Chapter 2

Visualization of scalar and vector fields

2.1 Examples on plotting of scalar and vector fields

We shall demonstrate visualization of scalar and vector fields using Matplotlib, Mayavi, and Matlab. The former two packages requires a bit of Python programming.

To exemplify visualization of scalar and vector fields with various tools, we use a common set of examples. A scalar function of x and y is visualized either as a flat two-dimensional plot with contour lines of the field, or as a three-dimensional surface where the height of the surface corresponds to the function value of the field. In the latter case we also add a three-dimensional parameterized curve to the plot.

To illustrate plotting of vector fields, we simply plot the gradient of the scalar field, together with the scalar field. Our convention for variable names goes as follows:

- `x`, `y` for one-dimensional coordinates along each axis direction.
- `xv`, `yv` for the corresponding vectorized coordinates in a 2D.
- `u`, `v` for the components of a vector field at points corresponding to `xv`, `yv`.

The following sections contain more mathematical details on the various scalar and vector fields we aim to plot.

2.1.1 Surface plots

We consider the 2D scalar field defined by

$$h(x,y) = \frac{h_0}{1+\frac{x^2+y^2}{R^2}}. \tag{2.1}$$

$h(x,y)$ may model the height of an isolated circular mountain, h being the height above sea level, while x and y are Cartesian coordinates on the earth's surface, h_0 the height of the mountain, and R the radius of the mountain. Since mountains are actually quite flat (or more precisely, their heights are small compared to the horizontal extent), we use meter as length unit for vertical distances (z direction) and km as length unit for horizontal distances (x and y coordinates). Prior to all code below we have initialized h_0 and R with the following values: $h_0 = 2277$ m and $R = 4$ km.

Grid for 2D scalar fields. Before we can plot $h(x,y)$, we need to create a rectangular grid in the xy plane with all the points used for plotting. With Matplotlib and Mayavi we need the code

```
x = y = np.linspace(-10., 10., 41)
xv, yv = np.meshgrid(x, y, indexing='ij', sparse=False)

hv = h0/(1 + (xv**2+yv**2)/(R**2))      # Elevation coordinates (m)
```

while in Matlab we must write

```
x = linspace(-10, 10, 41);
y = x;
[xv, yv] = meshgrid(x, y);

hv = h0./(1 + (xv.^2+yv.^2)/(R^2));      % Elevation coordinates (m)
```

The grid is based on equally spaced coordinates `x` and `y` in the interval $[-10, 10]$ km. Note the mysterious extra parameters to `meshgrid` here, which are needed in order for the coordinates to have the right order such that the arithmetics in the expression for `hv` becomes correct. The expression computes the surface value at the 41×41 grid points in one vectorized operation.

A surface plot of a 2D scalar field $h(x,y)$ is a visualization of the surface $z = h(x,y)$ in three-dimensional space. Most plotting packages have functions which can be used to create surface plots of 2D scalar fields. These can be either *wireframe plots*, where only lines connecting the grid points are drawn, or plots where the faces of the surface are colored. In Figure 2.1 we have shown two such plots of the surface $h(x,y)$. Section 2.2.1 presents the code which generates these plots.

2.1.2 Parameterized curve

To illustrate the plotting of three-dimensional parameterized curves, we consider a trajectory that represents a circular climb to the top of the mountain:

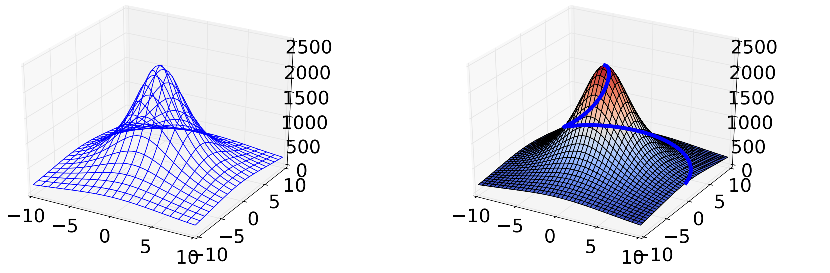

Figure 2.1: Two different plots of a mountain. The right plot also shows a trajectory to the top of the mountain.

$$\boldsymbol{r}(t) = \left(10\left(1-\frac{t}{2\pi}\right)\cos(t)\right)\boldsymbol{i} + \left(10\left(1-\frac{t}{2\pi}\right)\sin(t)\right)\boldsymbol{j} + \frac{h_0}{1+\frac{100(1-t/(2\pi))^2}{R^2}}\boldsymbol{k}. \tag{2.2}$$

Here $\boldsymbol{i}$, $\boldsymbol{j}$, and $\boldsymbol{k}$ denote the unit vectors in the x-, y-, and z-directions, respectively. In Python the coordinates of $\boldsymbol{r}(t)$ can be produced by

```
s = np.linspace(0, 2*np.pi, 100)
curve_x = 10*(1 - s/(2*np.pi))*np.cos(s)
curve_y = 10*(1 - s/(2*np.pi))*np.sin(s)
curve_z = h0/(1 + 100*(1 - s/(2*np.pi))**2/(R**2))
```

while in Matlab we must write

```
s = linspace(0, 2*pi, 100);
curve_x = 10*(1 - s/(2*pi)).*cos(s);
curve_y = 10*(1 - s/(2*pi)).*sin(s);
curve_z = h0./(1 + 100*(1 - s/(2*pi)).^2/(R^2));
```

The parameterized curve is shown together with the surface $h(x, y)$ in the right plot in Figure 2.1.

2.1.3 Contour lines

Contour lines are lines defined by the implicit equation $h(x, y) = C$, where C is some constant representing the contour level. Normally, we let C run over some equally spaced values, and very often, the plotting program computes the C values. To distinguish contours, one often associates each contour level C with its own color.

Figure 2.2 shows different ways contour lines can be used to visualize the surface $h(x, y)$. The first and last plot are visualizations utilizing two spatial dimensions. The first draws a small set of contour lines only, while the last one displays the surface as an image, whose colors reflect the values of the field, or equivalently, the height of the surface. The third plot actually combines three different types of contours, each type corresponding to keeping a coordinate constant and projecting the contours on a “wall”. The code used to generate these plots is presented in Section 2.2.2.

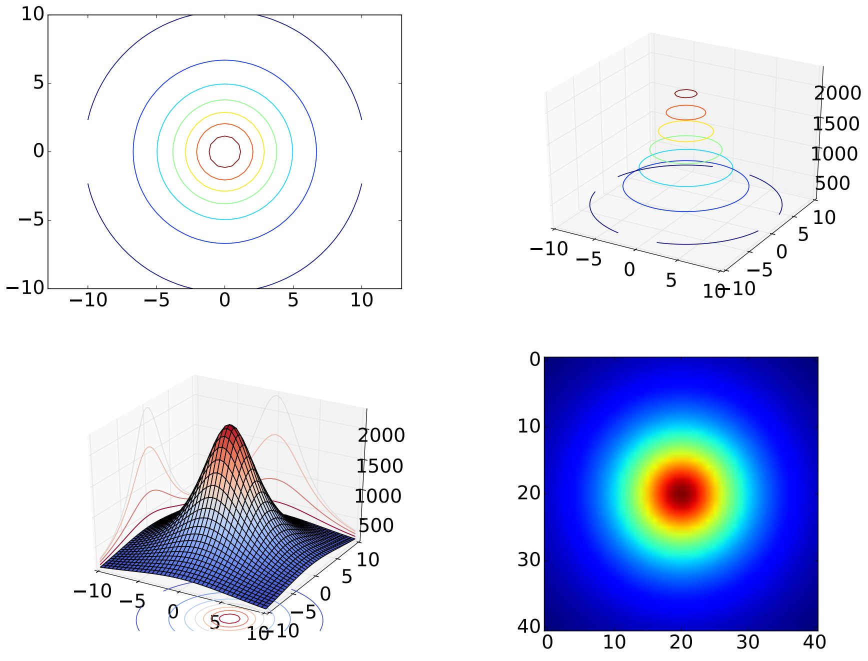

Figure 2.2: Different types of contour plots of a 2D scalar field in two and three dimensions.

2.1.4 The gradient vector field

The *gradient vector field* ∇h of a 2D scalar field $h(x, y)$ is defined by

$$\nabla h = \frac{\partial h}{\partial x}\boldsymbol{i} + \frac{\partial h}{\partial y}\boldsymbol{j}. \tag{2.3}$$

One learns in vector calculus that the gradient points in the direction where h increases most, and that the gradients are orthogonal to the contour lines. This is something we can easily illustrate by creating 2D plots of the contours and the gradient field. A challenge in making such plots is to get the right arrow lengths so that the arrows are well visible, but they do not collide and make a cluttered visual impression. Since the arrows are drawn at each point in a 2D grid, one way of controlling the number of arrows is to control the resolution of the grid.

So, let us create a grid with 20 instead of 40 intervals in the horizontal directions: In Python we write

```
x2 = y2 = np.linspace(-10.,10.,11)
x2v, y2v = np.meshgrid(x2, y2, indexing='ij', sparse=False)
h2v = h0/(1 + (x2v**2 + y2v**2)/(R**2)) # Surface on coarse grid
```

The gradient vector field of $h(x, y)$ can now be computed using the function `np.gradient`:

```
dhdx, dhdy = np.gradient(h2v) # dh/dx, dh/dy
```

In Matlab we instead write

```
x2 = linspace(-10, 10, 11);
y2 = x2;
[x2v, y2v] = meshgrid(x2, y2);
h2v = h0./(1 + (x2v.^2 + y2v.^2)/(R^2)); % Surface on coarse grid
```

The corresponding function for computing the gradient field in Matlab is called `gradient`:

```
[dhdx, dhdy] = gradient(h2v); % dh/dx, dh/dy
```

The gradient field (2.3) together with the contours appear in Figure 2.3, from which the orthogonality can be easily seen. Section 2.2.3 explains the code needed to make this plot.

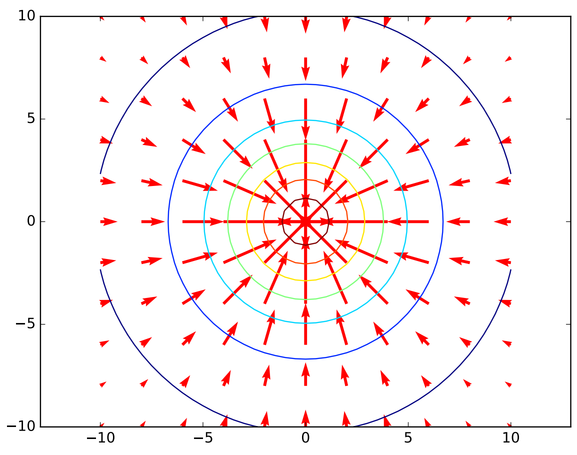

Figure 2.3: Gradient field with contour plot.

2.2 Matplotlib

We import any visualization package under the name `plt`, so for Matplotlib the import is done by

```
import matplotlib.pyplot as plt
```

When creating two-dimensional plots of scalar and vector fields, we shall make use of a Matplotlib `Axes` object, named `ax` and made by

```
fig = plt.figure(1)   # Get current figure
ax = fig.gca()        # Get current axes
```

For three-dimensional visualization, we need the following alternative lines:

```
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(1)
ax = fig.gca(projection='3d')
```

2.2.1 Surface plots

The Matplotlib functions for producing surface plots of 2D scalar fields are `ax.plot_wireframe` and `ax.plot_surface`. The first one produces a wireframe plot, and the second one colors the surface. The following code uses the functions to produce the plots shown in Figure 2.1, once the grid has been defined as in Section 2.1.1, and the coordinates of the parameterized curve have been computed as in Section 2.1.2.

```
fig = plt.figure(1)
ax = fig.gca(projection='3d')
ax.plot_wireframe(xv, yv, hv, rstride=2, cstride=2)

# Simple plot of mountain and parametric curve
fig = plt.figure(2)
ax = fig.gca(projection='3d')
from matplotlib import cm
ax.plot_surface(xv, yv, hv, cmap=cm.coolwarm, rstride=1, cstride=1)

# add the parametric curve. linewidth controls the width of the curve
ax.plot(curve_x, curve_y, curve_z, linewidth=5)
```

Recall that a final `plt.show()` command is necessary to force Matplotlib to show a plot on the screen.

Note that the second plot in this figure is drawn using a finer grid. This is controlled with the `rstride` and `cstride` parameters, which sets the number of grid lines in each direction. Setting one of these to 1 means that a grid line is drawn for every value in the grid in the corresponding direction, and setting to 2 means that a grid line will be drawn for every two values in the grid. You will normally need to experiment with such parameters to get a visually attractive plot.

A surface with colors reflecting the height of the surface needs specification of a *color map*, which is a mapping between function values and colors. Above we applied the common `coolwarm` scheme which goes from blue ("cool" color for minimum values) to red ("warm" color for maximum values). There are lots of colormaps to choose from, and you have to experiment to find appropriate choices according to your taste and to the problem at hand.

To the latter plot we also added the parameterized curve $\boldsymbol{r}(t)$, defined by (2.2), using the command `plot`. The attribute `linewidth` is increased here in order to make the curve thicker and more visible. By default, Matplotlib adds plots to each other without any need for `plt.hold('on')`, although such a command can indeed be used.

2.2.2 Contour plots

The following code exemplifies different types of contour plots. The first two plots (default two-dimensional and three-dimensional contour plots) are shown in Figure 2.2. The next four plots appear in Figure 2.4. Note that, when we asked Matplotlib to plot 10 contours, the response was, surprisingly, 9 contour lines, where one of the contours was incomplete. This kind of behavior may also be found in other plotting packages: The package will do its best to plot the requested number of complete contour lines, but there is no guarantee that this number is achieved exactly.

```
# Default two-dimensional contour plot with 7 colored lines
fig = plt.figure(3)
ax = fig.gca()
ax.contour(xv, yv, hv)
plt.axis('equal')

# Default three-dimensional contour plot
fig = plt.figure(4)
ax = fig.gca(projection='3d')
ax.contour(xv, yv, hv)

# Plot of mountain and contour lines projected on the coordinate planes
fig = plt.figure(5)
ax = fig.gca(projection='3d')
ax.plot_surface(xv, yv, hv, cmap=cm.coolwarm, rstride=1, cstride=1)
# zdir is the projection axis
# offset is the offset of the projection plane
ax.contour(xv, yv, hv, zdir='z', offset=-1000, cmap=cm.coolwarm)
ax.contour(xv, yv, hv, zdir='x', offset=-10,   cmap=cm.coolwarm)
ax.contour(xv, yv, hv, zdir='y', offset=10,    cmap=cm.coolwarm)

# View the contours by displaying as an image
fig = plt.figure(6)
ax = fig.gca()
ax.imshow(hv)

# 10 contour lines (equally spaced contour levels)
fig = plt.figure(7)
ax = fig.gca()
ax.contour(xv, yv, hv, 10)
plt.axis('equal')
```

```
# 10 black ('k') contour lines
fig = plt.figure(8)
ax = fig.gca()
ax.contour(xv, yv, hv, 10, colors='k')
plt.axis('equal')

# Specify the contour levels explicitly as a list
fig = plt.figure(9)
ax = fig.gca()
levels = [500., 1000., 1500., 2000.]
ax.contour(xv, yv, hv, levels=levels)
plt.axis('equal')

# Add labels with the contour level for each contour line
fig = plt.figure(10)
ax = fig.gca()
cs = ax.contour(xv, yv, hv)
plt.clabel(cs)
plt.axis('equal')
```

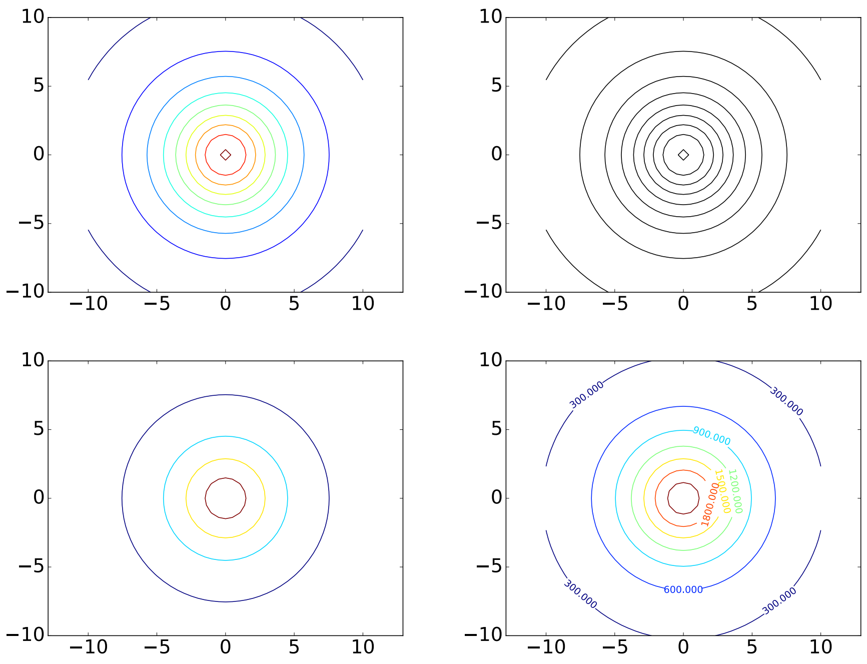

Figure 2.4: Some other contour plots with Matplotlib: 10 contour lines (upper left), 10 black contour lines (upper right), specified contour levels (lower left), and labeled levels (lower right).

2.2.3 Vector field plots

The code for plotting the gradient field (2.3) together with contours goes as explained below, once the grid has been defined as in Section 2.1.4. The corresponding plot is shown in Figure 2.3.

```
fig = plt.figure(11)
ax = fig.gca()
ax.quiver(x2v, y2v, dhdx, dhdy, color='r',
          angles='xy', scale_units='xy')
ax.contour(xv, yv, hv)
plt.axis('equal')
```

2.3 Mayavi

Mayavi is an advanced, free, easy to use, scientific data visualizer, with an emphasis on three-dimensional visualization techniques. The package is written in Python, and uses the Visualization Toolkit (VTK) in C++ for rendering graphics. Since VTK can be configured with different backends, so can Mayavi. Mayavi is cross platform and runs on most platforms, including Mac OS X, Windows, Linux.

The web page `http://docs.enthought.com/mayavi/mayavi/` collects pointers to all relevant documentation of Mayavi. We shall primarily deal with the `mayavi.mlab` module, which provides a simple interface to plotting of 2D scalar and vector fields with commands that mimic those of Matlab. Let us import this module under our usual name `plt` for a plotting package:

```
import mayavi.mlab as plt
```

The official documentation of the `mlab` module is provided in two places, one for the basic functionality and one for further functionality. Basic figure handling is very similar to the one we know from Matplotlib. Just as for Matplotlib, all plotting commands you do in `mlab` will go into the same figure, until you manually change to a new figure.

2.3.1 Surface plots

Mayavi has the functions `mesh` and `surf` for producing surface plots. These are similar, but `surf` assumes an orthogonal grid, and uses this assumption to make efficient data structures, while `mesh` makes no such assumptions on the grid. Here we only use orthogonal grids and hence apply `surf`. The following code plots the surface $h(x, y)$ in (2.1), as well as the parameterized curve $\boldsymbol{r}(t)$ in (2.2). The resulting graphics appears in Figure 2.5.

```
# Create a figure with white background and black foreground
plt.figure(1, fgcolor=(.0, .0, .0), bgcolor=(1.0, 1.0, 1.0))
# 'representation' controls the plot type, here a wireframe plot
```

```
plt.surf(xv, yv, hv, extent=(0,1,0,1,0,1), representation='wireframe')
# Decorate axes (nb_labels is the number of labels in each direction)
plt.axes(xlabel='x', ylabel='y', zlabel='z', nb_labels=5,
         color=(0., 0., 0.))
# Decorate the plot with a title (size is the size of the title)
plt.title('h(x,y)', size=0.4)

# Simple plot of mountain and parametric curve.
plt.figure(2, fgcolor=(.0, .0, .0), bgcolor=(1.0, 1.0, 1.0))
# The default for 'representation' is to color the surface elements.
plt.surf(xv, yv, hv, extent=(0,1,0,1,0,1))
# Add the parametric curve. tube_radius is the width of the curve
# (use 'extent' for auto-scaling)
plt.plot3d(curve_x, curve_y, curve_z, tube_radius=0.2,
           extent=(0,1,0,1,0,1))

plt.figure(3, fgcolor=(.0, .0, .0), bgcolor=(1.0, 1.0, 1.0))
# Use 'warp_scale' for vertical scaling
plt.surf(xv, yv, hv, warp_scale=0.01, color=(.5, .5, .5))
plt.plot3d(curve_x, curve_y, 0.01*curve_z, tube_radius=0.2)
```

`surf` can produce wireframe plots, as well as plots where the faces of the surface are colored. The parameter `representation` controls this, as exemplified in the first two plots. The first plot was also decorated with axes and a title.

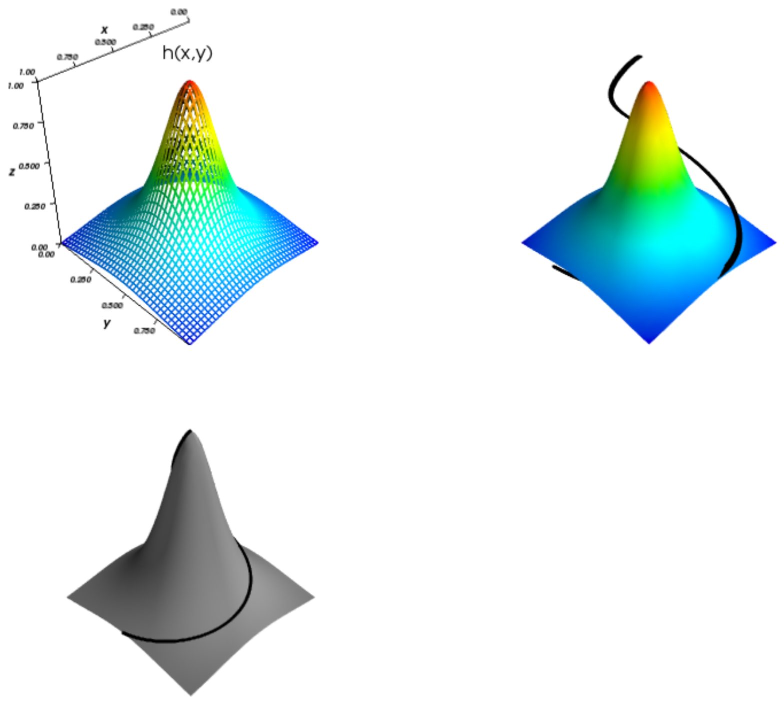

Figure 2.5: Surface plots produced with the `surf` function of Mayavi: The curve $\boldsymbol{r}(t)$ is also shown in the two last plots.

The calls to `plt.figure()` take three parameters: First the usual index for the plot, then two tuples of numbers , representing the RGB-values to be used for the foreground (`fgcolor`) and the background (`bgcolor`). White and black are (1,1,1) and (0,0,0), respectively. The foreground color is used for text and labels included in the plot. The `color` attribute in `plt.surf` adjusts the surface so that it is colored with small variations from the provided base color, here `(.5, .5, .5)`.

The command `plot3d` is used to plot the curve $\boldsymbol{r}(t)$. We have here increased the attribute `tube_radius` to make the curve thicker and more visible.

Mayavi does no auto-scaling of the axes by default (contrary to Matplotlib), so if the magnitudes in the vertical and horizontal directions are very different, as they are for $h(x, y)$, the plots may be very concentrated in one direction. We therefore need to apply some auto-scaling procedure. In Figure 2.5 two such procedures are exemplified. In the first two plots the parameter `extent` is used. It tells Mayavi to auto-scale the surface and curve to fit the contents described by the six listed values (we will return to what these values mean when we have a more illustrating example). Since the curve and the surface span different areas in space, we see that they are auto-scaled differently in the second plot, with the undesired effect that $\boldsymbol{r}(t)$ is not drawn on the surface. The last plot has avoided this problem by using the `warp_scale` parameter for scaling the vertical direction. Not all Mayavi functions accept this parameter. A remedy for this is to scale the z-coordinates manually, as here exemplified in the last `plot3d`-call. As is seen, the curve is drawn correctly with respect to the surface in the last plot. In the following we will use the `warp_scale` parameter to avoid such auto-scaling problems.

Subplots. The two plots in Figure 2.5 were created as separate figures. One can also create them as subplots within one figure:

```
plt.figure(4, fgcolor=(.0, .0, .0), bgcolor=(1.0, 1.0, 1.0))
plt.mesh(xv, yv, hv, extent=(0, 0.25, 0, 0.25, 0, 0.25),
         colormap='cool')
plt.outline(plt.mesh(
    xv, yv, hv,
    extent=(0.375, 0.625, 0, 0.25, 0, 0.25),
    colormap='Accent'))
plt.outline(plt.mesh(
    xv, yv, hv, extent=(0.75, 1, 0, 0.25, 0, 0.25),
    colormap='prism'), color=(.5, .5, .5))
```

The result is shown in Figure 2.6. Three separate `mesh` commands are run, each producing a new plot in the current figure. The commands use different values for the `colormap` attribute to color the surface in different ways. When this attribute is not provided, as in the code producing the two first plots in Figure 2.5, a default colormap is used.

The `plt.outline` command is used to create a frame around the subplots, and as seen, we exemplify this possibility for the last two subplots, but not the first one. We see that one of the two frames has a different color, obtained by setting the `color` attribute of the `plt.outline` command.

From the computer code it is hopefully clear that the six values listed in `extent` represent fractions of the cube `(0,1,0,1,0,1)`, where the corresponding plots are placed. The extents for the three plots are here defined such that they do not overlap.

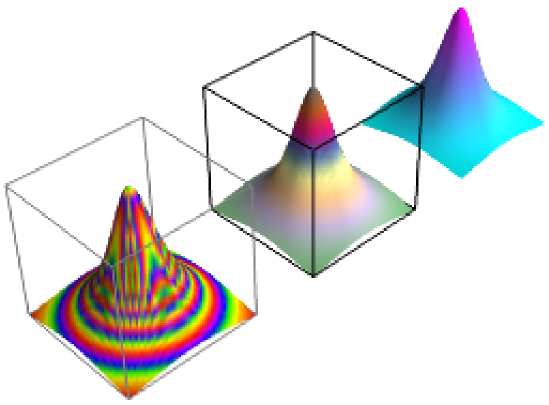

Figure 2.6: A plot with three subplots created with Mayavi.

2.3.2 Contour plots

The following code exemplifies how one can produce contour plots with Mayavi. The code is very similar to that of Matplotlib, but one difference is that the attribute `contours` now can represent the number of levels, as well as the levels themselves. The plots are shown in Figure 2.7.

```
# Default contour plot plotted together with surf.
plt.figure(5, fgcolor=(.0, .0, .0), bgcolor=(1.0, 1.0, 1.0))
plt.surf(xv, yv, hv, warp_scale=0.01)
plt.contour_surf(xv, yv, hv, warp_scale=0.01)

# 10 contour lines (equally spaced contour levels).
plt.figure(6, fgcolor=(.0, .0, .0), bgcolor=(1.0, 1.0, 1.0))
plt.contour_surf(xv, yv, hv, contours=10, warp_scale=0.01)

# 10 contour lines (equally spaced contour levels) together with surf.
# Black color for contour lines.
```

```
plt.figure(7, fgcolor=(.0, .0, .0), bgcolor=(1.0, 1.0, 1.0))
plt.surf(xv, yv, hv, warp_scale=0.01)
plt.contour_surf(xv, yv, hv, contours=10, color=(0., 0., 0.), \
                 warp_scale=0.01)

# Specify the contour levels explicitly as a list.
plt.figure(8, fgcolor=(.0, .0, .0), bgcolor=(1.0, 1.0, 1.0))
levels = [500., 1000., 1500., 2000.]
plt.contour_surf(xv, yv, hv, contours=levels, warp_scale=0.01)

# View the contours by displaying as an image.
plt.figure(9, fgcolor=(.0, .0, .0), bgcolor=(1.0, 1.0, 1.0))
plt.imshow(hv)
```

Note that there is no function in Mayavi which labels the contours.

Contour plots in Mayavi are shown in three-dimensional space, but you can rotate and look at them from above if you want a two-dimensional plot. Their visual appearance may be enhanced by also including the surface plot itself. We have done this for the top and middle left plots in Figure 2.7. There is a clear difference in visual impression between these two plots: in the first one, default surface- and contour coloring is used, resulting in less visible contours, but in the middle left plot (`plt.figure` 6), we set black contours to make them better stand out.

2.3.3 Vector field plots

Mayavi supports only vector fields in three-dimensional space. We will therefore visualize the two-dimensional gradient field (2.3) by adding a third component of zero. The following code plots this gradient field together with the contours of h.

```
plt.figure(11, fgcolor=(.0, .0, .0), bgcolor=(1.0, 1.0, 1.0))
plt.contour_surf(xv, yv, hv, contours=20, warp_scale=0.01)

# mode controls the style how vectors are drawn
# color controls the colors of the vectors
# scale_mode='none' ensures that vectors are drawn with the same length
plt.quiver3d(x2v, y2v, 0.01*h2v, dhdx, dhdy, np.zeros_like(dhdx),
             mode='arrow', color=(1,0,0), scale_mode='none')
```

This will produce a 3D view, which we again can rotate to obtain a 2D view. The result is shown in Figure 2.8, which is similar to Figure 2.3.

2.3.4 A 3D scalar field and its gradient field

Mayavi has functionality for drawing contour surfaces of 3D scalar fields. Let us consider the 3D scalar field

$$g(x, y, z) = z - h(x, y). \tag{2.4}$$

A three-dimensional grid for g can be computed as follows.

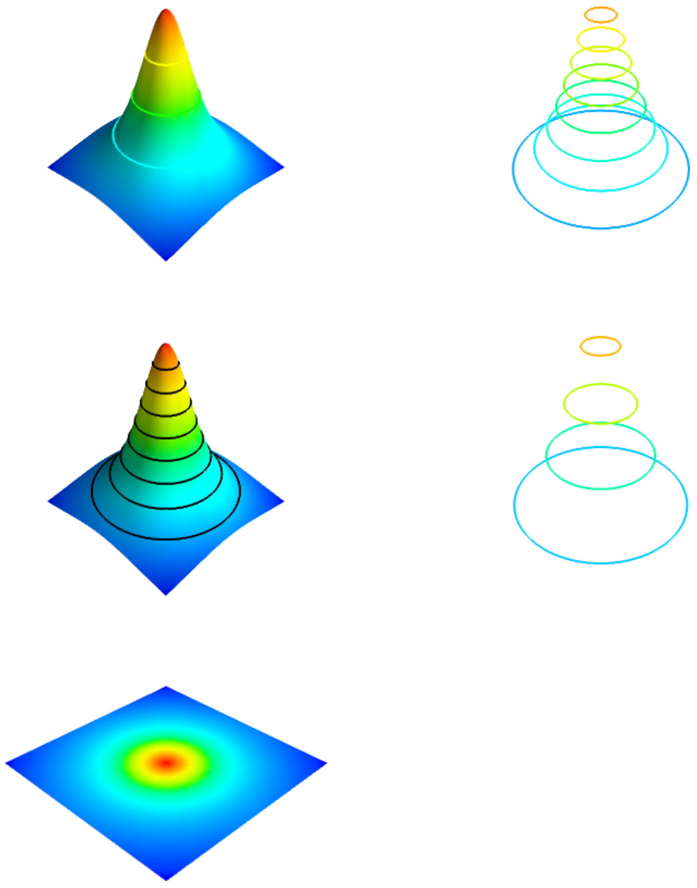
Figure 2.7: Some contour plots with Mayavi.

```
x = y = np.linspace(-10.,10.,41)
z = np.linspace(0, 50, 41)
xv, yv, zv = np.meshgrid(x, y, z, sparse=False, indexing='ij')
hv = 0.01*h0/(1 + (xv**2+yv**2)/(R**2))
gv = zv - hv
```

The contours are now surfaces defined by the implicit equation $g(x, y, z) = C$, corresponding to vertical shifts of the surface $h(x, y)$.

A corresponding vector field can be calculated:

$$\nabla g = \frac{\partial g}{\partial x}\boldsymbol{i} + \frac{\partial g}{\partial y}\boldsymbol{j} + \frac{\partial g}{\partial z}\boldsymbol{k}. \tag{2.5}$$

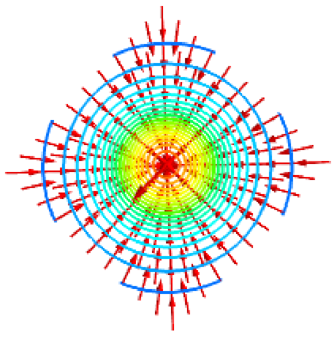

Figure 2.8: Gradient field with contour plot.

`numpy`'s gradient function can be used to compute a gradient vector field in 3D as well, but you need a three-dimensional grid for the field as input. For the field (2.4), the gradient field is computed as follows.

```
x2 = y2 = np.linspace(-10.,10.,5)
z2 = np.linspace(0, 50, 5)
x2v, y2v, z2v = np.meshgrid(x2, y2, z2, indexing='ij', sparse=False)
h2v = 0.01*h0/(1 + (x2v**2 + y2v**2)/(R**2))
g2v = z2v - h2v
dhdx, dhdy, dhdz = np.gradient(g2v)
```

Again we have used a coarser grid for the vector field.

To visualize the field (2.4) and its gradient field together, we draw enough contours, as we did in the 2D case in Figure 2.3. The following code can be used.

```
plt.figure(12, fgcolor=(.0, .0, .0), bgcolor=(1.0, 1.0, 1.0))
# opacity controls how contours are visible through each other
plt.contour3d(xv, yv, zv, gv, contours=7, opacity=0.5)
# scale_mode='none' says that the vectors should not be scaled
plt.quiver3d(x2v, y2v, z2v, dhdx, dhdy, dhdz, mode='arrow',\
             scale_mode='none', opacity=0.5)
```

The result is shown in Figure 2.9.

This example demonstrates some of the challenges in plotting three-dimensional vector fields. The vectors must not be too dense, and not too long. It is inevitable that contours shadow for one another. Fortunately, Mayavi supports an opacity setting, which controls how contours are visible through each other. Visualizing a 3D scalar field is clearly challenging, and we have only touched the subject.

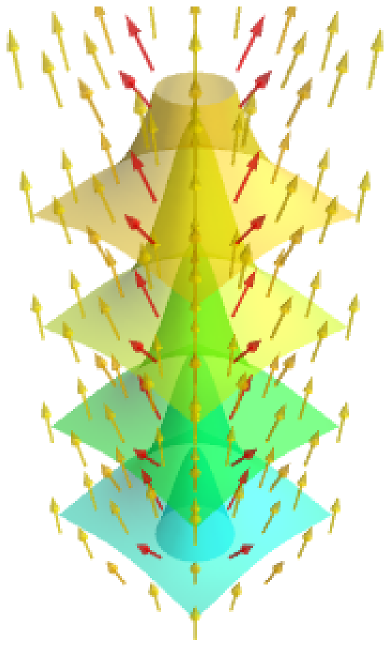

Figure 2.9: The 3D scalar field (2.4) and its gradient field.

2.3.5 Animations

It is straightforward to create animations with Mayavi. In the following code the function $h(x, y)$ is scaled vertically, for different scaling constants between 0 and 1, and each plot is saved in its own file. The files can then be combined to a standard video file.

```
plt.figure(13, fgcolor=(.0, .0, .0), bgcolor=(1.0, 1.0, 1.0))
s = plt.surf(xv, yv, hv, warp_scale=0.01)

for i in range(10):
    # s.mlab_source.scalars is a handle for the values of the surface,
    # and is updated here
    s.mlab_source.scalars = hv*0.1*(i+1)
    plt.savefig('tmp_%04d.png' % i)
```

2.4 Matlab

2.4.1 Surface plots

The Matlab functions for producing surface plots of 2D scalar fields are `mesh` and `surf`. `mesh` produces a wireframe plot, while `surf` colors the faces of the surface. The function `plot3` can be used to plot parameterized curves. The following code uses these functions to plot the surface and the curve.

```
% Simple plot of mountain

figure(1)
mesh(xv, yv, hv);

% Simple plot of mountain and parametric curve
figure(2)
surf(xv, yv, hv);
colormap winter

% add the parametric curve. LineWidth controls the width of the curve
hold on
plot3(curve_x, curve_y, curve_z, 'LineWidth', 20);
```

We now needed to `hold` the plot, in order to place the plots in the same figure. The code also illustrates how one can set a color map, and how we can adjust the thickness of a parametric curve. The corresponding plots are shown in Figure 2.10.

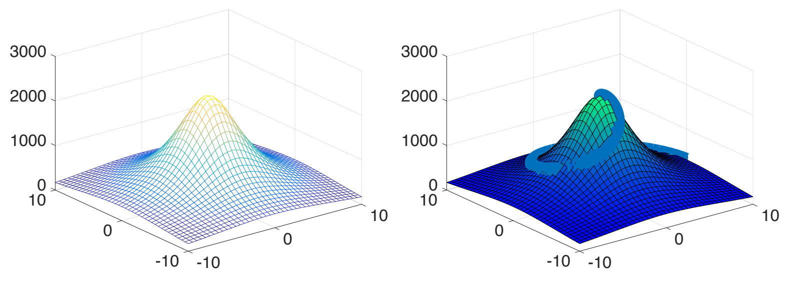

Figure 2.10: Two different plots of a mountain. The right plot also shows a trajectory to the top of the mountain.

2.4.2 Contour plots

The following code exemplifies different ways contour plots can be used to visualize the surface $h(x, y)$.

```
% Default two-dimensional contour plot with 7 colored lines
figure(3)
contour(xv, yv, hv);
axis equal

% Default three-dimensional contour plot
figure(4)
contour3(xv, yv, hv);

% View the contours by displaying as an image
figure(6)
pcolor(hv/max(max(abs(hv))));
```

```
% 10 contour lines (equally spaced contour levels)
figure(7)
contour(xv, yv, hv, 10);
axis equal

% 10 black ('k') contour lines
figure(8)
contour(xv, yv, hv, 10, 'k');
axis equal

% Specify the contour levels explicitly as a list
figure(9)
levels = [500, 1000, 1500, 2000];
contour(xv, yv, hv, levels);
axis equal

% Add labels with the contour level for each contour line
figure(10)
cs = contour(xv, yv, hv);
clabel(cs, 'FontSize',12);
axis equal
```

The first three plots are shown in Figure 2.11. The first two are default 2D- and 3D visualizations. The third displays the surface as an image, whose colors reflect the values of the field, or equivalently, the height of the surface. The next four plots appear in Figure 2.12. They manually set the number of contours, their color, and their levels. The last plot shows ho to label the contours.

2.4.3 Vector field plots

The gradient field (2.3) can be plotted together with the contours as follows

```
figure(11)
quiver(x2v, y2v, dhdx, dhdy, 'color', [1 0 0]);
hold on
contour(xv, yv, hv);
axis equal
```

This resulting plot is shown in Figure 2.13.

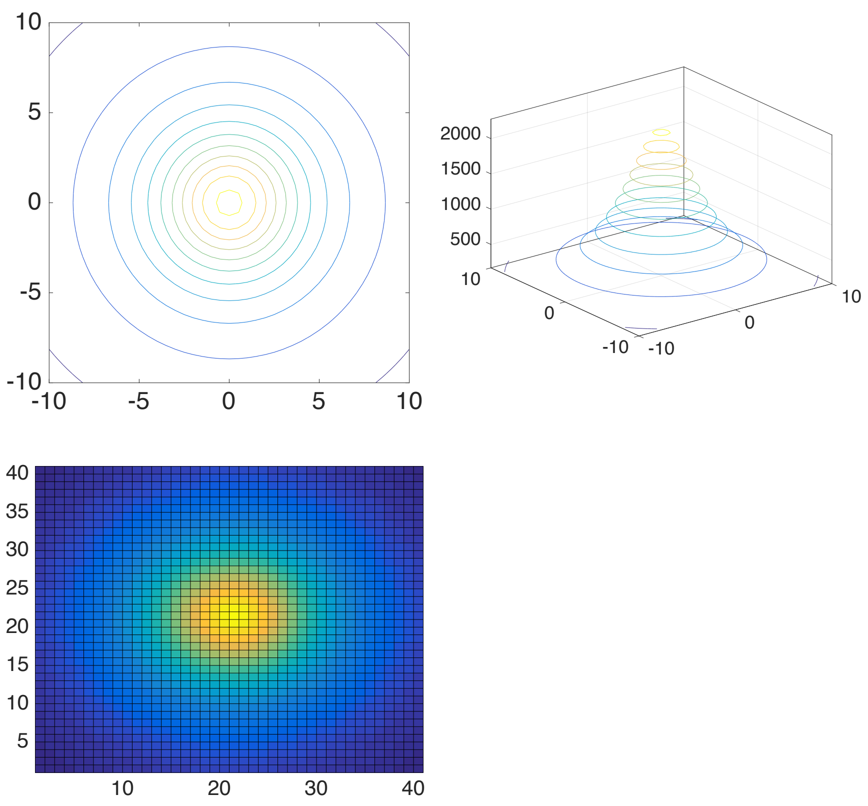

Figure 2.11: Different types of contour plots of a 2D scalar field in two and three dimensions.

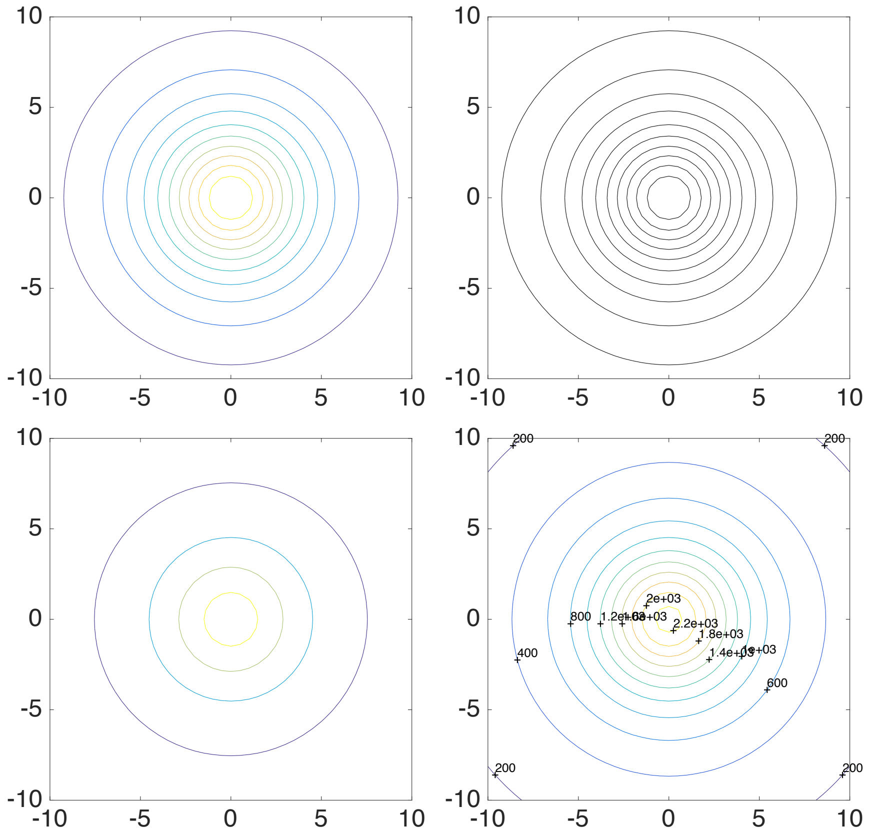

Figure 2.12: Some other contour plots: 10 contour lines (upper left), 10 black contour lines (upper right), specified contour levels (lower left), and labeled levels (lower right).

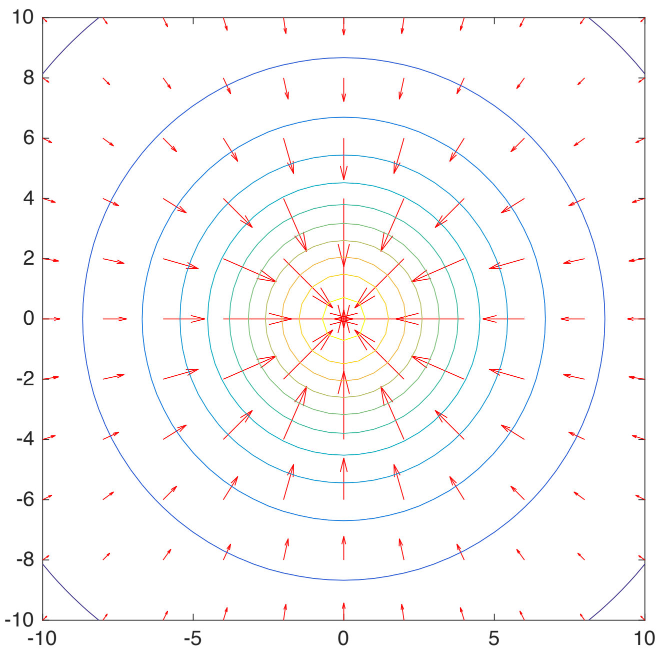

Figure 2.13: Gradient field with contour plot.