2DO:

- Follow Python programming standard, especially the use of spaces

Contents

Chapter 1

Linear algebra

Python has strong suppoert for numerical linear algebra, much like the functionality found in Matlab.

1.1 Basic operations

First we exemplify how to find the inverse and the determinant of a matrix, and how to compute the eigenvalues and eigenvectors:

```
>>> import numpy as np
>>> A = np.array([[2, 0], [0, 5]], dtype=float)

>>> np.linalg.inv(A)  # inverse matrix
array([[ 0.5,  0. ],
       [ 0. ,  0.2]])

>>> np.linalg.det(A)  # determinant
9.9999999999999982

>>> eig_values, eig_vectors = np.linalg.eig(A)
>>> eig_values
array([ 2.,  5.])
>>> eig_vectors
array([[ 1.,  0.],
       [ 0.,  1.]])
```

The eigenvectors are normalized so that they have length 1, which can make them seem more complicated than necessary in some cases.

The `np.dot` function is used for scalar or dot product as well as matrix-vector and matrix-matrix products:

```
>>> a = np.array([4, 0])
>>> b = np.array([0, 1])
>>> np.dot(A, a)         # matrix vector product
array([ 8.,  0.])
>>> np.dot(a, b)         # dot product between vectors
0
>>>
>>> B = np.ones((2, 2))  # 2x2 matrix with 1's
>>> np.dot(A, B)         # matrix-matrix product
array([[ 2.,  2.],
       [ 5.,  5.]])
```

Other typical operations in linear algebra, like the cross product $a \times b$ (between vectors of length 3) and finding the angle between vectors,

$$\theta = \cos^{-1}\left(\frac{a \cdot b}{||a||\,||b||}\right),$$

go like

```
>>> np.cross([1, 1, 1], [0, 0, 1])
array([ 1, -1,  0])

>>> np.arccos(np.dot(a, b)/(np.linalg.norm(a)*np.linalg.norm(b)))
1.5707963267948966
```

Various norms of matrices and vectors are well supported by NumPy. Some common examples are

```
>>> np.linalg.norm(A)        # Frobenius norm for matrices
5.3851648071345037
>>> np.sqrt(np.sum(A**2))    # Frobenius norm, direct formula
5.3851648071345037
>>> np.linalg.norm(a)        # l2 norm for vectors
4.0
```

See `pydoc numpy.linalg.norm` for information on other norms.

The transpose of a matrix `B` is obtained by `B.T`:

```
>>> B = np.array([[1, 2], [3, -4]], dtype=float)
>>> B.T                # the transpose
```

```
array([[ 1.,  3.],
       [ 2., -4.]])
```

The sum of all elements or of the elements in a particular row or column is computed by `np.sum`:

```
>>> np.sum(B)          # sum of all elements
2.0
>>> np.sum(B, axis=0)  # sum over index 0 (rows)
array([ 4., -2.])
>>> np.sum(B, axis=1)  # sum over index 1 (columns)
array([ 3., -1.])
```

The maximum or minimum value of an array is also often needed:

```
>>> np.max(B)          # max over all elements
3.0
>>> B.max()            # max over all elements, alternative syntax
3.0
>>> np.min(B)          # min over all elements
-4.0
>>> B.min()            # min over all elements, alternative syntax
-4.0
>>> np.abs(B).min()    # min absolute value
1.0
```

A very frequent application of computing the minimum absolute value occurs in test functions where we want to verify a result, e.g., that $AA^{-1} = I$, where I is the identity matrix:

```
>>> I = np.eye(2)   # identity matrix of size 2
>>> I
array([[ 1.,  0.],
       [ 0.,  1.]])
>>> np.abs(np.dot(A, np.linalg.inv(A)) - I).max()
0.0
```

It could be tempting to test $AA^{-1} = I$ using the syntax

```
>>> np.dot(A, np.linalg.inv(A)) == np.eye(2)
array([[ True,  True],
       [ True,  True]], dtype=bool)
```

but there are two major problems with this test:

1. the result is a boolean matrix, not suitable for an `if` test
2. using `==` for matrices with float elements may fail because of rounding errors

The second problem must be solved by computing differences and comparing them against small tolerances, as we did above. Here is an example where `==` fails:

```
>>> A = np.array([[4, 0], [0, 49]], dtype=float)
>>> np.dot(A, np.linalg.inv(A)) == np.eye(2)
array([[ True,  True],
       [ True, False]], dtype=bool)
```

(`1.0/49*49` is not exactly `1` because of rounding errors.)

The first problem is solved by using the `C.all()`, which returns one boolean variable `True` if all elements in the boolean array `C` are `True`, otherwise it returns `False`, as in the case above:

```
>>> (np.dot(A, np.linalg.inv(A)) == np.eye(2)).all()
False
```

NumPy can also strip down a matrix to its upper or lower triangular parts:

```
>>> np.triu(B)  # upper triangular part of B
array([[ 1.,  2.],
       [ 0., -4.]])
>>> np.tril(B)  # lower triangular part of B
array([[ 1.,  0.],
       [ 3., -4.]])
```

Indexing an element is done by `B[i,j]`, and a row or column is extracted as

```
>>> A[0,:]  # first row
array([ 2.,  0.])
>>> A[:,1]  # second column
array([ 0.,  5.])
```

NumPy also supports multiple values for the indices via the `np.ix_` function. Here is an example where we grab row 0 and 2, then column 1:

```
>>> C = np.array([[1,2,3],[4,5,6],[7,8,9]])
>>> C[np.ix_([0,2], [1])]  # row 0 and 2, then column 1
array([[2],
       [8]])
```

You can also use the colon-notation to pick out other parts of a matrix. If `C` is a 3×5-matrix,

```
C[1:3, 0:4]
```

gives a submatrix consisting of the two rows of `C` after the first, and the first four columns of `C` (recall that the upper limits, here `3` and `4`, are not included).

1.2 Row operations and Gaussian elimination

The perhaps most frequent operation in linear algebra is the solution of systems of linear algebraic equations: $Ax = b$, where A is a coefficient matrix, b is a given right-hand side vector, and x is the solution vector. The function `np.linalg.solve(A, b)` does the job:

```
>>> A = np.array([[1, 2], [-2, 2.5]])
>>> x = np.array([-1, 1], dtype=float)  # solution
>>> b = np.dot(A, x)                    # right-hand side

>>> np.linalg.solve(A, b)               # will it compute x?
array([-1.,  1.])
```

Implementing Gaussian elimination constitutes a good example on various row and column operations on a matrix. Some needed functionality is

```
A[[i, j]] = A[[j, i]]   # swap rows i and j
A[i] *= k               # multiply row i by a constant k
A[j] += k*A[i]          # add row i, multiplied by k, to row j
```

With these operations, Gaussian elimination is programmed as follows.

hpl 1: Here a new syntax `j:`. Must be explained.

```
m, n = shape(A)
for j in range(n - 1):
    for i in range(j + 1, m):
        A[i, j:] -= (A[i,j]/A[j,j])*A[j, j:]
```

In this code, we first eliminate the entries below the diagonal in the first column, by adding a scaled version of the first row to the other rows, Then the same procedure is applied for the second row, and so on. The result is an upper triangular matrix. The code can fail if some of the entries `A[j, j]` become zero along the way. To avoid this, we can swap rows if the problem arises. The following code implements the idea and will not fail, even if some of the columns are zero.

```
m, n = shape(A)
i = 0
for j in range(n):
    p = argmax(abs(A[i:m, j]))
    if p > 0: # swap rows
        A[[i, p + i]] = A[[p + i, i]]
    if A[i, j] != 0:
        for r in range(i + 1, m):
            A[r, j:] -= (A[r, j]/A[i, j])*A[i, j:]
        i += 1
    if i>m:
        break
```

The rank of a matrix can be computed by first performing Gaussian elimination, and then count the number of pivot columns in the code above:

```
PROVIDE EXAMPLE!
```

A more reliable way to compute the rank is to compute the singular value decomposition of `A`, and check how many of the singular values which are larger than a threshold `epsilon`:

```
PROVIDE EXAMPLE!
```

hpl 3: Didn't see where this info fits well - should have some application since the usefullness is not obvious to readers...

You can use the `np.ix_` notation to permute the rows and columns in a matrix. If you write `C[[2,0,1]]`, the third row is placed first, follows by the first and second rows. If you write `C[:,[2,0,1,4,3]]`, the third column is placed first, followed by the first, second, fifth, and fourth, respectively.

If `A` and `B` are matrices with equally many rows, the matrix where `A` and `B` are placed next to each other or on top of each other can be computed by writing

```
>>> np.hstack([A, B]) # stack two arrays horizontally
array([[ 1. ,  2. ,  1. ,  2. ],
       [-2. ,  2.5,  3. , -4. ]])
>>> np.vstack([A, B]) # stack two arrays vertically
array([[ 1. ,  2. ],
       [-2. ,  2.5],
       [ 1. ,  2. ],
       [ 3. , -4. ]])
>>> np.vstack([a, b]) # stack two vectors vertically
array([[ 4. ,  0. ],
       [ 1. ,  4.5]])
```

The related `np.concatenate` function is used to "append" arrays to each other:

```
>>> np.concatenate([A, B])
array([[ 1. ,  2. ],
       [-2. ,  2.5],
       [ 1. ,  2. ],
       [ 3. , -4. ]])
>>> np.concatenate([a, b])
array([ 4. ,  0. ,  1. ,  4.5])
```

hpl 4: I deleted material on matrix objects and the multiplication operator as there is already a subsection on this topic in the book.

The determinant (include definition, perhaps only in terms of expansion along rows/columns, and only say some things about its applications) is another concept which can be implemented in a smarter way. The following program computes the determinant in a straightforward way using its definition.

hpl 5: This function is recursive - too complicated for the book, I think. We do recursive stuff in chapter 9, though. I suggest we leave this function out.

```
def detdef(A):
    assert A.shape[0] == A.shape[1], 'The matrix must be quadratic!'
    n = A.shape[0]
    if n == 2: # The determinant of a 2x2-matrix is computed directly
        return A[0,0]*A[1,1] - A[0,1]*A[1,0]
    else: # For larger matrix we expand the determinant along column 0
        determinant = 0.0
        for k in xrange(n): # Create sub-matrix by removing column 0, row k.
            submatrix = vstack((A[0:k,1:n],A[k+1:n,1:n]))
            # Multiply with alternating sign
            determinant += (-1)**k * A[k,0] * detdef(submatrix)
        return determinant
```

The `detdef` function is recursive, and calls itself until we have a matrix where the determinant can be computed directly. The central part in the code is where $(n-1)\times(n-1)$-submatrices are constructed. Note that in the code we check that the matrices are quadratic. This code is not particularly fast either. The determinant can also be computed with the built-in method `linlag.det(A)`, and this runs much faster, as the following code verifies.

```
from numpy import *
import time

A=random.rand(9,9)
e0=time.time()
linalg.det(A)
print time.time()-e0
e0=time.time()
detdef(A)
print time.time()-e0
```

Here an arbitrary 9×9-matrix is constructed, and the determinant is computed and timed in the two different ways. The computation times is then written to the display. Run the code and see how much faster the built-in determinant function is! If you want, also try with an arbitrary 10×10-matrix, but then you should be patient while the code executes.

hpl 6: I think it's better to use `timeit` in IPython for such CPU time tests, but we leave the determinant code out.

1.2.1 Symbolic linear algebra

hpl 7: I have a short intro to SymPy in chapter 1 already, much like what you wrote in the SymPy file, so here I think we limit the attention to linear algebra with SymPy.

SymPy supports symbolic computations also for linear algebra operations. We may create a matrix and find its inverse and determinant:

```
>>> import sympy as sym
>>> A = sym.Matrix([[2, 0], [0, 5]])
>>> A**-1    # the inverse
Matrix([
[1/2,   0],
[  0, 1/5]])

>>> A.inv()  # the inverse
Matrix([
[1/2,   0],
[  0, 1/5]])

>>> A.det()  # the determinant
10
```

Note that the entries in the inverse matrix are rational numbers (`sym.Rational` objects to be precise).

Eigenvalues can also be computed exactly:

```
>>> A.eigenvals()
{2: 1, 5: 1}
```

The output is a dictionary (see Chapter 6) **hpl 8**: need exact reference here meaning here that 2 is an eigenvalue with multiplicity 1 and 5 is an eigenvalue with multiplicity 1. We can collect them in a list by

```
>>> e = list(A.eigenvals().keys())
>>> e
[2, 5]
```

Eigenvector computations has a somewhat complicated output:

```
>>> A.eigenvects()
[(2, 1, [Matrix([
[1],
[0]])]), (5, 1, [Matrix([
[0],
[1]])])]
```

The output is a list of three-tuples, one for each eigenvalue and eigenvector. The three-tuple contains the eigenvalue, its multiplicity, and the eigenvector as a `sym.Matrix` object. To isolate the first eigenvector, we can index the list and tuple:

```
>>> v1 = A.eigenvects()[0][2]
>>> v1
Matrix([
[1],
[0]])
```

The vector is a `sym.Matrix` object with two indices. To extract the vector elements in a plain list, we can do this:

```
>>> v1 = [v1[i,0] for i in range(v1.shape[0])]
>>> v1
[1, 0]
```

The following code extracts all eigenvectors as a list of 2-lists:

```
>>> v = [[t[2][0][i,0] for i in range(t[2][0].shape[0])]
         for t in A.eigenvects()]
>>> v
[[1, 0], [0, 1]]
```

The norm of a matrix or vector is an exact expression:

```
>>> A.norm()
sqrt(29)
>>> a = sym.Matrix([1, 2])
>>> a
Matrix([
[1],
[2]])
>>> a.norm()
sqrt(5)
```

The matrix-vector product and the dot product between vectors are done like this:

```
>>> A*a
Matrix([
[ 2],
[10]])
```

```
>>> b = sym.Matrix([2, -1])
>>> a.dot(b)
0
```

Solving linear systems exactly is also possible:

```
>>> x = sym.Matrix([-1, 1])/2
>>> x
Matrix([
[-1/2],
[ 1/2]])
>>> b = A*x
>>> x = A.LUsolve(b)  # does it compute x?
>>> x                 # x is a matrix object
Matrix([
[-1/2],
[ 1/2]])
```

Sometimes one wants to convert `x` to a plain `numpy` array with `float` values:

```
>>> x = np.array([float(x[i,0].evalf()) for i in range(x.shape[0])])
>>> x
array([-0.5,  0.5])
```

Exact row operations can be done as exemplified here:

```
>>> A[1,:] + 2*A[0,:]  # [0,5] + 2*[2,0]
Matrix([[4, 5]])
```

We refer to the online SymPy linear algebra tutorial for more information.

1.3 Exercises

Exercise 1: Verify linear algebra results

When we want to verify that a mathematical result is true, we often generate matritces or vectors with random elements and show that the result holds for these "arbitrary" mathematical objects. As an example, consider testing that $A + B = B + A$ for matrices A and B:

```
def test_addition():
    n = 4  # matrix size
    A = matrix(random.rand(n, n))
    B = matrix(random.rand(n, n))

    tol = 1E-14
    result1 = A + B
    result2 = B + A
    assert abs(result1 - result2).max() < tol
```

Use this technique to write test functions for the following mathematical results:

hpl 9: List a set of linear algebra results here to be verified computationally!

Filename: `verify_linalg`.

Chapter 2

Plotting in three dimensions

We will explain plotting in three dimensions using three different packages. As with plotting in chapter 5 we will go through Matplotlib and SciTools. Then we will go through Mayavi, which contain support for doing more advanced plotting operations.

The three packages are different in many respects, but there are some common denominators. First of all, they all support rotation of a figure by holding the mouse cursor down. This functionality is often useful in order to get a better view of a surface.

2.1 Matplotlib

The code below assumes that we have imported Matplotlib as in chapter 5, i.e. that we use the prefix `plt` for Matplotlib code:

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

Note that after each plot command you need to write `plt.show()` in order for the new plot to become visible.

2.1.1 Simple surface plots

Let us start by plotting a function over an interval in the xy-plane. As our function we will use

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

This formula describes the height of an isolated mountain, where h is the height above sea level, h_0 is the height at the top of the mountain, and R describes the radius of the mountain. we will measure h and h_0 in meters, and R in kilometers. Prior to the code below we have set $h_0 = 2277$ and $R = 4$. We first define a grid for the area closer to the mountain top than 10km in any direction.

```
t = np.linspace(-10., 10., 41)
x, y = np.meshgrid(t, t, indexing = 'ij', sparse = False)
```

Note the mysterious extra parameters to `meshgrid` here, which are needed in order for the coordinate axes to have the order we would expect in mathematics. `indexing = 'ij'` means that the coordinates are set as in a *xy*-coordinate system. `indexing = 'xy'` means that the coordinates are set as row/column indices in a matrix.

Now that we have the grid coordinates, we need to create the corresponding function values. This can be done as follows

```
h = h0/(1 + (x**2 + y**2)/(R**2))
```

The surface can now be plotted as follows:

```
fig = plt.figure()
ax = fig.gca(projection = '3d')
ax.plot_surface(x, y, h)
```

The result is shown in the left plot in Figure 2.1.

Figure 2.1: Two Matplotlib surface plots. Plot without colours (left), and a plot with colours (right) shown together with a parametric curve which is a trajectory to the top of the mountain.

Coloring can be used to get a better view of surfaces. We can use the `cm`-module to obtain a color map which can be used to color the surface.

```
from matplotlib import cm
```

This can then be passed to the `plot_surface` function above as follows:

```
ax.plot_surface(x, y, h, cmap = cm.coolwarm)
```

The result is shown in the right plot in Figure 2.1. One can choose from a wide range of different color maps.

Several curves and surface can be included in the same plot. The previously produced surface had been combined with a plot of a parametrized curve. This curve lies on the surface, which moves with constant angular and radial velocity towards the top of the mountain (i.e. it represents a climb to the top of the mountain). This curve can be produced as follows

```
t = np.linspace(0, 2*np.pi, 100)
xcoords = 10*(1 - t/(2*np.pi))*np.cos(t)
ycoords = 10*(1 - t/(2*np.pi))*np.sin(t)
zcoords = h0/(1 + 100*(1 - t/(2*np.pi))**2/(R**2))
```

and the following command plots the curve

```
ax.plot(xcoords, ycoords, zcoords)
```

2.1.2 Contour plots

In order to create a simple contour plot we can write the following

```
plt.contour(x, y, h)
```

The resulting plot is shown in Figure 2.2

By adding a parameter we can specify how many contour levels to plot. The contour levels will be automatically chosen.

```
plt.contour(x, y, h, 10)
```

The result is shown in the upper left plot of Figure 2.3.

Normally the level curves are drawn with different colours. This is very useful on the display, but is less practical if you want to paste the figure into a black and white document. If you write

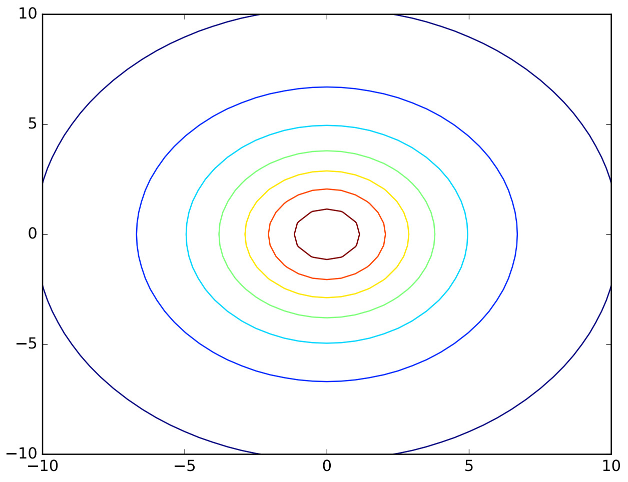

Figure 2.2: A simple contour plot with Matplotlib.

Figure 2.3: Some other contour plots with Matplotlib: With 10 levels (upper left), with 10 levels in black (upper right), with specified levels (lower left), and with labeled levels (lower right).

```
plt.contour(x, y, h, 10, colors = 'k')
```

The result is shown in the upper right plot of Figure 2.3. All level curves will be drawn in black ('k' is the symbol for black).

The automatically chosen levels above may not be the most interesting level curves, or may not be the levels we are interested in. We can specify which levels to include in the contour plot as follows.

```
levels = [500., 1000., 1500., 2000.]
plt.contour(x, y, h, levels = levels)
```

The result is shown in the lower left plot of Figure 2.3

The levels in a contour plot are not labeled with the corresponding values by default. To include a default kind of labeling of the levels, one can call the `clabel` function.

```
cs = plt.contour(x, y, h)
plt.clabel(cs)
```

The result is shown in the lower right plot in Figure 2.3

2.1.3 Plotting vector fields

Let us attempt to plot the vector field

$$\boldsymbol{v} = \left(x^2 + 2y - \frac{1}{2}\,xy\right)\boldsymbol{i} - 3y\boldsymbol{j}.$$

If we want to plot this over $[-5, 5]$, with 0.5 distance between the vectors, we first need to create a two-dimensional grid as follows.

```
t = np.linspace(-5, 5, 11)
x, y = np.meshgrid(t, t, sparse = False, indexing = 'ij')
vx = x**2 + 2*y - .5*x*y
vy = -3*y
```

The following code

```
plt.quiver(x, y, vx, vy, angles = 'xy', scale_units = 'xy', color = 'b')
```

plots the vector field. The two first parameters specify the x- and y-components of the points where vectors are to be drawn, and the next two parameters the actual components of the vectors. The result is shown in the left plot in Figure 2.4

Figure 2.4: A vector field plotted with Matplotlib (left), and contours together with the gradients (right).

The parameters `angles` and `scale_units` say that the vector field should be drawn as a gradient field, i.e. that the length of the vectors coincide with that of the vector $\boldsymbol{v}$. `color` indicates the colour to be used for the vectors (here blue). Often the vectors need to be scaled in order for them not to collide in the plot (alternatively one can have some more distance between the points in the arrays `x` and `y`. This can be done manually by using the `scale` parameter. Increasing this results in shorter vectors. Matplotlib will attempt to scale the plot automatically if the `scale` parameter is not set, in order to try to avoid that the vectors collide.

Let us round off by considering an example where we plot the contour lines of a function together with the gradient field of that function. The gradients are orthogonal to the contour lines, so let us see if we can observe this in our plot...

We then compute the gradient using the built-in function `gradient` in `numpy`.

```
dhx, dhy = np.gradient(hh)         # Beregn gradientvektoren (dh/dx,dh/dy)
```

Finally we plot the vector field together with the contour lines in the same plot as follows.

```
plt.quiver(xx, yy, dhy, dhx, color = 'r', angles = 'xy')

t = np.linspace(-10.,10.,21)
x, y = np.meshgrid(t, t, indexing = 'ij', sparse = False)
h = h0/(1+(x**2+y**2)/(R**2))
plt.hold('on')
plt.contour(x, y, h)
```

Here we have created a new grid which is finer, for plotting the surface. The result is shown in the right plot in Figure 2.4. Note that in the call to `quiver` the vector components `dhy` and `dhx` were reversed. This seems to be an error in Matplotlib.

2.2 SciTools

The code below assumes that we have imported SciTools as in chapter 5:

```
from scitools.easyviz import *
```

TODO 10: Curve plots in 3D using SciTools

2.2.1 Simple surface plots

SciTools has the function `ndgrid` which, just as `meshgrid`, can be used to create grids of all possible (x, y)-coordinate pairs from given one-dimensional coordinates. One can simply replace the call to `meshgrid` above with

```
x, y = ndgrid(tx, ty)
```

`ndgrid` is more flexible than `meshgrid` in the sense that it can be used to make three-dimensional grids also. Also, we see that the extra parameters `indexing` and `sparse` do not need to be set: They attain default values which can be used.

The surface can now be plotted simply by writing `mesh(X, Y, Z)`, so that this part of the code is shorter than the three lines needed with Matplotlib. The result is shown in the left plot in Figure 2.5.

Figure 2.5: Two SciTools surface plots. Simple plot (left), and a plot with colours (right).

We can write `surf(x, y, h)` instead of `mesh(x, y, h)` to get a graph where the surface elements have been coloured. The result is shown in the right plot in Figure 2.5. One can choose from a wide range of different color maps.

2.2.2 Contour plots

In order to create a simple contour plot we can write the following

```
contour(x, y, Z)
```

The resulting plot is shown in Figure 2.6

Figure 2.6: A simple SciTools contour plot.

By adding a parameter we can specify how many contour levels to plot. The contour levels will be automatically chosen.

```
contour(x, y, h, 10)
```

The result is shown in the upper left plot in Figure 2.7.

Figure 2.7: Some other SciTools contour plots: With 10 levels (upper left), with 10 levels in black (upper right), with specified levels (lower left), and with labeled levels (lower right).

Normally the level curves are drawn with different colours. This is very useful on the display, but is less practical if you want to paste the figure into a black and white document. If you write

```
contour(x, y, h, 10, 'k')
```

all level curves will be drawn in black ('k' is the symbol for black). The result is shown in the upper right plot in Figure 2.7.

The automatically chosen levels above may not be the most interesting level curves, or may not be the levels we are interested in. We can specify which levels to include in the contour plot as follows.

```
levels = [500., 1000., 1500., 2000.]
contour(x, y, h, levels = levels)
```

The result is shown in the lower left plot in Figure 2.7.

The levels in a contour plot are not labeled with the corresponding values by default. To include a default kind of labeling of the levels, one can use a parameter called `clabels`.

```
contour(x, y, h, clabels = 'on')
```

The result is shown in the lower right plot in Figure 2.7.

2.2.3 Plotting vector fields

Let ur return to the vector field

$$\boldsymbol{v} = \left(x^2 + 2y - \frac{1}{2}\,xy\right)\boldsymbol{i} - 3y\boldsymbol{j},$$

and let us reuse the code above to produce the vectors in the vector field. With SciTools one can instead write

```
quiver(x, y, vx, vy, 200, 'b')
```

to plot the vector field. The two last parameters represent scaling and color. The resulting plot is shown in Figure 2.8

Figure 2.8: A vector field plotted with SciTools (left), and contours together with the gradients (right).

Let us round off by considering again the previous example of contour/gradient plot. We again define a grid for the area closer to the mountain top than 10km in any direction, but this time we use `ndgrid` in SciTools.

```
tt = linspace(-10., 10., 11)
xx, yy = ndgrid(tt, tt)
hh = h0/(1 + (xx**2 + yy**2)/(R**2))
```

and we compute the gradient as before. Finally we plot the vector field together with the contour lines in the same plot as follows.

```
quiver(xx, yy, dhx, dhy, 0, 'r')

t = linspace(-10., 10., 21)
x,y = ndgrid(t,t)
h = h0/(1+(x**2+y**2)/(R**2))
hold('on')
contour(x, y, h, daspectmode = 'equal')
```

The result is shown in the right plot in Figure 2.8.

2.3 Mayavi

Mayavi[1] is an advanced, free, easy to use, scientific data visualizer, with an emphasis on 3D visualization. It is written in Python, and uses the Visualization Toolkit (VTK) for the graphics. It is cross platform and should run on any platform where both Python and VTK are available (this includes Mac OSX and Windows). There are several ways you can obtain a full environment for Mayavi.

We recommend that you use Anaconda for Mayavi, since Anaconda is cross-platform. This requires that you install a list of packages which Mayavi depends on[2] These are not only Python packages (which thus can be installed with `pip`), but also other types of dependencies (which can be installed with `conda`, a package managament tool which handles dependencies which are outside Python, contrary to `pip`. `conda`, just as `pip`, is bundled with your Anaconda installation).

You can install Canopy[3]. Canopy contains a full Python environment, and through the package manager tool of Canopy you can install Mayavi itself, VTK, and all the other dependencies of Mayavi. The package manager tool gives you

[1]The page `http://docs.enthought.com/mayavi/mayavi/` collects pointers to all relevant documentation or Mayavi

[2]A list of the packages you need can be found on `http://docs.enthought.com/mayavi/mayavi/installation.html`

[3]`https://www.enthought.com/products/canopy/`

access to install not only Mayavi, but also a whole list of packages from what is called a package index.

You can also use Ubuntu and the VMWare Fusion virtual machine, and install all components by hand (see the link for Anaconda to package dependencies).

We will only consider the Mayavi-functionality for simple plotting[4], such as the functionality we considered for Matplotlib and SciTools. The `mayavi.mlab` module provides a simple interface to such plotting, with an emphasis on 3D visualization. The following code will assume that we have imported everything from this module, and that we use the `ml` prefix for this module:

```
import mayavi.mlab as ml
```

Several of the functions in `mlab` have the same name as the counterparts in SciTools and Matplotlib, so it is smart to separate them using prefixes.

2.3.1 The basics

We first need to cover some of the basic figure handling function in the `mayavi.mlab` module[5]. Much of this is similar to the bascis for Matplotlib and SciTools, and we will therefore try to focus mainly on the differences.

The plotting commands you do in `mlab` will go to the current figure, also called a scene. This means that several plot commands will be shown together. You can work with several figure instances simultaneously. A figure can be accessed in three different ways:

1. as a `Figure` object representing it,
2. by a name representing it,
3. by a number representing the index of the plot

When either a Figure object, a name or an integer is supplied as parameter to the function `figure()`, `mlab` sets the current figure to be that particular Figure object, or the Figure object with that name or index. If no figure existed with the given name/index, a new figure object will be created with that name/index, and set as the current figure. **TODO 11**: Include this more general description of the usage of figure() in chapter 5 instead? This is a more natural place for this, as it is nice to know how to jump between a set of figures.

In the same way, the function `close()` closes a figure with the given handle, name, or index. To clear (not close) the current figure before the next plot, we

[4]The documentation for simple plotting functions can also be found on `http://docs.enthought.com/mayavi/mayavi/auto/mlab_helper_functions.html`. Further documentation can be found on `http://docs.enthought.com/mayavi/mayavi/auto/mlab_other_functions.html`.

[5]The documentation for basic figure handling functions can be found on `http://docs.enthought.com/mayavi/mayavi/auto/mlab_figure.html`

can run the command `clf()`. The function `gcf()` returns the current figure object. You will need to call this in order to get the object where you can set some the figure properties, such as the foreground or background colour (the foreground colour is used for text and labels included in the plot). This can also be done directly by writing

```
ml.figure(fgcolor = (.0, .0, .0), bgcolor = (1.0, 1.0, 1.0))
```

Here colours are represented as RGB-values, where the components are floating points between zero and one. In the above code the foreground and background colour for the current figure is set to black and white, respectively.

Many of the other functions are also similar to what we are used to.

- `ml.savefig(filename)` saves the current figure to file. The type of the file (png, jpeg, etc) is deduced fom the file extension
- `ml.show()` makes the current figure visible (it may be that the environment does this by itself).
- `ml.title()` to decorate the current figure with a title.
- `ml.axes()` for creating axes with labels and given colors, we can use the function axes(xlabel = 'x', ylabel = 'y')

The following code examplifies all the above in one plot.

```
ml.clf() # Clear
# Create figure with white background, black foreground
ml.figure(fgcolor = (.0, .0, .0), bgcolor = (1.0, 1.0, 1.0))
... Create and plot your figure
ml.title('My first plot')
ml.axes(xlabel = 'x', ylabel = 'y', zlabel = 'z', nb_labels = 5, color = (0., 0., 0.))
ml.show()
ml.savefig('plot.png')
```

2.3.2 Simple surface plots and subplots

Let us return to plotting the surface $h(x, y)$. Mayavi also has commands **`mesh`** and **`surf`**, which works similarly to Matplotlib and SciTools:

```
ml.mesh(x, y, h)
ml.surf(x, y, h)
```

Both functions color the surface, and accept many of the same parameters. For some purposes `surf` seems to be more general. The left plot of figure 2.9 shows the result when `mesh` is used.

Figure 2.9: Two Mayavi surface plots. Using `mesh` (left), and using `surf` (right). The right plot also shows our parametrized curve.

If the magnitudes in the vertical and horizontal directions are very different, the plots above will whow a plot which is very concentrated in one direction. The reason is that Mayavi does no auto-scaling of the axes per default (corresponding to `axis("equal")` in Matplotlib and Matlab). To ensure that the axes are auto-scaled to fit the contents of the surface one can include the parameter `extent` in the following way

```
ml.surf(x, y, h, extent = (0, 1, 0, 1, 0, 1))
```

Below we will return to what the six values in `extent` mean. Note that the behaviour may be unexpected when combining different plots which both have been auto-scaled. To illustrate this let us also autoscale our parametrized curve, obtained by climbing the mountain so that the angular speed is constant. For this we can use the `plot3d` command as folllows.

```
ml.plot3d(xcoords, ycoords, zcoords, tube_radius = 0.2)
```

The additional parameter `tube_radius` controls the width of the parametrized curve, and was increased here in order to make the curve more visible when shown together with `surf`. The right plot of figure 2.9 shows the result when both `surf` and `plot3d` are combined in the same plot. We see that the curce does not lie on the surface, as it should. The reason is that the ranges of the x-, y-, and z-values differ for the the curve and the surface, so that autoscaling will use different scales when placing them on the scene. We therefore have to be a bit cautious when applying the autoscaling in Mayavi, and in the following contour plots we will avoid this autoscaling problem by scaling everything first so that the coordinates are of comparable magnitude, so that there is no need for autoscaling in the first case.

One can also use other properties to scale the cases. Some of the plotting commands below accept a `warp_scale` parameter, which tells us how the the vertical scaling should be performed.

The two plots in figure 2.9 were created as separate figures. One can also create them as subplots within one figure, so that one figure is generated with several subplots. Consider the following code.

```
ml.outline(ml.mesh(x, y, h, extent = (0, 0.25, 0, 0.25, 0, 0.25), \\
          color = (.5, .5, .5)))
ml.outline(ml.mesh(x, y, h, extent = (0.375, 0.625, 0, 0.25, 0, 0.25), \\
          colormap = 'Accent'))
ml.outline(ml.mesh(x, y, h, extent = (0.75, 1, 0, 0.25, 0, 0.25), \\
          colormap = 'prism'))
```

The result is shown in the left plot in Figure 2.10.

Figure 2.10: A plot with three subplots created with Mayavi, consisting of the same surface drawn with different color maps.

From this it should be 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 defined so that they do not overlap. Three separate `mesh` commands are run, each producing a new plot in the current figure. The handles to the three plots are stored in the variables `surf1`, `surf2`, and `surf3`, and the function `outline` finally draws the corresponding box for the plot in question. We have dropped the axes and the title in the code here.

Note that the surfaces in Figure 2.10 are drawn with different colors. This is due to the `colormap` and `color` attributes in the call to `mesh`, which adjusts the colors when the surface is drawn. The `color` attribute, which we see also can be used as a parameter to `outline`, adjusts the surface so that it is colored with small variations from the provided base-color. One can choose from a wide range of different color maps.

2.3.3 Contour plots

Let us also duplicate the previous contour plots, using Mayavi instead of Matplotlib and SciTools. Contour plots with Mayavi are shown in 3D, contrary to Matplotlib and SciTools. Their visual appearance may be enhanced by also including the surface plot itself. Below we have done this for two of the contour plots. For comparison, we have not included the surface itself for the two last contour plots. There is a clear difference in visual impression between the two, and we see that the colour of the contours can conflict with the colour of the surface.

In order to create a simple contour plot we can write the following

```
ml.contour_surf(x, y, h)
```

The result is shown in the upper left plot in Figure 2.11.

Figure 2.11: Some contour plots created with Mayavi: Simple plot (upper left), with 10 levels (upper right), with 10 levels in black (lower left), and with specified levels (lower right).

By adding a parameter we can specify how many contour levels to plot. The contour levels will be automatically chosen by the application.

```
ml.contour_surf(x, y, h, contours = 10)
```

The result is shown in the upper right plot of Figure 2.11.

Normally the level curves are drawn with different colours. This is very useful on the display, but is less practical if you want to paste the figure into a black and white document. If you write

```
ml.contour_surf(x, y, h, contours = 10, color = (0., 0., 0.))
```

The result is shown in the lower left plot of Figure 2.11. All level curves will be drawn in black, as described by the components in the color `(0., 0., 0.)`.

The automatically chosen levels above may not be the most interesting level curves, or may not be the levels we are interested in. We can specify which levels to include in the contour plot as follows.

```
levels = [500., 1000., 1500., 2000.]
ml.contour_surf(x, y, h, contours = levels)
```

The result is shown in the lower right plot of Figure 2.11

TODO 12: I can't find out out how to set label text as in clabel

2.3.4 Plotting vector fields in 3D

Mayavis functionality for plotting vector fields is primarily intended for 3D also. As an example, let us attempt to plot a gravitational field. Newtons law of gravitation says that the gravitational force is proportional to

$$-\boldsymbol{r}/|\boldsymbol{r}|^3 = -(x,y,z)/\sqrt{x^2+y^2+z^2}^3,$$

where $\boldsymbol{r} = (x,y,z)$ is the position vector. Let us attempt to plot this vector field. We need a three-dimensional grid, so that `meshgrid` will not work. We can either use `ndgrid` from SciTools, or `mgrid` from `numpy` as follows If we want to plot this over $[-5, 5]$, with 0.5 distance between the vectors, we first need to create a two-dimensional grid as follows.

```
x, y, z = np.mgrid[.5:2:.2, .5:2:.2, .5:2:.2]
r3 = np.sqrt(x**2 + y**2 + z**2)**3
vx = -x/r3
vy = -y/r3
vz = -z/r3
```

The cube of the distance (`r3`) has here been computed once, since all three components are divided by this. Also we compute the vector field for a section in the first octant only, where all three components are between 0.5 and 2, and we do not make the grid too dense, so that individual vectors have smaller chance of colliding. The vector field can now be vizualized as follows.

```
ml.figure(fgcolor = (.0, .0, .0), bgcolor = (1.0, 1.0, 1.0))
ml.quiver3d(x, y, z, -x/r3, -y/r3, -z/r3, \\
            mode = 'arrow', colormap = 'jet', scale_factor = .5)
ml.axes(xlabel = 'x', ylabel = 'y', zlabel = 'z', \\
        nb_labels = 5, color = (0., 0., 0.))
```

The three first parameters to `quiver3d`, the function which actually plots the vector field, specify the x-, y-, and z-components of the points where vectors are to be drawn. The next three parameters are the actual components of the vectors. To improve the visualization of the field, we have set some of the optional properties to `quiver3d`:

- The `mode` parameter ensures that the vectors are drawn as arrow.
- The `colormap` parameter controls how the vectors are colored.
- The `scale_factor` parameter lets us manually scale the vector lengths, to ensure that they do not collide.

In addition we have set a foreground and background color for the figure, and added axes as above. The resulting plot is shown in Figure 2.12. This shows some of the challenges in plotting vecto fiels in 3D dimensions. It may be challenging to create instructive plots, since vectors are drawn over an area in 3D rather than 2D.

Figure 2.12: A vector field plotted with Mayavi.

2.3.5 Animations

We have previously seen how to produce animations with Matplotlib and SciTools. With the function `animate` in `mlab` we can also create animations.

This code will rotate the camera continuously. f is the current graphics.

```
from mayavi import mlab
@mlab.animate
def anim():
    f = mlab.gcf()
    while 1:
        f.scene.camera.azimuth(10)
        f.scene.render()
        yield

a = anim()
```