Three-Dimensional Plotting in Matplotlib

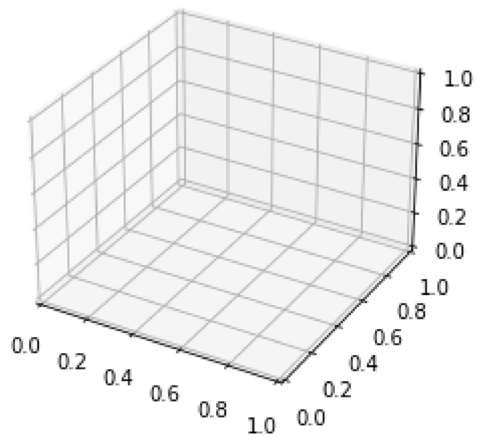
Matplotlib was initially designed with only two-dimensional plotting in mind. Around the time of the 1.0 release, some three-dimensional plotting utilities were built on top of Matplotlib's two-dimensional display, and the result is a convenient (if somewhat limited) set of tools for three-dimensional data visualization. Three-dimensional plots are enabled by importing the `mplot3d` toolkit, included with the main Matplotlib installation:

```
In [1]: from mpl_toolkits import mplot3d
```

Once this submodule is imported, a three-dimensional axes can be created by passing the keyword `projection='3d'` to any of the normal axes creation routines, as shown here (see the following figure):

```
In [2]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
```

```
In [3]: fig = plt.figure()
        ax = plt.axes(projection='3d')
```

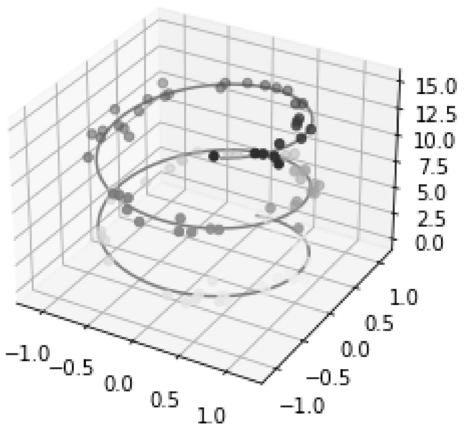


With this three-dimensional axes enabled, we can now plot a variety of three-dimensional plot types. Three-dimensional plotting is one of the functionalities that benefits immensely from viewing figures interactively rather than statically, in the notebook; recall that to use interactive figures, you can use `%matplotlib notebook` rather than `%matplotlib inline` when running this code.

Three-Dimensional Points and Lines

The most basic three-dimensional plot is a line or collection of scatter plots created from sets of (x, y, z) triples. In analogy with the more common two-dimensional plots discussed earlier, these can be created using the `ax.plot3D` and `ax.scatter3D` functions. The call signature for these is nearly identical to that of their two-dimensional counterparts, so you can refer to Simple Line Plots (04.01-Simple-Line-Plots.ipynb) and Simple Scatter Plots (04.02-Simple-Scatter-Plots.ipynb) for more information on controlling the output. Here we'll plot a trigonometric spiral, along with some points drawn randomly near the line (see the following figure):

In [4]:
```
ax = plt.axes(projection='3d')

# Data for a three-dimensional line
zline = np.linspace(0, 15, 1000)
xline = np.sin(zline)
yline = np.cos(zline)
ax.plot3D(xline, yline, zline, 'gray')

# Data for three-dimensional scattered points
zdata = 15 * np.random.random(100)
xdata = np.sin(zdata) + 0.1 * np.random.randn(100)
ydata = np.cos(zdata) + 0.1 * np.random.randn(100)
ax.scatter3D(xdata, ydata, zdata, c=zdata, cmap='Greens');
```



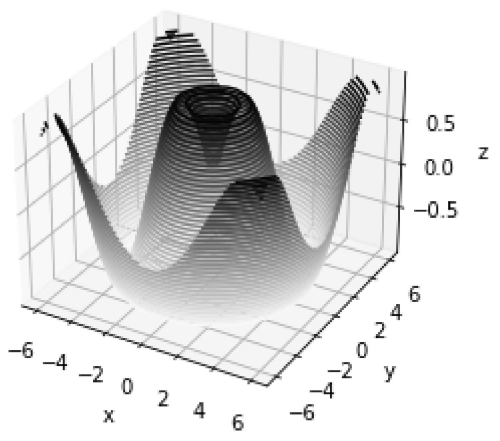
Notice that scatter points have their transparency adjusted to give a sense of depth on the page. While the three-dimensional effect is sometimes difficult to see within a static image, an interactive view can lead to some nice intuition about the layout of the points.

Three-Dimensional Contour Plots

Analogous to the contour plots we explored in Density and Contour Plots (04.04-Density-and-Contour-Plots.ipynb), `mplot3d` contains tools to create three-dimensional relief plots using the same inputs. Like `ax.contour`, `ax.contour3D` requires all the input data to be in the form of two-dimensional regular grids, with the *z* data evaluated at each point. Here we'll show a three-

```
In [5]: def f(x, y):
            return np.sin(np.sqrt(x ** 2 + y ** 2))

        x = np.linspace(-6, 6, 30)
        y = np.linspace(-6, 6, 30)

        X, Y = np.meshgrid(x, y)
        Z = f(X, Y)
```

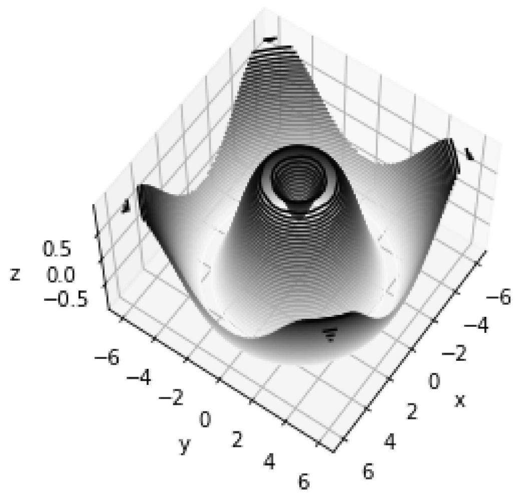
```
In [6]: fig = plt.figure()
        ax = plt.axes(projection='3d')
        ax.contour3D(X, Y, Z, 40, cmap='binary')
        ax.set_xlabel('x')
        ax.set_ylabel('y')
        ax.set_zlabel('z');
```



Sometimes the default viewing angle is not optimal, in which case we can use the `view_init` method to set the elevation and azimuthal angles. In the following example, visualized in the following figure, we'll use an elevation of 60 degrees (that is, 60 degrees above the x-y plane) and an azimuth of 35 degrees (that is, rotated 35 degrees counter-clockwise about the z-axis):

In [7]:
```
ax.view_init(60, 35)
fig
```

Out[7]:

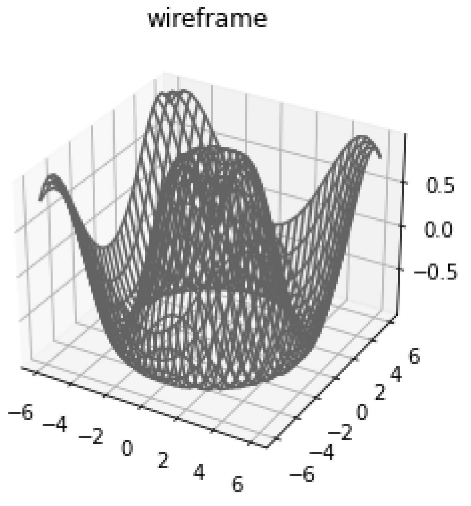


Again, note that this type of rotation can be accomplished interactively by clicking and dragging when using one of Matplotlib's interactive backends.

Wireframes and Surface Plots

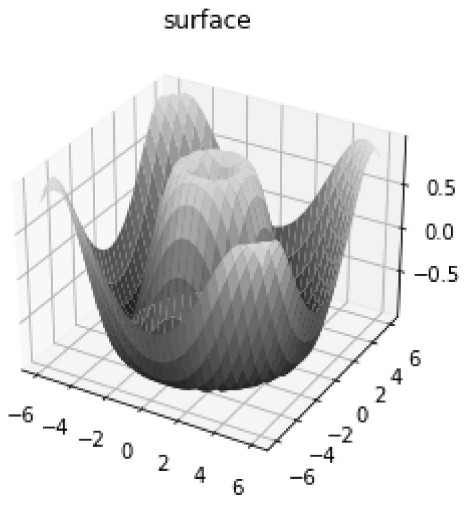
Two other types of three-dimensional plots that work on gridded data are wireframes and surface plots. These take a grid of values and project it onto the specified three-dimensional surface, and can make the resulting three-dimensional forms quite easy to visualize. Here's an example of using a wireframe (see the following figure):

In [8]:
```
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.plot_wireframe(X, Y, Z)
ax.set_title('wireframe');
```



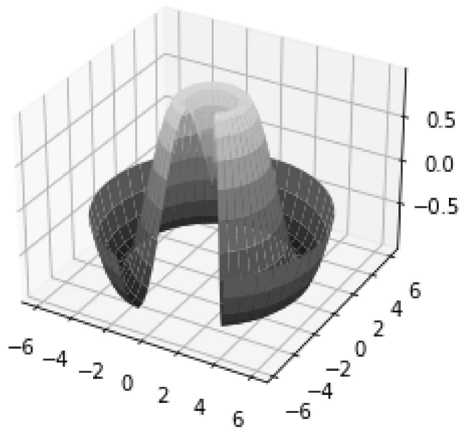
A surface plot is like a wireframe plot, but each face of the wireframe is a filled polygon. Adding a colormap to the filled polygons can aid perception of the topology of the surface being visualized, as you can see in the following figure:

```
In [9]: ax = plt.axes(projection='3d')
        ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                        cmap='viridis', edgecolor='none')
        ax.set_title('surface');
```



Though the grid of values for a surface plot needs to be two-dimensional, it need not be rectilinear. Here is an example of creating a partial polar grid, which when used with the `surface3D` plot can give us a slice into the function we're visualizing (see the following figure):

```
In [10]: r = np.linspace(0, 6, 20)
         theta = np.linspace(-0.9 * np.pi, 0.8 * np.pi, 40)
         r, theta = np.meshgrid(r, theta)

         X = r * np.sin(theta)
         Y = r * np.cos(theta)
         Z = f(X, Y)

         ax = plt.axes(projection='3d')
         ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                         cmap='viridis', edgecolor='none');
```



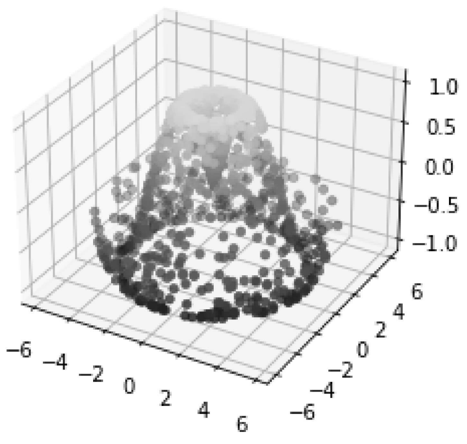
Surface Triangulations

For some applications, the evenly sampled grids required by the preceding routines are too restrictive. In these situations, triangulation-based plots can come in handy. What if rather than an even draw from a Cartesian or a polar grid, we instead have a set of random draws?

```
In [11]: theta = 2 * np.pi * np.random.random(1000)
         r = 6 * np.random.random(1000)
         x = np.ravel(r * np.sin(theta))
         y = np.ravel(r * np.cos(theta))
         z = f(x, y)
```

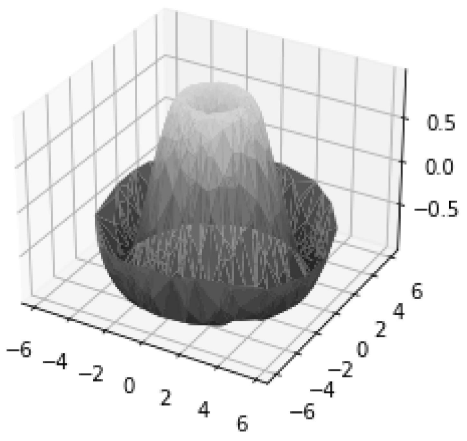
We could create a scatter plot of the points to get an idea of the surface we're sampling from, as shown in the following figure:

In [12]:
```
ax = plt.axes(projection='3d')
ax.scatter(x, y, z, c=z, cmap='viridis', linewidth=0.5);
```



This point cloud leaves a lot to be desired. The function that will help us in this case is `ax.plot_trisurf`, which creates a surface by first finding a set of triangles formed between adjacent points (remember that `x`, `y`, and `z` here are one-dimensional arrays); the following figure shows the result:

In [13]:
```
ax = plt.axes(projection='3d')
ax.plot_trisurf(x, y, z,
                cmap='viridis', edgecolor='none');
```



The result is certainly not as clean as when it is plotted with a grid, but the flexibility of such a triangulation allows for some really interesting three-dimensional plots. For example, it is actually possible to plot a three-dimensional Möbius strip using this, as we'll see next.

Example: Visualizing a Möbius Strip

A Möbius strip is similar to a strip of paper glued into a loop with a half-twist, resulting in an object with only a single side! Here we will visualize such an object using Matplotlib's three-dimensional tools. The key to creating the Möbius strip is to think about its parametrization: it's a two-dimensional strip, so we need two intrinsic dimensions. Let's call them θ, which ranges from 0 to 2π around the loop, and w, which ranges from –1 to 1 across the width of the strip:

```
In [14]: theta = np.linspace(0, 2 * np.pi, 30)
         w = np.linspace(-0.25, 0.25, 8)
         w, theta = np.meshgrid(w, theta)
```

Now from this parametrization, we must determine the (*x*, *y*, *z*) positions of the embedded strip.

Thinking about it, we might realize that there are two rotations happening: one is the position of the loop about its center (what we've called θ), while the other is the twisting of the strip about its axis (we'll call this ϕ). For a Möbius strip, we must have the strip make half a twist during a full loop, or $\Delta\phi = \Delta\theta/2$:

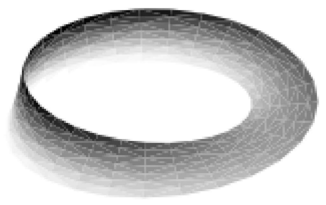
```
In [15]: phi = 0.5 * theta
```

Now we use our recollection of trigonometry to derive the three-dimensional embedding. We'll define r, the distance of each point from the center, and use this to find the embedded (x, y, z) coordinates:

```
In [16]: # radius in x-y plane
         r = 1 + w * np.cos(phi)

         x = np.ravel(r * np.cos(theta))
         y = np.ravel(r * np.sin(theta))
         z = np.ravel(w * np.sin(phi))
```

Finally, to plot the object, we must make sure the triangulation is correct. The best way to do this is to define the triangulation *within the underlying parametrization*, and then let Matplotlib project this triangulation into the three-dimensional space of the Möbius strip. This can be accomplished as follows (see the following figure):

In [17]:
```
# triangulate in the underlying parametrization
from matplotlib.tri import Triangulation
tri = Triangulation(np.ravel(w), np.ravel(theta))

ax = plt.axes(projection='3d')
ax.plot_trisurf(x, y, z, triangles=tri.triangles,
                cmap='Greys', linewidths=0.2);

ax.set_xlim(-1, 1); ax.set_ylim(-1, 1); ax.set_zlim(-1, 1)
ax.axis('off');
```



Combining all of these techniques, it is possible to create and display a wide variety of three-dimensional objects and patterns in Matplotlib.