6 Three-dimensional graphics

6.1 Introduction

6.1.1 Three-dimensional data sets

In the previous chapter, we saw that the Python *matplotlib* module is excellent for producing graphs of a single function, $y = f(x)$. Actually, it can do a little more than this. Suppose u is an independent variable, and a number of values of u, usually uniformly spaced, are given. Suppose we define a parametrized curve $x = x(u)$, $y = y(u)$. The *matplotlib* `plt.plot` function can easily draw such curves. As a concrete simple example, suppose θ is defined on $[0, 2\pi]$. Then $x = \cos\theta$, $y = \sin\theta$ defines a familiar curve, the unit circle. As a more intricate example consider *Lissajous' figures*, of which an example is $x = \cos(3\theta)$, $y = \sin(5\theta)$. A code snippet to draw an unadorned version of this is

```
import numpy as np
import matplotlib.pyplot as plt

theta=np.linspace(0,2*np.pi,401)
x=np.cos(3*theta)
y=np.sin(5*theta)
plt.ion()
plt.plot(x,y,lw=2)
```

Also *matplotlib* is good at producing contour curves. More specifically, suppose z is a function of two variables, $z = Z(x, y)$. Then, at least locally and subject to certain conditions, and possibly interchanging x and y, we can invert the relation as $y = Y(x, z)$. If we now fix the value of z, say $z = z_0$, then we obtain a family of curves $y = Y(x, z_0)$ parametrized by z_0. These are the contour curves that *matplotlib* can plot with the `plt.contour` function.

Now suppose we want to extend these capabilities to three dimensions. We shall consider three cases, with a concrete example for each. Our examples are artificial in the sense that they are predefined analytically. However, in setting them up we have to construct finite sets of discrete data. In the real world, we would instead use our own finite sets of discrete data, derived either from experiment or a complicated numerical simulation.

https://doi.org/10.1017/CBO9781107447875.007 Published online by Cambridge University Press

The first is a parametrized curve $\mathbf{x}(t) = (x(t), y(t), z(t))$, and as a specific example we consider the curve $C_{nm}(a)$

$$x = (1 + a\cos(nt))\cos(mt),\ y = (1 + a\cos(nt))\sin(mt),\ z = a\sin(nt)$$

where $t \in [0, 2\pi]$, n and m are integers and $0 < a < 1$. This is a spiral wrapped round a circular torus, with major and minor radii 1 and a respectively, a three-dimensional generalization of Lissajous' figures discussed above.

The second is a surface of the form $z = z(x, y)$. Here we consider the artificial but specific example

$$z = e^{-2x^2-y^2}\cos(2x)\cos(3y) \quad \text{where } -2 \leqslant x \leqslant 2,\ -3 \leqslant y \leqslant 3.$$

For the third case, we shall treat the more general case of a parametrized surface $x = x(u, v)$, $y = y(u, v)$, $z = z(u, v)$, which reduces to the case above if we choose $x = u$, $y = v$. As a concrete example, we consider the self-intersecting minimal surface discovered by Enneper

$$x = u(1 - u^2/3 + v^2),\ y = v(1 - v^2/3 + u^2),\ z = u^2 - v^2 \quad \text{where } -2 \leqslant u, v \leqslant 2.$$

It is not difficult to construct similar examples with a higher number of dimensions.

There is an issue which is almost trivial in two dimensions, namely the relationship, if any, between the data values. In two dimensions, when we call `plt.plot(x,y)`, `x` and `y` are treated as linear lists, and a line is drawn between contiguous data points. If we believe the data to be uncorrelated, then `plt.scatter(x,y)` would simply show the data points.

In three dimensions, there is more freedom. For example, if we were to plot the second example above when x and y were uniform arrays, e.g., as generated by `np.mgrid`, we would be tessellating a two-dimensional surface by rectangles. The Enneper surface above is a more general tiling by quadrilaterals. However, we might consider also a tessellation by triangles, or even impose no structure at all and simply plot the points. In higher dimensions, there are many more possibilities.

6.1.2 The reduction to two dimensions

The screen of a VDU and a paper page are both two dimensional, and so any representation of a three- or higher-dimensional object must ultimately be reduced to two dimensions. There are (at least) two standard processes to achieve this. In order to avoid too much abstraction in the presentation, let us assume that the object has been described in terms of coordinates $(x, y, z, w, \ldots)$.

The first reduction process, often called the "cut plane" technique is to impose some arbitrary condition or conditions on the coordinates so as to reduce the dimensionality. The simplest and most common approach is to assume constant values for all but two of the coordinates, e.g., $z = z_0$, $w = w_0, \ldots$, thus giving a restricted but two dimensional view of the object. Hopefully, by doing this several times for a (not too large) set of choices for $z_0, w_0, \ldots$, we will be able to recover valuable information about the object.

https://doi.org/10.1017/CBO9781107447875.007 Published online by Cambridge University Press

An example of this approach is the contour curve procedure in *matplotlib* mentioned in the previous section.

The second common reduction process is *projection*. Suppose first that the object is three dimensional. Suppose we choose an "observer direction", a unit vector in the three-dimensional space, and project the object onto the two-plane orthogonal to the observer direction. (If distinct object points project to the same image point we need to consider whether, and if so how, we want to distinguish them. This difficult issue is being ignored in this introduction.) In the standard notation for three-dimensional vectors, let $\mathbf{n}$ be the observer direction with $\mathbf{n} \cdot \mathbf{n} = 1$. Let $\mathbf{x}$ be an arbitrary three-vector. Then the projection operator $\mathcal{P}$ is defined by

$$\mathcal{P}(\mathbf{x}) = \mathbf{x} - (\mathbf{x} \cdot \mathbf{n})\mathbf{n},$$

and it is easy to see that this vector is orthogonal to $\mathbf{n}$, i.e., $\mathbf{n} \cdot \mathcal{P}(\mathbf{x}) = 0$ for all $\mathbf{x}$. In physical terms, the projection onto the plane spanned by the $\mathcal{P}(\mathbf{x})$ is what a distant observer would see. The projection technique also works in higher dimensions, but the physical interpretation is not as direct. Hopefully, by choosing enough directions we can hope to "visualize" the object.

Another point which should be noted is that if an object is to be resolved with N points per dimension and there are d dimensions, the number of points needed will be N^d. Thus if $d \geqslant 3$, we cannot expect to achieve resolutions as fine as in the two-dimensional case.

6.2 Visualization software

There are two tasks here which need to be considered separately, although the first is often glossed over.

In principle we need to convert the data to one of the more-or-less standard data formats. Perhaps the best known of these are the *HDF (Hierarchical Data Format)* mentioned in Section 4.5.2 and the *VTK (Visualization Tool Kit)*,[1] although there are plenty of others in the public domain. These formats aim at providing maximum generality in their descriptions of "objects" to be visualized, and so come with a fairly steep learning curve. The choice of a particular format is usually predicated by a choice of visualization package.

The task of a visualization package is to take data in a specified format and produce two-dimensional views of them. Packages must be fast, so that whether either cut plane or projection techniques are used, figures can be produced and altered with a minimal time lag. Given that they also need to produce a variety of outputs, images, movies etc., they tend to be very complicated with again a steep learning curve. Because they are not trivial to write, they are often commercial products with a very expensive price tag. Fortunately, there are some top quality examples in the public domain, and a far from complete list includes (in alphabetical order) *Mayavi*,[2] *Paraview*,[3] both of which are

[1] Its website is `http://www.vtk.org`.
[2] Its website is `http://mayavi.sourceforge.net`.
[3] It can be found at `http://www.paraview.org`.

https://doi.org/10.1017/CBO9781107447875.007 Published online by Cambridge University Press

based on *VTK*, and *Visit*[4] which uses *silo* or *HDF*. All three offer a Python interface to access their features. There is however a snag with the last two of the three examples listed. Like Python, they are complicated programmes which are still being developed. Because of the complexity involved in compiling them they all offer ready-to-run binary versions for all of the major platforms. Naturally, the Python interfaces only work with the Python version that was used when the binary programme was created, and this may not be the same version as that of the end user. However, because it is part of many Python distribution packages, *mayavi* should not suffer from this problem.

The philosophy of this book is both to try to minimize the addition of external software to the Python base, and, more importantly, to eliminate steep learning curves. Python avoids both of these disadvantages by offering two different approaches which are very similar to what we have already discussed. The first is a *mplot3d* module for *matplotlib*. The second is a *mlab* module which can be loaded from *mayavi* and which is itself a Python package. This makes many of the features of *mayavi* available with a *matplotlib*-like interface. For the documentation, see the user guide Ramachandandran and Variquaux (2009).

In the next three sections, we look at the two approaches to visualization for each of our three examples. The fastest and most satisfactory way to master visualization is to try out the code snippets and then experiment with them and with other examples. In each section, we first supply a code snippet to construct the data set. This will be common to both approaches, and so this first snippet should be pasted into the second or third snippet, which actually carries out the visualization. This should make it easier to reuse the latter snippets with the reader's own data set.

As usual, the use of the *IPython* interpreter is strongly recommended. No special considerations are needed when using *matplotlib*. However, if the *mayavi* package *mlab* is to be used, then *ipython* should be invoked from the command line with

```
ipython --gui=wx
```

assuming your version is 0.11 or later. Older versions need

```
ipython -wthread
```

6.3 A three-dimensional curve

We now consider the curve $C_{mn}(a)$ from the first example of Section 6.1.1. The data to be visualized are easily constructed using the following snippet, which we refer to as snippet A. This snippet is common to both of the drawing approaches illustrated below.

```
1 # This is snippet A
2
3 import numpy as np
```

[4] It is at `http://wci.llnl.gov/codes/visit`.

https://doi.org/10.1017/CBO9781107447875.007 Published online by Cambridge University Press

```
4
5  theta=np.linspace(0,2*np.pi,401)
6  a=0.3   # specific but arbitrary choice of the parameters
7  m=11
8  n=9
9  x=(1+a*np.cos(n*theta))*np.cos(m*theta)
10 y=(1+a*np.cos(n*theta))*np.sin(m*theta)
11 z=a*np.sin(n*theta)
```

6.3.1 Visualizing the curve with *mplot3d*

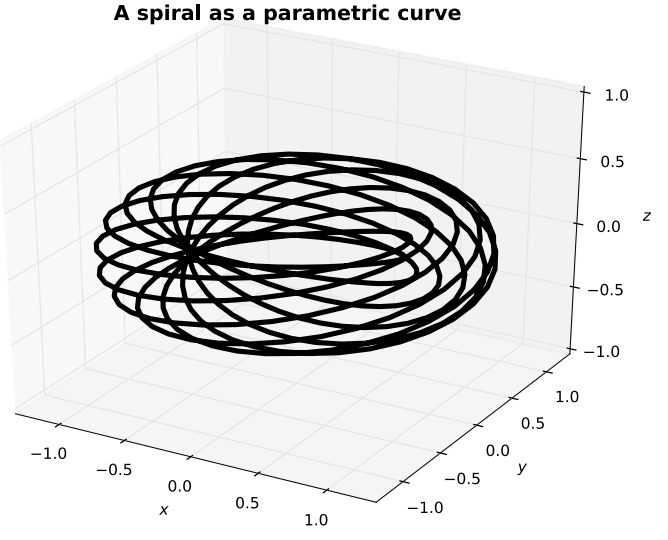

Figure 6.1 An example of a curve wrapped round a torus using *matplotlib.mplot3d* .

The next snippet uses the first to visualize the curve using *matplotlib* package *mplot3d*.

```
1  # Insert snippet A here.
2  import matplotlib.pyplot as plt
3  from mpl_toolkits.mplot3d import Axes3D
4
5  plt.ion()
6  fig=plt.figure()
7  ax=Axes3D(fig)
8  ax.plot(x,y,z,'g',linewidth=4)
9  ax.set_zlim3d(-1.0,1.0)
10 ax.set_xlabel('x')
```

https://doi.org/10.1017/CBO9781107447875.007 Published online by Cambridge University Press

```
11 ax.set_ylabel('y')
12 ax.set_zlabel('z')
13 ax.set_title('A spiral as a parametric curve',
14            weight='bold',size=16)
15 #ax.elev, ax.azim = 60, -120
16 fig.savefig('torus.pdf')
```

Line 2 should be familiar, and line 3 introduces the `Axes3D` class object which will be needed for carrying out the visualization. Next lines 6 and 7 join up the two concepts, `ax` is an instance of the object `Axes3D` tied to an instance `fig` of the *matplotlib* `Figure` class. Thus the remaining code follows the style of Section 5.9. Line 8 actually draws the curve. The parameters of `ax.plot` are almost identical to those of the two-dimensional function `plt.plot`. Likewise lines 9–14 and 16 carry out operations familiar from the two-dimensional case.

The figure should be shown in the familiar *matplotlib* window, where the interactive buttons behave as before (see Section 5.2.4). There is important new functionality however. Simply moving the mouse over the figure with the left button depressed will change the observer direction, and the azimuth and elevation of the current direction is displayed at the lower right corner.

As written, the snippet will save the figure drawn with the default values of azimuth and elevation, $-60°$ and $30°$ respectively, which are rarely the most useful ones. By experimenting, we can establish more desirable values. The commented out line 15 shows how to set them to values that are more desirable for this particular image. If you rerun the snippet with this line uncommented and amended, then the desired figure will be saved, as in, e.g., Figure 6.1. Figures for presentations often look better if a title is present, whereas they are usually unnecessary for books. As a compromise, both Figures 6.1 and 6.2 carry titles, to show how to implement them, but they are omitted for the rest of this chapter.

6.3.2 Visualizing the curve with *mlab*

The following snippet shows the default way to visualize the curve in *mayavi*. (Remember to invoke *ipython* with the `--gui=wx` parameter.)

```
1 # Insert snippet A here.
2 from mayavi import mlab
3
4 mlab.plot3d(x,y,z,np.sin(n*theta),
5           tube_radius=0.025,colormap='spectral')
6 mlab.axes(line_width=2,nb_labels=5)
7 mlab.title('A spiral wrapped around a torus',size=0.6)
8 #mlab.show() # not needed when using ipython
9 mlab.savefig('torus1.png')
```

https://doi.org/10.1017/CBO9781107447875.007 Published online by Cambridge University Press

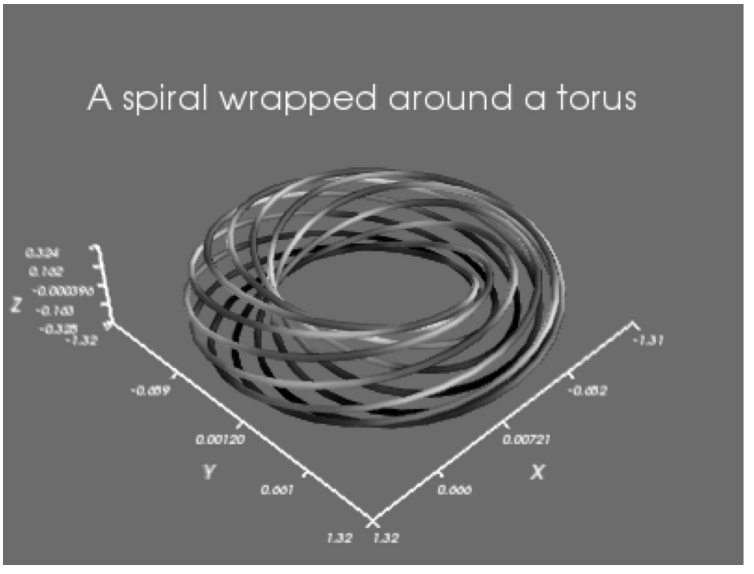

Figure 6.2 An example of a curve wrapped round a torus using the *mayavi mlab* module with the default settings.

The result is shown in Figure 6.2. Lines 4 and 5 draw the curve. Here we have chosen to colour the curve according the value of $\sin(n\theta)$, where θ is the curve's parameter, using the colour map `spectral`. (Colour maps are discussed below.) Lines 6 and 7 produces axes and a title. The docstrings give further possibilities. (Note that the *mayavi* developers have chosen the Matlab-compatible version of names for functions to decorate figures. See Section 5.9 for a discussion of this issue.) Finally, line 9 saves a copy to disk. Note that although the documentation suggests that `mlab.savefig` can save to a variety of formats, this appears to depend on the back-end used. My back-end only saves successfully `png` files. However, if you installed *ImageMagick* as mentioned in Section 5.10.2, its command line `convert` can be used to convert a `png` file to many other formats.

The window here is rather different to that of *matplotlib*. Once the figure has been drawn we can change the viewing angle by dragging the mouse across it with the left button depressed. The right button offers pan and zoom facilities. At the top left, there are 13 small buttons for interacting with the figure. By default, the background colour is black, and button 13 (on the right) allows us to change this. Button 12 is used for saving the current scene. The formats available are implementation-dependent, but all implementations should allow the scene to be saved as a `png` file. Buttons 2–11 do fairly obvious things, but button 1 (on the left) is far more intriguing . Pressing it reveals the *mayavi* "pipeline" that *mlab* constructed to draw the figure. For example, clicking on "Colors and legends" shows the chosen colour map or "Look Up Table", abbreviated to "LUT". Clicking on the colour bar reveals the 60 available choices. The reader is urged to experiment further with the other pipeline elements!

https://doi.org/10.1017/CBO9781107447875.007 Published online by Cambridge University Press

The default choices of grey background, white axes, labels and title are very effective for visual presentations, but print formats usually expect a white background with black axes, labels and title. The easiest way to do this is via button 13, as mentioned in the preceding paragraph. Alternatively, we can modify our code. In *mayavi*, colour is managed by a triple of floating-point numbers (rgb values) stored as a tuple, and so we need to manage these for the foreground and background. Thus in the snippet above we might add the definitions

```
black=(0,0,0)
white=(1,1,1)
```

and then modify the figure, axis and title commands, e.g.

```
mlab.figure(bgcolor=white)
mlab.axes(line_width=2,nb_labels=5,color=black)
mlab.title('A spiral wrapped around a torus',size=0.6,
          color=black)
```

The interactive approach is simpler and more versatile.

6.4 A simple surface

In this section, we consider the simple surface defined above in Section 6.1.1 to be the graph of

$$z = e^{-2x^2-y^2}\cos(2x)\cos(3y) \quad \text{where } -2 \leqslant x \leqslant 2,\ -3 \leqslant y \leqslant 3.$$

The following snippet, referred to later as snippet B, sets up the data.

```
import numpy as np

xx, yy=np.mgrid[-2:2:81j, -3:3:91j]
zz=np.exp(-2*xx**2-yy**2)*np.cos(2*xx)*np.cos(3*yy)
```

6.4.1 Visualizing the simple surface with *mplot3d*

Figure 6.3 was drawn using the following code snippet

```
# Insert snippet B here to set up the data
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

plt.ion()
fig=plt.figure()
ax=Axes3D(fig)
ax.plot_surface(xx,yy,zz,rstride=4,cstride=3,color='c',alpha=0.9)
```

https://doi.org/10.1017/CBO9781107447875.007 Published online by Cambridge University Press

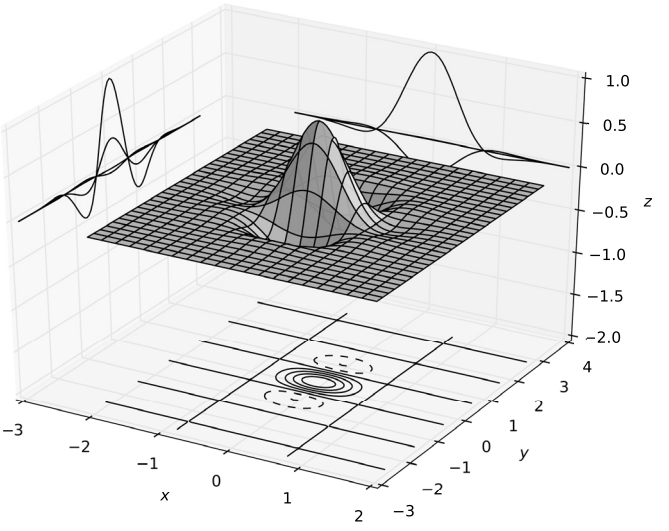

Figure 6.3 An example of a simple surface with three contour plots using the *matplotlib.mplot3d* module.

```
9  ax.contour(xx,yy,zz,zdir='x',offset=-3.0,colors='black')
10 ax.contour(xx,yy,zz,zdir='y',offset=4.0,colors='blue')
11 ax.contour(xx,yy,zz,zdir='z',offset=-2.0)
12 ax.set_xlim3d(-3.0,2.0)
13 ax.set_ylim3d(-3.0,4.0)
14 ax.set_zlim3d(-2.0,1.0)
15 ax.set_xlabel('x')
16 ax.set_ylabel('y')
17 ax.set_zlabel('z')
18 #ax.set_title('Surface plot with contours',
19 #             weight='bold',size=18)
```

Here lines 1–7 have appeared before in the snippet used to visualize a parametric curve. Line 8 draws the surface. The parameters `rstride=4, cstride=3` mean that every fourth row and third column are actually used to construct the plot. The parameter `alpha=0.9` controls the opacity[5] of the surface, a floating-point number in the range [0, 1]. See the function's docstring for further potential parameters. Again we have seen lines 15–19 before.

If a "wire frame" representation of the surface is preferred, it can be obtained by replacing line 8 by

```
ax.plot_wireframe(xx,yy,zz,rstride=4,cstride=3)
```

[5] A high alpha means that the surface is very opaque, i.e., mesh lines behind the hump are not seen, whereas a low alpha renders the hump transparent.

https://doi.org/10.1017/CBO9781107447875.007 Published online by Cambridge University Press

The function's docstring lists the further possibilities.

Another new feature here is the drawing of contour plots. Recall that in Section 5.8 we showed how *matplotlib* could visualize the level contours of a surface $z = z(x, y)$. In line 9 of the current snippet, we do the same after rearranging the functional relationship to be $x = x(y, z)$. We have to position the plot as a yz-plane somewhere and the parameter `offset=-3` puts it in the plane $x = -3$. Now the default x-range was set by snippet B to be $x \in [-2, 2]$, which would render the contour plot invisible. Therefore, line 12 resets the x-range to be $x \in [-3, 2]$. Lines 10 and 11 and 13 and 14 deal with the other coordinate directions. Of course, there is no obligation to draw all three contour plots, or even one or two of them. Your figures will certainly look less cluttered without them!

6.4.2 Visualizing the simple surface with *mlab*

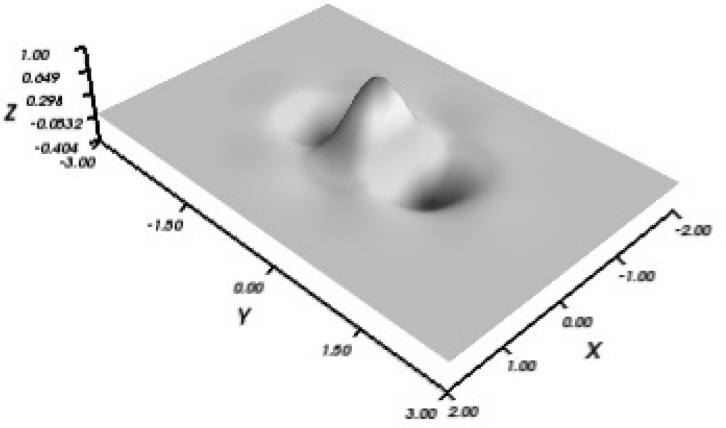

Figure 6.4 An example of a simple surface visualized using *mayavi*'s *mlab* module.

Figure 6.4 was drawn using the following code snippet

```
1  # Insert snippet B here to set up the data
2
3  from mayavi import mlab
4  fig=mlab.figure()
5  s=mlab.surf(xx,yy,zz,representation='surface')
6  ax=mlab.axes(line_width=2,nb_labels=5)
7  #mlab.title('Simple surface plot',size=0.4)
```

https://doi.org/10.1017/CBO9781107447875.007 Published online by Cambridge University Press

The new feature here is line 5, and perusal of the relevant docstring is worthwhile. Note in particular `'surface'` is the default, and replacing it with `'wireframe'` produces an alternative representation. Setting `warp_scale` to a float value will magnify (or contract) the z-axis.

Besides the interactive controls described in Section 6.3.2, there are other ways to obtain information about and to control the figure. Typing `mlab.view()` into the interpreter will produce six numbers. The first is the "azimuth" of the observer or "camera" direction, measured in degrees in the range $[0, 360]$. The second is the "elevation" in the range $[0, 180]$. The third is the distance of the observer, and the final three give the focal point of the camera, which by default is the origin. Any of these parameters can be set by invoking the function, e.g.,

```
mlab.view(azimuth=300,elevation=60)
```

Although interactive button 12 can be used to save the figure

```
mlab.savefig('foo.png')
```

will do the same job. If, like the author, you find it difficult to rotate the figure while keeping the z-axis vertical, this can be ameliorated by adding the lines

```
from tvtk.api import tvtk
fig.scene.interactor.interactor_style=tvtk.InteractorStyleTerrain()
```

after line 4 of the snippet.

6.5 A parametrically defined surface

We turn now to the visualization of Enneper's surface, defined in Section 6.1.1 by

$$x = u(1 - u^2/3 + v^2),\ y = v(1 - v^2/3 + u^2),\ z = u^2 - v^2 \quad \text{where } -2 \leqslant u, v \leqslant 2.$$

A code snippet, henceforth referred to as snippet C, to generate the data is

```
1 # This is snippet C
2 import numpy as np
3 [u,v]=np.mgrid[-2:2:51j, -2:2:61j]
4 x,y,z=u*(1-u**2/3+v**2),v*(1-v**2/3+u**2),u**2-v**2
```

Here the difference between the two plotting tools becomes significant.

6.5.1 Visualizing Enneper's surface using *mplot3d*

A code snippet to draw Figure 6.5 is

```
1 # Insert snippet C here
2 import matplotlib.pyplot as plt
```

https://doi.org/10.1017/CBO9781107447875.007 Published online by Cambridge University Press

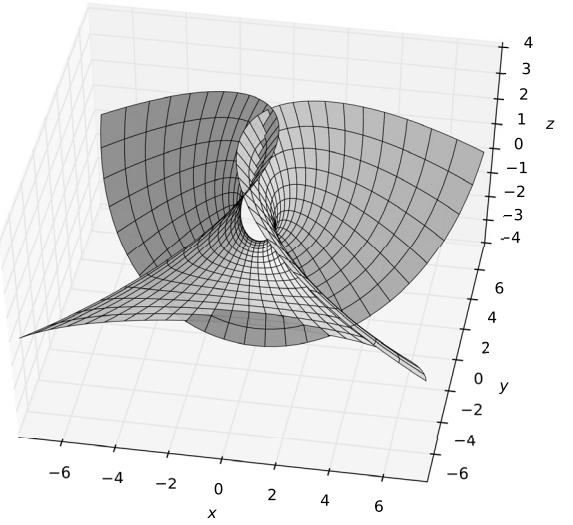

Figure 6.5 Enneper's surface visualized using *matplotlib*'s *mplot3d* module.

```
3  from mpl_toolkits.mplot3d import Axes3D
4
5  plt.ion()
6  fig=plt.figure()
7  ax=Axes3D(fig)
8  ax.plot_surface(x.T,y.T,z.T,rstride=2,cstride=2,color='r',
9                  alpha=0.2,linewidth=0.5)
10 ax.elev, ax.azim = 50, -80
11 ax.set_xlabel('x')
12 ax.set_ylabel('y')
13 ax.set_zlabel('z')
14 #ax.set_title('A parametric surface plot',
15 #               weight='bold',size=18)
```

The new features are all in line 8. Notice first that the underlying code is based on image-processing conventions, and to accord with *matplotlib* conventions we need to supply the transpose of the x-, y- and z-arrays. This is a non-trivial self-intersecting surface, and its visualization requires a judicious choice of the opacity, line width and stride parameters.

There is a great deal of computation going on here, which will tax even the fastest processors. In particular, interactive panning and tilting will be slowed down significantly.

https://doi.org/10.1017/CBO9781107447875.007 Published online by Cambridge University Press

6.5.2 Visualizing Enneper's surface using *mlab*

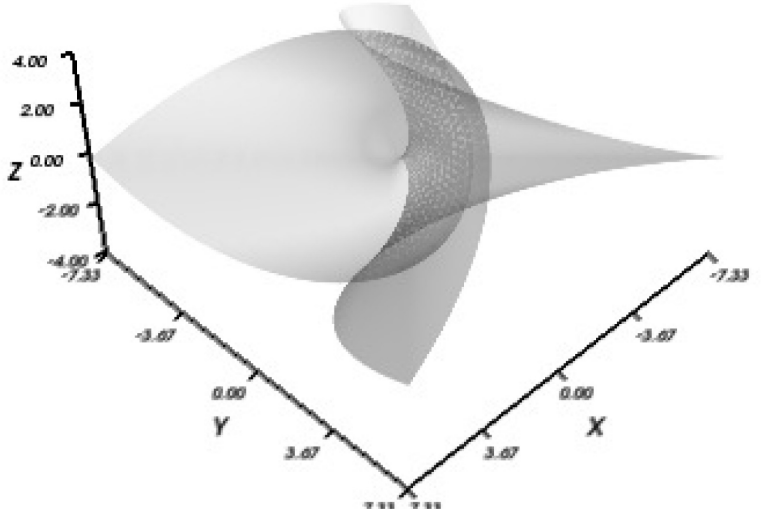

Figure 6.6 Enneper's surface visualized using *mayavi*'s *mlab* module.

The *mlab* module can also be used to visualize Enneper's surface. Figure 6.6 can be drawn with the code snippet

```
# Insert snippet C here to set up the data

from mayavi import mlab
fig=mlab.figure()
s=mlab.mesh(x,y,z,representation='surface',
          line_width=0.5,opacity=0.5)
mlab.axes(line_width=2,nb_labels=5)
#mlab.title('A parametrically defined surface',size=0.4)
```

Interactive panning and tilting is noticeably faster using *Mayavi's mlab* module, than using *mplot3d*. Why is this so? Well *matplotlib* and *mplot3d* use "vector graphics" which deal with shapes and are infinitely stretchable. *Mayavi* and *mlab* use "bitmapped graphics" which create and store each pixel of the picture. Its files are large but easy to create. The big disadvantage is that only a limited amount of expansion or contraction is possible without significant loss of visual quality. This why the function `mlab.savefig` only works satisfactorily with a bitmap format such as `png`.

https://doi.org/10.1017/CBO9781107447875.007 Published online by Cambridge University Press

6.6 Three-dimensional visualization of a Julia set

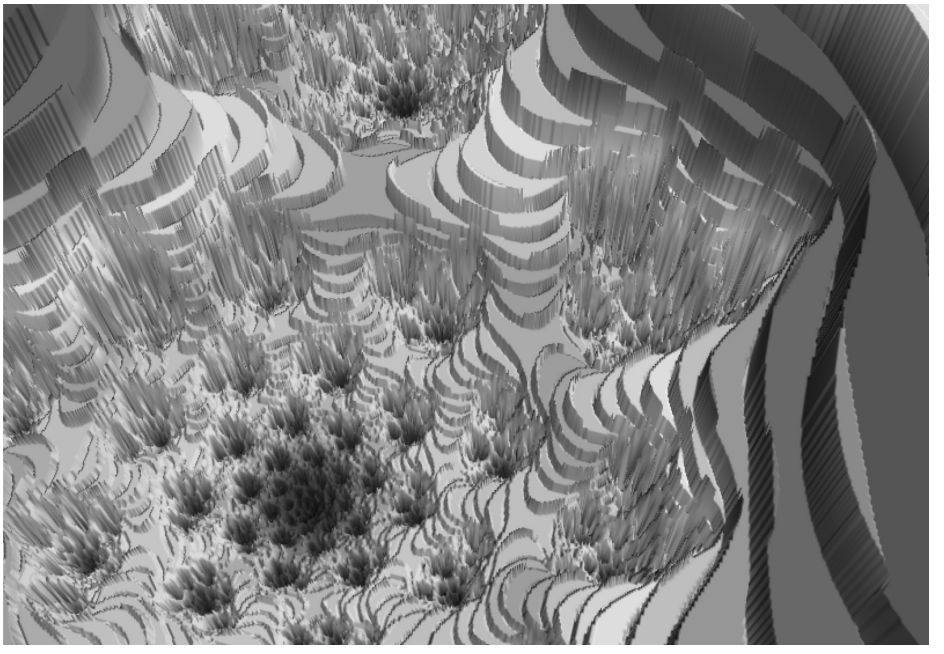

Figure 6.7 A Julia set for the mapping $z \to z^2 + c$ with $c = -0.7 - 0.4i$. Points near the top of the "canyon" correspond to trajectories which escaped to infinity early, while those at the bottom have yet to escape.

We turn now to the use of *mlab* to visualize figures that are more complicated. In Section 5.11, we sketched an introduction to fractal sets and showed how to construct the classic visualization of the Mandelbrot set for the mapping $z \to z^2 + c$ as realized by the iteration (5.1). Here we shall be concerned with the Julia set for the same mapping. Now we hold the parameter c fixed and let the iterates z_n depend on the initial value z_0. Then the escape parameter ϵ depends on z_0 rather than c, $\epsilon = \epsilon(z_0)$. Instead of the high-resolution, but fixed, image of Section 5.11, we look for an interactive, but lower-resolution picture, a so-called *canyon view*, as exemplified in the following self-contained snippet, which is adapted from the Mayavi documentation, Ramachandandran and Variquaux (2009), see also Mayavi Community (2011), and is not optimized for speed.

```
1 import numpy as np
2
3 # Set up initial grid
4 x,y=np.ogrid[-1.5:0.5:1000j,-1.0:1.0:1000j]
5 z=x+1j*y
6 julia = np.zeros(z.shape)
7 c=-0.7-0.4j
```

https://doi.org/10.1017/CBO9781107447875.007 Published online by Cambridge University Press

```
8
9   # Build the Julia set
10  for it in range(1,101):
11      z=z**2+c
12      escape=z*z.conj()>4
13      julia+=(1/float(it))*escape
14
15  from mayavi import mlab
16  mlab.figure(size=(800,600))
17  mlab.surf(julia,colormap='gist_ncar',
18          warp_scale='auto',vmax=1.5)
19  mlab.view(15,30,500,[-0.5,-0.5,2.0])
20  mlab.show()
```

Lines 1–7 are straightforward. The grid `julia` is set initially to zeros, and an arbitrary choice for c is made. In lines 10–13, we perform 100 iteration steps. `escape` is a grid of booleans which are `False` if the corresponding $|z_n| \leq 2$ and are `True` otherwise. For the escaped points, we increment `julia` by a small amount. Finally, lines 16–19 draw the figure. The parameters in lines 17–19 were chosen by trial and error. The output of this snippet is shown in Figure 6.7.

The reader is urged to try out this code, and then to make changes. The most passive way to do this is simply to change the code and then rerun it. More versatility can be obtained by using the cursor to modify the picture and button 13 to change values. Then typing, e.g., `mlab.view()` into the interpreter will reveal the current chosen values. Pressing button 1 brings up a *mayavi* window. The *mayavi* "pipeline" is shown upper left. Left clicking on any pipeline item will reveal the options chosen and offer the means to change them. With a little practice it's actually very easy and adaptable.

Obviously, in an introductory text we cannot survey all of the possibilities for three-dimensional graphics. In particular, we have not covered the important topic of animations. The reader is therefore urged to explore further the abilities of the *mlab* module. It is fast, versatile and easy to use, although it lacks the extremely high resolution of *matplotlib*.

https://doi.org/10.1017/CBO9781107447875.007 Published online by Cambridge University Press