Anders Malthe-Sørenssen

Introduction to mechanics

Integrating numerical and analytical methods

Preface

This compendium is intended as a support for teaching in preparatory course in computer programming for students in the MENA and LEP programs, which usually do not have prior knowledge of programming from INF1100 course. The aim of the course is to provide a good practical background to preform calculations in Python and MATLAB relevant to FYS-MEK1110.

Most of the text in this compendium is taken from chapter 2 in Anders Malthe-Sørenssen's book: "Introduction to mechanics-Integrating numerical and analytical methods", but has been slightly modified by Svenn-Arne Dragly and Milad H. Mobarhan. The modifications made includes rewriting MATLAB codes to Python and including the section "Terminal basics".

One essential point is to notice that you learn programming by doing it, not by just reading how to do it. Therefore is it important that you spend a lot of time to write your own programs, which of course don't need to be complex. You will find some basic exercises in this compendium, which we recommend you to do strongly. In addition a more extensive exercise is included which is based on the FYS-MEK1110 syllabus. If you are interested to do more exercises, we recommend you to have look in the textbook used in INF1100. A more detailed compendium about programming , with emphasis on Python, for MENA and LEP students can be find at `http://folk.uio.no/masan/INFForkurs/`.

We hope that this compendium can help you to get an overview in basic programming and make the programming which comes later in FYS-MEK1110 more interesting and affordable. We will also thank Anders Malthe-Sørenssen for lending lots of material.

Svenn-Arne Dragly & Milad H. Mobarhan

Contents

Chapter 1

Getting started with programming

Our approach is to use programming techniques and tools to study physics. You will therefore need to know a few programming basics in order to profit from this approach. However, if you do not have a relevant background in introductory scientific programming, do not despair. Experience shows that you can learn to program through your first physics course – many students have done this successfully and with good results. In order to prepare you for the main text, this chapter provides an introduction to programming.

1.1 Terminal basics

While you may not have to use a terminal at all while programming, it is an awesome place to get down and dirty with the concepts of programming. On the machines in the computer lab, select Applications → Utilities → Terminal to open up a terminal. On Mac OS X you need to look for the Terminal application in the Applications folder. In a terminal window one can write commands, and the line following the command shows the output. As an example if you type:

```
date
```

and press enter, the output is the date and time. It is necessary to know some few terminal commands which are useful to know:

- The current directory, also known as working directory, is where you are, which is equivalent to having a window open and viewing the files. In terminal you can find your working directory by typing:

  ```
  pwd
  ```

- To list the files and directories at your current working directory, type

  ```
  ls
  ```

  which stands for “list files“.

- In order to make a new folder i your current directory, you can type "mkdir" (make directory) followed by the name you choose for the new folder, in this case "test"

  ```
  mkdir test
  ```

- A folder can be removed using "rmdir" (remove directory) if the folder is empty, else use "rm -r":

  ```
  rmdir test
  ```

- You can go to another directory with "cd" (change directory). If for example your current directory is "dir1" and a folder name "dir2" is in this directory and you want to go to "dir2" directory, you can type

  ```
  cd dir2
  ```

  If you want go back to "dir1" from "dir2" you simply type:

```
cd ..
```

- An empty file is created by typing:

  ```
  touch file
  ```

  In this case a new file with "file" is created in the working directory. Note that if you want to make a specific type of file you can just add extension to the name; .tex,.py, .m etc..

- A file can be removed using "rm" (remove):

  ```
  rm name_of_file
  ```

- In order to move files from one directory to another, type

  ```
  mv name_of_file destination
  ```

  In this case "destination" should be changed with destination's path. For example if your working directory is "dir1", containing a directory "dir2" and a file "test", and you want to move "test" to "dir2", you should type:

  ```
  mv test dir2
  ```

- To copy files, type:

  ```
  cp name_of_file destination
  ```

  To copy folders you need to type "cp -r".

These examples illustrate that when we type a command it is executed right after. If we want to to execute several commands after each other, it is natural to join them to a program. We will have closer look to this in next section.

1.2 A Python calculator

We are in this text using the editor Spyder to work with Python. When you start Spyder, you get a window (figure **??**) where you can type commands to be executed. Click on the `Console window` in the lower right corner and type:

```
>>> 9*4
```

(The `>` characters in front of the command indicates that you are supposed to type this into a Python terminal, and not the regular terminal.) Press enter, and the following will show up:

```
36
```

Notice the difference between the text *you* type, which is preceded by `>>`, and the results generated by the program, which in this text are typeset in red.

Python can be used as an advanced calculator by typing expression on the command line:

```
>> 3*2**3+4
```

```
28
```

Standard operators are plus (`+`), minus (`-`), multiplication (`*`), division (`/`), and power (`**`). Powers of ten are input using `e`:

```
>>> 4.5e4
```

```
45000
```

```
>>> 2.5e-10
```

```
2.5e-10
```

which also shows how Python displays numbers.

Python has most mathematical functions and constants built in, such as `pi`, `cos`, `sin`, and `exp` To use them, we need to load the `pylab` module first:

```
>>> from pylab import *
```

After this, we are ready to use mathematical functions:

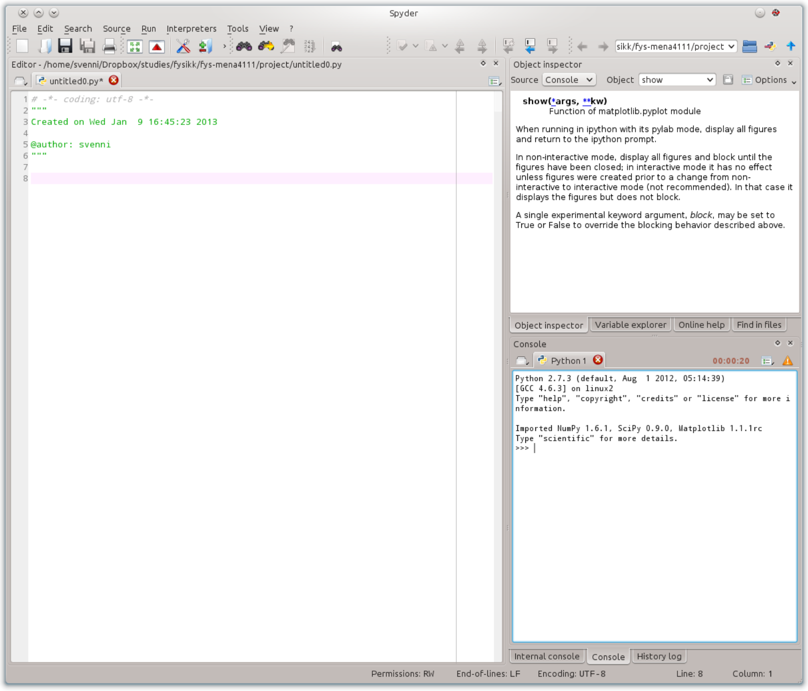

Figure 1.1: *Window appearing when you start Spyder. Start typing into the command window in the lower right corner.*

```
>>> 4*pi
```

```
12.566370614359172
```

Python uses radians for the trigonometric functions:

```
>>> sin(pi/6)
```

```
0.49999999999999994
```

As you can see, Python does not always round off as you may expect, even though the answer is close to the exact answer ($\sin(\pi/6) = 0.5$). You can find a list of useful syntax, functions and expressions in the summary 1.8.

Python becomes more useful when you have a formula you want to use. For example, you may want to use the formula:

$$T_F = \frac{9}{5} T_C + 32 \ , \tag{1.1}$$

to find the temperature, T_F, in Fahrenheit, given the temperature T_C in centigrade. We may type this formula directly into Python.

```
>>> TF = 9./5.*TC + 32.
```

If you type in the above formula and press enter, you will we bump into the following error:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'TC' is not defined
```

Ooops. That did not work, because Python does not yet know the value of T_C. We give T_C a value and retype the formula:

```
>>> TC = 40.
```

```
>>> TF = 9./5.*TC + 32.
```

To see the answer, type

```
>>> print TF
```

```
104.0
```

Note one important thing here. We are typing periods after each number. This is because Python differs between integers $(1, 2, 3, 4, \ldots)$ and real numbers $(1.0, 2.3, 4.9, \ldots)$. Typing a period is simply a shorthand of typing a zero decimal. As an example, the following two inputs are equal:

```
>> TF = 9./5.*TC + 32.
>> TF = 9.0/5.0*TC + 32.0
```

Integer division is not the same as real number division. With integer division, $9/5 = 1$, while with real numbers $9.0/5.0 = 1.8$. (This will change in the next version of Python where all numbers are assumed to be real, but for now we need to be explicit about when we are using real numbers.)

Instead of retyping the formula, you can use the up arrow to find your previously typed commands and execute them again. We have now defined a variable, `TC`. You can see the value given to `TC` by typing:

```
>>> print TC
```

```
40
```

Notice that we assigned a value to the variable `TF` through a calculation. We have not introduced a function for `TF`. What does this mean? It means that if you change the value of `TC`, the value of `TF` will not change automatically unless you retype the formula for `TF`. You can check this by assigning `TC` a new value, and then asking Python for the value of `TF`:

```
>>> TC = 50
```

```
>>> print TF
```

```
104.0
```

This is an important aspect of a programming language such as Python: a variable does not change value unless you assign a new value to it!

1.3 Scripts and functions

However, we do not want to type in the whole formula each time we want to calculate a new value for T_F. Instead we can make a *script*, a group of several statements, or a *function*, similar to an internal function such as `sin`.

1.3.1 Scripts

We can group several statements into a *script*, which we can reuse. You do this by opening a new file in the `File` menu in Spyder: `File`→ `New file...`. This opens a new window with an editor. Here you can now type (or copy) the commands we already used:

```
TC = 40.
TF = 9./5.*TC + 32.
print TF
```

Now, we need to save the script. In the editor window you do: `File`→`Save`. You must give the script a name and choose where to place it. This will generate a file with an extension `.m` – we call such a file an m-file, because it shows that the file contains a Python script/program. You run the program from the editor window by typing the `F5` key. A dialog box will show up. Select ”Execute in current Python or IPython interpreter“, leave all other this options as is and click the `Run` button. As a result the commands in the script are executed as if they were typed into the Python window, and the resulting output is shown in the Python window:

```
104.0
```

You can now change the value of `TC` in the script and rerun the script to redo the calculation for another temperature. Notice that we wrote the script so that the temperature `TC` is assigned inside the script. This means that if you change the value of `TC` on the command line, for example by typing:

```
>>> TC = 45.
```

and then run the script – the script will not use this new value of TC, but instead use the value from inside the script.

Writing scripts to solve simple problems will be our standard operating procedure throughout this text. This is an efficient way to develop a simple program, change the parameters (such as changing TC), and rerunning the program with new parameters. While this is practical for developing short programs and solving simple problems, it is not good programming practice. In general, we encourage the development of good programming practices, but in this text we will prioritize making the code as simple as possible.

1.3.2 Functions

From a programming perspective, it is better to introduce a *function* to calculate the temperature. A user defined function acts just like a predefined mathematical function such as `sin` or `exp`. We define a function by opening a new m-file: Push `File`→`New`→`Blank M-file`. We define a function by typing the following into the editor:

```
def convertF(TC);
    # Converts from centigrade to Fahrenheit
    TF = 9./5.*TC + 32.
    return TF
```

and save with the name `convertF.m`

What do these statements do? We define a function by the command `function`. First, we write what the function should return. Here, the function returns the variable `TF`. We introduce the name of the function, `convertF`, and the arguments that we need to input when we use the function – here the only argument is the temperature in centigrades, `TC`. Inside the function we must calculate the value of `TF`, because this is the value the function is supposed to calculate.

We call our new function by typing:

```
>>> convertF(45)
```

```
113.0
```

Notice that Python requires each such used-defined function to be in a separate file, and that the file must be in the search path for Python. This means that the file `convertF.m` must be in the current directory or in the standard Python directory for this to work. I suggest that you always save the functions you need in the same directory as you save the scripts you are currently working on, and that you make new directories for each problem you are working on.

A particular feature of functions is that the internal calculations and variables used inside the function are lost as soon as the function is finished. For example, Python may use several calculation steps if we call the `sin` function, but this is hidden from us. Outside the function we only see the result of the function. To illustrate this, we could break our short function into several steps:

```
def convertF2(TC);
    # Converts from centigrade to Fahrenheit
    ratio = 9./5.
    constant = 32
    TF = factor*TC + constant;
    return TF
```

Here, we have introduced two internal variables, `ratio` and `constant`, that are forgotten as soon as the function is finished. For example, if we type:

```
>>> convertF2(45)
```

```
113.0
```

```
>>> print ratio
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'ratio' is not defined
```

we see that Python does not know the value of `ratio` after the function has done its work.

Functions are powerful and necessary tools of more advanced programming techniques, and will be gradually introduced throughout the text. But initially we try to make the programs as simple as possible, and we will then use simple scripting as our main tool.

1.4 Plotting data-sets

Python not only works as a numerical calculator, it also has advanced data visualization methods for visualizing data. For example, as part of a laboratory exercise you may have measured the volume and mass of a set of steel spheres. You number the measurements using the index, i, and record masses m_i and volumes V_i in the following table:

i	1	2	3	4	5	6
m_i	1 kg	2 kg	4 kg	6 kg	9 kg	11 kg
V_i	0.13 l	0.26 l	0.50 l	0.77 l	1.15 l	1.36 l

where we have used that 1 litre = 1l = 1dm^3.

Such a sequence of numbers are stored in an *array* (or a *vector*) in Python. We define the sequence of masses and volumes in Python using

```
>>> m = [1, 2, 4, 6, 9, 11]
>>> V = [0.13, 0.26, 0.50, 0.77, 1.15, 1.36]
```

We can find an individual mass value by:

```
>>> print m[1]
```

```
1
```

```
>>> print m[4]
```

```
6
```

There are now 6 values for the masses, numbered `m(1)` to `m(6)`. We call the array `m` a 6×1 array, or a vector of length 6. The volumes are stored the same way:

```
>>> print V[1]
```

```
0.13
```

```
>>> print V[4]
```

```
0.77
```

The enumeration of the two arrays is identical: element `m[4]` of the masses corresponds to element `V[4]` of the volumes.

The relation between m and V is illustrated by plotting V as a function of m. This is done by the `plot` command:

```
>>> plot(m,V,'o')
```

where the string `'o'` ensures that a small circles is plotted at each data-point. The plot command makes a “scatter” plot – it contains a point for each of the data-points $m(i), V(i)$ in the two arrays. The two arrays must therefore be the same length – they must have the same number of elements. We annotate the axes by:

```
>>> xlabel('m [kg]')
>>> ylabel('V [l]')
```

where the `xlabel` refers to the first array `m` in the the `plot(m,V,'o')` command, and the `ylabel` refers to the second array – the `V` array. The resulting plot in Python is shown in figure 1.2.

Where did the units (kg and liters) go when we defined the mass `m` and the volume `V`? We cannot use units when we introduce digital representations of the numbers. We can only input numbers into Python, and we have to keep track of the units. This is why we specified the units along the axes in the `xlabel` and `ylabel` commands.

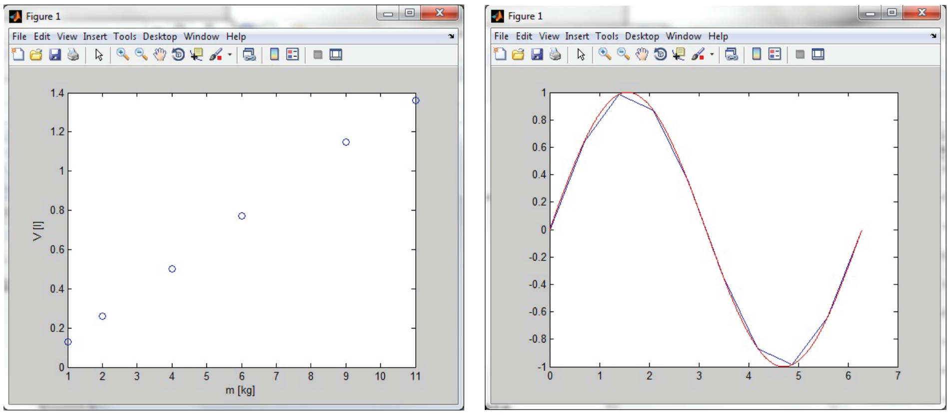

Figure 1.2: *The Python window with the plot of V as a function of m (Left) and with the plot of $\sin(x)$ as a function of x for 10 points in blue and for 1000 points in red. (Right)*

1.5 Plotting a function

Python cannot plot a function such as $\sin(x)$ directly. We must first generate two sequences of numbers, one sequence for the x'es and one sequence for the corresponding values of $\sin(x)$, and then plot the two sequences against each other. While this may sound complicated, Python has functions that ensure that you can almost directly write the mathematical expression into Python.

1.5.1 Loops

We want to make a sequence of x'es, such as $0.0, 0.1, 0.2, 0.3, \ldots$ etc, and then for each x_i we want to calculate the corresponding value for $\sin(x_i)$:

i	1	2	3	4	5	...	n
x_i	0.0	0.1	0.2	0.3	0.4	...	10.0
$\sin(x_i)$	$\sin(0.0)$	$\sin(0.1)$	$\sin(0.2)$	$\sin(0.3)$	$\sin(0.4)$	...	$\sin(10.0)$

where we generate x_i from 0.0 to 10.0 in steps of 0.1. How do we generate such an array in Python? First, we have to generate the array[1]. How many elements do we need? Going from 0.0 to 10.0 in steps of 0.1 we need:

$$n = \frac{10.0 - 0.0}{0.1} + 1 \ , \tag{1.2}$$

steps, where we have added one in order to include the last step (otherwise we would stop at 9.9 instead of at 10.0. We define an array of this length by:

```
>>> n = ceil((10.0-0.0)/0.1)+1
>>> x = zeros((n,1))
```

Here, the function `ceil()` rounds up after the division, and the function `zeros((n,1))` generates and returns an array of size `n` by `1` which is filled with zeros. Now we need to fill the array:

```
>>> x[0] = 0.0;
>>> x[1] = 0.1;
>>> x[2] = 0.2;
>>> x[3] = 0.3;
>>> x[4] = 0.4;
...
>>> x[n-1] = 10.0
```

[1] In Python it is not necessary to define the size of the array before it is filled. We could just fill it as we go along, but this is not good coding practice, it will lead to very slow codes for large arrays, and may cause surprising errors in your programs. We will therefore always predefine the size of arrays.

Note that Python starts counting on zero, so the first index is 0, while the last index is $n-1$.

Fortunately, there is a more efficient way of doing this – by using a `for`-loop!. A `for`-loop allows us to loop through a list of values $1, 2, 3, \dots, n$ for the variable `i`, and then execute a set of commands at each step – exactly what we need. We can replace the long list of `x(1) = 0.0` etc by the loop:

```
>>> for i in range(int(n)):
...     x[i] = i*0.1
```

If you type this in, nothing will execute until you press Enter twice. Note also that you need to indent the first line after the colon in the line with the `for` statement. Indenting the line means that you add four spaces to the beginning of the line. This is because Python expects the body of the loop to be indented.

You can check the generated values of `x` by:

```
>>> print x
```

```
[[ 0. ]
 [ 0.1]
 [ 0.2]
 [ 0.3]
 [ 0.4]
 [ 0.5]
 [ 0.6]
 [ 0.7]
 [ 0.8]
 [ 0.9]
...
```

Notice how we specify the range of the loop, by specifying a sequence of numbers by use of the `range(int(n))` function. Typing `range(int(n))` at the command prompt gives you exactly the list of values for `i`. The use of `int(n)` here is solely to convert `n` from a real number to an integer. The reason is that range expects integers and will not handle real numbers. To us, the variable `n` is an integer, but it is currently represented as a real number internally in Python, so we need to convert it explicitly.

Now, let us put this into a small script. And let us also calculate the value for the function $\sin(x)$:

```
from pylab import *

x0 = 0.0
x1 = 10.0
dx = 0.1
n = ceil((x1 - x0) / dx) + 1
x = zeros((n, 1))
y = zeros((n, 1))
for i in range(int(n)):
    x[i] = x0 + (i - 1) * dx
    y[i] = sin(x[i])

plot(x, y)
show()
```

Which both generates and plots the function $\sin(x)$. The variables `x0`, `x1`, and `dx` provide the start, stop and step of the x-values used. You can now change them and rerun the script to generate plots for other ranges or with other resolutions.

Notice that `y(i) = sin(x[i])` must appear inside the loop – that is before the end, otherwise it would only be executed once, using the value `i` had at the end of the loop. Putting commands outside a loop that should be inside a loop is a common mistake – sometimes also done by experienced programmers.

The `while`-loop

The `for`-loop is probably be the loop-structure you will use the most, but there are also other tools for making a loop. For example, the `while`-loop. In the `while`-loop the commands inside the loop are executed until the expression in the while command is true. It does not automatically update a counter either. For example, we could have implemented the same program above using a while loop in the following way:

```
from pylab import *
x0 = 0.0
x1 = 10.0
dx = 0.1
n = ceil((x1-x0)/dx) + 1
x = zeros((n,1))
y = zeros((n,1))
i = 0
while i<=n:
    i = i + 1
    x[i] = x0 + (i-1)*dx
    y[i] = sin(x[i])

plot(x,y)
show()
```

ere you notice that we must assign `i=0` before the loop, and `i=i+1` inside the loop, since we now need to update the counter "manually" inside the loop. We also introduce an "expression" `i<=n` which may be false (having the value 0) or true (having the value 1). The loop continues until the expression becomes false. Notice that a common source of error is to generate `while` loops that continue forever, for example because you have forgotten to update the counter inside the loop. You will notice this because your program never ends: Python will never stop or plot your results.

You may wonder what the point of the `while`-loop is, since it looks more cumbersome than the `for`-loop. We will use the `while` loop when we want to continue a calculation for an unknown number of steps. For example, you may want to find the motion of a falling ball until it hits the ground. But you may not know beforehand how many steps you need before it hits the ground. For example, the position of the ball may be given by $x(t) = 1000 - 4.9 \cdot t^2$[2]. We would then calculate the position for time in steps of dt as long as x is positive using the following program:

This script is now a bit more complicated, and needs some explanation. First, we notice that the variable we update in intervals now is `t` and not `x` as before. Otherwise the update of `t` is as before. However, there is a common "trick" with such while loops: We have to calculate the value of `y[1]` before the loop starts, otherwise the first time we enter the loop, the expression may not be true, and the loop would never start. This is also a common mistake. Therefore, ensure that you understand why and what is done before the `while`-loop starts in this script. The rest of the script follows the example from above.

1.5.2 Vectorization

While loops are generally powerful and useful methods, there is a much simpler way to generate sequences of numbers and plot functions in Python– and the method also allows your code to stay much more similar to the mathematical formulas. This method is called *vectorization*.

We can make a sequence of x'es in several ways using functions that are built into Python instead of using a loop. For example, the function `linspace` generates a sequence of numbers that are equally spaced from the start `0` to the end `10.0`:

```
>>> x = linspace(0,10,10)
>>> print x
```

```
[ 0.          1.11111111  2.22222222  3.33333333  4.44444444
  5.55555556  6.66666667  7.77777778  8.88888889 10.        ]
```

In this case we generated 10 numbers, but you can fill in with your wanted resolution. An alternative to specifying the number of points you want – as we do with `linspace` – is to specify the step size, the expression `arange(0.0, 10.0, 0.3)` returns an array starting at the value 0 and ending at 10.0 in steps of 0.3[3]:

[2] We can solve this particular problem analytically, to find when x becomes zero, but there will be cases we cannot solve analytically, and general tools are needed.

[3] Notice the small difference between the two methods: Using `linspace` ensures that the first and the last numbers are included in the list, but when you use `arange(0.0, 10.0, 0.3)` the

```
>>> x = arange(0.0, 10.0, 0.3)
```

```
array([ 0. ,  0.3,  0.6,  0.9,  1.2,  1.5,  1.8,  2.1,  2.4,  2.7,
    3. ,
        3.3,  3.6,  3.9,  4.2,  4.5,  4.8,  5.1,  5.4,  5.7,  6. ,
            6.3,
        6.6,  6.9,  7.2,  7.5,  7.8,  8.1,  8.4,  8.7,  9. ,  9.3,
            9.6,
        9.9])
```

Ok - so that was simply a simpler way of generating the array `x`. Why is this so much simpler? Because of a powerful and nice feature of Python called vectorization: We can apply the function $\sin(x)$ to the whole array `x`. Python will then apply the function to each of the elements in `x` and return a new array with the same number of elements as `x`. The three lines:

```
>>> x = linspace(0,10,10);
>>> y = sin(x);
>>> plot(x,y);
```

are equivalent to the program:

```
from pylab import *
x0 = 0.0
x1 = 10.0
dx = 1.0
n = ceil((x1-x0)/dx) + 1
x = zeros((n,1))
y = zeros((n,1))
for i in range(int(n)):
    x[i] = x0 + (i-1)*dx;
    y[i] = sin(x[i]);

plot(x,y);
show()
```

Notice how simple the vectorized Python code is – it is almost identical to the mathematical formula. We only have to define the range of `x`-values before we call the `sin(x)` function. Beautiful and powerful.

The program above generates the blue plot in figure 1.2. However, this plot has too sharp corners, because we have too few data-points. Let us generate 1000 points of x-values, and plot $\sin(x)$ with this resolution in the same plot:

```
>>> x = linspace(0,10,1000)
>>> y = sin(x)
>>> hold('on')
>>> plot(x,y,'-r')
>>> hold('off')
```

The result is shown in figure 1.2 with a red line. Here, we have used a few more tricks. We use the command `hold('on')` to ensure that Python does not generate a new plot, which would remove the previous one, but instead plots the data in the same plot as we have already used. Typing `hold('off')` stops this behavior – otherwise all subsequent plots will be part of the same plot. We have also used the string `'-r` to tell Python that we want a red line. You can find more plotting methods in the summary at the end of the chapter.

The vectorization technique is very general, and usually allows you to translate a mathematical formula to Python by typing in the corresponding expression in Python, for example, we can plot the function

$$f(x) = x^2 e^{-ax} \sin(\pi x) \ , \tag{1.3}$$

from $x = 0$ to $x = 10$ by typing (when $a = 1$):

```
>>> x = linspace(0,10,1000)
>>> a = 1.0
>>> f = x**2.*exp(-a*x)*sin(pi*x)
>>> plot(x,f)
```

As soon as you have learned to transcribe mathematical expressions from the mathematical notation to Python you are ready to calculate and plot any function in Python.

---

last number is 9.9 and not 10.0!

The technique of vectorization is a powerful and efficient technique. Python is usually very fast at calculating vectorized commands, and we can often write very elegant programs using such techniques, ensuring that the Python code follows the mathematical formulation closely, which makes the code easy to understand.

1.6 Random numbers

Sometimes you need randomness to enter your physical simulation. For example, you may want to model the motion of a tiny dust of grain in air bouncing about due to random hits by the air molecules, so called brownian motion. As a result you want the grain to move a *random* distance during a given time interval. How do we create random numbers on the computer? Unfortunately, we cannot generate really random numbers, but most programs have decent pseudo-random number generators, that generates a sequence of numbers that appear to be random. In Python, we can simulate the throw of a dice using

```
>>> randint(6) + 1
```

```
3
```

where `randint(n)` generates a random integer between 0 and $n-1$ – where each outcome has the same probability. If you type the command several times, you will get a new answer each time. Python includes several functions that returns random numbers: It can generate random real numbers between 0 and 1 using the `rand` function and normal-distributed numbers (with average 0 and standard derivation 1) using the function `randn`.

1.7 Conditions – `if/else/end`

Now, if we return to discuss the motion of the grain of dust, we want to model its motion according to a simple rule: I throw a dice, if I get between 1 and 3, the grain moves a step forward, otherwise it moves a step backward. How can we handle such conditions? We need a set of conditional statements – so that we can perform a given set of commands when a particular condition is fulfilled – we need an `if`-statement:

```
if expr:
    <statement a1>
    <statement a2>
    ..
else:
    <statement b1>
    <statement b2>
    ..
```

Here the expression `(expr)` is an expression such as `randint(6)>3` which may be true or false. If the expression is true, statements `a1`, `a2`, ... are executed, otherwise the statements `b1`, `b2`, ... are executed.

Let us use this to find the motion of the grain. Every time we throw the dice, the grain moves a distance $dx = \pm 1$. If the grain is at position x_i at step i, the grain will be at a position

$$x_{i+1} = x_i + dx \; . \tag{1.4}$$

We can use this rule and an `if`-statement to write the script to find the position at subsequent steps $i = 1, 2, \ldots$:

```
from pylab import *
n = 1000
x = zeros((n,1))
for i in range(1,n-1):
    if (randint(6) + 1 <=3):
        dx = -1
    else:
        dx = 1
    x[i+1] = x[i] + dx

plot(x)
```

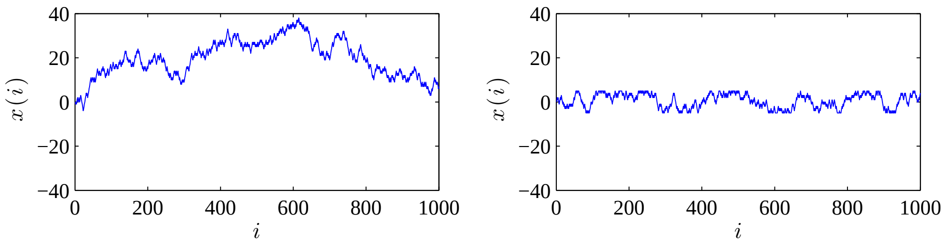

Figure 1.3: *Plot of the position $x(i)$ of a random walker (a bouncing grain of dust) as a function of the number of steps i done (left), and when the walker is constrained to the zone $-5 \leq x \leq +5$ (right).*

```
xlabel('i')
ylabel('x(i)')
show()
```

The resulting motion is shown in figure 1.3.

We will use `if`-statements throughout the text, often to enforce particular conditions on the motion. For example, we could add a level of complexity to the motion of the grain by requiring that the grain moved inside a narrow channel of width 10: The grain cannot move outside a region spanning from -5 to +5:

```
from pylab import *
n = 1000
x = zeros((n,1))
for i in range(1,n):
    if (randint(2)==1):
        dx = -1
    else:
        dx = +1;

    x[i] = x[i-1] + dx
    if (x[i]>5):
        x[i] = 5

    if (x[i]<-5):
        x[i] = -5

plot(x)
xlabel('i')
ylabel('x(i)')
show()
```

The resulting motion x_i as a function of i is shown in figure 1.3.

For the interested reader, we include a particularly compact formulation of the random walk

```
>>> x = cumsum(2*(randint(1,7,1000)<=3)-1)
>>> plot(x)
```

1.8 Reading real data

When you work with physics you need to handle real data: NASA publishes data for the motion of most stellar objects; Your mobile phone has an accelerometer and a GPS that measures thousands of data-points in a few seconds. You do not want to type these numbers by hand. Therefore you need to be able to read files containing numbers. For example, the motion of a sprinter running 100m is given in the file `run100m.d`. The file looks like this if you open it in a text editor (such as emacs, textedit, winedit):

```
  0.0000000e+000  -2.1155775e-001
  1.0000000e-002  -1.7485406e-001
  2.0000000e-002  -1.3798607e-001
  3.0000000e-002  -1.0095306e-001
  4.0000000e-002  -6.3754256e-002
  5.0000000e-002  -2.6388915e-002
  ...
```

A total of 972 lines of data. The first column gives the time, measured in seconds, and the second column gives the position of the runner, measured in meters. Fortunately, it is very simple to read such as file into Python. It is done by a single command:

```
>>> run100m = loadtxt("run100m.d")
```

We split the data into two arrays `t` and `x` by:

```
>>> t = run100m[:,1]
>>> x = run100m[:,2]
```

and plot the data using

```
>>> plot(t,x)
```

If you experience a problem where Python cannot find the file, getting an error message like:

```
>>> loadtxt("run100m.d")
```

```
...
IOError: [Errno 2] No such file or directory: 'run100m.d'
```

It means that the file `run100m.d` is not in your current working directory.

Example 1.1: Plot of function and derivative

Problem: Plot the function

$$f(x) = e^{-x^2} \ , \tag{1.5}$$

and its derivative by using the formula:

$$f'(x) \simeq \frac{f(x+h) - f(x-h)}{2h} \ , \tag{1.6}$$

as an approximation for the derivative on the interval $-5 \leq x \leq 5$. You may use the value $h = 0.001$ for h.

Solution: The function can be plotted directly by a vectorized approach:

```
>>> from pylab import *
>>> x = linspace(-5,5,1000)
>>> f = exp(-x**2)
>>> plot(x,f)
>>> show()
```

In order to use the numerical approximation for the derivative, we need to perform the approximation for each of the x-values in the `x`-array. We access them by a `for`-loop through the 1000 elements in the `x`-array:

```
>>> h = 0.001
>>> df = zeros((1000,1))
>>> for i in range(1000):
...     df[i] =
    (exp(-(x[i]+h)**2)-exp(-(x[i]-h)**2))/(2*h)
...
>>> plot(x,df)
>>> show()
```

The resulting plot is shown in figure 1.4.

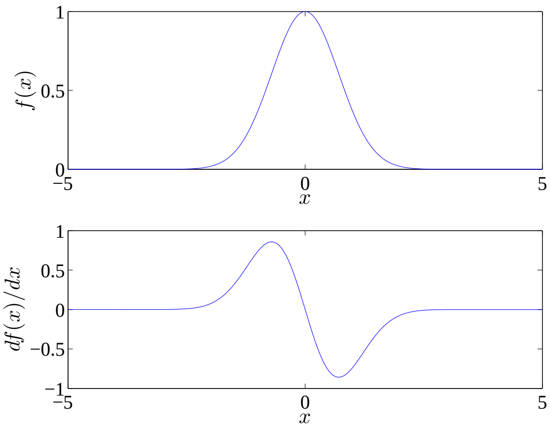

Figure 1.4: *Plot of $f(x)$ as a function of x and its derivative df/dx as a function of x calculated using a numerical method.*

Even simple problems such as these are useful to implement as scripts saved in a file, since this makes debugging – the process of finding and removing errors in the script – simpler. If you make a small mistyping, you have to retype all the commands when you operate on the command line, but if you use a script, you simply make a small change in the script, rerun, and that is it.

Summary – Chapter 1

Using Python as a calculator

- Direct calculations on the command line

```
>>> 10.0*sin(pi/3.)+4.0**3
```

- Defining and reusing variables

```
>>> a = 2.0
>>> b = 4.5
>>> c = a**2 + b**2
```

- Vectorized plotting of functions

```
>>> x = linspace(0,10,0.01)
>>> y = exp(-x)*sin(x)
>>> plot(x,y)
```

Functions and scripts

- A script is a sequence of executable commands stored in a separate `.m` file.
  - All variables are available on from the command line afterwards
  - The script is run by typing `F5` in the editor
  - Scripts allow rapid rerunning a program after changes in parameters

A function has the syntax

```
def myfunction(a,b,c):
   v = a*b*c;
   d = v**2;
   y = 2*d;
   return y
```

- Variables defined inside the function, such as `v` and `d` are not available outside the function

Plotting

- You plot two arrays `t` and `x` versus each other by

```
plot(t,x,’-b’)
xlabel(’t [s]’)
ylabel(’x [m]’)
```

- Line markers and colors are:

Colors		Lines	
`b`	blue	`-`	solid
`g`	green	`:`	dotted
`r`	red	`-.`	dashdot
`c`	cyan	`--`	dashed
`m`	magenta	`(none)`	no line
`y`	yellow		
`k`	black		
`w`	white		

- Plotting symbols are:

Symbols		Symbols	
`.`	point	`v`	triangle (down)
`o`	circle	`^`	triangle (up)
`x`	x-mark	`<`	triangle (left)
`+`	plus	`>`	triangle (right)
`*`	star	`p`	pentagram
`d`	diamond	`h`	hexagram

- Plotting several plots in the same figure:

```
# Either
plot(t1,x1,’-b’,t2,x2,’-r’)
# Or
plot(t1,x1,’-b’)
hold(’on’)
plot(t2,x2,’-r’)
hold(’off’)
```

- Plotting several plots above each other:

```
subplot(2,1,1)
plot(t1,x1,’-b’)
subplot(2,1,2)
plot(t2,x2,’-r’)
```

- Saving a figure to a file: either by using the save button from the figure window. You can also save a figure as a pdf from the command line by

```
savefig(’myfigure.pdf’)
```

where `myfigure.pdf` is the name of the generated file,

Loops

- `for`-loops run a counter sequentially through a list of values

```
for i in range(100):
    x[i] = sin(i/100.0)
```

- `while`-loops run until a given expression is true

```
i = 0;
while (i<100):
    i = i + 1
    x[i] = sin(i/100.0)
```

Expressions

- `if`-statements are used to run a sequence of commands given a particular expression is true:

```
if (x>10.0):
    y = 10.0
else:
    y = -10.0
```

- Expressions return true (1) or false (0):

Expression	Name	Example
`==`	equal	`(x==0.0)`
`!=`	no-equal	`(x!=0.0)`
`>=`	greater than or equal	`(x>=0.0)`
`<=`	less than or equal	`(x<=0.0)`
`>=`	greater than	`(x>0.0)`
`<=`	less than	`(x<0.0)`

- Expressions can be joined using logical operators such as *or* and *and*:

Operator	Name	Example
`and`	logical AND	`(x==0.0) and (y>0.0)`
`or`	logical OR	`(x==0.0) or (y>0.0)`

Exercises – Chapter 1

1.1: Seconds.

(a) Write a script that calculates the number of seconds, s, given the number of hours, h, according to the formula:

$$s = 3600 \cdot h \ , \tag{1.7}$$

(b) Use the script to find the number of seconds in 1.5 hours, 12 hours, and 24 hours.

1.2: Spherical mass.

(a) Write a script that calculates the mass of a sphere given its radius r and mass densit ρ according to the formula:

$$m = \frac{4\pi}{3}\rho r^3 \ , \tag{1.8}$$

(b) Use the script to find the mass of a sphere of steel of radius $r = 1$mm, $r = 1$m, and $r = 10$m.

1.3: Angle.

(a) Write a function that for a point (x, y) returns the angle θ from the x-axis using the formula:

$$\theta = \arctan\left(\frac{y}{x}\right) \ , \tag{1.9}$$

(b) Find the angles θ for the points $(1, 1)$, $(-1, 1)$, $(-1, -1)$, $(1, -1)$.

(c) How would you need to change the function to return values of θ in the range $[0, 2\pi]$?

1.4: Unit vector.

(a) Write a function that returns the two-dimensional unit vector corresponding to an angle θ with the x-axis. You can use the formula:

$$\begin{bmatrix} u_x \\ u_y \end{bmatrix} = \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix} \ . \tag{1.10}$$

where θ is given in radians.

(b) Find the unit vectors for $\theta = 0, \pi/6, \pi/3, \pi/2, 3\pi/2$.

(c) Rewrite the function to instead take the argument θ in degrees.

1.5: Plotting the normal distribution.
The normal distribution, often called the Gaussian distribution, is given as:

$$P(x; \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \ , \tag{1.11}$$

where μ is the average and σ is the standard deviation.

(a) Make a function `normal(x,mu,sigma)` that returns the normal distribution value, $P(x, \mu, \sigma)$ as given by the formula.

(b) Use this function to plot the normal distribution for $-5 < x < 5$ for $\mu = 0$ and $\sigma = 1$.

(c) Plot the normal distribution for $-5 < x < 5$ for $\mu = 0$ and $\sigma = 2$ and for $\sigma = 0.5$ in the same plot.

(d) Plot the normal distribution for $-5 < x < 5$ for $\sigma = 1$ and $\mu = 0, 1, 2$ in three subplots above each other.

1.6: Plotting $1/x^n$.
The function $f(x; n)$ is given as:

$$f(x; n) = \frac{1}{x^n} \ . \tag{1.12}$$

(a) Make a function `fvalue(x,n)` which returns the value of $f(x; n)$.

(b) Use this function to plot $1/x$, $1/x^2$ and $1/x^3$ in the same plot for $-1 < x < 1$.

1.7: Plotting $\sin(x)/x^n$.
The function $g(x; n)$ is given as:

$$g(x; n) = \frac{\sin(x)}{x^n} \ . \tag{1.13}$$

(a) Make a function `gvalue(x,n)` which returns the value of $g(x; n)$.

(b) Use this function to plot $\sin(x)/x$, $\sin(x)/x^2$ and $\sin(x)/x^3$ in the same plot for $-5 < x < 5$.

(c) Use the help function to find out how to place legends for each of the plots into the figure.

1.8: Logistic map. The iterative mapping

$$x(i + 1) = r\, x(i)\,(1 - x(i)) \ , \tag{1.14}$$

is called the logistic map.

(a) Make a function `logistic(x,r)` which returns the value of $x(i + 1)$ given $x(i)$ and r as inputs.

(b) Write a script with a loop to calculate the first 100 steps of the logistic map starting from $x(1) = 0.5$. Store all the values in an array `x` with $n = 100$ elements and plot x as a function of the number of steps i:

(c) Explore the logistic map for $r = 1.0$, 2.0, 3.0 and 4.0.

1.9: Numerical integration. Given a function $f(x)$ we can find the integral from 0 to b:

$$\int_0^b f(x)dx \ , \tag{1.15}$$

using the following formula:

$$\int_0^x f(x)dx \simeq \sum_{i=0}^{n} f(x_i)\Delta x \ , \tag{1.16}$$

where $x_i = b \cdot (i/n)$ and $\Delta x = b/n$.
Let us use use this technique to calculate the integral of $f(x) = \sin(x)/x$.

(a) Define a function `myfunc(x)` which returns the value of $f(x) = \sin(x)/x$ for a given value of x.

(b) Write a script that calculates the integral of $f(x) = \sin(x)/x$ from 0 to 1 using the numerical scheme presented above using a `for`-loop.

So far you have calculated the specific integral from 0 to b. Now, we want to find the function $g(x)$ which is given as the integral:

$$g(x) = \int_0^x f(x)dx \; , \tag{1.17}$$

where $f(x) = \sin(x)/x$ as above. We find this function by simply calculating the values of the integral at all the points $x_i = b \cdot (i/n)$:

$$g(x_0) = 0 \tag{1.18}$$
$$g(x_1) = g(x_0) + f(x_0) \cdot \Delta x \tag{1.19}$$
$$g(x_2) = g(x_1) + f(x_1) \cdot \Delta x \tag{1.20}$$
$$g(x_3) = g(x_2) + f(x_2) \cdot \Delta x \tag{1.21}$$
$$\ldots = \ldots \tag{1.22}$$
$$g(x_n) = g(x_{n-1}) + f(x_{n-1}) \cdot \Delta x \tag{1.23}$$
$$\tag{1.24}$$

(c) Write a script to calculate the values $g(x_i)$ given $f(x) = \sin(x)/x$ for $n = 1000$ x'es in the range from 0 to $b = 1$. Plot $g(x)$ as a function of x.

(d) What would you need to change to instead find the integral of $f(x) = x\exp(-x^4)$ on the interval from 0 to 2?

1.10: Euler's method. In mechanics, we often use Euler's method to determine the motion of an object given how the acceleration depends on the velocity and position of an object. For example, we may know that the acceleration $a(x, v)$ is given as:

$$a(x, v) = -kx - cv \; . \tag{1.25}$$

If we know the position x and the velocity v at a time $t = 0$:

$$x(0) = x_0 = 0 \; , \tag{1.26}$$

and

$$v(0) = v_0 = 1 \; , \tag{1.27}$$

we can use Euler's method to find the position and velocity after a small timestep Δt:

$$v_1 = v(t_0 + \Delta t) = v(t_0) + a(v(t_0), x(t_0))\Delta t \tag{1.28}$$
$$x_1 = x(t_0 + \Delta t) = x(t_0) + v(t_0)\Delta t \tag{1.29}$$
$$v_2 = v(t_1 + \Delta t) = v(t_1) + a(v(t_1), x(t_1))\Delta t \tag{1.30}$$
$$x_2 = x(t_1 + \Delta t) = x(t_1) + v(t_1)\Delta t \tag{1.31}$$
$$\tag{1.32}$$

and so on. We can therefore use this scheme to find the position $x(t)$ and the velocity $v(t)$ as function of time at the discrete values $t_i = i\Delta t$ in time.

(a) Write a function `acceleration(v,x,k,C)` which returns the value of $a(x, v) = -kx - Cv$.

(b) Write a script that calculates the first 100 values of $x(t_i)$ and $v(t_i)$ when $k = 10$, $C = 5$, and $\Delta t = 0.01$. Plot $x(t)$, $v(t)$, and $a(t)$ as functions of time.

(c) What would you need to change to instead find $x(t)$ and $v(t)$ is the acceleration was given as $a(v, x) = k\sin(x) - Cv$?

1.11: Throwing two dice. You throw a pair of six-sided dice and sum the number from each of the dice:

$$Z = X_1 + X_2 \; , \tag{1.33}$$

where Z is the sum of the results from dice 1, X_1, and dice 2, X_2. If we perform this experiment many times (N), we can find the average and standard deviation from standard estimators from statistics. The average, $\langle Z \rangle$, of Z is estimated from:

$$\langle Z \rangle = \frac{1}{N}\sum_{j=1}^{N} Z_j \; , \tag{1.34}$$

and the standard deviation, ΔZ, is estimated from:

$$\Delta Z = \frac{1}{N-1}\left(\sum_{j=1}^{N}(Z_j - \langle Z \rangle\right)^2 \; . \tag{1.35}$$

(a) Write a function that returns an array of N values for Z.

(b) Write a function that returns an estimate of the average of an array `z` using the formula provided.

(c) Write a function that returns an estimate of the standard deviation of an array `z` using the formula provided.

(d) Find the average and standard deviation for $N = 100$ throws of two dice.

1.12: Reading data. The file `trajectory.dat` contains a list of numbers:

```
t0 x0 y0
t1 x1 y1
t2 x2 y2
.. .. ..
tn xn yn
```

corresponding to the time `t(i)` measured in seconds, and the x and y positions `x(i)` and `y(i)` measured in meters for the trajectory of a projectile.

(a) Read the data file into the arrays `t`, `x`, and `y`.

(b) Plot the x and y positions as function of time in two plots above each other.

(c) Plot the (x, y) position of the object in a plot with x and y on the two axes.

1.13: Numerical derivative of a data-set. The file `trajectoryy.dat` contains a list of numbers:

```
t0 y0
t1 y1
t2 y2
.. .. ..
tn yn
```

corresponding to the time `t(i)` measured in seconds, and the y position `y(i)` measured in meters for the trajectory of a projectile.

(a) Read the data file into the arrays `t`, and `y`.

(b) Plot $y(t)$ as function of time.

For a data-set `t(i)`, `y(i)`, you can estimate the time derivative of the corresponding function $y(t_i)$ at the time t_i using:

$$v(t_i) \simeq \frac{y(t_i + \Delta t) - y(t_i)}{t_{i+1} - t_i} = \frac{y(t_{i+1}) - y(t_i)}{t_{i+1} - t_i} \; , \tag{1.36}$$

where $y(t_i) =$`y(i)` and $t_i =$`t(i)`.

(c) Write a script to calculate the time derivative $v(t_i)$ of the dataset using this formula. Implement using a `for`-loop. (Remember that this definition of the numerical derivative is not defined for the last point in the array).

(d) Plot the position $y(t)$ and the derivative $v(t)$ as functions of time in two plots above each other.

1.14: Numerical integration of a data-set. The file `velocityy.dat` contains a list of numbers:

```
t0 v0
t1 v1
t2 v2
.. .. ..
tn vn
```

corresponding to the time `t(i)` measured in seconds, and the velocity `y(i)` measured in meters per second for the trajectory of a projectile.

(a) Read the data file into the arrays `t`, and `v`.

(b) Plot $v(t)$ as function of time.

For a data-set `t(i)`, `v(i)`, you can estimate the function corresponding to the integral of $v(t)$ with respect to t at the times t_i using the iterative scheme:

$$y(t_1) \simeq y(t_0) + v(t_0)\,(t_1 - t_0) \tag{1.37}$$

$$y(t_2) \simeq y(t_1) + v(t_1)\,(t_2 - t_1) \tag{1.38}$$

$$\ldots \simeq \ldots \tag{1.39}$$

$$y(t_n) \simeq y(t_{n-1}) + v(t_{n-1})\,(t_n - t_{n-1}) \tag{1.40}$$

$$\tag{1.41}$$

where $v(t_i) =$ `v(i)` and $t_i =$ `t(i)`. You can assume that the motion starts at $y(t_0) = 0.0$m at $t = t_0$.

(c) Write a script to calculate the time integral $y(t_i)$ of the dataset using this formula. Implement using a `for`-loop.

(d) Plot the position $y(t)$ and the derivative $v(t)$ as functions of time in two plots above each other.

Project: Sliding on snow

In this project we address the motion of an object sliding on a slippery surface such as a ski sliding in a snowy track. You will learn how to find the equation of motion for sliding systems both analytically and numerically, and to interpret the results.

We start by studying a simplified situation called frictional motion: A block is sliding on a surface as illustrated in, moving with a velocity v in the positive x-direction. The forces from the interactions with the surface results in an acceleration:

$$a = \begin{cases} -\mu(|v|)g & v > 0 \\ 0 & v = 0 \\ \mu(|v|)g & v < 0 \end{cases} ,$$

where $g = 9.8\text{m/s}^2$ is the acceleration of gravity. Let us first assume that $\mu(v) = \mu = 0.1$ for the surface. That is, we assume that the coefficient of friction does not depend on the velocity of the block. We give the block a push and release it with a velocity of 5m/s.

(a)

Find the the velocity, $v(t)$, of the block.

(b)

How long time does it take until the block stops?

(c)

Write a program where you find $v(t)$ numerically using Euler's or Euler-Cromer's method. (Hint: You can find a program example in the textbook.) Use the program to plot $v(t)$ and compare with your analytical solution. Use a timestep of $\Delta t = 0.01$.

The description of friction provided above is too simplified. The coefficient of friction is generally not independent of velocity. For dry friction, the coefficient of friction can in some cases be approximated by the following formula:

$$\mu(v) = \mu_d + \frac{\mu_s - \mu_d}{1 + v/v^*} ,$$

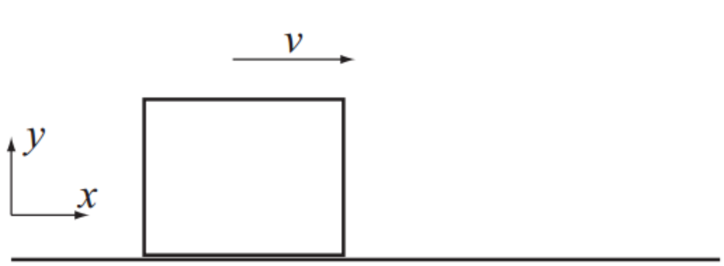

Figure 1.5: *A block moving on a slippery surface.*

where $\mu_d = 0.1$ often is called the dynamic coefficient of friction, $\mu_s = 0.2$ is called the static coefficent of friction, and $v^* = 0.5$m/s is a characteristic velocity for the contact between the block and the surface.

(d)

Show that the acceleration of the block is:

$$a(v) = -\mu_d g - g\frac{\mu_s - \mu_d}{1 + v/v^*} \ ,$$

for $v > 0$.

(e)

Use your program to find $v(t)$ for the more realistic model, with the same starting velocity, and compare with your previous results. Are your results reasonable? Explain.

The model we have presented so far is only relevant at small velocities. At higher velocities the snow or ice melts, and the coefficient of friction displays a different dependency on velocity:

$$\mu(v) = \mu_m \left(\frac{v}{v_m}\right)^{-\frac{1}{2}} \text{ when } v > v_m \ ,$$

where v_m is the velocity where melting becomes important. For lower velocities the model presented above with static and dynamic friction is still valid.

(f)

Show that

$$\mu_m = \mu_d + \frac{\mu_s - \mu_d}{1 + v_m/v^*} \ ,$$

in order for the coefficient of friction to be continuous at $v = v_m$.

(g)

Modify your program to find the time development of v for the block when $v_m =$ 1.5m/s. Compare with the two other models above: The model without velocity dependence and the model for dry friction. Comment on the results.

(h)

The process may become more clearer if you plot the acceleration for all the three models in the same plot. Modify your program to plot $a(t)$, plot the results, and comment on the results. What would happen if the initial velocity was much higher or much lower than 5m/s?

Appendices

Solutions to exercises

Chapter 1

1.1(a)

```
s = 3600 * h
```

1.1(b) 5400s, 43200s, 86499s

1.2(a)

```
m = (4*pi/3)*rho*r**3
```

1.3(a)

```
def angle(x,y):
    theta = arctan(float(y)/float(x))
    return theta
```

1.4(a)

```
def vector(theta):
    ux = cos(theta)
    uy = sin(theta)
    return array([ux, uy])
```

1.5(a)

```
def normal(x,mu,sigma):
    return 1/sqrt(2*pi*sigma**2) *
        exp(-(x-mu)**2/(2*sigma**2))
```

1.5(b)

```
from pylab import *
x = linspace(-5,5,100)
P = normal(x,0,1)
plot(x,P)
show()
```

1.5(c)

```
hold('on')
P = normal(x,0,2)
plot(x,P,'-r')
P = normal(x,0,0.5)
plot(x,P,'-g')
```

1.5(d)

```
from pylab import *
x = linspace(-5,5,1000)
P = normal(x,0,1)
subplot(3,1,1)
plot(x,P,'-b')
P = normal(x,1,1)
subplot(3,1,2)
P = normal(x,2,)
subplot(3,1,3)
plot(x,P,'-g')
show()
```

1.6(a)

```
def fvalue(x,n):
    f = 1.0/(x**n)
    return f
```

1.6(b)

```
from pylab import *
x = linspace(-1,1,1000)
f1 = fvalue(x,1)
f2 = fvalue(x,2)
f3 = fvalue(x,3)
plot(x,f1,x,f2,x,f3)
show()
```

1.7(a)

```
def gvalue(x,n):
    g = sin(x)/(x**n)
    return g
```

1.7(b)

```
from pylab import *
x = linspace(-5,5,1000)
g1 = gvalue(x,1)
g2 = gvalue(x,2)
g3 = gvalue(x,3)
plot(x,g1,x,g2,x,g3)
show()
```

1.8(a)

```
def logistic(x,r):
    g = r*x*(1-x)
    return g
```

1.8(b)

```
r = 1.0
n = 100
x = zeros((n,1))
x[0] = 0.5
for i in range(1,100):
    x[i] = logistic(x[i-1]
i = range(1,100)
plot(i,x)
show()
```

1.9(a)

```
def myfunc(x):
    f = sin(x)/x
    return x
```

1.9(b)

```
n = 100
b = 1.0
dx = b/n
sum = 0.0
for i in range(n):
    xi = b*i/float(n)
    fxi = myfunc(xi)
    sum += fxi*dx
```

1.9(c)

```
n = 1000
b = 1.0
dx = b/n
x = zeros((n,1))
f = zeros((n,1))
g = zeros((n+1,1))
g[0] = 0.0
for i in range(n):
    x[i] = b*i/float(n)
    f[i] = myfunc(x[i])
    g[i+1] = g[i] + f[i]*dx
plot(x,g)
show()
```

1.9(d) You only need to chang the function `myfunc` and the value of b in the script.

```
def myfunc(x):
    f = x * exp(-x**4)
    return f
```

1.10(a)

```
def acceleration(v,x,k,C):
    a = -k*x - C*v
    return a
```

1.10(b)

```
from pylab import *
k = 10
C = 5
n = 100
dt = 0.01
x = zeros((n,1))
v = zeros((n,1))
a = zeros((n,1))
t = zeros((n,1))
x[0] = x0
v[0] = v0
for i = range(1,n)
   a[i] = acceleration(v[i],x[i],k,C)
   v[i+1] = v[i] + a[i]*dt
   x[i+1] = x[i] + v[i]*dt
   t[i+1] = t[i] + dt
subplot(3,1,1)
plot(t,a)
xlabel('t')
ylabel('a')
subplot(3,1,2)
plot(t,v)
xlabel('t')
ylabel('v')
subplot(3,1,3)
plot(t,x)
xlabel('t')
ylabel('x')
```

1.10(c) You only need to chang the function `acceleration`.

```
def acceleration(v,x,k,C):
    a = k * sin(x) - C*v
    return a
```

1.11(a) In vectorized notation:

```
def dice(N)
    Z = zeros((N,1))
    for i in range(N):
        X1 = randint(1,7)
        X2 = randint(1,7)
        Z[i] = X1 + X2
    return Z
```

1.11(b)

```
def average(z)
    N = len(z)
    sum = 0
    for i in range(N)
        sum += z[i]
    return sum/float(N)
```

1.11(c)

```
def standarddeviation(z)
    ave = average(z)
    N = len(z)
    sum = 0
    for i in range(N)
        sum += (z[i] - ave)
    return sum**2/float(N-1)
```

1.12(a)

```
from pylab import *
trajectory = loadtxt("trajectory.dat")
t = trajectory[:,0];
x = trajectory[:,0];
y = trajectory[:,2];
```

1.12(b)

```
subplot(2,1,1)
plot(t,x)
xlabel('t [s]')
ylabel('x [m]')
subplot(2,1,2)
plot(t,y)
xlabel('t [s]')
ylabel('y [m]')
show()
```

1.12(c)

```
plot(x,y)
xlabel('x [m]')
ylabel('y [m]')
show()
```

1.13(a)

```
from pylab import *
trajectory = loadtxt("trajectory.dat")
t = trajectory[:,0];
y = trajectory[:,1];
```

1.13(b)

```
plot(t,y)
xlabel('t [s]')
ylabel('y [m]')
show()
```

1.13(c)

```
n = len(t)
v = zeros((n-1,1))
for i in range(n-1):
    v[i] = (y[i+1] - y[i])/(t[i+1]-t[i])
```

1.13(d)

```
subplot(2,1,1)
plot(t,y)
xlabel('t [s]')
ylabel('y [m]')
subplot(2,1,2)
plot(t[0:n-1],v)
xlabel('t [s]')
ylabel('v [m/s]')
show()
```

1.14(a)

```
from pylab import *
velocity = loadtxt("velocityy.dat")
t = velocityy[:,0]
v = velocityy[:,1]
```

1.14(b)

```
plot(t,v)
xlabel('t [s]')
ylabel('v [m/s]')
show()
```

1.14(c)

```
n = len(t)
y = zeros(n,1)
y[0] = 0.0
for i in range(n-1):
    y[i+1] = y[i] + v[i]*(t[i+1]-t[i])
```

1.14(d)

```
subplot(2,1,1)
plot(t,y)
xlabel('t [s]')
ylabel('y [m]')
subplot(2,1,2)
plot(t,v)
xlabel('t [s]')
ylabel('v [m/s]')
show()
```