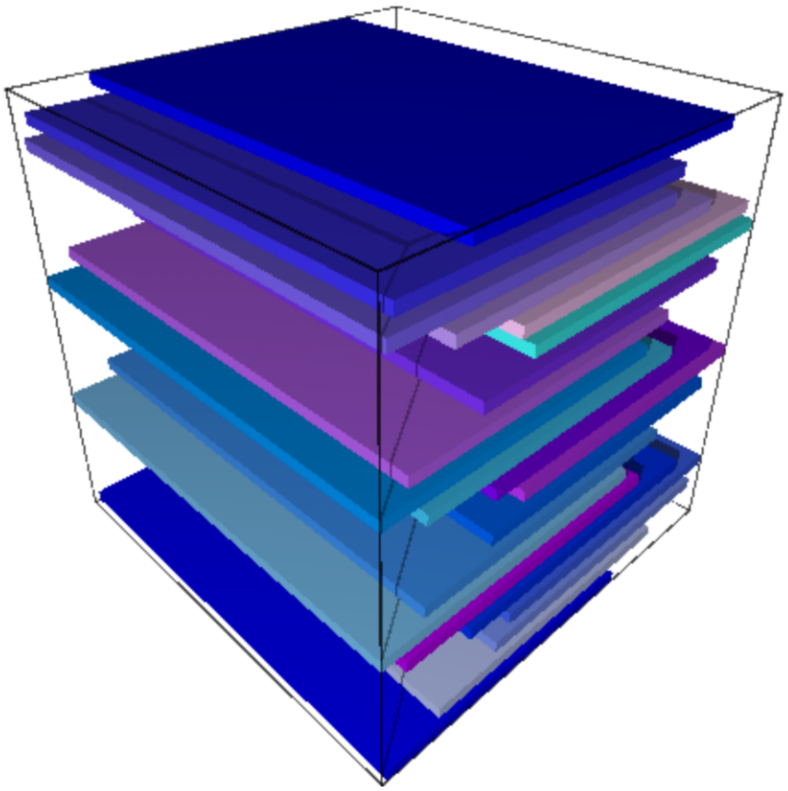
```
sage: show(sum(Box([2,3,1], color="red").translate((0,0,6*i))
....:          for i in [0..3]))
```

A sinusoidal stack of multicolored boxes:





```
sage: B = sum(Box([2,4,1/4],                                              #␣
↪needs sage.symbolic
....:             color=(i/4,i/5,1)).translate((sin(i),0,5-i))
....:         for i in [0..20])
sage: show(B, figsize=6)                                                  #␣
↪needs sage.symbolic
```



bounding_box()

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Box
sage: Box([1,2,3]).bounding_box()
((-1.0, -2.0, -3.0), (1.0, 2.0, 3.0))
```

x3d_geometry()

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Box
sage: Box([1,2,1/4]).x3d_geometry()
"<Box size='1.0 2.0 0.25'/>"
```

sage.plot.plot3d.shapes.**ColorCube**(*size*, *colors*, *opacity=1*, ***kwds*)

Return a cube with given size and sides with given colors.

INPUT:

- `size` – 3-tuple of sizes (same as for box and frame)
- `colors` – a list of either 3 or 6 colors
- `opacity` – (default: 1) opacity of cube sides
- `**kwds` – passed to the face constructor

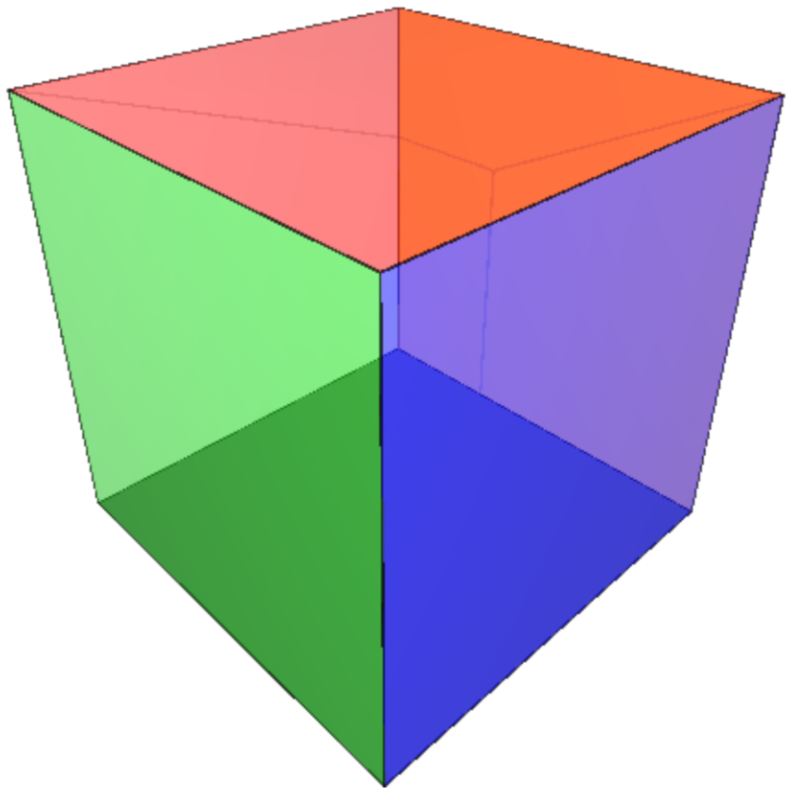
OUTPUT:

a 3d graphics object

EXAMPLES:

A color cube with translucent sides:

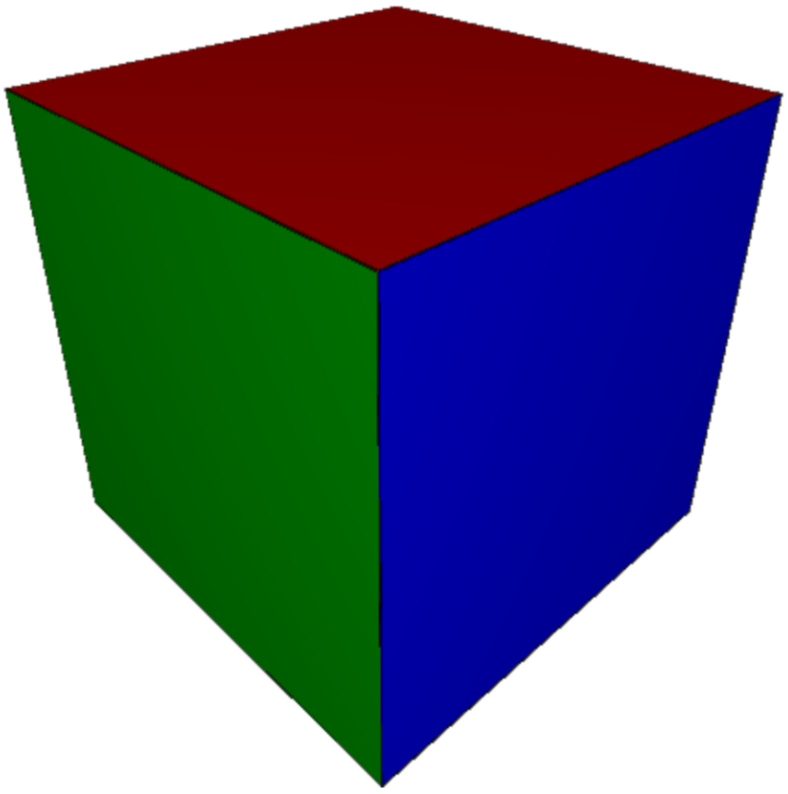
```
sage: from sage.plot.plot3d.shapes import ColorCube
sage: c = ColorCube([1,2,3], ['red', 'blue', 'green', 'black', 'white', 'orange'],␣
↪opacity=0.5)
sage: c.show()
```



```
sage: list(c.texture_set())[0].opacity
0.5
```

If you omit the last 3 colors then the first three are repeated (with repeated colors on opposing faces):

```
sage: c = ColorCube([0.5,0.5,0.5], ['red', 'blue', 'green'])
```



class sage.plot.plot3d.shapes.**Cone**

Bases: *ParametricSurface*

A cone, with base in the xy-plane pointing up the z-axis.

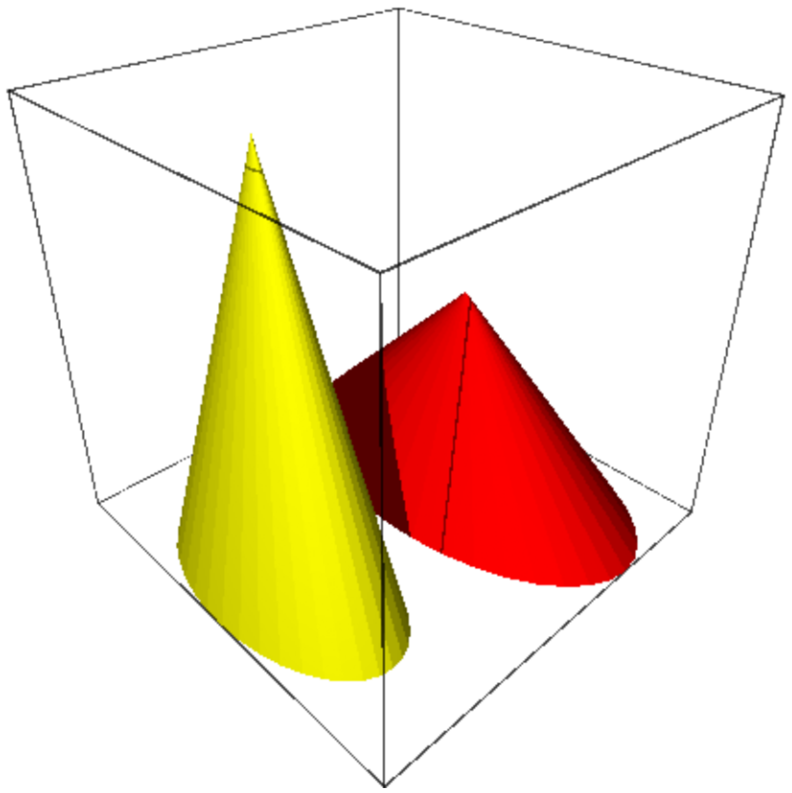
INPUT:

- `radius` – positive real number
- `height` – positive real number
- `closed` – whether or not to include the base (default `True`)
- `**kwds` – passed to the ParametricSurface constructor

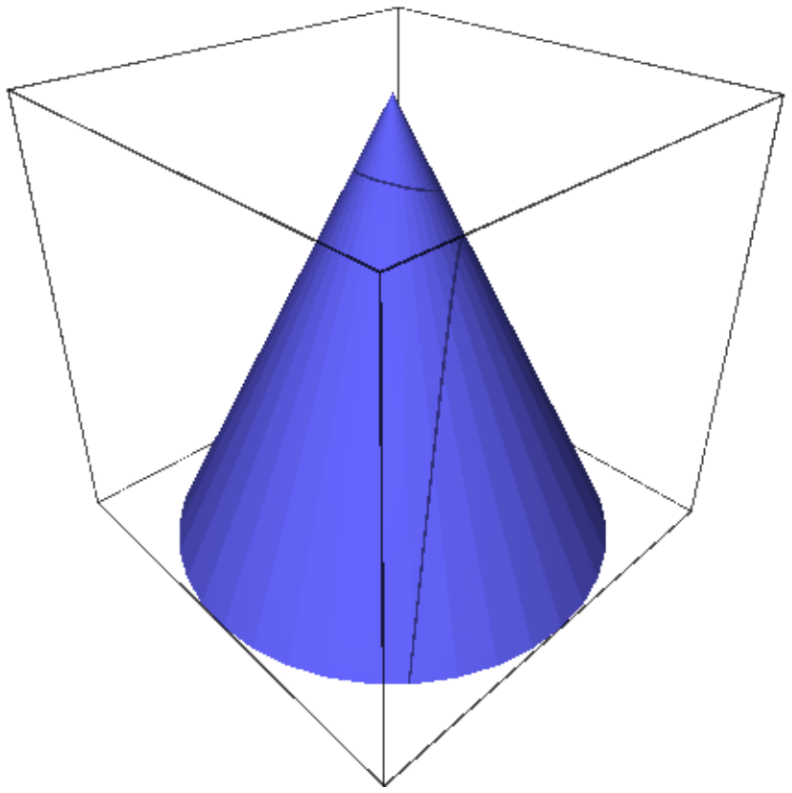
EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Cone
sage: c = Cone(3/2, 1, color='red')
sage: c += Cone(1, 2, color='yellow').translate(3, 0, 0)
sage: c.show(aspect_ratio=1)
```

We may omit the base:



```
sage: Cone(1, 1, closed=False)
Graphics3d Object
```



A spiky plot of the sine function:

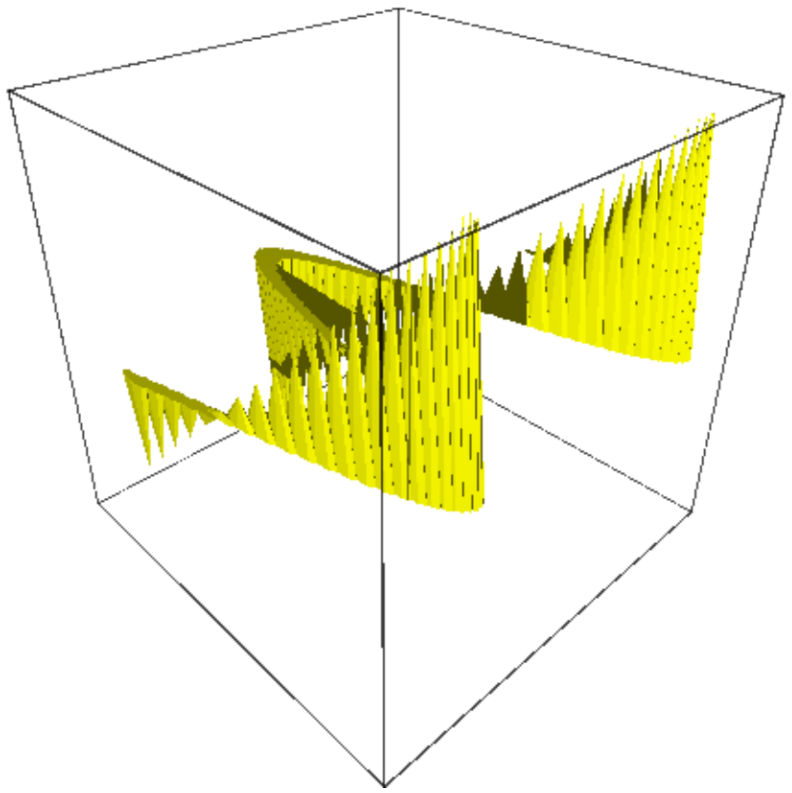
```
sage: sum(Cone(.1, sin(n), color='yellow').translate(n, sin(n), 0)          #␣
↪needs sage.symbolic
....:     for n in [0..10, step=.1])
Graphics3d Object
```

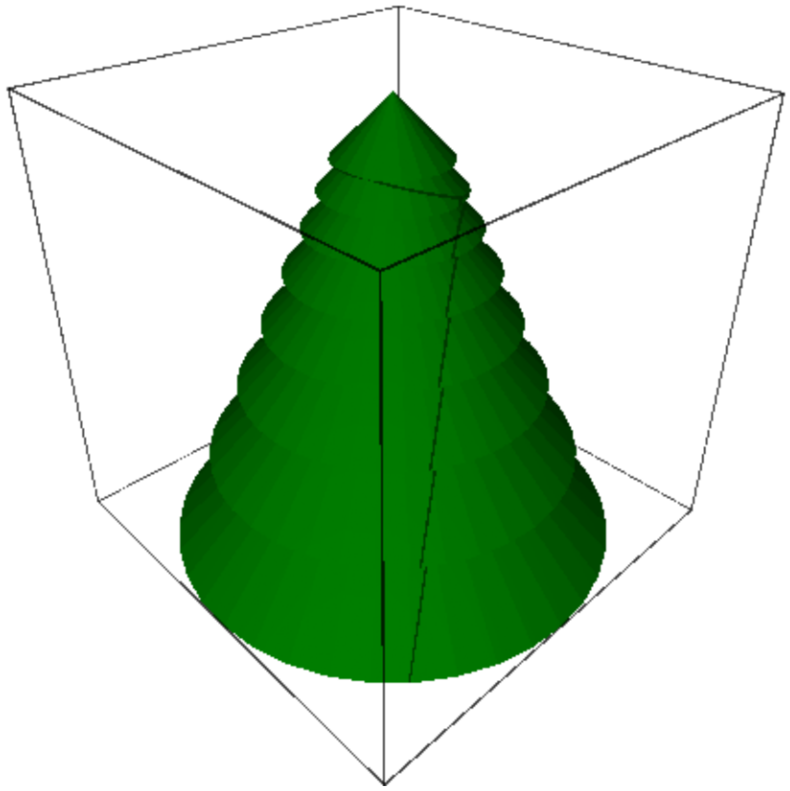
A Christmas tree:

```
sage: T = sum(Cone(exp(-n/5), 4/3*exp(-n/5),                                #␣
↪needs sage.symbolic
....:              color=(0, .5, 0)).translate(0, 0, -3*exp(-n/5))
....:         for n in [1..7])
sage: T += Cone(1/8, 1, color='brown').translate(0, 0, -3)                  #␣
↪needs sage.symbolic
sage: T.show(aspect_ratio=1, frame=False)                                   #␣
↪needs sage.symbolic
```

get_grid(*ds*)

Return the grid on which to evaluate this parametric surface.





EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Cone
sage: Cone(1, 3, closed=True).get_grid(100)
([1, 0, -1], [0.0, 1.2566..., 2.5132..., 3.7699..., 5.0265..., 0.0])
sage: Cone(1, 3, closed=False).get_grid(100)
([1, 0], [0.0, 1.2566..., 2.5132..., 3.7699..., 5.0265..., 0.0])
sage: len(Cone(1, 3).get_grid(.001)[1])
38
```

x3d_geometry()

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Cone
sage: Cone(1, 3).x3d_geometry()
"<Cone bottomRadius='1.0' height='3.0'/>"
```

class sage.plot.plot3d.shapes.**Cylinder**

Bases: *ParametricSurface*

A cone, with base in the xy-plane pointing up the z-axis.

INPUT:

- `radius` – positive real number
- `height` – positive real number
- `closed` – whether or not to include the ends (default `True`)
- `**kwds` – passed to the `ParametricSurface` constructor

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Cylinder
sage: c = Cylinder(3/2, 1, color='red')
sage: c += Cylinder(1, 2, color='yellow').translate(3, 0, 0)
sage: c.show(aspect_ratio=1)
```

We may omit the base:

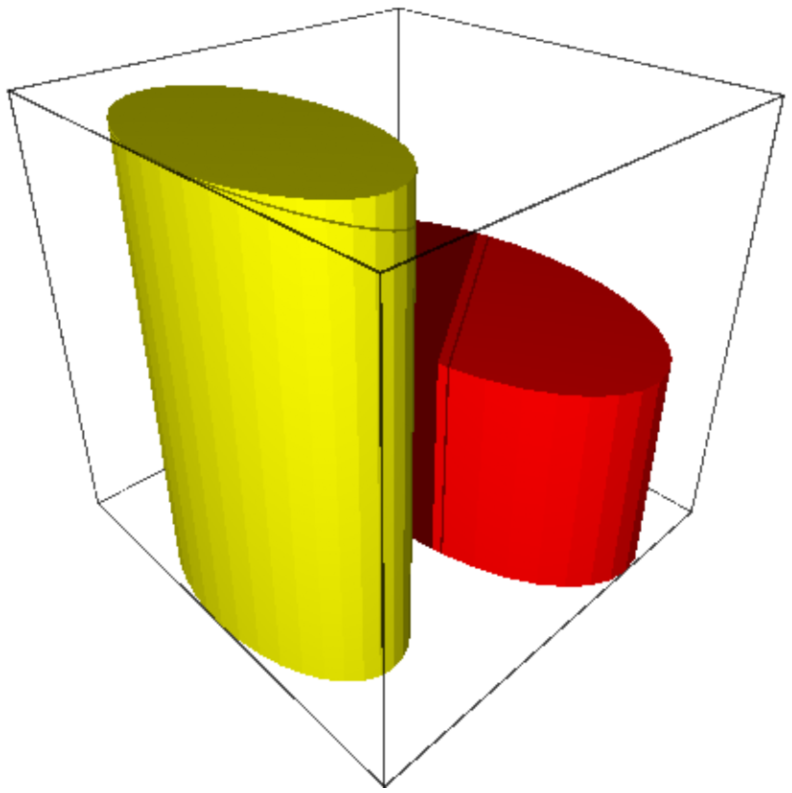
```
sage: Cylinder(1, 1, closed=False)
Graphics3d Object
```

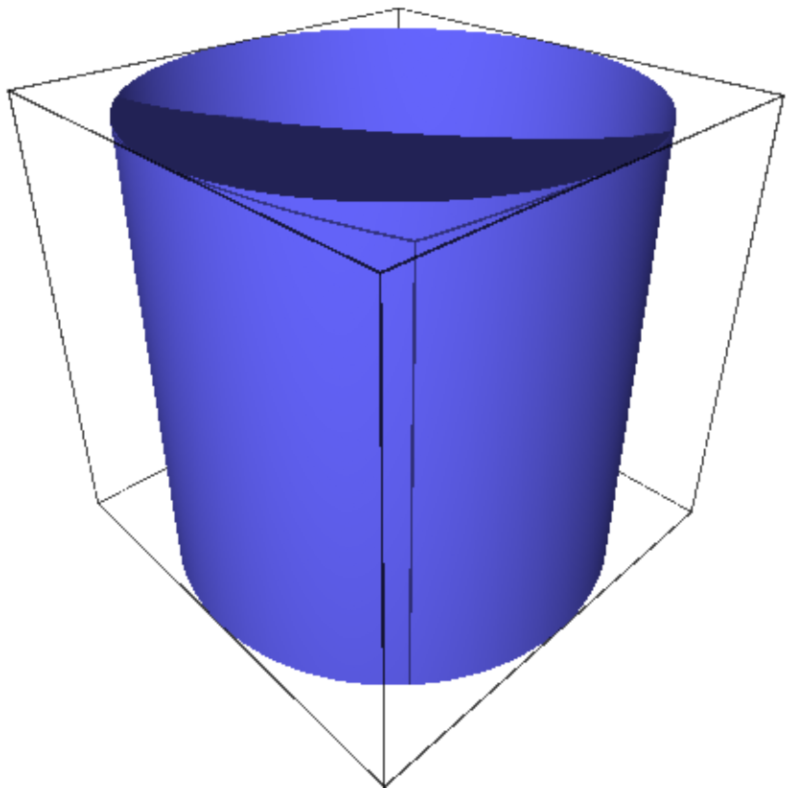
Some gears:

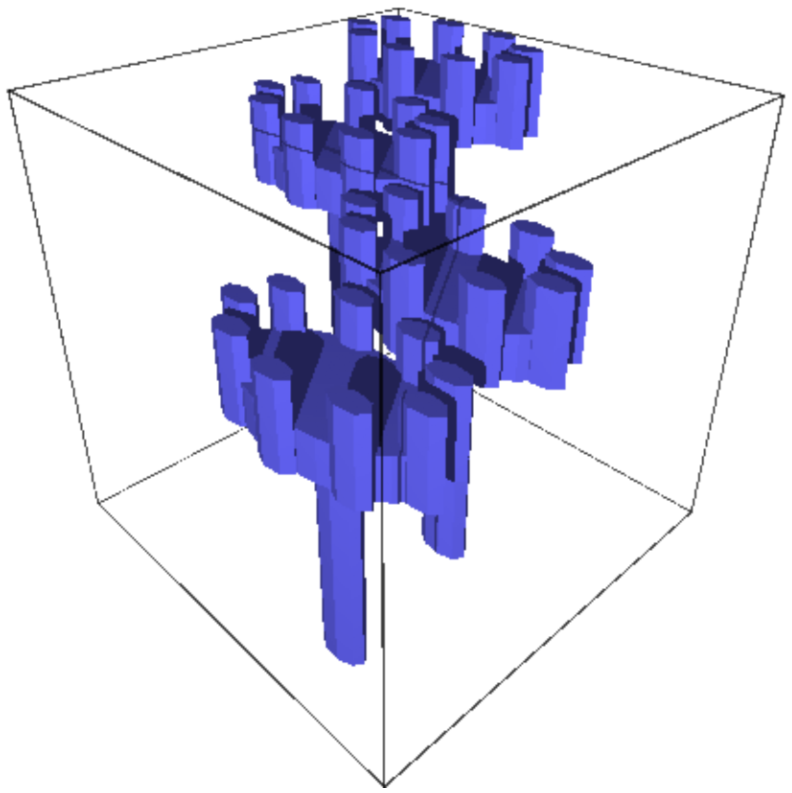
```
sage: # needs sage.symbolic
sage: G = Cylinder(1, .5) + Cylinder(.25, 3).translate(0, 0, -3)
sage: G += sum(Cylinder(.2, 1).translate(cos(2*pi*n/9), sin(2*pi*n/9), 0)
....:          for n in [1..9])
sage: G += G.translate(2.3, 0, -.5)
sage: G += G.translate(3.5, 2, -1)
sage: G.show(aspect_ratio=1, frame=False)
```

bounding_box()

EXAMPLES:







```
sage: from sage.plot.plot3d.shapes import Cylinder
sage: Cylinder(1, 2).bounding_box()
((-1.0, -1.0, 0), (1.0, 1.0, 2.0))
```

get_endpoints(*transform=None*)

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Cylinder
sage: from sage.plot.plot3d.transform import Transformation
sage: Cylinder(1, 5).get_endpoints()
((0, 0, 0), (0, 0, 5.0))
sage: Cylinder(1, 5).get_endpoints(Transformation(trans=(1,2,3),
....:                                             scale=(2,2,2)))
((1.0, 2.0, 3.0), (1.0, 2.0, 13.0))
```

get_grid(*ds*)

Return the grid on which to evaluate this parametric surface.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Cylinder
sage: Cylinder(1, 3, closed=True).get_grid(100)
([2, 1, -1, -2], [0.0, 1.2566..., 2.5132..., 3.7699..., 5.0265..., 0.0])
sage: Cylinder(1, 3, closed=False).get_grid(100)
([1, -1], [0.0, 1.2566..., 2.5132..., 3.7699..., 5.0265..., 0.0])
sage: len(Cylinder(1, 3).get_grid(.001)[1])
38
```

get_radius(*transform=None*)

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Cylinder
sage: from sage.plot.plot3d.transform import Transformation
sage: Cylinder(3, 1).get_radius()
3.0
sage: Cylinder(3, 1).get_radius(Transformation(trans=(1,2,3),
....:                                          scale=(2,2,2)))
6.0
```

jmol_repr(*render_params*)

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Cylinder
```

For thin cylinders, lines are used:

```
sage: C = Cylinder(.1, 4)
sage: C.jmol_repr(C.default_render_params())
['\ndraw line_1 width 0.1 {0 0 0} {0 0 4.0}\ncolor $line_1  [102,102,255]\n']
```

For anything larger, we use a pmesh:

```
sage: C = Cylinder(3, 1, closed=False)
sage: C.jmol_repr(C.testing_render_params())
['pmesh obj_1 "obj_1.pmesh"\ncolor pmesh  [102,102,255]']
```

tachyon_repr(*render_params*)

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Cylinder
sage: C = Cylinder(1/2, 4, closed=False)
sage: C.tachyon_repr(C.default_render_params())
'FCylinder\n   Base 0 0 0\n   Apex 0 0 4.0\n   Rad 0.5\n   texture...     '
sage: C = Cylinder(1, 2)
sage: C.tachyon_repr(C.default_render_params())
    ['Ring Center 0 0 0 Normal 0 0 1 Inner 0 Outer 1.0 texture...',
     'FCylinder\n   Base 0 0 0\n   Apex 0 0 2.0\n   Rad 1.0\n   texture...
↪',
     'Ring Center 0 0 2.0 Normal 0 0 1 Inner 0 Outer 1.0 texture...']
```

x3d_geometry()

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Cylinder
sage: Cylinder(1, 2).x3d_geometry()
"<Cylinder radius='1.0' height='2.0'/>"
```

sage.plot.plot3d.shapes.**LineSegment**(*start*, *end*, *thickness=1*, *radius=None*, ***kwds*)

Create a line segment, which is drawn as a cylinder from start to end with radius `radius`.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import LineSegment, Sphere
sage: P = (0,0,0.1)
sage: Q = (0.5,0.6,0.7)
sage: S = Sphere(.2, color='red').translate(P)
sage: S += Sphere(.2, color='blue').translate(Q)
sage: S += LineSegment(P, Q, .05, color='black')
sage: S.show()
```

```
sage: S = Sphere(.1, color='red').translate(P)
sage: S += Sphere(.1, color='blue').translate(Q)
sage: S += LineSegment(P, Q, .15, color='black')
sage: S.show()
```

AUTHOR:

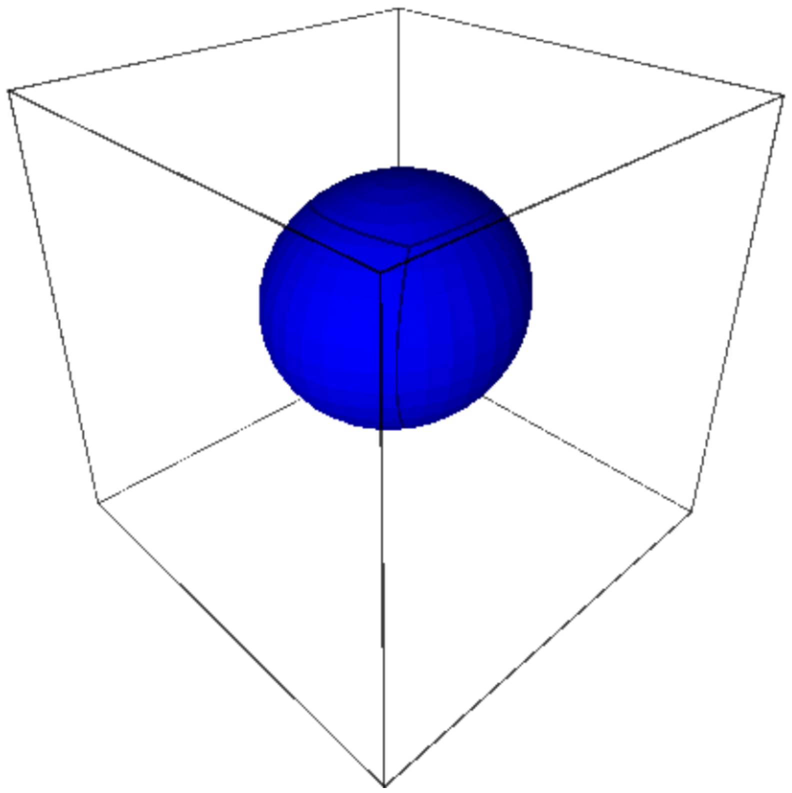
- Robert Bradshaw

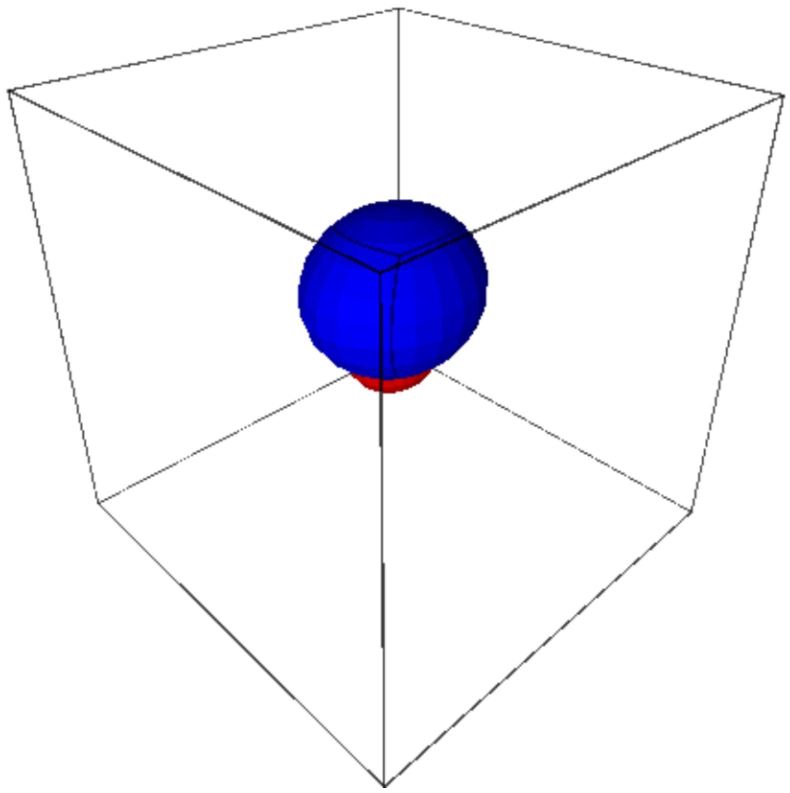
class sage.plot.plot3d.shapes.**Sphere**

Bases: *ParametricSurface*

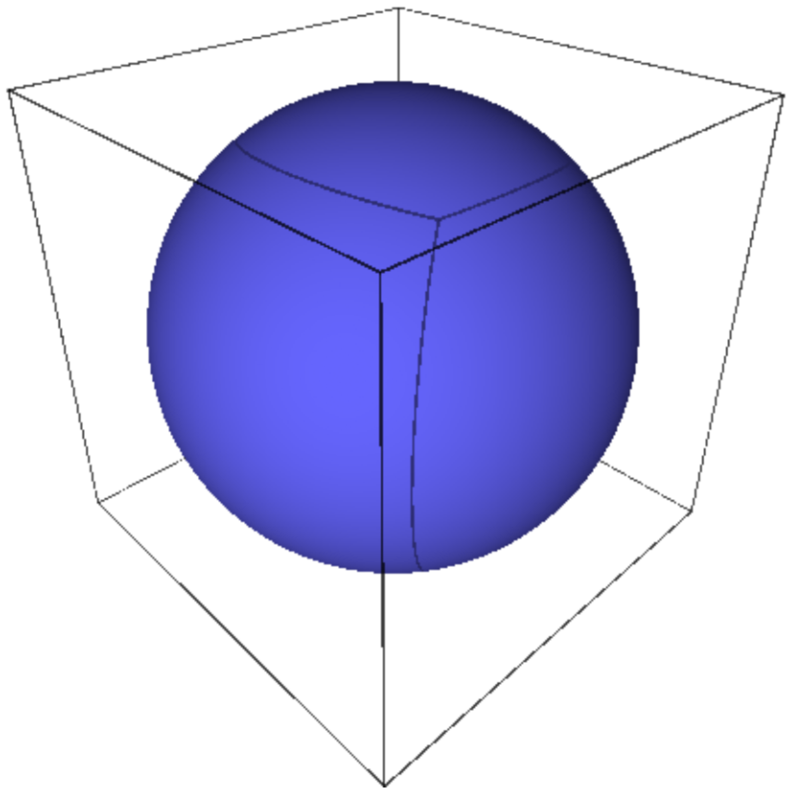
This class represents a sphere centered at the origin.

EXAMPLES:





```
sage: from sage.plot.plot3d.shapes import Sphere
sage: Sphere(3)
Graphics3d Object
```



Plot with `aspect_ratio=1` to see it unsquashed:

```
sage: S = Sphere(3, color='blue') + Sphere(2, color='red').translate(0,3,0)
sage: S.show(aspect_ratio=1)
```

Scale to get an ellipsoid:

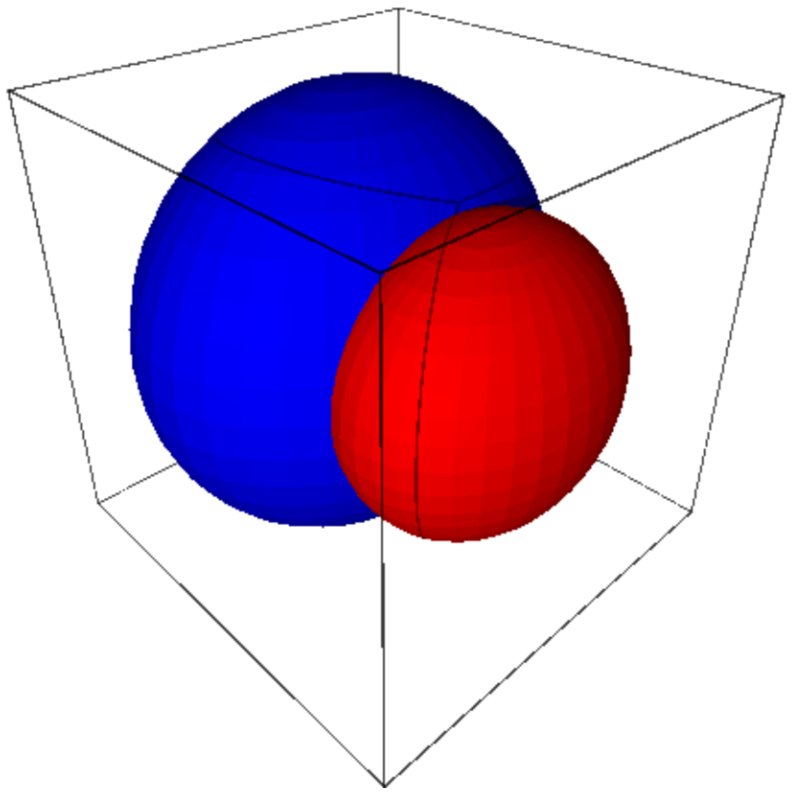
```
sage: S = Sphere(1).scale(1,2,1/2)
sage: S.show(aspect_ratio=1)
```

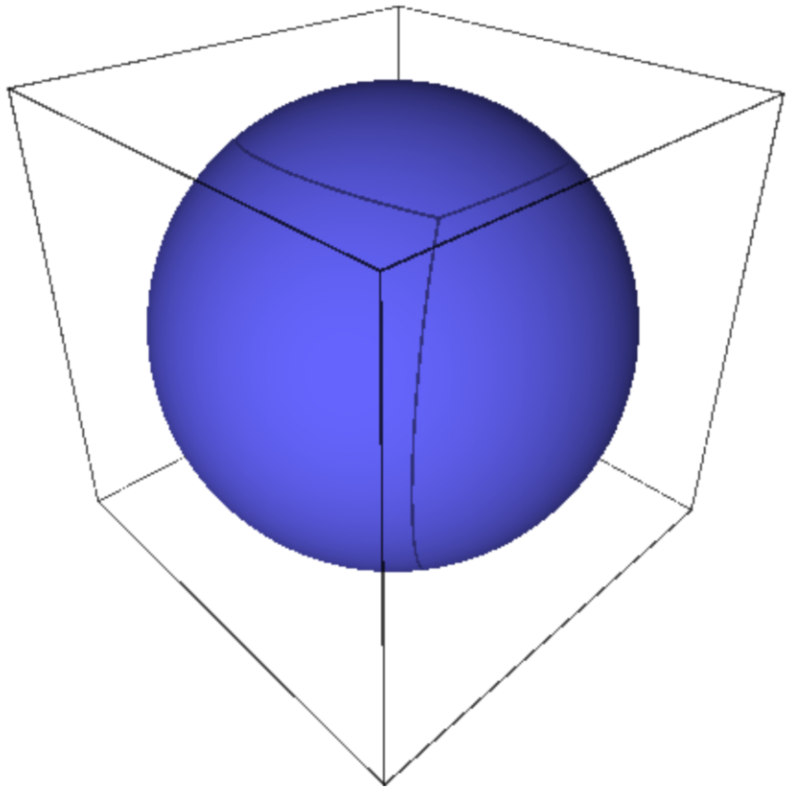
bounding_box()

Return the bounding box that contains this sphere.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Sphere
sage: Sphere(3).bounding_box()
((-3.0, -3.0, -3.0), (3.0, 3.0, 3.0))
```





get_grid(*ds*)

Return the range of variables to be evaluated on to render as a parametric surface.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Sphere
sage: Sphere(1).get_grid(100)
([-10.0, ..., 10.0],
 [0.0, ..., 3.141592653589793, ..., 0.0])
```

jmol_repr(*render_params*)

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Sphere
```

Jmol has native code for handling spheres:

```
sage: S = Sphere(2)
sage: S.jmol_repr(S.default_render_params())
['isosurface sphere_1  center {0 0 0} sphere 2.0\ncolor isosurface  [102,102,
↪255]']
sage: S.translate(10, 100, 1000).jmol_repr(S.default_render_params())
[['isosurface sphere_1  center {10.0 100.0 1000.0} sphere 2.0\ncolor isosurface␣
↪ [102,102,255]']]
```

It cannot natively handle ellipsoids:

```
sage: Sphere(1).scale(2, 3, 4).jmol_repr(S.testing_render_params())
[['pmesh obj_2 "obj_2.pmesh"\ncolor pmesh  [102,102,255]']]
```

Small spheres need extra hints to render well:

```
sage: Sphere(.01).jmol_repr(S.default_render_params())
['isosurface sphere_1 resolution 100 center {0 0 0} sphere 0.01\ncolor␣
↪isosurface  [102,102,255]']
```

tachyon_repr(*render_params*)

Tachyon can natively handle spheres. Ellipsoids rendering is done as a parametric surface.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Sphere
sage: S = Sphere(2)
sage: S.tachyon_repr(S.default_render_params())
'Sphere center 0 0 0 Rad 2.0 texture...'
sage: S.translate(1, 2, 3).scale(3).tachyon_repr(S.default_render_params())
[['Sphere center 3.0 6.0 9.0 Rad 6.0 texture...']]
sage: S.scale(1,1/2,1/4).tachyon_repr(S.default_render_params())
[['TRI V0 0 0 -0.5 V1 0.308116 0.0271646 -0.493844 V2 0.312869 0 -0.493844',
  'texture...',
   ...
  'TRI V0 0.308116 -0.0271646 0.493844 V1 0.312869 0 0.493844 V2 0 0 0.5',
  'texture...']]
```

x3d_geometry()

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Sphere
sage: Sphere(12).x3d_geometry()
"<Sphere radius='12.0'/>"
```

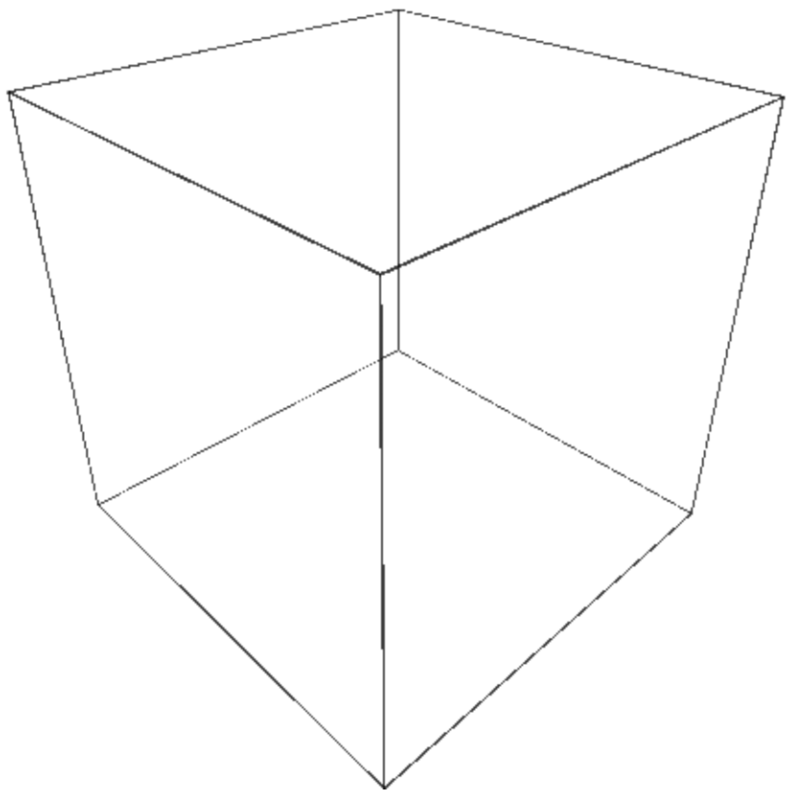
class sage.plot.plot3d.shapes.**Text**(*string*, ***kwds*)

Bases: *PrimitiveObject*

A text label attached to a point in 3d space. It always starts at the origin, translate it to move it elsewhere.

EXAMPLES:

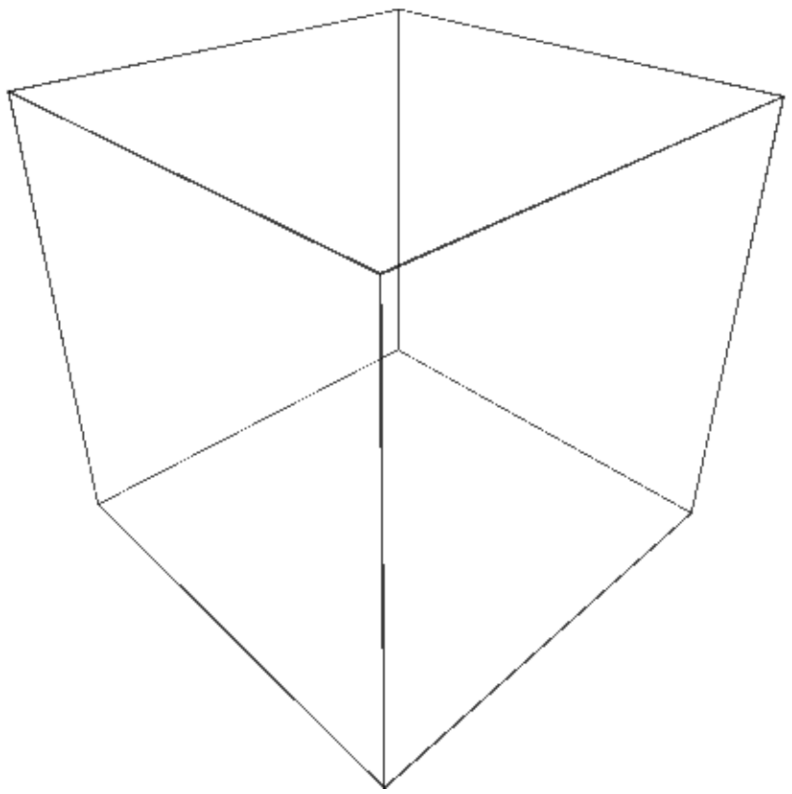
```
sage: from sage.plot.plot3d.shapes import Text
sage: Text("Just a lonely label.")
Graphics3d Object
```



```
sage: pts = [(RealField(10)^3).random_element() for k in range(20)]
sage: sum(Text(str(P)).translate(P) for P in pts)
Graphics3d Object
```

bounding_box()

Text labels have no extent:



```
sage: from sage.plot.plot3d.shapes import Text
sage: Text("Hi").bounding_box()
((0, 0, 0), (0, 0, 0))
```

jmol_repr(*render_params*)

Labels in jmol must be attached to atoms.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Text
sage: T = Text("Hi")
sage: T.jmol_repr(T.testing_render_params())
['select atomno = 1', 'color atom  [102,102,255]', 'label "Hi"']
sage: T = Text("Hi").translate(-1, 0, 0) + Text("Bye").translate(1, 0, 0)
sage: T.jmol_repr(T.testing_render_params())
[[['select atomno = 1', 'color atom  [102,102,255]', 'label "Hi"']],
 [['select atomno = 2', 'color atom  [102,102,255]', 'label "Bye"']]]
```

obj_repr(*render_params*)

The obj file format does not support text strings:

```
sage: from sage.plot.plot3d.shapes import Text
sage: Text("Hi").obj_repr(None)
''
```

tachyon_repr(*render_params*)

Strings are not yet supported in Tachyon, so we ignore them for now:

```
sage: from sage.plot.plot3d.shapes import Text
sage: Text("Hi").tachyon_repr(None)
''
```

threejs_repr(*render_params*)

Return representation of the text suitable for plotting in three.js.

EXAMPLES:

```
sage: T = text3d("Hi", (1, 2, 3), color='red', fontfamily='serif',
....:            fontweight='bold', fontstyle='italic', fontsize=20,
....:            opacity=0.5)
sage: T.threejs_repr(T.default_render_params())
[('text',
  {'color': '#ff0000',
   'fontFamily': ['serif'],
   'fontSize': 20.0,
   'fontStyle': 'italic',
   'fontWeight': 'bold',
   'opacity': 0.5,
   'text': 'Hi',
   'x': 1.0,
   'y': 2.0,
   'z': 3.0})]
```

x3d_geometry()

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Text
sage: Text("Hi").x3d_geometry()
"<Text string='Hi' solid='true'/>"
```

class sage.plot.plot3d.shapes.**Torus**

Bases: *ParametricSurface*

INPUT:

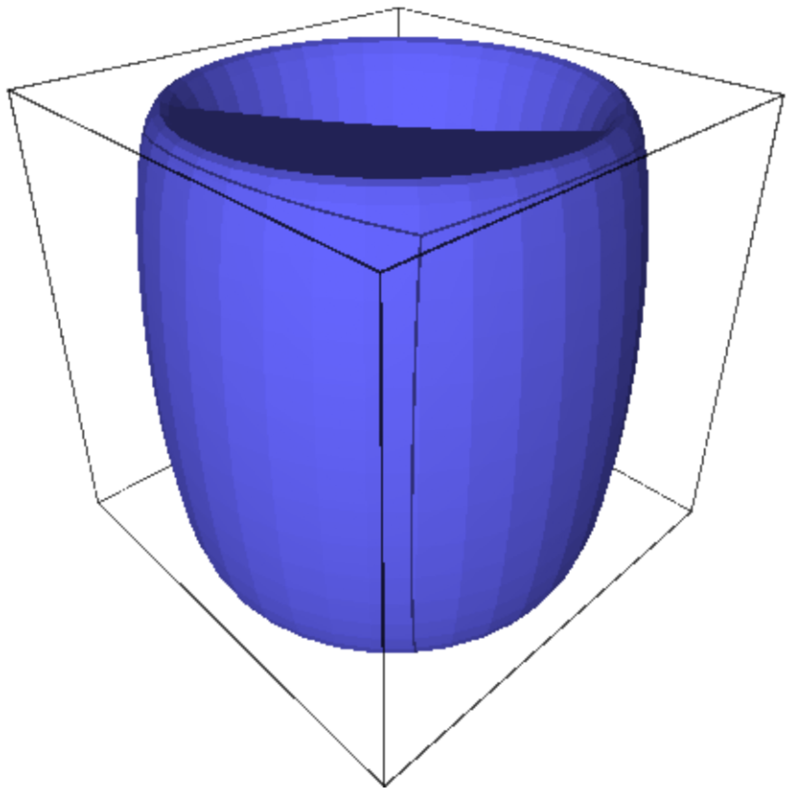
- R – (default: 1) outer radius
- r – (default: .3) inner radius

OUTPUT:

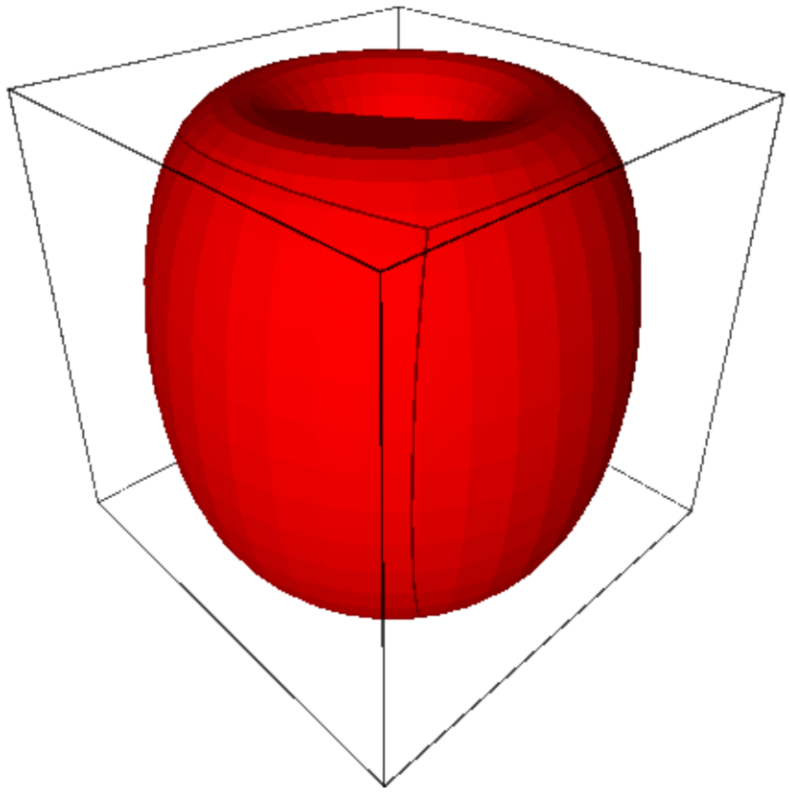
a 3d torus

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Torus
sage: Torus(1, .2).show(aspect_ratio=1)
```



```
sage: Torus(1, .7, color='red').show(aspect_ratio=1)
```



A rubberband ball:

```
sage: show(sum(Torus(1, .03, color=(1, t/30.0, 0)).rotate((1,1,1), t)
....:          for t in range(30)))
```

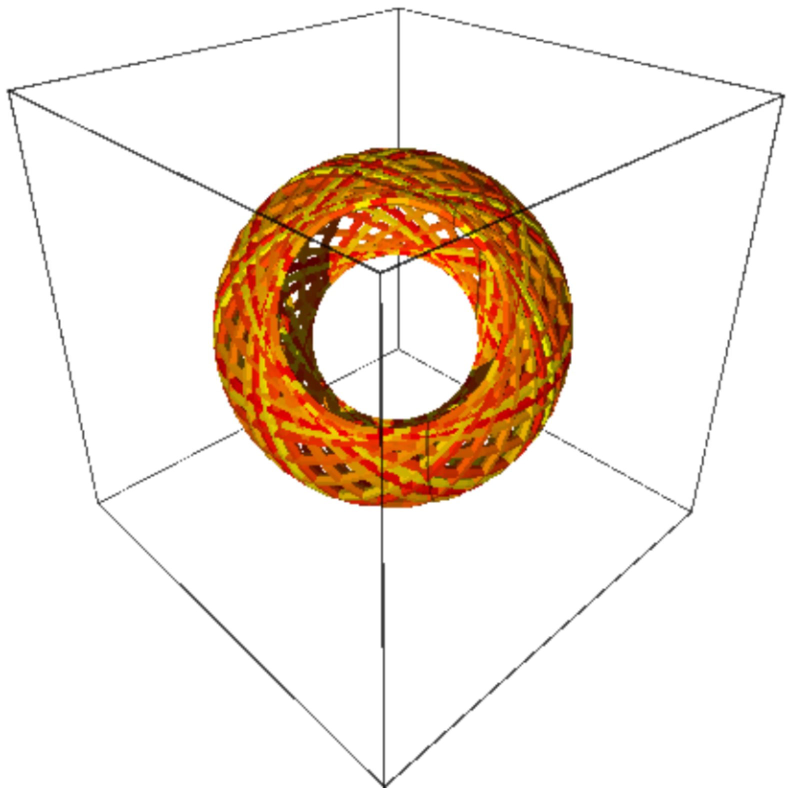
Mmm… doughnuts:

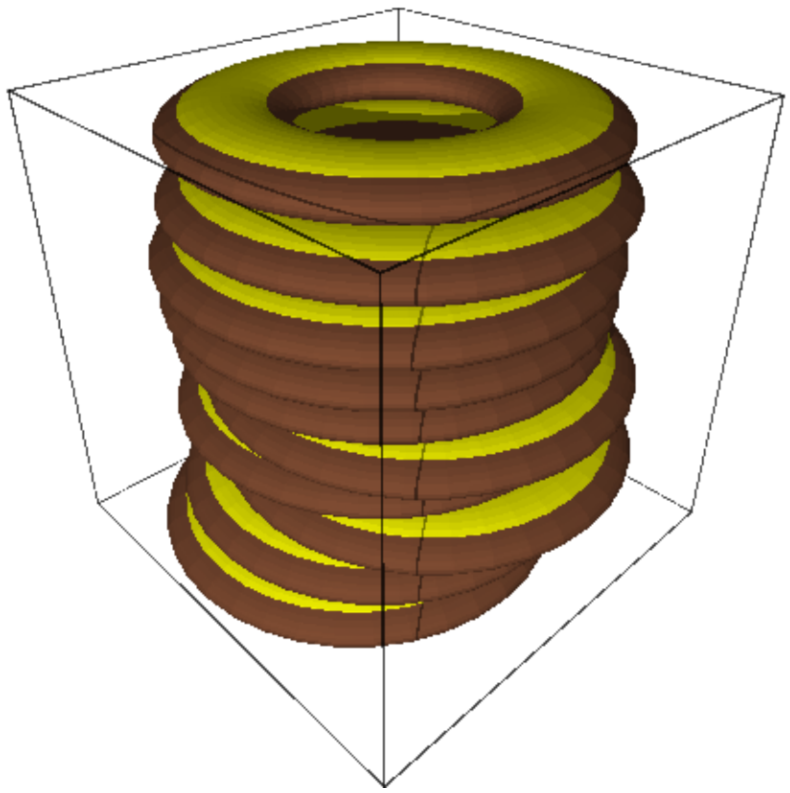
```
sage: D = Torus(1, .4, color=(.5, .3, .2))
sage: D += Torus(1, .3, color='yellow').translate(0, 0, .15)
sage: G = sum(D.translate(RDF.random_element(-.2, .2),
....:                     RDF.random_element(-.2, .2),
....:                     .8*t)
....:         for t in range(10))
sage: G.show(aspect_ratio=1, frame=False)
```

get_grid(*ds*)

Return the range of variables to be evaluated on to render as a parametric surface.

EXAMPLES:





```
sage: from sage.plot.plot3d.shapes import Torus
sage: Torus(2, 1).get_grid(100)
([0.0, -1.047..., -3.141592653589793, ..., 0.0],
 [0.0, 1.047..., 3.141592653589793, ..., 0.0])
```

sage.plot.plot3d.shapes.**arrow3d**(*start*, *end*, *width=1*, *radius=None*, *head_radius=None*, *head_len=None*, ***kwds*)

Create a 3d arrow.

INPUT:

- `start` – (x,y,z) point; the starting point of the arrow
- `end` – (x,y,z) point; the end point
- `width` – (default: 1); how wide the arrow is
- `radius` – (default: `width/50.0`) the radius of the arrow
- `head_radius` – (default: `3*radius`); radius of arrow head
- `head_len` – (default: `3*head_radius`); len of arrow head

EXAMPLES:

The default arrow:

```
sage: arrow3d((0,0,0), (1,1,1), 1)
Graphics3d Object
```

A fat arrow:

```
sage: arrow3d((0,0,0), (1,1,1), radius=0.1)
Graphics3d Object
```

A green arrow:

```
sage: arrow3d((0,0,0), (1,1,1), color='green')
Graphics3d Object
```

A fat arrow head:

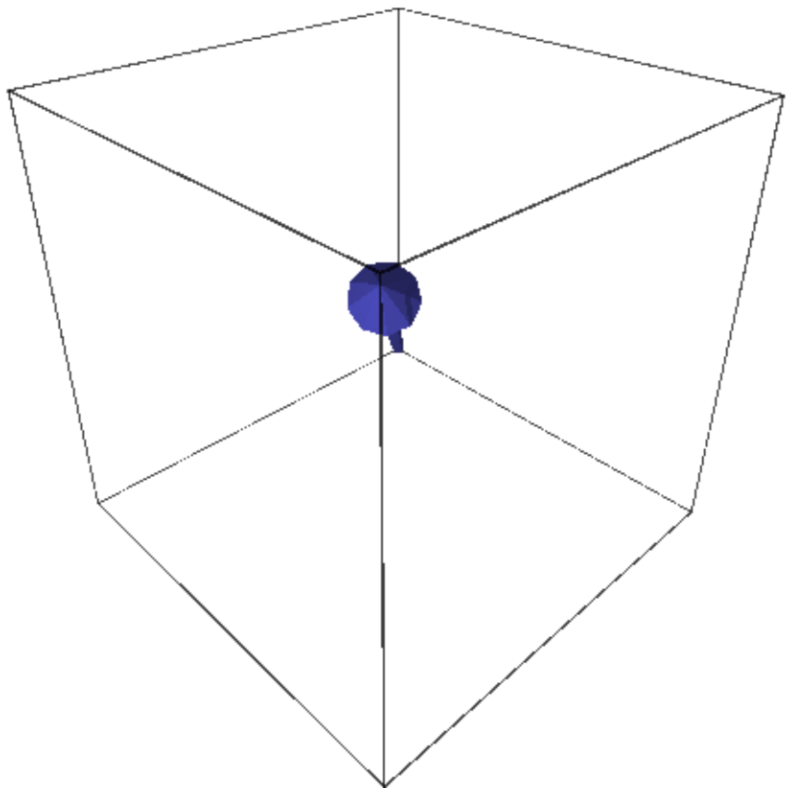
```
sage: arrow3d((2,1,0), (1,1,1), color='green', head_radius=0.3,
....:         aspect_ratio=[1,1,1])
Graphics3d Object
```

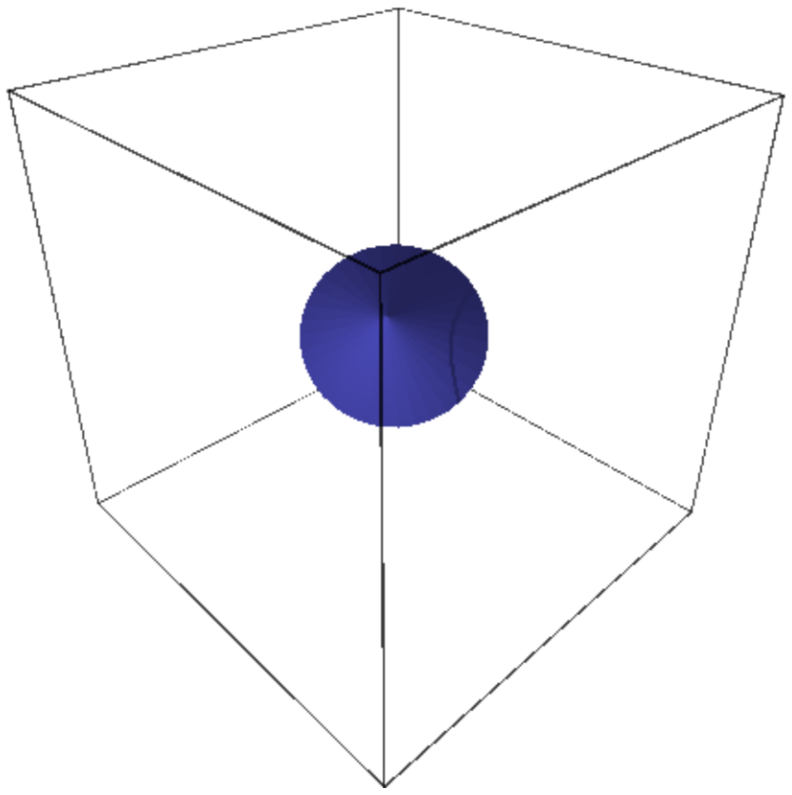
Many arrows arranged in a circle (flying spears?):

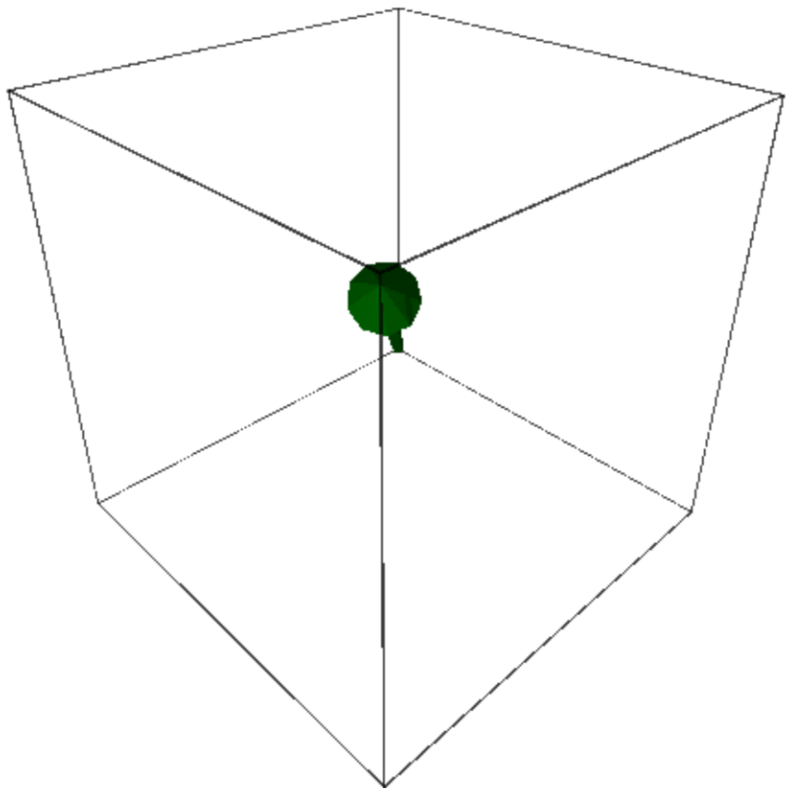
```
sage: sum(arrow3d((cos(t),sin(t),0), (cos(t),sin(t),1))                    #␣
↪needs sage.symbolic
....:     for t in [0,0.3,..,2*pi])
Graphics3d Object
```

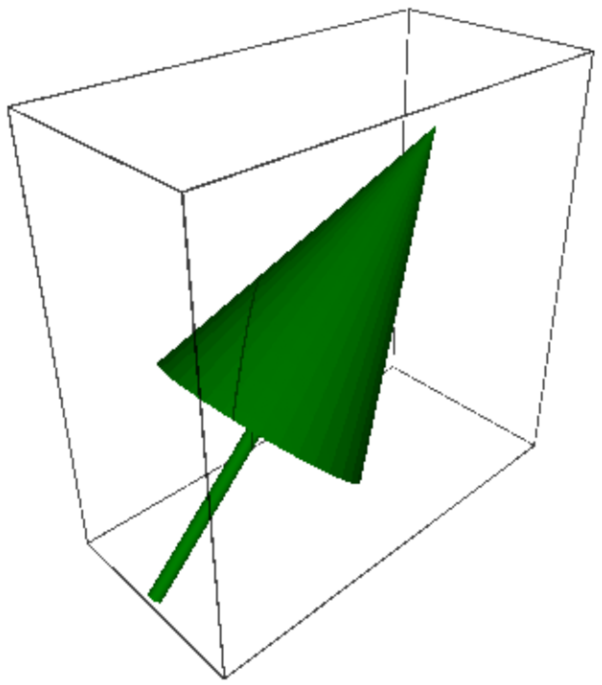
Change the width of the arrow. (Note: for an arrow that scales with zoom, please consider the *`line3d()`* function with the option `arrow_head=True`):

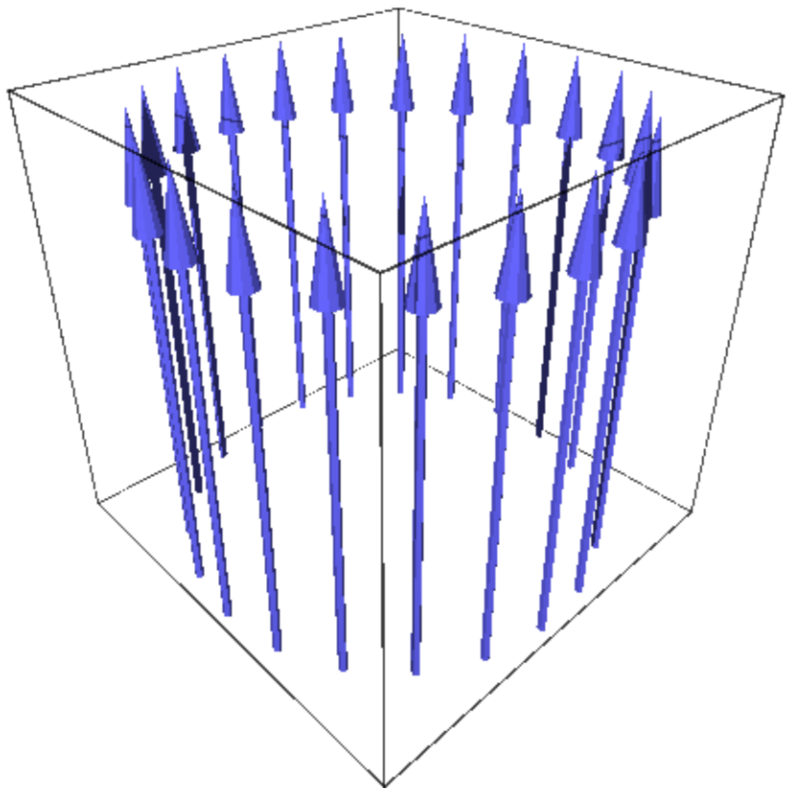
```
sage: arrow3d((0,0,0), (1,1,1), width=1)
Graphics3d Object
```

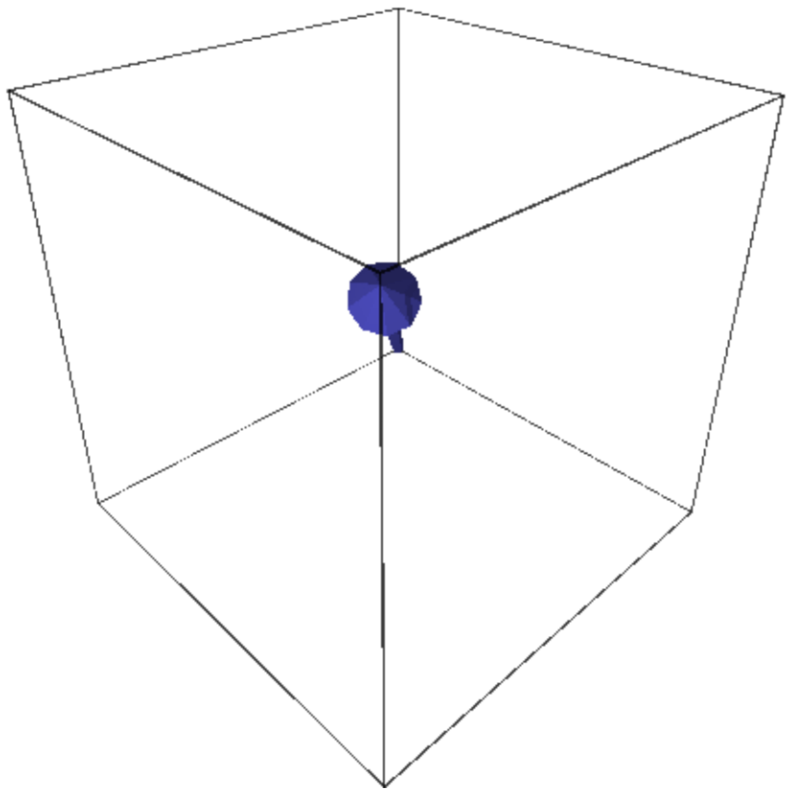












sage.plot.plot3d.shapes.**validate_frame_size**(*size*)

Check that the input is an iterable of length 3 with all elements nonnegative and coercible to floats.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import validate_frame_size
sage: validate_frame_size([3,2,1])
[3.0, 2.0, 1.0]
```

3.3 Classes for Lines, Frames, Rulers, Spheres, Points, Dots, and Text

AUTHORS:

- William Stein (2007-12): initial version
- William Stein and Robert Bradshaw (2008-01): Many improvements

class sage.plot.plot3d.shapes2.**Line**(*points*, *thickness=5*, *corner_cutoff=0.5*, *arrow_head=False*, ***kwds*)

Bases: *PrimitiveObject*

Draw a 3d line joining a sequence of points.

This line has a fixed diameter unaffected by transformations and zooming. It may be smoothed if `corner_cutoff < 1`.

INPUT:

- `points` – list of points to pass through
- `thickness` – (optional, default 5) diameter of the line
- `corner_cutoff` – (optional, default 0.5) threshold for smoothing (see *corners()*).
- `arrow_head` – (optional, default `False`) if `True` make this curve into an arrow

The parameter `corner_cutoff` is a bound for the cosine of the angle made by two successive segments. This angle is close to 0 (and the cosine close to 1) if the two successive segments are almost aligned and close to π (and the cosine close to -1) if the path has a strong peak. If the cosine is smaller than the bound (which means a sharper peak) then no smoothing is done.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes2 import Line
sage: Line([(i*math.sin(i), i*math.cos(i), i/3) for i in range(30)],
....:      arrow_head=True)
Graphics3d Object
```

Smooth angles less than 90 degrees:

```
sage: Line([(0,0,0),(1,0,0),(2,1,0),(0,1,0)], corner_cutoff=0)
Graphics3d Object
```

Make sure that the `corner_cutoff` keyword works (github issue #3859):

```
sage: N = 11
sage: c = 0.4
sage: sum(Line([(i,1,0), (i,0,0), (i,cos(2*pi*i/N), sin(2*pi*i/N))],          #␣
↪needs sage.symbolic
....:          corner_cutoff=c,
....:          color='red' if -cos(2*pi*i/N)<=c else 'blue')
....:     for i in range(N+1))
Graphics3d Object
```

bounding_box()

Return the lower and upper corners of a 3-D bounding box for `self`.

This is used for rendering and `self` should fit entirely within this box. In this case, we return the highest and lowest values of each coordinate among all points.

corners(*corner_cutoff=None*, *max_len=None*)

Figure out where the curve turns too sharply to pretend it is smooth.

INPUT:

- `corner_cutoff` – (optional, default `None`) If the cosine of the angle between adjacent line segments is smaller than this bound, then there will be a sharp corner in the path. Otherwise, the path is smoothed. If `None`, then the default value 0.5 is used.
- `max_len` – (optional, default `None`) Maximum number of points allowed in a single path. If this is set, this creates corners at smooth points in order to break the path into smaller pieces.

The parameter `corner_cutoff` is a bound for the cosine of the angle made by two successive segments. This angle is close to 0 (and the cosine close to 1) if the two successive segments are almost aligned and close to π (and the cosine close to -1) if the path has a strong peak. If the cosine is smaller than the bound (which means a sharper peak) then there must be a corner.

OUTPUT:

List of points at which to start a new line. This always includes the first point, and never the last.

EXAMPLES:

No corners, always smooth:

```
sage: from sage.plot.plot3d.shapes2 import Line
sage: Line([(0,0,0),(1,0,0),(2,1,0),(0,1,0)], corner_cutoff=-1).corners()
[(0, 0, 0)]
```

Smooth if the angle is greater than 90 degrees:

```
sage: Line([(0,0,0),(1,0,0),(2,1,0),(0,1,0)], corner_cutoff=0).corners()
[(0, 0, 0), (2, 1, 0)]
```

Every point (corners everywhere):

```
sage: Line([(0,0,0),(1,0,0),(2,1,0),(0,1,0)], corner_cutoff=1).corners()
[(0, 0, 0), (1, 0, 0), (2, 1, 0)]
```

jmol_repr(*render_params*)

Return representation of the object suitable for plotting using Jmol.

obj_repr(*render_params*)

Return complete representation of the line as an object.

stl_binary_repr(*render_params*)

Return an empty list, as this is not useful for STL export.

EXAMPLES:

```
sage: L = line3d([(1,2,3), (4,5,6)]).translate(-1, -2, -3)
sage: L.stl_binary_repr(L.default_render_params())
[]
```

tachyon_repr(*render_params*)

Return representation of the line suitable for plotting using the Tachyon ray tracer.

threejs_repr(*render_params*)

Return representation of the line suitable for plotting with three.js.

EXAMPLES:

```
sage: L = line3d([(1,2,3), (4,5,6)], thickness=10, color=(1,0,0), opacity=0.5)
sage: L.threejs_repr(L.default_render_params())
[('line',
  {'color': '#ff0000',
   'linewidth': 10.0,
   'opacity': 0.5,
   'points': [(1.0, 2.0, 3.0), (4.0, 5.0, 6.0)]})]
```

class sage.plot.plot3d.shapes2.**Point**(*center*, *size=1*, ***kwds*)

Bases: `PrimitiveObject`

Create a position in 3-space, represented by a sphere of fixed size.

INPUT:

- `center` – point (3-tuple)
- `size` – (default: 1)

EXAMPLES:

We normally access this via the `point3d` function. Note that extra keywords are correctly used:

```
sage: point3d((4,3,2),size=2,color='red',opacity=.5)
Graphics3d Object
```

bounding_box()

Returns the lower and upper corners of a 3-D bounding box for `self`.

This is used for rendering and `self` should fit entirely within this box. In this case, we simply return the center of the point.

jmol_repr(*render_params*)

Return representation of the object suitable for plotting using Jmol.

obj_repr(*render_params*)

Return complete representation of the point as a sphere.

`stl_binary_repr`(*render_params*)

Return an empty list, as this is not useful for STL export.

EXAMPLES:

```
sage: P = point3d((1,2,3)).translate(-1, -2, -3)
sage: P.stl_binary_repr(P.default_render_params())
[]
```

`tachyon_repr`(*render_params*)

Return representation of the point suitable for plotting using the Tachyon ray tracer.

`threejs_repr`(*render_params*)

Return representation of the point suitable for plotting with three.js.

EXAMPLES:

```
sage: P = point3d((1,2,3), color=(0,1,0), opacity=0.5, size=10)
sage: P.threejs_repr(P.default_render_params())
[('point',
  {'color': '#00ff00', 'opacity': 0.5, 'point': (1.0, 2.0, 3.0), 'size': 10.0})]
```

`sage.plot.plot3d.shapes2.bezier3d`(*path*, *opacity=1*, *color='blue'*, *aspect_ratio=[1, 1, 1]*, *thickness=2*, ***options*)

Draw a 3-dimensional bezier path.

Input is similar to bezier_path, but each point in the path and each control point is required to have 3 coordinates.

INPUT:

- **`path` – a list of curves, which each is a list of points. See further**
  detail below.
- `thickness` – (default: 2)
- `color` – a string (`"red"`, `"green"` etc) or a tuple (r, g, b) with r, g, b numbers between 0 and 1
- `opacity` – (default: 1) if less than 1 then is transparent
- `aspect_ratio` – (default: [1,1,1])

The path is a list of curves, and each curve is a list of points. Each point is a tuple (x,y,z).

The first curve contains the endpoints as the first and last point in the list. All other curves assume a starting point given by the last entry in the preceding list, and take the last point in the list as their opposite endpoint. A curve can have 0, 1 or 2 control points listed between the endpoints. In the input example for path below, the first and second curves have 2 control points, the third has one, and the fourth has no control points:

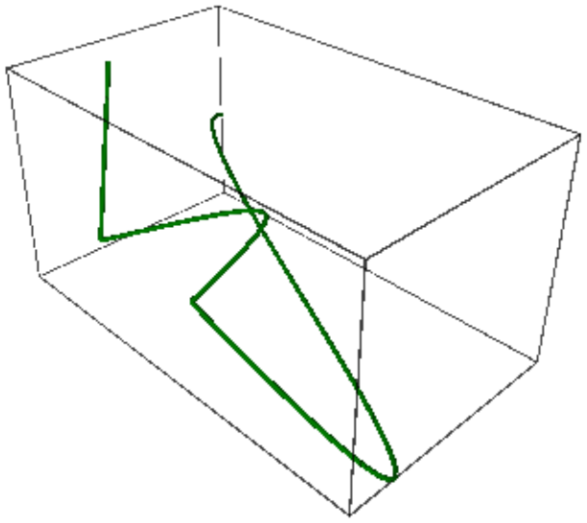
```
path = [[p1, c1, c2, p2], [c3, c4, p3], [c5, p4], [p5], ...]
```

In the case of no control points, a straight line will be drawn between the two endpoints. If one control point is supplied, then the curve at each of the endpoints will be tangent to the line from that endpoint to the control point. Similarly, in the case of two control points, at each endpoint the curve will be tangent to the line connecting that endpoint with the control point immediately after or immediately preceding it in the list.

So in our example above, the curve between p1 and p2 is tangent to the line through p1 and c1 at p1, and tangent to the line through p2 and c2 at p2. Similarly, the curve between p2 and p3 is tangent to line(p2,c3) at p2 and tangent to line(p3,c4) at p3. Curve(p3,p4) is tangent to line(p3,c5) at p3 and tangent to line(p4,c5) at p4. Curve(p4,p5) is a straight line.

EXAMPLES:

```
sage: path = [[(0,0,0),(.5,.1,.2),(.75,3,-1),(1,1,0)],
....:         [(.5,1,.2),(1,.5,0)], [(.7,.2,.5)]]
sage: b = bezier3d(path, color='green'); b                                  #␣
↪needs sage.symbolic
Graphics3d Object
```



To construct a simple curve, create a list containing a single list:

```
sage: path = [[(0,0,0),(1,0,0),(0,1,0),(0,1,1)]]
sage: curve = bezier3d(path, thickness=5, color='blue'); curve              #␣
↪needs sage.symbolic
Graphics3d Object
```

sage.plot.plot3d.shapes2.**frame3d**(*lower_left*, *upper_right*, ***kwds*)

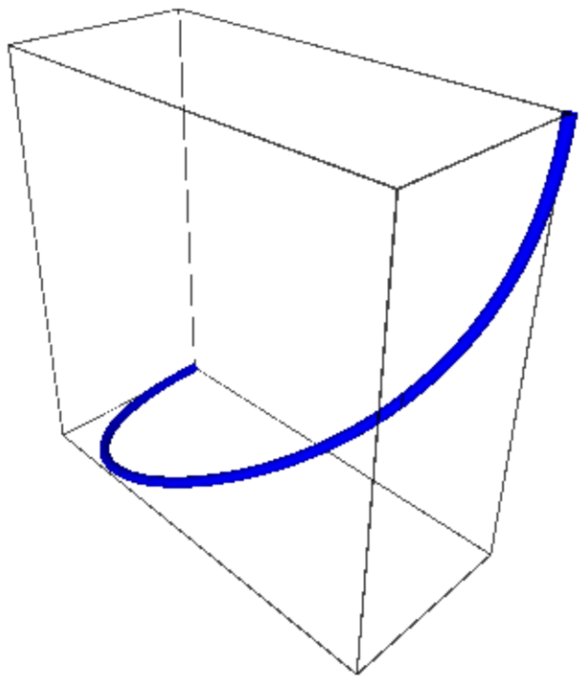
Draw a frame in 3-D.

Primarily used as a helper function for creating frames for 3-D graphics viewing.

INPUT:

- `lower_left` – the lower left corner of the frame, as a list, tuple, or vector.
- `upper_right` – the upper right corner of the frame, as a list, tuple, or vector.

EXAMPLES:



A frame:

```
sage: from sage.plot.plot3d.shapes2 import frame3d
sage: frame3d([1,3,2],vector([2,5,4]),color='red')
Graphics3d Object
```

This is usually used for making an actual plot:

```
sage: y = var('y')                                                          #␣
↪needs sage.symbolic
sage: plot3d(sin(x^2+y^2), (x,0,pi), (y,0,pi))                              #␣
↪needs sage.symbolic
Graphics3d Object
```

sage.plot.plot3d.shapes2.**frame_labels**(*lower_left*, *upper_right*, *label_lower_left*, *label_upper_right*, *eps=1*, ***kwds*)

Draw correct labels for a given frame in 3-D.

Primarily used as a helper function for creating frames for 3-D graphics viewing - do not use directly unless you know what you are doing!

INPUT:

- `lower_left` – the lower left corner of the frame, as a list, tuple, or vector.
- `upper_right` – the upper right corner of the frame, as a list, tuple, or vector.
- `label_lower_left` – the label for the lower left corner of the frame, as a list, tuple, or vector. This label must actually have all coordinates less than the coordinates of the other label.
- `label_upper_right` – the label for the upper right corner of the frame, as a list, tuple, or vector. This label must actually have all coordinates greater than the coordinates of the other label.
- `eps` – (default: 1) a parameter for how far away from the frame to put the labels.

EXAMPLES:

We can use it directly:

```
sage: from sage.plot.plot3d.shapes2 import frame_labels
sage: frame_labels([1,2,3],[4,5,6],[1,2,3],[4,5,6])
Graphics3d Object
```

This is usually used for making an actual plot:

```
sage: # needs sage.symbolic
sage: y = var('y')
sage: P = plot3d(sin(x^2+y^2), (x,0,pi), (y,0,pi))
sage: a,b = P._rescale_for_frame_aspect_ratio_and_zoom(1.0,[1,1,1],1)
sage: F = frame_labels(a, b, *P._box_for_aspect_ratio("automatic",a,b))
sage: F.jmol_repr(F.default_render_params())[0]
[['select atomno = 1', 'color atom  [76,76,76]', 'label "0.0"']]
```

sage.plot.plot3d.shapes2.**line3d**(*points*, *thickness=1*, *radius=None*, *arrow_head=False*, ***kwds*)

Draw a 3d line joining a sequence of points.

One may specify either a thickness or radius. If a thickness is specified, this line will have a constant diameter regardless of scaling and zooming. If a radius is specified, it will behave as a series of cylinders.

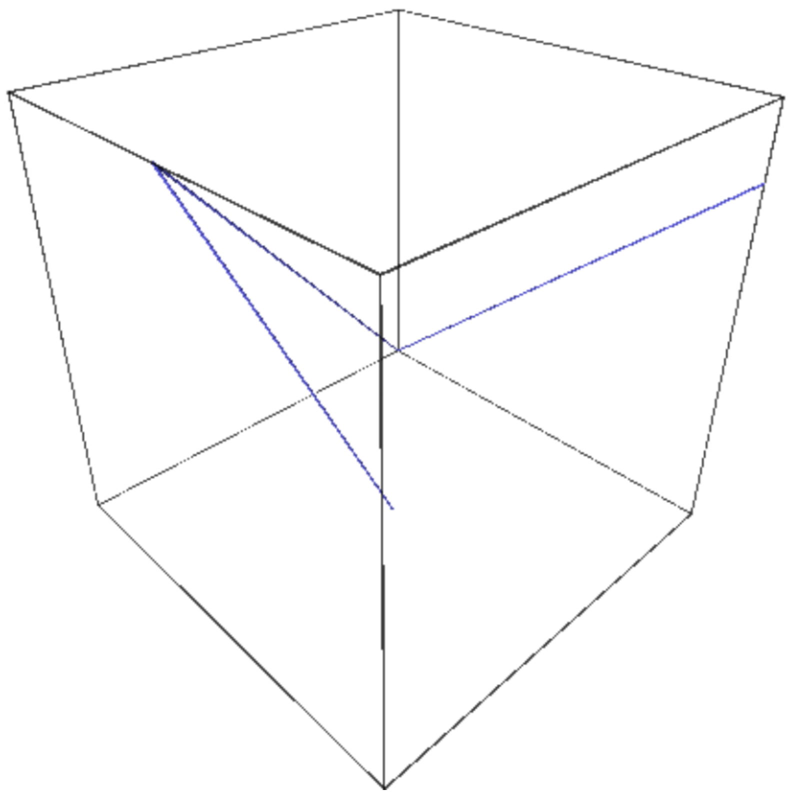
INPUT:

- `points` – a list of at least 2 points
- `thickness` – (default: 1)
- `radius` – (default: None)
- `arrow_head` – (default: False)
- `color` – a string (`"red"`, `"green"` etc) or a tuple (r, g, b) with r, g, b numbers between 0 and 1
- `opacity` – (default: 1) if less than 1 then is transparent

EXAMPLES:

A line in 3-space:

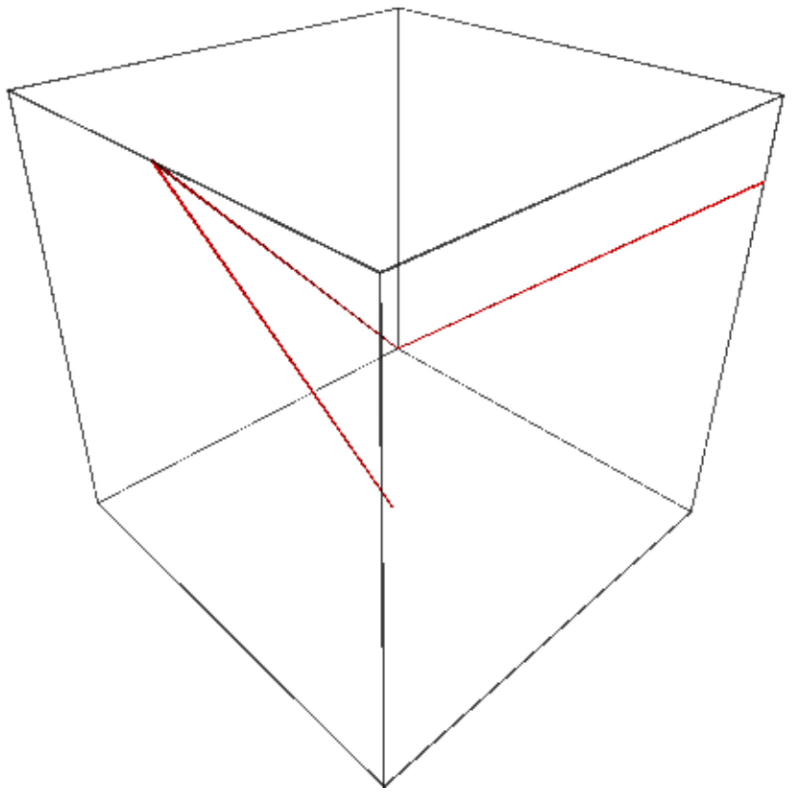
```
sage: line3d([(1,2,3), (1,0,-2), (3,1,4), (2,1,-2)])
Graphics3d Object
```



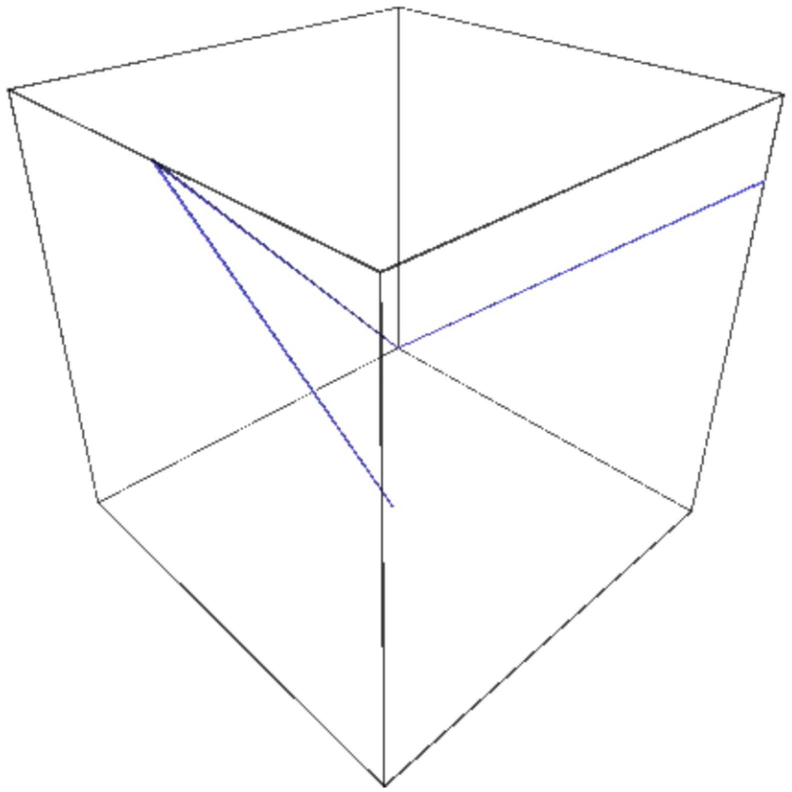
The same line but red:

```
sage: line3d([(1,2,3), (1,0,-2), (3,1,4), (2,1,-2)], color='red')
Graphics3d Object
```

The points of the line provided as a numpy array:



```
sage: import numpy
sage: line3d(numpy.array([(1,2,3), (1,0,-2), (3,1,4), (2,1,-2)]))
Graphics3d Object
```



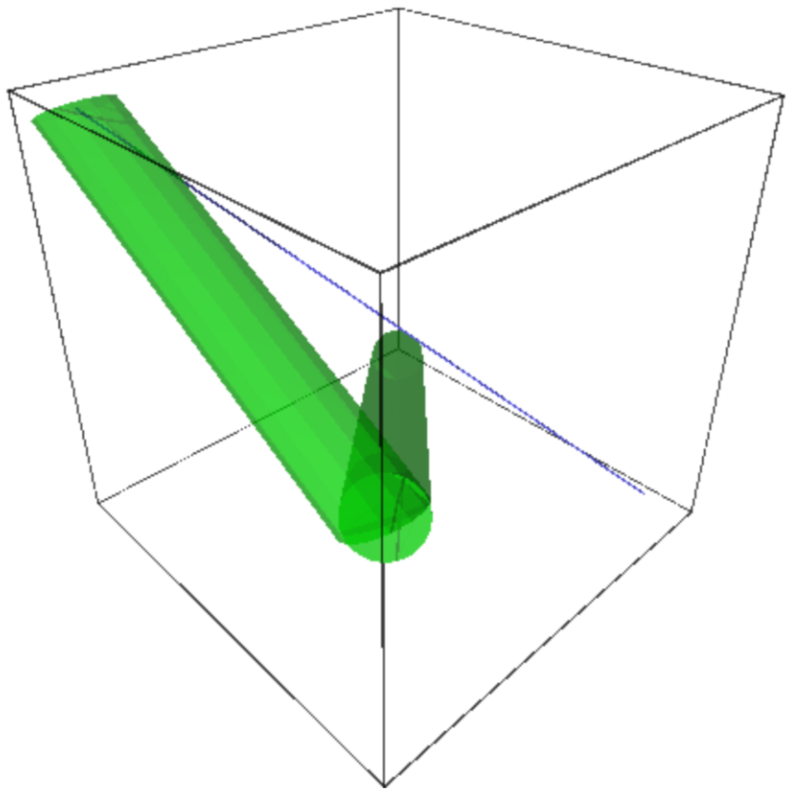
A transparent thick green line and a little blue line:

```
sage: line3d([(0,0,0), (1,1,1), (1,0,2)], opacity=0.5, radius=0.1,
....:        color='green') + line3d([(0,1,0), (1,0,2)])
Graphics3d Object
```

A Dodecahedral complex of 5 tetrahedra (a more elaborate example from Peter Jipsen):

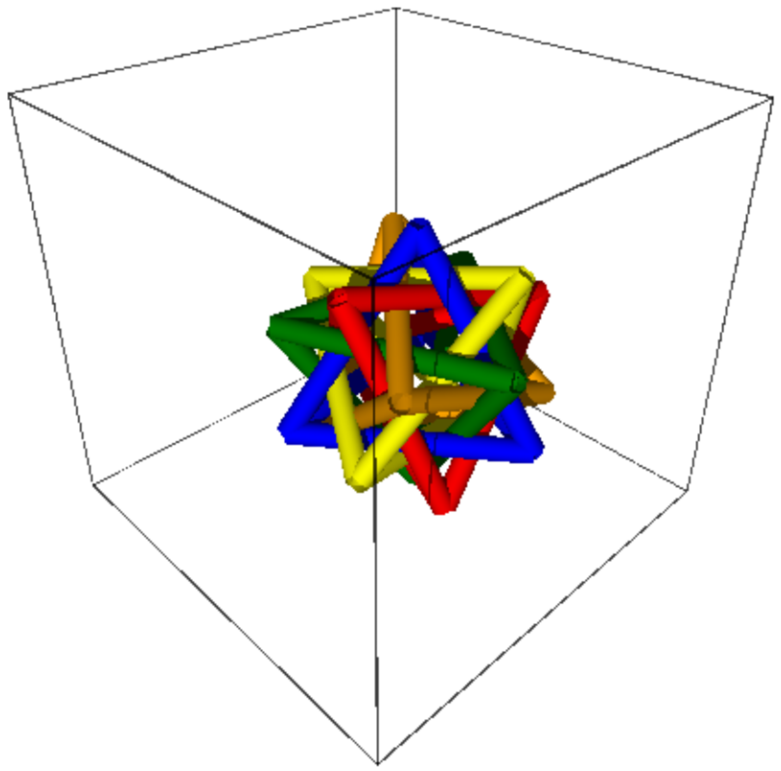
```
sage: def tetra(col):
....:     return line3d([(0,0,1), (2*sqrt(2.)/3,0,-1./3), (-sqrt(2.)/3, sqrt(6.)/3,-
↪1./3),\
....:            (-sqrt(2.)/3,-sqrt(6.)/3,-1./3), (0,0,1), (-sqrt(2.)/3, sqrt(6.)/3,
↪-1./3),\
....:            (-sqrt(2.)/3,-sqrt(6.)/3,-1./3), (2*sqrt(2.)/3,0,-1./3)],\
....:            color=col, thickness=10, aspect_ratio=[1,1,1])

sage: from math import pi
sage: v  = (sqrt(5.)/2-5/6, 5/6*sqrt(3.)-sqrt(15.)/2, sqrt(5.)/3)
sage: t  = acos(sqrt(5.)/3)/2
```

(continues on next page)



(continued from previous page)

```
sage: t1 = tetra('blue').rotateZ(t)
sage: t2 = tetra('red').rotateZ(t).rotate(v,2*pi/5)
sage: t3 = tetra('green').rotateZ(t).rotate(v,4*pi/5)
sage: t4 = tetra('yellow').rotateZ(t).rotate(v,6*pi/5)
sage: t5 = tetra('orange').rotateZ(t).rotate(v,8*pi/5)
sage: show(t1+t2+t3+t4+t5, frame=False)
```



`sage.plot.plot3d.shapes2.point3d`(*v*, *size=5*, ***kwds*)

Plot a point or list of points in 3d space.

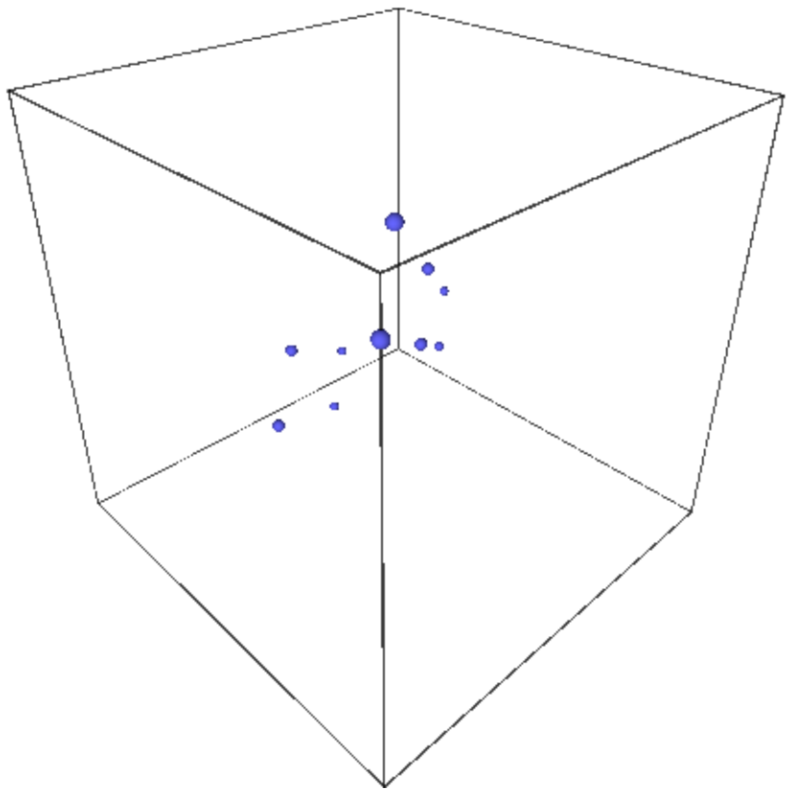
INPUT:

- `v` – a point or list of points
- `size` – (default: 5) size of the point (or points)
- `color` – a string (`"red"`, `"green"` etc) or a tuple (r, g, b) with r, g, b numbers between 0 and 1
- `opacity` – (default: 1) if less than 1 then is transparent

EXAMPLES:

```
sage: sum(point3d((i,i^2,i^3), size=5) for i in range(10))
Graphics3d Object
```

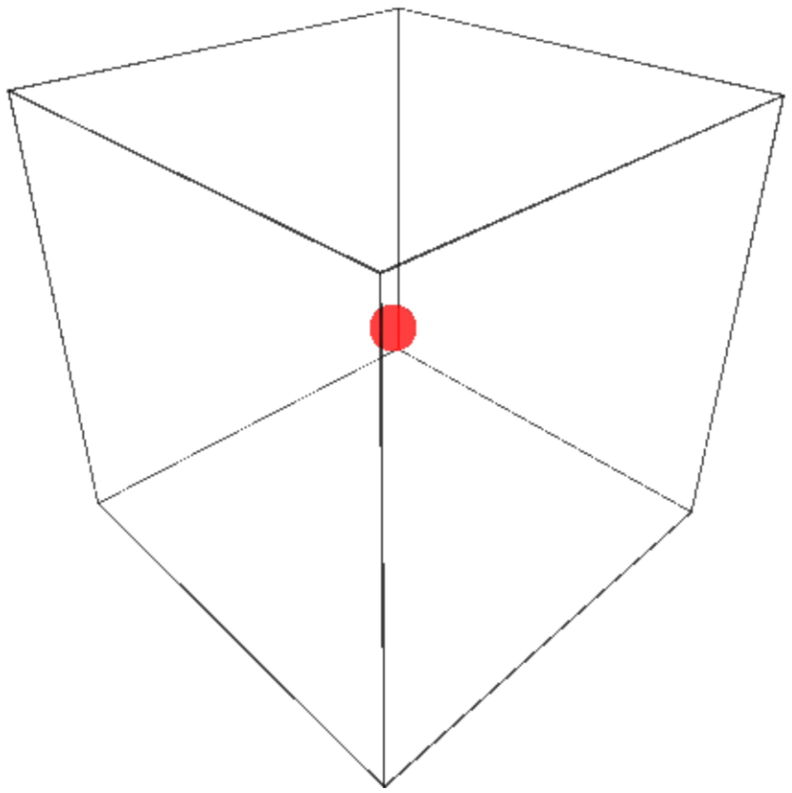
We check to make sure this works with vectors and other iterables:



```
sage: pl = point3d([vector(ZZ,(1, 0, 0)), vector(ZZ,(0, 1, 0)), (-1, -1, 0)])
sage: print(point(vector((2,3,4))))
Graphics3d Object

sage: c = polytopes.hypercube(3)                                        #␣
↪needs sage.geometry.polyhedron
sage: v = c.vertices()[0];  v                                           #␣
↪needs sage.geometry.polyhedron
A vertex at (1, -1, -1)
sage: print(point(v))                                                   #␣
↪needs sage.geometry.polyhedron
Graphics3d Object
```

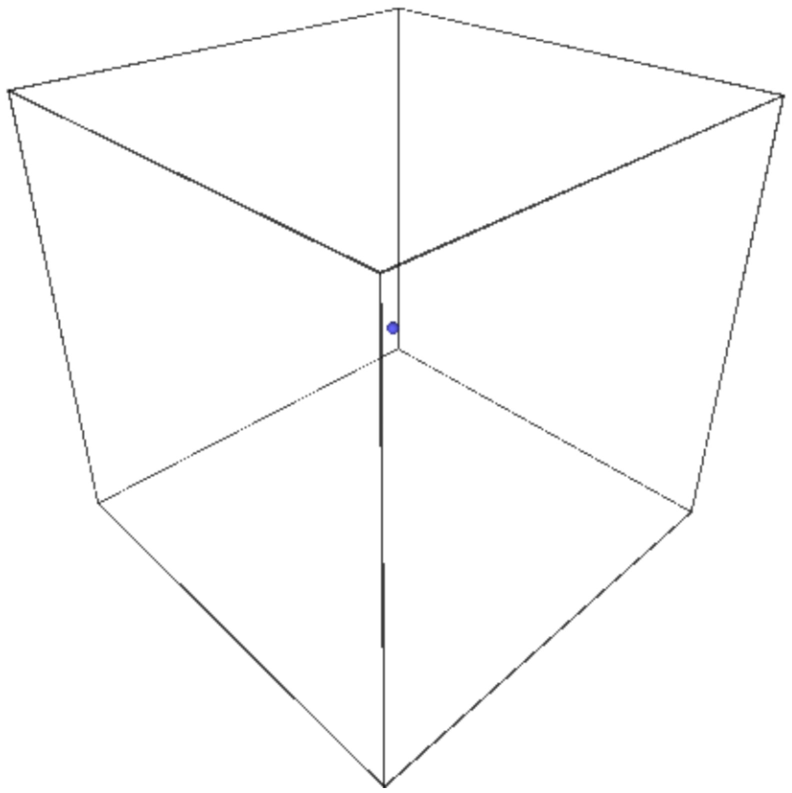
We check to make sure the options work:

```
sage: point3d((4,3,2), size=20, color='red', opacity=.5)
Graphics3d Object
```

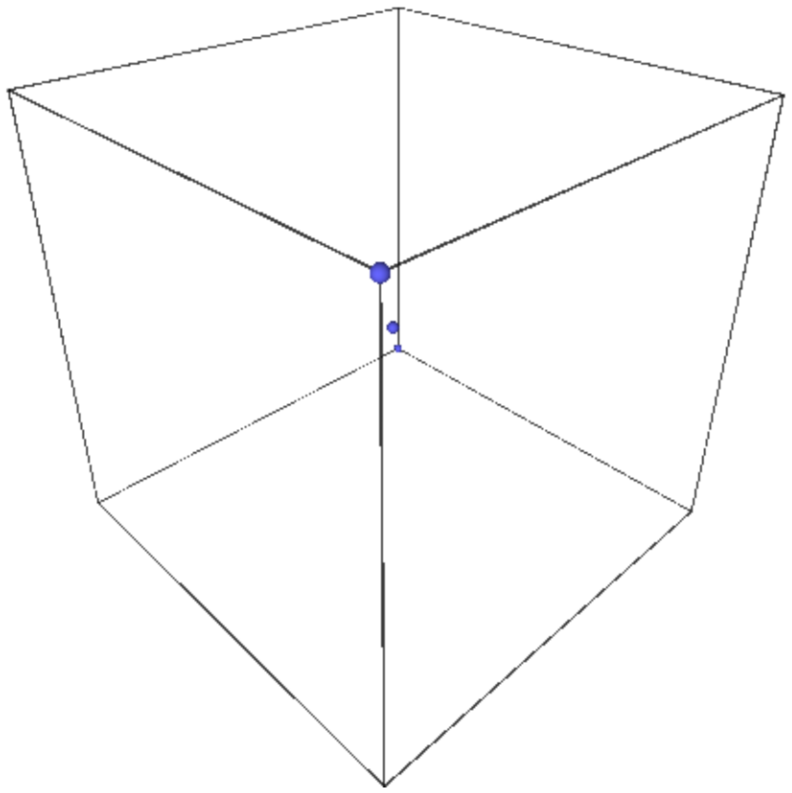


numpy arrays can be provided as input:

```
sage: import numpy
sage: point3d(numpy.array([1,2,3]))
Graphics3d Object
```



```
sage: point3d(numpy.array([[1,2,3], [4,5,6], [7,8,9]]))
Graphics3d Object
```



We check that iterators of points are accepted (github issue #13890):

```
sage: point3d(iter([(1,1,2),(2,3,4),(3,5,8)]), size=20, color='red')
Graphics3d Object
```

`sage.plot.plot3d.shapes2.`**`polygon3d`**(*points*, *opacity=1*, *color=(0, 0, 1)*, ***options*)

Draw a polygon in 3d.

INPUT:

- `points` – the vertices of the polygon

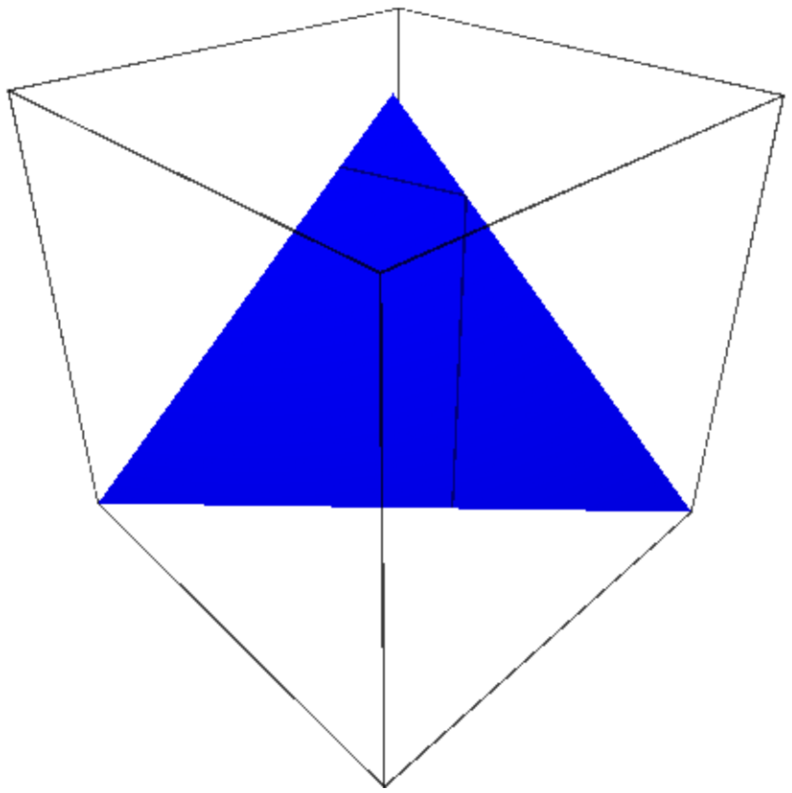
Type `polygon3d.options` for a dictionary of the default options for polygons. You can change this to change the defaults for all future polygons. Use `polygon3d.reset()` to reset to the default options.

EXAMPLES:

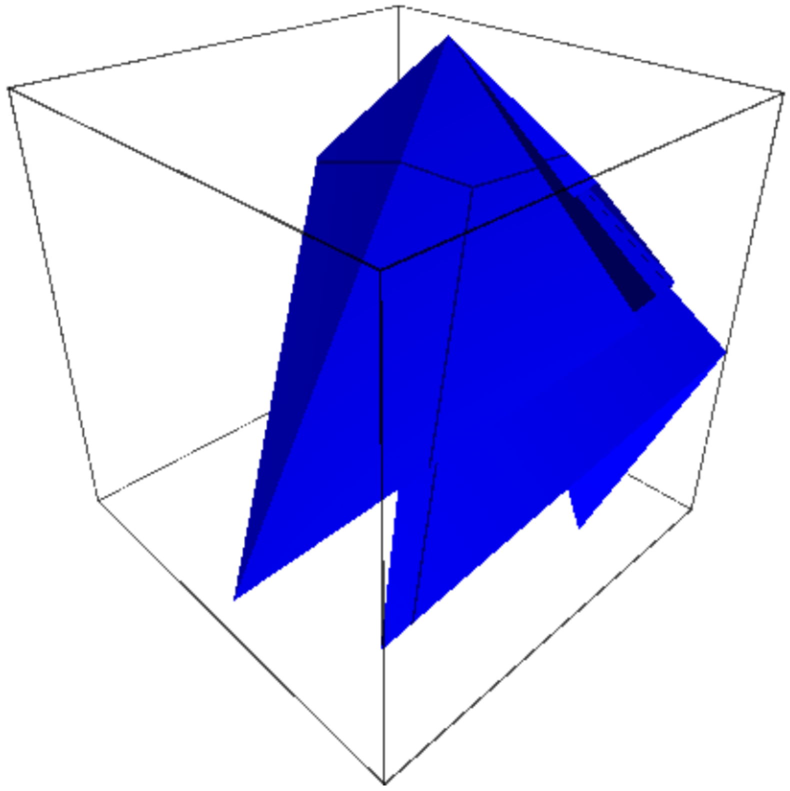
A simple triangle:

```
sage: polygon3d([[0,2,0], [1.5,1,3], [3,0,0]])
Graphics3d Object
```

Some modern art – a random polygon:



```
sage: v = [(randrange(-5,5), randrange(-5,5), randrange(-5, 5))
....:      for _ in range(10)]
sage: polygon3d(v)
Graphics3d Object
```



A bent transparent green triangle:

```
sage: polygon3d([[1, 2, 3], [0,1,0], [1,0,1], [3,0,0]],
....:           color=(0,1,0), opacity=0.7)
Graphics3d Object
```

This is the same as using `alpha=0.7`:

```
sage: polygon3d([[1, 2, 3], [0,1,0], [1,0,1], [3,0,0]],
....:           color=(0,1,0), alpha=0.7)
Graphics3d Object
```

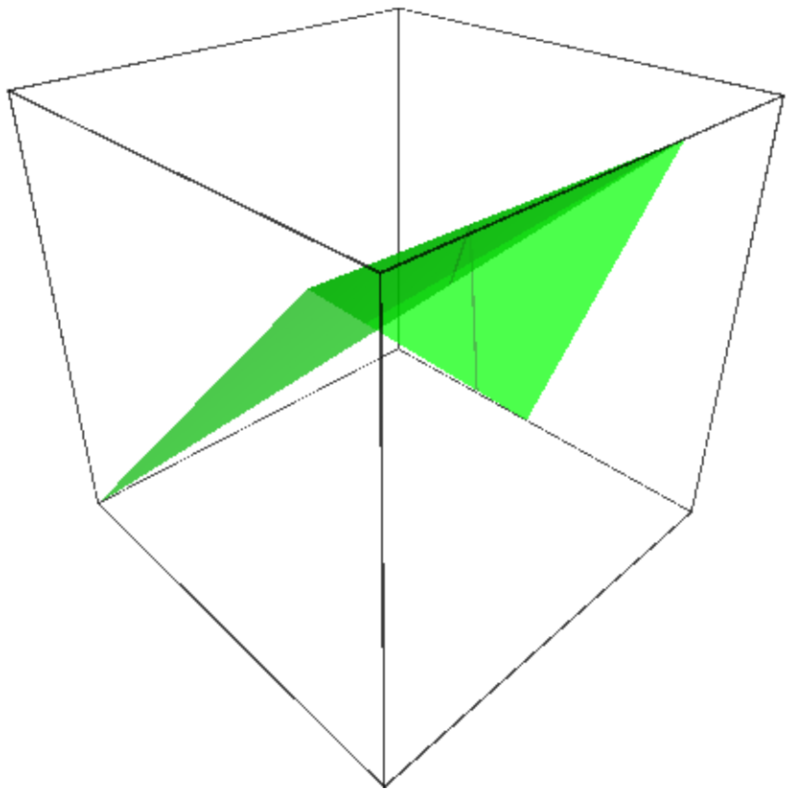
`sage.plot.plot3d.shapes2.`**`polygons3d`**(*faces*, *points*, *opacity=1*, *color=(0, 0, 1)*, ***options*)

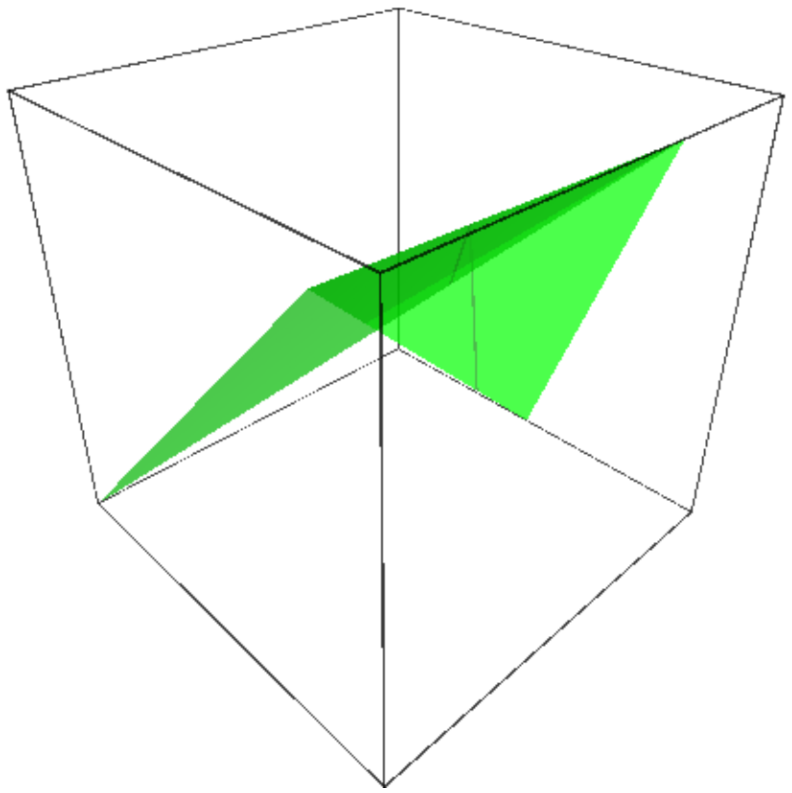
Draw the union of several polygons in 3d.

Useful to plot a polyhedron as just one `IndexFaceSet`.

INPUT:

- `faces` – list of faces, every face given by the list of indices of its vertices



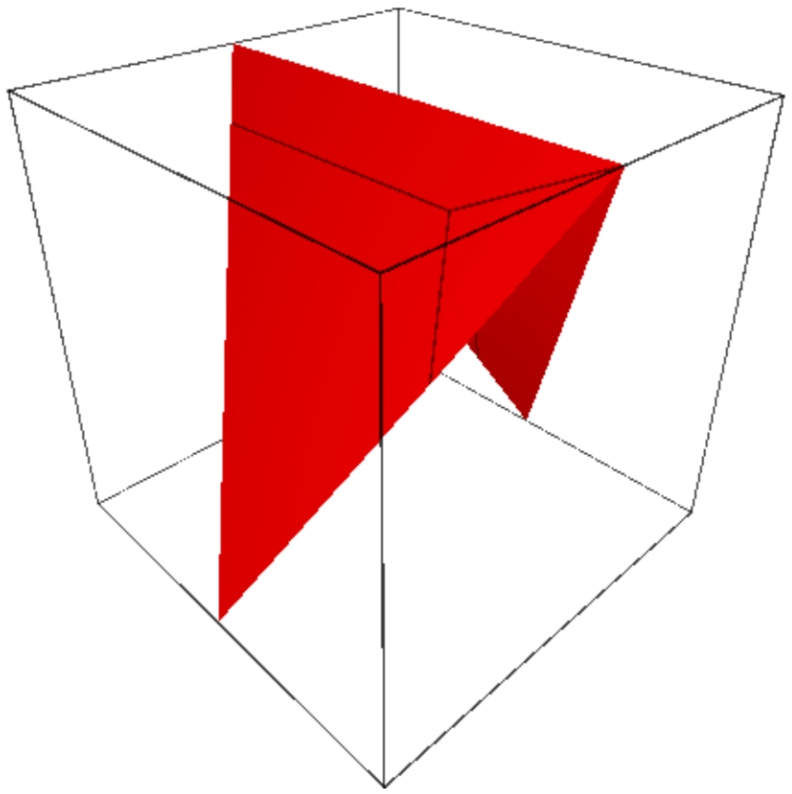


- `points` – coordinates of the vertices in the union

EXAMPLES:

Two adjacent triangles:

```
sage: f = [[0,1,2],[1,2,3]]
sage: v = [(-1,0,0),(0,1,1),(0,-1,1),(1,0,0)]
sage: polygons3d(f, v, color='red')
Graphics3d Object
```



`sage.plot.plot3d.shapes2.`**`ruler`**(*start*, *end*, *ticks=4*, *sub_ticks=4*, *absolute=False*, *snap=False*, ***kwds*)

Draw a ruler in 3-D, with major and minor ticks.

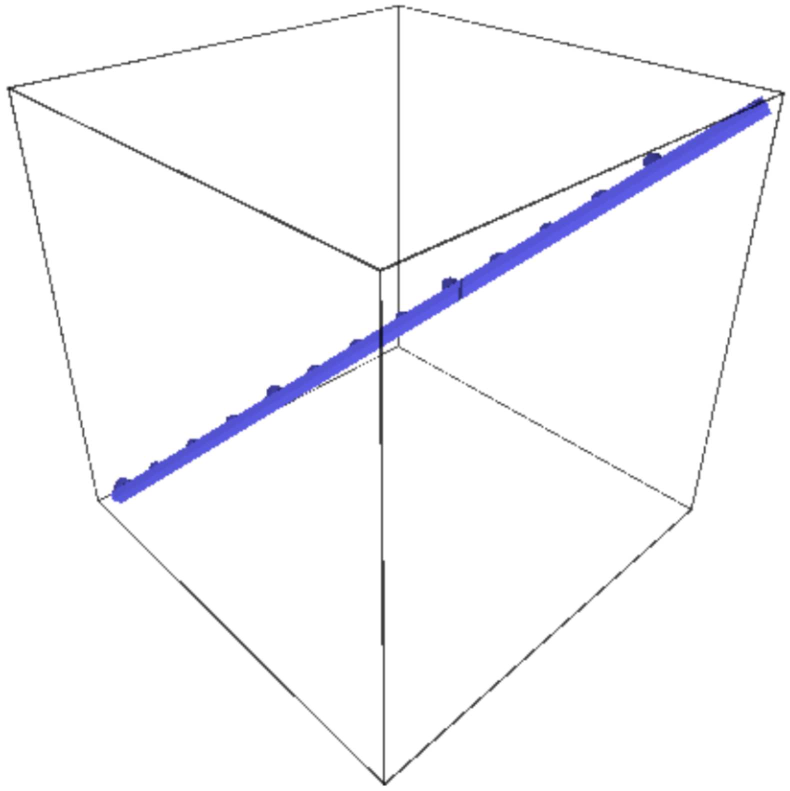
INPUT:

- `start` – the beginning of the ruler, as a list, tuple, or vector.
- `end` – the end of the ruler, as a list, tuple, or vector.
- `ticks` – (default: 4) the number of major ticks shown on the ruler.
- `sub_ticks` – (default: 4) the number of shown subdivisions between each major tick.
- `absolute` – (default: `False`) if `True`, makes a huge ruler in the direction of an axis.
- `snap` – (default: `False`) if `True`, snaps to an implied grid.

EXAMPLES:

A ruler:

```
sage: from sage.plot.plot3d.shapes2 import ruler
sage: R = ruler([4, 2, 1],vector([3, 3, 2])); R
Graphics3d Object
```



A ruler with some options:

```
sage: R = ruler([4, 2, 1],vector([3, 3, 2]),ticks=6, sub_ticks=2, color='red'); R
Graphics3d Object
```

The keyword `snap` makes the ticks not necessarily coincide with the ruler:

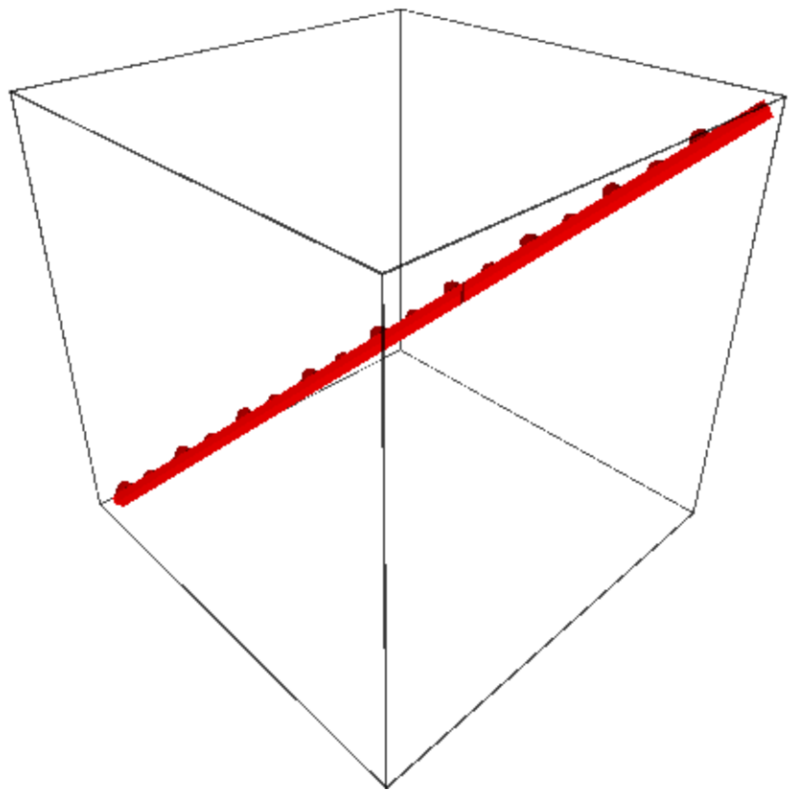
```
sage: ruler([4, 2, 1],vector([3, 3, 2]),snap=True)
Graphics3d Object
```

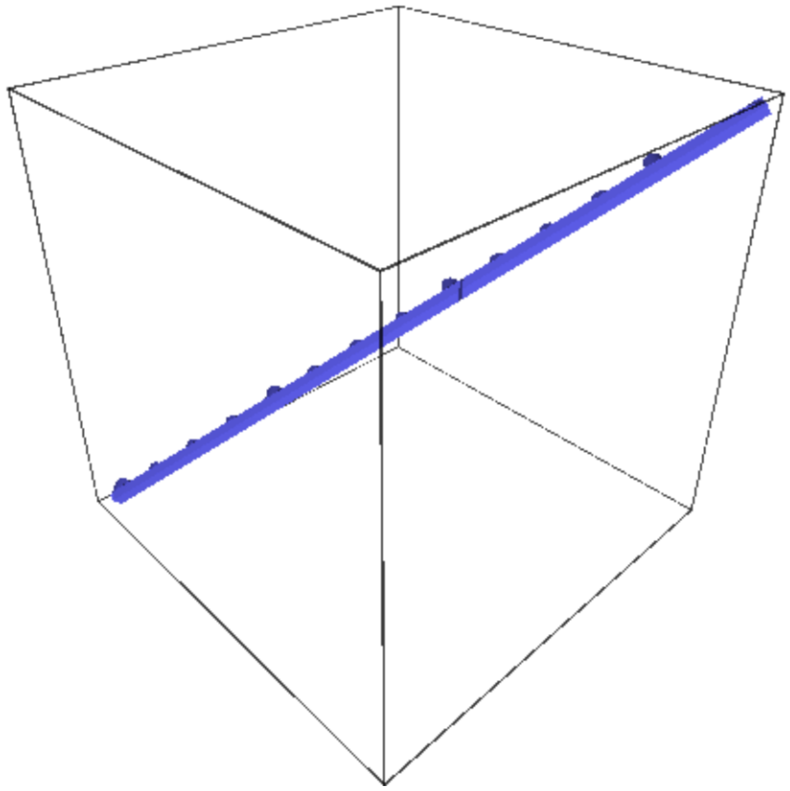
The keyword `absolute` makes a huge ruler in one of the axis directions:

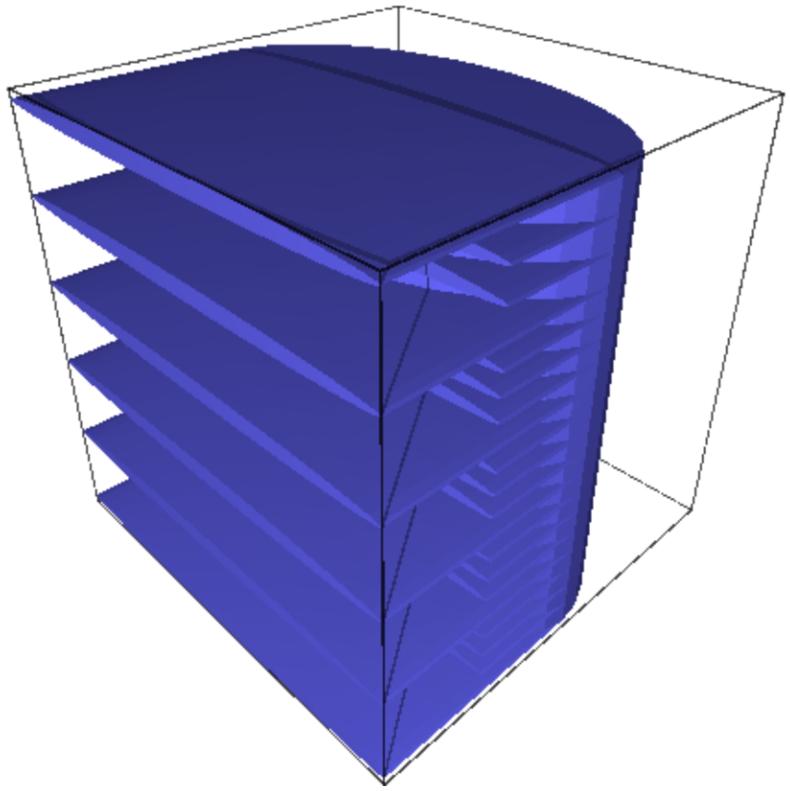
```
sage: ruler([1,2,3],vector([1,2,4]),absolute=True)
Graphics3d Object
```

`sage.plot.plot3d.shapes2.`**`ruler_frame`**(*lower_left*, *upper_right*, *ticks=4*, *sub_ticks=4*, ***kwds*)

Draw a frame made of 3-D rulers, with major and minor ticks.







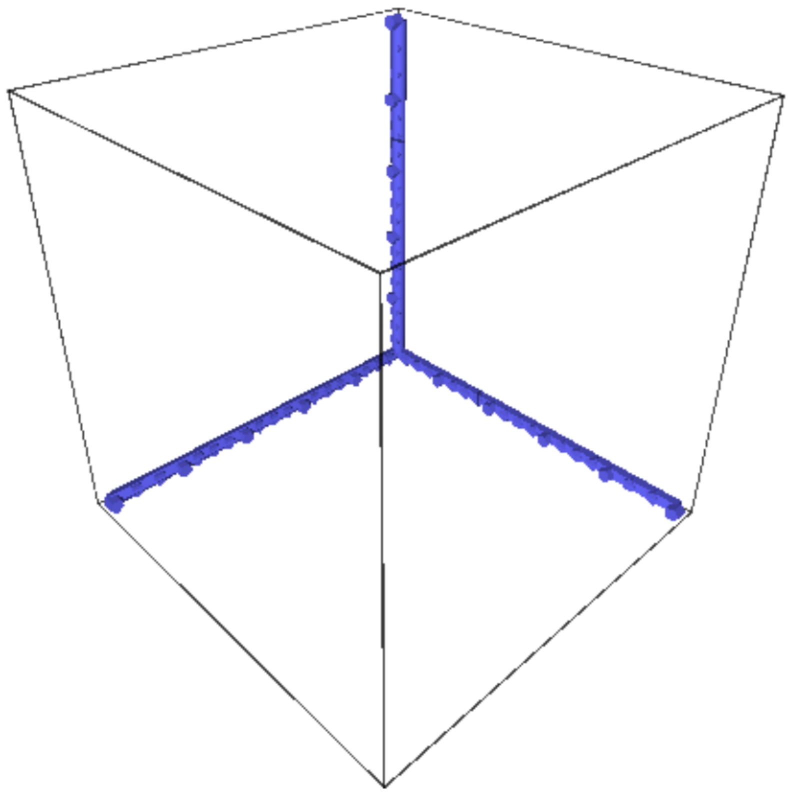
INPUT:

- `lower_left` – the lower left corner of the frame, as a list, tuple, or vector.
- `upper_right` – the upper right corner of the frame, as a list, tuple, or vector.
- `ticks` – (default: 4) the number of major ticks shown on each ruler.
- `sub_ticks` – (default: 4) the number of shown subdivisions between each major tick.

EXAMPLES:

A ruler frame:

```
sage: from sage.plot.plot3d.shapes2 import ruler_frame
sage: F = ruler_frame([1,2,3],vector([2,3,4])); F
Graphics3d Object
```



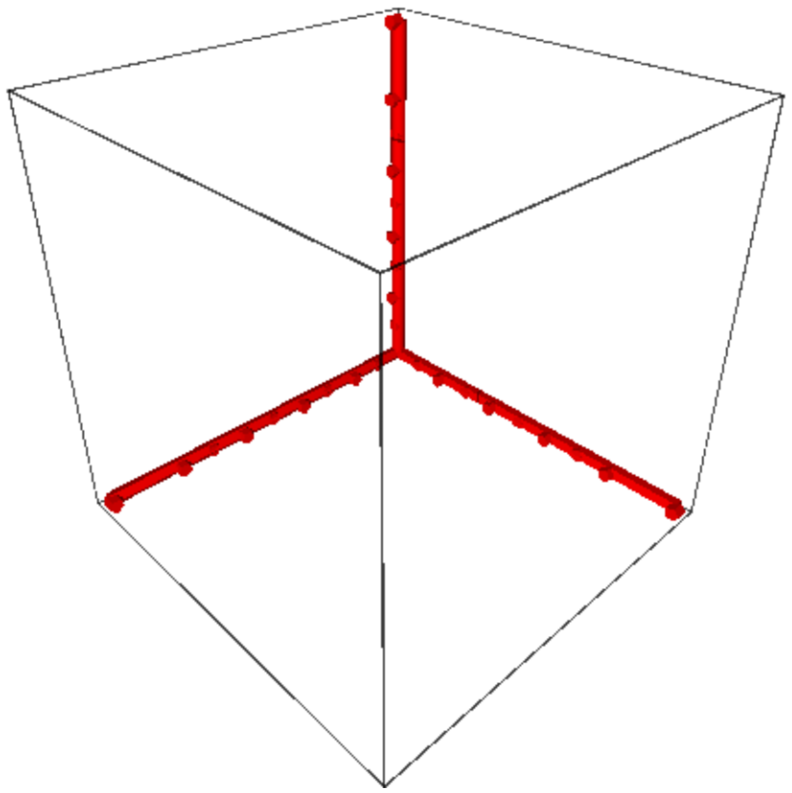
A ruler frame with some options:

```
sage: F = ruler_frame([1,2,3],vector([2,3,4]),ticks=6, sub_ticks=2, color='red'); F
Graphics3d Object
```

`sage.plot.plot3d.shapes2.`**`sphere`**(*center=(0, 0, 0)*, *size=1*, ***kwds*)

Return a plot of a sphere of radius `size` centered at (x, y, z).

INPUT:



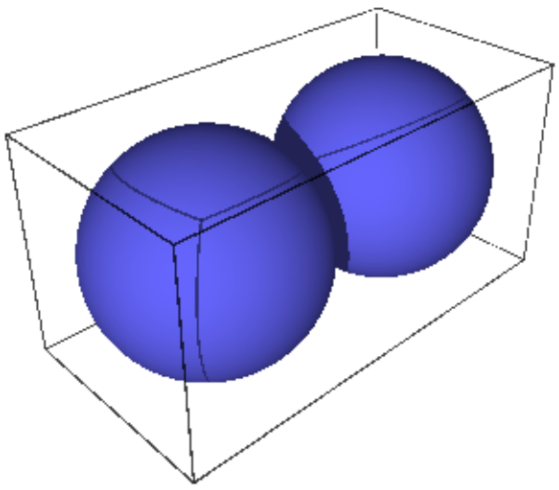
- (x, y, z) – center (default: (0,0,0))
- `size` – the radius (default: 1)

EXAMPLES: A simple sphere:

```
sage: sphere()
Graphics3d Object
```

Two spheres touching:

```
sage: sphere(center=(-1,0,0)) + sphere(center=(1,0,0), aspect_ratio=[1,1,1])
Graphics3d Object
```



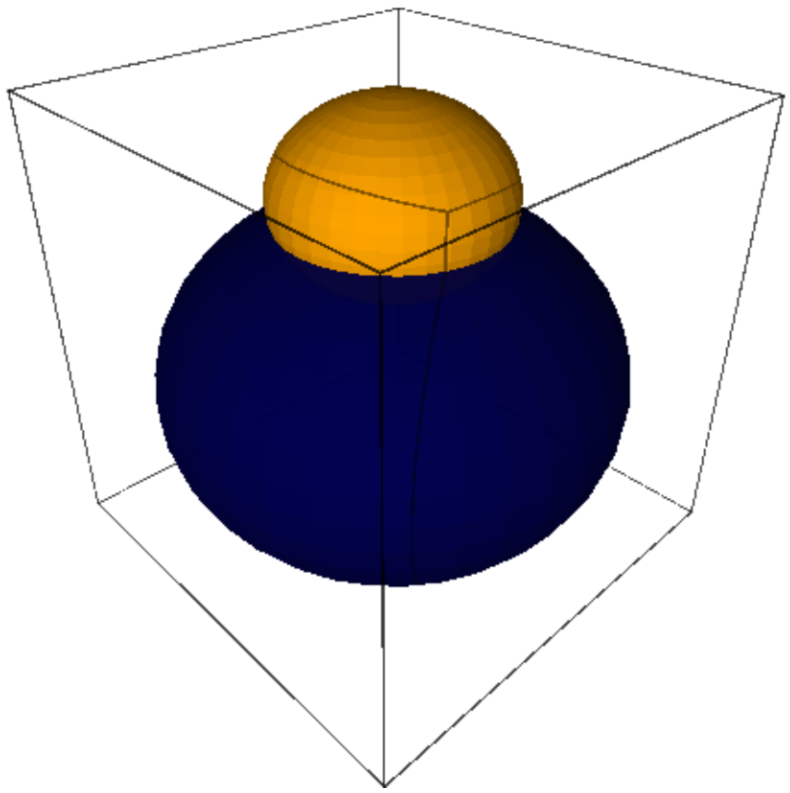
Spheres of radii 1 and 2 one stuck into the other:

```
sage: sphere(color='orange') + sphere(color=(0,0,0.3),
....:                               center=(0,0,-2), size=2, opacity=0.9)
Graphics3d Object
```

We draw a transparent sphere on a saddle.

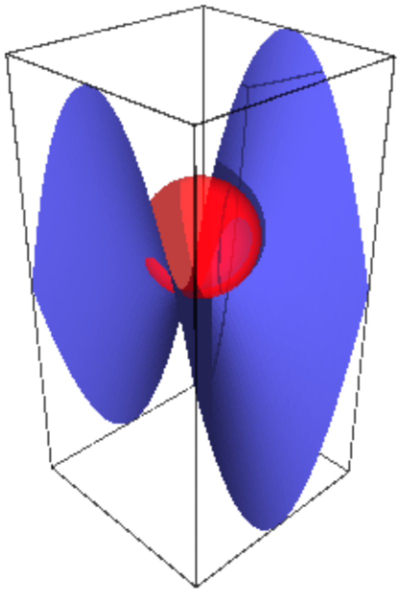
```
sage: u,v = var('u v')                                                       #␣
↪needs sage.symbolic
sage: saddle = plot3d(u^2 - v^2, (u,-2,2), (v,-2,2))                         #␣
```

(continues on next page)



(continued from previous page)

```
↪needs sage.symbolic
sage: sphere((0,0,1), color='red', opacity=0.5, aspect_ratio=[1,1,1]) + saddle   #␣
↪needs sage.symbolic
Graphics3d Object
```



sage.plot.plot3d.shapes2.**text3d**(*txt*, *x_y_z*, ***kwds*)

Display 3d text.

INPUT:

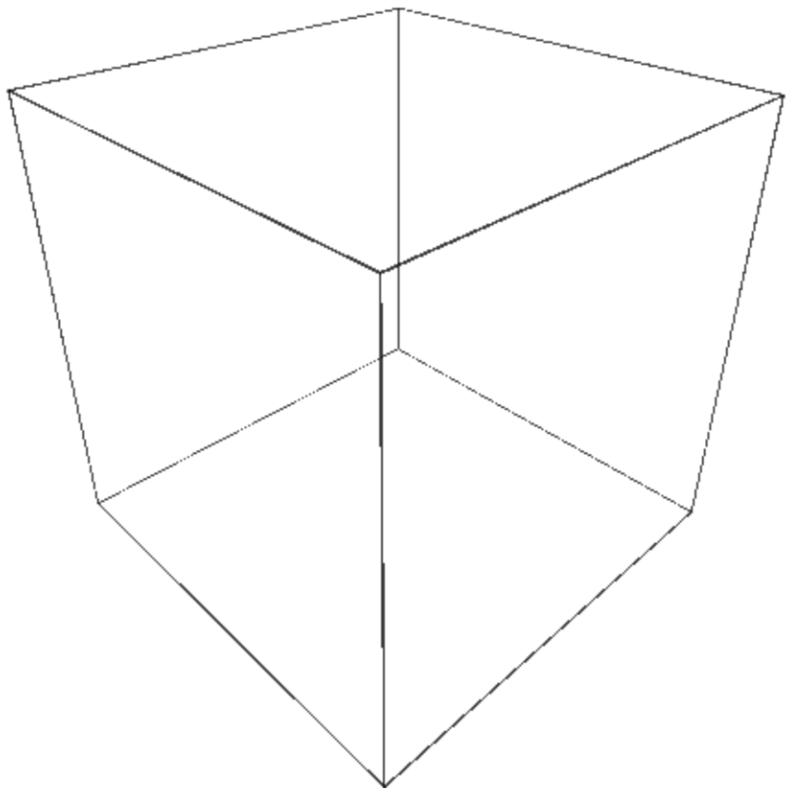
- `txt` – some text
- `(x,y,z)` – position tuple (x, y, z)
- `**kwds` – standard 3d graphics options

EXAMPLES:

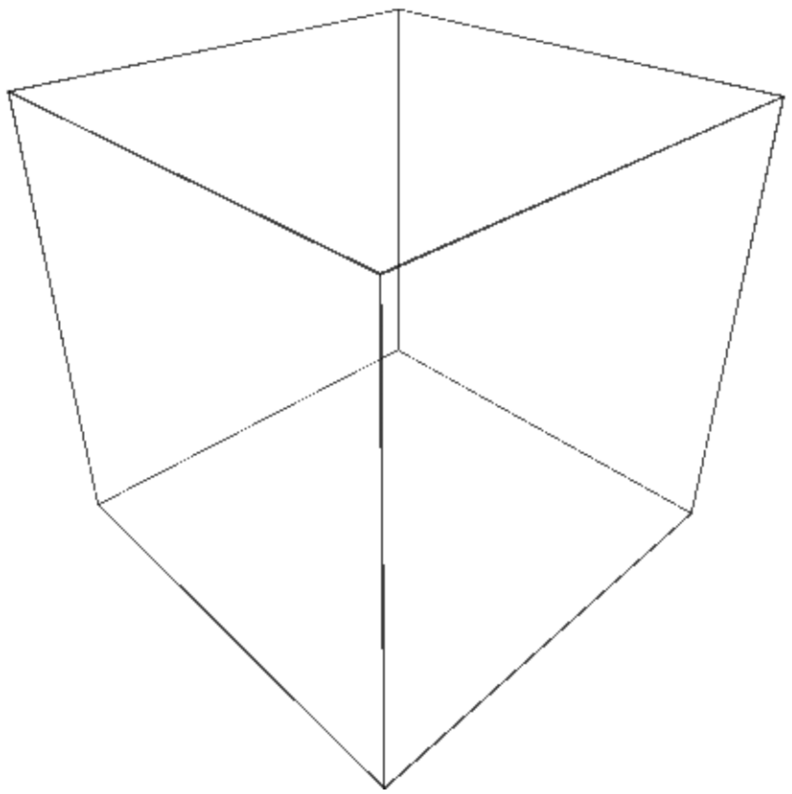
We write the word Sage in red at position (1,2,3):

```
sage: text3d("Sage", (1,2,3), color=(0.5,0,0))
Graphics3d Object
```

We draw a multicolor spiral of numbers:



```
sage: sum([text3d('%.1f'%n, (cos(n),sin(n),n), color=(n/2,1-n/2,0))
....:     for n in [0,0.2,..,8]])
Graphics3d Object
```



Another example:

```
sage: text3d("Sage is really neat!!",(2,12,1))
Graphics3d Object
```

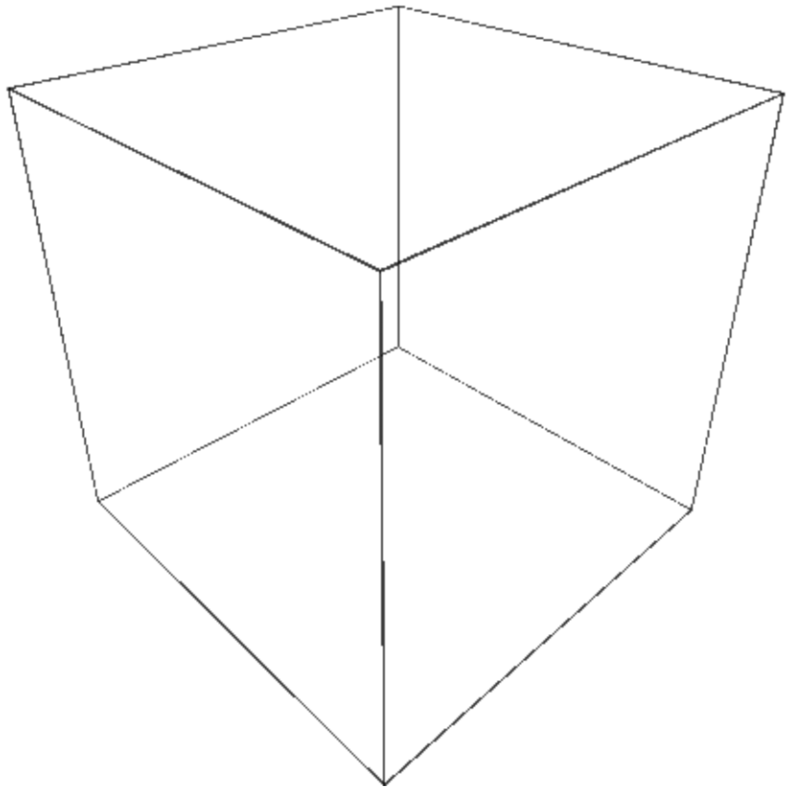
And in 3d in two places:

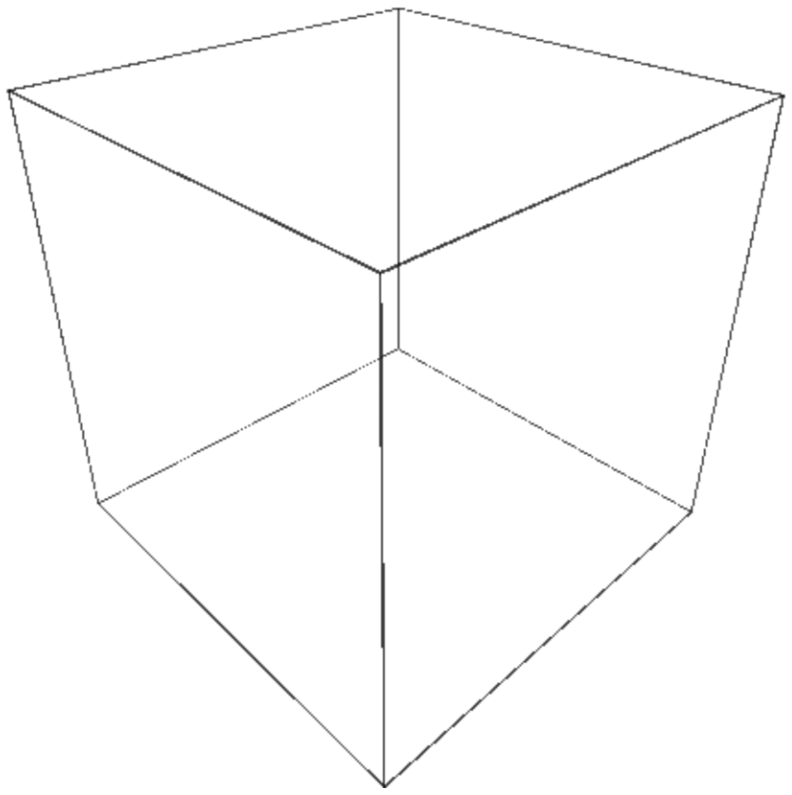
```
sage: text3d("Sage is...",(2,12,1), color=(1,0,0)) + text3d("quite powerful!!",(4,
↪10,0), color=(0,0,1))
Graphics3d Object
```

Adjust the font size, family, style, and weight (Three.js viewer only):

```
sage: t0 = text3d("Pixel size", (0, 0, 0), fontsize=20)
sage: t1 = text3d("Percentage size", (0, 0, 1), fontsize='300%')
sage: t2 = text3d("Keyword size", (0, 0, 2), fontsize='x-small')
sage: t3 = text3d("Single family", (0, 0, 3), fontfamily='serif')
sage: t4 = text3d("Family fallback", (0, 0, 4), fontfamily=['Consolas', 'Lucida␣
↪Console', 'monospace'])
```

(continues on next page)





(continued from previous page)

```
sage: t5 = text3d("Another way", (0, 0, 5), fontfamily='Consolas, Lucida Console,␣
↪monospace')
sage: t6 = text3d("Style", (0, 0, 6), fontstyle='italic')
sage: t7 = text3d("Keyword weight", (0, 0, 7), fontweight='bold')
sage: t8 = text3d("Integer weight (1-1000)", (0, 0, 8), fontweight=800) # 'extra bold
↪'
sage: sum([t0, t1, t2, t3, t4, t5, t6, t7, t8]).show(viewer='threejs', frame=False)
```

Adjust the text's opacity (Three.js viewer only):

```
sage: def echo(o):
....:     return text3d("Echo!", (0, 0, o), opacity=o)
sage: show(sum([echo(o) for o in (0.1, 0.2, .., 1)]), viewer='threejs')
```

3.4 Platonic solids

EXAMPLES: The five platonic solids in a row:

```
sage: G = tetrahedron((0,-3.5,0), color='blue') + cube((0,-2,0),color=(.25,0,.5))
sage: G += octahedron(color='red') + dodecahedron((0,2,0), color='orange')
sage: G += icosahedron(center=(0,4,0), color='yellow')
sage: G.show(aspect_ratio=[1,1,1])
```

All the platonic solids in the same place:

```
sage: G = tetrahedron(color='blue',opacity=0.7)
sage: G += cube(color=(.25,0,.5), opacity=0.7)
sage: G += octahedron(color='red', opacity=0.7)
sage: G += dodecahedron(color='orange', opacity=0.7) + icosahedron(opacity=0.7)
sage: G.show(aspect_ratio=[1,1,1])
```

Display nice faces only:

```
sage: icosahedron().stickers(['red','blue'], .075, .1)
Graphics3d Object
```

AUTHORS:

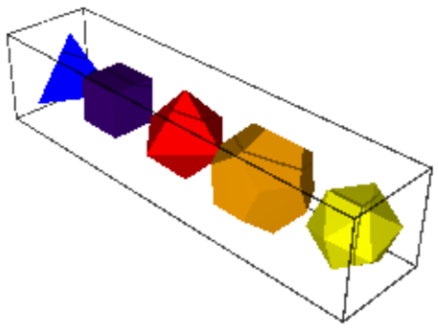
- Robert Bradshaw (2007, 2008): initial version
- William Stein

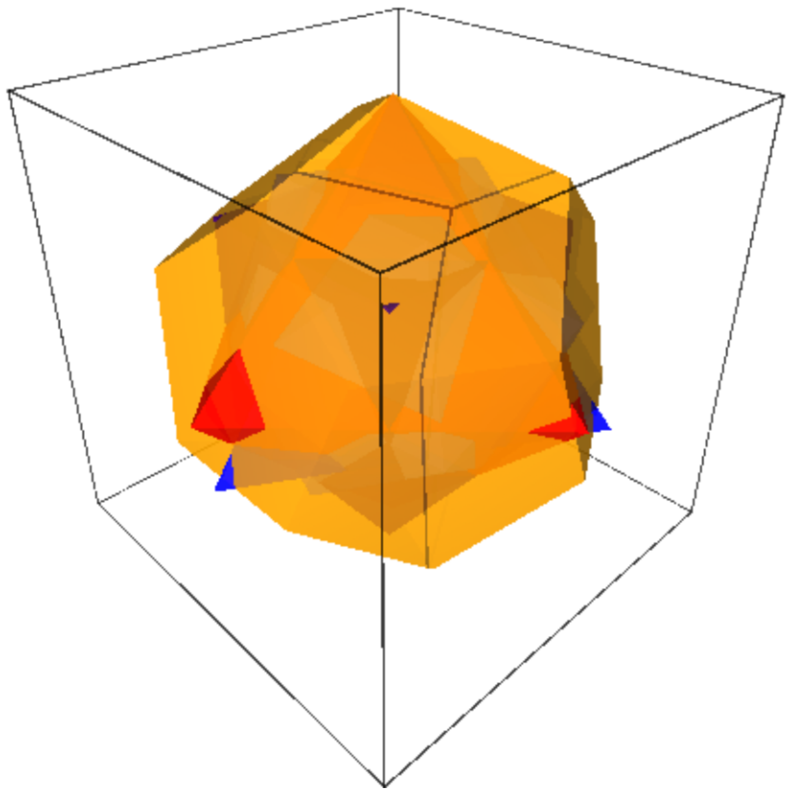
`sage.plot.plot3d.platonic.`**`cube`**(*center=(0, 0, 0)*, *size=1*, *color=None*, *frame_thickness=0*, *frame_color=None*, ***kwds*)

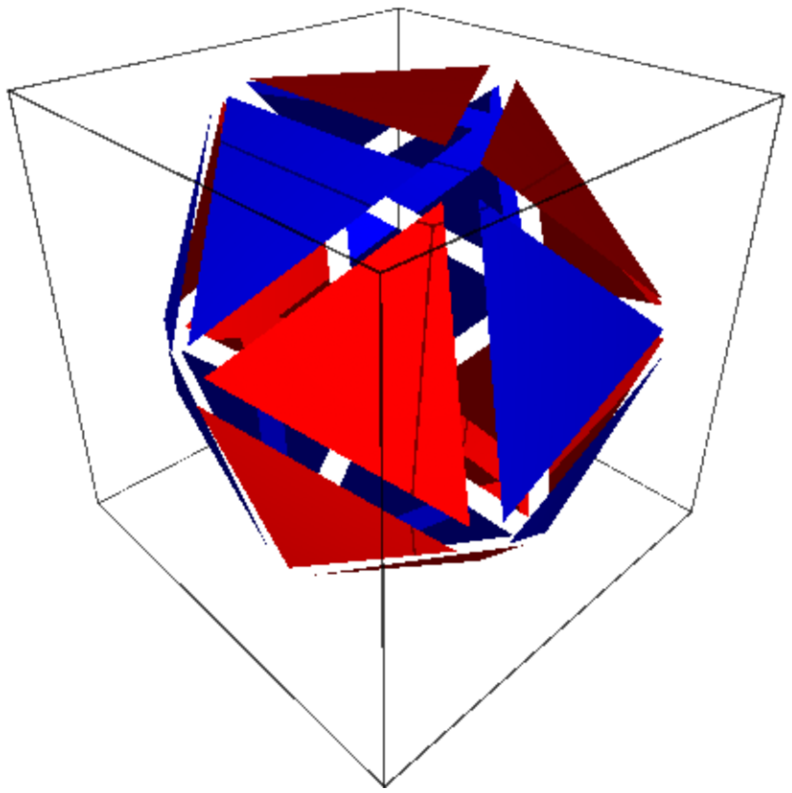
A 3D cube centered at the origin with default side lengths 1.

INPUT:

- `center` – (default: (0,0,0))
- `size` – (default: 1) the side lengths of the cube
- `color` – a string that describes a color; this can also be a list of 3-tuples or strings length 6 or 3, in which case the faces (and oppositive faces) are colored.





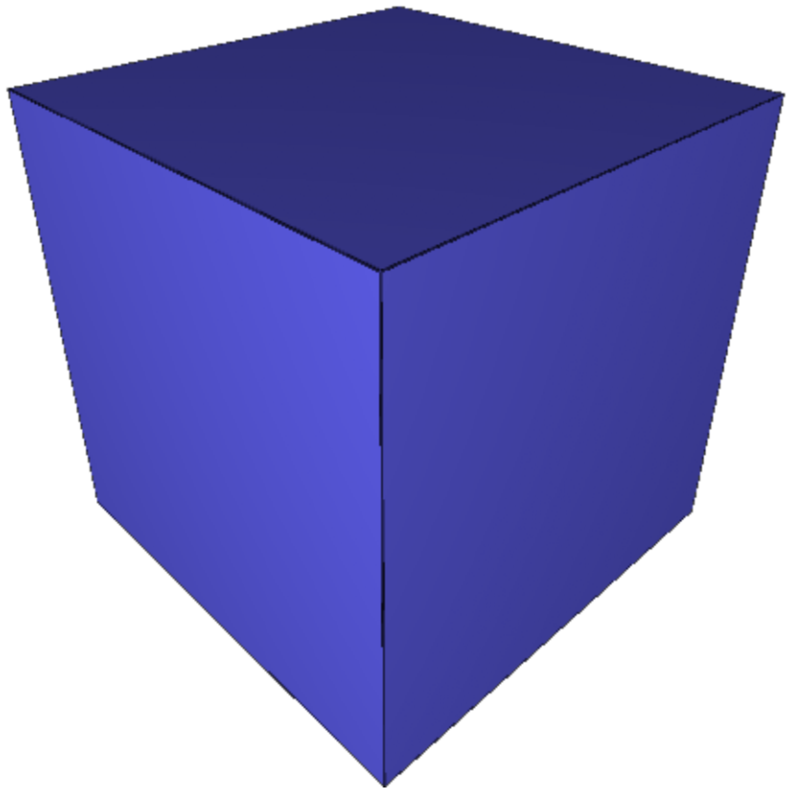


- `frame_thickness` – (default: 0) if positive, then thickness of the frame
- `frame_color` – (default: None) if given, gives the color of the frame
- `opacity` – (default: 1) if less than 1 then it's transparent

EXAMPLES:

A simple cube:

```
sage: cube()
Graphics3d Object
```



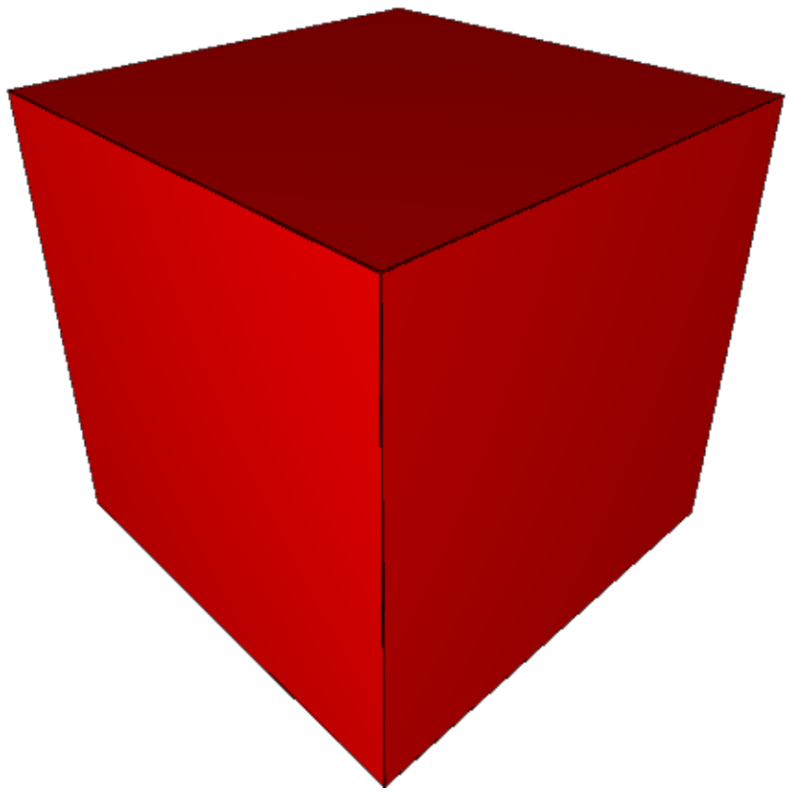
A red cube:

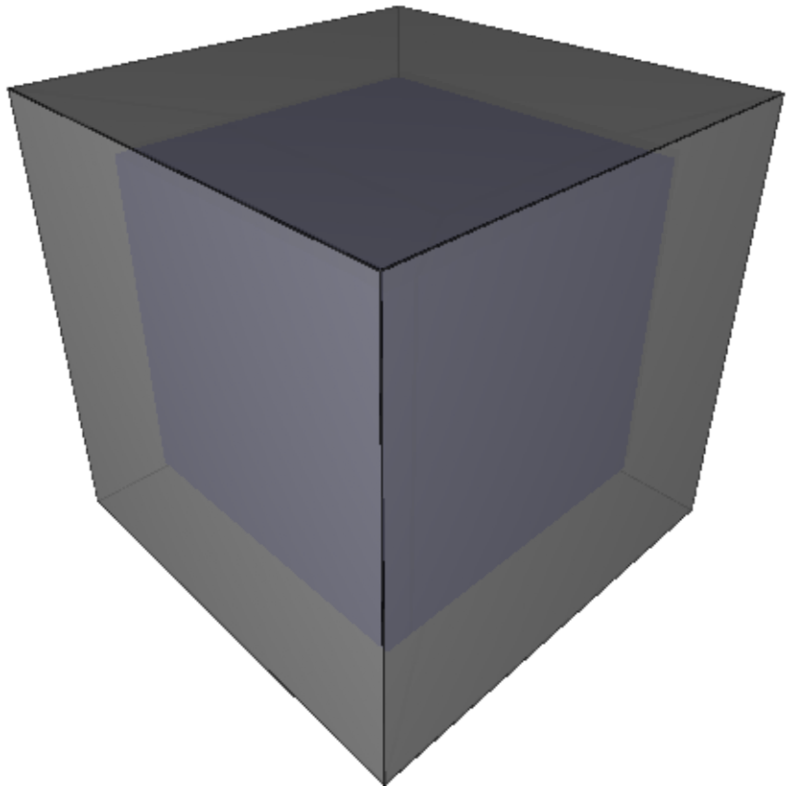
```
sage: cube(color="red")
Graphics3d Object
```

A transparent grey cube that contains a red cube:

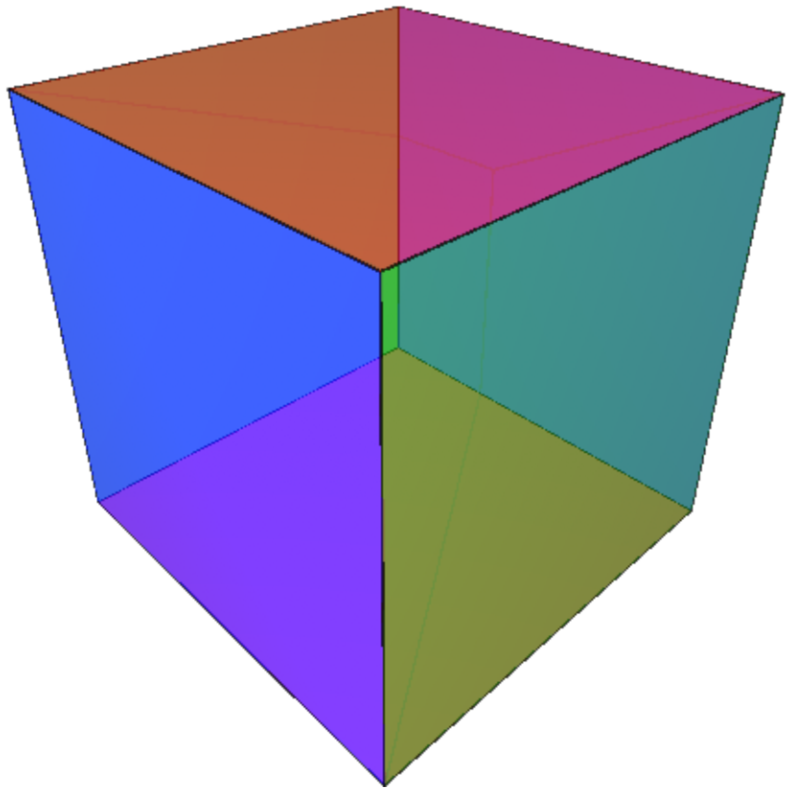
```
sage: cube(opacity=0.8, color='grey') + cube(size=3/4)
Graphics3d Object
```

A transparent colored cube:





```
sage: cube(color=['red', 'green', 'blue'], opacity=0.5)
Graphics3d Object
```



A bunch of random cubes:

```
sage: v = [(random(), random(), random()) for _ in [1..30]]
sage: sum([cube((10*a,10*b,10*c), size=random()/3, color=(a,b,c)) for a,b,c in v])
Graphics3d Object
```

Non-square cubes (boxes):

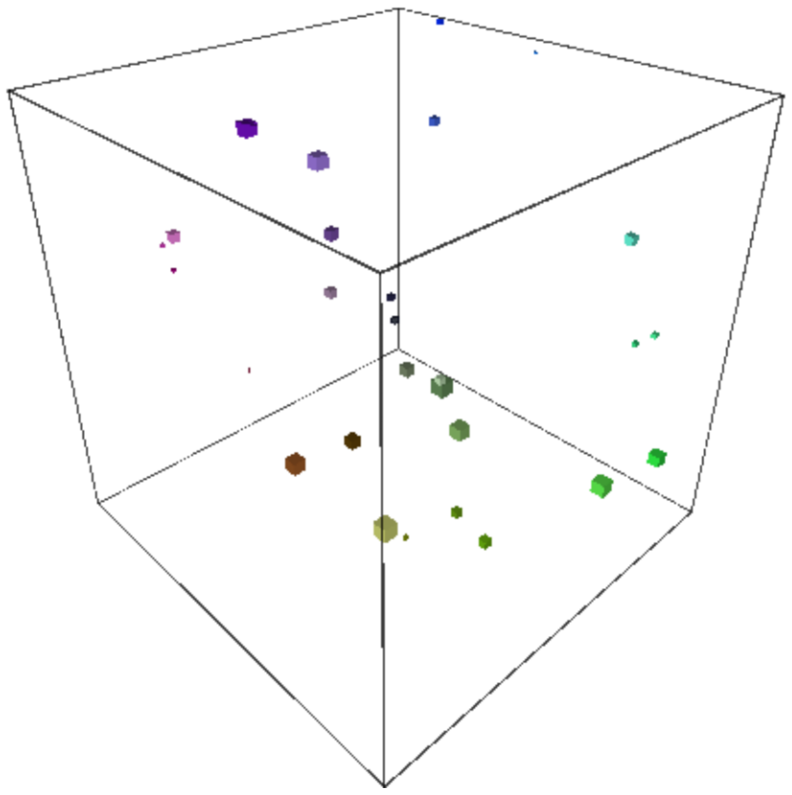
```
sage: cube(aspect_ratio=[1,1,1]).scale([1,2,3])
Graphics3d Object
```

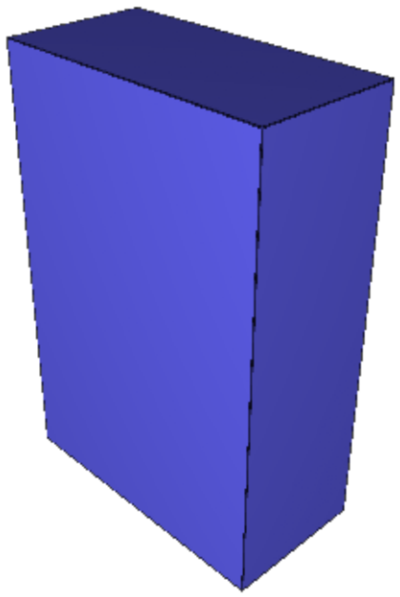
```
sage: cube(color=['red', 'blue', 'green'],aspect_ratio=[1,1,1]).scale([1,2,3])
Graphics3d Object
```

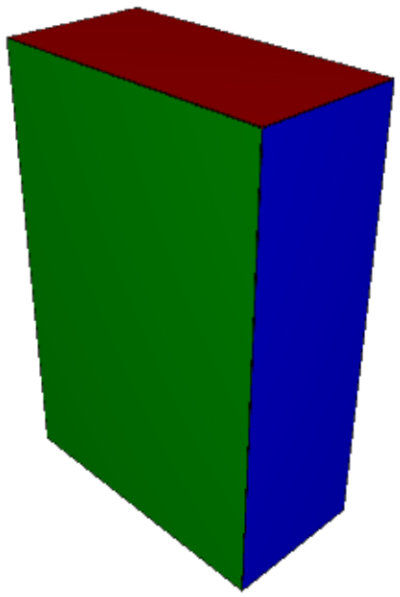
And one that is colored:

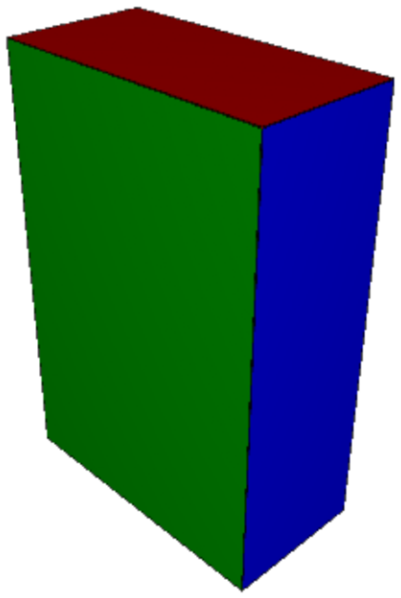
```
sage: cube(color=['red', 'blue', 'green', 'black', 'white', 'orange'],
....:      aspect_ratio=[1,1,1]).scale([1,2,3])
Graphics3d Object
```

A nice translucent color cube with a frame:

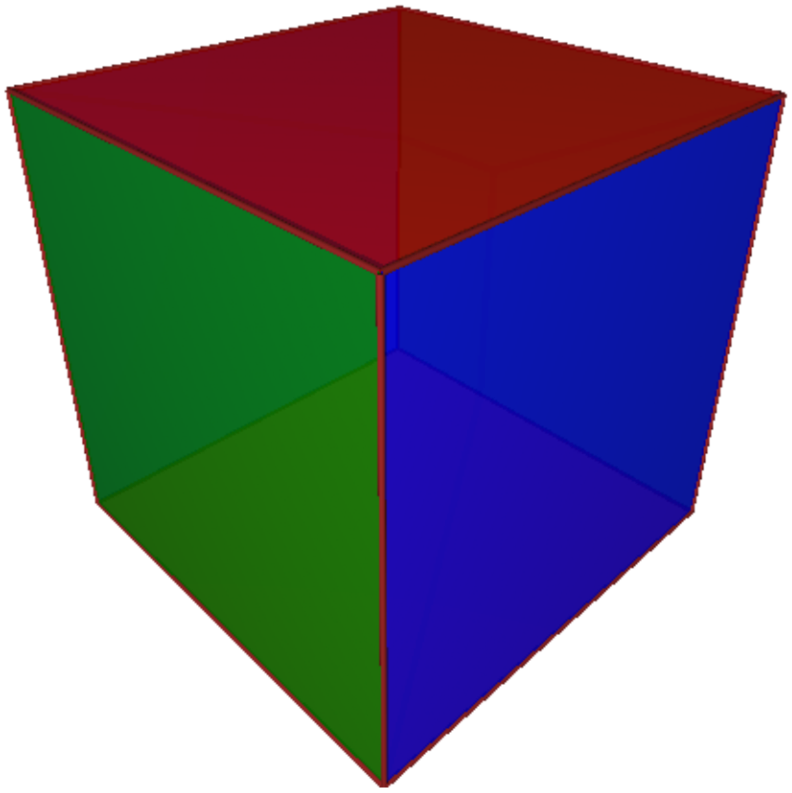








```
sage: c = cube(color=['red', 'blue', 'green'], frame=False, frame_thickness=2,
....:          frame_color='brown', opacity=0.8)
sage: c
Graphics3d Object
```



A raytraced color cube with frame and transparency:

```
sage: c.show(viewer='tachyon')
```

This shows github issue #11272 has been fixed:

```
sage: cube(center=(10, 10, 10), size=0.5).bounding_box()
((9.75, 9.75, 9.75), (10.25, 10.25, 10.25))
```

AUTHORS:

- William Stein

sage.plot.plot3d.platonic.**dodecahedron**(*center=(0, 0, 0)*, *size=1*, ***kwds*)

A dodecahedron.

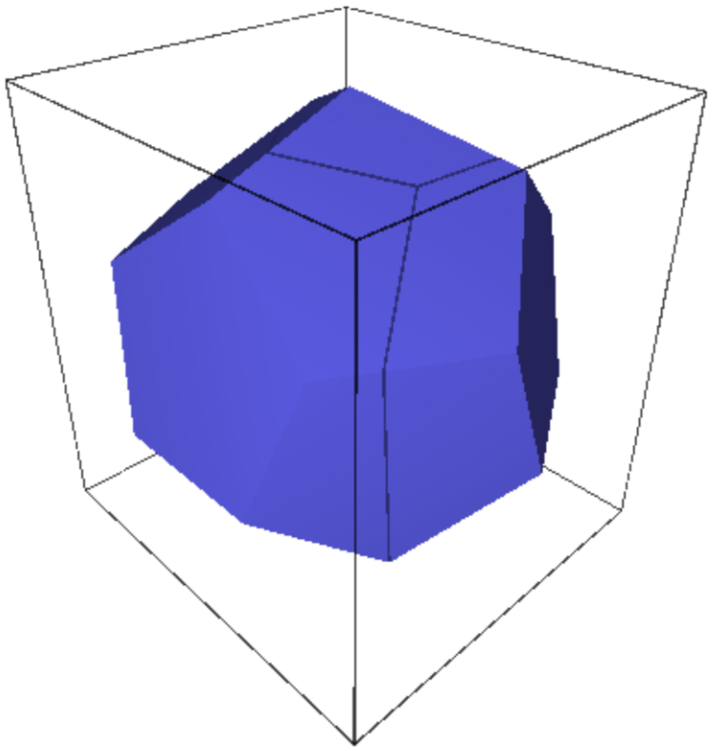
INPUT:

- `center` – (default: (0,0,0))
- `size` – (default: 1)

- `color` – a string that describes a color; this can also be a list of 3-tuples or strings length 6 or 3, in which case the faces (and oppositive faces) are colored.
- `opacity` – (default: 1) if less than 1 then is transparent

EXAMPLES: A plain Dodecahedron:

```
sage: dodecahedron()
Graphics3d Object
```



A translucent dodecahedron that contains a black sphere:

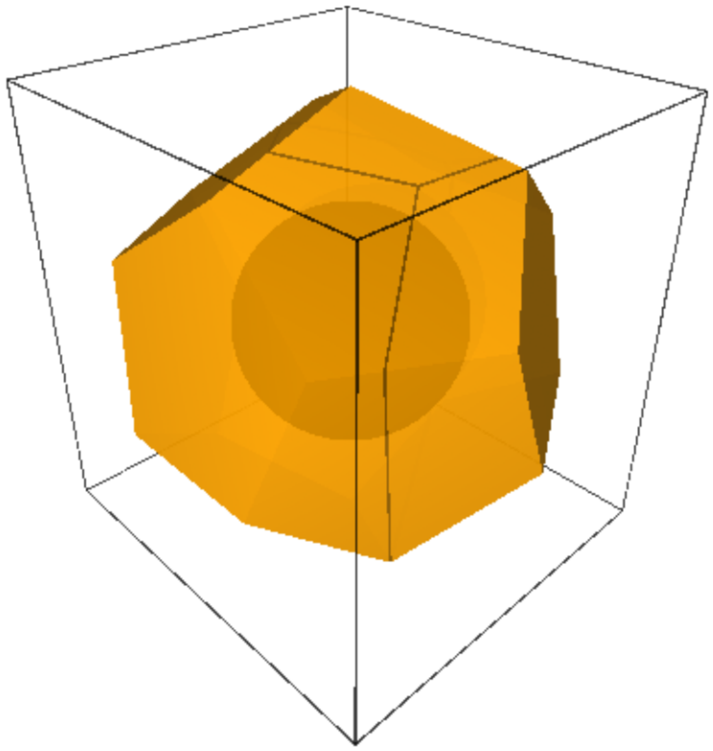
```
sage: G = dodecahedron(color='orange', opacity=0.8)
sage: G += sphere(size=0.5, color='black')
sage: G
Graphics3d Object
```

CONSTRUCTION: This is how we construct a dodecahedron. We let one point be $Q = (0, 1, 0)$.

Now there are three points spaced equally on a circle around the north pole. The other requirement is that the angle between them be the angle of a pentagon, namely $3\pi/5$. This is enough to determine them. Placing one on the xz-plane we have.

$P_1 = \left(t, 0, \sqrt{1-t^2}\right)$

$P_2 = \left(-\frac{1}{2}t, \frac{\sqrt{3}}{2}t, \sqrt{1-t^2}\right)$



$P_3 = \left(-\frac{1}{2}t, \frac{\sqrt{3}}{2}t, \sqrt{1-t^2}\right)$

Solving $\frac{(P_1-Q)\cdot(P_2-Q)}{|P_1-Q||P_2-Q|} = \cos(3\pi/5)$ we get $t = 2/3$.

Now we have 6 points $R_1, ..., R_6$ to close the three top pentagons. These can be found by mirroring P_2 and P_3 by the yz-plane and rotating around the y-axis by the angle θ from Q to P_1. Note that $\cos(\theta) = t = 2/3$ and so $\sin(\theta) = \sqrt{5}/3$. Rotation gives us the other four.

Now we reflect through the origin for the bottom half.

AUTHORS:

- Robert Bradshaw, William Stein

`sage.plot.plot3d.platonic.`**`icosahedron`**`(`*center=(0, 0, 0)*, *size=1*, ***kwds*`)`

An icosahedron.

INPUT:

- `center` – (default: (0, 0, 0))
- `size` – (default: 1)
- `color` – a string that describes a color; this can also be a list of 3-tuples or strings length 6 or 3, in which case the faces (and oppositive faces) are colored.
- `opacity` – (default: 1) if less than 1 then is transparent

EXAMPLES:

```
sage: icosahedron()
Graphics3d Object
```

Two icosahedra at different positions of different sizes.

```
sage: p = icosahedron((-1/2,0,1), color='orange')
sage: p += icosahedron((2,0,1), size=1/2, color='red', aspect_ratio=[1,1,1])
sage: p
Graphics3d Object
```

`sage.plot.plot3d.platonic.`**`index_face_set`**`(`*face_list*, *point_list*, *enclosed*, ***kwds*`)`

Helper function that creates `IndexFaceSet` object for the tetrahedron, dodecahedron, and icosahedron.

INPUT:

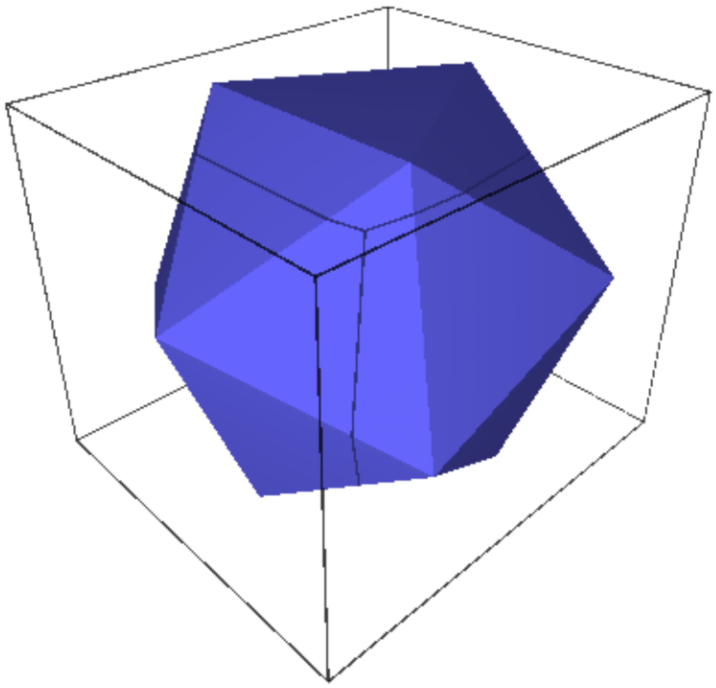
- `face_list` – list of faces, given explicitly from the solid invocation
- `point_list` – list of points, given explicitly from the solid invocation
- `enclosed` – boolean (default passed is always `True` for these solids)

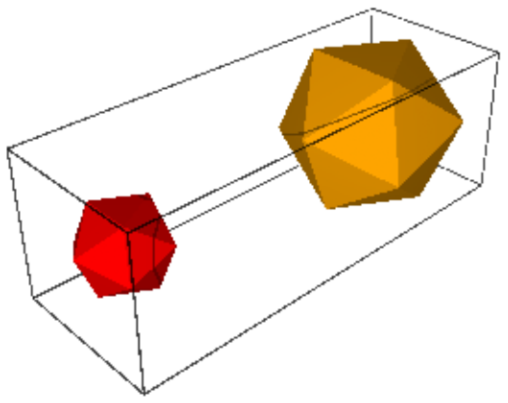
`sage.plot.plot3d.platonic.`**`octahedron`**`(`*center=(0, 0, 0)*, *size=1*, ***kwds*`)`

Return an octahedron.

INPUT:

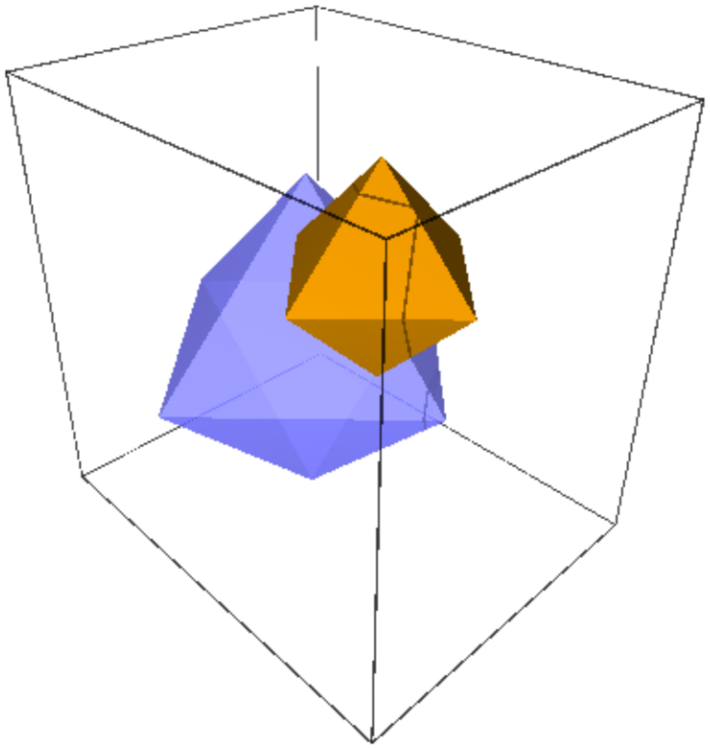
- `center` – (default: (0,0,0))
- `size` – (default: 1)
- `color` – a string that describes a color; this can also be a list of 3-tuples or strings length 6 or 3, in which case the faces (and oppositive faces) are colored.
- `opacity` – (default: 1) if less than 1 then is transparent





EXAMPLES:

```
sage: G = octahedron((1,4,3), color='orange')
sage: G += octahedron((0,2,1), size=2, opacity=0.6)
sage: G
Graphics3d Object
```



sage.plot.plot3d.platonic.**prep**(*G*, *center*, *size*, *kwds*)

Helper function that scales and translates the platonic solid, and passes extra keywords on.

INPUT:

- `center` – 3-tuple indicating the center (default passed from *index_face_set()* is the origin $(0, 0, 0)$)
- `size` – number indicating amount to scale by (default passed from *index_face_set()* is 1)
- `kwds` – a dictionary of keywords, passed from solid invocation by *index_face_set()*

sage.plot.plot3d.platonic.**tetrahedron**(*center=(0, 0, 0)*, *size=1*, ***kwds*)

A 3d tetrahedron.

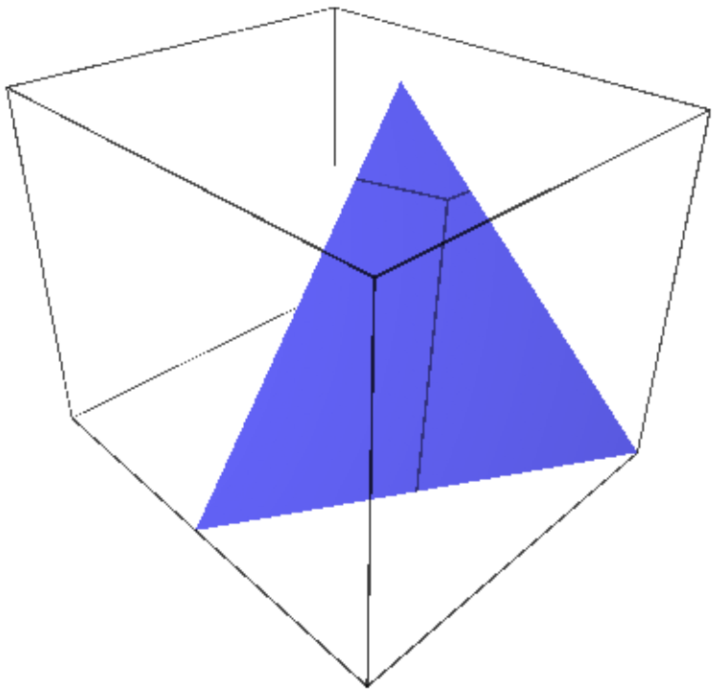
INPUT:

- `center` – (default: (0,0,0))
- `size` – (default: 1)
- `color` – a string (`"red"`, `"green"`, etc) or a tuple (r, g, b) with r, g, b numbers between 0 and 1

- `opacity` – (default: 1) if less than 1 then is transparent

EXAMPLES: A default colored tetrahedron at the origin:

```
sage: tetrahedron()
Graphics3d Object
```



A transparent green tetrahedron in front of a solid red one:

```
sage: tetrahedron(opacity=0.8, color='green') + tetrahedron((-2,1,0),color='red')
Graphics3d Object
```

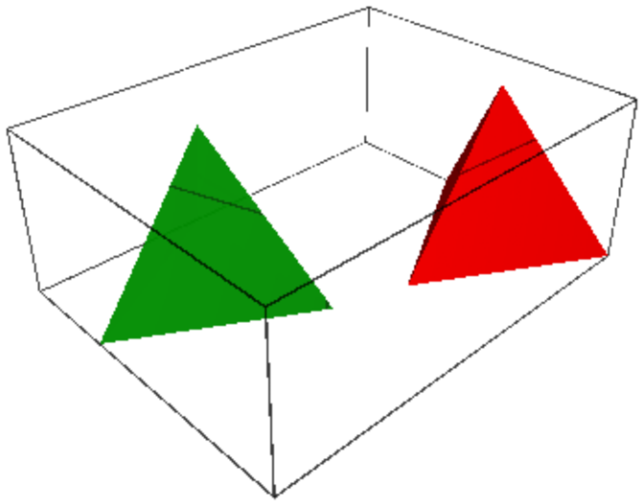
A translucent tetrahedron sharing space with a sphere:

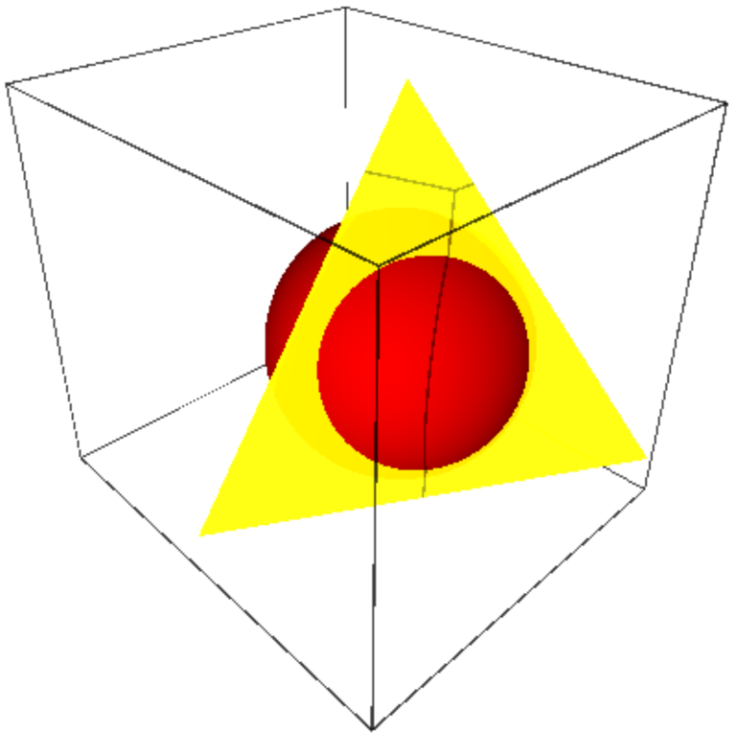
```
sage: tetrahedron(color='yellow',opacity=0.7) + sphere(size=.5, color='red')
Graphics3d Object
```

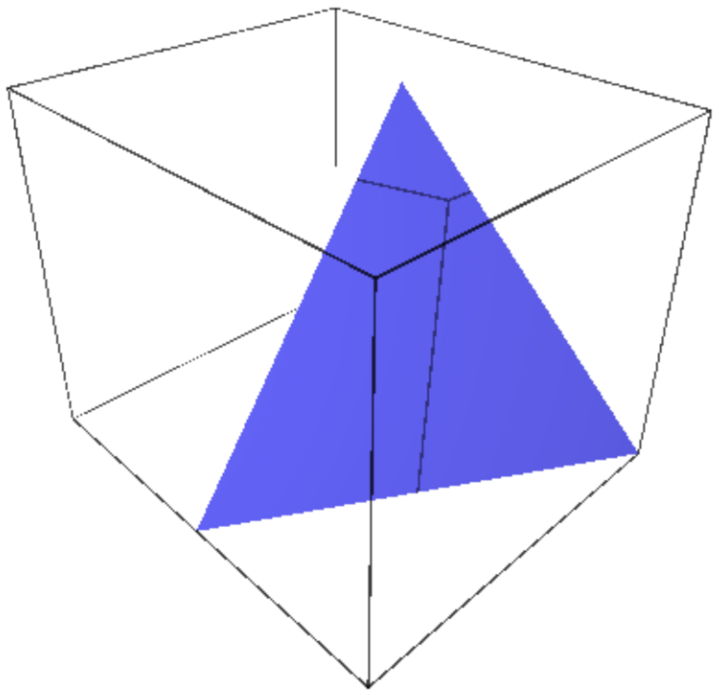
A big tetrahedron:

```
sage: tetrahedron(size=10)
Graphics3d Object
```

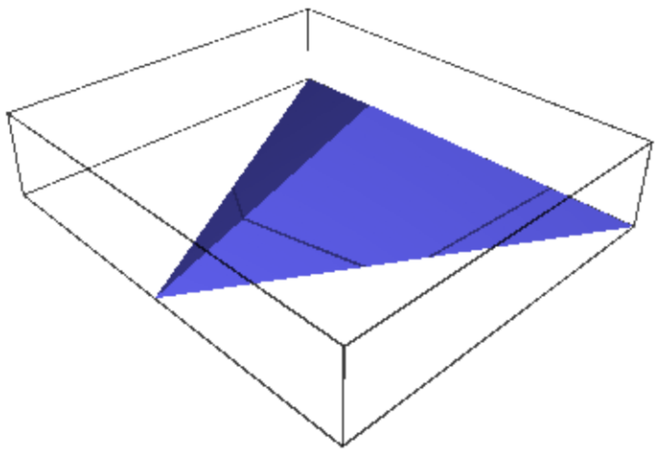
A wide tetrahedron:







```
sage: tetrahedron(aspect_ratio=[1,1,1]).scale((4,4,1))
Graphics3d Object
```



A red and blue tetrahedron touching noses:

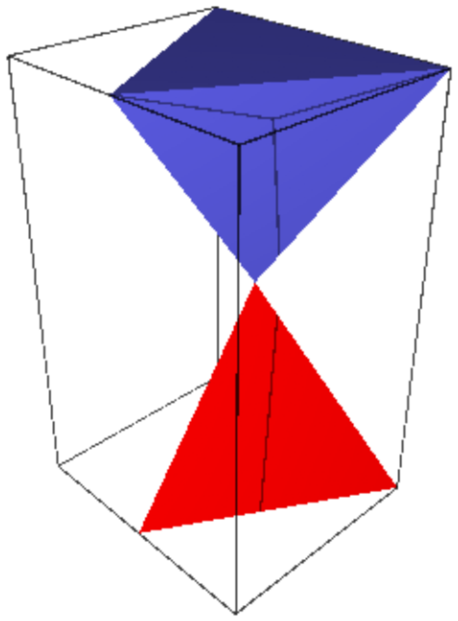
```
sage: tetrahedron(color='red') + tetrahedron((0,0,-2)).scale([1,1,-1])
Graphics3d Object
```

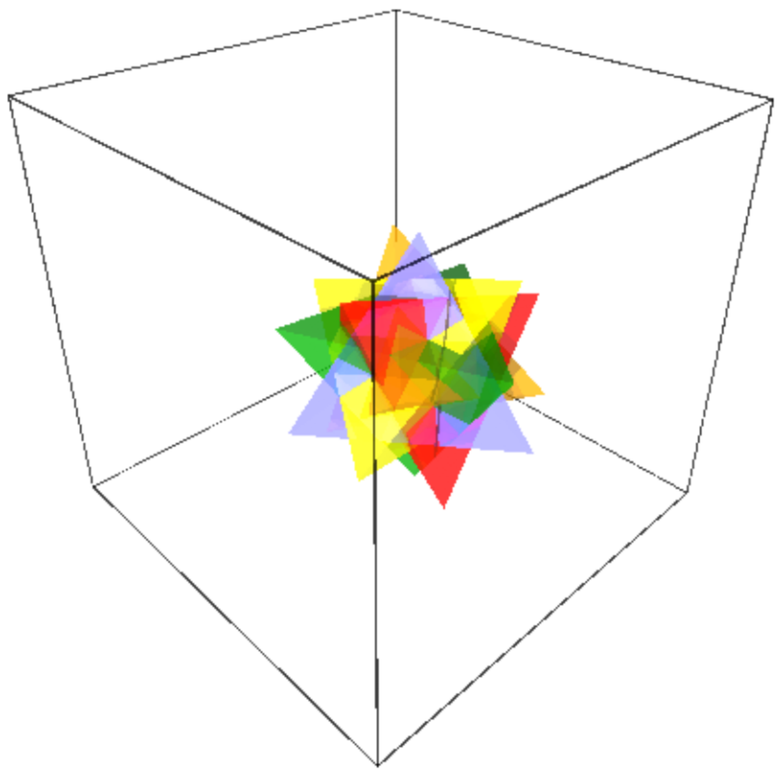
A Dodecahedral complex of 5 tetrahedra (a more elaborate example from Peter Jipsen):

```
sage: from math import pi
sage: v = (sqrt(5.)/2-5/6, 5/6*sqrt(3.)-sqrt(15.)/2, sqrt(5.)/3)
sage: t = acos(sqrt(5.)/3)/2
sage: t1 = tetrahedron(aspect_ratio=(1,1,1), opacity=0.5).rotateZ(t)
sage: t2 = tetrahedron(color='red', opacity=0.5).rotateZ(t).rotate(v,2*pi/5)
sage: t3 = tetrahedron(color='green', opacity=0.5).rotateZ(t).rotate(v,4*pi/5)
sage: t4 = tetrahedron(color='yellow', opacity=0.5).rotateZ(t).rotate(v,6*pi/5)
sage: t5 = tetrahedron(color='orange', opacity=0.5).rotateZ(t).rotate(v,8*pi/5)
sage: show(t1+t2+t3+t4+t5, frame=False, zoom=1.3)
```

AUTHORS:

- Robert Bradshaw and William Stein





3.5 Parametric surface

Graphics 3D object for triangulating surfaces, and a base class for many other objects that can be represented by a 2D parametrization.

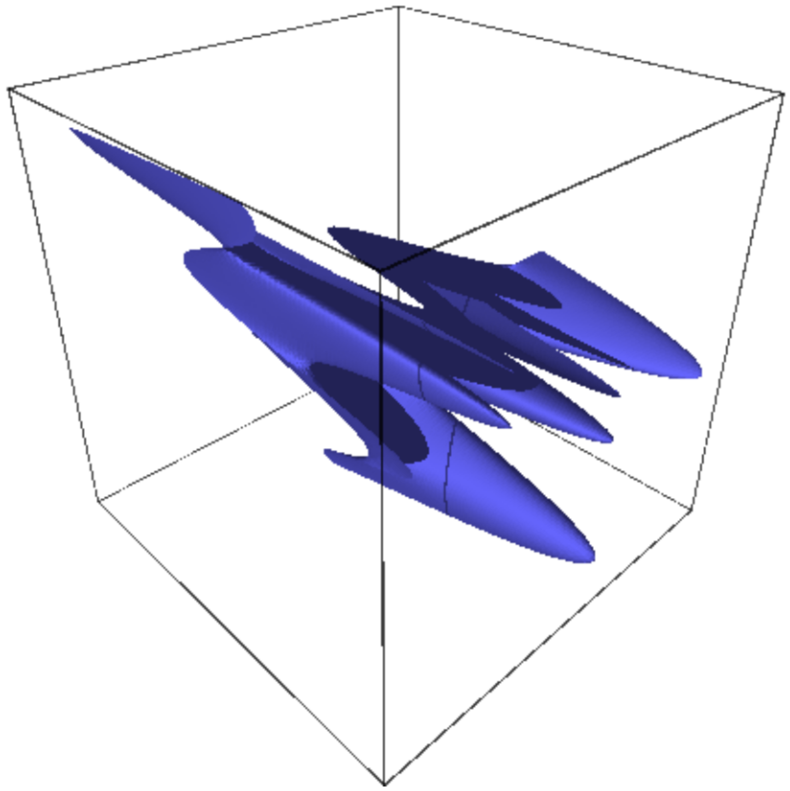
It takes great care to turn degenerate quadrilaterals into triangles and to propagate identified points to all attached polygons. This is not so much to save space as it is to assist the raytracers/other rendering systems to better understand the surface (and especially calculate correct surface normals).

AUTHORS:

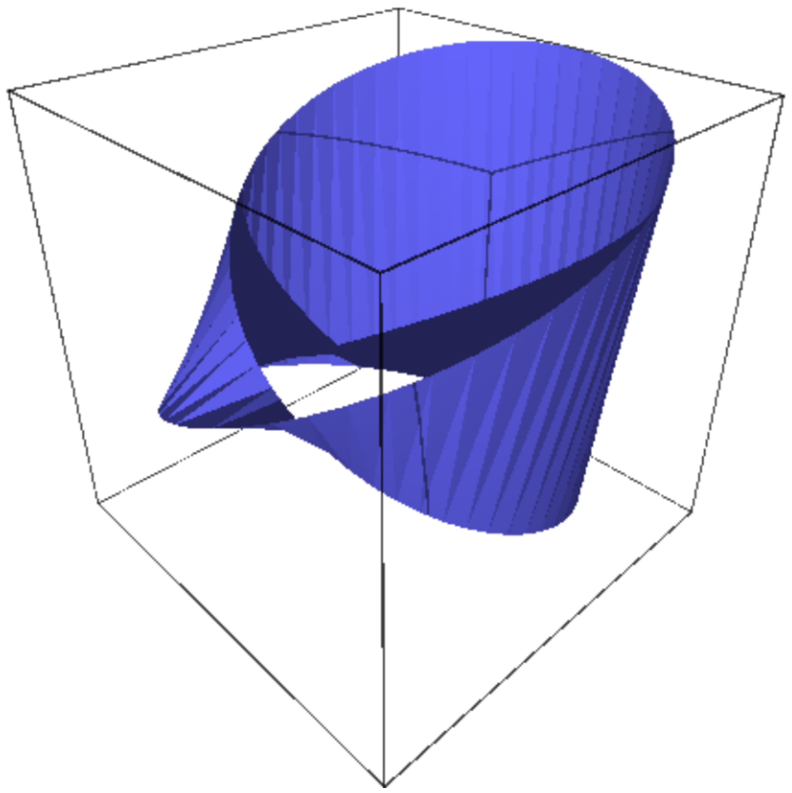
- Robert Bradshaw (2007-08-26): initial version

EXAMPLES:

```
sage: from sage.plot.plot3d.parametric_surface import ParametricSurface, MoebiusStrip
sage: def f(x,y): return x+y, sin(x)*sin(y), x*y
sage: P = ParametricSurface(f, (srange(0,10,0.1), srange(-5,5.0,0.1)))
sage: show(P)
```

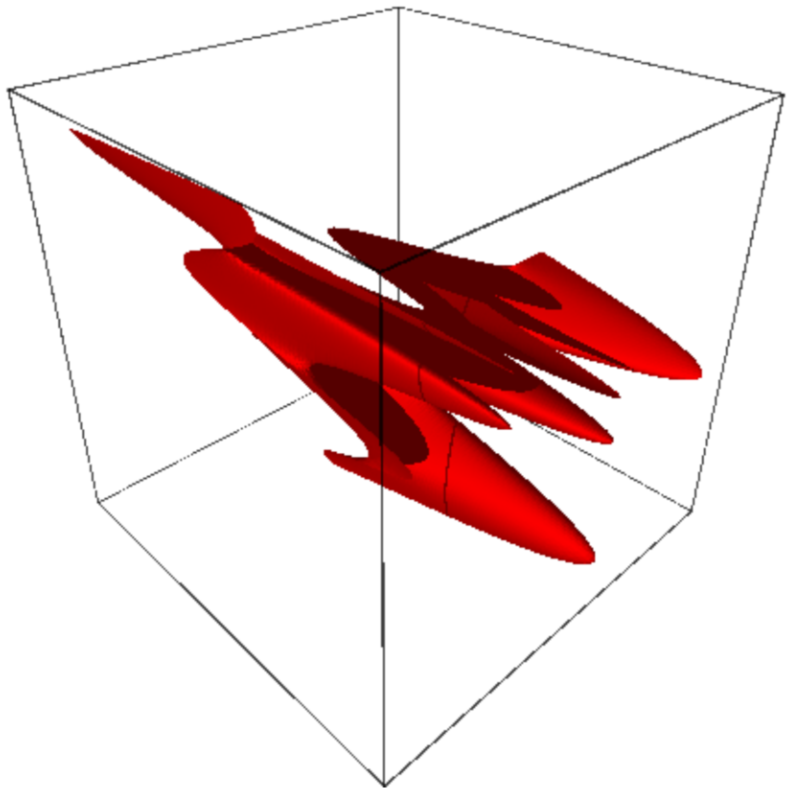


```
sage: S = MoebiusStrip(1,.2)
sage: S.is_enclosed()
False
sage: show(S)
```



By default, the surface is colored with one single color.

```
sage: P = ParametricSurface(f, (srange(0,10,0.1), srange(-5,5.0,0.1)),
....:  color="red")
sage: P.show()
```



One can instead provide a coloring function and a colormap:

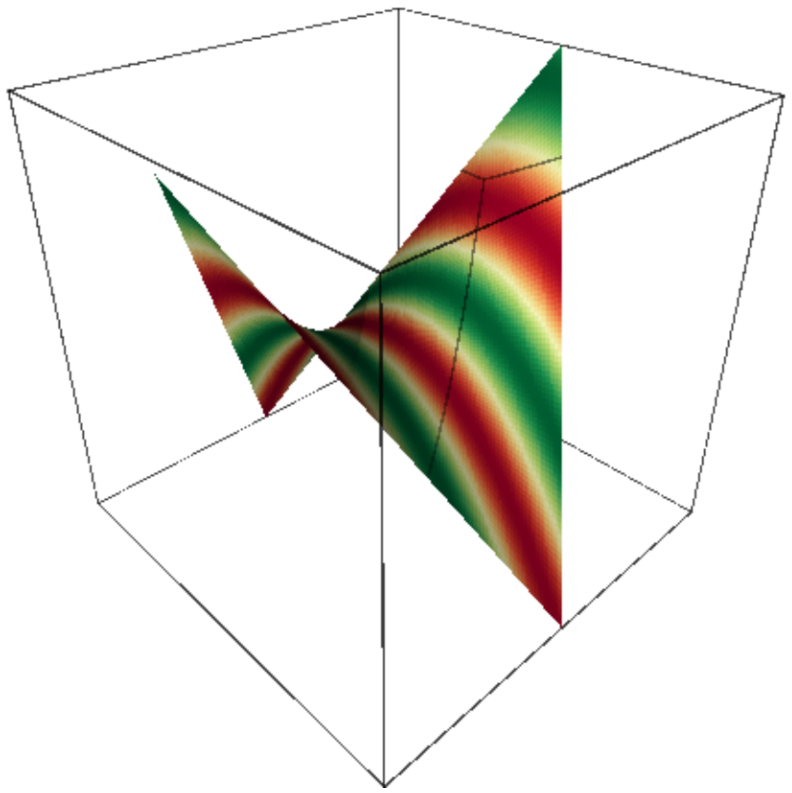
```
sage: def f(x,y): return x+y, x-y, x*y
sage: def c(x,y): return sin((x+y)/2)**2
sage: cm = colormaps.RdYlGn
sage: P = ParametricSurface(f, (srange(-5,5,0.1), srange(-5,5.0,0.1)), color=(c,cm))
sage: P.show(viewer='tachyon')
```

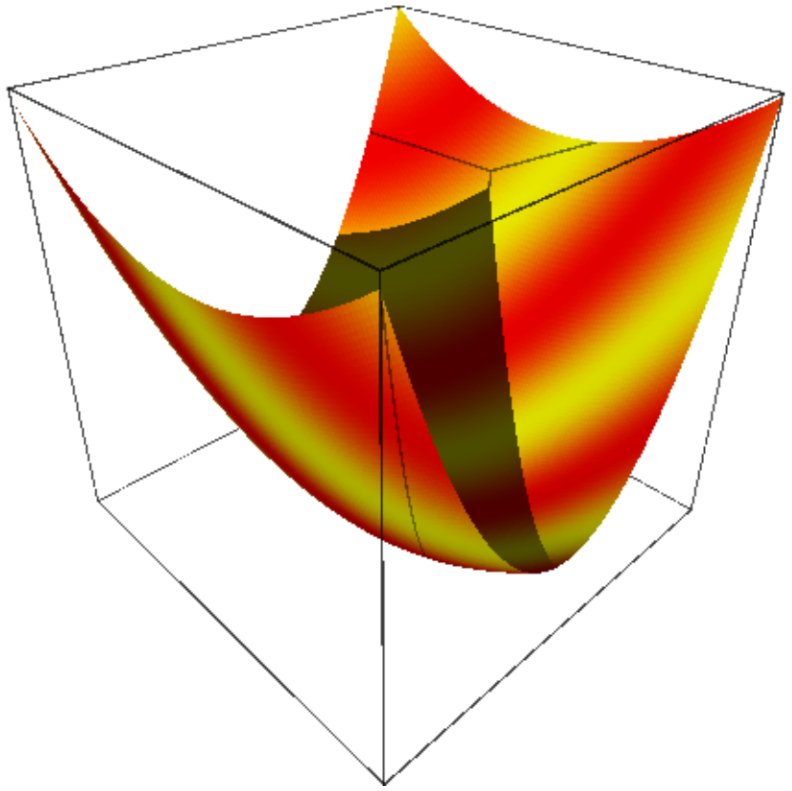
Note that the coloring function should rather have values between 0 and 1. This value is passed to the chosen colormap.

Another colored example:

```
sage: colm = colormaps.autumn
sage: def g(x,y): return x, y, x**2 + y**2
sage: P = ParametricSurface(g, (srange(-10,10,0.1), srange(-5,5.0,0.1)), color=(c,colm))
sage: P.show(viewer='tachyon')
```

Note: One may override `eval()` or `eval_c()` in a subclass rather than passing in a function for greater speed. One





also would want to override get_grid.

Todo: actually remove unused points, fix the below code:

```
S = ParametricSurface(f=lambda xy: (xy[0],xy[1],0), domain=(range(10),range(10)))
```

class `sage.plot.plot3d.parametric_surface.`**`MoebiusStrip`**(*r*, *width*, *twists=1*, ***kwds*)

Bases: *`ParametricSurface`*

Base class for the *`MoebiusStrip`* graphics type. This sets the basic parameters of the object.

INPUT:

- `r` – a number which can be coerced to a float, serving roughly as the radius of the object
- `width` – a number which can be coerced to a float, which gives the width of the object
- `twists` – (default: 1) an integer, giving the number of twists in the object (where one twist is the 'traditional' Möbius strip)

EXAMPLES:

```
sage: from sage.plot.plot3d.parametric_surface import MoebiusStrip
sage: M = MoebiusStrip(3,3)
sage: M.show()
```

`eval`(*u*, *v*)

Return a tuple for x, y, z coordinates for the given `u` and `v` for this MoebiusStrip instance.

EXAMPLES:

```
sage: from sage.plot.plot3d.parametric_surface import MoebiusStrip
sage: N = MoebiusStrip(7,3,2) # two twists
sage: N.eval(-1,0)
(4.0, 0.0, -0.0)
```

`get_grid`(*ds*)

Return appropriate u and v ranges for this MoebiusStrip instance.

This is intended for internal use in creating an actual plot.

INPUT:

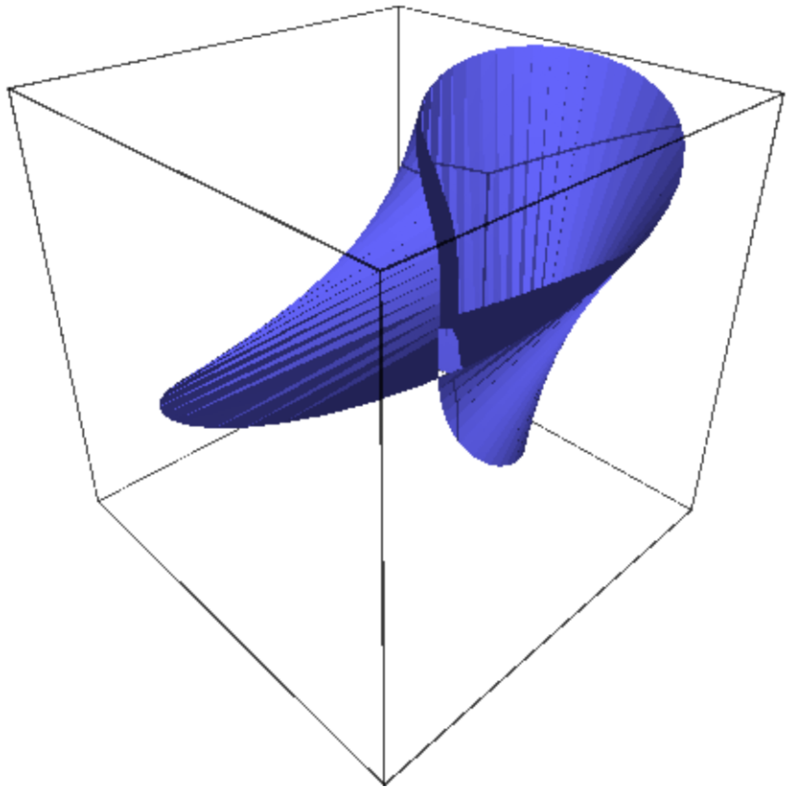
- `ds` – A number, typically coming from a RenderParams object, which helps determine the increment for the v range for the MoebiusStrip object.

EXAMPLES:

```
sage: from sage.plot.plot3d.parametric_surface import MoebiusStrip
sage: N = MoebiusStrip(7,3,2) # two twists
sage: N.get_grid(N.default_render_params().ds)
([-1, 1], [0.0, 0.12566370614359174, 0.25132741228718347, 0.37699111843077515, .
↪..])
```

class `sage.plot.plot3d.parametric_surface.`**`ParametricSurface`**

Bases: *`IndexFaceSet`*



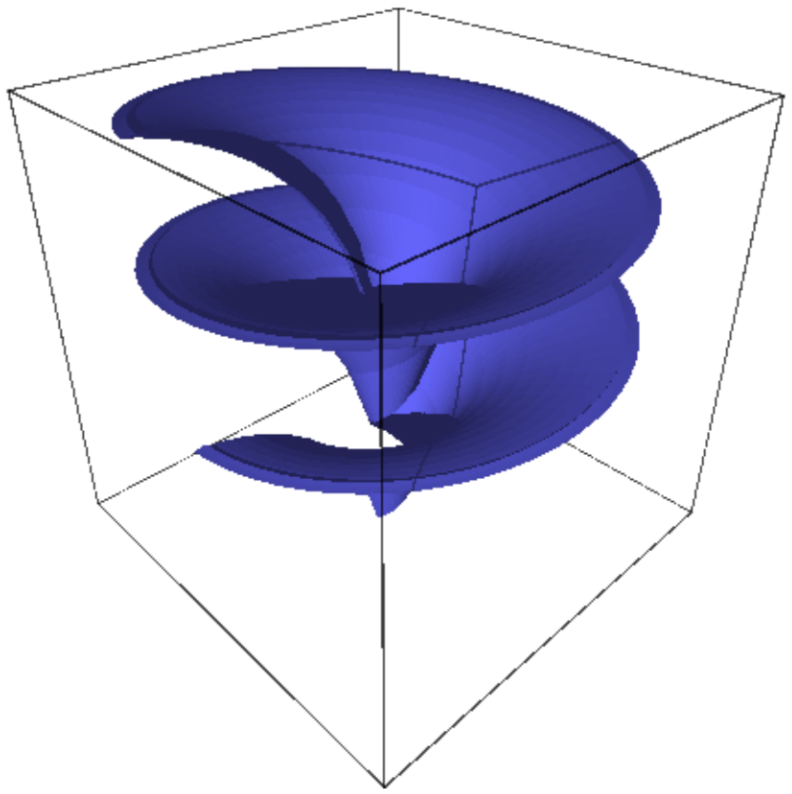
Base class that initializes the ParametricSurface graphics type. This sets options, the function to be plotted, and the plotting array as attributes.

INPUT:

- `f` - (default: `None`) The defining function. Either a tuple of three functions, or a single function which returns a tuple, taking two python floats as input. To subclass, pass `None` for `f` and override `eval_c` or `eval` instead.
- `domain` - (default: `None`) A tuple of two lists, defining the grid of u, v values. If `None`, this will be calculated automatically.
- `color` - (default: `None`) A pair (h, c) where h is a function with values in $[0, 1]$ and c is a colormap. The color of a point p is then defined as the composition $c(h(p))$

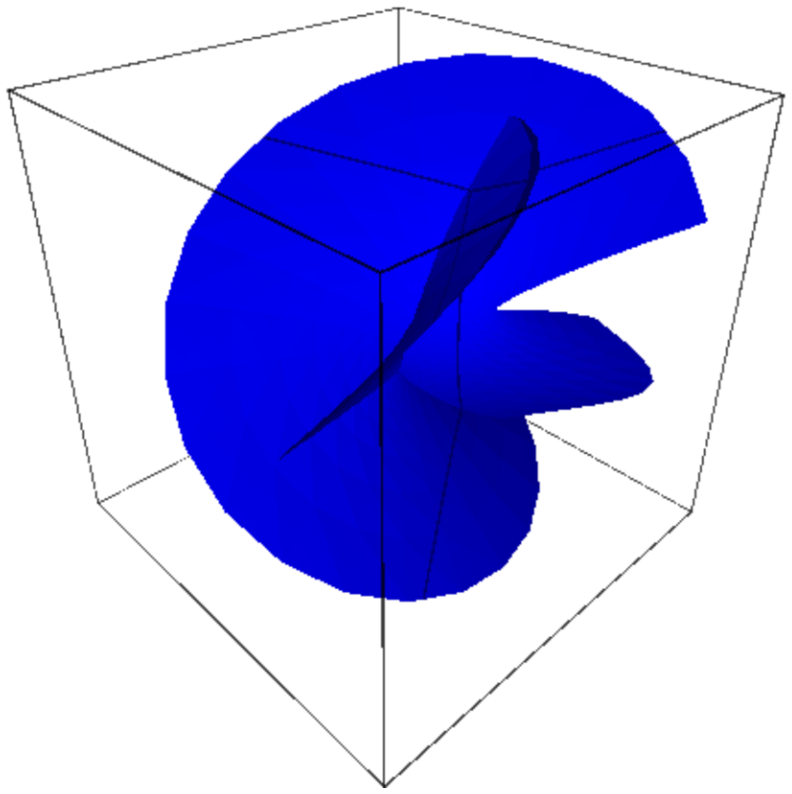
EXAMPLES:

```
sage: from sage.plot.plot3d.parametric_surface import ParametricSurface
sage: def f(x,y): return cos(x)*sin(y), sin(x)*sin(y), cos(y)+log(tan(y/2))+0.2*x
sage: S = ParametricSurface(f, (srange(0,12.4,0.1), srange(0.1,2,0.1)))
sage: show(S)

sage: len(S.face_list())
2214
```



The Hessenberg surface:

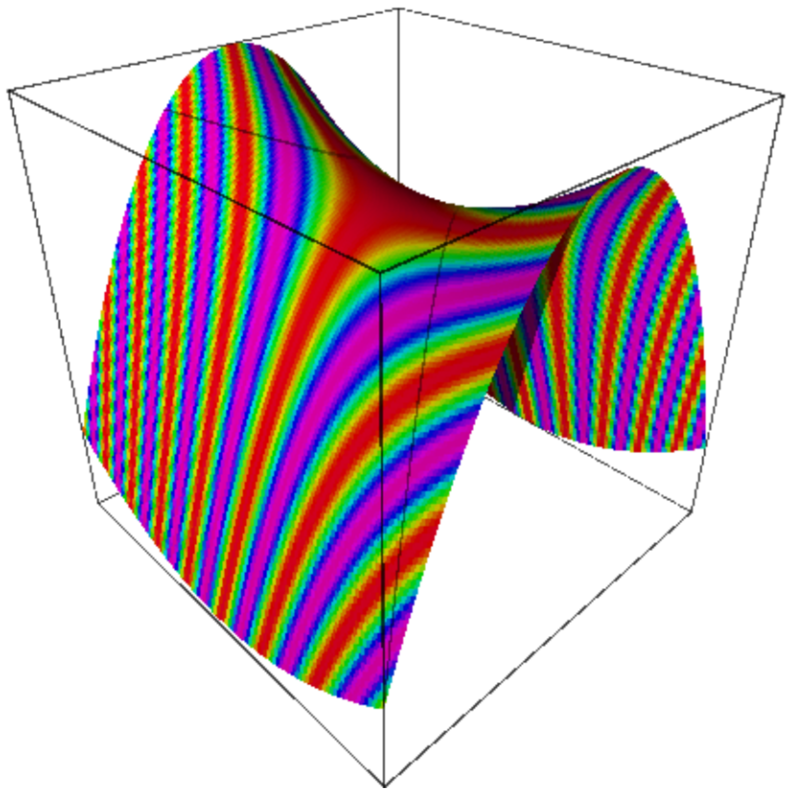
```
sage: def f(u,v):
....:     a = 1
....:     from math import cos, sin, sinh, cosh
....:     x = cos(a)*(cos(u)*sinh(v)-cos(3*u)*sinh(3*v)/3) + sin(a)*(
....:         sin(u)*cosh(v)-sin(3*u)*cosh(3*v)/3)
....:     y = cos(a)*(sin(u)*sinh(v)+sin(3*u)*sinh(3*v)/3) + sin(a)*(
....:         -cos(u)*cosh(v)-cos(3*u)*cosh(3*v)/3)
....:     z = cos(a)*cos(2*u)*cosh(2*v)+sin(a)*sin(2*u)*sinh(2*v)
....:     return (x,y,z)
sage: v = srange(float(0),float((3/2)*pi),float(0.1))
sage: S = ParametricSurface(f, (srange(float(0),float(pi),float(0.1)),
....:                srange(float(-1),float(1),float(0.1))), color="blue")
sage: show(S)
```



A colored example using the color keyword:

```
sage: def g(x,y): return x, y, - x**2 + y**2
sage: def c(x,y): return sin((x-y/2)*y/4)**2
sage: cm = colormaps.gist_rainbow
sage: P = ParametricSurface(g, (srange(-10,10,0.1),
....:   srange(-5,5.0,0.1)),color=(c,cm))
sage: P.show(viewer='tachyon')
```

bounding_box()



Return the lower and upper corners of a 3D bounding box for `self`.

This is used for rendering and `self` should fit entirely within this box.

Specifically, the first point returned should have x, y, and z coordinates should be the respective infimum over all points in `self`, and the second point is the supremum.

EXAMPLES:

```
sage: from sage.plot.plot3d.parametric_surface import MoebiusStrip
sage: M = MoebiusStrip(7,3,2)
sage: M.bounding_box()
((-10.0, -7.53907349250478..., -2.9940801852848145),
 (10.0, 7.53907349250478..., 2.9940801852848145))
```

default_render_params()

Return an instance of RenderParams suitable for plotting this object.

dual()

Return an `IndexFaceSet` which is the dual of the *ParametricSurface* object as a triangulated surface.

EXAMPLES:

As one might expect, this gives an icosahedron:

```
sage: D = dodecahedron()
sage: D.dual()
Graphics3d Object
```

But any enclosed surface should work:

```
sage: from sage.plot.plot3d.shapes import Torus
sage: T =  Torus(1, .2)
sage: T.dual()
Graphics3d Object
sage: T.is_enclosed()
True
```

Surfaces which are not enclosed, though, should raise an exception:

```
sage: from sage.plot.plot3d.parametric_surface import MoebiusStrip
sage: M = MoebiusStrip(3,1)
sage: M.is_enclosed()
False
sage: M.dual()
Traceback (most recent call last):
...
NotImplementedError: this is only implemented for enclosed surfaces
```

eval(*u*, *v*)

get_grid(*ds*)

is_enclosed()

Return a boolean telling whether or not it is necessary to render the back sides of the polygons (assuming, of course, that they have the correct orientation).

This is calculated in by verifying the opposite edges of the rendered domain either line up or are pinched together.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Sphere
sage: Sphere(1).is_enclosed()
True

sage: from sage.plot.plot3d.parametric_surface import MoebiusStrip
sage: MoebiusStrip(1,0.2).is_enclosed()
False
```

jmol_repr(*render_params*)

Return a representation of the object suitable for plotting using Jmol.

json_repr(*render_params*)

Return a representation of the object in JSON format as a list with one element, which is a string of a dictionary listing vertices, faces and colors.

obj_repr(*render_params*)

Return a complete representation of object with name, texture, and lists of vertices, faces, and back-faces.

plot()

Draw a 3D plot of this graphics object, which just returns this object since this is already a 3D graphics object. Needed to support PLOT in doctrings, see github issue #17498

EXAMPLES:

```
sage: S = parametric_plot3d( (sin, cos, lambda u: u/10), (0, 20))
sage: S.plot() is S
True
```

tachyon_repr(*render_params*)

Return representation of the object suitable for plotting using Tachyon ray tracer.

threejs_repr(*render_params*)

Return a representation of the surface suitable for plotting with three.js.

EXAMPLES:

```
sage: _ = var('x,y')
sage: P = plot3d(x^2-y^2, (x, -2, 2), (y, -2, 2))
sage: P.threejs_repr(P.default_render_params())
[('surface',
  {'color': '#6666ff',
   'faces': [[0, 1, 2, 3],
    ...
   'opacity': 1.0,
   'vertices': [{'x': -2.0, 'y': -2.0, 'z': 0.0},
    ...
    {'x': 2.0, 'y': 2.0, 'z': 0.0}]})]
```

triangulate(*render_params=None*)

Call self.eval_grid() for all (u, v) in urange $\times$ vrange to construct this surface.

The most complicated part of this code is identifying shared vertices and shrinking trivial edges. This is not done so much to save memory, rather it is needed so normals of the triangles can be calculated correctly.

x3d_geometry()

Return XML-like representation of the coordinates of all points in a triangulation of the object along with an indexing of those points.

3.6 Graphics 3D object for representing and triangulating isosurfaces

AUTHORS:

- Robert Hanson (2007): initial Java version, in Jmol.
- Carl Witty (2009-01): first Cython version.
- Bill Cauchois (2009): improvements for inclusion into Sage.

class sage.plot.plot3d.implicit_surface.**ImplicitSurface**

Bases: *IndexFaceSet*

bounding_box()

Return a bounding box for the `ImplicitSurface`, as a tuple of two 3-dimensional points.

EXAMPLES:

Note that the bounding box corresponds exactly to the x-, y-, and z- range:

```
sage: from sage.plot.plot3d.implicit_surface import ImplicitSurface
sage: G = ImplicitSurface(0, (0, 1), (0, 1), (0, 1))
sage: G.bounding_box()
((0.0, 0.0, 0.0), (1.0, 1.0, 1.0))
```

color_function

colormap

contours

f

gradient

jmol_repr(*render_params*)

Return a representation of this object suitable for use with the Jmol renderer.

json_repr(*render_params*)

Return a representation of this object in JavaScript Object Notation (JSON).

obj_repr(*render_params*)

Return a representation of this object in the .obj format.

plot_points

region

smooth

tachyon_repr(*render_params*)

Return a representation of this object suitable for use with the Tachyon renderer.

threejs_repr(*render_params*)

Return a representation of the surface suitable for plotting with three.js.

EXAMPLES:

```
sage: from sage.plot.plot3d.implicit_surface import ImplicitSurface
sage: _ = var('x,y,z')
sage: G = ImplicitSurface(x + y + z, (x,-1, 1), (y,-1, 1), (z,-1, 1))
sage: G.threejs_repr(G.default_render_params())
[('surface',
  {'color': '#6666ff',
   'faces': [[0, 1, 2],
    ...
   'opacity': 1.0,
   'vertices': [{'x': ...,
     'y': -0.9743589743589...,
     'z': -0.02564102564102...},
    ...
    {'x': -1.0, 'y': 0.9487179487179..., 'z': 0.05128205128205...}]})]
```

triangulate(*force=False*)

The IndexFaceSet will be empty until you call this method, which generates the faces and vertices according to the parameters specified in the constructor for ImplicitSurface.

Note that if you call this method more than once, subsequent invocations will have no effect (this is an optimization to avoid repeated work) unless you specify `force=True` in the keywords.

EXAMPLES:

```
sage: from sage.plot.plot3d.implicit_surface import ImplicitSurface
sage: var('x,y,z')
(x, y, z)
sage: G = ImplicitSurface(x + y + z, (x,-1, 1), (y,-1, 1), (z,-1, 1))
sage: len(G.vertex_list()), len(G.face_list())
(0, 0)
sage: G.triangulate()
sage: len(G.vertex_list()) > 0, len(G.face_list()) > 0
(True, True)
sage: G.show() # This should be fast, since the mesh is already triangulated.
```

vars

xrange

yrange

zrange

class sage.plot.plot3d.implicit_surface.**MarchingCubes**

Bases: `object`

Handles marching cube rendering.

Protocol:

1. Create the class.
2. Call `process_slice` once for each X slice, from self.nx > x >= 0.
3. Call `finish()`, which returns a list of strings.

Note: Actually, only 4 slices ever exist; the caller will re-use old storage.

---

color_function

colormap

contour

finish()

Return the results of the marching cubes algorithm as a list.

The format is specific to the subclass implementing this method.

gradient

nx

ny

nz

region

results

smooth

transform

xrange

yrange

zrange

class sage.plot.plot3d.implicit_surface.**MarchingCubesTriangles**

Bases: *MarchingCubes*

A subclass of MarchingCubes that returns its results as a list of triangles, including their vertices and normals (if `smooth=True`).

And also their vertex colors if a vertex coloring function is given.

add_triangle(*v1*, *v2*, *v3*)

Called when a new triangle is generated by the marching cubes algorithm to update the results array.

process_cubes(*_left*, *_right*)

process_slice(*x*, *slice*)

Process a single slice of function evaluations at the specified x coordinate.

EXAMPLES:

```
sage: from sage.plot.plot3d.implicit_surface import MarchingCubesTriangles
sage: import numpy as np
sage: cube_marcher = MarchingCubesTriangles((-2, 2), (-2, 2), (-2, 2), 4, (10,
↪)*3, smooth=False)
sage: f = lambda x, y, z: x^2 + y^2 + z^2
sage: slices = np.zeros((10, 10, 10), dtype=np.double)
sage: for x in reversed(range(0, 10)):
....:     for y in range(0, 10):
....:         for z in range(0, 10):
....:             slices[x, y, z] = f(*[a * (4 / 9) -2 for a in (x, y, z)])
....:     cube_marcher.process_slice(x, slices[x, :, :])
sage: faces = cube_marcher.finish()
sage: faces[0][0]
{'x': 1.555555555555..., 'y': -1.111111111111..., 'z': -0.555555555555...}
```

We render the isosurface using IndexFaceSet:

```
sage: from sage.plot.plot3d.index_face_set import IndexFaceSet
sage: IndexFaceSet([tuple((p['x'], p['y'], p['z']) for p in face) for face in␣
↪faces])
Graphics3d Object
```

slices

x_vertices

y_vertices

y_vertices_swapped

z_vertices

z_vertices_swapped

class sage.plot.plot3d.implicit_surface.**VertexInfo**

Bases: object

sage.plot.plot3d.implicit_surface.**render_implicit**(*f*, *xrange*, *yrange*, *zrange*, *plot_points*, *cube_marchers*)

INPUT:

- f - a (fast!) callable function
- xrange - a 2-tuple (x_min, x_max)
- yrange - a 2-tuple (y_min, y_may)
- zrange - a 2-tuple (z_min, z_maz)
- plot_points - a triple of integers indicating the number of function evaluations in each direction.
- cube_marchers - a list of cube marchers, one for each contour.

OUTPUT:

A representation of the isosurface, in the format specified by the individual cube marchers.

CHAPTER

FOUR

INFRASTRUCTURE

4.1 Texture support

This module provides texture/material support for 3D Graphics objects and plotting. This is a very rough common interface for Tachyon, x3d, and obj (mtl). See *Texture* for full details about options and use.

Initially, we have no textures set:

```
sage: sage.plot.plot3d.base.Graphics3d().texture_set()
set()
```

However, one can access these textures in the following manner:

```
sage: G = tetrahedron(color='red') + tetrahedron(color='yellow') + tetrahedron(color='red
↪', opacity=0.5)
sage: [t for t in G.texture_set() if t.color == colors.red] # we should have two red␣
↪textures
[Texture(texture..., red, ff0000), Texture(texture..., red, ff0000)]
sage: [t for t in G.texture_set() if t.color == colors.yellow] # ...and one yellow
[Texture(texture..., yellow, ffff00)]
```

And the Texture objects keep track of all their data:

```
sage: T = tetrahedron(color='red', opacity=0.5)
sage: t = T.get_texture()
sage: t.opacity
0.5
sage: T # should be translucent
Graphics3d Object
```

AUTHOR:

- Robert Bradshaw (2007-07-07) Initial version.

class `sage.plot.plot3d.texture.`**`Texture`**(*id*, *color=(0.4, 0.4, 1)*, *opacity=1*, *ambient=0.5*, *diffuse=1*, *specular=0*, *shininess=1*, *name=None*, ***kwds*)

Bases: `WithEqualityById`, `SageObject`

Class representing a texture.

See documentation of `Texture.__classcall__` for more details and examples.

EXAMPLES:

We create a translucent texture:

```
sage: from sage.plot.plot3d.texture import Texture
sage: t = Texture(opacity=0.6)
sage: t
Texture(texture..., 6666ff)
sage: t.opacity
0.6
sage: t.jmol_str('obj')
'color obj translucent 0.4 [102,102,255]'
sage: t.mtl_str()
'newmtl texture...\nKa 0.2 0.2 0.5\nKd 0.4 0.4 1.0\nKs 0.0 0.0 0.0\nillum 1\nNs 1.0\
↪nd 0.6'
sage: t.x3d_str()
"<Appearance><Material diffuseColor='0.4 0.4 1.0' shininess='1.0' specularColor='0.
↪0 0.0 0.0'/></Appearance>"
```

hex_rgb()

EXAMPLES:

```
sage: from sage.plot.plot3d.texture import Texture
sage: Texture('red').hex_rgb()
'ff0000'
sage: Texture((1, .5, 0)).hex_rgb()
'ff7f00'
```

jmol_str(*obj*)

Convert Texture object to string suitable for Jmol applet.

INPUT:

- obj - str

EXAMPLES:

```
sage: from sage.plot.plot3d.texture import Texture
sage: t = Texture(opacity=0.6)
sage: t.jmol_str('obj')
'color obj translucent 0.4 [102,102,255]'
```

```
sage: sum([dodecahedron(center=[2.5*x, 0, 0], color=(1, 0, 0, x/10)) for x in␣
↪range(11)]).show(aspect_ratio=[1,1,1], frame=False, zoom=2)
```

mtl_str()

Convert Texture object to string suitable for mtl output.

EXAMPLES:

```
sage: from sage.plot.plot3d.texture import Texture
sage: t = Texture(opacity=0.6)
sage: t.mtl_str()
'newmtl texture...\nKa 0.2 0.2 0.5\nKd 0.4 0.4 1.0\nKs 0.0 0.0 0.0\nillum 1\nNs␣
↪1.0\nd 0.6'
```

tachyon_str()

Convert Texture object to string suitable for Tachyon ray tracer.

EXAMPLES:

```
sage: from sage.plot.plot3d.texture import Texture
sage: t = Texture(opacity=0.6)
sage: t.tachyon_str()
'Texdef texture...\n  Ambient 0.3333333333333333 Diffuse 0.6666666666666666␣
↪Specular 0.0 Opacity 0.6\n   Color 0.4 0.4 1.0\n   TexFunc 0'
```

x3d_str()

Convert Texture object to string suitable for x3d.

EXAMPLES:

```
sage: from sage.plot.plot3d.texture import Texture
sage: t = Texture(opacity=0.6)
sage: t.x3d_str()
"<Appearance><Material diffuseColor='0.4 0.4 1.0' shininess='1.0' specularColor=
↪'0.0 0.0 0.0'/></Appearance>"
```

sage.plot.plot3d.texture.**is_Texture**(*x*)

Deprecated. Use `isinstance(x, Texture)` instead.

EXAMPLES:

```
sage: from sage.plot.plot3d.texture import is_Texture, Texture
sage: t = Texture(0.5)
sage: is_Texture(t)
doctest:...: DeprecationWarning: Please use isinstance(x, Texture)
See https://github.com/sagemath/sage/issues/27593 for details.
True
```

sage.plot.plot3d.texture.**parse_color**(*info*, *base=None*)

Parse the color.

It transforms a valid color string into a color object and a color object into an RBG tuple of length 3. Otherwise, it multiplies the info by the base color.

INPUT:

- `info` - color, valid color str or number
- `base` - tuple of length 3 (optional, default: `None`)

OUTPUT:

A tuple or color.

EXAMPLES:

From a color:

```
sage: from sage.plot.plot3d.texture import parse_color
sage: c = Color('red')
sage: parse_color(c)
(1.0, 0.0, 0.0)
```

From a valid color str:

```
sage: parse_color('red')
RGB color (1.0, 0.0, 0.0)
sage: parse_color('#ff0000')
RGB color (1.0, 0.0, 0.0)
```

From a non valid color str:

```
sage: parse_color('redd')
Traceback (most recent call last):
...
ValueError: unknown color 'redd'
```

From an info and a base:

```
sage: opacity = 10
sage: parse_color(opacity, base=(.2,.3,.4))
(2.0, 3.0, 4.0)
```

4.2 Indexed face sets

Graphics3D object that consists of a list of polygons, also used for triangulations of other objects.

Usually these objects are not created directly by users.

AUTHORS:

- Robert Bradshaw (2007-08-26): initial version
- Robert Bradshaw (2007-08-28): significant optimizations

Todo: Smooth triangles using vertex normals

class sage.plot.plot3d.index_face_set.**EdgeIter**

Bases: object

A class for iteration over edges

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import *
sage: S = Box(1,2,3)
sage: len(list(S.edges())) == 12   # indirect doctest
True
```

class sage.plot.plot3d.index_face_set.**FaceIter**

Bases: object

A class for iteration over faces

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import *
sage: S = Box(1,2,3)
```

(continues on next page)

(continued from previous page)

```
sage: len(list(S.faces())) == 6   # indirect doctest
True
```

class sage.plot.plot3d.index_face_set.**IndexFaceSet**

Bases: *PrimitiveObject*

Graphics3D object that consists of a list of polygons, also used for triangulations of other objects.

Polygons (mostly triangles and quadrilaterals) are stored in the c struct `face_c` (see transform.pyx). Rather than storing the points directly for each polygon, each face consists a list of pointers into a common list of points which are basically triples of doubles in a `point_c`.

Moreover, each face has an attribute `color` which is used to store color information when faces are colored. The red/green/blue components are then available as floats between 0 and 1 using `color.r,color.g,color.b`.

Usually these objects are not created directly by users.

EXAMPLES:

```
sage: from sage.plot.plot3d.index_face_set import IndexFaceSet
sage: S = IndexFaceSet([[(1,0,0),(0,1,0),(0,0,1)], [(1,0,0),(0,1,0),(0,0,0)]])
sage: S.face_list()
[[(1.0, 0.0, 0.0), (0.0, 1.0, 0.0), (0.0, 0.0, 1.0)],
 [(1.0, 0.0, 0.0), (0.0, 1.0, 0.0), (0.0, 0.0, 0.0)]]
sage: S.vertex_list()
[(1.0, 0.0, 0.0), (0.0, 1.0, 0.0), (0.0, 0.0, 1.0), (0.0, 0.0, 0.0)]

sage: def make_face(n): return [(0,0,n),(0,1,n),(1,1,n),(1,0,n)]
sage: S = IndexFaceSet([make_face(n) for n in range(10)])
sage: S.show()

sage: point_list = [(1,0,0),(0,1,0)] + [(0,0,n) for n in range(10)]
sage: face_list = [[0,1,n] for n in range(2,10)]
sage: S = IndexFaceSet(face_list, point_list, color='red')
sage: S.face_list()
[[(1.0, 0.0, 0.0), (0.0, 1.0, 0.0), (0.0, 0.0, 0.0)],
 [(1.0, 0.0, 0.0), (0.0, 1.0, 0.0), (0.0, 0.0, 1.0)],
 [(1.0, 0.0, 0.0), (0.0, 1.0, 0.0), (0.0, 0.0, 2.0)],
 [(1.0, 0.0, 0.0), (0.0, 1.0, 0.0), (0.0, 0.0, 3.0)],
 [(1.0, 0.0, 0.0), (0.0, 1.0, 0.0), (0.0, 0.0, 4.0)],
 [(1.0, 0.0, 0.0), (0.0, 1.0, 0.0), (0.0, 0.0, 5.0)],
 [(1.0, 0.0, 0.0), (0.0, 1.0, 0.0), (0.0, 0.0, 6.0)],
 [(1.0, 0.0, 0.0), (0.0, 1.0, 0.0), (0.0, 0.0, 7.0)]]
sage: S.show()
```

A simple example of colored IndexFaceSet (github issue #12212):

```
sage: from sage.plot.plot3d.index_face_set import IndexFaceSet
sage: from sage.plot.plot3d.texture import Texture
sage: point_list = [(2,0,0),(0,2,0),(0,0,2),(0,1,1),(1,0,1),(1,1,0)]
sage: face_list = [[0,4,5],[3,4,5],[2,3,4],[1,3,5]]
sage: col = rainbow(10, 'rgbtuple')
sage: t_list = [Texture(col[i]) for i in range(10)]
sage: S = IndexFaceSet(face_list, point_list, texture_list=t_list)
sage: S.show(viewer='tachyon')
```

add_condition(*condition*, *N=100*, *eps=1e-06*)

Cut the surface according to the given condition.

This allows to take the intersection of the surface with a domain in 3-space, in such a way that the result has a smooth boundary.

INPUT:

- `condition` – boolean function on ambient space, that defines the domain
- `N` – max number of steps used by the bisection method (default: 100) to cut the boundary triangles that are not entirely within the domain.
- `eps` – target accuracy in the intersection (default: 1.0e-6)

OUTPUT:

an `IndexFaceSet`

This will contain both triangular and quadrilateral faces.

EXAMPLES:

```
sage: var('x,y,z')                                                        #␣
↪needs sage.symbolic
(x, y, z)
sage: P = implicit_plot3d(z-x*y,(-2,2),(-2,2),(-2,2))                     #␣
↪needs sage.symbolic
sage: def condi(x,y,z):
....:     return bool(x*x+y*y+z*z <= Integer(1))
sage: R = P.add_condition(condi, 20); R                                   #␣
↪needs sage.symbolic
Graphics3d Object
```

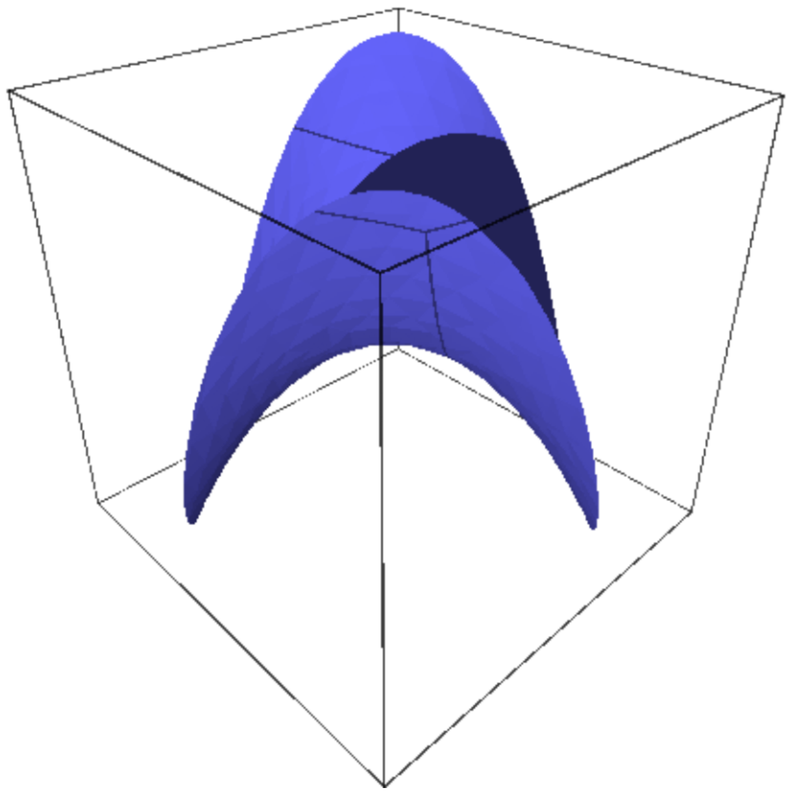
An example with colors:

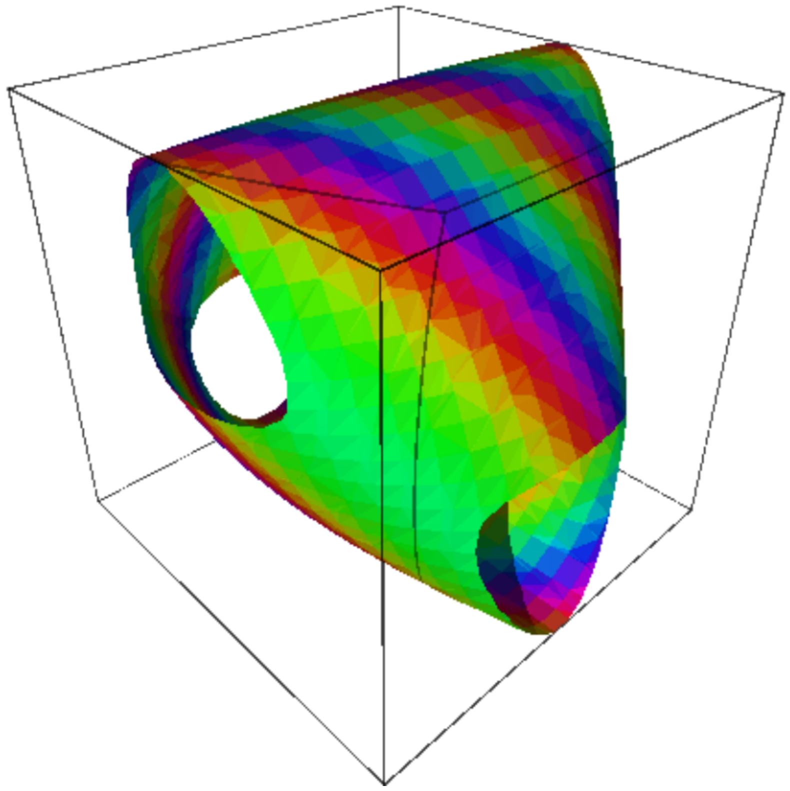
```
sage: def condi(x,y,z):
....:     return bool(x*x+y*y <= 1.1)
sage: cm = colormaps.hsv
sage: cf = lambda x,y,z: float(x+y) % 1
sage: P = implicit_plot3d(x**2+y**2+z**2-1-x**2*z+y**2*z,                 #␣
↪needs sage.symbolic
....:                     (-2,2),(-2,2),(-2,2),color=(cm,cf))
sage: R = P.add_condition(condi,40); R                                    #␣
↪needs sage.symbolic
Graphics3d Object
```

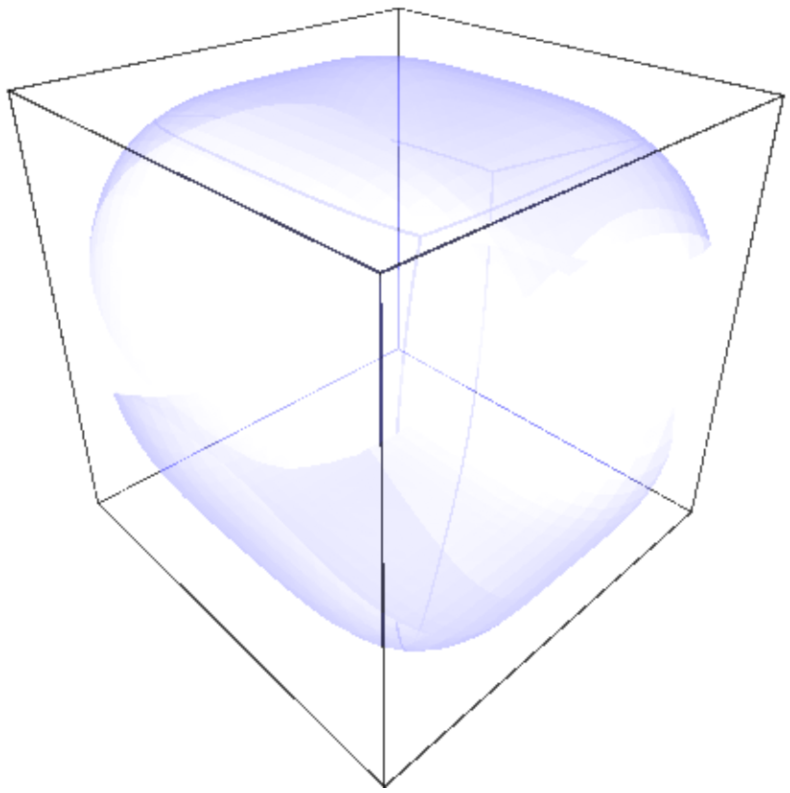
An example with transparency:

```
sage: P = implicit_plot3d(x**4+y**4+z**2-4, (x,-2,2), (y,-2,2), (z,-2,2),    #␣
↪needs sage.symbolic
....:                     alpha=0.3)
sage: def cut(a,b,c):
....:     return a*a+c*c > 2
sage: Q = P.add_condition(cut,40); Q                                      #␣
↪needs sage.symbolic
Graphics3d Object
```

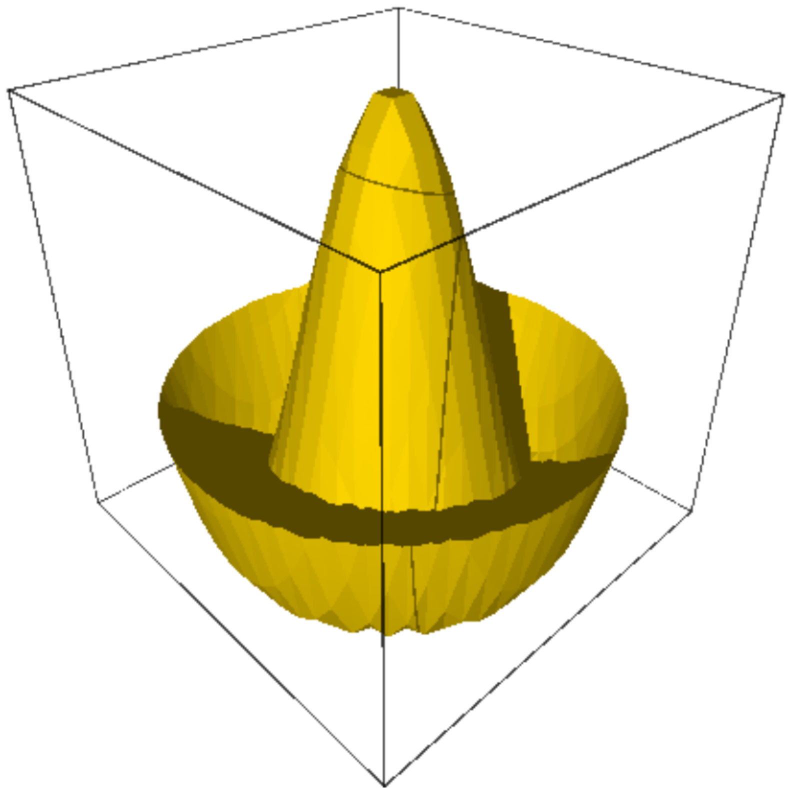
A sombrero with quadrilaterals:







```
sage: P = plot3d(-sin(2*x*x+2*y*y)*exp(-x*x-y*y), (x,-2,2), (y,-2,2),        #␣
↪needs sage.symbolic
....:            color='gold')
sage: def cut(x,y,z):
....:     return x*x+y*y < 1
sage: Q = P.add_condition(cut);Q                                              #␣
↪needs sage.symbolic
Graphics3d Object
```



Todo:

- Use a dichotomy to search for the place where to cut,
- Compute the cut only once for each edge.

bounding_box()

Calculate the bounding box for the vertices in this object (ignoring infinite or NaN coordinates).

OUTPUT:

a tuple ((low_x, low_y, low_z), (high_x, high_y, high_z)), which gives the coordinates of opposite corners of the bounding box.

EXAMPLES:

```
sage: x,y = var('x,y')                                                    #␣
↪needs sage.symbolic
sage: p = plot3d(sqrt(sin(x)*sin(y)), (x,0,2*pi), (y,0,2*pi))             #␣
↪needs sage.symbolic
sage: p.bounding_box()                                                    #␣
↪needs sage.symbolic
((0.0, 0.0, 0.0), (6.283185307179586, 6.283185307179586, 0.9991889981715697))
```

dual(***kwds*)

Return the dual.

EXAMPLES:

```
sage: S = cube()
sage: T = S.dual()
sage: len(T.vertex_list())
6
```

edge_list()

Return the list of edges.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import *
sage: S = Box(1,2,3)
sage: S.edge_list()[0]
((1.0, -2.0, 3.0), (1.0, 2.0, 3.0))
```

edges()

An iterator over the edges.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import *
sage: S = Box(1,2,3)
sage: list(S.edges())[0]
((1.0, -2.0, 3.0), (1.0, 2.0, 3.0))
```

face_list(*render_params=None*)

Return the list of faces.

Every face is given as a tuple of vertices.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import *
sage: S = Box(1,2,3)
sage: S.face_list(S.default_render_params())[0]
[(1.0, 2.0, 3.0), (-1.0, 2.0, 3.0), (-1.0, -2.0, 3.0), (1.0, -2.0, 3.0)]
```

faces()

An iterator over the faces.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import *
sage: S = Box(1,2,3)
sage: list(S.faces()) == S.face_list()
True
```

has_local_colors()

Return True if and only if every face has an individual color.

EXAMPLES:

```
sage: from sage.plot.plot3d.index_face_set import IndexFaceSet
sage: from sage.plot.plot3d.texture import Texture
sage: point_list = [(2,0,0),(0,2,0),(0,0,2),(0,1,1),(1,0,1),(1,1,0)]
sage: face_list = [[0,4,5],[3,4,5],[2,3,4],[1,3,5]]
sage: col = rainbow(10, 'rgbtuple')
sage: t_list=[Texture(col[i]) for i in range(10)]
sage: S = IndexFaceSet(face_list, point_list, texture_list=t_list)
sage: S.has_local_colors()
True

sage: from sage.plot.plot3d.shapes import *
sage: S = Box(1,2,3)
sage: S.has_local_colors()
False
```

index_faces()

Return the list over all faces of the indices of the vertices.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import *
sage: S = Box(1,2,3)
sage: S.index_faces()
[[0, 1, 2, 3],
 [0, 4, 5, 1],
 [0, 3, 6, 4],
 [5, 4, 6, 7],
 [6, 3, 2, 7],
 [2, 1, 5, 7]]
```

index_faces_with_colors()

Return the list over all faces of (indices of the vertices, color).

This only works if every face has its own color.

See also:

has_local_colors()

EXAMPLES:

A simple colored one:

```
sage: from sage.plot.plot3d.index_face_set import IndexFaceSet
sage: from sage.plot.plot3d.texture import Texture
sage: point_list = [(2,0,0),(0,2,0),(0,0,2),(0,1,1),(1,0,1),(1,1,0)]
```

(continues on next page)

(continued from previous page)

```
sage: face_list = [[0,4,5],[3,4,5],[2,3,4],[1,3,5]]
sage: col = rainbow(10, 'rgbtuple')
sage: t_list = [Texture(col[i]) for i in range(10)]
sage: S = IndexFaceSet(face_list, point_list, texture_list=t_list)
sage: S.index_faces_with_colors()
[([0, 4, 5], '#ff0000'),
 ([3, 4, 5], '#ff9900'),
 ([2, 3, 4], '#cbff00'),
 ([1, 3, 5], '#33ff00')]
```

When the texture is global, an error is raised:

```
sage: from sage.plot.plot3d.shapes import *
sage: S = Box(1,2,3)
sage: S.index_faces_with_colors()
Traceback (most recent call last):
...
ValueError: the texture is global
```

is_enclosed()

Whether or not it is necessary to render the back sides of the polygons.

One is assuming, of course, that they have the correct orientation.

This is may be passed in on construction. It is also calculated in *sage.plot.plot3d.parametric_surface.ParametricSurface* by verifying the opposite edges of the rendered domain either line up or are pinched together.

EXAMPLES:

```
sage: from sage.plot.plot3d.index_face_set import IndexFaceSet
sage: IndexFaceSet([[(0,0,1),(0,1,0),(1,0,0)]]).is_enclosed()
False
```

jmol_repr(*render_params*)

Return a jmol representation for `self`.

json_repr(*render_params*)

Return a json representation for `self`.

obj_repr(*render_params*)

Return an obj representation for `self`.

partition(*f*)

Partition the faces of `self`.

The partition is done according to the value of a map $f : \mathbf{R}^3 \rightarrow \mathbf{Z}$ applied to the center of each face.

INPUT:

- f – a function from $\mathbf{R}^3$ to $\mathbf{Z}$

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import *
sage: S = Box(1,2,3)
```

(continues on next page)

(continued from previous page)

```
sage: len(S.partition(lambda x,y,z: floor(x+y+z)))
6
```

sticker(*face_list*, *width*, *hover*, ***kwds*)

Return a sticker on the chosen faces.

stickers(*colors*, *width*, *hover*)

Return a group of IndexFaceSets.

INPUT:

- `colors` – list of colors/textures to use (in cyclic order)
- `width` – offset perpendicular into the edge (to create a border) may also be negative
- `hover` – offset normal to the face (usually have to float above the original surface so it shows, typically this value is very small compared to the actual object

OUTPUT:

Graphics3dGroup of stickers

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Box
sage: B = Box(.5,.4,.3, color='black')
sage: S = B.stickers(['red','yellow','blue'], 0.1, 0.05)
sage: S.show()
sage: (S+B).show()
```

stl_binary_repr(*render_params*)

Return data for STL (STereoLithography) representation of the surface.

The STL binary representation is a list of binary strings, one for each triangle.

EXAMPLES:

```
sage: G = sphere()
sage: data = G.stl_binary_repr(G.default_render_params()); len(data)
1368
```

tachyon_repr(*render_params*)

Return a tachyon object for `self`.

EXAMPLES:

A basic test with a triangle:

```
sage: G = polygon([(0,0,1), (1,1,1), (2,0,1)])
sage: s = G.tachyon_repr(G.default_render_params()); s
['TRI V0 0 0 1 V1 1 1 1 V2 2 0 1', ...]
```

A simple colored one:

```
sage: from sage.plot.plot3d.index_face_set import IndexFaceSet
sage: from sage.plot.plot3d.texture import Texture
sage: point_list = [(2,0,0),(0,2,0),(0,0,2),(0,1,1),(1,0,1),(1,1,0)]
sage: face_list = [[0,4,5],[3,4,5],[2,3,4],[1,3,5]]
```

(continues on next page)

(continued from previous page)

```
sage: col = rainbow(10, 'rgbtuple')
sage: t_list = [Texture(col[i]) for i in range(10)]
sage: S = IndexFaceSet(face_list, point_list, texture_list=t_list)
sage: S.tachyon_repr(S.default_render_params())
['TRI V0 2 0 0 V1 1 0 1 V2 1 1 0',
'TEXTURE... AMBIENT 0.3 DIFFUSE 0.7 SPECULAR 0 OPACITY 1.0... COLOR 1 0 0 ...␣
↪TEXFUNC 0',...]
```

threejs_repr(*render_params*)

Return representation of the surface suitable for plotting with three.js.

EXAMPLES:

A simple triangle:

```
sage: G = polygon([(0,0,1), (1,1,1), (2,0,1)])
sage: G.threejs_repr(G.default_render_params())
[('surface',
  {'color': '#0000ff',
   'faces': [[0, 1, 2]],
   'opacity': 1.0,
   'vertices': [{'x': 0.0, 'y': 0.0, 'z': 1.0},
    {'x': 1.0, 'y': 1.0, 'z': 1.0},
    {'x': 2.0, 'y': 0.0, 'z': 1.0}]})]
```

The same but with more options applied:

```
sage: G = polygon([(0,0,1), (1,1,1), (2,0,1)], color='red', opacity=0.5,
....:             render_order=2, threejs_flat_shading=True,
....:             single_side=True, mesh=True, thickness=10, depth_write=True)
sage: G.threejs_repr(G.default_render_params())
[('surface',
  {'color': '#ff0000',
   'depthWrite': True,
   'faces': [[0, 1, 2]],
   'linewidth': 10.0,
   'opacity': 0.5,
   'renderOrder': 2.0,
   'showMeshGrid': True,
   'singleSide': True,
   'useFlatShading': True,
   'vertices': [{'x': 0.0, 'y': 0.0, 'z': 1.0},
    {'x': 1.0, 'y': 1.0, 'z': 1.0},
    {'x': 2.0, 'y': 0.0, 'z': 1.0}]})]
```

vertex_list()

Return the list of vertices.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import *
sage: S = polygon([(0,0,1), (1,1,1), (2,0,1)])
sage: S.vertex_list()[0]
(0.0, 0.0, 1.0)
```

vertices()

An iterator over the vertices.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import *
sage: S = Cone(1,1)
sage: list(S.vertices()) == S.vertex_list()
True
```

x3d_geometry()

Return the x3d data.

EXAMPLES:

A basic test with a triangle:

```
sage: G = polygon([(0,0,1), (1,1,1), (2,0,1)])
sage: print(G.x3d_geometry())

<IndexedFaceSet coordIndex='0,1,2,-1'>
  <Coordinate point='0.0 0.0 1.0,1.0 1.0 1.0,2.0 0.0 1.0'/>
</IndexedFaceSet>
```

A simple colored one:

```
sage: from sage.plot.plot3d.index_face_set import IndexFaceSet
sage: from sage.plot.plot3d.texture import Texture
sage: point_list = [(2,0,0),(0,2,0),(0,0,2),(0,1,1),(1,0,1),(1,1,0)]
sage: face_list = [[0,4,5],[3,4,5],[2,3,4],[1,3,5]]
sage: col = rainbow(10, 'rgbtuple')
sage: t_list = [Texture(col[i]) for i in range(10)]
sage: S = IndexFaceSet(face_list, point_list, texture_list=t_list)
sage: print(S.x3d_geometry())

<IndexedFaceSet solid='False' colorPerVertex='False' coordIndex='0,4,5,-1,3,4,5,
↪-1,2,3,4,-1,1,3,5,-1'>
  <Coordinate point='2.0 0.0 0.0,0.0 2.0 0.0,0.0 0.0 2.0,0.0 1.0 1.0,1.0 0.0 1.
↪0,1.0 1.0 0.0'/>
  <Color color='1.0 0.0 0.0,1.0 0.6000000000000001 0.0,0.7999999999999998 1.0 0.
↪0,0.20000000000000018 1.0 0.0' />
</IndexedFaceSet>
```

class sage.plot.plot3d.index_face_set.**VertexIter**

Bases: `object`

A class for iteration over vertices

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import *
sage: S = Box(1,2,3)
sage: len(list(S.vertices())) == 8   # indirect doctest
True
```

`sage.plot.plot3d.index_face_set.`**`cut_edge_by_bisection`**`(`*pointa*, *pointb*, *condition*, *eps=1e-06*, *N=100*`)`

Cut an intersecting edge using the bisection method.

Given two points (pointa and pointb) and a condition (boolean function), this calculates the position at the edge (defined by both points) where the boolean condition switches its value.

INPUT:

- `pointa`, `pointb` – two points in 3-dimensional space
- `N` – max number of steps in the bisection method (default: 100) to cut the boundary triangles that are not entirely within the domain.
- `eps` – target accuracy in the intersection (default: 1.0e-6)

OUTPUT:

intersection of the edge defined by `pointa` and `pointb`, and `condition`.

EXAMPLES:

```
sage: from sage.plot.plot3d.index_face_set import cut_edge_by_bisection
sage: cut_edge_by_bisection((0.0,0.0,0.0), (1.0,1.0,0.0),
....:                       lambda x,y,z: x**2+y**2+z**2 < 1, eps=1.0E-12)
(0.7071067811864395, 0.7071067811864395, 0.0)
```

`sage.plot.plot3d.index_face_set.`**`midpoint`**`(`*pointa*, *pointb*, *w*`)`

Return the weighted mean of two points in 3-space.

INPUT:

- `pointa`, `pointb` – two points in 3-dimensional space
- `w` – a real weight between 0 and 1.

If the weight is zero, the result is `pointb`. If the weight is one, the result is `pointa`.

EXAMPLES:

```
sage: from sage.plot.plot3d.index_face_set import midpoint
sage: midpoint((1,2,3),(4,4,4),0.8)
(1.60000000000000, 2.40000000000000, 3.20000000000000)
```

`sage.plot.plot3d.index_face_set.`**`sticker`**`(`*face*, *width*, *hover*`)`

Return a sticker over the given face.

4.3 Transformations

`class sage.plot.plot3d.transform.`**`Transformation`**

Bases: `object`

`avg_scale``()`

`get_matrix``()`

`is_skew``(`*eps=1e-05*`)`

`is_uniform``(`*eps=1e-05*`)`

is_uniform_on(*basis*, *eps=1e-05*)

max_scale()

transform_bounding_box(*box*)

transform_point(*x*)

transform_vector(*v*)

sage.plot.plot3d.transform.**rotate_arbitrary**(*v*, *theta*)

Return a matrix that rotates the coordinate space about the axis v by the angle `theta`.

INPUT:

- `theta` - real number, the angle

EXAMPLES:

```
sage: from sage.plot.plot3d.transform import rotate_arbitrary
```

Try rotating about the axes:

```
sage: rotate_arbitrary((1,0,0), 1)
[                 1.0                  0.0                  0.0]
[                 0.0   0.5403023058681398   0.8414709848078965]
[                 0.0  -0.8414709848078965   0.5403023058681398]
sage: rotate_arbitrary((0,1,0), 1)
[ 0.5403023058681398                  0.0  -0.8414709848078965]
[                0.0                  1.0                  0.0]
[ 0.8414709848078965                  0.0   0.5403023058681398]
sage: rotate_arbitrary((0,0,1), 1)
[ 0.5403023058681398   0.8414709848078965                  0.0]
[-0.8414709848078965   0.5403023058681398                  0.0]
[                0.0                  0.0                  1.0]
```

These next two should be the same (up to floating-point errors):

```
sage: rotate_arbitrary((1,1,1), 1)  # rel tol 1e-15
[  0.6935348705787598   0.6390560643047186 -0.33259093488347846]
[-0.33259093488347846   0.6935348705787598   0.6390560643047186]
[  0.6390560643047186  -0.3325909348834784   0.6935348705787598]
sage: rotate_arbitrary((1,1,1), -1)^(-1)  # rel tol 1e-15
[  0.6935348705787598   0.6390560643047186 -0.33259093488347846]
[-0.33259093488347846   0.6935348705787598   0.6390560643047186]
[  0.6390560643047185 -0.33259093488347835   0.6935348705787598]
```

Make sure it does the right thing...:

```
sage: rotate_arbitrary((1,2,3), -1).det()
1.0000000000000002
sage: rotate_arbitrary((1,1,1), 2*pi/3) * vector(RDF, (1,2,3))  # rel tol 2e-15
(1.9999999999999996, 2.9999999999999996, 0.9999999999999999)
sage: rotate_arbitrary((1,2,3), 5) * vector(RDF, (1,2,3))  # rel tol 2e-15
(1.0000000000000002, 2.0, 3.000000000000001)
sage: rotate_arbitrary((1,1,1), pi/7)^7  # rel tol 2e-15
[-0.33333333333333337   0.6666666666666671   0.6666666666666665]
```

(continues on next page)

(continued from previous page)

```
[  0.6666666666666665 -0.33333333333333337   0.6666666666666671]
[  0.6666666666666671   0.6666666666666667 -0.33333333333333326]
```

AUTHORS:

- Robert Bradshaw

ALGORITHM:

There is a formula. Where did it come from? Lets take a quick jaunt into Sage's calculus package…

Setup some variables:

```
sage: vx,vy,vz,theta = var('x y z theta')
```

Symbolic rotation matrices about X and Y axis:

```
sage: def rotX(theta): return matrix(SR, 3, 3, [1, 0, 0,  0, cos(theta), -
↪sin(theta), 0, sin(theta), cos(theta)])
sage: def rotZ(theta): return matrix(SR, 3, 3, [cos(theta), -sin(theta), 0,␣
↪ sin(theta), cos(theta), 0, 0, 0, 1])
```

Normalizing y so that $|v| = 1$. Perhaps there is a better way to tell Maxima that $x^2 + y^2 + z^2 = 1$ which would make for a much cleaner calculation:

```
sage: vy = sqrt(1-vx^2-vz^2)
```

Now we rotate about the x-axis so v is in the xy-plane:

```
sage: t = arctan(vy/vz)+pi/2
sage: m = rotX(t)
sage: new_y = vy*cos(t) - vz*sin(t)
```

And rotate about the z axis so v lies on the x axis:

```
sage: s = arctan(vx/new_y) + pi/2
sage: m = rotZ(s) * m
```

Rotating about v in our old system is the same as rotating about the x-axis in the new:

```
sage: m = rotX(theta) * m
```

Do some simplifying here to avoid blow-up:

```
sage: m = m.simplify_rational()
```

Now go back to the original coordinate system:

```
sage: m = rotZ(-s) * m
sage: m = rotX(-t) * m
```

And simplify every single entry (which is more effective that simplify the whole matrix like above):

```
sage: m.parent()([x.simplify_full() for x in m._list()])  # long time;␣
↪random
[                                       -(cos(theta) - 1)*x^2 + cos(theta) ␣
```

(continues on next page)

(continued from previous page)

```
↪             -(cos(theta) - 1)*sqrt(-x^2 - z^2 + 1)*x + sin(theta)*abs(z) ␣
↪    -((cos(theta) - 1)*x*z^2 + sqrt(-x^2 - z^2 + 1)*sin(theta)*abs(z))/z]
[             -(cos(theta) - 1)*sqrt(-x^2 - z^2 + 1)*x - sin(theta)*abs(z) ␣
↪                           (cos(theta) - 1)*x^2 + (cos(theta) - 1)*z^2 + 1 -
↪((cos(theta) - 1)*sqrt(-x^2 - z^2 + 1)*z*abs(z) - x*z*sin(theta))/abs(z)]
[    -((cos(theta) - 1)*x*z^2 - sqrt(-x^2 - z^2 + 1)*sin(theta)*abs(z))/z -
↪((cos(theta) - 1)*sqrt(-x^2 - z^2 + 1)*z*abs(z) + x*z*sin(theta))/abs(z) ␣
↪                                           -(cos(theta) - 1)*z^2 + cos(theta)]
```

Re-expressing some entries in terms of y and resolving the absolute values introduced by eliminating y, we get the desired result.

4.4 Adaptive refinement code for 3d surface plotting

AUTHOR:

- Tom Boothby – Algorithm design, code
- Joshua Kantor – Algorithm design
- Marshall Hampton – Docstrings and doctests

Todo:

- Parametrizations (cylindrical, spherical)
- Massive optimization

class sage.plot.plot3d.tri_plot.**PlotBlock**(*left*, *left_c*, *top*, *top_c*, *right*, *right_c*, *bottom*, *bottom_c*)

Bases: `object`

A container class to hold information about spatial blocks.

class sage.plot.plot3d.tri_plot.**SmoothTriangle**(*a*, *b*, *c*, *da*, *db*, *dc*, *color=0*)

Bases: *Triangle*

A class for smoothed triangles.

get_normals()

Returns the normals to vertices a, b, and c.

str()

Returns a string representation of the SmoothTriangle of the form

a b c color da db dc

where a, b, and c are the triangle corner coordinates, da, db, dc are normals at each corner, and color is the color.

class sage.plot.plot3d.tri_plot.**Triangle**(*a*, *b*, *c*, *color=0*)

Bases: `object`

A graphical triangle class.

get_vertices()

Returns a tuple of vertex coordinates of the triangle.

`set_color`(*color*)

This method will reset the color of the triangle.

`str`()

Returns a string representation of an instance of the Triangle class of the form

a b c color

where a, b, and c are corner coordinates and color is the color.

`class sage.plot.plot3d.tri_plot.TriangleFactory`

Bases: `object`

`get_colors`(*list*)

Parameters: list: an iterable collection of values which can be cast into colors – typically an RGB triple, or an RGBA 4-tuple

Returns: a list of single parameters which can be passed into the set_color method of the Triangle or SmoothTriangle objects generated by this factory.

`smooth_triangle`(*a*, *b*, *c*, *da*, *db*, *dc*, *color=None*)

Parameters:

- a, b, c : triples (x,y,z) representing corners on a triangle in 3-space
- da, db, dc : triples (dx,dy,dz) representing the normal vector at each point a,b,c

Returns: a SmoothTriangle object with the specified coordinates and normals

`triangle`(*a*, *b*, *c*, *color=None*)

Parameters: a, b, c : triples (x,y,z) representing corners on a triangle in 3-space

Returns: a Triangle object with the specified coordinates

`class sage.plot.plot3d.tri_plot.TrianglePlot`(*triangle_factory*, *f*, *min_x__max_x*, *min_y__max_y*, *g=None*, *min_depth=4*, *max_depth=8*, *num_colors=None*, *max_bend=0.3*)

Bases: `object`

Recursively plots a function of two variables by building squares of 4 triangles, checking at every stage whether or not each square should be split into four more squares. This way, more planar areas get fewer triangles, and areas with higher curvature get more triangles.

`extrema`(*list*)

If the num_colors option has been set, this expands the TrianglePlot's _min and _max attributes to include the minimum and maximum of the argument list.

`interface`(*n*, *p*, *p_c*, *q*, *q_c*)

Takes a pair of lists of points, and compares the (n)th coordinate, and "zips" the lists together into one. The "centers", supplied in p_c and q_c are matched up such that the lists describe triangles whose sides are "perfectly" aligned. This algorithm assumes that p and q start and end at the same point, and are sorted smallest to largest.

`plot_block`(*min_x*, *mid_x*, *max_x*, *min_y*, *mid_y*, *max_y*, *sw_z*, *nw_z*, *se_z*, *ne_z*, *mid_z*, *depth*)

Recursive triangulation function for plotting.

First six inputs are scalars, next 5 are 2-dimensional lists, and the depth argument keeps track of the depth of recursion.

str()

Return a string listing the objects in the instance of the TrianglePlot class.

triangulate(*p*, *c*)

Pass in a list of edge points (p) and center points (c). Triangles will be rendered between consecutive edge points and the center point with the same index number as the earlier edge point.

sage.plot.plot3d.tri_plot.**crossunit**(*u*, *v*)

This function computes triangle normal unit vectors by taking the cross-products of the midpoint-to-corner vectors. It always goes around clockwise so we're guaranteed to have a positive value near 1 when neighboring triangles are parallel. However – crossunit doesn't really return a unit vector. It returns the length of the vector to avoid numerical instability when the length is nearly zero – rather than divide by nearly zero, we multiply the other side of the inequality by nearly zero – in general, this should work a bit better because of the density of floating-point numbers near zero.

CHAPTER

FIVE

BACKENDS

5.1 The Tachyon 3D Ray Tracer

Given any 3D graphics object `M` one can compute a raytraced representation by typing `M.show(viewer='tachyon')`. For example, we draw two translucent spheres that contain a red tube, and render the result using Tachyon.

```
sage: S = sphere(opacity=0.8, aspect_ratio=[1,1,1])
sage: L = line3d([(0,0,0),(2,0,0)], thickness=10, color='red')
sage: M = S + S.translate((2,0,0)) + L
sage: M.show(viewer='tachyon')
```

A number of options can be given to the *show()* method and correspondingly to the *save()* method for saving the generated image to a file:

```
sage: M.show(viewer='tachyon',
....:    antialiasing=True, raydepth=3,
....:    figsize=[12,8], # the image resolution is 100*figsize
....:    camera_position=[4, 4.4, 1], # a distant camera position combined with
....:    zoom=3, # a large zoom factor will decrease perspective distortion.
....:    updir=(0, -0.1, 1), # the camera is slightly tilted
....:    viewdir=(-2.,-2.,-0.5), # slightly off-center
....:    light_position=(4.0, -3.0, 2.0),
....:   )
```

One can also directly control Tachyon by creating a `Tachyon` object and adding elements of the scene one by one, which gives a huge amount of flexibility. For example, here we directly use Tachyon to draw 3 spheres on the coordinate axes:

```
sage: t = Tachyon(xres=500, yres=500, camera_position=(2,0,0))
sage: t.light((4,3,2), 0.2, (1,1,1))
sage: t.texture('t2', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0, color=(1,0,0))
sage: t.texture('t3', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0, color=(0,1,0))
sage: t.texture('t4', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0, color=(0,0,1))
sage: t.sphere((0,0.5,0), 0.2, 't2')
sage: t.sphere((0.5,0,0), 0.2, 't3')
sage: t.sphere((0,0,0.5), 0.2, 't4')
sage: t.show()
```

For scenes with many reflections it is helpful to increase the raydepth option, and turn on antialiasing. The following scene is an extreme case with many reflections between four cotangent spheres:

```
sage: t = Tachyon(camera_position=(0,-4,1), xres=800, yres=600, raydepth=12,
....:             aspectratio=.75, antialiasing=4)
sage: t.light((0.02,0.012,0.001), 0.01, (1,0,0))
sage: t.light((0,0,10), 0.01, (0,0,1))
sage: t.texture('s', color=(.8,1,1), opacity=.9, specular=.95, diffuse=.3, ambient=0.05)
sage: t.texture('p', color=(0,0,1), opacity=1, specular=.2)
sage: t.sphere((-1,-.57735,-0.7071),1,'s')
sage: t.sphere((1,-.57735,-0.7071),1,'s')
sage: t.sphere((0,1.15465,-0.7071),1,'s')
sage: t.sphere((0,0,0.9259),1,'s')
sage: t.plane((0,0,-1.9259),(0,0,1),'p')
sage: t.show() # long time
```

Different projection options are available. The following examples all use a sphere and cube:

```
sage: cedges = [[[1, 1, 1], [-1, 1, 1]], [[1, 1, 1], [1, -1, 1]],
....: [[1, 1, 1], [1, 1, -1]], [[-1, 1, 1], [-1, -1, 1]], [[-1, 1, 1],
....: [-1, 1, -1]], [[1, -1, 1], [-1, -1, 1]], [[1, -1, 1], [1, -1, -1]],
....: [[-1, -1, 1], [-1, -1, -1]], [[1, 1, -1], [-1, 1, -1]],
....: [[1, 1, -1], [1, -1, -1]], [[-1, 1, -1], [-1, -1, -1]],
....: [[1, -1, -1], [-1, -1, -1]]]
```

The default projection is `'perspective'`:

```
sage: t = Tachyon(xres=800, yres=600, camera_position=(-1.5,0.0,0.0), zoom=.2)
sage: t.texture('t1', color=(0,0,1))
sage: for ed in cedges:
....:     t.fcylinder(ed[0], ed[1], .05, 't1')
sage: t.light((-4,-4,4), .1, (1,1,1))
sage: t.show()
```

Another option is `projection='fisheye'`, which requires frustum information. The frustum data is (bottom angle, top angle, left angle, right angle):

```
sage: t = Tachyon(xres=800, yres=600, camera_position=(-1.5,0.0,0.0),
....: projection='fisheye', frustum=(-1.2, 1.2, -1.2, 1.2))
sage: t.texture('t1', color=(0,0,1))
sage: for ed in cedges:
....:     t.fcylinder(ed[0], ed[1], .05, 't1')
sage: t.light((-4,-4,4), .1, (1,1,1))
sage: t.show()
```

Finally there is the `projection='perspective_dof'` option.

```
sage: T = Tachyon(xres=800, antialiasing=4, raydepth=10,
....:             projection='perspective_dof', focallength='1.0', aperture='.0025')
sage: T.light((0,5,7), 1.0, (1,1,1))
sage: T.texture('t1', opacity=1, specular=.3)
sage: T.texture('t2', opacity=1, specular=.3, color=(0,0,1))
sage: T.texture('t3', opacity=1, specular=1, color=(1,.8,1), diffuse=0.2)
sage: T.plane((0,0,-1), (0,0,1), 't3')
sage: ttlist = ['t1', 't2']
sage: tt = 't1'
```

(continues on next page)

(continued from previous page)

```
sage: T.cylinder((0,0,.1), (1,1/3,0), .05, 't3')
sage: for q in srange(-3, 100, .15):
....:     if tt == 't1':
....:         tt = 't2'
....:     else:
....:         tt = 't1'
....:     T.sphere((q, q/3+.3*sin(3*q), .1+.3*cos(3*q)), .1, tt)
sage: T.show()
```

Image files in the ppm format can be used to tile planes or cover cylinders or spheres. In this example an image is created and then used to tile the plane:

```
sage: T = Tachyon(xres=800, yres=600, camera_position=(-2.0,-.1,.3),
....:             projection='fisheye', frustum=(-1.0, 1.0, -1.0, 1.0))
sage: T.texture('t1',color=(0,0,1))
sage: for ed in cedges:
....:     T.fcylinder(ed[0], ed[1], .05, 't1')
sage: T.light((-4,-4,4),.1,(1,1,1))
sage: fname_png = tmp_filename(ext='.png')
sage: fname_ppm = tmp_filename(ext='.ppm')
sage: T.save(fname_png)
sage: r2 = os.system('convert '+fname_png+' '+fname_ppm)    # optional -- ImageMagick

sage: # optional - imagemagick
sage: T = Tachyon(xres=800, yres=600,
....:             camera_position=(-2.0,-.1,.3),
....:             projection='fisheye',
....:             frustum=(-1.0, 1.0, -1.0, 1.0))
sage: T.texture('t1', color=(1,0,0), specular=.9)
sage: T.texture('p1', color=(1,1,1), opacity=.1,
....:           imagefile=fname_ppm, texfunc=9)
sage: T.sphere((0,0,0), .5, 't1')
sage: T.plane((0,0,-1), (0,0,1), 'p1')
sage: T.light((-4,-4,4), .1, (1,1,1))
sage: T.show()
```

AUTHOR:

- John E. Stone (johns@megapixel.com): wrote tachyon ray tracer
- William Stein: sage-tachyon interface
- Joshua Kantor: 3d function plotting
- Tom Boothby: 3d function plotting n'stuff
- Leif Hille: key idea for bugfix for texfunc issue (github issue #799)
- Marshall Hampton: improved doctests, rings, axis-aligned boxes.
- Paul Graham: Respect global verbosity settings (github issue #16228)

Todo:

- clean up trianglefactory stuff

class sage.plot.plot3d.tachyon.**Axis_aligned_box**(*min_p*, *max_p*, *texture*)

Bases: object

Box with axis-aligned edges with the given min and max coordinates.

str()

Return the scene string of the axis-aligned box.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import Axis_aligned_box
sage: aab = Axis_aligned_box((0,0,0),(1,1,1),'s')
sage: aab.str()
'\n        box min  0.0 0.0 0.0  max  1.0 1.0 1.0  s\n        '
```

class sage.plot.plot3d.tachyon.**Cylinder**(*center*, *axis*, *radius*, *texture*)

Bases: object

An infinite cylinder.

str()

Return the scene string of the cylinder.

EXAMPLES:

```
sage: t = Tachyon()
sage: from sage.plot.plot3d.tachyon import Cylinder
sage: c = Cylinder((0,0,0),(1,1,1),.1,'s')
sage: c.str()
'\n        cylinder center  0.0 0.0 0.0  axis  1.0 1.0 1.0  rad 0.1 s\n        '
```

class sage.plot.plot3d.tachyon.**FCylinder**(*base*, *apex*, *radius*, *texture*)

Bases: object

A finite cylinder.

str()

Return the scene string of the finite cylinder.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import FCylinder
sage: fc = FCylinder((0,0,0),(1,1,1),.1,'s')
sage: fc.str()
'\n        fcylinder base  0.0 0.0 0.0  apex  1.0 1.0 1.0  rad 0.1 s\n        '
```

class sage.plot.plot3d.tachyon.**FractalLandscape**(*res*, *scale*, *center*, *texture*)

Bases: object

Axis-aligned fractal landscape.

Does not seem very useful at the moment, but perhaps will be improved in the future.

str()

Return the scene string of the fractal landscape.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import FractalLandscape
sage: fl = FractalLandscape([20,20],[30,30],[1,2,3],'s')
sage: fl.str()
'\n        scape res  20 20  scale  30 30  center  1.0 2.0 3.0  s\n        '
```

class sage.plot.plot3d.tachyon.**Light**(*center*, *radius*, *color*)

Bases: `object`

Represent lighting objects.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import Light
sage: q = Light((1,1,1), 1, (1,1,1))
sage: q._center
(1.0, 1.0, 1.0)
```

str()

Return the tachyon string defining the light source.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import Light
sage: q = Light((1,1,1), 1, (1,1,1))
sage: print(q.str())
        light center  1.0 1.0 1.0
              rad 1.0
              color  1.0 1.0 1.0
```

class sage.plot.plot3d.tachyon.**ParametricPlot**(*f*, *t_0*, *t_f*, *tex*, *r=0.1*, *cylinders=True*, *min_depth=4*, *max_depth=8*, *e_rel=0.01*, *e_abs=0.01*)

Bases: `object`

Parametric plotting routines.

str()

Return the tachyon string representation of the parameterized curve.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import ParametricPlot
sage: f = lambda t: (t,t^2,t^3)
sage: q = ParametricPlot(f,0,1,'s')
sage: q.str()[9:69]
'sphere center  0.0 0.0 0.0  rad 0.1 s\n        \n        fcyli'
```

tol(*est*, *val*)

Check relative, then absolute tolerance.

If both fail, return `False`.

This is a zero-safe error checker.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import ParametricPlot
sage: f = lambda t: (t,t^2,t^3)
sage: q = ParametricPlot(f,0,1,'s')
sage: q.tol([0,0,0],[1,0,0])
False
sage: q.tol([0,0,0],[.0001,0,0])
True
```

class sage.plot.plot3d.tachyon.**Plane**(*center*, *normal*, *texture*)

Bases: `object`

An infinite plane.

str()

Return the scene string of the plane.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import Plane
sage: p = Plane((1,2,3),(1,2,4),'s')
sage: p.str()
'\n        plane center  1.0 2.0 3.0  normal  1.0 2.0 4.0  s\n        '
```

class sage.plot.plot3d.tachyon.**Ring**(*center*, *normal*, *inner*, *outer*, *texture*)

Bases: `object`

An annulus of zero thickness.

str()

Return the scene string of the ring.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import Ring
sage: r = Ring((0,0,0), (1,1,0), 1.0, 2.0, 's')
sage: r.str()
'\n        ring center  0.0 0.0 0.0  normal  1.0 1.0 0.0  inner 1.0 outer 2.0 s\
↪n        '
```

class sage.plot.plot3d.tachyon.**Sphere**(*center*, *radius*, *texture*)

Bases: `object`

A class for creating spheres in tachyon.

str()

Return the scene string for the sphere.

EXAMPLES:

```
sage: t = Tachyon()
sage: from sage.plot.plot3d.tachyon import Sphere
sage: t.texture('r', color=(.8,0,0), ambient = .1)
sage: s = Sphere((1,1,1), 1, 'r')
sage: s.str()
'\n        sphere center  1.0 1.0 1.0  rad 1.0 r\n        '
```

class sage.plot.plot3d.tachyon.**Tachyon**(*xres=350*, *yres=350*, *zoom=1.0*, *antialiasing=False*, *aspectratio=1.0*, *raydepth=8*, *camera_position=None*, *camera_center=None*, *updir=[0, 0, 1]*, *look_at=[0, 0, 0]*, *viewdir=None*, *projection='PERSPECTIVE'*, *focallength=''*, *aperture=''*, *frustum=''*)

Bases: `WithEqualityById`, `SageObject`

Create a scene the can be rendered using the Tachyon ray tracer.

INPUT:

- `xres` – (default 350)
- `yres` – (default 350)
- `zoom` – (default 1.0)
- `antialiasing` – (default `False`)
- `aspectratio` – (default 1.0)
- `raydepth` – (default 8)
- `camera_position` – (default (-3, 0, 0))
- `updir` – (default (0, 0, 1))
- `look_at` – (default (0,0,0))
- `viewdir` – (default `None`), otherwise list of three numbers
- `projection` – `'PERSPECTIVE'` (default), `'perspective_dof'` or `'fisheye'`.
- `frustum` – (default `''`), otherwise list of four numbers. Only used with `projection='fisheye'`.
- `focallength` – (default ‘’), otherwise a number. Only used with `projection='perspective_dof'`.
- `aperture` – (default ‘’), otherwise a number. Only used with `projection='perspective_dof'`.

OUTPUT: A Tachyon 3d scene.

Note that the coordinates are by default such that z is up, positive y is to the {left} and x is toward you. This is not oriented according to the right hand rule.

EXAMPLES: Spheres along the twisted cubic.

```
sage: t = Tachyon(xres=512,yres=512, camera_position=(3,0.3,0))
sage: t.light((4,3,2), 0.2, (1,1,1))
sage: t.texture('t0', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0, color=(1.
↪0,0,0))
sage: t.texture('t1', ambient=0.1, diffuse=0.9, specular=0.3, opacity=1.0, color=(0,
↪1.0,0))
sage: t.texture('t2', ambient=0.2, diffuse=0.7, specular=0.5, opacity=0.7, color=(0,
↪0,1.0))
sage: k=0
sage: for i in srange(-1,1,0.05):
....:     k += 1
....:     t.sphere((i,i^2-0.5,i^3), 0.1, 't%s'%(k%3))
sage: t.show()
```

Another twisted cubic, but with a white background, got by putting infinite planes around the scene.

```
sage: t = Tachyon(xres=512,yres=512, camera_position=(3,0.3,0), raydepth=8)
sage: t.light((4,3,2), 0.2, (1,1,1))
sage: t.texture('t0', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0, color=(1.
↪0,0,0))
sage: t.texture('t1', ambient=0.1, diffuse=0.9, specular=0.3, opacity=1.0, color=(0,
↪1.0,0))
sage: t.texture('t2', ambient=0.2,diffuse=0.7, specular=0.5, opacity=0.7, color=(0,
↪0,1.0))
sage: t.texture('white', color=(1,1,1))
sage: t.plane((0,0,-1), (0,0,1), 'white')
sage: t.plane((0,-20,0), (0,1,0), 'white')
sage: t.plane((-20,0,0), (1,0,0), 'white')
```

```
sage: k=0
sage: for i in srange(-1,1,0.05):
....:     k += 1
....:     t.sphere((i,i^2 - 0.5,i^3), 0.1, 't%s'%(k%3))
....:     t.cylinder((0,0,0), (0,0,1), 0.05,'t1')
sage: t.show()
```

Many random spheres:

```
sage: t = Tachyon(xres=512,yres=512, camera_position=(2,0.5,0.5), look_at=(0.5,0.5,
↪0.5), raydepth=4)
sage: t.light((4,3,2), 0.2, (1,1,1))
sage: t.texture('t0', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0, color=(1.
↪0,0,0))
sage: t.texture('t1', ambient=0.1, diffuse=0.9, specular=0.3, opacity=1.0, color=(0,
↪1.0,0))
sage: t.texture('t2', ambient=0.2, diffuse=0.7, specular=0.5, opacity=0.7, color=(0,
↪0,1.0))
sage: k=0
sage: for i in range(100):
....:     k += 1
....:     t.sphere((random(),random(), random()), random()/10, 't%s'%(k%3))
sage: t.show()
```

Points on an elliptic curve, their height indicated by their height above the axis:

```
sage: # needs sage.schemes
sage: t = Tachyon(camera_position=(5,2,2), look_at=(0,1,0))
sage: t.light((10,3,2), 0.2, (1,1,1))
sage: t.texture('t0', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0, color=(1,
↪0,0))
sage: t.texture('t1', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0, color=(0,
↪1,0))
sage: t.texture('t2', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0, color=(0,
↪0,1))
sage: E = EllipticCurve('37a')                                              #␣
↪needs sage.schemes
sage: P = E([0,0])                                                          #␣
↪needs sage.schemes
sage: Q = P                                                                 #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.schemes
sage: n = 100
sage: for i in range(n):   # increase 20 for a better plot                      #␣
↪needs sage.schemes
....:     Q = Q + P
....:     t.sphere((Q[1], Q[0], ZZ(i)/n), 0.1, 't%s'%(i%3))
sage: t.show()                                                                  #␣
↪needs sage.schemes
```

A beautiful picture of rational points on a rank 1 elliptic curve.

```
sage: # needs sage.schemes
sage: t = Tachyon(xres=1000, yres=800, camera_position=(2,7,4),
....:             look_at=(2,0,0), raydepth=4)
sage: t.light((10,3,2), 1, (1,1,1))
sage: t.light((10,-3,2), 1, (1,1,1))
sage: t.texture('black', color=(0,0,0))
sage: t.texture('red', color=(1,0,0))
sage: t.texture('grey', color=(.9,.9,.9))
sage: t.plane((0,0,0),(0,0,1),'grey')
sage: t.cylinder((0,0,0),(1,0,0),.01,'black')
sage: t.cylinder((0,0,0),(0,1,0),.01,'black')
sage: E = EllipticCurve('37a')                                                  #␣
↪needs sage.schemes
sage: P = E([0,0])                                                              #␣
↪needs sage.schemes
sage: Q = P                                                                     #␣
↪needs sage.schemes
sage: n = 100
sage: for i in range(n):                                                        #␣
↪needs sage.schemes
....:     Q = Q + P
....:     c = i/n + .1
....:     t.texture('r%s'%i,color=(float(i/n),0,0))
....:     t.sphere((Q[0], -Q[1], .01), .04, 'r%s'%i)
sage: t.show()                               # long time                        #␣
↪needs sage.schemes
```

A beautiful spiral.

```
sage: t = Tachyon(xres=800, yres=800, camera_position=(2,5,2), look_at=(2.5,0,0))
sage: t.light((0,0,100), 1, (1,1,1))
sage: t.texture('r', ambient=0.1, diffuse=0.9, specular=0.5,
....:           opacity=1.0, color=(1,0,0))
sage: for i in srange(0,50,0.1):
....:     t.sphere((i/10.0,sin(i),cos(i)), 0.05, 'r')
sage: t.texture('white', color=(1,1,1), opacity=1, specular=1, diffuse=1)
sage: t.plane((0,0,-100), (0,0,-100), 'white')
sage: t.show()
```

If the optional parameter `viewdir` is not set, the camera center should not coincide with the point which is looked at (see github issue #7232):

```
sage: t = Tachyon(xres=80,yres=80, camera_position=(2,5,2), look_at=(2,5,2))
Traceback (most recent call last):
...
ValueError: camera_position and look_at coincide
```

Use of a fisheye lens perspective.

```
sage: T = Tachyon(xres=800, yres=600, camera_position=(-1.5,-1.5,.3),
....:             projection='fisheye', frustum=(-1.0, 1.0, -1.0, 1.0))
sage: T.texture('t1', color=(0,0,1))
sage: cedges = [[[1, 1, 1], [-1, 1, 1]], [[1, 1, 1], [1, -1, 1]],
....:           [[1, 1, 1], [1, 1, -1]], [[-1, 1, 1], [-1, -1, 1]],
....:           [[-1, 1, 1], [-1, 1, -1]], [[1, -1, 1], [-1, -1, 1]],
....:           [[1, -1, 1], [1, -1, -1]], [[-1, -1, 1], [-1, -1, -1]],
....:           [[1, 1, -1], [-1, 1, -1]], [[1, 1, -1], [1, -1, -1]],
....:           [[-1, 1, -1], [-1, -1, -1]], [[1, -1, -1], [-1, -1, -1]]]
sage: for ed in cedges:
....:     T.fcylinder(ed[0], ed[1], .05, 't1')
sage: T.light((-4,-4,4), .1, (1,1,1))
sage: T.show()
```

Use of the `projection='perspective_dof'` option. This may not be implemented correctly.

```
sage: T = Tachyon(xres=800, antialiasing=4, raydepth=10,
....:             projection='perspective_dof', focallength='1.0', aperture='.0025')
sage: T.light((0,5,7), 1.0, (1,1,1))
sage: T.texture('t1', opacity=1, specular=.3)
sage: T.texture('t2', opacity=1, specular=.3, color=(0,0,1))
sage: T.texture('t3', opacity=1, specular=1, color=(1,.8,1), diffuse=0.2)
sage: T.plane((0,0,-1), (0,0,1), 't3')
sage: ttlist = ['t1', 't2']
sage: tt = 't1'
sage: T.cylinder((0,0,.1), (1,1/3,0), .05, 't3')
sage: for q in srange(-3, 100, .15):
....:     if tt == 't1':
....:         tt = 't2'
....:     else:
....:         tt = 't1'
....:     T.sphere((q, q/3+.3*sin(3*q), .1+.3*cos(3*q)), .1, tt)
sage: T.show()
```

axis_aligned_box(*min_p*, *max_p*, *texture*)

Create an axis-aligned box with minimal point `min_p` and maximum point `max_p`.

EXAMPLES:

```
sage: t = Tachyon()
sage: t.axis_aligned_box((0,0,0),(2,2,2),'s')
```

cylinder(*center*, *axis*, *radius*, *texture*)

Create the scene information for a infinite cylinder with the given center, axis direction, radius, and texture.

EXAMPLES:

```
sage: t = Tachyon()
sage: t.texture('c')
sage: t.cylinder((0,0,0),(-1,-1,-1),.1,'c')
```

fcylinder(*base*, *apex*, *radius*, *texture*)

Finite cylinders are almost the same as infinite ones, but the center and length of the axis determine the extents of the cylinder.

The finite cylinder is also really a shell, it does not have any caps. If you need to close off the ends of the cylinder, use two ring objects, with the inner radius set to 0.0 and the normal set to be the axis of the cylinder. Finite cylinders are built this way to enhance speed.

EXAMPLES:

```
sage: t = Tachyon()
sage: t.fcylinder((1,1,1),(1,2,3),.01,'s')
sage: len(t.str())
451
```

fractal_landscape(*res*, *scale*, *center*, *texture*)

Axis-aligned fractal landscape.

Not very useful at the moment.

EXAMPLES:

```
sage: t = Tachyon()
sage: t.texture('s')
sage: t.fractal_landscape([30,30],[80,80],[0,0,0],'s')
sage: len(t._objects)
2
```

light(*center*, *radius*, *color*)

Create a light source of the given center, radius, and color.

EXAMPLES:

```
sage: q = Tachyon()
sage: q.light((1,1,1),1.0,(.2,0,.8))
sage: q.str().split('\n')[17]
'        light center  1.0 1.0 1.0 '
```

parametric_plot(*f*, *t_0*, *t_f*, *tex*, *r=0.1*, *cylinders=True*, *min_depth=4*, *max_depth=8*, *e_rel=0.01*, *e_abs=0.01*)

Plot a space curve as a series of spheres and finite cylinders.

Example (twisted cubic)

```
sage: f = lambda t: (t,t^2,t^3)
sage: t = Tachyon(camera_position=(5,0,4))
sage: t.texture('t')
sage: t.light((-20,-20,40), 0.2, (1,1,1))
sage: t.parametric_plot(f,-5,5,'t',min_depth=6)
sage: t.show(verbose=1)
tachyon ...
```

(continues on next page)

(continued from previous page)

```
Scene contains 482 objects.
...
```

plane(*center*, *normal*, *texture*)

Create an infinite plane with the given center and normal.

plot(*f*, *xmin_xmax*, *ymin_ymax*, *texture*, *grad_f=None*, *max_bend=0.7*, *max_depth=5*, *initial_depth=3*, *num_colors=None*)

INPUT:

- `f` - Function of two variables, which returns a float (or coercible to a float) (xmin,xmax)
- `(ymin,ymax)` - defines the rectangle to plot over texture: Name of texture to be used Optional arguments:
- `grad_f` - gradient function. If specified, smooth triangles will be used.
- `max_bend` - Cosine of the threshold angle between triangles used to determine whether or not to recurse after the minimum depth
- `max_depth` - maximum recursion depth. Maximum triangles plotted = 2^{2*max_depth}
- `initial_depth` - minimum recursion depth. No error-tolerance checking is performed below this depth. Minimum triangles plotted: 2^{2*min_depth}
- `num_colors` - Number of rainbow bands to color the plot with. Texture supplied will be cloned (with different colors) using the texture_recolor method of the Tachyon object.

Plots a function by constructing a mesh with nonstandard sampling density without gaps. At very high resolutions (depths 10) it becomes very slow. Cython may help. Complexity is approx. $O(2^{2*maxdepth})$. This algorithm has been optimized for speed, not memory - values from f(x,y) are recycled rather than calling the function multiple times. At high recursion depth, this may cause problems for some machines.

Flat Triangles:

```
sage: t = Tachyon(xres=512, yres=512, camera_position=(4,-4,3),
....:             viewdir=(-4,4,-3), raydepth=4)
sage: t.light((4.4,-4.4,4.4), 0.2, (1,1,1))
sage: def f(x,y): return float(sin(x*y))
sage: t.texture('t0', ambient=0.1, diffuse=0.9, specular=0.1,
....:           opacity=1.0, color=(1.0,0,0))
sage: t.plot(f, (-4,4), (-4,4), "t0", max_depth=5, initial_depth=3,         #␣
↪needs sage.symbolic
....:         num_colors=60)  # increase min_depth for better picture
sage: t.show(verbose=1)                                                       #␣
↪needs sage.symbolic
tachyon ...
Scene contains 2713 objects.
...
```

Plotting with Smooth Triangles (requires explicit gradient function):

```
sage: t = Tachyon(xres=512, yres=512, camera_position=(4,-4,3),
....:             viewdir=(-4,4,-3), raydepth=4)
sage: t.light((4.4,-4.4,4.4), 0.2, (1,1,1))
sage: def f(x,y): return float(sin(x*y))
sage: def g(x,y): return (float(y*cos(x*y)), float(x*cos(x*y)), 1)
```

(continues on next page)

(continued from previous page)

```
sage: t.texture('t0', ambient=0.1, diffuse=0.9, specular=0.1,
....:           opacity=1.0, color=(1.0,0,0))
sage: t.plot(f, (-4,4), (-4,4), "t0", max_depth=5, initial_depth=3,        #␣
↪needs sage.symbolic
....:          grad_f=g)  # increase min_depth for better picture
sage: t.show(verbose=1)                                                     #␣
↪needs sage.symbolic
tachyon ...
Scene contains 2713 objects.
...
```

Preconditions: f is a scalar function of two variables, grad_f is None or a triple-valued function of two variables, min_x != max_x, min_y != max_y

```
sage: f = lambda x,y: x*y
sage: t = Tachyon()
sage: t.plot(f,(2.,2.),(-2.,2.),'')
Traceback (most recent call last):
...
ValueError: plot rectangle is really a line; make sure min_x != max_x and min_y␣
↪!= max_y
```

ring(*center*, *normal*, *inner*, *outer*, *texture*)

Create the scene information for a ring with the given parameters.

EXAMPLES:

```
sage: t = Tachyon()
sage: t.ring([0,0,0], [0,0,1], 1.0, 2.0, 's')
sage: t._objects[0]._center
(0.0, 0.0, 0.0)
```

save(*filename='sage.png'*, *verbose=None*, *extra_opts=''*)

Save rendering of the tachyon scene

INPUT:

- `filename` - (default: 'sage.png') output filename; the extension of the filename determines the type. Supported types include:
- `tga` - 24-bit (uncompressed)
- `bmp` - 24-bit Windows BMP (uncompressed)
- `ppm` - 24-bit PPM (uncompressed)
- `rgb` - 24-bit SGI RGB (uncompressed)
- `png` - 24-bit PNG (compressed, lossless)
- `verbose` - integer (default: `None`); if no verbosity setting is supplied, the verbosity level set by `sage.misc.verbose.set_verbose` is used.
- `0` - silent
- `1` - some output
- `2` - very verbose output

- `extra_opts` - passed directly to tachyon command line. Use tachyon_rt.usage() to see some of the possibilities.

EXAMPLES:

```
sage: q = Tachyon()
sage: q.light((1,1,11), 1,(1,1,1))
sage: q.texture('s')
sage: q.sphere((0,0,0),1,'s')
sage: tempname = tmp_filename()
sage: q.save(tempname)
```

save_image(*filename=None*, **args*, ***kwds*)

Save an image representation of `self`.

The image type is determined by the extension of the filename. For example, this could be `.png`, `.jpg`, `.gif`, `.pdf`, `.svg`. Currently this is implemented by calling the *save()* method of self, passing along all arguments and keywords.

Note: Not all image types are necessarily implemented for all graphics types. See *save()* for more details.

EXAMPLES:

```
sage: q = Tachyon()
sage: q.light((1,1,11), 1,(1,1,1))
sage: q.texture('s')
sage: q.sphere((0,-1,1),1,'s')
sage: tempname = tmp_filename()
sage: q.save_image(tempname)
```

show(***kwds*)

Create a PNG file of the scene.

This method attempts to display the graphics immediately, without waiting for the currently running code (if any) to return to the command line. Be careful, calling it from within a loop will potentially launch a large number of external viewer programs.

OUTPUT:

This method does not return anything. Use *save()* if you want to save the figure as an image.

EXAMPLES:

This example demonstrates how the global Sage verbosity setting is used if none is supplied. Firstly, using a global verbosity setting of 0 means no extra technical information is displayed, and we are simply shown the plot.

```
sage: h = Tachyon(xres=512, yres=512, camera_position=(4,-4,3),
....:             viewdir=(-4,4,-3), raydepth=4)
sage: h.light((4.4,-4.4,4.4), 0.2, (1,1,1))
sage: def f(x,y): return float(sin(x*y))
sage: h.texture('t0', ambient=0.1, diffuse=0.9, specular=0.1,
....:           opacity=1.0, color=(1.0,0,0))
sage: h.plot(f, (-4,4), (-4,4), "t0", max_depth=5, initial_depth=3,         #␣
↪needs sage.symbolic
....:        num_colors=60)  # increase min_depth for better picture
```

(continues on next page)

(continued from previous page)

```
sage: from sage.misc.verbose import set_verbose, get_verbose
sage: set_verbose(0)
sage: h.show()                                                          #␣
↪needs sage.symbolic
```

This second example, using a “medium” global verbosity setting of 1, displays some extra technical information then displays our graph.

```
sage: s = Tachyon(xres=512, yres=512, camera_position=(4,-4,3),
....:             viewdir=(-4,4,-3), raydepth=4)
sage: s.light((4.4,-4.4,4.4), 0.2, (1,1,1))
sage: def f(x,y): return float(sin(x*y))
sage: s.texture('t0', ambient=0.1, diffuse=0.9, specular=0.1,
....:           opacity=1.0, color=(1.0,0,0))
sage: s.plot(f, (-4,4), (-4,4), "t0", max_depth=5, initial_depth=3,     #␣
↪needs sage.symbolic
....:        num_colors=60)  # increase min_depth for better picture
sage: set_verbose(1)
sage: s.show()                                                          #␣
↪needs sage.symbolic
tachyon ...
Scene contains 2713 objects.
...
```

The last example shows how you can override the global Sage verbosity setting, my supplying a setting level as an argument. In this case we chose the highest verbosity setting level, 2, so much more extra technical information is shown, along with the plot.

```
sage: set_verbose(0)
sage: d = Tachyon(xres=512, yres=512, camera_position=(4,-4,3),
....:             viewdir=(-4,4,-3), raydepth=4)
sage: d.light((4.4,-4.4,4.4), 0.2, (1,1,1))
sage: def f(x,y): return float(sin(x*y))
sage: d.texture('t0', ambient=0.1, diffuse=0.9, specular=0.1,
....:           opacity=1.0, color=(1.0,0,0))
sage: d.plot(f,(-4,4),(-4,4),"t0",max_depth=5,initial_depth=3,          #␣
↪needs sage.symbolic
....:        num_colors=60)  # increase min_depth for better picture
sage: get_verbose()
0
sage: d.show(verbose=2)                                                 #␣
↪needs sage.symbolic
tachyon ...
Scene contains 2713 objects.
...
Scene contains 1 non-gridded objects
...
```

smooth_triangle(*vertex_1*, *vertex_2*, *vertex_3*, *normal_1*, *normal_2*, *normal_3*, *texture*)

Create a triangle along with a normal vector for smoothing.

EXAMPLES:

```
sage: t = Tachyon()
sage: t.light((1,1,1),.1,(1,1,1))
sage: t.texture('s')
sage: t.smooth_triangle([0,0,0],[0,0,1],[0,1,0],[0,1,1],[-1,1,2],[3,0,0],'s')
sage: t._objects[2].get_vertices()
([0, 0, 0], [0, 0, 1], [0, 1, 0])
sage: t._objects[2].get_normals()
([0, 1, 1], [-1, 1, 2], [3, 0, 0])
```

sphere(*center*, *radius*, *texture*)

Create the scene information for a sphere with the given center, radius, and texture.

EXAMPLES:

```
sage: t = Tachyon()
sage: t.texture('sphere_texture')
sage: t.sphere((1,2,3), .1, 'sphere_texture')
sage: t._objects[1].str()
'\n        sphere center  1.0 2.0 3.0  rad 0.1 sphere_texture\n        '
```

str()

Return the complete tachyon scene file as a string.

EXAMPLES:

```
sage: t = Tachyon(xres=500,yres=500, camera_position=(2,0,0))
sage: t.light((4,3,2), 0.2, (1,1,1))
sage: t.texture('t2', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0,␣
↪color=(1,0,0))
sage: t.texture('t3', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0,␣
↪color=(0,1,0))
sage: t.texture('t4', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0,␣
↪color=(0,0,1))
sage: t.sphere((0,0.5,0), 0.2, 't2')
sage: t.sphere((0.5,0,0), 0.2, 't3')
sage: t.sphere((0,0,0.5), 0.2, 't4')
sage: 'PLASTIC' in t.str()
True
```

texfunc(*type=0*, *center=(0, 0, 0)*, *rotate=(0, 0, 0)*, *scale=(1, 1, 1)*, *imagefile=''*)

INPUT:

- `type` - (default: 0)

  0. No special texture, plain shading
  1. 3D checkerboard function, like a rubik's cube
  2. Grit Texture, randomized surface color
  3. 3D marble texture, uses object's base color
  4. 3D wood texture, light and dark brown, not very good yet
  5. 3D gradient noise function (can't remember what it looks like)
  6. Don't remember
  7. Cylindrical Image Map, requires ppm filename (with path)

8. Spherical Image Map, requires ppm filename (with path)
9. Planar Image Map, requires ppm filename (with path)

- `center` - (default: (0,0,0))
- `rotate` - (default: (0,0,0))
- `scale` - (default: (1,1,1))

EXAMPLES: We draw an infinite checkerboard:

```
sage: t = Tachyon(camera_position=(2,7,4), look_at=(2,0,0))
sage: t.texture('black', color=(0,0,0), texfunc=1)
sage: t.plane((0,0,0),(0,0,1),'black')
sage: t.show()
```

texture(*name*, *ambient=0.2*, *diffuse=0.8*, *specular=0.0*, *opacity=1.0*, *color=(1.0, 0.0, 0.5)*, *texfunc=0*, *phong=0*, *phongsize=0.5*, *phongtype='PLASTIC'*, *imagefile=''*)

INPUT:

- `name` - string; the name of the texture (to be used later)
- `ambient` - (default: 0.2)
- `diffuse` - (default: 0.8)
- `specular` - (default: 0.0)
- `opacity` - (default: 1.0)
- `color` - (default: (1.0,0.0,0.5))
- `texfunc` - (default: 0); a texture function; this is either the output of self.texfunc, or a number between 0 and 9, inclusive. See the docs for self.texfunc.
- `phong` - (default: 0)
- `phongsize` - (default: 0.5)
- `phongtype` - (default: "PLASTIC")

EXAMPLES:

We draw a scene with 4 spheres that illustrates various uses of the texture command:

```
sage: t = Tachyon(camera_position=(2,5,4), look_at=(2,0,0), raydepth=6)
sage: t.light((10,3,4), 1, (1,1,1))
sage: t.texture('mirror', ambient=0.05, diffuse=0.05, specular=.9,
....:           opacity=0.9, color=(.8,.8,.8))
sage: t.texture('grey', color=(.8,.8,.8), texfunc=3)
sage: t.plane((0,0,0),(0,0,1),'grey')
sage: t.sphere((4,-1,1), 1, 'mirror')
sage: t.sphere((0,-1,1), 1, 'mirror')
sage: t.sphere((2,-1,1), 0.5, 'mirror')
sage: t.sphere((2,1,1), 0.5, 'mirror')
sage: show(t)  # known bug (trac #7232)
```

texture_recolor(*name*, *colors*)

Recolor default textures.

EXAMPLES:

```
sage: t = Tachyon()
sage: t.texture('s')
sage: q = t.texture_recolor('s',[(0,0,1)])
sage: t._objects[1]._color
(0.0, 0.0, 1.0)
```

triangle(*vertex_1*, *vertex_2*, *vertex_3*, *texture*)

Create a triangle with the given vertices and texture.

EXAMPLES:

```
sage: t = Tachyon()
sage: t.texture('s')
sage: t.triangle([1,2,3],[4,5,6],[7,8,10],'s')
sage: t._objects[1].get_vertices()
([1, 2, 3], [4, 5, 6], [7, 8, 10])
```

class sage.plot.plot3d.tachyon.**TachyonSmoothTriangle**(*a*, *b*, *c*, *da*, *db*, *dc*, *color=0*)

Bases: *SmoothTriangle*

A triangle along with a normal vector, which is used for smoothing.

str()

Return the scene string for a smoothed triangle.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import TachyonSmoothTriangle
sage: t = TachyonSmoothTriangle([-1,-1,-1],[0,0,0],[1,2,3],[1,0,0],[0,1,0],[0,0,
↪1])
sage: t.str()
'\n        STRI V0  ...  1.0 0.0 0.0  N1  0.0 1.0 0.0   N2  0.0 0.0 1.0 \n     ␣
↪       0\n        '
```

class sage.plot.plot3d.tachyon.**TachyonTriangle**(*a*, *b*, *c*, *color=0*)

Bases: *Triangle*

Basic triangle class.

str()

Return the scene string for a triangle.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import TachyonTriangle
sage: t = TachyonTriangle([-1,-1,-1],[0,0,0],[1,2,3])
sage: t.str()
'\n        TRI V0  -1.0 -1.0 -1.0   V1  0.0 0.0 0.0    V2  1.0 2.0 3.0 \n     ␣
↪     0\n        '
```

class sage.plot.plot3d.tachyon.**TachyonTriangleFactory**(*tach*, *tex*)

Bases: *TriangleFactory*

A class to produce triangles of various rendering types.

get_colors(*list*)

Return a list of color labels.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import TachyonTriangleFactory
sage: t = Tachyon()
sage: t.texture('s')
sage: ttf = TachyonTriangleFactory(t, 's')
sage: ttf.get_colors([(1,1,1)])
['SAGETEX1_0']
```

smooth_triangle(*a, b, c, da, db, dc, color=None*)

Create a TachyonSmoothTriangle.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import TachyonTriangleFactory
sage: t = Tachyon()
sage: t.texture('s')
sage: ttf = TachyonTriangleFactory(t, 's')
sage: ttfst = ttf.smooth_triangle([0,0,0],[1,0,0],[0,0,1],[1,1,1],[1,2,3],[-1,-
↪1,2])
sage: ttfst.str()
'\n        STRI V0  0.0 0.0 0.0  ...'
```

triangle(*a, b, c, color=None*)

Create a TachyonTriangle with vertices a, b, and c.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import TachyonTriangleFactory
sage: t = Tachyon()
sage: t.texture('s')
sage: ttf = TachyonTriangleFactory(t, 's')
sage: ttft = ttf.triangle([1,2,3],[3,2,1],[0,2,1])
sage: ttft.str()
'\n        TRI V0  1.0 2.0 3.0   V1  3.0 2.0 1.0    V2  0.0 2.0 1.0 \n          ␣
↪  s\n        '
```

class sage.plot.plot3d.tachyon.**Texfunc**(*ttype=0, center=(0, 0, 0), rotate=(0, 0, 0), scale=(1, 1, 1), imagefile=''*)

Bases: object

Create a texture function.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import Texfunc
sage: t = Texfunc()
sage: t._ttype
0
```

str()

Return the scene string for this texture function.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import Texfunc
sage: t = Texfunc()
sage: t.str()
'0'
```

class sage.plot.plot3d.tachyon.**Texture**(*name*, *ambient=0.2*, *diffuse=0.8*, *specular=0.0*, *opacity=1.0*, *color=(1.0, 0.0, 0.5)*, *texfunc=0*, *phong=0*, *phongsize=0*, *phongtype='PLASTIC'*, *imagefile=''*)

Bases: object

Store texture information.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import Texture
sage: t = Texture('w')
sage: t.str().split()[2:6]
['ambient', '0.2', 'diffuse', '0.8']
```

recolor(*name*, *color*)

Return a texture with the new given color.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import Texture
sage: t2 = Texture('w')
sage: t2w = t2.recolor('w2', (.1,.2,.3))
sage: t2ws = t2w.str()
sage: color_index = t2ws.find('color')
sage: t2ws[color_index:color_index+20]
'color  0.1 0.2 0.3  '
```

str()

Return the scene string for this texture.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import Texture
sage: t = Texture('w')
sage: t.str().split()[2:6]
['ambient', '0.2', 'diffuse', '0.8']
```

sage.plot.plot3d.tachyon.**tostr**(*s*, *length=3*, *out_type=<class 'float'>*)

Convert vector information to a space-separated string.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import tostr
sage: tostr((1,1,1))
' 1.0 1.0 1.0 '
sage: tostr('2 3 2')
'2 3 2'
```

5.2 Three.js JavaScript WebGL Renderer

A web-based interactive viewer using the Three.js JavaScript library maintained by https://threejs.org.

The viewer is invoked by adding the keyword argument `viewer='threejs'` to the command `show()` or any three-dimensional graphic. The scene is rendered and displayed in the users's web browser. Interactivity includes

- Zooming in or out with the mouse wheel or pinching on a touch pad
- Rotation by clicking and dragging with the mouse or swiping on a touch pad
- Panning by right-clicking and dragging with the mouse or swiping with three fingers on a touch pad

The generated HTML file contains all data for the scene apart from the JavaScript library and can be saved to disk for sharing or embedding in a web page. The option `online` can be set to `true` to provide links to the required files in an online content delivery network. Alternately the required files can be downloaded from the Three.js GitHub repository and linked directly from the web server.

Options currently supported by the viewer:

- `aspect_ratio` – (default: [1,1,1]) list or tuple of three numeric values
- `auto_scaling` – (default: [False, False, False]) list or tuple of three booleans; set to True to automatically scale down the corresponding direction if it is too large
- `axes` – (default: False) Boolean determining whether coordinate axes are drawn
- `axes_labels` – (default: ['x','y','z']) list or tuple of three strings; set to False to remove all labels
- `axes_labels_style` – (default: None) list of three dicts, one per axis, or a single dict controlling all three axes; supports the same styling options as *`text3d()`* such as `color`, `opacity`, `fontsize`, `fontweight`, `fontstyle`, and `fontfamily`
- `color` – (default: 'blue') color of the 3D object
- `decimals` – (default: 2) integer determining decimals displayed in labels
- `depth_write` – (default: True for opaque surfaces, False for transparent surfaces) whether to write the surface's depth into the depth buffer for the purpose of occluding objects behind it
- `frame` – (default: True) Boolean determining whether frame is drawn
- `online` – (default: False) Boolean determining whether the local standard package files are replaced by links to an online content delivery network
- `opacity` – (default: 1) numeric value for transparency of lines and surfaces
- `page_title` – (default: None) string containing the title of the generated HTML page; often displayed in the browser window's title bar, its tab list, and/or the operating system's task bar
- `projection` – (default: 'perspective') the type of camera projection to use; 'perspective' or 'orthographic'
- `radius` – (default: None) numeric value for radius of lines; use to render lines thicker than available using `thickness` or on Windows platforms where `thickness` is ignored
- `render_order` – (default: 0) numeric value for rendering order of transparent surfaces; objects render from lowest to highest value ensuring that lower-valued objects render completely
- `single_side` – (default: False) Boolean determining whether both sides of a surface material are rendered; set to True to reduce rendering artifacts for closed transparent surfaces
- `theme` – (default: 'light') the color scheme to use for the scene and user interface; 'light' or 'dark'
- `thickness` – (default: 1) numeric value for thickness of lines

- `viewpoint` – (default: None) list or tuple of the form [[x,y,z],angle] setting the initial viewpoint of the scene, where angle is in degrees; can be determined using the 'Get Viewpoint' option of the information menu

In addition, the following animation-related options are supported:

- `animate` – (default: depends) whether to enable animation. Automatically set to `True` if animation data is present in the plot. If `False`, all frames of animation will be displayed simultaneously.
- `animation_controls` – (default: True) whether to include the playback slider and buttons (play, pause, etc.) in the page
- `auto_play` – (default: True) whether to immediately start playing the animation when the page loads. Recommend setting `animation_controls=True` to be able to start playback.
- `delay` – (default: 20) an integer amount of time between consecutive frames of animation, in hundredths of a second
- `loop` – (default: True) whether to loop the animation or have it stop after reaching the end. Can be toggled on the page itself if `animation_controls` is set.

Clicking on the information icon in the lower right-hand corner of the viewer opens a menu of available actions. These include saving the three-dimensional scene as a static PNG image or as complete HTML source code.

AUTHORS:

- Paul Masson (2016): Initial version
- Joshua Campbell (2020): Animation support

EXAMPLES:

Three spheres of different color and opacity:

```
sage: p1 = sphere(color='red', opacity='.5')
sage: p2 = sphere((-1,-1,1), color='cyan', opacity='.3')
sage: p3 = sphere((1,-1,-1), color='yellow', opacity='.7')
sage: show(p1 + p2 + p3, viewer='threejs')
```

A parametric helix:

```
sage: parametric_plot3d([cos(x),sin(x),x/10], (x,0,4*pi), color='red', viewer='threejs')
Graphics3d Object
```

An `interactive()` animation:

```
sage: def build_frame(t):
....:     e = parametric_plot3d([sin(x-t), 0, x], (x, 0, 2*pi), color='red')
....:     m = parametric_plot3d([0, -sin(x-t), x], (x, 0, 2*pi), color='green')
....:     return e + m
sage: frames = [build_frame(t) for t in (0, pi/32, pi/16, .., 2*pi)]
sage: plot = animate(frames).interactive()
sage: show(plot, delay=5, auto_play=False, projection='orthographic')
```

CHAPTER

SIX

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

INDEX