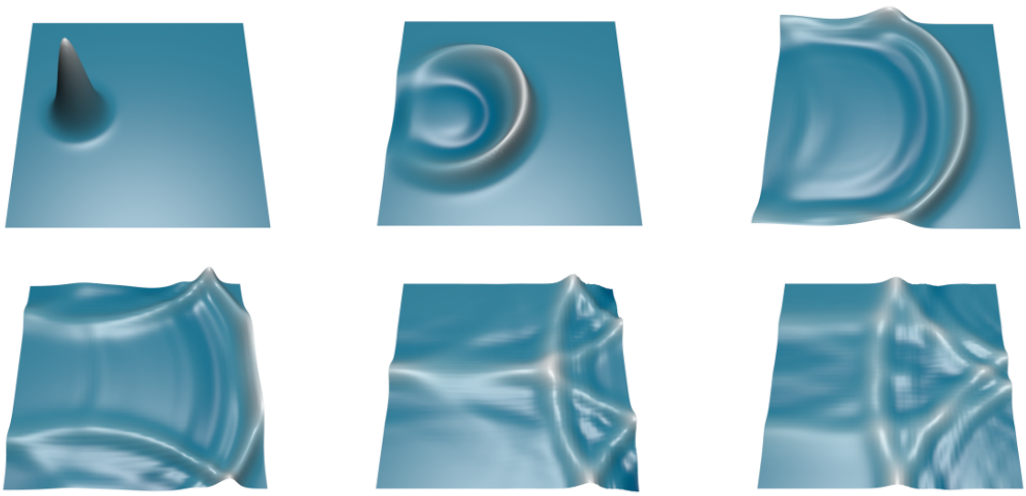
Numerical Techniques for Chemical Engineers

using Python 2.7

Nicolas Renaud & Ferdinand Grozema



TU Delft 2016

Contents

Chapter 1
Introduction to Python

Computers exceptionally good at performing repetitive task at a very high speed, and with virtually zero probability of mistakes. The word computer already implies that the main task of computers is to do computations or calculations, which is the kind of task that occurs very often in engineering disciplines but also in basic sciences, including chemistry and physics. Using computers, a wide variety of mathematical problems can be solved, generally not by deriving analytical solution to a certain mathematical problems but by coming up with a numerical solution. While analytical solutions are sometimes preferred, in a majority of cases it is not even possible to get to such a solution with making very severe approximations. In order to use a computer to help us solve out problems in science an engineering, we need a way of instructing the computer what to do; i.e. we need a programming language. In this course we use a language called Python, which is an easily usable high-level programming language that is very quickly becoming a standard in scientific computing. One of the big advantages is that it is open source (free) and can be used on virtually all platforms (Windows, Mac, Linux). The language is easy to learn and programs written in Python are generally easy to read and often much shorter than comparable programs written in languages such as C or Fortran. A major difference between Python and these languages is that C or Fortran codes have to be compiled into an executable format before they can be run, while in Python the code is interpreted while it is running. The disadvantage of this is that computationally intensive programs run a lot slower than pre-compiled ones written in C or Fortran. However, Python comes packed with an enormous range of standard modules that contain many precompiled functions and algorithms for many common (numerical) tasks. Examples include opening and reading files, complicated mathematical functions, plotting data in graphs and advanced numerical methods. The version of Python that is currently most used is version 2.7 and that is what we will use in this course. In this chapter we will give a general overview the programming language Python, covering most of the basic features that all programming languages share and that will be used in the chapters that follow.

1.1 Python version and documentation

In order to use Python, we need to at least install an interpreter that allows us to run Python codes. In addition, there are many useful tools, for instance graphical user interfaces (GUIs) that make programming in Python more convenient. For this course we will use a Windows installation called Python(x,y) which includes a wide range of tools for science and engineering. It also provides the Spyder editor that is specifically designed for editing Python code. If you wish to install Python(x,y) on your own computer it can be downloaded at:

http://www.python.com

On installation there are options to install a variety of packages, which are not all necessary. The important ones for the purpose of this course are Python, NumPy, SciPy, Spyder and Matplotlib.

There is very extensive documentation available for Python and there are many online resources, including tutorials and examples for specific applications. Python documentation can be easily accessed through the Help function in the Spyder editor. The entire manual and other documentation are also available on:

http://docs.python.org

1.2 Basic calculations in Python

The first way in which Python can be used is as a basic calculator, where you can just print the result of a simple equation, for instance to calculate the height of a ball from the surface of the earth with an initial speed v_0:

The height of the ball at $t = 0.6$ is calculated in a Python program as:

```
1
2 print 5*0.6 - 0.5*9.81*0.6**2
```

This line of code already illustrates the use of the basic numerical operations adding (+), subtracting (-), multiplication (*) and division (/). The square is indicated by **2. This line of text is a complete Python program that can be saved in a file, for instance ball.py. In the Spyder environment that we will use in this course, we can run this program by selecting 'Run' from the menu. The output of this calculation is 1.2342, which will appear if the calculation is executed.
The line of code in the grey area also illustrates how pieces of example code in this text can be recognised. The words in red are the so-called reserved words in Python. They indicate a specific action that you want to happen in the code, in this case printing the answer to the calculation on the screen. These reserved words can not be used for anything else in your code, for instance as the name of a variable..

1.2.1 Variables

While the piece of code above does something useful, it is not easy to recognise the different parameters, let alone vary them easily. For instance selecting a different time, t, or a different initial velocity requires us to know the equation and what each number stands for. It is much easier to change the different parameters if the equation would be written in terms of the variables. In programming languages, and Python is no exception, it is possible to define v_0, g, t and y as variables and combine them into to the right hand side of the equation for our ball. The result can be written as:

```
1
2 v0 = 5
3 g = 9.81
4 t = 0.6
5 y = v0*t - 0.5*g*t**2
6 print y
```

This illustrates the way in which variables in Python are defined: by setting a name equal to a numerical value or an expression containing variables that have been previously defined. The variable

is automatically of the type that is defined by the numerical value that is assigned to it; v_0 is an integer, while g and t are floating point numbers. This assignment of variables also marks a difference with more conventional programming languages such as C and Fortran where variables have to be declared (with the type of the variable indicated) before they are first used. The reason for this is that the code in these languages is compiled before running it, and all memory allocations are sorted out already at that time. In Python, there is no precompiled code and all these issues are sorted out at run time. When the code is written in this way it is much easier to read as in the example above since the original equation can be recognised and the different variables have a certain meaning. In this way it is much easier to modify the numerical value of the variables since they can be found more easily, but the result of the calculation is exactly the same: 1.2342.

The names of the variables can be freely chosen and it is good practice to use names that are easy to understand; i.e. some descriptive name. This makes it easy to go back after you have written an code and actually understand what you wrote before. The naming of the variables is fully free in principle, as can be seen in the example below. The result of the code is exactly the same as before.

```
1
2 initial_velocity = 5
3 acceleration_of_gravity = 9.81
4 TIME = 0.6
5 VerticalPositionOfBall = initial_velocity*TIME - \
6                                0.5*acceleration_of_gravity*TIME**2
7 print VerticalPositionOfBall
```

This version illustrates two things. First of all, when an expression becomes too long it may become convenient or necessary to extend it over multiple lines. This can be done by putting a backslash at the very end of the line (make sure that there are no blank spaces behind the backslash). Secondly, the example shows that it may be a good idea to keep the names of variables reasonably compact because most people would agree that the readability of this last version is worse than the previous one. The result of the code is still exactly the same.

1.2.2 Comments

So far we have only the discussed the actual program statements that actually 'do' something. For a readable code it is advisable to add comments that do not actually do something but they describe the code. This makes it a lot easier follow what is going on in the code and will make it easier to understand and modify a code at a later stage. In Python, comments start the the hashtag character. All text after this character is a comment and will not be considered when the code is executed. Our gravity code with comments looks like this:

```
1 # Program to calculate the height of a ball that moves in the vertical direction
2 v0 = 5          # Initial velocity
3 g = 9.81        # acceleration of gravity
4 t = 0.6                 # time
5 y = v0*t - 0.5*g*t**2   # calculation of the position of the ball
6 print y
```

This program again does exactly the same thing as before, but it is much easier to understand for other people, they may actually figure out what is going on in the code when they read it without explanation by the programmer. Any code that consist of more than a few lines will greatly benefit from the inclusion of comments and a smart choice of the names of variables.

1.2.3 Printing text and formatting numbers

In the programs we have looked at so far the output was just a single number, which basically does the job we would like but it is in many cases nice to have a somewhat more informative line of output. Ideally, we would also like to have some control over the formatting of the number in the output, for instance the number of decimal spaces. An example of such output is given below:

At t=0.6 s, the height of the ball is 1.23 m.

Such kind of output can be achieved using the print statement that we have seen before but the way it is used is a little more complicated. The formatting that is used here is the so-called printf formatting that has its origin in the programming language C. A line of code that supplies such output is the following:

```
1 print 'At t=%g s, the height of the ball is %.2f m.' % (t, y)
```

This line is a bit complicated although the output as specified can be recognised. The print statement prints everything that is enclosed in quotes that can be either double or single. The 'string' contains two slots where numerical values belonging to a variable can be inserted. After the string contained in quotes, there is a percentage sign that is followed by the variables that are to be inserted in parentheses, separated by a comma. The slots in the string itself start with a percentage sign: %g and %.2f in this case. The information that follows behind the percentage sign defines the way in which the number is formatted. Here g means that the real number should be written as compactly as possible, either in a decimal or scientific notation. This is useful for printing numbers on the screen, but it is nice to have some control over how numbers are specified exactly. The second slot, %.2g offers some more more flexibility, in this case a 'floating point number' is written in a decimal notation with two decimals.

Other options that are sometimes useful are scientific notations with %e or %E in which a number is written in scientific format with either 'e' or 'E'. In this notation is is also possible to exactly specify how the number is to be written. For instance %14.6E means that a floating point number is written in scientific notation in a field of 14 characters with 6 decimal places, with capital 'E'. An integer number can be inserted as %d.

Our program, including the print statement now reads:

```
1 v0 = 5
2 g = 9.81
3 t = 0.6
4 y = v0*t - 0.5*g*t**2
5 print 'At t=%g s, the height of the ball is %.2f m.' % (t, y)
```

Using this program it is easy to experiment with some of the different possibilities for printing the output of the numbers that generated.

1.3 Some general remarks on Python and syntax

As we have seen above, computer programs consist of a collection of statements that result in very specific actions. This can be calculations or printing txt or numbers on the screen. For the correct interpretation of the statements it is important to follow the rule of a programming language, or the

'syntax' when writing lines of code. In Python, each line in principle contains a single instruction, and we have already seen that a line of code can be extended over multiple lines. It is in principle allowed to write multiple instructions on a single line, separated by a semicolon. We could have written our program for calculating the height of the ball as:

```
1 v0=5;g=9.81;t=0.6;y=v0*t-0.5*g*t**2;print y
```

Clearly, it is not the best idea to collect all statements on a single line; this line is almost impossible to read! In this line of code, the spaces, for instance around '=' are omitted, which is also true for other mathematical operations. A general convention is to use one blank space around =, - and + and no spaces around *, / and **. It should be noted that this is just a matter of readability, omitting the spaces is in principle correct. Clearly, some spaces are also essential in a line of code, for instance omitting the space behind the print statement would result in an error.

1.3.1 Mathematical equality versus assignments

When we first wrote the line of code that contained the equation for calculating the height of a ball at time %t the result looked very much like a mathematical equation. In mathematics the '=' sign means that the expressions on both sides of this sign are in fact equal. In Python (and most other programming languages) the '=' indicates what we call an assignment. An assignment basically means the following: calculate everything on the left of the '=' and store the result in the variable on the left side. Although this may seem very similar to the mathematical meaning it is in fact very different. Consider the line below:

```
1 y = y + 3
```

Mathematically this is clearly wrong; the value of a variable can not be the same as that same variable plus 3. In a Python program the right side is evaluated, meaning that the current value stored in %y is taken, 3 is added to it and the result is assigned to the variable %y. The old value that was stored in %y is now lost since it was overwritten by the newly calculated value. A short code that illustrates this is shown below:

```
1 y = 3
2 print y
3 y = y + 4
4 print y
5 y = y*y
6 print y
```

This program will result in the output of three numbers: 3, 7 and 49.

1.3.2 Possible error: integer division

In Python version 2.x, which is what we use in this course, the way in which a division is performed depends on the nature of the variable. An example of a possible problem is given in the short code below, which converts a temperature in Celcius to Fahrenheit:

```
1 C = 21
2 F = (9/5)*C + 32
3 print F
```

First of all, on line 3 where the actual calculation takes place contains parentheses. In this line they are not strictly necessary but a wrong placement of parentheses can result in problems. All the numbers in this example are integer numbers and in Python 2.x it is assumed that the division is a so-called

integer division. Running this code results in the value 53 being printed. Clearly, when evaluating this expression with a calculator a different number is obtained (69.8). In the integer division in this code it is evaluated how many times 5 fully fits in 9. This is only one time since 2*5 is 10. The result is a 'truncated' answer: 1, which is multiplied by 21 in this case, after which 32 is added. An easy solution is to make sure that Python evaluates the numbers are 'floating point' numbers:

```
1 C = 21.0
2 F = (9.0/5.0)*C + 32.0
3 print F
```

In this version of the code there is no doubt what the nature of the numbers is: they are all real numbers or floating point numbers and the division is a floating point division. This is a problem that occurs in many programming languages, for instance the newer version of Python (3) (but also MATLAB) automatically assumes that a floating point division is meant, even if both numbers are integer.

1.3.3 Precedence of arithmetic operators

Formulas in Python are evaluated in the same way as they would normally be evaluated in mathematics. This means that power operations, a^4 coded as a**4 have precedence over multiplications and division, which in turn have precedence over addition and subtraction. We can use parentheses in our code to change the way an expression is evaluated.

1.4 Importing modules in Python

The basic types of operations that are present in Python are very limited, but it is already possible to perform many interesting calculations. As we have seen, the basic arithmetic operations are present, but if we need to calculate a square root for instance this is not included in the standard operations. The good thing about Python is that there is a wide range of modules that can be included that contain useful functions. For instance the module 'math' contains a square root function (sqrt) and many other mathematical functions, including sin, cos, exp and log. In order to use these functions it is necessary to 'import' these functions or a whole module into you Python program, which is done by the import statement:

```
1 import math
2 v0 = 9.0
3 answer = math.sqrt(v0)
4 print answer
```

As can be seen in this code, if the module is imported in this way, the function name requires a prefix which is the name of the module where it comes from. This prefix can be a bit annoying, especially if the function is used very often. An alternative import syntax makes it possible to skip the prefix, and only specific functions can be imported:

```
1 from math import sqrt
2 v0 = 9.0
3 answer = sqrt(v0)
4 print answer
```

After this, the function sqrt can be used without the prefix. It is also possible to import more than one function at once:

```
from math import sqrt, exp, sin
```

or simply importing all function from a certain module:

```
from math import *
```

In the last statement all functions in the math module are imported and can be used in your code without the prefix. In general it is advised to import only the functions are actually used in a program because importing all results in a lot of names in the program that are not used, but can also nit be used for variables anymore, however, it is quite convenient, especially in the case of the math module, to just import all functions.

Sometimes it can be convenient to import modules and functions and give them new names in one go:

```
import math as m
#now the math module is called m in the program
v = m.sin(m.pi)         #pi is a constant in the math module that is now called 'm'

from math import log as ln
v = ln(5)

from math import sin as s, cos as c, log as ln
v = s(x)*c(x) + ln(x)
```

Again, all the red words are reserved words in Python!

The math module is only one of the many modules that can be used in Python, the virtually endless list contains modules related to designing web interfaces, doing operations on all sorts of graphical formats (pictures), interacting with the operating system of a computer, etc. In this course we are dealing mostly with the use of Python in a scientific/engineering environment and we focus on numerical mathematics techniques. For the purpose of this course there are two additional modules that we will encounter in the chapters that follow. One is a collection of a wide variety of pre-programmed numerical methods to solve all sorts of problems, including integration, matrix operations, solving differential equations, etc. The lib ray that contains these functions is called numpy. The second module that is useful is called matplotlib, which contains functions to make a graphical representation of data on screen or in a picture. These modules will be introduced in later chapters but they are imported in the same way as the math module here.

1.5 Loops and lists

As mentioned in the introductory part of this chapter, computers are exceptionally well-suited to perform (boring) repetitive tasks in a automatic way. In computer programs such task can conveniently be performed in loops of which different types exist. Additionally, what is often useful for large collections of data (a row of numbers) to store them in a smart way, for instance in a vectors or an array. In Python, such storage can de done in a 'list'. Loops and lists together with functions and if statements form the basic core of programming operations that are used in this course and a thorough understanding of these concepts is essential for understanding the following chapters.

1.5.1 While loops

In the previous parts of this chapter we have evaluated a mathematical formula by inserting a single value and printing the answer, for instance the conversion of temperatures from Celcius to Fahrenheit. Our task now is to print a table with a conversion list with two columns, the temperatures in both scales. For instance a table running from -20 to +40 degrees C in steps of 5 degrees.

A very naive solution to this problem is shown in the following code:

```
C = -20;        F = 9.0/5*C + 32;       print C, F
C = -15;        F = 9.0/5*C + 32;       print C, F
C = -10;        F = 9.0/5*C + 32;       print C, F
C =  -5;        F = 9.0/5*C + 32;       print C, F
C = 0;  F = 9.0/5*C + 32;       print C, F
C = 5;  F = 9.0/5*C + 32;       print C, F
C = 15; F = 9.0/5*C + 32;       print C, F
C = 20; F = 9.0/5*C + 32;       print C, F
C = 25; F = 9.0/5*C + 32;       print C, F
C = 30; F = 9.0/5*C + 32;       print C, F
C = 35; F = 9.0/5*C + 32;       print C, F
C = 40; F = 9.0/5*C + 32;       print C, F
```

This does the job perfectly, twelve almost identical lines of code (with three statements on each line) that give the required result. Now imagine wanting a table in steps 0.01 degrees? It is clear who is doing all the hard work here, in terms of typing the lines of code. While it works perfectly well, it isn't using the capabilities of computers to the fullest. The main issue is: all lines do the same thing, only the value in the variable C changes. All programming languages are equipped with a construct to deal with such problems in a very easy way, so-called loops. In Python there are two kinds of loops, while-loops and for-loops.

A while loop is used to repeat a set of statements as long as a certain condition is true. We will use this type of loop here as an example to generate the list temperatures for our conversion table. A sequence of operations, or an algorithm for this is listed below in text form.

1. Print a line with dashes
2. C = -20
3. While C < 42:
    - F = 9C/5 + 32
    - Print C and F
    - Increment C by 5
4. Print line with dashes

This list of operations that we want to perform is a translation of our task (making a conversion table) into what we call an algorithm. We have already written it in such a way that we can directly translate it into a pice of code in Python, in this case using a while loop. This illustrates the main task of a programmer: translating a particular problem or task (in our case a scientific or engineering problem) into a set of instruction that tell a computer how to solve the problem or execute the task. The implementation of this algorithm in a Python codes now very easy and yields a very short program:

```
print '----------------------------'    # print table heading
C = -20                                 # initialize C
dC = 5                                  # set increment
```

```
4 While C <= 40:                          # loop heading with condition
5         F = (9.0/5)*C + 32              # first statement in loop
6         print C, F                      # second statement
7         C = C + dC                      # third statement
8 print '---------------------------'     # bottom of table (not in loop)
```

This small code contains one of the most important features of programming in Python: the block of statements that is to be executed in the while loop has to be indented by the same amount. All operations under point three in the algorithm above start in the code on the same position of the line. In this way, the Python interpreter knows that these three statement should be 'inside the loop'. At the top pf the loop, line 4, Python checks whether the condition is true (C <= 40), which is certainly true at the beginning since the initial value of C is -20. Subsequently it executes the three lines inside the loop (including printing the table entry) and goes back to the top of the loop, line 4. Inside the loop the value of C has changed so the check has to be performed again. This cycle of events is repeated until, at some point, the condition is not true anymore: C has the value of 45 after twelve cycles. At this point, the program does not go into the loop anymore but skips to the first statement after the loop, line 8. The colon : behind the while statement on line 4 is essential because it marks the start of the block of code on the next lines to be executed. It is interesting to experiment with this code, for instance giving the last line the same indentation as those inside the loop. In this case the lines in the table will be separated by a line of dashes.

It is easy to see from this code that this way of solving this problem is much more convenient and powerful than typing a separate line of code for each temperature. On top of that, it is very easy now to make a much longer list, incremented every 1 degree for instance by just changing the values of dC.

1.5.2 Boolean expressions

An important aspect of while loops is that at the top of the loop there is an expression of which it is evaluated whether it is true. This type of expression is called a boolean expression, which can either be true or false. Other comparisons that can be useful are listed in the following fragment:

```
1 C == 40 # C equals 40
2 C != 40 # C does not equal 40
3 C >= 40 # C is large than or equal to 40
4 C > 40  # C is larger than 40
5 C < 40  # C is smaller than 40
```

The result of a boolean expression can be inverted (changed fro True to False or the other way around) by putting the keyword 'not' in front of it:

```
1 not C == 40     # if C equals 40 the result will be 'False'
```

Boolean expressions can also be combined with the keywords 'and' and 'or':

```
1 while x > 0 and y <= 1:
2         print x, y
```

If both separate conditions are true the overall result is 'True', and in all other cases the result will be 'False'.

1.5.3 Lists

Until now all the calculations we have made using variables contained a single number in an isolated variable. In many cases, especially in mathematics, it is more natural if these number are grouped

together in a list, or an array as it is called in many text about programming. An illustration is the list of temperatures in the examples above. In Python a list can be made just assigning a variable name to a list of numbers that is contained in square brackets, separated by commas:

```
C = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
```

The variable C now refers to a lis object that contains 13 list elements and all these elements are integer numbers. Each element of the list can be assessed separately using an index that refers to its position in the list. The index runs from 0 to 12 in this case. To indicate the third element in the list we can write C[2], which refers to an object of the type int, with the value -10.

Some basic operations can be performed on lists for instance appending elements at the end:

```
C = [ -10, -5, 0, 5, 10, 15, 20, 25, 30]
C.append(35)            #add a new element at the end
```

Two lists can be added

```
C = [ -10, -5, 0, 5, 10, 15, 20, 25, 30, 35]
C = C + [40, 45]               #extend C at the end
```

or elements can be inserted at specific places in the list:

```
C = [ -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
C.insert(0, 15)                # insert 15 a a new element on index 0
```

Elements can be deleted:

```
del C[2]
```

and the length of a list can be examined by:

```
len(C)
```

There are many more operations that can be performed on lists that may be handy in certain situations but the ones we will use in this course are quite limited. The combination of a while (or for) loop and lists can be very powerful since the elements can be addressed one by one in a loop in a very short piece of code. For instance in the code below, a list is filled with temperatures starting from -50 up to 200 degrees in steps of 2.5 degrees:

```
C =[]
C_value = -50
C_max = 200
while C_value <= C_max
        C.append(C_value)
        C_value += 2.5
```

In the last line we have used the operator +=, which has the equivalent effect of

```
C_value = C_value + 2.5
```

but is a lot shorter.

1.5.4 For loops

For collections of data that are assembled in a list, we often want to perform the same operation, walking through the elements of that list. In Python (and other computer languages) there is a simple way to walk through such a list in a for loop. For instance a loop can be used to print all the elements in a list:

```
degrees = [0, 10, 20, 40, 100]
for C in degrees:
        print 'list element: ', C
print 'The list has', len(degrees), 'elements.'
```

This for loop runs over all elements in the list and in every pass the variable C takes the value of one of the elements of the list. Again, the for specification ends with a colon : and the lines that follow below with the same indentation are executed in each cycle of the loop. Using the for loop we can now take a list of temperature in Celcius and convert them to Fahrenheit:

```
Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
for C in Cdegrees:
        F = (9.0/5)*C + 32
        print C, F
```

Executing this piece of code gives a rather ugly looking table, but we have seen before that the way numbers are printed can be influenced in the print statement very easily:

```
Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
for C in Cdegrees:
        F = (9.0/5)*C + 32
        print '%5d %5.1f' % (C, F)
```

Now the temperature are both printed in a fixed field-width of 5 characters. The temperature in Celcius is an integer %5d, while the temperature in Fahrenheit is a floating point number that is printed with one decimal, %5.1f.

We have now encountered two different kinds of loops, for-loops and while-loops. These loops differ in the way they are implemented but in principle, either of the two can be used to produce any loop. In fact, in many programming languages only a single type of loop construct is available, and this works perfectly fine. The for-loop just above can be programmed using a while-loop with exactly the same result:

```
Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
index = 0
while index < len(Cdegrees):
        C = Cedegrees(index)
        F = (9.0/5)*C + 32
        print '%5d %5.1f' % (C, F)
        index += 1
```

This code is slightly longer any some would argue that it is less clear than the implementation with the for loop, but both give exactly the same result. Therefore, it is up to the programmer to decide which construct to use and it is advised to take the version that is the most intuitive for the specific application.

1.5.5 The range construction

A useful construction in combination with for-loops is the 'range' construction. Instead of typing the entries in the list of temperatures in Celcius we can generate this list using a for-loop. The range construction can work in several ways:

- range(n) generates intergers 0, 1, 2, ..., n-1

- range(start, stop, step) generates a list of integers start, start+step, start+2*step etc., up to, but not including stop. For example range(2, 8, 3) generates 2 and 5, but not 8.

- range (start, stop) is the same as range(start, stop, 1)

Using the range construct a for loop over a range of integers can be written as:

```
for i in range(start, stop, step):
        ...
```

And we can use this construction to generate the list of temperatures as before:

```
Cdegrees = []
for C in range(-20, 45, 5):
        Cdegrees.append(C)
```

The upper limit in this case needs to be large than 40 to ensure that 40 is included in the list. The range construction only generates a sequence of integers, which means that we can not use it directly to make a list of temperature that increase in steps of 2.5 degrees. We can still use the range construction but it requires an extra operation:

```
Cdegrees = []
for C in range(0, 21):
        C = -10 + i*2.5
        Cdegrees.append(C)
```

Finally, it is perfectly fine to include a for or while loop (or more than one) inside the list of statements that is executed inside another loop. This leads to so-called nested loops that are encountered very often in numerical methods:

```
Cdegrees = []
for x in range(1, 11):
        for y in range(1,11):
                z = x**2  + y**2
                print x, y, z
```

This code wil will print the value of z, for every combination of x and y, where x and y range from 1 to 10.

1.6 Structured programming and functions

So far all our programs have been in the form of a single list of statements, which is perfectly fine for short programs. When programs get longer, it becomes more complicated to follow the flow the program, especially when there are loops and nested loops with many statement inside. Moreover, in many cases the same sequence of statements appears many times in the same program. For these common tasks it is possible to define our own functions, and they can be called just like the built in functions or the functions that we import from a library.

As an example consider the code below that calculates and prints the factorial of the range of integer numbers from 1 to 10:

```
for n in range(1, 11):
        result = 1
        for m in range (1, n+1):
                result = result*m
                print n, result
```

While this is a very short piece of code it is already hard to imagine what the code is doing without going through every line and trying to reproduce the result. It would help greatly if we put some comments behind every statement but we can also make the code a lot more readable by introducing some 'structure' in the code itself. For instance, we can define a function that calculates the factorial, using the def statement:

```
def myfactorial(n):
        result = 1
        for m in range (1, n+1):
                result = result*m
                return(result)

for n in range(1, 11):
                print n, myfactorial(n)
```

In this code we have first defined a function called 'myfactorial' which uses a loop that calculates the factorial of the variable n. In the lines below, the function is called with the argument n and the result is printed. The last two lines of the code are perfectly clear now, without comments: we make a loop over integers from 1 to 10 and for each number we print a line with the number itself and the factorial of that number. The actual calculation of the factorial is done in a function, in this case a function that we have defined ourselves called 'myfactorial'. This function takes the number that we supply to it (in the variable n) calculates the factorial and returns the result to the main program. The introduction of a very general function to perform an often occurring calculation has two effect:

- We have a separate function that we can call any time we need a factorial. This happens to be a function that is also available in the module math: math.factorial, so in this case there was no need to define ourselves, but we can define any function we want.

- The program now has a more clear structure: it is split up into easily readable chucks, each performing their own little subtask. The task in a function stand on its own, it does not depend on the rest of the program, only the result is used in the rest of the program. At the same time the main program has become much more simple to read because the complication of the actual calculation of the factorial has been removed and put into a separate piece of code, with a convenient name that actually describes what is happening: the calculation of the factorial!

It should be noted that in large pieces of codes it is always advisable to split up the code into manageable parts, functions. This keeps the task for the programmer manageable since small part of code can be implemented separately, even by different programmers. When writing the main code of the program above (the last two lines) we just assume that there is a function that calculates the factorial and use it. We can go back later and actually implement this small subtask, or even better, get someone else to do it.

The names of the variables that are used inside a function have nothing to do with those that we use in the main program, and they also do not have to be the same. The following is equivalent to the program before:

```
def myfactorial(n):
        result = 1
        for m in range (1, n+1):
                result = result*m
        return(result)

```

```
for number in range(1, 11):
                print number, myfactorial(number)
```

It is also possible to define functions that take two arguments:

```
def myproduct(n, m):
        return(n*m)
```

1.7 Branching: if-else statements

In a computer program there are often decisions to be made, on basis of the results of calculations or the numbers that are read from a file etc. Decision can force to program to go into different 'branches'. If a certain condition is met we do one thing, if not, we do another thing. In Pythin this can be done using a an if-else statement which is very intuitive: it basically does what the code says:

```
if C < -273.15:
        print '%g degrees Celcius is non-physical!' % C
        print ' The Fahreheit temperature will not be computed.'
else:
        F = 9.0/5*C + 32
        print F
print 'end of program'
```

The two print statement on lines 2 and 3 are only executed if C < 273.15 is True, otherwise it is skipped and the code on lines 5 and 6 is executed. The final printing statement on line 7 is always executed since the indentation tells Pythin that it is not inside the block of statements in the if-else statement. The else can also be skipped :

```
if C < -273.15:
        print '%g degrees Celcius is non-physical!' % C
F = 9.0/5*C + 32
print F
print 'end of program'
```

In this case the conversion in Fahrenheit is not in the if-else block and is therefore always executed, regardless of whether it makes sense or not.

These statements have only given the choice between two options but it is intact possible using the elif keyword (short for else if) to 'branch' the program into as many different flows are needed:

```
if condition1:
        <block of statements>
elif condition2:
        <block of statements>
elif condition3:
        <block of statements>
else:
        <block of statements>
<next statement>
```

There are many other ways of using the if statement and some of these will appear in the code that is used in further chapter in this course. In most cases it is easy to understand how these contracts

work because as with a lot of statements in Python: they actually make some sense in normal language too. It is also a good execs to find your own background information about the if-else statement since the online documentation is virtually endless, including many examples. This is true for all other Python constructs discussed in this chapter.

1.8 Exercise

1.8.1 Height of a ball

For the first exercise you will reproduce the example presented in the section 1.2.3. You will therefore write a small program that automatically computes the height of a ball that falls vertically following

$$y = v_0 t - \frac{1}{2} g t^2 \tag{1.1}$$

and print the answer. You'll define the variable as shown in page 8. We now ask the question : How long does it take the ball to reach a certain height y_c ? Posing $y = y_c$ in the equation above and rearranging the terms we find

$$\frac{1}{2} g t^2 - v_0 t + y_c = 0 \tag{1.2}$$

You can solve this equation using the well known formula for a quadratic equation and find the expression of the two roots

$$t_{\pm} = \left(v_0 \pm \sqrt{v_0^2 - 2 g y_c} \right) / g \tag{1.3}$$

There is two times because the ball can reach y_c on its way up or down. Update your code so that it computes the values of t_{pm} for $y_c = 0.2$ Note: the square root function can be loaded from the *math* module.

```
1 import math as m
2 m.sqrt(4.)
```

1.8.2 Growth of money in a bank

Let p be a bank interest rate in percent per year. An initial amount A has then grow to

$$A_n = A_0 (1 + \frac{p}{100})^n \tag{1.4}$$

after n years. Make a program that computes and print on the screen how much money $A_0 = 1000$ euros have grown to after $n = 1, 2, 3, 4..., N$ years with a 5% interest rate. Do do that, make a **for** loop over the value of n and store the successive values of in a list $A_n = [1000, ...]$. Take $N = 50$. Your code should look like the following :

```
1 A0 =  ....
2 N = ...
3 p = ...
4 A = []
5 ....
6 for i in range(N):
7         A.append( .... )
8         print 'year = %d   \t  money = %1.6f' %(i,A[i])
```

Notice that we declare A as a list and then use the $append()$ command to add values into the list. Notice as well the use of the format $\%1.6f$ that forces Python to print the money with 6 numbers after the dot. Note as well the \t character that prints a tab.

Modify your code so that it stops when you became a millionaire !! You can do that with a **while** loop. Careful with the while loop that can run forever if the conditions is always true. It is usually a good idea to add an additional condition that we know will force the while loop to exit at some point. See the example below:

```
n = 0                   # counter
N_MAX = 1000            # maximum number of iterations
An = 0.                 # initial value

# while loop with two conditions
while (An < 1E6) and (n < N_MAX):

        An = ....                       # compute the new value
        A.append(An)                    # store it in A
        n += 1                          # don't forget to increment n !!!

# print the result
print 'It took %d years to become a millionaire !!' %( ... )
```

How long does it take to become a millionaire ? However what happen in the code above if you set N_MAX = 100 ? This is because the *print* statement is executed regardless of the fact that you re a millionaire or not. Modify the code below by introducing if ... $else$ branching statement to print the number of year it took to become a millionaire if you became one or to print that you never became a millionaire otherwise.

```
if(....):
        print 'It took %d years to become a millionaire !!' %( ... )
else:
        print 'you never became a millionaire :('
```

1.8.3 Celsius Fahrenheit Conversion

The formula for converting Fahrenheit degrees to Celsius reads

$$T_C = \frac{5}{9}(T_F - 32) \tag{1.5}$$

On the other hand converting Celsius to Farenheit can be done with

$$T_F = \frac{9}{5}T_C + 32 \tag{1.6}$$

Write two function $cel2far$ and $far2cel$ that implement these formulas for one temperature (each function takes one argument that is a floating point number). To verify the implementation, you can convert a Celsius temperature to Fahrenheit and then back to Celsius again. That is, you can check that a temperature c equals far2cel(cel2far(c)).

Update your code so that two function takes not a single temperature but a list of temperature $T_X = [10.1, -20.44, 40.23...]$. each function will contain a loop that goes over all the temperature contained in the list and will check that these temperatures are valid. Reminder, a Celsius temperature cannot

be lower than -273.15 and a Fahrenheit cannot be lower than -459.67. If some temperatures in the list are not valid, your program will print that these temperatures do not make sense and will not convert them.

1.8.4 Your first chemical engineering project: How to cook a perfect egg

As an egg cooks, the proteins first denature and then coagulate. When the temperature exceeds a critical point, reactions begin and proceed faster as the temperature increases. In the egg white the proteins start to coagulate for temperatures above 63 C, while in the yolk the proteins start to coagulate for temperatures above 70 C. For a soft boiled egg, the white needs to have been heated long enough to coagulate at a temperature above 63 C, but the yolk should not be heated above 70 C. For a hard boiled egg, the center of the yolk should be allowed to reach 70 C. The following formula expresses the time t it takes (in seconds) for the center of the yolk to reach the temperature T_y (in Celsius degrees):

$$t = \frac{M^{2/3} c \rho^{1/3}}{K \pi^2 (4\pi/3)^{2/3}} \ln \left[0.76 \frac{T_o - T_w}{T_y - T_w} \right] \tag{1.7}$$

Here, M, ρ, c, and K are properties of the egg: M is the mass, ρ is the density, c is the specific heat capacity, and K is thermal conductivity. Relevant values are $\rho = 1.038$ g cm^{-3}, c = 3.7 Jg^{-1} K^{-1}, and K = 5.4 10^{-3} W cm^{-1} K^{-1}. Furthermore, T_w is the temperature (in C degrees) of the boiling water, and T_o is the original temperature (in C degrees) of the egg before being put in the water. Write a function that take as input the different temperatures and the mass of the egg and return the time t.

The eggs can be stored in the fridge ($T_o = 4$ C) or at room temperature (T_o = 20 C). Furthermore we have two types of eggs: small ones with $M = 47$ grams and bigger ones with $M = 67$ grams. Compute the time it take to cook to obtain soft and hard boiled eggs with these two types of eggs stored at room temperature or in the fridge.

Chapter 2
Data Processing with Numpy, Scipy & Matplotlib

In the last chapter we have seen how to use computer programs to solve simple problems. In this chapter we will see dedicated methods geared toward scientific computing. Scientific computing in Python is quite simple thanks to specific modules called *numpy* and *scipy* that contains a large number of functions and methods to model and solve complex engineering problems that can be visualized using the module *matplotlib*.

2.1 Basics of Numpy

The *numpy* module provides a powerful linear algebra data structures to Pyhton. In particular it allows to construct vectors and matrices very easily and contains a multitude of linear algebra routines. In this section we present the main features of the numpy module. Additional functionality will be introduced throughout the next chapter when needed. You can also find all the information relative to the numpy module in the online documentation http://www.numpy.org

Like all modules, *numpy* needs to be imported before usage. Therefore to use the numpy module in your code the following line must be present before any call to any numpy routine

```
1 import numpy as np
```

We have here given a nickname, i.e. *np*, to shorten the calls to numpy. Feel free to given another nickname or not at all as you prefer.

2.1.1 Numpy arrays

Numpy provides essentially all the tools to manipulate vectors, matrices, etc .... The simplest way to create a vector is through the method *numpy.array*(). An easy way to create a vector is to pass to the array() method a Python list as

```
1 v = np.array([1,2,3],dtype=np.float64)
```

that creates an 1D array

$$v = \begin{pmatrix} 1.0 & 2.0 & 3.0 \end{pmatrix} \tag{2.1}$$

Note that we specify here the type of numbers, i.e. 64-bit floating point numbers. In the example shown above, all the numbers are integer, specifying the type as we've done here forces these integer to be converted into floats. Adding a dot after each number would also do the trick.

The array we've just create looks very similar to the Lists introduced in the last chapter. To understand the use and advantages of such arrays over Python lists let's examine the following snippet of code

```
import numpy as np

# define two Python Lists
a = [1.,2.,3.]
b = [4.,5.,6.]

# tranform these lists
# in arrays
A = np.array(a,dtype=np.float64)
B = np.array(b,dtype=np.float64)

# compute the sum
c = a+b
C = A+B

print c
print C
```

The sum of two list as in the variable c above leads to $c = [1,2,3,4,5,6]$, i.e, the concatenation and not the sum of the two list. Not exactly what we want for scientific computing. However using numpy arrays the sum becomes $C = [5,7,9]$ as expected. This example illustrate what numpy does: it provides numerical and mathematical operations for Python.

A large number of methods can be used to create 1D arrays. A few of them are reported in the snippet of code below

```
v = np.zeros(5)    # [0 0 0 0 0]
v = np.ones(5)     # [1 1 1 1 1]
v = np.arange(5)   # [0 1 2 3 4]

# a vector containing 100 elements ranging between -pi and pi
v = np.linspace(-np.pi,np.pi,100)
```

Similarly we can creates matrices with numpy for example with the command

```
mat = np.array( [ [1,2,3] , [4,5,6] , [7,8,9] ])
```

Note the extra set of brackets. This commands creates the matrix

$$\text{mat} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \tag{2.2}$$

We will see in details how to handle matrices in the next chapter.

2.1.2 Arrays indexing and slicing

As for the lists each element of an array can be accessed individually with its index. Reminder: the index starts at 0 !. We can therefore use loops to fill up arrays as in the example below

```
N = 100
X = np.linspace(0,np.pi,N)
Y = np.zeros(N)
for i in range(N):
        Y[i] = np.sin(X[i])
```

where we create a vector $Y = \sin(X)$ for $X = [-\pi; \pi]$. You can also access parts or 'slices' of an array by giving a range of index as in

```
Y = np.arange(20)
Y[10:20] = np.zeros(10)
```

This instructions create a vector $Y = [0, 1, 2, ..., 19]$ and then replace its last 10 elements by zeros. The general instruction to specify this range is

```
Y[start:end:increment]
```

where start/end define the first and last index in the range and increment the distance between two elements of the range. For example

```
Y[0:10:2]
```

specify the indexes [0,2,4,6,8], i.e. an increment of 2. If you do not specify the increment a value of 1 is assumed. Similarly if you do not specify the value of end a value of 0 is assumed. Similarly the default value of end is the last element. Hence

```
Y[:10]  # first 10 elements
Y[10:]  # last 10 elements
Y[:]      # all the elements !
```

defines the elements of Y from index 0 to 9 and from 10 to the last element. Finally the elements can be accessed from the end of the vector with negative indexes. The last, second to last and so on elements are then accessed with

```
Y[-1] # last element
Y[-2] # second to last
...
```

We can also slice matrices using the same syntax. The indexing of matrix elements follow the same logic than the indexing of arrays. Hence the command `mat[i,j]` give the elements of the matrix on the i-th line and j-th column.For example for the matrix given in eq. 2.2, the command `x = mat[1,2]` gives a value of 6 to the variable x. Reminder the index starts at 0, therefore mat[0,0] is the element on the first line and first column.

We can also access entire block of the matrix by defining range of indexes as for the 1D vectors. We can for example access en entire column/line of a matrix with the command

```
col_0 = mat[:,0]                # first column
lin_0 = mat[1,:]                # second line
```

We explore the manipulation of matrices in greater details during the next chapter.

2.1.3 Filling up arrays : Loop VS Vectorization

We can create arrays using different methods. A straightforward approach is to use loops as we' ve in the example at the begining of the section 2.1.2. However the use of loop is generally slow and must be replaced when possible with vectors operations also called vectorization. The equivalent vector implementation of the code above simply reads

```
1 N = 100
2 X = np.linspace(0,np.pi,N)
3 Y = np.sin(X)
```

While they leads to the exact same results the vector implementation is much faster. To illustrate that we have measured the time required to create the vector Y for different number of points.

	$N = 10$	$N = 1E3$	$N = 1E5$	$N = 1E7$
t_{loop}	3.2 1E-5	1.7 1E-3	1.6 1E-1	15.5
t_{vect}	2.0 1E-6	1.4 1E-5	1.2 1E-3	1.2 1E-1

The power of the vectorization comes from the architecture of the computer that are capable of processing multiple data through one instruction via their register. We will no go into the details but think about using vectorization as much as possible in your code. Most of the times the vetorization seems the most logical solution to create an array and therefore you will end up using vectorization without even noticing it. However sometimes a little thinking is necessary to harvest the power offer by vector operations.

2.1.4 A possible error : the copy issue

It occurs quite often that one makes a copy of an array (or a slice of it) and copy it in another array. This is example the case in the code below

```
1 X = np.arange(20).astype('float64')
2 Y = X[0:10]
3 Y += np.ones(10)
```

This code creates an array X and second one Y that contains the 10 first elements of X. Note here the use of the method $astype()$ to force the number to be floats. Note as well the use of the command $+=$ that adds a certain number (here 1) to all the elements of Y. However if we now print X we will see that its 10 first elements have also been incremented by 1 ! This sneaky issue is due to the way *numpy* handles memory on the computer. Many solutions exists to circumvent this issue as illustrated below

```
1 # these are example of safe copies
2 A = np.arange(20).astype('float64')
3
4 B = np.copy(X[:10])
5
6 C = np.array(X[:10], copy=true)
7
8 D = np.zeros(10)
9 D[:] = X[:10]
```

We have barely scratch the surface of what *numpy* is capable of. We will introduce more possibilities during the next chapters to complete the presentation of numpy. A companion module of *numpy*,

called *scipy* will also be introduced in the following. Scipy provides even more sophisticated methods for scientific computing, as for example the manipulation of sparse matrices, and is intensively used.

2.2 Plotting data with Matplotlib

The graphical visualization of numerical data is very important in scientific computing, to check and analyze the results of the simulations. The module Matplotlib is one of many libraries that adds plotting capabilities to Python. To use Matplotlib we will use here the pyplot library that need to be imported in the python code as

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

Matplotlib is a very large library, so large that entire books have been written to document the different styles and capabilities it offers. All the information of the module are nicely explained in its online documentation http://matplotlib.org/index.html. The documentation contains many examples and tutorials to help you mastering Matplotlib. We present here a few example of basic plots.

2.2.1 Simple Curves

A typical example for the plotting of data is reported in the snippet of code below

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  X = np.linspace(0,2,100)
5
6  # ploting the data
7  plt.plot(X,X**2,label='X^2',color='black',linewidth=3)
8  plt.plot(X,np.sin(X),'--',label='sin',color='blue',linewidth=2)
9  plt.plot(X,np.exp(X),'o-',label='exp',color='red',linewidth=2)
10
11 plt.xlabel('Time',fontsize=20)
12 plt.ylabel('F(t)',fontsize=20)
13
14 plt.title('My plot')
15 plt.legend(loc=2)
16
17 plt.show()
```

This code creates a vector X with 200 elements ranging between 0 and 2. It then plot X^2, $\sin(X)$ $\exp(X)$ with three different line style. This style can be controlled with several arguments. By default the plot uses plain lines (as for X^2). This can be changed with the arguments '–' and 'o-' that replace the plain line by a dashed line or a point-line. The color of the line can be modified with the argument *color* and its width by the argument *linewidth*. The label of the axis can be controlled with the command plt.xlabel() and plt.ylabel(). The legend of the plot need to be 'plotted' as well with the command plt.legend(). The argument of this commands specify the location of the legend on the graph. Finally the plot will be displayed only if the command plt.show() is present. This opens a window where the plot is displayed and can be saved and modified (to a certain extend). The code above will produce the plot presented in Fig 2.1

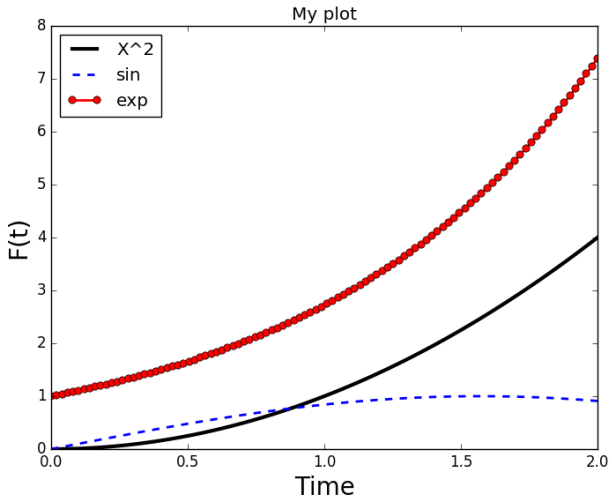


Figure 2.1: A plot showing a quadratic and sinus and an exponential on the same plot with different line style and legends.

2.2.2 Contour plot

Matplotlib can also plot more complicated data such as the contours of a given function. Let's assume we have a function

$$f(x,y) = (1 - \frac{x}{2} + x^2 + y^3)\exp(-x^2 - y^2) \tag{2.3}$$

and we want to plot the contours, i.e. the lines where the functions is equal to a certain value. This can easily be done using matplotlib as shown below.

```
1  import pylab
2  import numpy as np
3
4  def f(x,y):
5          f = (1.0 - 0.5*x + x**2 + y**3)*np.exp(-x**2-y**2)
6          return f
7
8  n = 256
9  x = np.linspace(-2,4,n)
10 y = np.linspace(-2,4,n)
11
12 X,Y = np.meshgrid(x,y)
13
14 C = pylab.contour(X,Y,f(X,Y),8)
15 pylab.clabel(C,inline=1)
16 pylab.show()
```

This code uses the module pylab which is included in matplotlib and provides more flexibility than pyplot. Note the use of the function np.meshgrid() that creates a grid of points encoded in the matrices X,Y. These matrices are then used to compute the function $f(x,y)$ using vectorization. A double nested loop over x and y is also possible to compute all the valued of f but would take much more time. This

code produces the plot shown in Fig 2.2.

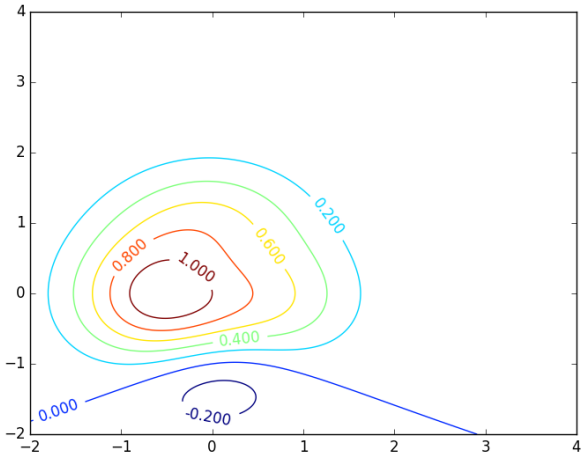


Figure 2.2: Contour plot of the function $f(x, y) = (1 - \frac{x}{2} + x^2 + y^3)\exp(-x^2 - y^2)$.

2.3 Linear Regression and Curve Fitting

A very common situation in chemistry and chemical engineering is that a certain quantity of a system is measured as a function of a certain parameter. Examples include measurements of the vapour pressure of a liquid as a function of temperature or the product yield as a function of temperature in a chemical reactor. In some cases there is a linear relation between the parameter that is varied and the quantity that is measured.

An example of such data is shown in Figure 2.3 on the left side where the yield of a chemical reaction is plotted as a function of temperature. The data clearly follow a linear relation. Trends in data, either linear or not, tend to be more useful if they can be described by a mathematical expression that approximately matches of 'fits' the data. In this section we discuss how fits to experimental data can be made using Python. We start of with linear regression, which is treated in considerable detail in order to illustrate the general concept and problems that may occur. The method discussed can be extended in a straightforward way to include fits to polynomials of any order. For these general regression fits we use a routine that is supplied by Numpy. Finally, we describe a routine from the Scipy module that allows fitting to any user-defined function.

2.3.1 A simple example: linear regression

The simplest example for making fits to experimental data is a linear regression fit. This kind of fit occurs very often in engineering and it illustrates the concepts very nicely. Therefore we treat it in some detail. The table below summarises some data from an engineer who had measured the yield of a certain reaction as a function of temperature. When the yield is plotted against temperature a clear linear trend emerges. This means that we will attempt to fit this data using a linear relation, $p(x_i) = c_1 x_i + c_0$.

T(degrees C)	Yield(%)
100	45
110	51
120	54
130	61
140	66
150	70
160	74
170	78
180	85
190	89

The goal of the linear regression, or the of the fitting to the data is to obtain an equation that describes the data as accurately as possible. The most important step is the selection of a measure that tells us how accurately the fit describes the data. One possibility is to minimise the absolute value of the difference between the data points and the fitted line. This turns out to be difficult in practice. A common and much more convenient choice is to minimise the sum of the square of the differences between the linear fit and the data points, E_{ls}:

$$E_{ls} = \sum_{i=1}^{n} (y_i - p(x_i))^2 \tag{2.4}$$

The function $p(y_i)$ can be a polynomial of any order but as mentioned above we will restrict ourselves to a linear function. We can therefore replace the function $p(y_i)$ by the linear equation in terms of

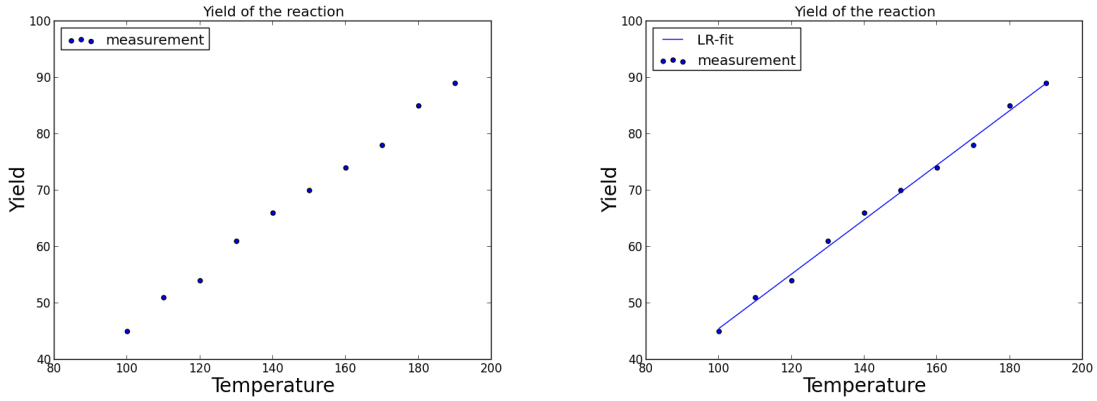


Figure 2.3: Scatterplot of the data in the table above(left), and the same scatterplot with the linear regression fit(right).

c_0 and c_1. The goal of linear regression is to minimise E_{ls}, i.e. to determine the values of the coefficients for which E_{ls} is smallest. Since 2.4 is a quadratic equation we know that the minimum occurs where the derivative of this function is zero. Therefore, we take the derivatie of equation 2.4 with respect to the two unknown parameters:

$$\frac{\partial E_{ls}}{\partial c_0} = -2\sum_{i=1}^{n}(y_i - c_1 x_i - c_0) = 0, \tag{2.5}$$

$$\frac{\partial E_{ls}}{\partial c_1} = -2\sum_{i=1}^{n}((y_i - c_1 x_i - c_0)\cdot x_i) = 0. \tag{2.6}$$

These equations can be simplified noting that constants can be factored out of the sum, which gives:

$$\sum_{i=1}^{n}(y_i) = c_1\sum_{i=1}^{n}(x_i) + n\cdot c_0, \tag{2.7}$$

$$\sum_{i=1}^{n}(y_i x_i) = c_1\sum_{i=1}^{n}(x_i^2) + c_0\sum_{i=1}^{n}(x_i). \tag{2.8}$$

The results is two linear equations with two unknown variables, c_0 and c_1 that can be solved easily by hand, giving:

$$c_1 = \frac{\sum_{i=1}^{n}(y_i x_i) - \frac{\sum_{i=1}^{n}(x_i)\sum_{i=1}^{n}(y_i)}{n}}{\sum_{i=1}^{n}(x_i^2) - \frac{(\sum_{i=1}^{n}(x_i))^2}{n}} \tag{2.9}$$

$$c_0 = \frac{\sum_{i=1}^{n}(y_i) - c_1)\sum_{i=1}^{n}(x_i)}{n} \tag{2.10}$$

Although these equations look rather fearsome, in a computer program it is very easy calculate all the sums, especially when using Numpy array routines. In the code below the sums are evaluated using the numpy routine numpy.sum, while the product of the elements in the arrays are evaluated using numpy.dot. After sorting out the sums the sums, the evaluation of the two coefficients and hence the best possible linear fit is calculated in two lines of code (lines 16 and 17).

After calculation the best possible linear fit the code prints the two coefficients to the screen and plots both the experimental data (using matplotlib.scatter) and the linear fit. The output of the program is shown in Figure 2.3 (left) and it is clear that it yields a nice linear fit. The resulting fit is the best

possible in the sense that it minimises the least square difference between the measured data and the fitted line.

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

# creating numpy arrays, fill them with data
temp = np.array([ 100., 110., 120., 130., 140., 150., 160., 170., 180., 190.])
yld = np.array([ 45., 51., 54., 61., 66., 70., 74., 78., 85., 89.])
N = np.size(temp)

# first we calculate all sums
Xsum = np.sum(temp)
Ysum = np.sum(yld)
XYsum = np.sum(np.dot(temp, yld))
X2sum = np.sum(np.dot(temp, temp))

# working out the least square linear fit
c1 = (XYsum - Xsum*Ysum/N)/(X2sum - Xsum**2/N)
c0 = (Ysum - c1*Xsum)/N

print c0, c1

regression = c0 + c1*temp

plt.plot(temp, regression, label='LR-fit')              #plotting the fitted line
plt.scatter(temp, yld, alpha=1.0, label='experiment')   #plotting experimental data
plt.xlabel("Temperature", fontsize=20)
plt.ylabel("Yield", fontsize=20)
plt.legend(loc=2)
plt.title('Yield of the reaction')
plt.show()
```

The approach used here can be easily extended to polynomials of any order and the evaluation is equally simple. In fact, as mentioned above, regression fits of this type result in sets of linear equations that can be combined into a matrix notation, in this case:

$$\begin{pmatrix} n & \sum_{i=1}^{n}(x_i) \\ \sum_{i=1}^{n}(x_i) & \sum_{i=1}^{n}(x_i^2) \end{pmatrix} \cdot \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{n}(y_i) \\ \sum_{i=1}^{n}(y_i x_i) \end{pmatrix} \tag{2.11}$$

The solution of such sets of linear equation is the subject of the next chapter, where methods to solve them are treated in detail. The extension to polynomials of higher order is very straightforward in the way and numpy has very efficient routines for solving such systems. In the code below the matrices A and y are set up to solve the coefficients c_0 and c_1. The results and the coefficients are exactly the same as for the code above where we explicitly wrote the calculation of the coefficients.

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

# creating numpy arrays, fill them with data
temp = np.array([ 100., 110., 120., 130., 140., 150., 160., 170., 180., 190.])
yld = np.array([ 45., 51., 54., 61., 66., 70., 74., 78., 85., 89.])
```

```
7
8  # first we calculate all sums
9  Xsum = np.sum(temp)
10 Ysum = np.sum(yld)
11 XYsum = np.sum(np.dot(temp, yld))
12 X2sum = np.sum(np.dot(temp, temp))
13
14 #Setting up the arrays
15 A = np.array([[N, Xsum], [Xsum, X2sum]])
16 f = np.array([Ysum, XYsum])
17
18 #This is where the set of linear equations is solved by numpy
19 c = np.linalg.solve(A, f)
20
21 print c
22
23 regression = c[1]*temp + c[0]
24
25 plt.plot(temp, regression, label='LR-fit')              #plotting the fitted line
26 plt.scatter(temp, yld, alpha=1.0, label='experiment')   #plotting experimental data
27 plt.xlabel("Temperature", fontsize=20)
28 plt.ylabel("Yield", fontsize=20)
29 plt.legend(loc=2)
30 plt.title('Yield of the reaction')
31 plt.show()
```

2.3.2 General polynomial regression with Numpy

While it is very instructive to write the code for fitting a function to experimental data it is usually not necessary. In many software packages (Origin, Igor, MATLAB) there are standard routines available for fitting functions to any kind of data. Similarly, in the Python modules Numpy and Scipy there is a variety of routines available for curve fitting. In the section above we have written our own code for linear regression of any data, no matter how large the data set is. For this type of fit there is a simple routine available in numpy, called numpy.polyfit. The code below uses this routine for the data that we have specified above and running the code will result is exactly the same coefficients and plots as we have seen above.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  # creating numpy arrays, fill them with data
5  temp = np.array([ 100., 110., 120., 130., 140., 150., 160., 170., 180., 190.])
6  yld = np.array([ 45., 51., 54., 61., 66., 70., 74., 78., 85., 89.])
7
8  regression = np.polyfit(temp, yld, 1)
9  print regression
10
11 regressionvalues = regression[0]*temp + regression[1]
12
```

```
13 plt.plot(temp, regressionvalues, label='LR-fit')                #plotting the fitted line
14 plt.scatter(temp, yld, alpha=1.0, label='experiment')   #plotting experimental data
15 plt.xlabel("Temperature", fontsize=20)
16 plt.ylabel("Yield", fontsize=20)
17 plt.legend(loc=2)
18 plt.title('Yield of the reaction')
19 plt.show()
```

2.3.3 Fitting to any function using Scipy

When the data that you need to fit is not described by a polynomial. The routine optimize.curve_fit from scipy can be used to fit any function. This function can be anything, it does not have to be an normal analytical function. As shown in the example below, you can defined your own function (lines 5-6) and then just use that, on line 12, to fit data points to it. In this example the data point are generated using the same function and on line 10 a random gaussian noise is added. The latter is just for making the data a bit noisy for the purpose of the example. The resulting (noisy) data with the exponential fit are shown in Figure 2.4.

The optimize/curve_fit routine return two things: the optimised coefficients belonging to the function in an array 'popt' and the covariance matrix 'pcov' that contains information about the accuracy of the fit. As said before, this routine will work for any function that you define yourself and is therefore very useful for fitting experimental data.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from scipy.optimize import curve_fit
4
5  def func(x, a, b, c):
6      return a * np.exp(-b * x) + c
7
8  x = np.linspace(0,4,50)
9  y = func(x, 2.5, 1.3, 0.5)
10 yn = y + 0.2*np.random.normal(size=len(x))
11
12 popt, pcov = curve_fit(func, x, yn)
13
14 print popt
15
16 plt.plot(x, func(x, *popt), 'r-', label="Fitted Curve")         #plotting the fitted line
17 plt.scatter(x, yn, alpha=1.0, label='Noisy data')       #plotting experimental data
18 plt.xlabel("X", fontsize=20)
19 plt.ylabel("Y", fontsize=20)
20 plt.legend(loc=1)
21 plt.title('curve_fit example')
22 plt.show()
```

2.3.4 Some additional remarks on fitting

The examples shown this section have consisted of quite nice data that accurately follows a certain trend. This is not always the case and the correct selection of data requires some attention. One of the

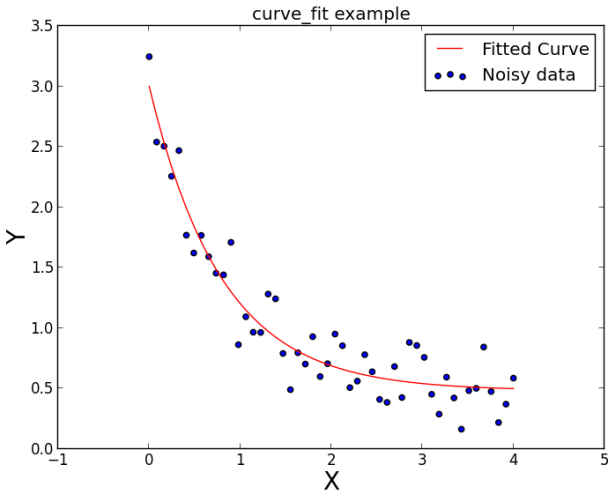


Figure 2.4: Exponentially decaying trend fitted with the general curve_fit routine from scipy.

very general issues that are true for all curve fits is the fact that each point weight equally in the fit. This means that if there is one point out of 10 that is very far off the trend, for instance a linear trend, the result can be a very bad linear fit. And example is shown in Figure 2.5 where on of the points in the linear regression example from above is moved way off the linear trend. This results in a line that tries to include this point, resulting in a line that clearly does not describe the data. A solution to this is data selection before the fitting procedure is performed.

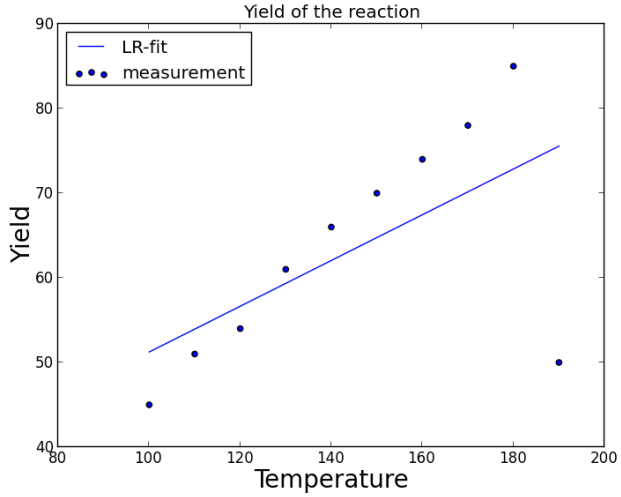


Figure 2.5: Illustration of the effect of outliers. One point that is far off the linear trend can make the resulting linear trend very bad.

A related issue may occur in the fitting of nonlinear functions, for instance an exponential function. Since in almost all curve fitting routines the sum of the square of the difference between the data and the fitted curve is minimised, it is the absolute difference that plays a role. For an exponential function

of which the values can span several orders of magnitude this may result in a poor correspondence in the tail of the exponential, where the absolute differences may still be small but the relative difference can be very big. As a result, after fitting, the exponential fit will follow the high points with reasonable accuracy but plotting it on a log scale typically shows quite severe deviations in the tail of the exponential.

Finally, fitting to a general function is something that can be done quite easily with a wide variety of software packages, but it is usually a good idea to think about it for a few seconds before diving into it. As an example you may think about the exponential fit mentioned above, for instance the rate of charge transfer in a donor-acceptor molecule as a function of distance:

$$k_{CT}(d) = A_0 e^{\frac{-\beta}{d}}, \tag{2.12}$$

which can easily be recast in the form of a linear equation, plotting the natural logarithm of the rate against the inverse of the distance:

$$\ln k_{CT} = \ln(A_0) - \beta \frac{1}{d}. \tag{2.13}$$

Fitting of this linear relation can easily be done, for instance by linear regression as described in detail above. The result will in most case be a more evenly weighted fit where the low points are better described than when just fitting an exponential directly. The latter is particularly relevant when the correspondence to an exponential is not perfect, as is usually the case.

2.4 Statistical analysis of data

The rigorous statistical analysis of data is a critical aspect of of real world science and engineering problems. Experimental results often comes in the form of a gigantic data set that needs further processing to extract meaningful information. There is many software that are dedicated to such analysis, among other we can cite IBM SPSS or the open source package R. Powerful tools have also been developed to perform statistical analysis directly within Python. This is for example the case of the module scipy.stats() that contains a vast number of methods able to process complex data. The motivation of briefly introducing this tool is that you will most likely have to perform statistical analysis of data in your Bachelor project but also in virtually any other activities you will be part of in your future.

2.4.1 Reading data from a file

Before data can be processed in your Python code, we first need to read the data (and later maybe write the data). Several options are available to read/write data from file depending on its format. The most common formats are space-separated values where each line contains several numbers separated by a blank space, and comma-separated values (csv) where the numbers are separated by commas. This format is very popular to share data between different programs. Excel can for example export spreadsheets in .csv format.

Space-Separated Values Data files containing formatted with space separated values can be easily be imported in your program with the *numpy.readtxt*(). Similarly you can write a data to a file with the command *numpy.savetxt*(). For example let's assume you want to import the data below in your program:

```
0.0000000000000000e+00 1.0335112030082749 26e-02 2.3163270997060947 39e-01 6.0032493924442098 53e-02
1.0000000000000000e+00 7.1770642324816813 50e-01 9.5619307680399456 83e-01 1.5938776623920480 89e-01
2.0000000000000000e+00 1.6175148673169381 65e-02 8.7028794897329153 63e-01 2.9816472290973206 06e-01
3.0000000000000000e+00 4.5260251182925070 88e-01 5.1292243505104606 62e-01 5.6523406715017410 21e-01
4.0000000000000000e+00 8.4636471310525507 32e-01 8.8580085603530700 49e-01 2.8846465387684050 44e-01
5.0000000000000000e+00 7.1639560427795256 90e-01 4.5115339999550385 65e-01 2.7086836231493671 70e-01
6.0000000000000000e+00 1.4061054551760143 74e-01 7.2723287511373468 92e-01 6.2563011564523796 08e-01
7.0000000000000000e+00 7.9005715562627498 73e-01 7.2388472570913708 90e-01 5.8381454385991193 86e-01
8.0000000000000000e+00 3.0173647794478275 50e-01 2.3791265874998046 57e-01 5.0269957187007441 31e-02
9.0000000000000000e+00 2.2155545192195424 95e-02 5.5169534811500187 12e-01 6.7399237841926828 98e-01
1.0000000000000000e+01 3.9192609735072458 93e-01 3.7180977583135099 71e-01 3.5509383042649467 21e-01
1.1000000000000000e+01 8.5308295800209810 18e-01 9.7336420147661051 33e-01 1.4965367325656075 97e-01
1.2000000000000000e+01 5.0043047835602538 78e-01 5.0495602166505948 46e-02 6.5969266891456546 10e-02
1.3000000000000000e+01 6.9677428922567852 25e-01 6.5467209289717009 92e-01 1.2390910945722888 31e-01
1.4000000000000000e+01 7.6486857977726152 58e-01 4.8737347619647253 26e-01 4.6663974891056614 34e-01
1.5000000000000000e+01 5.9715878120536391 99e-01 8.3627236121393955 24e-01 2.7953488167767537 31e-01
```

Figure 2.6: Example of a data file using space separated values formatting.

This data file contains on the first column a list of number going from $0, 1, 2...., N$ and a bunch of random numbers on the rest of the columns. The name of the file is supposed to be rand.dat. This data can be loaded into a variable using the command

```
1 matrix_data = np.loadtxt('rand.dat')
```

This command will create a matrix named *matrix_data* that contains all the numbers present in the file. You can access slices of this data using the slicing methods seen above. For example the first column can be accessed using the command

```
1 first_col = matrix_data[:,0]
```

NOTE : Do not forget about the copy issue presented in section 1.

Similarly the values contained in a matrix can be written to a file using the command

```
1 np.savetxt('my_file.dat',mat_data)
```

This command will automatically create a file called my_file.dat that contains all the elements of the matrix *mat_data*. The save and read commands have a lot of other argument as for example the format used to print each number (%f,%e,...). The default options work just fine in most cases.

Comma-separated values If the space separated values seems the most logical format, comma-separated values formatting is still pretty popular. These files cannot be imported via the np.readtxt() command as they contains commas. A typical .csv file is shown on the figure below

```
Header of the file containing description
0.000000e+00, 6.535564e-01, 3.327769e-01, 9.851758e-01, 7.217739e-01, 3.985093e-01, 2.376603e-01
1.000000e+00, 8.236840e-01, 7.473312e-01, 1.462667e-01, 9.015324e-01, 7.048819e-01, 4.159476e-01
2.000000e+00, 8.249669e-01, 8.195542e-01, 7.300686e-01, 8.758938e-01, 8.357664e-01, 7.855417e-01
3.000000e+00, 9.460394e-01, 3.145725e-01, 9.307185e-03, 2.341569e-01, 9.375186e-01, 9.428466e-01
4.000000e+00, 3.648219e-02, 6.920992e-01, 6.473924e-01, 6.579842e-01, 8.623746e-01, 5.869553e-01
5.000000e+00, 5.241851e-01, 1.930601e-01, 4.060758e-01, 5.901589e-01, 3.301334e-01, 9.698347e-01
6.000000e+00, 6.163998e-01, 7.646866e-01, 4.380311e-01, 7.652511e-01, 9.135752e-01, 6.638132e-01
7.000000e+00, 3.436248e-01, 8.594320e-01, 8.622408e-01, 7.950208e-01, 4.800632e-01, 8.478111e-02
8.000000e+00, 9.048130e-01, 7.348746e-01, 3.123458e-01, 8.974092e-02, 7.681376e-03, 7.785226e-01
9.000000e+00, 1.944695e-01, 1.542351e-01, 1.375626e-02, 4.383933e-01, 4.966759e-01, 9.782849e-01
1.000000e+01, 5.654962e-01, 2.768553e-01, 2.410246e-01, 7.177440e-01, 4.258534e-02, 2.880694e-01
1.100000e+01, 7.482819e-01, 9.730457e-01, 9.017000e-01, 5.473594e-01, 7.788328e-03, 9.113045e-01
1.200000e+01, 5.149410e-01, 8.131253e-01, 9.020773e-01, 3.929632e-01, 6.835638e-01, 3.013046e-01
1.300000e+01, 8.847980e-01, 5.225700e-01, 9.801060e-01, 2.429533e-01, 2.004973e-01, 4.673773e-01
1.400000e+01, 2.731030e-01, 8.134976e-01, 3.372684e-01, 6.762793e-01, 8.901596e-01, 3.182948e-01
1.500000e+01, 4.579045e-01, 5.340556e-01, 3.586887e-02, 9.951234e-01, 3.073260e-01, 6.100090e-01
```

Figure 2.7: Example of a data file using comma separated values formatting.

This file contains a header, i.e. a first line containing a description of the data it contains. Then the values are separated by a commas on the following lines. The values contained in this file can be loaded in a variable using the command numpy.genfromtxt()

```
matrix_data = np.genfromtxt('data.csv,delimiter=',')
```

This command allows loading data in a generic format in a numpy array by specifying the delimiter used in the file. If the Header line starts with a "#" sign it will be ignore during the loading. If it does not begin with a '#' symbol, the loading of the data will most likely fail. Therefore one must always check his *.csv file before trying to load it with np.genfromtxt(). A dedicated module csv has been implemented for Python (https://docs.python.org/2/library/csv.html). This module offers advanced reading and writing capabilities that are beyond the scope of this introduction course. Besides, the syntax required to operate this module is a bit complicated for Python beginners.

General file format It is always possible to manually load all the values contained in a file using the standard input/output functionality of Python. This can be be done by reading the file line by line and storing the values contained on each line in a List or a numpy.array. This method can be used to read any type of files with any formatting (or no formatting at all). We illustrate this approach in the snippet of code below that reads a .csv file with a header.

```
import numpy as np

def read_csv(filename):

        # open the file
        f = open(filename,'r')

        # read all the lines
        data_string = f.readlines()
        nLines = len(data_string)

        # close the filename
        f.close()

        # declare a list where we store the data
        data_float = []

        # for all the lines but the first
        # since it s a header
        for iL in range(1,nLines):

                # split the line at each comma
                line = data_string[iL].split(',')

                # append the float to the list
                data_float.append([float(x) for x in line])

        # return the float
        return np.array(data_float)
```

A lot of explaining is now required. As you might have guessed, the command `open()` opens the file called `filename` and the option 'r' sates that the file is for reading only (This is the de-

fault option). Other options such as 'w' allows writing to a file but we won't see that in this chapter. The method `f.readlines()` allows reading all the lines contained in the file and store them in the variable `data_string`. This variable is a 1D array where each element correspond to one line. Hence `data_string[0]` is the first line stored as a string of characters and so on. The next important command in the code above is `line = data_string[iL].split(',')`. This commands takes one individual line (`data_string[iL]`) and split it at each comma . For example the string '1.00, 2.00, 3.00' will become a 1D vector containing 3 separate elements : ['1.00', '2.00', '3.00']. All these elements are still strings of character and not numbers. To convert these strings we use the function `float(x)` that convert a string to a float if possible. Note that we use a rather Pythonic syntax `data_float.append([float(x) for x in line])`.

The method of reading line by line and processing the data in the form we want is a bit more complicated that the python oneliners presented above but allows a much greater flexibility. Besides it is sometimes impossible to use the `np.loadtxt()` or `np.genfromtxt()` and the method shown here is the only solution capable of doing the job correctly.

2.4.2 Simple statistics and histograms

Once we have loaded the data into a proper numpy array we can start processing its elements. We assume that in the file presented above, the first column represent the time of the experiment and the each following column the value of certain physical quantity measured at these times. We can therefore plot the time evolution of this data using matplotlib as shown in the figure 2.8.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  DATA = np.loadtxt('measurements.dat')
5  X = DATA[:,0]
6  Y = DATA[:,1:]
7
8  plt.plot(X,Y[:,1],color='#FF003F',label='1')
9  plt.plot(X,Y[:,0],color='#00AFFF',label='1')
10
11 plt.xlabel('time',fontsize=15)
12 plt.ylabel('Measurements',fontsize=15)
13 plt.show()
```

The data looks very noisy and not much information can be extracted from inspection of the plots. We can just observe that the mean value of the blue curve seems lower than the red ones. To obtain a more quantitative analysis of the data we therefore need to compute statistical values. Different modules such as Numpy and Scipy offers a vast library of statistical function that can be accessed easily. For example the mean values of the elements of a vector X defined as

$$\mu = \frac{1}{N}\sum_{n=0}^{N-1} X[n] \tag{2.14}$$

can be calculated via the two methods

```
1  m = np.mean(X)
2  m = X.mean()
```

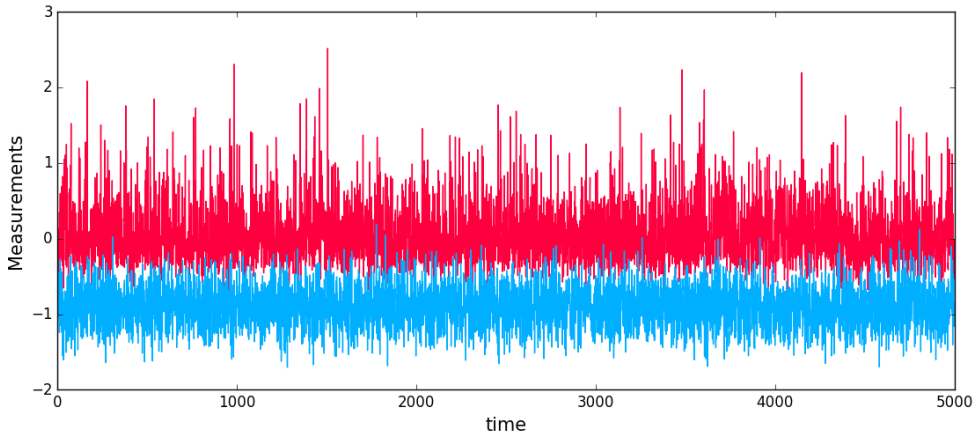


Figure 2.8: Time evolution of two measurements contained in the data file

providing that X is a Numpy array. Similar syntax exist for the minimum and maximum values contained in an array (`min` and `max`). The variance and standard deviation of a time series, defined by

$$\text{var} = \frac{1}{n}\sum_{n=1}^{N-1}(X[n]-\mu)^2 \qquad \sigma = \sqrt{\frac{1}{n}\sum_{n=1}^{N-1}(X[n]-\mu)^2} \tag{2.15}$$

where μ is the mean value of the vector. The variance and standard deviation can be calculated with the commands `X.var()` and `X.std()` ou the equivalent syntax `np.var(X) np.std(X)`. These quantities define how disperse the values are around the mean values and are extremely important in engineering to evaluate the quality of product or processes or the fidelity of a given apparatus. Finally the median of a time series, i.e. the value that is superior to half of the number contained in the series and inferior to the second half is given by `np.median(X)`.The little snippet of code below show basic usage of these functions

```
nLine, nCol = Y.shape
for iCol in range(nCol):
        print '\n === Data Column %02d' %(iCol+1)
        print("\tMean : %1.6f" %(Y[:,iCol].mean()))
        print("\tMedian : %1.6f" %(np.median(Y[:,iCol])))
        print("\tMinimum : %1.6f" %(Y[:,iCol].min()))
        print("\tMaximum : %1.6f" %(Y[:,iCol].max()))
        print("\tVariance : %1.6f" %(Y[:,iCol].var()))
        print("\tStd deviation : %1.6f" %(Y[:,iCol].std()))
```

This code will produce the values reported in the table below for the first two columns.

	mean	median	min	max	var	std
0	-0.89	-0.89	-1.17	0.19	0.075	0.27
1	0.06	-0.01	-0.75	2.51	0.14	0.38

The calculation of these quantities confirm our observations. The mean values of the first two columns are -0.89 and 0.06 respectively. In addition the variance of the first time series is smaller than for the second indicating a smaller spread of values around the mean. Finally we can see that the mean

and median values are equal for the first time series but not for the second.

To understand what this means It is very useful to plot these data not as a function of the time but as an histogram. Matplotlib offers an easy interface to such histogram. Hence the command `plt.hist(X,bins=nbin,...)` will directly an histogram of the data contained in X using a certain number of intervals (or bins) to compute the count. This is illustrated in the snippet of code below.

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

# inport the data
DATA = np.loadtxt('measurements.dat')
X = DATA[:,0]
Y = DATA[:,1:]

# plot histograms
plt.hist(Y[:,0],bins=50,facecolor='#00AFFF',alpha=0.5)
plt.hist(Y[:,1],bins=50,facecolor='#FF003F',alpha=0.5)
plt.ylabel('Count',fontsize=15)
plt.xlabel('Measurements',fontsize=15)
plt.show()
```

You can see in this code that the command hist() accept different arguments that can be used to specify the color of the histogram or its level of transparency (alpha). This code plot the figure 2.9.

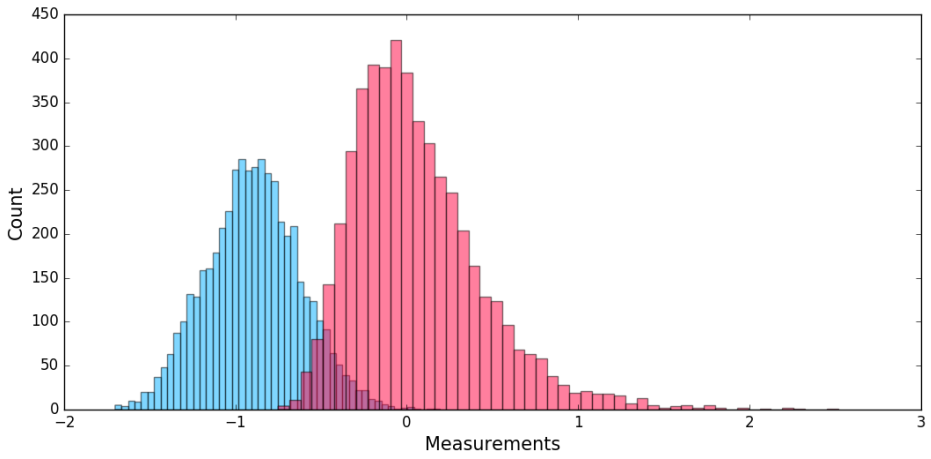


Figure 2.9: Histogram of the values contained in the first two columns of the data file

Plotting the histogram allows to identify the main differences between the two data set. The blue distribution seems symmetric around its mean value which explains why its mean is identical to its median. However the red distributions is not symmetric, leading to the difference between its mean and median value. You can also see a long tail in the high value number. These two distributions consequently look like a normal and Poisson distribution respectively.

2.4.3 Quartile and boxplots

In descriptive statistics, i.e. the quantitative description of a collection of information, the quartiles of a data sets are the three points that divides the the data set in four equal groups. The quartiles are a generalization of the median. The median splits the data set in half with half the number being smaller than the median and the other half larger. The median is hence the second quartile point noted Q_2. The first quartile (Q_1) splits the lowest 25% of the data from the highest 75 %. Similarly the third quartile Q_3 splits the lowest 75 % of the data from the highest 25 %. An illustration of the quartile is represented in the figure 2.10.a for a normal distribution.

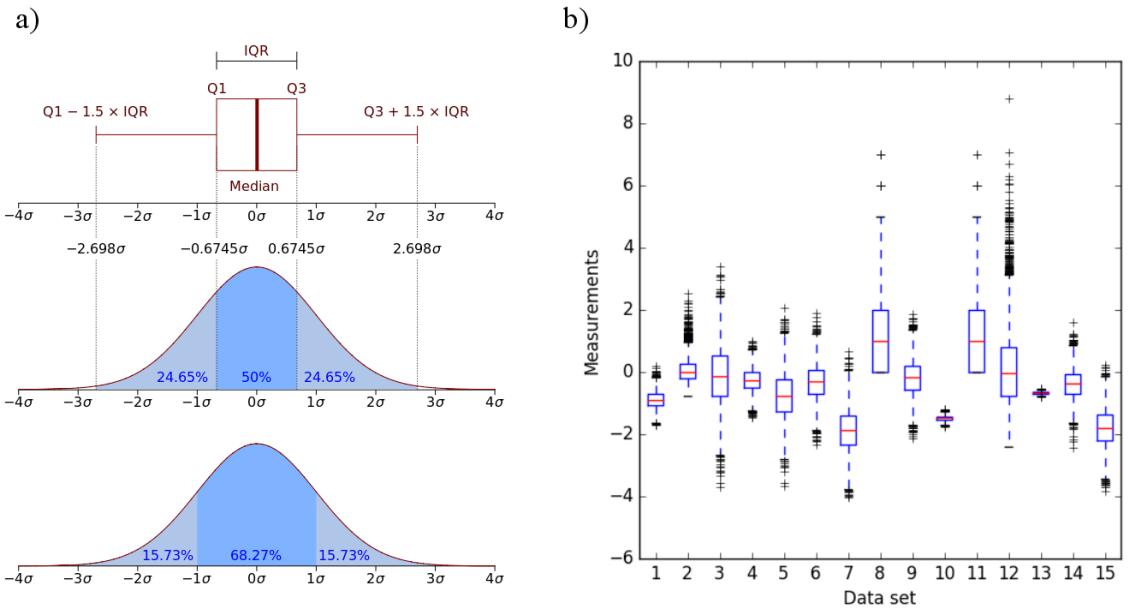


Figure 2.10: a) Graphical representation of the quartile for a normal distribution (from the wikipedia page about quartile) b) Boxplot of the different data sets contained in the data file

The quartiles of a data set X can be computed via the commands

```
1 Q1 = np.percentile(X,25)
2 Q2 = np.percentile(X,50)
3 Q3 = np.percentile(X,75)
```

The last number have to be an number between 0 and 100 that defines the interval we desire. We have chosen here 25, 50 and 75 to obtain the quartiles but other numbers are possible. The quartiles are generally plotted using the so called box-plot. This plot type is a convenient way of graphically depicting a distribution of points. As illustrated in the figure 2.10.a, the box-plot consists of a box extending from Q_1 to Q_3 with a solid line marking Q_2. The plot also shows so called whiskers that extend down to $Q_1 - 1.5 \times \text{IQR}$ and up to $Q_3 + 1.5 \times \text{IQR}$ where IQR is the interquartile range defined as $\text{IQR} = Q_3 - Q_1$. Additional points can be plotted to mark the elements outside the whiskers range. This complicated plot is accessible with one single command in python as shown in the snippet of code below.

```
1 plt.boxplot(Y)
```

```
2 plt.xlabel('Data set')
3 plt.ylabel('Measurements')
4 plt.show()
```

where Y is here a matrix where each column contains a separate data set. This commands directly create the boxplot shown on the figure 2.10.b. This type of plots is very common in descriptive statistics as it gives a general feeling of the distribution in a very compact way. Plotting the corresponding histograms would lead to a very complicated figure will the histograms overlapping each other and therefore impeding the analysis of the data sets. The box-plot allows to represent several distributions at the same time without loosing information.

2.4.4 Comparing incomplete data set with t-tests

When analyzing real world or experimental data we often have to do with a very limited number of points to define a distribution. Collecting data is most often hard and/or expensive and we therefore never have enough data points to obtain histogram as smooth as the ones represented in the figure 2.9. For example let's assume that we have the measured the concentration of dissolved oxygen in at different points of a river. We have therefore gathered 25 measurements at both locations. The histograms of these measurements are show in the figure 2.11.

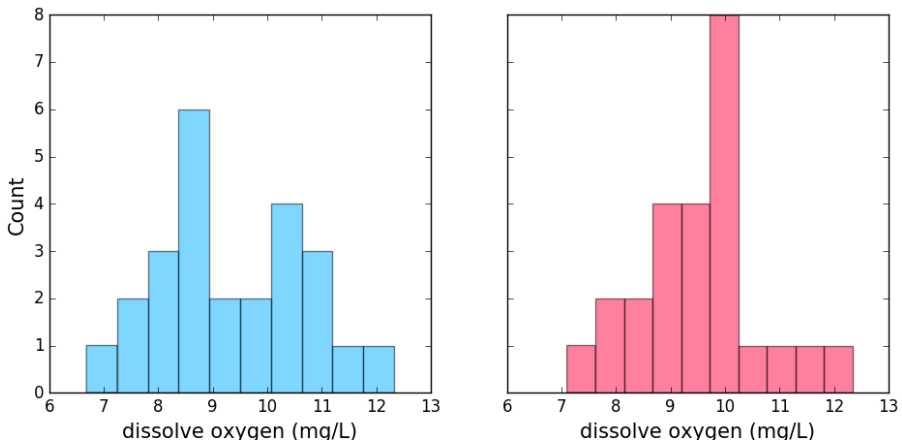


Figure 2.11: Dissolved oxygen concentration at two different points of a river. Due to the small number of data points in the sets the distributions are nor really clear.

An important question that arises very often when comparing different data set is whether or not these two distributions have the same mean or not. In other word, can we say that the average temperature at these two points are really different or not ? If more data would be collected we could answer this question easily as the mean of the two data sets would converge to the same value or not. However if we cannot gather more data (as it is often the case) we have to answer the question differently. This can be done using a t-test, which compares two (or more) data sets and compute the probability that the true mean of the two sets is equal or not. Computing this probability is a very complex task that. However it is made very easy with the Scipy command

```
1 t, prob = scipy.stats.ttest_ind(data1,data2,equal_var=True)
```

where `data1` and `data2` are the two data set to compare. The last argument specify if the variance of the two set is supposed to be the same or not. When comparing data measured in the same conditions we can assume that they are. However when comparing experimental data with values from the literature

of from theoretical calculations is preferable not to make that assumption and set this argument to False. The value t return by this command is hard to interpret. However the value of *prob* directly gives the probability that the two means are the same. If $prob < 0.05$ we can say with 95 % confidence that the two means are different. Similarly if $prob > 0.95$ we can say with 95 % confidence that the two means are the same. In the case presented above the probability is 0.70. It is therefore difficult to assess if the two means are the same or not and more data point must be gathered before making any strong statement.

2.4.5 Subjective data and z-score

In some cases the data collected is somehow subjective. This is for example the case if we ask 100 people to rate 20 different movies on a scale from 1 to 10. Some people would give very good grades even to the movie they didn't really like. Other would be more critical and gives scores ranging only between 4 and 6 to all movies. On the contrary some people will use the entire spectrum of grades. When we have to analyze such subjective distribution it is important to normalize the scores using the so-called z-score also known as standard score. Let's X be all the grades given by a single person to the 20 different movies. The z-score of X is given by

$$X_Z = \frac{X - \mu}{\sigma} \tag{2.16}$$

where μ and σ are the mean and standard deviation of the distribution. Thus the z-score of of each person are now centered around 0 and have a unit standard deviation. Good movie will gather positive z-score and bad ones negative z-score. A z-score of 2.0 means that the movie is 2 standard deviation better than average. A data set can automatically be transferred to a z-score with the command

```
1 zscore = scipy.stats.zscore(X)
```

2.5 Exercises

2.5.1 Copying arrays

In this first exercise we propose manipulate some arrays to get familiar with Numpy and Matplotlib. To this end create a 1D array called T containing N=250 elements ranging between 0 and 2π. Then compute

```
1 X = np.sin(T)
2 Y = X
3 Y *= 2
```

Plot both vector using Matplotlib. Do this look okay to you ? What should you fix ?

2.5.2 Temperature Evolution

You can download from the black board the file called 'temperature.csv'. This file contains the global temperature anomaly recorded by the NASA each month between 1881 and 2015 as well as yearly and seasonal average. This data is available at
http://data.giss.nasa.gov/gistemp/. As defined in the same page a temperature anomaly means a departure from a reference value or long-term average. A positive anomaly indicates that the observed temperature was warmer than the reference value, while a negative anomaly indicates that the observed temperature was cooler than the reference value.

This file is a .csv file that can be easily treated with Python for processing. Download the file and open it with a text editor. As seen on the header of the file the first column corresponds to the year, the 12 next columns give the temperature each month of that year. The next column gives the yearly average and then the december-november average. We have then on the last four columns the four seasonal averages. Each temperature is given here in 0.01 C.

In this exercise we propose to visualize and analyze the temperature variations using Numpy and Matplotlib. To do that you can load the data contained in the file using one of the methods see in section 2.4.1. You will obtain a matrix where the first column contains the years and the rest of the matrix the temperature. To visualize the temperature evolution you will first:

- Plot the histogram of the yearly average and winter average temperature
- Plot the evolution through the year of the temperature in January and August

You should obtain something similar to the figure 2.1.12. You can see that the histograms overlap a lot making the visualization complicated. The time series (we refer as time series to any data that evolve in time) are very noisy showing a increasing trend with cyclic variations.

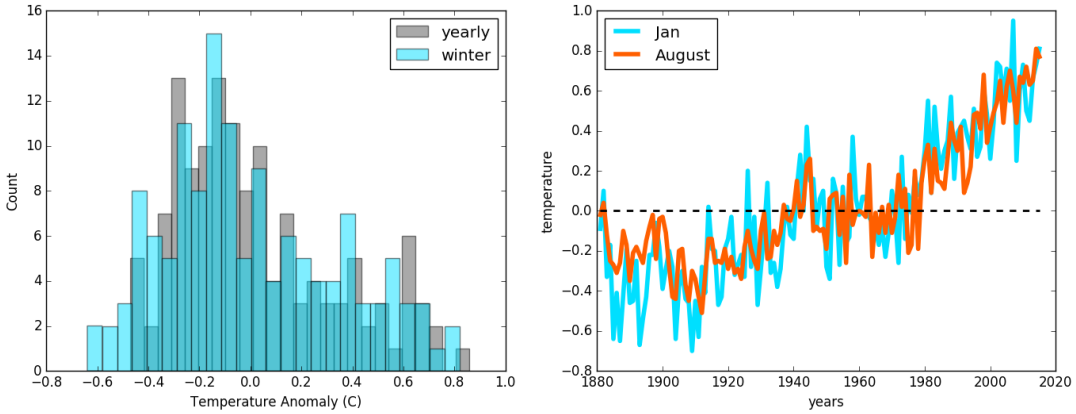


Figure 2.12: Left - Histogram of the yearly average and winter average temperature Right - time evolution of the temperature in January and August

To smooth out the cyclic variation of the temperature it is interesting to compute the rolling average of the temperature over a certain number of month. As it names indicate, the rolling average over $2N+1$ point is defined by :

$$X[k] = \frac{1}{2N+1} \sum_{n=-N}^{N} X[k-n] \tag{2.17}$$

i.e. the rolling average of X at point k is given by the mean of X over the points $[k-N, ...k, ...k+N]$. We propose here to compute the rolling average of the yearly temperature, i.e the 14th columns of the data file, between the years 1900 and 2000. We propose to compute the average temperature over a time interval of 3,5,....19 years. Hence if the interval is 3 years the value of N in equation 2.17 should be 1. If the interval is 7 years N=3 and so on ...

(1) - Compute the moving average of the yearly mean temperature from years 1900 to 2000 using an interval of 3,5,... 19 years. You should obtain a picture similar to the Fig 2.13.

HINT - Locating a given value in an array : if you have an array called YEAR that goes from 1881 1882, ... 2015 and you want to find out the index k where YEAR[k] = 1900 you can use the command `k = np.argwhere(YEAR==1900)`.

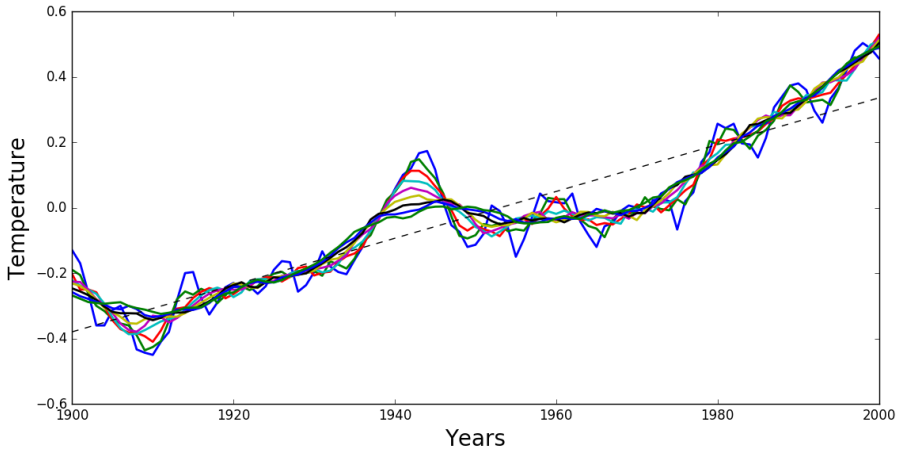


Figure 2.13: Moving average over periods of 3,5,7,...19 years of the yearly mean temperature. These plots are way less spiky than the one shown above. The linear fit of the temperature obtained in *(2)* is also shown.

(2) - Fit the moving average of the yearly temperature computed over an interval of 19 years using linear regression. To do so you can use any method that has been presented in section 2.3. The linear fit we've obtained is shown in Fig. 2.13.

(3) - To obtain a general picture of the temperature evolution we would like to plot the box-plot of all the monthly temperatures recorder during each decade. To do so, store all the monthly temperatures recorded between 1890 and 1900 in a given variable (It might be for example be a line of a matrix). Then do the same for the all the monthly temperatures recorded between 1900 and 1910 and so on. You can then use the command boxplot() of matplotlib to create the figure shown in Fig. 2.14.

HINT - Flattening an array : You can flatten a 2D array to a 1D array with the command `flatten()`. . For example the piece of code

```
1 X = np.arange(9).reshape(3,3)
2 Y = X.flatten()
```

is equivalent to

$$X = \begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix} \longrightarrow Y = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{pmatrix} \tag{2.18}$$

The command `flatten()` literally flattens a mutli dimensional array to a 1D array containing the same values.You can use this command to obtain a 1D vector containing all the monthly temperatures recorded in a given decade.

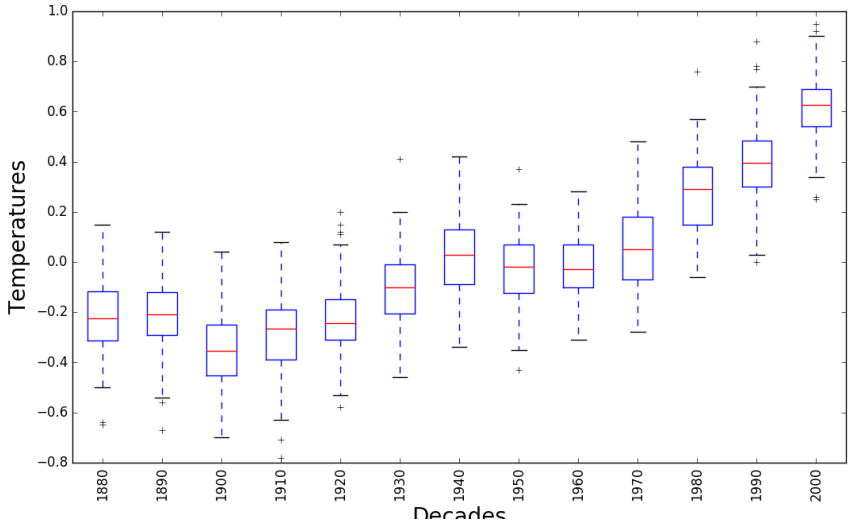


Figure 2.14: Boxplots of the temperature distribution over each decade

2.5.3 Experimental Data Processing

You've been hired by a new start-up to characterize the reliability of their manufacturing process. The start up builds little devices that when illuminated with a ultra-short laser pulse shines from a brief period of time. To characterize their devices they record the light coming out of the device and obtain for each device a curve showing the light intensity as a function of time. Few of these curves are shown in Fig. 2.15. These curves can be fitted with an exponential function

$$F(t) = \exp(-kt) \tag{2.19}$$

where k is the decay parameter. Of course each device will lead to a different value of k. The fit of few of curves are shown in the Fig. 2.15.

To facilitate your analysis the company has built 250 devices on three different days. They've measured the light coming out of these devices and gave you 3 space separated files 'exp_data_1.dat', 'exp_data_2.dat' and 'exp_data_3.dat' that you can download on the Blackboard. Each file contains 251 columns where the first column is the time (i.e. the x-axis) and the 250 others are the light intensity coming out of the 250 devices built and tested that day. The company wants you to estimate the distribution of decay parameters obtained for all these devices.

To analyze the data you will do for each file :

- Load the data in your program
- Fit the 250 signals it contains with an exponential decay to evaluate the corresponding value of k. You will end up with a distribution of k-values that characterize all the devices tested that day.
- Plot the histogram of the k-value. You should obtain a figure similar to Fig. 2.16 (we show here the distributions obtained for the three files.)
- Estimate the mean value and standard deviation of each of the distribution.

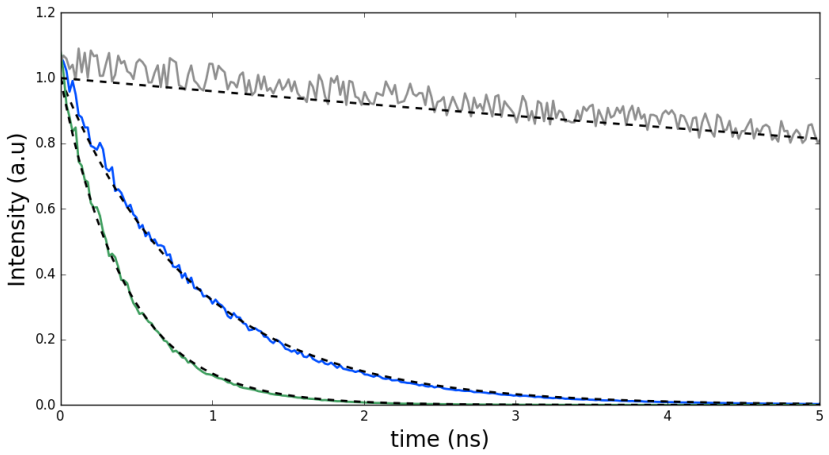


Figure 2.15: Light intensity measured for 3 different devices. The dash line show the result of the exponential fit performed to evaluate the values of k for each device. You can already see that very different k-values are expected for the different devices.

The target of the company was to obtain a mean value of k ranging between $1.5 < \mu < 2.5$ with a standard deviation of $\sigma < 2.0$. Is this goal achieved for the different days ? The company also wants to know if the devices created on different days are similar or not. Use a t-test on the three distributions of k-values to answer that question.

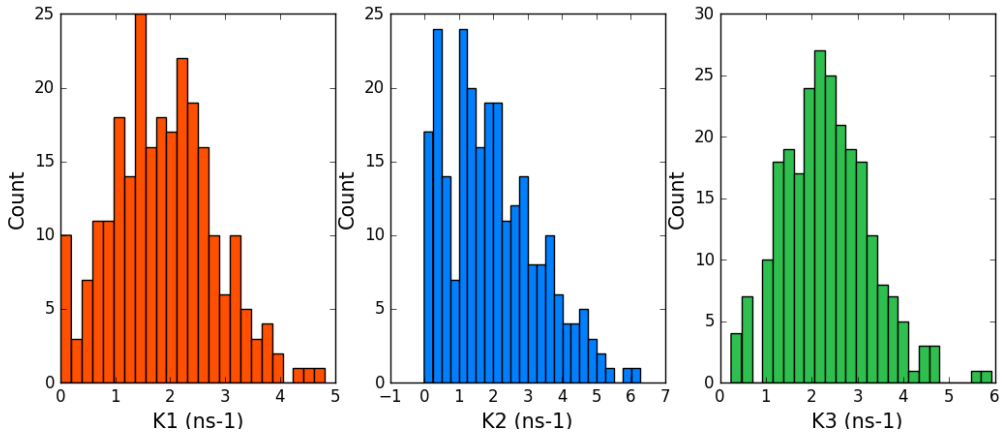


Figure 2.16: Histograms of the k-values obatined by fitting the data contained in the three files.

2.5.4 The power of vectorization

In this exercise we propose to quantify how much speed up we can expect from the vectorization of the computation of a 2D matrix. To do so we propose to compute the function

$$f(x, y) = \sin(x) \times \cos(y) \tag{2.20}$$

for $x \in [-\pi; \pi]$ and $y \in [-\pi; \pi]$. We will compute the value of this function for each pair of points x_i, y_j and store this value in a matrix `f[i,j]`. The number of points along the x-axis and y-axis is referred to

$N_x = N_y = N$ and we will vary the value of N from 10 to 2000.

To measure the time it takes to execute a certain block of operations we can use the module `time`. The utilization of this module is illustrated in the following snippet of code:

```
import time
...
# start the chronometer
t0 = time.clock()

# ...
# block of operations
# ...

# stop the chronometer
t1 = time.clock()

# print the timing
print 'It took  %f sec to complete the operation' %(t1-t0)
```

To compute the matrix `f[i,j]`. we can use two different solution We can use nested **for** loops running through both indexes as in:

```

x = np.linspace(-np.pi,np.pi,N)
y = np.linspace(-np.pi,np.pi,N)

t0 = time.clock()
for iY in range(N):
        for iX in range(N):
                F[iY,iX] = .....
t1 = time.clock()
```

Or we can use the methods presented in section 2.2.2 where we first create a grid and compute the value of $f(x,y)$ using vectorization as in

```
t0 = time.clock()
X,Y = np.meshgrid(x,y)
F = .....
t1 = time.clock()
```

Write a program that compute the matrix with these two methods and measure the time required to compute the matrix in both cases. To evaluate the difference between the two methods use the following values of N : 100, 200, 500, 1000, 2000. Using my laptop I obtained the following times

	100	200	500	1000	2000
loop	0.02	0.10	0.69	3.11	11.23
vect.	1E-5	0.002	0.011	0.05	0.19

In addition you can (if you want) plot these times as well as the matrix using Matplotlib. You can plot the matrix directly with the command

```
plt.matshow(matrix, cmap=plt.cm.jet)
```

The colormap (cmap) defines the colors used to print the matrix. Many colormaps are available in Matplotlib, see for example http://matplotlib.org/examples/color/colormaps_reference.html. To plot the time use a semilog axis with the commands

```
1 plt.semilogy(X,Tq,'o-',linewidth=2,color="#FF000F",label='loop')
```

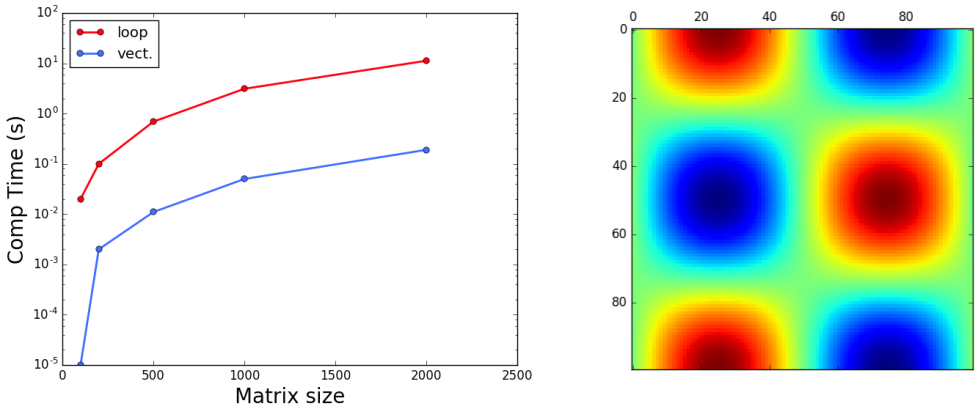


Figure 2.17: Left - Computation time required to evaluate a matrix of increasing size with the loop and vectorization approach. Right - Matrix defined by $f(x, y) = \sin(x) \times \cos(y)$.

Bonus : Can you find an other way of computing the matrix using vectorization ?

Chapter 3
Vectors, Matrices, and Linear Systems

We have seen in the last chapter the basics of matrix manipulation with Numpy. In this chapter we'll a bit more in detail how to generate matrices (2D arrays) and how to perform linear algebra operations using Numpy (matrix-vector product, matrix-matrix product, eigenvalues ...). We will also show ho to generate 2D plot using matplotlib and Mayavi. The handling of matrices is very important in any scientific problems as it allows for very efficient solution to a great number of problems. In particular matrices play a huge role in the solution of linear system of equations. We will present here two different methods to solve linear system using direct or iterative approaches based on the manipulation of matrices.

3.1 Vectors and Matrices

The correct handling of vectors and matrices is critical in engineering problems. Such handling is used to solve linear equation, differential equation, electronic structure calculations, etc .... As we have (briefly) already seen during the last chapter, manipulating vectors and matrices can be done in python with the use of the module Numpy. We present here the creation and manipulation of these objects and introduced some capabilities of the Numpy module.

3.1.1 Simple vectors and matrices

A vector is a one-dimensional object that is usually represented as :

$$V = \begin{pmatrix} v_0 & v_1 & \dots & v_N \end{pmatrix} \tag{3.1}$$

where v_0 is the first element of the vector, v_1 the second .... As we have already seen such array can be constructed with the command

```
1 v=np.array([12.24,23.10,7.89,1.03,0.01])
```

where we have set $v_0 = 12.24$, $v_1 = 23.10$ and so on. Matrices are tow-dimensional arrays that are represented as:

$$A = \begin{pmatrix} a_{0,0} & a_{0,1} & \dots & a_{0,N} \\ a_{1,0} & a_{1,1} & \dots & a_{1,N} \\ \vdots & \ddots & \dots & \vdots \\ a_{N,0} & a_{N,1} & \dots & a_{N,N} \end{pmatrix} \tag{3.2}$$

Where A is the matrix and $a_{i,j}$ its element on the i-th line and j-th column. Matrices can be directly defined with the numpy array module. For example the code

```
1 M = np.array([[1,2,3],[4,5,6],[7,8,9]])
```

creates the matrix:

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \tag{3.3}$$

Note that a matrix is here defined as an array whose elements are themselves arrays. Hence notice the extra set of brackets that surround the three arrays.

3.1.2 Slicing vectors

It is often useful to extract part of a large matrix. If the indexes of the elements to be extracted are contiguous, we can use the ":" operator to extract part of matrices. The sign ":" can be used to generate a series of indexes of an array. Hence in the code

```
1 u = np.arange(10)       #u = [0 1 2 3 4 5 6 7 8 9]
2 v =  u[1:3]             #v = [1 2]
```

v contains the values of u ranging from the first to the second index (remember the index starts at 0). In general the statement

```
1 v =  u[a:b]
```

extracts the elements of u ranging from a to $b-1$ and place them in v. If you omit the a or b, the default values $a = 0$ and $b = N$ where N is the total length of the vectors will be used. We have therefore

```
1 u = np.arange(10)       #u = [0 1 2 3 4 5 6 7 8 9]
2 v = u[:4]               #v = [0 1 2 3]
3 v = u[6:]               #v = [6 7 8 9]
4 v = u[:]                #v = [0 1 2 3 4 5 6 7 8 9]
5 v = u[1:-2]             #v = [1 2 3 4 5 6 7]
```

Note on line 5 the use of a negative index. Negative indexes are taken from the *end* of the vector. Hence -1 corresponds to the last element of the array, -2 the second last and so on.

3.1.3 Slicing Matrices

A similar approach can be used to extract elements from a matrix by specifying on which line and column the element is. As for 1D array the indexing of matrices in Python starts at 0. Hence the python statement

```
1 mat[1,1]
```

refers to the element of the 2nd line and 2nd column. To extract given lines and column of a matrix one can use:

```
1 mat = np.arange(25).reshape(5,5)
2 col = mat[:,2]
3 line = mat[3,:]
```

prints the element on the 2nd line and 2nd column. Similarly the statement:

```
1 sub = mat[1:4,1:4]
```

extracts a submatrix containing only the lines and column ranging between the first and the 3rd. If the indexes of the elements to be extracted are not contiguous, one can use the numpy function *ix*. This function construct an open mesh from sequences given in argument. This mesh can be used as a mask to extract a submatrix.

```
mat = np.arange(25).reshape(5,5)
ind1 = [0,2,4]
ind2 = [0,2,4]
sub = mat[np.ix_(ind1,ind2)]
```

This code creates the following matrices

$$\text{mat} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 & 9 \\ 10 & 11 & 12 & 13 & 14 \\ 15 & 16 & 17 & 18 & 19 \\ 20 & 21 & 22 & 23 & 24 \end{pmatrix} \longrightarrow \text{sub} = \begin{pmatrix} 0 & 2 & 4 \\ 10 & 12 & 14 \\ 20 & 22 & 24 \end{pmatrix} \tag{3.4}$$

The submatrix is generated by taking only the elements on the 0th, 2nd and 4th line and 0th, 2nd and 4th column. If the same indexes have been chosen here for the columns and lines ind1 can be different from ind2 as long as they do not contain index that are larger than the size of the matrix.

It is finally possible to change the values of the elements contains in a line, a column or a part of a matrix.

```
mat = np.arange(25).reshape(5,5).astype(float)

mat[:,2] = np.random.rand(5)
mat[1,:] = np.random.rand(5)

ind2 = [0,2,4]
mat[np.ix_(ind1,ind2)] = np.random.rand(3,3)
```

Note : the use of the function astype(float) which forces the python to interpret the number in the matrix as floating points and not integer which is the default for np.arrange. Without it, all the random numbers contained between 0 and 1 subsequently introduced in the matrix will be interpreted as integer and hence would all be 0.

3.1.4 Creating complicated Matrices with Numpy

We can also use the module numpy to generate matrices or arbitrary size.

```
import numpy as np
N = 10                          # Number of columns
M = 15                          # Number of rows
Z = np.zeros((N,M))             # a null matrix note the double bracket
O = np.ones((N,M))              # a matrix with ones everywhere
E = np.eye(N)                   # a square diagonal matrix  (identity matrix)
```

For example the command

```
X = np.arange(N*M).reshape(N,M)
```

creates a 1D array ranging from 0 to $(N * M) - 1$ and reshape this 1D array in a matrix of N rows and M columns. Numpy can be used to create random matrices. This can be done with the command

```
R = np.random.rand(N,M)
```

which creates a $N \times M$ matrix whose elements are random numbers uniformly distributed between 0 and 1. The generation of random numbers will play a great role in chapter 5 and 6. Nested loops can also be used to generate matrices. hence if we want to create a $N \times M$ matrix given by $M_{i,j} = i * j$ we can use the following code:

```
mat = np.zeros((N,M))
for iL in range(N):
        for iC in range(M):
                mat[iL,iC] = iL*iC
```

In this code the first loop, called outer loop, goes through all the rows while the second one (the inner loop) goes through the columns. When possible a faster way to generate matrices is to use numpy routines. For example the matrix creates above can be obtained via the outer product of two vectors. Suppose we have two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ which are $M \times 1$ and $N \times 1$ column vectors. Then the outer product of these two vectors is given by:

$$\boldsymbol{u} \otimes \boldsymbol{v} = \boldsymbol{u}\boldsymbol{v}^T = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} \begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix} = \begin{pmatrix} u_1 v_1 & u_1 v_2 & u_1 v_3 \\ u_2 v_1 & u_2 v_2 & u_2 v_3 \\ u_3 v_1 & u_3 v_2 & u_3 v_3 \end{pmatrix} \tag{3.5}$$

here $M = N = 3$. In compact notation $(\boldsymbol{u} \otimes \boldsymbol{v})_{ij} = u_i v_j$. The final matrix can be directly obtained via the numpy *outer* function:

```
u = np.random.rand(10)
v = np.random.rand(10)
mat = np.outer(u,v)
```

Note : The inner product of two vectors, $\langle \boldsymbol{u}, \boldsymbol{v} \rangle = \boldsymbol{u}^T \boldsymbol{v} = \sum_i u_i v_i$, can also be calculated via the *inner* function of numy

```
u = np.random.rand(10)
v = np.random.rand(10)
IP = np.inner(u,v)
```

3.1.5 Visualizing a matrix with Matplotlib

To illustrate how to visualize 2D data we will study the sinus cardinal function given by:

$$f(x,y) = \frac{\sin(a\sqrt{x^2+y^2})}{\pi\sqrt{x^2+y^2}} \tag{3.6}$$

This 2D function can be visualized via a small Python script that that generates two 1D vector (X,Y) and compute the sinus cardinal for each points and plot this function using matplotlib. Two options are available to create the matrix: either using nested loops or a vectorized approach.

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm
import numpy as np


# parameter for the grid
min_grid, max_grid = -10,10
```

```
nX, nY = 100, 100

# create 1D vectors
X = np.linspace(min_grid,max_grid,nX)
Y = np.linspace(min_grid,max_grid,nY)

# create grids
X, Y = np.meshgrid(X, Y)

# precompute the distance of each point to the center
R = np.sqrt(X**2 + Y**2)

# compute the matrix to show
a = 1.
F = np.sin(a*R)/(np.pi*R)


# show the matrix in 3D
fig = plt.figure()
ax = fig.gca(projection='3d')
plt.cla()
surf = ax.plot_surface(X, Y, F, rstride=1, cstride=1,
        cmap=cm.jet,linewidth=0, antialiased=True)
plt.show()
```

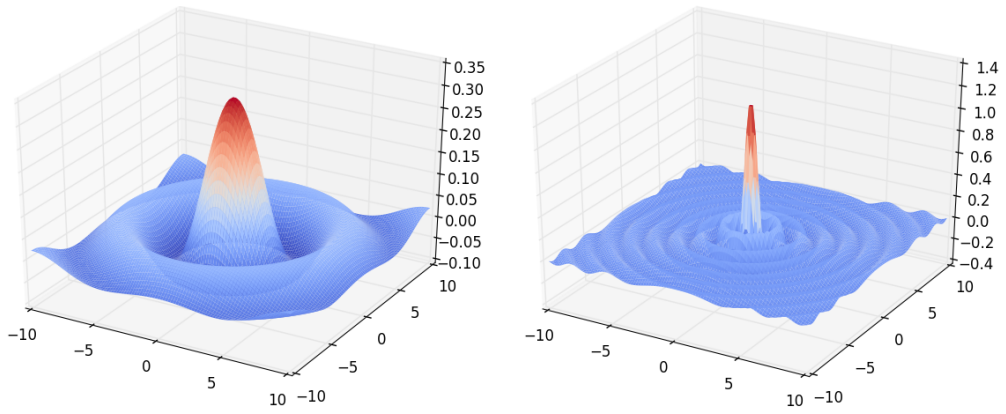


Figure 3.1: Representation of two sinus cardinal functions obtained for 2 different values of the width parameter: $a = 2$ and $a = 4$

3.1.6 Visualizing a matrix with Mayavi

While Matplotlib can plot 3D surfaces the rendering is generally not terrible. Other modules dedicated to the representation of 3D objects have been developed for Python. One of this module is called Mayavi. An example of a simple program using Mayavi to plot a 3D surface is reported below

```
import numpy as np
from mayavi import mlab
```

```
3
4  # parameter for the grid
5  min_grid, max_grid = -10,10
6  nX, nY = 250, 250
7
8  # create 1D vectors
9  x = np.linspace(min_grid,max_grid,nX)
10 y = np.linspace(min_grid,max_grid,nY)
11
12 # create grids
13 X, Y = np.meshgrid(x, y)
14
15 # precompute the distance of each point to the center
16 R = np.sqrt(X**2 + Y**2)
17
18 # compute the matrix to show
19 a = 4.
20 F = np.sin(a*R)/(np.pi*R)
21
22 s = mlab.surf(X.T, Y.T, F)
23 mlab.show()
```

As you can see the syntax is similar to what would be used with matplotlib except that we here use the command `s = mlab.surf(X.T, Y.T, F)` to plot the surface.This code opens a new window where the surface is displayed. Many options a e directly accessible from this window to control the aspect of the plot. An example of such plot is reported on the Fig. 3.2.

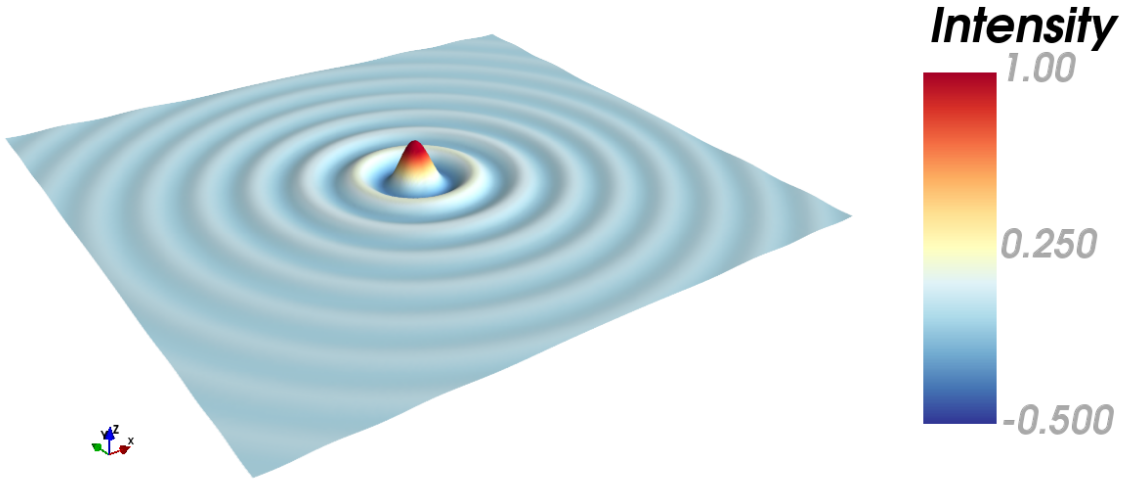


Figure 3.2: Representation of a 3D surface with mayavi.

3.1.7 Linear Algebra with numpy

A large number of routine for linear algebra have been implemented in Numpy as for example the multipliation of two matrices A and B defined by:

$$A = \begin{pmatrix} A_{11} & \dots & A_{1m} \\ \vdots & \ddots & \vdots \\ A_{n1} & \dots & A_{nm} \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} B_{11} & \dots & B_{1p} \\ \vdots & \ddots & \vdots \\ B_{m1} & \dots & B_{mp} \end{pmatrix} \tag{3.7}$$

The product of these two matrices is given by :

$$C = A \times B = \begin{pmatrix} C_{11} & \dots & C_{1m} \\ \vdots & \ddots & \vdots \\ C_{n1} & \dots & C_{nm} \end{pmatrix} \quad \text{with} \quad C_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj} \tag{3.8}$$

This complex operation can be performed with the function *matmul* or *dot* of numpy:

```
m = 10
n = 15
p = 5
A = np.random.rand(n,m)
B = np.random.rand(m,p)
C = np.dot(A,B)
```

Numpy also contains a lot of different method to manipulate matrices. The calculations of the eigenvalues, determinant, inverse .... of a matrix can therefore be performed with only one line of code such as :

```
import numpy.linalg as npla
det = npla.det(A)        # determinant of the matrix A
w,v = npla.eig(A)        # eigenvalues (w) and eigenvectors(v) of the matrix A
Am1 = npla.inv(A)        # inverse of the matrix A
```

3.1.8 Example : Matrix-Vector multiplication

To illustrate how to manipulate matrices we are going to see the implementation of a matrix-vector multiplication: $\mathbf{u} = A \cdot \mathbf{v}$. This operation reads:

$$\begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_N \end{pmatrix} = \begin{pmatrix} a_{0,0} & a_{0,1} & \dots & a_{0,N} \\ a_{1,0} & a_{1,1} & \dots & a_{1,N} \\ \vdots & \ddots & \dots & \vdots \\ a_{N,0} & a_{N,1} & \dots & a_{N,N} \end{pmatrix} \cdot \begin{pmatrix} v_0 \\ v_1 \\ \vdots \\ v_N \end{pmatrix} \tag{3.9}$$

where

$$u_i = \sum_{j=0}^{N} a_{i,j} \cdot v_j \tag{3.10}$$

To compute the vector $\mathbf{u}$ we therefore need to use a loop where for each iteration we compute the inner product of a give line of A with the the vector $\mathbf{v}$. In Python notation the i-th line of the matrix can be accessed with the command `A[i,:]`and the inner product of this line with the vector $\mathbf{v}$ can be calculated with

```
np.inner(A[i,:],v)
```

Therefore the main part of the calculations of this matrix-vector product is the loop

```
for i in range(nLine):
        u[i] = np.inner(A[i,:],v)
```

where we store in the i-th element of the vector u, the inner product of the i-th line of A with v. This for loop can be included in a Python program as shown below

```
import numpy as np

def mat_vect(A,v):

        nLine = A.shape[0]
        nCol = A.shape[1]
        sizeV = v.shape[0]


        if nCol != sizeV:
                print 'Error : Size inconsistent'
                return

        u = np.zeros(nLine)
        for iL in range(nLine):
                u[iL] = np.inner(A[iL,:],v)

        return u

A = np.random.rand(4,5)
v = np.random.rand(5)

u = mat_vect(A,v)
ucheck = np.dot(A,v)

print np.sum(u-ucheck)
```

This program contains a function that performs the multiplication of a matrix with a vector and return the resulting vector. Note that the function first check if the size of the matrix and vector are compatible or not using the command 'shape'.

3.2 System of linear equations

Many engineering problems involve multiple unknown $(x_1, x_2, ...x_N)$ that are related by multiple linear equation. If the system is truly linear, the unknown are not multiplied by each other $(x_i x_j)$ or themselves (x_k^2) and there is no non-linear term such as $\sin(x_i)$. In such case we can write the system of equation as

$$a_1 x_1 + a_2 x_2 + \dots a_n x_n = f_1 \tag{3.11}$$
$$b_1 x_1 + b_2 x_2 + \dots n_n x_n = f_2 \tag{3.12}$$
$$\vdots \tag{3.13}$$
$$z_1 x_1 + z_2 x_2 + \dots z_n x_n = f_n \tag{3.14}$$

In this system of equation, a_i, b_i ... and f_i are numbers and the system as n-equations and n-unknowns. We can rewrite this system in form of a matrix

$$\begin{pmatrix} a_1 & a_2 & \dots & a_n \\ b_1 & b_2 & \dots & b_n \\ \vdots & & \ddots & \vdots \\ z_1 & z_2 & \dots & z_n \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{pmatrix} \tag{3.15}$$

The first matrix contains all the constants, the middle one the unknown and the left-hand side the solution of each equation. This problem is often write $\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{f}$. Given a set of constants a_i, b_i ... and f_i, the goal is to find the values of all the different unknowns.

3.2.1 Example of linear systems: Distillation Column

Distillation are used to separate a mix of compounds based on different their different boiling temperature. The mathematical modeling of such process involve 1000's of non linear equations. Let us explore a simplification of this model that is illustrated in figure 3.2. In this example the input mixture contains three compounds: 30 kg/s of methane (M), 25 kg/s of ethane(E) and 10 kg/s of propane(P). The input mixture is separated in three out flow streams: the overhead that is rich in methane (90 %), the middle stream rich in ethane (50 %) and the bottom flow rich in propane (70 %). The figure contains additional information about the mass fraction of each flow, noted x_i and the total mass flow rates of each stream given in kg/s and noted m_i.

Ideal distillation columns are operated in steady state and every kg of a given compound that enters the column must be matched by the same amount of the same compound leaving the column. Using the principle of Conservation of mass $m_{in} = m_{out}$ we can equate the total mass entering the column with the total mass leaving it:

$$30 = 0.9m_1 + 0.3m_2 + 0.1m_3 \tag{3.16}$$

We can derive similar linear equations for ethane and propane and write the resulting systems in a matrix form of the type $\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{f}$. This leads to the matrix equation:

$$\begin{pmatrix} 0.9 & 0.3 & 0.1 \\ 0.1 & 0.5 & 0.2 \\ 0.0 & 0.2 & 0.7 \end{pmatrix} \cdot \begin{pmatrix} m_1 \\ m_2 \\ m_3 \end{pmatrix} = \begin{pmatrix} 30.0 \\ 25.0 \\ 10.0 \end{pmatrix} \tag{3.17}$$

3.2.2 Solving linear equation with numpy

The system of equation 3.17 can be solved directly with numpy. For that we need to use a library that contains common linear algebra routines. The most versatile one is numpy.linalg and need to be imported in the python script. To ease the syntax this library can be given a shorter name as for example 'nl'. We can then use function nl.solve() to solve linear systems.

```
1 import numpy as np
2 import numpy.linalg as nl
3
4 # create the matrix A
5 A = np. array ([[0.9 ,0.3 ,0.1] ,
6                 [0.1 ,0.5 ,0.2] ,
7                 [0.0 ,0.2 ,0.7]])
```

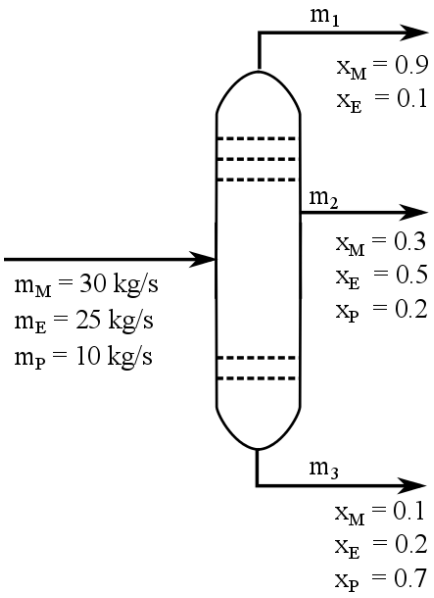


Figure 3.3: Simplified model of a distillation column. A flow composed of methane, ethane and propane enters the column. The three compounds are then separated in 3 streams each mainly composed of one of the compounds.

```
8
9  # create the right-hand side
10 f = np. array ([30.0 ,25.0 ,10.0])
11
12 # solve the  linear system with np.solve()
13 x = nl.solve(A,f)
14
15 # print the solution
16 print (x)
17
18 # check the solution
19 check_sol = np.dot(A,x)
20 print check_sol
21 print f
```

The output of the script is

$$m_1 = 17.88\ kg/s \qquad m_2 = 45.96\ kg/s \qquad m_3 = 1.15\ kg/s \tag{3.18}$$

We have therefore solved the linear equation and determined the mass flow rates of each stream. The validity of the solution is here verified by directly computing the product $\boldsymbol{A} \cdot \boldsymbol{x}$ and comparing this solution with the vector f. The method implemented by np.solve() are said to be 'exact' meaning as exact as a computer can be. In other words the first 8-12 digits of the results will be exact. Methods that can compute exact solutions are called *direct*. These methods are suitable for relatively small linear systems (up to $\sim$ 1000 equations). For larger systems we will prefer *iterative* solutions that are not exact but can tackle much larger systems.

3.2.3 A direct solution : the Gauss-Jordan algorithm

The Gauss-Jordan algorithm, also referred to as Gaussian elimination, relies on the use of an augmented matrix, made from the matrix A to which the column vector f is added on the right side. The methods used transformation of this augmented matrix following:

$$\left(\begin{array}{cccc|c} a_1 & a_2 & \dots & a_n & f_1 \\ b_1 & b_2 & \dots & b_n & f_2 \\ \vdots & & \ddots & \vdots & \vdots \\ z_1 & z_2 & \dots & z_n & f_n \end{array}\right) \longrightarrow \left(\begin{array}{cccc|c} a'_1 & a'_2 & \dots & a'_n & f'_1 \\ 0 & b'_2 & \dots & b'_n & f'_2 \\ \vdots & & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & z'_n & f'_n \end{array}\right) \longrightarrow \left(\begin{array}{cccc|c} 1 & 0 & \dots & 0 & x_1 \\ 0 & 1 & \dots & 0 & x_2 \\ \vdots & & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & x_n \end{array}\right) \tag{3.19}$$

As seen in the above equation the algorithm takes place in two steps. First we transform A into an upper matrix, i.e. a matrix containing non-null elements only on and above its diagonal. Then we back-substitute the solution of the equation to obtain the solution of the linear system. To perform these matrix transformations, only 3 operations, that do not change the solutions of the linear system are allowed. These operations are

- interchange two lines
- multiply/divide a line by a non-null scalar
- add/substract to one line a scalar multiple of another line

Let's try to apply this method to the linear system 3.17. The augmented matrix of this system reads:

$$\left(\begin{array}{ccc|c} 0.9 & 0.3 & 0.1 & 30 \\ 0.1 & 0.5 & 0.2 & 25 \\ 0.0 & 0.2 & 0.7 & 10 \end{array}\right) \tag{3.20}$$

We will refer to the first second and third row of this matrix by L_1, L_2 and L_3 respectively. Our first goal it to eliminate the term $b_1 = 0.1$ in L_2. To do so, we can perform the following operation: $L_2 = L_2 - \frac{0.1}{0.9}L_1$. The result of this operation is shown in the second matrix of eq. 3.21 where $b_1 = 0$. We can then eliminate the term $c_2 = 0.2$ from this new matrix by performing $L_3 = L_3 - \frac{0.2}{0.46}L_2$, leading to the last matrix in eq. 3.21.

$$\left(\begin{array}{ccc|c} 0.9 & 0.3 & 0.1 & 30 \\ 0.1 & 0.5 & 0.2 & 25 \\ 0.0 & 0.2 & 0.7 & 10 \end{array}\right) \rightarrow \left(\begin{array}{cccc|c} 0.90 & 0.30 & 0.10 & 30.00 \\ 0.00 & 0.46 & 0.18 & 21.66 \\ 0.00 & 0.20 & 0.70 & 10.00 \end{array}\right) \rightarrow \left(\begin{array}{cccc|c} 0.90 & 0.30 & 0.10 & 30.00 \\ 0.00 & 0.46 & 0.18 & 21.66 \\ 0.00 & 0.00 & 0.61 & 0.71 \end{array}\right) \tag{3.21}$$

Note: all floating numbers have here been truncated to their second digit to save space. As mentioned above we have transform the matrix A in an upper triangular matrix. We can now use back-substitution to solve the system by starting from the bottom row. We therefore perform $L3 = L3/0.61$ leads directly to $m_3 = 1.15$ as seen in the first matrix in eq. 3.22. We can use that new matrix to solve for m_2 by performing $L_2 = 1/0.46(L_2 - 0.18 * L_3)$. Finally we solve for m_1 by doing: $L_1 = 1/0.90(L_1 - 0.30L_2 - 0.10L_3)$.

$$\left(\begin{array}{cccc|c} 0.90 & 0.30 & 0.10 & 30.00 \\ 0.00 & 0.46 & 0.18 & 21.66 \\ 0.00 & 0.00 & 1.00 & 1.15 \end{array}\right) \rightarrow \left(\begin{array}{cccc|c} 0.90 & 0.30 & 0.10 & 30.00 \\ 0.00 & 1.00 & 0.0 & 45.96 \\ 0.00 & 0.00 & 1.00 & 1.15 \end{array}\right) \rightarrow \left(\begin{array}{cccc|c} 1.00 & 0.00 & 0.00 & 17.88 \\ 0.00 & 1.00 & 0.0 & 45.96 \\ 0.00 & 0.00 & 1.00 & 1.15 \end{array}\right) \tag{3.22}$$

As expected the last column contains the solution of this linear system. Although the method used by numpy.solve() are a bit more complex that the Gauss-Jordan algorithm shown here it relies on the same general principle and leads of course to the same solution.

3.2.4 Implementing the Gauss-Jordan Algorithm

As explained above, the Gauss-Jordan algorithm consists in two main operation: 1 - the transformation of the matrix to an upper diagonal form 2 - the backward substitution. To implement this algorithm we will therefore write two functions that each perform one of these two operations.

A simple implementation of the Gauss-Jordan implementation is reported in the code below

```
import numpy as np

#########################################################
# Function to decompose the matrix
# in an upper triangular form
#########################################################

def LU(A,f):

        # get the size of the system
        n = len(f)

        # check the size
        if (A.shape[0] != n) or (A.shape[1] != n):
                print ' \t Inconsistent size in LU decomposition'
                info = 0
                return M,info

        # create the augmented matrix
        M = np.zeros((n,n+1))
        M[:,:-1] = A
        M[:,-1] = f

        # loop through all the colum
        # to get rid of the lower part
        for iC in range(n-1):

                # for each column loop over all the lines
                # that are below the diagonal
                # to set to 0 their elements
                for iL in range(iC+1,n):

                        # check if the diagonal element
                        # is null
                        if M[iC,iC] == 0:

                                print ' \t Zero on the diagonal, LU failed'
                                info = 0
                                return M,info

                        # eliminate the element
                        M[iL,:] = M[iL,:] - M[iL,iC]/M[iC,iC] * M[iC,:]

```

```
        # if we succed we return info = 1 and the upper augmetned matrix
        info = 1
        return M,info
```

This file contains two functions `LU()` and `BS()` that respectively compute the upper diagonal form of the matrix and perform the backward substitution. The function LU receive a matrix A and the right hand side f as arguments. The first few lines of code check the dimensions of the matrix. Then we loop over all the columns (with exception of the last one) of the matrix to eliminate all the terms below the diagonal using the method exposed above. Note that we also check that the diagonal element M[iC,iC] is not null as it would lead to an error when we try to divide by M[iC,iC]. A more advance version of the Gauss-Jordan algorithm is required to treat such cases. Once all the terms are eliminated we return the modified augmented matrix M as well as an integer called info, that equals 1 if the elimination was successful and 0 otherwise.

The implementation of the backward substitution is illustrated in the code below.

```

#########################################################
# Function to backsubstitute the results
# and get the final solution
#########################################################
def BS(M):

        # get the size of the matrix
        n = M.shape[0]

        # loop over all the lines
        # starting by the end
        for iL in range(n-1,-1,-1):

                # check if we have diagonal elements on the diagonal
                if M[iL,iL] == 0:
                        print ' \t Zero on the diagonal, LU failed'
                        info = 0
                        return M,info


                # divide the line by the diagonal element of M
                M[iL,:] /= M[iL,iL]

                # loop over all the lines that are above this one
                for iLL in range(iL-1,-1,-1):
                        M[iLL,:] -= M[iLL,iL]*M[iL,:]

        info = 1
        return M,info


#########################################################
#########################################################
```

This function takes the upper diagonal augmented matrix M, created by the LU function as an input argument. We start here from the last row of the matrix to eliminate the upper diagonal part until a form similar to last one in eq. 3.19 is obtained. The function returns the diagonal form of the matrix M where the last column hold the solution of the system.

To facilitate the use of these two functions we can write our own Python module. This is very easy as it is simply done by writing the two functions presented above in the same file that we will call 'gauss_jordan.py'. This file can now be used as a module in any other file located in the same folder. To use this module we just have to import it (exactly as Numpy etc ...). Thus if we write a file called 'main.py' we can use the function written in 'gauss_jordan.py' as illustrated below:

```
1  import numpy as np
2  from gauss_jordan import *
3
4
5  # create the matrix A
6  A = np. array (
7  [[0.9, 0.3 ,0.1] ,
8   [0.1 ,0.5 ,0.2] ,
9   [0.0 ,0.2 ,0.7]])
10
11 # create the right-hand side
12 f = np. array ([30.0 ,25.0 ,10.0])
13
14 # decompose the matrix
15 M,info = LU(A,f)
16
17 # backsubstitute
18 M,info = BS(M)
19
20 print M[:,-1]
```

3.2.5 Computational cost of the Gauss-Jordan algorithm

The Gaussian elimination and backward substitution are relatively expensive computationally. During the Gaussial elimination we must loop over over the the n column and n rows of the matrix to eliminate all the terms. For each entry we then must compute n multiplications leading to an overall cost of n^3 operations. To estimate the time required to solve linear systems of increasing size, with our method and with the numpy.solve() method we can use the snippet of code below

```
1  import numpy as np
2  import numpy.linalg as npla
3  import gauss_jordan as gj
4  import time
5  import matplotlib.pyplot as plt
6
7  # size of the matrix to be calculated
8  SIZE = [10,50,100,500,1000]
9
10 # create list
```

```
cpu_time_numpy = []
cpu_time_mycode = []

# loop over the size
for size in SIZE :

        # create the system
        A = np.random.rand(size,size)
        f = np.random.rand(size)

        # nummpy
        t0 = time.clock()
        npla.solve(A,f)
        cpu_time_numpy.append(time.clock()-t0)

        # mycode
        t0 = time.clock()
        M,info = gj.LU(A,f)
        M,info = gj.BS(M)
        cpu_time_mycode.append(time.clock()-t0)

# plot the results
plt.semilogy(SIZE,cpu_time_numpy,'o-',linewidth=3, color='black',label='Numpy')
plt.semilogy(SIZE,cpu_time_mycode,'o-',linewidth=4, color='#005FFF',label='My code')
plt.xlabel('Size of the system')
plt.ylabel('Computation Time')
plt.xlim([0,1250])
plt.legend(loc=2)
plt.show()
```

The results of this code are shown in Fig. 3.3. As you can see finding the solution of relatively big linear system is rather inexpensive. However it its pretty clear from that graph that numpy.solve() is much more efficient than our simple Gauss-Jordan implementation. Numpy uses different algorithm to solve linear systems that are beyond the scope of this course and that are usually presented at the graduate level.

3.2.6 Iterative methods : the Jacobi iteration

The motivation behind the iterative methods is the large computational cost and the numerical instability that is sometimes encountered during direct methods such as the Gauss-Jordan method. Iterative methods propose then to use minimization techniques to solve linear systems. The quantity to be minimized is here the so-called residual $\mathbf{res} = \mathbf{f} - A \cdot \mathbf{x}_0$, where $\mathbf{x}_0$ is a vector of solution that is optimized during the algorithm to minimize **res**. The minimization stops when the residual is small enough, i.e. when its norm is smaller than a given threshold.

To illustrate the Jacobi method let's take once again the linear system 3.17. The Jacobi iteration is then made on a series of solution vectors $\boldsymbol{m}^{(k)} = [m_1^{(k)}, m_2^{(k)}, m_3^{(k)}]$. The case $k = 0$ corresponds to our initial guess from which we will compute the second iterations i.e. $\boldsymbol{m}^{k=1}$ and so on. If we change

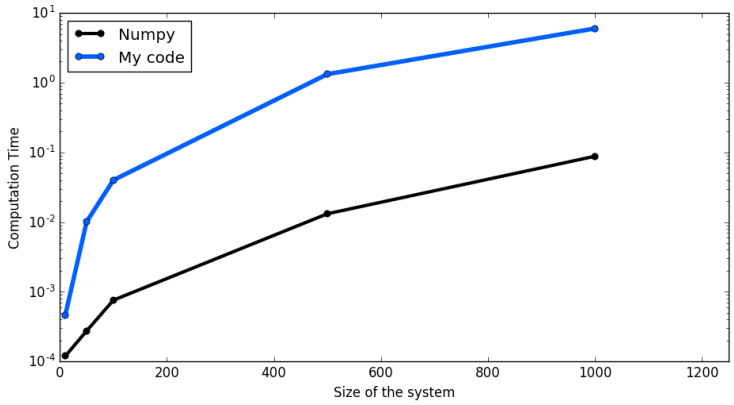


Figure 3.4: Computational time required to solve linear systems of increasing size with the Numpy method and with our own algorithm.

slightly our notation and write the linear system as

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{3n} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} m_1 \\ m_2 \\ m_3 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix} \tag{3.23}$$

the Jacobi method proposes to solve this system iteratively by computing:

$$m_i^{(k+1)} = \frac{1}{a_{ii}} \left(f_i - \sum_{j \neq i} a_{ij} m_j^{(k)} \right) \tag{3.24}$$

Let's apply this method to our problem of column distillation. We'll guess that the values of the mass flow rates are $m_1^{(0)} = m_2^{(0)} = m_3^{(0)} = 20$. As we'll see, even with this bad guess the Jacobi method converges rapidly to a correct solution. The first iteration of the method leads to:

$$m_1^{(1)} = \frac{1}{0.9}(30.0 - 0.3 * 20 - 0.1 * 20) = 24.44 \tag{3.25}$$

$$m_2^{(1)} = \frac{1}{0.5}(25.0 - 0.1 * 20 - 0.2 * 20) = 38.00 \tag{3.26}$$

$$m_3^{(1)} = \frac{1}{0.7}(10.0 - 0.0 * 20 - 0.2 * 20) = 8.57 \tag{3.27}$$

If we compare the the new value of the masses we can see that they are collectively closer from the exact solution that our guess. If we iterate a second time we have

$$m_1^{(2)} = \frac{1}{0.9}(30.0 - 0.3 * 38.00 - 0.1 * 8.57) = 19.71 \tag{3.28}$$

$$m_2^{(2)} = \frac{1}{0.5}(25.0 - 0.1 * 24.44 - 0.2 * 8.57) = 42.62 \tag{3.29}$$

$$m_3^{(2)} = \frac{1}{0.7}(10.0 - 0.0 * 24.44 - 0.2 * 38.00) = 2.10 \tag{3.30}$$

which is even closer from the exact solution. If we keep iterating that way the solution will converge quickly to values that are really close from the exact solution given by the Gauss-Jordan method. In the exercise 3.3.2 we will code ourselves the Jacobi iterations and compare its performance with exact solutions for different linear systems.

3.2.7 Iterative methods : Gauss-Seidel method

An improvement of the Jacobi method is provided by the so-called Gauss-Seidel method. In this approach the $k+1$-th iteration of the solution is obtained via the equation:

$$m_i^{(k+1)} = \frac{1}{a_{ii}} \left(f_i - \sum_{j<i} a_{ij} m_j^{(k+1)} - \sum_{j>i} a_{ij} m_j^{(k)} \right) \tag{3.31}$$

At the difference of the Jacobi method, we always use here the most recent information we have of the solution. For example the calculation of $m_2^{(k+1)}$ in the Jacobi method only depends on $m_{1,3}^{(k)}$. However in the Gauss-Seidel method $m_2^{(k+1)}$ is calculated from $m_1^{(k+1)}$ (that we compute from $m_{2,3}^{(k)}$) and $m_3^{(k)}$. The code below show a possible implementation of the Gauss-Seidel algorithm

```
import numpy as np

############################################################
## Implementation of the Gauss Seidel algorithm
## A matrix of the linear system
## f right hand side
## x0 initial guess of the solution
############################################################

def gauss_seidel(A,f,x0,ITER_MAX = 100, tol = 1E-8,_debug_=1):

        # size of the system
        n = A.shape[0]

        # initialize the residual
        res = np.linalg.norm(f-np.dot(A,x0))

        # init the new vector
        x_new = np.zeros(n)

        # copy the guess
        x = np.array(x0,copy=True)

        # init niter
        niter = 0

        # loop over the
        while (res>tol) and (niter<ITER_MAX):

                # loop over all the lines
                for i in range(n):

                        # initialize the sums
                        sum1, sum2 = 0.0, 0.0

                        # loop over the line elements
                        for j in range(n):
```

```
38
39                              # if j<i we use the new values
40                              if j<i:
41                                      sum1 += A[i,j]*x_new[j]
42                              # else we use the old ones
43                              elif j>i:
44                                      sum2 += A[i,j]*x[j]
45
46                      # we store the new values
47                      x_new[i] = (f[i]-sum1-sum2)/A[i,i]
48
49              # change the old solution to the new one
50              x = x_new
51
52              # compute the new residual
53              res = np.linalg.norm(f-np.dot(A,x))
54
55              # debug a bit
56              if _debug_:
57                      print "Current solution : ",
58                      for iprint in range(n):
59                              print 'm_%d = %1.6f  ' %(iprint+1,x[iprint]),
60                      print "res/tol = %1.2e/%1.2e" %(res,tol),
61                      print ""
62
63              # increment niter
64              niter += 1
65
66      # print the final status of the algorithm
67      if niter == ITER_MAX:
68              info = 0
69              print 'Warning : iteration has not converged'
70              print 'res = %e tol = %e' %(np.linalg.norm(res),tol)
71
72      else:
73              info = 1
74              print 'Iteration has  converged'
75              print 'res = %e tol = %e' %(np.linalg.norm(res),tol)
76
77      return x,info
```

This code takes the matrix A, the right hand side f and the initial guess xo as mandatory arguments. Three other arguments are also present, i.e. the maximum number of iterations, the tolerance for the residual and a debug flag. These three arguments have default values that will be used if these arguments are not specified when calling the function. The main part of the code consists of a while loop that keeps running as long as the residual is larger than the tolerance and the maximum number of iterations is not reached. At each iterations the values of the solutions are updated according to eq. 3.31. The residual is then calculated with the command `np.linalg.norm()` that computes the norm of a vector.

This function is written in a file called 'gauss_seidel.py' that can then be used as a module in any other program. For example the program below import this module and compute the solution of the distillation column presented above.

```
import numpy as np
from gauss_seidel import *


# create the matrix A
A = np. array ([[0.9 ,0.3 ,0.1] ,
                [0.1 ,0.5 ,0.2] ,
                [0.0 ,0.2 ,0.7]])

# create the right-hand side
f = np. array ([30.0 ,25.0 ,10.0])

# our initial guess
x0 = np.array([20,20,20])

# compute the solution
x,info = gauss_seidel(A,f,x0)
```

Different variations exists on this approach such as the Successive over-relaxation method. Other iterative of techniques such as the Conjugated Gradient are routinely used to solve linear systems. These techniques are beyond the scope of this introduction are will be covered at the graduate level.

3.3 Exercises

3.3.1 Matrix Multiplication

To understand how to handle matrices, we first going to write a function that computes the product of two matrices. This function will take two matrices as input and return their product. The equation for the matrix product is given in eq. 3.8. To obtain the number of lines and columns of a matrix use the function:

```
M.shape[0] # number of rows
M.shape[1] # number of columns
```

We have seen during the lecture how to implement the multiplication of a matrix with a vector. The matrix-matrix multiplication follows very similar pattern. Hence the main part of the code will involve two nested loops as in the pseudo-code below.

```
def matmul(A,B):
        .....
        for iL in range(nLine):
                for iC in range(nCol):

                        C[iL,iC] = ......

        return C
```

To test your function generate two random matrices with the command

```
N = 100
A = np.random.rand(N,N)
B = np.random.rand(N,N)
```

that create two square random matrices of size N. You can check that your function works correctly by comparing the results it gives to the results provided by the command `C = np.dot(A,B)`. In addition you can time the performance of your function, i.e. the time required to perform the matrix multiplication using the module time.

```
import time
start = time.clock()
# put the instruction you want here
end = time.clock()
print 'Calculation done in %d sec' %(end-start)
```

Compare the time required to compute the multiplication of matrices of size $N = 10, 50, 100, 500, 1000$ with your function. Compare that time with the time required to perform the same operation with the function $np.dot()$.

3.3.2 Two-stage distillation column

As part of your first project as an engineer you are assigned the task to compute the output mass flow rates m_1, m_2, m_3 of the distillation column shown in Fig. 3.5. This column separates the three compound contained in the input flow in two-stages. Hint : to solve the system write a system of linear equation with 1: the balance of the total mass flow rate, 2: the mass flow rate of methane and 3: the mass flow rate of ethane.

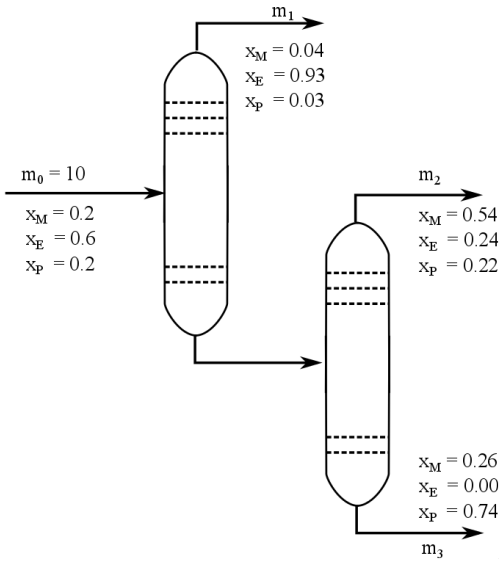


Figure 3.5: Simplified model of a two-stage distillation column. A flow composed of methane, ethane and propane enters the column. The three compounds are then separated in 2 streams in the first column and two the bottom steam enters an additional column that perform another distillation.

3.3.3 Plotting 2D functions

The sinus cardinal function represented in Fig. 3.1 and 3.2 represent the diffraction patter observed when a source of light is forced to pass through a circular aperture. If this aperture is a rectangle of width W and height H the diffraction pattern is given by

$$I(x,y) = \text{sinc}^2(W * x) \times \text{sinc}^2(H * y) \tag{3.32}$$

The sinus cardinal function: $\text{sinc}(x) = \frac{\sin(\pi x)}{\pi x}$ is already implemented in Numpy and is accessible via the command

```
1 F = np.sinc(x)
```

Write a python program that visualize this function in the domain $x, y \in [-2; 2]$ and with $W = 2$ and $H = 1$. You can use either Matplotlib or Mayavi to plot the surface. Using scripts similar to the ones presented in section 3.1.5 and 3.1.6, we ve obtained the plots represented below in Fig. 3.6.

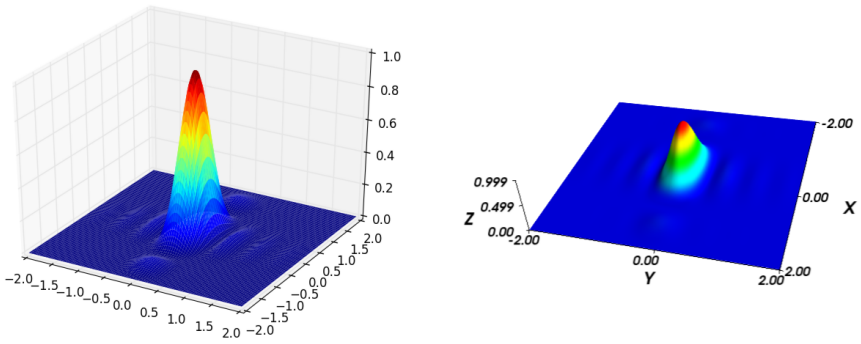


Figure 3.6: Representation of the function $I(x,y) = \text{sinc}^2(W * x) \times \text{sinc}^2(H * y)$ with Matplotlib and Mayavi.

In a real experiment these signals are measured with CCD captors that have a intensity threshold. Hence, at each point the intensity cannot be larger than a certain value. This can be expressed as

$$\text{if} \quad I(x_i, y_j) \geq I_{max} \quad \text{then} \quad I(x_i, y_j) = I_{max} \tag{3.33}$$

This threshold could be implement with an **if** statement within nested **for** loops. However a much more efficient is provided by the command

```
1 I[I>0.05] = 0.05
```

where I is the matrix of the intensities and 0.05 the value of the threshold chosen here. This command directly finds all the elements of I that are larger than 0.05 and replace them by 0.05. Implement this threshold and plot the function with the command

```
1 plt.matshow(I,cmap=cm.gray)
2 plt.show()
```

Using threshold values of 0.1 and 0.001 and $W = H = 4.0$ we've obtained the two plots shown below. A small enough threshold allows to represent the diagonal spots of the diffraction pattern.

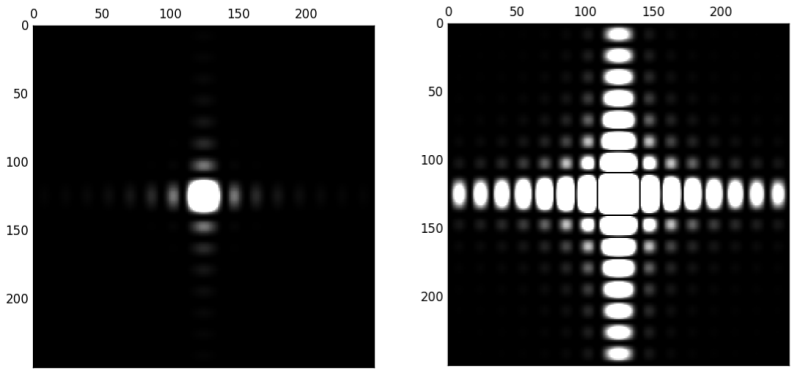


Figure 3.7: Representation of the function $I(x,y) = \text{sinc}^2(W * x) \times \text{sinc}^2(H * y)$ with the command matshow of matplotlib. Two threshold have been chose Left - 0.1 and Right - 0.001

3.3.4 Lake contamination

You've been hired by a environmental firm to estimate the concentration of polychlorinated bipheny (PCB) in the Great Lakes. You ve been provided the diagram shown in Fig. 3.8. The concentration of PCB in each lake is constant and therefore all the PCB that enters must leave the lake. Use this knowledge to write the balance equation of each lake. For example the balance of Lake Superior reads:

$$180\frac{kg}{yr} = Q_{SH}C_S \tag{3.34}$$

Once you have written these equations rearrange them so that all only term without any unknown appears on the right-hand side. You can then solve this system of equation with one of the methods seen in this chapter and report the concentration of PCB contained in each lake to your firm.

Your firm is also considering a bypass that would go directly from lake Michigan to Lake Ontario with a flow rate of 20 km^3/yr in order to reduce the concentration of PCB in lake Michigan. Examine the potential impact of this bypass.

3.3.5 Jacobi Iteration

Your project manager now wants you to implement the Jacobi methods to compare the performance of this methods with the numpy.solve() method. Write a function that takes 3 input arguments: the matrix A, the left-hand vector $\boldsymbol{f}$ and a guess vector for the solution of the linear system $\boldsymbol{x}_0$. The function will return the final value of the solution. In the function you will first determine the residual of the initial guess. You will then use a *while* loop to iterate the solution vector $\boldsymbol{m}^{(k)}$ and update the residual $\mathbf{res} = \mathbf{f} - A \cdot \mathbf{x}^k$. The *while* loop will stop only if the norm of the residual is smaller than a threshold that you will define. Typically this threshold should be lower than 10^{-5} to obtain accurate solutions. The norm of a vector can be obtain with the function

```
1 numpy.linalg.norm(X)
```

To ensure that the while loop stops even if the solution is not converging you can impose a maximum number of iterations $ITERMAX$ and combine several conditions to stop the *while* loop

```
1 while (res>tol) and (niter<ITER_MAX):
```

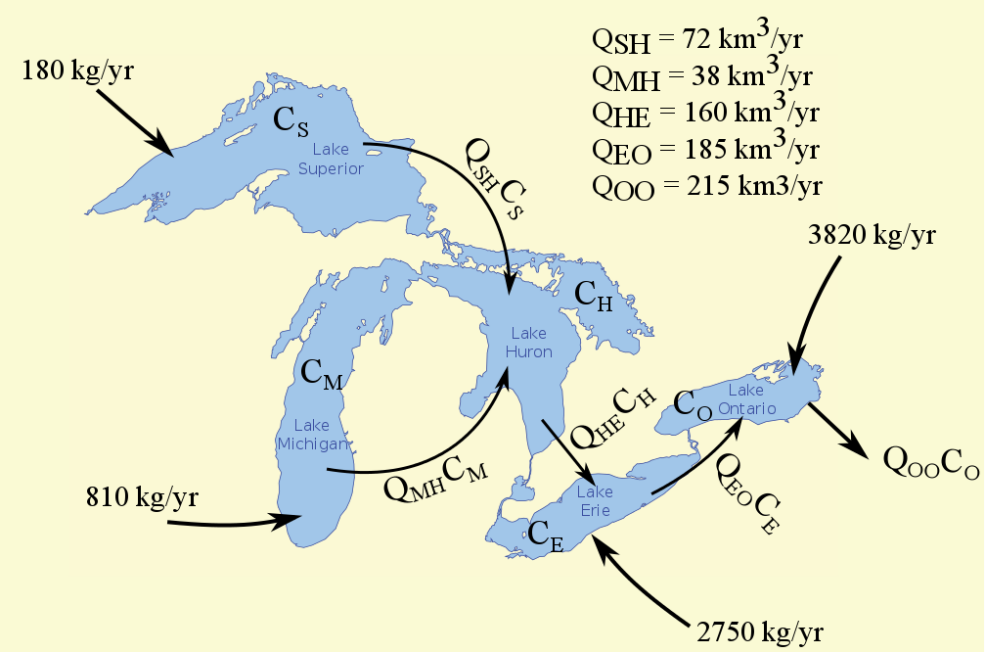


Figure 3.8: PCB contamination of the Great Lakes. On this diagram you can see all the input/output mass flow rates in the different lakes. The output of each lake is given by the product of its concentration C_i and a constant Q_{ij} whose value are given in the table.

```
        # block of instruction
        # ...
        res = .....
        niter += 1
```

Here the loop will continue as long as $(res > tol)$ and $niter < ITERMAX)$. Don't forget to update *res* and increment *niter* within the loop otherwise it will never stops. Ever. Test your Jacobi function to solve the linear system 3.17. Hint : The implementation of the Jacobi algorithm is very similar to the Gauss-Seidel algorithm whose implementation has been shown above.

3.3.6 Bonus : Gauss-Jordan Algorithm

In this exercise you will implement, without copy/pasting the solution provided in section 3.2.4 the Gauss-Jordan algorithm. To do so you will write two functions. The first that takes 2 input arguments (A and f) and return the augmented matrix (M) in its upper diagonal form. The second function will take this matrix as input argument and return the final solution vector. During the back substitution you may need for loops that run 'backward', i.e. with the variables going from N, $N-1$, ... $N-P$. You can do that easily with

```
for i in range(N,N-P,-1):
        # block of instruction
```

You will test your code with the linear system 3.17.

Chapter 4
Integration, Differentiation and Nonlinear systems

The basics of integration and differentiation are usually covered in the first courses of calculus. You should therefore be able to perform such calculation for most simple functions. However it is sometimes impossible to do so because you don't know the expression of the function. This happens frequently when analyzing experimental data for which you only know the numerical values of the function but not its expression. In such cases, numerical approximation for the derivation and integration are quite handy. In this chapter we will describe standard approaches for the numerical evaluation of integrals and derivatives. We will then see how to uses these expressions for the solution of non-linear systems of equations.

4.1 Numerical Integration

In this section we introduce simple methods allowing to evaluate a definite integral of the form

$$I = \int_a^b f(x)dx \tag{4.1}$$

We'll see that this integral can be approximated by summing up the area of a series of geometrical shapes like rectangles or trapezes. We then present more advanced methods that are already implemented in various Python modules.

4.1.1 Composite methods for numerical integration

The calculation of a definite integral in 1D is identical to calculating the area between the X-axis and the curve. To approximate the definite integral of a given function we can subdivide the x-axis into n intervals of length h. Decreasing h can lead to very accurate approximation of any integrals.

The simplest method to compute such integral is represented in the figure 4.1. As seen in this figure the X-axis is divided in a series of interval. The height of the rectangle filling the interval between x_i and x_{i+1} is set to $f(x_i)$, i.e. the value of the function at the beginning of the domain. Hence the approximation of the interval reads

$$\int_a^b f(x)dx \approx h \sum_{j=1}^{N-1} f(x_j) \tag{4.2}$$

The implementation of this simple method, and the evaluation of its performance, is the focus of the first exercise.

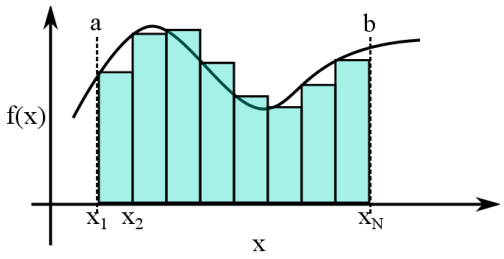


Figure 4.1: Illustration of simplest composite method for the numerical integration of curves. The interval is divided in small rectangle that spans the integration interval. The integral is obtained by summing up the area of all the rectangle.

Many methods have been developed to improve the accuracy of numerical integration. On of these methods, called the composite midpoint method, is represented in Fig. 4.2a. As seen on this figure, the area under the curve is also divided in a series of rectangles of width of h. However the height of each rectangle is given by value the function at the midpoint of the rectangle's width. The equation describing this integral is therefore given by:

$$\int_a^b f(x)dx \approx h \sum_{j=1}^{N-1} f(x_j) \tag{4.3}$$

where x_j is the midpoint of the j-th interval. A similar approach allows for an even greater reduction of the error. This method called the trapezoidal rule and uses trapeze and not rectangle to subdivide the integration interval. A graphical illustration of this method is given in Figure 4.2b. The equation giving the numerical value of the integral is now

$$\int_a^b f(x)dx \approx \frac{h}{2}\left[\sum_{j=0}^{N-1} f(x_j) + f(x_{j+1})\right] \tag{4.4}$$

The trapezoidal rule is generally more accurate than the midpoint method while having the same computational cost. It is therefore usually preferred to obtain fast and relatively accurate results. The trapezoidal rules is straightforward to implement in Pyhton and is the focus of the second exercise.

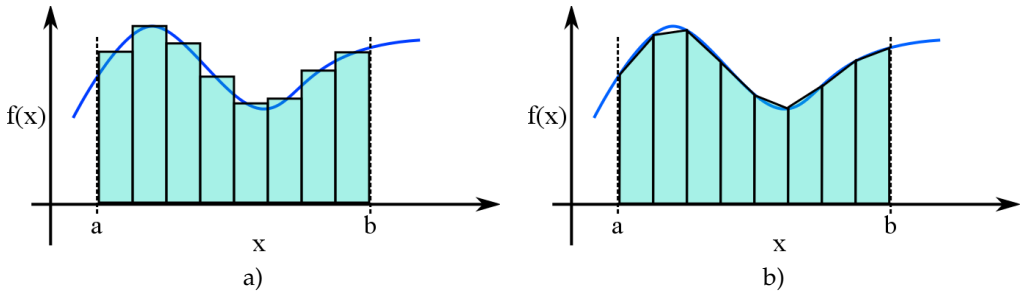


Figure 4.2: Illustration of the two different composite methods to evaluate the definite integral $\int_a^b f(x)dx$. a) the composite midpoint method where rectangles are used. b) the trapezoidal rule where trapezes are used to subdivide the integration interval.

4.1.2 Quadrature methods with Numpy

Several Python modules have of course many methods already implemented to compute the definite integral of a given function. In the example below we use the module *scipy* to evaluate an Gaussian quadrature approach. In this more advanced approach the grid used to compute the integral is not uniform. The Gaussian quadrature yield exact results for polynomial of degree $2n-1$ or less by taking suitable position x_i and weight w_i to compute the definite integral as

$$\int_{-1}^{1} f(x)dx = \sum_{i=1}^{n} w_i f(x_i) \tag{4.5}$$

In the Gauss-Legendre the points x_i, (called the i-th Gauss node) is the i-th root of the Legendre polynomial P_n. In order to apply this method to an arbitrary interval $[a;b]$ a change of interval to $[-1,1]$ must be performed. This is done simply by

$$\int_a^b f(x)dx = \frac{b-a}{2}\int_{-1}^{1} f(\frac{b-1}{2}x + \frac{a+b}{2})dx \tag{4.6}$$

$$= \frac{b-a}{2}\sum_{i=1}^{n} w_i f(\frac{b-1}{2}x_i + \frac{1+b}{2}) \tag{4.7}$$

It can be shown that the the weights w_i are then given by:

$$w_i = \frac{2}{(1-x_i)^2[P_n'(x_i)]^2} \tag{4.8}$$

If the implementation of this method and similar ones is outside the scope of this introduction course, they are the methods of choice to numerically evaluate integrals as they lead to great numerical accuracy at a very low computational cost. For example in the example below we show that the Gaussian quadrature implemented in *scipy* returns the value of the integral of $f(x) = x\sin(x)$ with an error of 10^{-12} which is the close to machine precision, i.e. the highest accuracy possible on a computer.

```
1  import numpy as np
2  import math
3  import scipy.integrate.quadrature as quad
4
5  def func(x):
6          return x*np.sin(x)
7
8  def exactIntegral(a,b):
9          Iab = -b*np.cos(b)+np.sin(b)+a*np.cos(a)-np.sin(a)
10         return Iab
11
12 a = 0.0
13 b = 2.0
14
15 exact = exactIntegral(a,b)
16 estimate = quad(func,a,b)
17
18 print "Exact %1.6f Numerical %1.6f" %(exact,estimate[0])
19 print "Error %1.3e" %np.abs(exact-estimate[0])
```

4.1.3 Multiple Integrals

We discuss briefly here how to numerically solve double integrals with constant limit such as

$$I = \int_a^b \left(\int_c^d f(x,y)dy \right) dx \tag{4.9}$$

The inner integral can be calculated with any of the methods seen previously. For example using the midpoint rule yields

$$I_1 = \int_c^d f(x,y)dy = h_y \sum_{j=0}^{N_y} f(x,y_j) \tag{4.10}$$

where y_j is the midpoint of the j-th interval. Introducing this expression in I and using the midpoint method again leads to

$$\begin{aligned} I &= \int_a^b \left(h_y \sum_{j=0}^{N_y} f(x,y_j) \right) dx & (4.11) \\ &= h_y \sum_{j=0}^{N_y} \int_a^b f(x,y_j)dx = h_x h_y \sum_{i=0}^{N_x} \sum_{j=0}^{N_y} f(x_i,y_j)dx & (4.12) \end{aligned}$$

As we could have anticipated, the integral is now the sum of the volume of small rectangular column that spans the xy plane. Any other composite method can be used to evaluate double integrals.

4.2 Numerical Differentiation

The expression for the numerical calculation of the the derivatives can be obtained via a Taylor expansion of the the function. Following this expansion, the value of the function at a distance h away from one point x_i is given by:

$$\begin{aligned} f(x_i+h) &= f(x_i) + h\frac{df(x_i)}{dx} + \frac{h^2}{2}\frac{d^2f(x_i)}{dx^2} + \dots & (4.13) \\ &= f(x_i) + h\frac{df(x_i)}{dx} + O(h^2) & (4.14) \end{aligned}$$

where the term $O(h^2)$ means that the magnitude of the error is on the order of h^2.

4.2.1 First derivative

To obtain an expression for the first derivative of f we can rearrange eq.4.14:

$$\begin{aligned} \frac{df(x_i)}{dx} &= \frac{f(x_i+h)-f(x_i)}{h} + O(h) & (4.15) \\ &\approx \frac{f(x_i+h)-f(x_i)}{h} & (4.16) \end{aligned}$$

This approximation is called the forward approximation of the first derivative and the error is here on the order of h. Numerical differentiation are often performed on experimental data such where the values of f come from the experimental apparatus as a series of points arranged in a 1D array separated

by a fixed distance on the X axis. To make that explicit let's change the notation to $f(x = 0) = f_0$, $f(h) = f_1$, $f(2h) = f_2$, etc ... Using this notation the forward approximation of the first derivative reads

$$\frac{df_i}{dx} = \frac{f_{i+1} - f_i}{h} \tag{4.17}$$

The use of this approximation is illustrated in Fig. 4.3.a. As you see this approximation becomes more and more accurate when the distance between the points decreases. Another solution exists to improve the accuracy of the numerical derivative without changing the distance between the points. To derive this expression let's write the backward approximation of the derivative. This approximation is given by the Taylor approximation of $f(x_i - h)$:

$$f(x_i - h) = f(x_i) - h\frac{df(x_i)}{dx} + O(h^2) \tag{4.18}$$

Using the same notation than before we can write

$$\frac{df_i}{dx} = \frac{f_i - f_{i-1}}{h} \tag{4.19}$$

This backward approximation has the same $O(h)$ accuracy than the forward approximation and it use is illustrated in Fig. 4.1.b. However one can combine the backward and forward substitution and write:

$$2\frac{df(x_i)}{dx} = \frac{f_{i+1} - f_i}{h} + \frac{f_i - f_{i-1}}{h} = \frac{f_{i+1} - f_{i-1}}{h} \tag{4.20}$$

and therefore

$$\frac{df(x_i)}{dx} = \frac{f_{i+1} - f_{i-1}}{2h} \tag{4.21}$$

This approximation is called the centered difference approximation and it can be shown that its accuracy is on the order $O(h^2)$ instead of $O(h)$ for the forward and backward approximation. This means that for a given step size h, the centered difference approximation will give much more accurate results that the backward and forward approximation. The centered approximation is illustrated in Fig. 4.3.c.

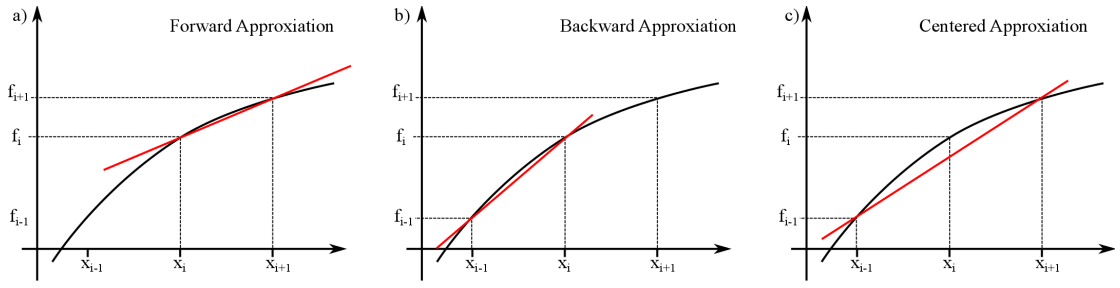


Figure 4.3: Illustration of the different numerical approximation for the first derivatives of a function a) forward approximation, b) backward approximation c) centered approximation

4.2.2 Example

We can illustrate the the convergence of the three different methods by writing a small Python code that compares the results of the three different methods to the exact values of the derivatives. We

choose here to compute the derivative of the function

$$f(x) = x\sin(x) \tag{4.22}$$

We first write a little of code to compute and plot the first derivative of this function. The snippet of code below compute the derivative using the centered difference approximation.

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

# compute the derivative numerically
def df_dx(y,x):
        h = x[1]-x[0]
        df = 0.5/h*(np.roll(y,-1)-np.roll(y,1))
        df[0] = 1./h*(y[1]-y[0])
        df[-1] = 1./h*(y[-1]-y[-2])
        return df

# compute the derivative analytically
def df_dx_exact(x):
        return np.sin(x)+x*np.cos(x)

# function to derive
def func(x):
        return x*np.sin(x)

# main part of the code
X = np.linspace(0,4*np.pi,1E5)
F = func(X)
dF = df_dx(F,X)
dF_exact = df_dx_exact(X)

# plot the result
plt.plot(X,F,color='black',linewidth=3,label='f')
plt.plot(X,dF,color='#006FFF',linewidth=3,label='df')
plt.plot(X[0:-1:5E3],dF_exact[0:-1:5E3],'o',markersize=8,color='#006FFF',label='df_exact')
plt.xlabel('X',fontsize=20)
plt.ylabel('f(X)',fontsize=20)
plt.legend(loc=2)
plt.show()
```

Note the use of the function

```
np.roll(y,1)
```

that returns an array identical to y except that all the elements have been 'rolled' along the vector. For example

$$y = [1,2,3,4,5] \longrightarrow y_{\text{roll}} = [5,1,2,3,4] \tag{4.23}$$

Note as well that the first and last element of the derivative cannot be evaluated using the central difference as the arrays stops at f[0] and f[-1]. We therefore have to use the forward and backward

difference for these two points. The comparison between the numerical and analytic expression of the derivative are reported in Fig. 4.4.

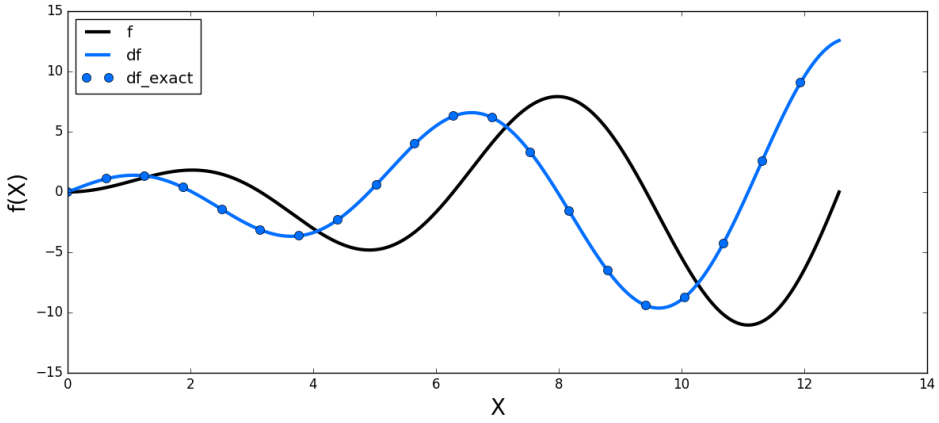


Figure 4.4: Example of numerical derivation of the function $f(x) = x\sin(x)$ and comparison with its exact derivative.

We can furthermore compared the accuracy of the forward, backward and central method for the calculation of the derivative. To do so the snippet of code below computes the value of the derivative of $f(x)$ at $x = 1.0$ and for several values of the interval h.

```
1  import numpy as np
2
3  def fun(x):
4          return x*np.sin(x)
5
6  def exactDeriv(x):
7          return np.sin(x) + x*np.cos(x)
8
9  x0 = 1.0
10 exact = exactDeriv(x0)
11
12 for i in range(1,7):
13
14         h = 10**(-i)
15
16         fa = (fun(x0+h)-fun(x0))/h
17         ba = (fun(x0)-fun(x0-h))/h
18         ca = (fun(x0+h)-fun(x0-h))/2/h
19
20         print " h = %1.4e" %h
21
22         print " forward approximation error %1.4e " %np.abs(exact-fa)
23         print " bacward approximation error %1.4e " %np.abs(exact-ba)
24         print " centered approximation error %1.4e " %np.abs(exact-ca)
25         print ""
```

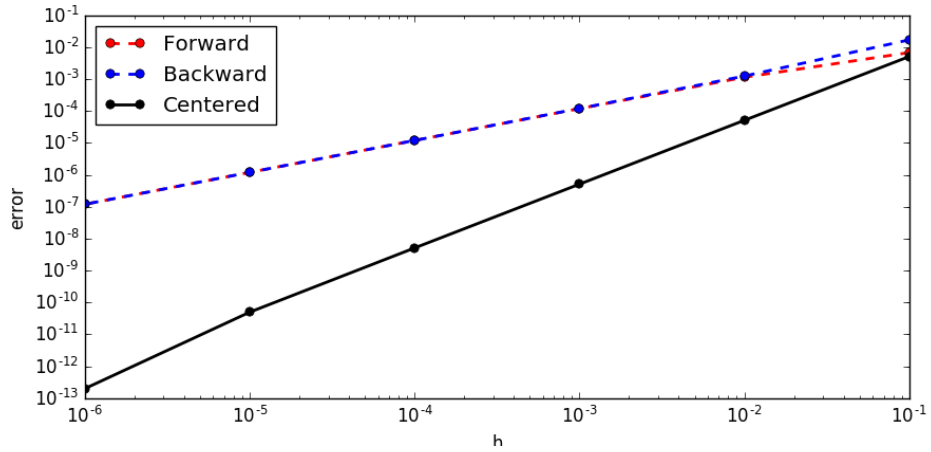


Figure 4.5: Illustration of the error obtained for the different numerical approximation of the derivative. As you can see the centered approximation leads to smaller error down to $h = 10^{-6}$.

As you can see in Figure 4.5, the error decreases faster for the centered approximation than for the backward and forward approximation.

4.2.3 Second derivative

The second derivative can also be evaluated numerically. To derive the expression we start once again from the Taylor approximation of a function at $x_i \pm h$:

$$f(x_i + h) = f(x_i) + h\frac{df(x_i)}{dx} + \frac{h^2}{2}\frac{d^2f(x_i)}{dx^2} + O(h^3) \tag{4.24}$$

$$f(x_i - h) = f(x_i) - h\frac{df(x_i)}{dx} + \frac{h^2}{2}\frac{d^2f(x_i)}{dx^2} - O(h^3) \tag{4.25}$$

If we add up these two expressions we obtain

$$f(x_i + h) + f(x_i - h) = 2f(x_i) + h^2\frac{d^2f(x_i)}{dx^2} + O(h^4) \tag{4.26}$$

This equation can be rearranged to obtain an approximation for the second derivative. Using the simplified notation for 1D array we have

$$\frac{d^2f_i}{dx^2} \approx \frac{f_{i+1} - 2f_i + f_{i-1}}{h^2} \tag{4.27}$$

which is $O(h^2)$ accurate. This is called the centered difference for the second derivative. More accurate expressions have been derived for the second derivative. However the expression 4.27 is used in most engineering as it is accurate and fast. The function $np.roll()$ can also be used to compute rapidly the values of the second derivative

```
1 d2f = 1./h/h * (np.roll(f,-1)-2*f+np.roll(f,1))
```

Note however that the elements $d2f[0]$ and $d2f[-1]$ are not evaluated correctly. To evaluate these elements we use the forward and backward expression

$$\frac{d^2 f_i}{dx^2} = \frac{1}{h^2}(f_{i+2} - 2f_{i+1} + f_i) + O(h) \tag{4.28}$$

$$\frac{d^2 f_i}{dx^2} = \frac{1}{h^2}(f_i - 2f_{i-1} + f_{i-2}) + O(h) \tag{4.29}$$

4.2.4 Partial derivatives in two dimensions

We have seen how to extract numerical expression of the first and second derivatives of 1D arrays. Similar techniques can be used to derive the partial derivatives of a function $f(x, y)$ that depends then on two variables. Let's adopt the notation

$$f(x_i, y_j) = f_{ij} \tag{4.30}$$

We suppose that the grid points are evenly spaced with an increment h in the x direction and k in the y direction. For the first derivatives the central difference formula directly leads to:

$$\frac{d}{dx} f(x_i, y_j) = \frac{1}{2h}(f_{i+1,j} - f_{i-1,j}) \tag{4.31}$$

$$\frac{d}{dy} f(x_i, y_j) = \frac{1}{2k}(f_{i,j+1} - f_{i,j-1}) \tag{4.32}$$

The second partial derivatives are also given by know expression:

$$\frac{d^2}{dx^2} f(x_i, y_j) = \frac{1}{h^2}(f_{i+1,j} - 2f_{i,j} + f_{i-1,j}) \tag{4.33}$$

$$\frac{d^2}{dy^2} f(x_i, y_j) = \frac{1}{k^2}(f_{i,j+1} - 2f_{i,j} + f_{i,j-1}) \tag{4.34}$$

The mixed partial derivatives can then be obtained by combining first derivatives expression

$$\frac{d^2}{dxdy} f(x_i, y_j) = \frac{d}{dx}\left(\frac{d}{dy}\right) f(x_i, y_j) \tag{4.35}$$

$$= \frac{d}{dx}\left(\frac{1}{2k}(f_{i,j+1} - f_{i,j-1})\right) \tag{4.36}$$

$$= \frac{1}{4hk}\left(f_{i+1,j+1} - f_{i-1,j+1} - f_{i+1,j-1} + f_{i-1,j-1}\right) \tag{4.37}$$

4.2.5 Summary

There is many ways of deriving finite difference expression of the derivatives of a given function depending on the method used (forward,backward,central) and the desired accuracy. We tabulate here the different expressions that will be useful in the following of the course.

Forward difference : Error of order h	**Forward difference : Error of order h^2**
$\frac{dy_i}{dx} = \frac{1}{h}(y_{i+1} - y_i)$ $\frac{d^2y_i}{dx^2} = \frac{1}{h^2}(y_{i+2} - 2y_{i+1} + y_i)$	$\frac{dy_i}{dx} = \frac{1}{2h}(-y_{i+2} + 4y_{i+1} - 3y_i)$ $\frac{d^2y_i}{dx^2} = \frac{1}{h^2}(-y_{i+3} + 4y_{i+2} - 5y_{i+1} + 2y_i)$

Backward difference : Error of order h	**Backward difference : Error of order h^2**
$\frac{dy_i}{dx} = \frac{1}{h}(y_i - y_{i-1})$ $\frac{d^2y_i}{dx^2} = \frac{1}{h^2}(y_i - 2y_{i-1} + y_{i-2})$	$\frac{dy_i}{dx} = \frac{1}{2h}(3y_i - 4y_{i-1} + y_{i-2})$ $\frac{d^2y_i}{dx^2} = \frac{1}{h^2}(2y_i - 5y_{i-1} + 4y_{i-2} - y_{i-3})$

Central difference : Error of order h^2

$$\frac{d^2f_i}{dx^2} = \frac{f_{i+1} - 2f_i + f_{i-1}}{h^2}$$
$$\frac{d^2}{dxdy}f(x_i, y_j) = \frac{1}{4hk}\left(f_{i+1,j+1} - f_{i-1,j+1} - f_{i+1,j-1} + f_{i-1,j-1}\right)$$

4.3 NonLinear Systems

The calculation of the derivative of a given function occurs in many engineering problem. A classic example is search for the minimum of a function as we will see in Exercise XX. Another application is the the solution of nonlinear equations and nonlinear system of equation as presented here.

4.3.1 Nonlinear equations

In contrast with linear systems studied in the last chapter, nonlinear equations can contain powers of the unknown (x_i^3), product of different unknown ($x_i x_j$) or more complicated terms ($\cos(x_i)$, e^{x_k} ...). For example let's solve the equation

$$f(x) = x^2 - 2x = 0 \tag{4.38}$$

A method to solve this non-linear equation can be derived from the Taylor expansion

$$f(x) = f(x_0) + (x - x_0)f'(x_0) + \frac{(x - x_0)^2}{2!}f''(x_0) + O(x - x_0)^3 \tag{4.39}$$

where the last term tells us that the error is on the order of $(x - x_0)^3$). Here we want to find $f(x) = 0$. If we have a guess, called x_0, for the value of x that leads to $f(x) = 0$ we can rewrite the Taylor expansion as

$$0 = f(x_0) + (x - x_0)f'(x_0) \tag{4.40}$$

If we rearrange the terms of this equation we obtain a better estimate of the value of x for which $f(x) = 0$

$$x = x_0 - \frac{f(x_0)}{f'(x_0)} = x_0 - f(x_0)\frac{2h}{f(x_0+h) - f(x_0-h)} \tag{4.41}$$

where the expression of the numerical derivative (eq. 4.21) has been used. We can iterate this procedure until convergence to obtain an accurate value for the solution of any nonlinear equation. The little snippet of code below use this method, called the Newton's method, to solve a simple nonlinear equation.

```
import numpy as np

# non linear function
def func(x):
        return x**2 - 2*x


# compute the derivative numerically
def fprime(func, **kwargs):
        x = kwargs.get('x')
        h = kwargs.get('h')
        return (func(x+h)-func(x-h))/2./h

# initial guess
x0 = 5.0

# step for the derivatives
step = 1E-3

# initial convergence and tolerance
eps, tol = 1, 1E-6

# iterations and maximum iterations
niter, MAXITER = 1, 100

# loop until convergence is reached
while (eps>tol) and (niter<MAXITER):

        # compute the function and its derivative at x=x0
        f = func(x0)
        fp = fprime(func,x=x0,h=step)

        # compute the new guess with the newton method
        x1 = x0 - f/fp

        # compute the covergence criteria and
        eps = np.abs(x1-x0)

        # if we have converged we print the solution
        if (eps<tol):
```

```
41                print '\n \tConvergence reached'
42                print '\tSolution at x = %f\n' %(x1)
43
44        # else we update the old guess
45        else:
46                x0 = x1
```

The illustration of this method is given in Figure 4.6. As we can see in this figure, the Newton's method approximate the nonlinear equation as by a line tangent to the function at the current guess of the zero's position. That line is then used to compute a new estimation of the solution that is hopefully better than the previous one. The process is then repeated until convergence.

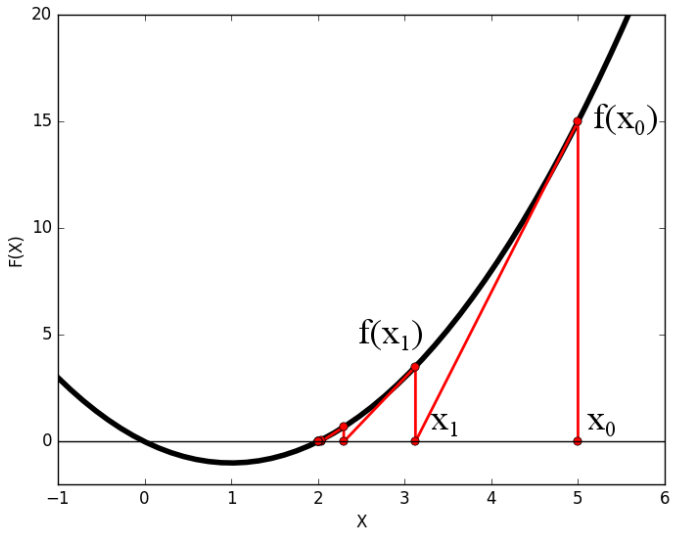


Figure 4.6: Illustration for the Newton's method used to find the solution of nonlinear equation.

4.3.2 Two coupled nonlinear equations

The Newton method presented above can be extended to solve systems involving simultaneous nonlinear equations. We first present a simple example with only two equations and then sketch the general solutions for k equations in the next section.

Suppose we have a model of 2 unknown x_1 and x_2 related to each other by two nonlinear equations that have the general form

$$f_1(x_1, x_2) = 0 \tag{4.42}$$
$$f_2(x_1, x_2) = 0 \tag{4.43}$$

where f_1 and f_2 are non linear. We can use the Taylor series to expand these functions around our estimate of the solution noted $x_1^{(0)}$ and $x_2^{(0)}$

$$f_1(x_1, x_2) = f_1(x_1^{(0)}, x_2^{(0)}) + \frac{\partial f_1}{\partial x_1}|_{x^{(0)}} \cdot (x_1 - x_1^{(0)}) + \frac{\partial f_1}{\partial x_2}|_{x^{(0)}} \cdot (x_2 - x_2^{(0)}) + ... \tag{4.44}$$

$$f_2(x_1, x_2) = f_2(x_1^{(0)}, x_2^{(0)}) + \frac{\partial f_2}{\partial x_1}|_{x^{(0)}} \cdot (x_1 - x_1^{(0)}) + \frac{\partial f_2}{\partial x_2}|_{x^{(0)}} \cdot (x_2 - x_2^{(0)}) + ... \tag{4.45}$$

Since we search for the zeros of these functions we can set the left side of these equations to 0. If we truncate the Taylor series to the first order and rearrange the equations we obtain

$$\frac{\partial f_1}{\partial x_1}|_{x^{(0)}} \cdot (x_1 - x_1^{(0)}) + \frac{\partial f_1}{\partial x_2}|_{x^{(0)}} \cdot (x_2 - x_2^{(0)}) = -f_1(x_1^{(0)}, x_2^{(0)}) \tag{4.46}$$

$$\frac{\partial f_2}{\partial x_1}|_{x^{(0)}} \cdot (x_1 - x_1^{(0)}) + \frac{\partial f_2}{\partial x_2}|_{x^{(0)}} \cdot (x_2 - x_2^{(0)}) = -f_2(x_1^{(0)}, x_2^{(0)}) \tag{4.47}$$

We define now the quantity

$$\delta_1^{(0)} = x_1 - x_1^{(0)} \tag{4.48}$$

$$\delta_2^{(0)} = x_2 - x_2^{(0)} \tag{4.49}$$

to simplify the above equations to

$$\frac{\partial f_1}{\partial x_1}|_{x^{(0)}} \cdot \delta_1^{(0)} + \frac{\partial f_1}{\partial x_2}|_{x^{(0)}} \cdot \delta_2^{(0)} = -f_1(x_1^{(0)}, x_2^{(0)}) \tag{4.50}$$

$$\frac{\partial f_2}{\partial x_1}|_{x^{(0)}} \cdot \delta_1^{(0)} + \frac{\partial f_2}{\partial x_2}|_{x^{(0)}} \cdot \delta_2^{(0)} = -f_2(x_1^{(0)}, x_2^{(0)}) \tag{4.51}$$

Similarly to the case of linear equation the system above can be written as a matrix equation

$$\begin{pmatrix} \frac{\partial f_1}{\partial x_1}|_{x^{(0)}} & \frac{\partial f_1}{\partial x_2}|_{x^{(0)}} \\ \frac{\partial f_2}{\partial x_1}|_{x^{(0)}} & \frac{\partial f_2}{\partial x_2}|_{x^{(0)}} \end{pmatrix} \cdot \begin{pmatrix} \delta_1^{(0)} \\ \delta_2^{(0)} \end{pmatrix} = \begin{pmatrix} f_1^{(0)} \\ f_2^{(0)} \end{pmatrix} \tag{4.52}$$

or even

$$\mathcal{J} \cdot \boldsymbol{\delta} = -\mathbf{f} \tag{4.53}$$

The matrix $\mathcal{J}$ is called the *Jacobian* of the system. The elements of the Jacobian can be computed numerically for any differentiable function. Therefore the matrix eq. 4.52 can be solved using any techniques presented in the last chapter for the solution of linear systems. This solution provide values for $\delta_1^{(0)}$ and $\delta_2^{(0)}$ that can be added to our initial guess of the solution to obtain a better guess of the solution

$$x_1^{(1)} = x_1^{(0)} + \delta_1^{(0)} \tag{4.54}$$

$$x_2^{(1)} = x_2^{(0)} + \delta_2^{(0)} \tag{4.55}$$

These new values can now be used in eq. 4.52 to obtain even better estimate and so on.

4.3.3 Example of two coupled nonlinear equations

As an example suppose we want to solve the set of nonlinear equations

$$f_1(x_1, x_2) = x_1^3 + x_2^2 = 0 \tag{4.56}$$
$$f_2(x_1, x_2) = x_1^2 - x_2^3 = 0 \tag{4.57}$$

We can write the Jacobian of this system right away

$$\mathcal{J} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{pmatrix} = \begin{pmatrix} 3x_1^2 & 2x_2 \\ 2x_1 & -3x_2^2 \end{pmatrix} \tag{4.58}$$

We start the Newton algorithm by taking our initial guess for the solutions as

$$x_1^{(0)} = 1 \tag{4.59}$$
$$x_2^{(0)} = 2 \tag{4.60}$$

In that case the first iteration of the method leads to the linear system of equation

$$\begin{pmatrix} 3 & 4 \\ 2 & -12 \end{pmatrix} \cdot \begin{pmatrix} \delta_1^{(0)} \\ \delta_2^{(0)} \end{pmatrix} = - \begin{pmatrix} 5 \\ -7 \end{pmatrix} \tag{4.61}$$

Solving this matrix equation with for example the Jacobi iteration leads to:

$$\delta_1^{(0)} = -0.7272727 \tag{4.62}$$
$$\delta_2^{(0)} = -0.70454545 \tag{4.63}$$

and therefore

$$x_1^{(1)} = x_1^{(0)} + \delta_1^{(0)} = 1.0 - 0.7272727 = 0.272727272 \tag{4.64}$$
$$x_2^{(1)} = x_2^{(0)} + \delta_2^{(0)} = 2.0 - 0.70454545 = 1.29545455 \tag{4.65}$$

We can now use these new values of the solution to recompute the Jacobian $\mathcal{J}$ and the right-hand side vector $\mathbf{f}$ to obtain $\delta_1^{(1)}$ and $\delta_2^{(1)}$ and then new solution. The Newton method is implemented in the snippet of code below.

```
import numpy as np

# non linear function
def func(x1,x2):
        return np.array([x1**3 + x2**2,x1**2 - x2**3])

# compute the Jacobian
def jacobian(func, **kwargs):
        x1 = kwargs.get('x1')
        x2 = kwargs.get('x2')
        J = np.array([[3*x1**2,2*x2 ],[2*x1,-3*x2**2]])
        return J

# initial guess
x0 = np.array([1.0,2.0])
```

```
16
17 print 'initial guess'
18 print "x1 = %f \t x2 = %f" %(x0[0],x0[1])
19 print ''
20 # initial convergence and tolerance
21 eps, tol = 1, 1E-6
22
23 # iterations and maximum iterations
24 niter, MAXITER = 1, 1000
25
26 # loop until convergence is reached
27 while (eps>tol) and (niter<MAXITER):
28
29         # compute the function and the jacobian
30         f = func(x0[0],x0[1])
31         j = jacobian(func,x1=x0[0],x2=x0[1])
32
33         # solve the linear system
34         delta = np.linalg.solve(j,-f)
35
36         # uptade the solution
37         x1 = x0 + delta
38
39         # check if we have converged
40         f1 = func(x1[0],x1[1])
41         eps = np.linalg.norm(f1)
42
43         print "x1 = %f \t x2 = %f \t eps = %f" %(x1[0],x1[1],eps)
44
45
46         if (eps < tol):
47                 print '\n \tConvergence reached in %d iteration' %niter
48                 print '\tSolution at x = [%f %f]\n' %(x1[0],x1[1])
49
50         else:
51                 x0 = x1
52                 niter += 1
```

As we can see the code converges toward the correct solution, i.e. $x_1 = -1$ $x_2 = 1$ in 613 iterations. As it can be seen by printing the convergence criteria, the error increases quite dramatically in the first iterations of the method. This explains why so many iterations are required to converge toward the solution.

4.3.4 Reduced Newton Method

To limit the number of step taken during the optimization we can search along the direction of the Newton step for a point where the error is less than the error at the starting point. This can be done simply by adding the following snippet of code at line 32 of the code shown above

```
1         # uptade the solution
```

```
        isearch = 0
        max_search = 10
        eps1 = 2*eps

        while (eps1 > eps) and (isearch<max_search):

                # new solution
                x1 = x0 + 0.5**(isearch)*delta

                # check if we have converged
                f1 = func(x1[0],x1[1])
                eps1 = np.linalg.norm(f1)

                # increment
                isearch += 1

        # update the convergence
        eps = eps1
```

As we can see on this script the new solution is here recalculated as

$$x^{(n+1)} = x^{(n)} + 0.5^{\text{isearch}}\delta^{(n)} \tag{4.66}$$

with isearch is incremented from 0 to 10 as long as the error is larger than the one obtained with $x^{(n)}$. Using this simple optimization of the Newton step, the method converges in only 11 iterations to the correct solution.

4.3.5 Multiple nonlinear equations

The Newton method can be generalized to a system of k coupled nonlinear equation

$$f_1(x_1, \dots x_k) = 0 \tag{4.67}$$
$$\dots\dots \tag{4.68}$$
$$f_k(x_1, \dots x_k) = 0 \tag{4.69}$$

Using the same demonstration than the one provided above we can write this system as a matrix equation

$$\begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_k} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_k} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_k}{\partial x_1} & \frac{\partial f_k}{\partial x_2} & \cdots & \frac{\partial f_k}{\partial x_k} \end{pmatrix} \cdot \begin{pmatrix} \delta_1 \\ \delta_2 \\ \vdots \\ \delta_k \end{pmatrix} = - \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_k \end{pmatrix} \tag{4.70}$$

The elements of the Jacobian can be either calculated via their analytic expressions or via the numerical expression of the first derivatives given in eq. 4.21.

4.4 Exercises Integration

4.4.1 Numerical integration with the composite method

In the first exercise you will implement the simplest composite method for the numerical integration of simple functions illustrated in Fig. 4.1. As a reminder In this method the definite integral

$$I = \int_a^b f(x)dx \tag{4.71}$$

is approximated in the by the summing the area of a number of rectangles that spans the curve to be integrated. The formula of this composite method reads

$$\int_a^b f(x)dx \approx h \sum f(x_j) \tag{4.72}$$

where h is the width of the rectangle and $f(x_j)$ the height of the rectangle starting at x_j. Write a function that compute this integral. This function should take as argument, the beginning and end of the integration interval a and b, the number of integration points N and the function to be integrated $func$. You will compare the results provided by code with results given by the routine `scipy.integrate.quadrature` presented in section 4.1.2. To test your code compute the integral of

$$f(x) = 2\pi x^2 \sin(\pi x) \exp(-x/2\pi) \tag{4.73}$$

between 0 and 0.1. Plot the difference in the $\Delta_{rect} = I_{rect} - I_{sp}$ (where I_{rect} and I_{sp} are the values of the integral obtained with your composite function and the quadrature of scipy) as a function of N. HINT : the structure of the code should look like that

```
import .....

# function that we want to integrate
def func(x):
        return .....

# function that perform the
numerical integration with rectangles
def int_rect(a,b,N,func):
        ....
        for i in range(....):
                I += ....
        return I

# integration interval
a, b = 0, 10

# different number of points
# in the interval
N = [10,100,1000,...]

for n in ...

        # compute the integral with
        # your function
        I = int_rect( .... )

        # compute the integral with scipy
        Isp = quad( .... )
```

```
30
31          # difference
32          Delta = .....
33
34  plt.loglog(....)
35  plt.show
```

NOTE : the function `plt.loglog` plot the curve on log-log axis.

4.4.2 Numerical integration with the trapeze method

Modify your code to use the trapeze method instead of the rectangle method. As a reminder, the trapeze method uses trapeze to approximate the integral of a given curve. The formula for trapeze integration reads:

$$\int_a^b f(x)dx \approx \frac{h}{2}\left[\sum f(x_j) + f(x_{j+1})\right] \tag{4.74}$$

The structure of the code is very similar to the one used in the previous exercise. You will write an additional function `int_trap` to compute this integral. The structure of this function should read:

```
1  # function that perform the
2  numerical integration with rectangles
3  def int_trap(a,b,N,func):
4          ....
5          for i in range(....):
6                  I += ....
7          return I
```

To compare the performance of the rectangle and trapeze method you will plot on top of Δ_{rect} computed in the last exercise, $\Delta_{trap} = I_{trap} - I_{sp}$ (where I_{trap} and I_{sp} are the values of the integral obtained with your trapeze function and the quadrature of scipy) as a function of N.

4.4.3 Enthalpy calculation

The calculation of the amount of energy required to increase the temperature of a material is a common problem in engineering. For example, if we have a cubic meter of nitrogen how much energy is required to increase its energy by 1^oC ? A refinery has hired you to automate such calculation and compare different techniques to do it.

The enthalpy change associated with a temperature change from T_1 to T_2 is given by

$$\Delta H = \int_{T_1}^{T_2} C_p(T)dT \tag{4.75}$$

where $C_p(T)$ is the heat capacity of the materials at a constant pressure p. This quantity is often given as a polynomial. For example the heat capacity of nitrogen gas (given in $kJ/(mol^oC)$) is

$$C_p(T) = 0.0290 + 0.2199 \times 10^{-5}T + 0.5723 \times 10^{-8}T^2 - 2.871 \times 10^{-12}T^3 \tag{4.76}$$

where T is in **Celsius**. On the other hand the heat capacity of carbon is given by the polynomial expression

$$C_p(T) = 0.1118 + 1.095 \times 10^{-5}T + 489.1/T^2 \tag{4.77}$$

where T is in **Kelvin**. The refinery wants you to compute the the enthalpy change for both material starting at $T = 20\ ^oC$ and raising the temperature to $T = 100\ ^oC$. Report the enthaply change in $kJ/(mol)$. You should numerically evaluate the integral using the `scipy.integrate.quadrature` function.

The answers are 2.33 kJ/mol for nitrogen and 9.65 kJ/mol for carbon.

4.5 Exercises Non-Linear System

4.5.1 Implementation of the Newton method

This exercise propose to implement the Newton method for the solution of a non-linear equation. The Newton method has been presented in section 4.3.1 and a pseudo code is given on page 11. You can use this structure to create your own implementation of the Newton's method.

As a reminder the Newton method allow to find the solution of a non linear equation, as for example $f(x) = x^2 - 2x = 0$, iteratively. We therefore start with a guess x_0. Most likely this guess does not satisfy $f(x_0) = 0$, i.e, it is not the solution you're looking for. We therefore have to improve the quality of our solution. To do that the Newton method relies of the iterative procedure

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{4.78}$$

where $f'(x_n)$ is the derivative of $f(x)$ evaluated at $x = x_n$. As demonstrated during the lecture, this iterative procedure converges toward the exact solution leading to $f(x_N) \approx 0$. Implement the Newton method and test your implementation by finding the solution of

$$\cos(x)\exp(-\frac{x}{2\pi}) - \frac{x}{\pi} = 0 \tag{4.79}$$

Start with an initial guess of $x_0 = 5$. Your method should reach the solution $x = 1.127117$ in a few iterations. The Newton method applied to this problem is illustrated in Fig. 4.7. What happen if you change the initial guess to $x_0 = -5$?

4.5.2 Application of the Newton method : Tank explosion

A propane tank was installed in a backyard. Unfortunately on a particularly hot day the tank exploded. You've been hired to calculate the number of mole n of propane contained in the tank just before the explosion. To do so you will use the van der Waals equation of state

$$(P + \frac{a}{(V/n)^2})(V/n - b) - RT = 0 \tag{4.80}$$

where

$$a = \frac{27}{64}\frac{R^2T_C^2}{P_c} \qquad b = \frac{1}{8}\frac{RT_C}{P_c} \tag{4.81}$$

You firm has provided you with the last reading of the tank $T = 384$ K and P = 4891.3 kPa. The volume of the tank is $V = 0.15\ m^3$. In addition the critical temperature and pressure of propane are $T_c = 369.9$ K and $P_c = 4254.6$ kPa. Finally the gas constant is $R = 0.008314\ m^3kPa/(mol.K)$. Solve the van der Waals equation to obtain the number of mole n in the tank at the time of the explosion. To solve this equation you can use the Newton method implemented in the previous exercise. Check that

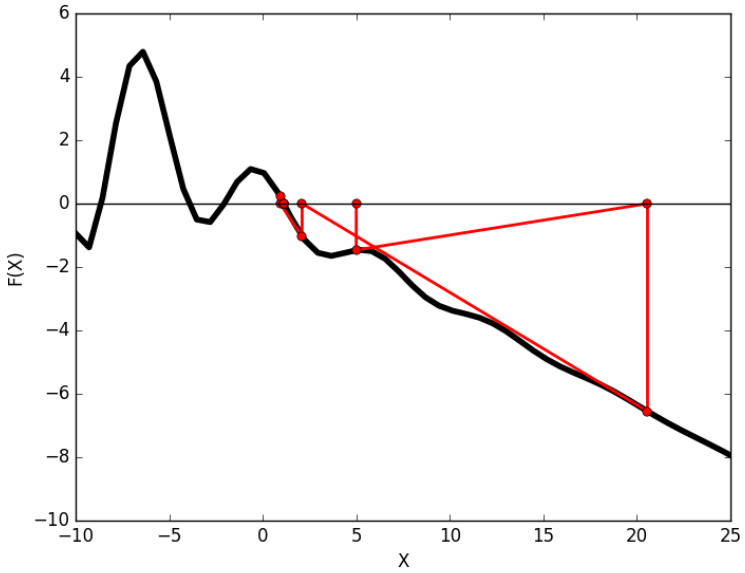


Figure 4.7: Illustration of the functioning of the Newton method applied to eq 4.79. Starting from $x_0 = 5$ the method converges to the solution of the equation, i.e. $x = 1.127117$.

there is only one solution of this equation by plotting the left side of the van der Waals equation as a function of n.

The answer is $n = 546.51$.

4.6 Exercises Bonus

4.6.1 Numerical Integration with the Simpson's rule

You've been hired by a software company to implement the so-called Simpson's rule for the evaluation of integrals. The Simpson's rules is a composite method similar to the midpoint and trapeze approach presented above. At the difference of these methods the Simpson's rule uses a quadratic polynomial to perform the integral over each interval piece. Thus the integral between the points x_i and x_{i+1} is given by

$$\int_{x_i}^{x_{i+2}} f(x)dx \approx \frac{h}{3}\left[f_i + 4f_{i+1} + f_{i+2}\right] \tag{4.82}$$

Write a function that takes two 1D array as arguments, that represent the x and $f(x)$ values. The function will return the value of the integral given by the Simpson rule. The function will be checked by computing the integral $\int_0^2 x\sin(x)dx$. Note that the number of points in the array must be odd. Note as well that eq. 4.82 goes from x_i to x_{i+2}. Be careful not to count things twice ! Compare the accuracy of the Simpson rule with those of the trapezoid method.

4.6.2 Minimization of a 2D function : the Gradient descent

Let's assume that we need to find the minimum of a function of two variables: $f(x,y)$. To do so we can starts at a given point (x_0, y_0), compute the gradient of the function at that point $\nabla f(x,y)$ and move 'a little bit' in the direction of the gradient to reach a second point (x_1, y_1). Hence the relation to from one point to the next one is given by

$$x^{n+1} = x_n - \alpha \nabla f(x,y)|_{x_n} \tag{4.83}$$

Implement this approach, called the gradient descent approach to find the minimum of two functions

$$f_1(x,y) = x^2 + 4 * y^2 \qquad x,y \in [-3;3] \tag{4.84}$$

$$f_2(x,y) = -\sin(\frac{x^2}{2} - \frac{y^2}{4} + 3)\cos(2x + 1 - e^y) \qquad x,y \in [-3;1] \tag{4.85}$$

In each case you will start the search for different initial guess of the solution (x_0) and report on the minimum found by the gradient descent approach. You will obtain figure similar to the one represented in Fig. 4.7.

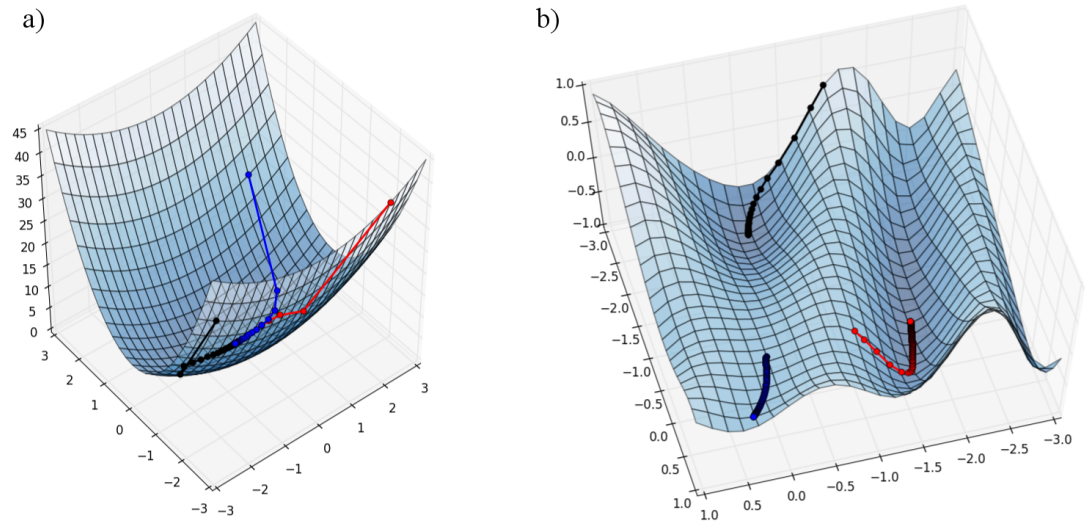


Figure 4.8: Finding the minimum of a 2D surface via the gradient descent approach on two different surface a) $f_1(x,y)$ and b) $f_2(x,y)$ and using 3 different initial guess.

Chapter 5
Ordinary differential equation

Quite often in engineering problems we want to know how a given system that is initially in a know state, evolves if we change one parameters. For example what happens to the concentration of a chemical specie if we suddenly introduce a reactant in the batch. In these problems we generally want to know how fast and how important are the changes. For example consider the following chemical reaction

$$\mathrm{A} + \mathrm{B} \underset{\mathrm{k_2}}{\overset{\mathrm{k_1}}{\rightleftharpoons}} \mathrm{C} + \mathrm{D} \xrightarrow{\mathrm{k_3}} \mathrm{E}$$

that takes place in a reactor. To model this reaction we can apply the material balance: Input + Generation = Output + Accumulation. However in a closed reactor, the Input and Output are null leaving only : Generation = Accumulation. Assuming that there is no volume change in the reactor we can therefore write

$$\frac{dC_A}{dt} = -k_1 C_A C_B + k_2 C_C C_D \tag{5.1}$$
$$\frac{dC_B}{dt} = -k_1 C_A C_B + k_2 C_C C_D \tag{5.2}$$
$$\frac{dC_C}{dt} = k_1 C_A C_B - k_2 C_C C_D - k_3 C_C C_D \tag{5.3}$$
$$\frac{dC_D}{dt} = k_1 C_A C_B + k_2 C_C C_D - k_3 C_C C_D \tag{5.4}$$
$$\frac{dC_E}{dt} = +k_3 C_C C_D \tag{5.5}$$

that constitutes a system of *first order nonlinear* differential equations where C_X is the concentration of the X in the reactor. The solution of the this system provides the evolution of these concentration in time under a given initial condition. This initial condition is here the concentration at the beginning of the reaction, i.e. $C_X(t = 0)$. Such problem, where the initial conditions are know is called an *initial value problem*. In contrast when the variables are know at know for different values of the independent variable (here t), the system of equation is called a *boundary value problem*. The solution of initial value problem is presented in section 5.3 and those of boundary value problem in section 5.4

5.1 Transformation to a canonical form

The chemical problem presented above is in the ideal form to be solved numerically. In this so-called *canonical form* the system of equation contains only first derivatives and none of the terms is directly

dependent on the independent variable (i.e. t). Any equation where none of the term depend on the explicit variable is *autonomous*. The system is however *nonlinear* as terms such as $C_X C_Y$ appear in the equation. More complicated ODEs are often encountered in chemical engineering problem. This is fir example the case of the following equation.

$$\text{2nd order} \qquad \frac{d^2y}{dt^2} + y\frac{dy}{dt} = e^{-t} \tag{5.6}$$

$$\text{3rd order} \qquad \frac{d^3y}{dt^3} + a\frac{d^2y}{dt^2} + b\frac{dy}{dt} + y = 0 \tag{5.7}$$

Equation 5.6 contains a second derivative and is therefore of the 2nd order. In addition one term e^{-t} depends explicitly on the independent variable. Any equation where on term depends on the explicit variable is *non-autonomous*. Similarly eq. 5.7 contains not only a second but also a third derivative and is therefore of the third order.

Numerical integration of ODEs is much more convenient and computationally effective when the system is in a canonical form: first order only and autonomous. Simple substitution can be used to transform complex systems of equations in a canonical form.

5.1.1 High-order autonomous equations

Consider the third-order equation 5.7. We want here to transform this high-order ODE in a series of coupled first-order ODEs. To do so we can introduce new variables as

$$y_1 \equiv \frac{dy}{dt} \tag{5.8}$$

$$y_2 \equiv \frac{dy_1}{dt} = \frac{d^2y}{dt^2} \tag{5.9}$$

and therfore $\frac{dy_2}{dt} \equiv d^3y/dt^3$. Using these new variables we can rewrite eq. 5.7 as

$$\frac{dy}{dt} = y_1 \tag{5.10}$$

$$\frac{dy_1}{dt} = y_2 \tag{5.11}$$

$$\frac{dy_2}{dt} = -ay_2 - by_1 - y \tag{5.12}$$

and we have succeeded in transforming a high-order ODE in a series of coupled first order ODEs.

5.1.2 Non-autonomous equations

Consider now eq. 5.6. This equation is of the second order and is non-autonomous. To transform this equation in a canonical form we can introduce the following variables

$$y_1 \equiv \frac{dy}{dt} \tag{5.13}$$

$$y_2 \equiv e^{-t} \tag{5.14}$$

With these substitutions we obtain the system of equations

$$\frac{dy}{dt} = y_1 \tag{5.15}$$
$$\frac{dy_1}{dt} \equiv \frac{d^2y}{dt^2} = -yy_1 + y_2 \tag{5.16}$$
$$\frac{dy_2}{dt} \equiv -e^{-t} = -y_2 \tag{5.17}$$

Once again we have reduced a higher-order non-autonomous ODE in a canonical form that is much easier to solve numerically. Note however that these equations are still *nonlinear* due to the term yy_1 in the equation for dy_1/dt.

5.2 Linear Ordinary Differential Equations

Many physicochemical problems are model by a set of linear ordinary equation with constant coefficients. This is for example the case of the simple reactions

$$\mathrm{A} \underset{k_2}{\overset{k_1}{\rightleftharpoons}} \mathrm{B} \underset{k_4}{\overset{k_3}{\rightleftharpoons}} \mathrm{C}$$

By applying the same method than for the previous equation we can derive a set of equations that model this reaction. We hence obtain

$$\frac{dC_A(t)}{dt} = -k_1C_A + k_2C_B \tag{5.18}$$
$$\frac{dC_B(t)}{dt} = k_1C_A + k_4C_C - (k_2 + k_3)C_B \tag{5.19}$$
$$\frac{dC_C(t)}{dt} = -k_4C_C + k_3C_B \tag{5.20}$$
$$\tag{5.21}$$

As for the linear systems of equation a matrix form can be adopted to write this system of ODEs in a more convenient way. We can hence write

$$\begin{pmatrix} \frac{dC_A}{dt} \\ \frac{dC_B}{dt} \\ \frac{dC_C}{dt} \end{pmatrix} = \begin{pmatrix} -k_1 & k_2 & 0 \\ k_1 & -(k_2+k_3) & k_4 \\ 0 & k_3 & -k_4 \end{pmatrix} \cdot \begin{pmatrix} C_A \\ C_B \\ C_C \end{pmatrix} \tag{5.22}$$

or using a compact notation

$$\frac{d}{dt}\mathbf{C} = \mathbf{K} \cdot \mathbf{C} \tag{5.23}$$

To obtain a solution for this matrix equation, let's recall the solution for a single differential equation of the type

$$\frac{d}{dt}y = ky \tag{5.24}$$

with the initial condition $y(t_0 = 0) = y_0$. A solution of this equation can be obtained by separating the variables and integrating both sides of the equation

$$\int_{t_0}^{t} \frac{dy}{y} = \int_{t_0}^{t} kdt \longrightarrow ln\frac{y}{y_0} = kt \longrightarrow y = e^{kt}y_0 \tag{5.25}$$

The matrix equation 5.23 can be integrated in a similar fashion to obtain its solution. This solution is then given by:

$$\mathbf{C} = e^{\mathbf{K}t}\mathbf{C}_0 \tag{5.26}$$

As shown in appendix one can demonstrate that this solution satisfy the differential equation 5.23. In this equation the $e^{\mathbf{K}t}$ is the exponential of a matrix. As demonstrated in appendix, such exponential is given by

$$e^{\mathbf{K}t} = \mathbf{U}e^{\mathbf{\Lambda}t}\mathbf{U}^{-1} \tag{5.27}$$

where $\mathbf{U}$ is the matrix of the *eigenvectors* of $\mathbf{K}$ and $\mathbf{\Lambda}$ diagonal matrix that contains the *eigenvalues* $(\lambda_1, \lambda_2, ..., \lambda_N)$ of $\mathbf{K}$. The exponential of such diagonal matrix is simply given by

$$e^{\mathbf{\Lambda}t} = \begin{pmatrix} e^{\lambda_1 t} & 0 & 0 & \dots & 0 \\ 0 & e^{\lambda_2 t} & 0 & \dots & 0 \\ 0 & 0 & e^{\lambda_2 t} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & e^{\lambda_N t} \end{pmatrix} \tag{5.28}$$

While it is possible to write a small piece of code that computes the exponential of a matrix via its full diagonalization, the scipy module contains a function scipy.linalg.expm() that allows computing such exponential in just one line. The snippet of code below compares these two methods and compare their performance.

```
import numpy as np
import scipy.linalg as scln
import time

# create a matrix
N = 500
A = 0.1*np.random.rand(N,N)
t = 0.01

# compute the exponential with scipy
t0 = time.clock()
expA_scipy = scln.expm(A)
t1 = time.clock()

# compute the exponential directly
t2 = time.clock()
Lambda, U = np.linalg.eig(A)
expA_direct = np.dot(U,np.dot(np.diag(np.exp(Lambda)),np.linalg.inv(U)))
t3 = time.clock()

# compute the error and print the performance
error = np.abs(np.sum(expA_scipy-expA_direct))/N/N

if error < 1E-3:
        print 'Scipy %1.6f sec. \t Direct %1.6f' %((t1-t0),(t3-t2))
else:
        print 'error %1.6e' %(error)
```

The solution 5.26 can be used to compute the evolution of the concentration of A, B and C in the rector. To do so we need the initial value of the each concentration, i.e. the vector $\mathbf{C}_0$. Let's assume that these concentration are given by

$$C_A(0) = 1 \qquad C_B(0) \qquad C_C(0) \tag{5.29}$$

We also assume that the rate constants are given by

$$k_1 = 1\ \text{min}^{-1} \qquad k_2 = 0\ \text{min}^{-1} \qquad k_3 = 2\ \text{min}^{-1} \qquad k_4 = 3\ \text{min}^{-1} \tag{5.30}$$

According to eq. 5.26, the concentrations at any time t_p are given by

$$\begin{pmatrix} C_A(t_p) \\ C_B(t_p) \\ C_C(t_p) \end{pmatrix} = \exp\left[\begin{pmatrix} -1 & 0 & 0 \\ 1 & -2 & 3 \\ 0 & 2 & -3 \end{pmatrix} t_p \right] \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \tag{5.31}$$

As in most numerical simulation, the total evolution time is divided in small intervals of same length (Δt): $T = [0, \Delta t, 2\Delta t, 3\Delta t, ..., N_{tot}\Delta_t]$, and we want to compute the solution of the equation for each time values. To avoid recalculating the matrix exponential at each time step, we can use a convenient property of matrices. Let's say we want the solution at $t_n = n\Delta t$. The solution at this time step can be written as

$$\mathbf{C}(t_n = n\Delta t) = e^{\mathbf{K}n\Delta t}\mathbf{C}_0 = \left(e^{\mathbf{K}\Delta t}\right)^n \mathbf{C}_0 \tag{5.32}$$

And hence only $e^{\mathbf{K}\Delta t}$ must be calculated. If we need to calculate all the solutions between $t_0 = 0\Delta t$ and $t_f = N_{tot}\Delta t$, as we often do, we can use the recurrence relationship

$$\mathbf{C}_{n+1} = e^{\mathbf{K}\Delta t}\mathbf{C}_n \tag{5.33}$$

where $\mathbf{C}_n \equiv \mathbf{C}(t_n = n\Delta t)$. The following snippet of code illustrate how to implement this in Python and the results of this code are illustrated in Fig. 5.1. As we can see on this graph the solution respect the conversation of mass principle with $C_A(t) + C_B(t) + C_C(t) = cte$. In addition the asymptotic solution respect the equilibrium condition $C_B/C_C = k_4/k_3$.

```
import numpy as np

# rate constants
k1, k2, k3, k4 = 1, 0, 2, 3

# initial condition
C0 = np.array([1,0,0])

# evolution time
tmax, nT = 5.0, 251
dT = tmax/(nT-1)


# matrix of the rates
K = np.array([[-1,0,0],[1,-2,3],[0,2,-3]])

# compute the exponential of the matrix
Lambda, U = np.linalg.eig(K)
eKdt = np.dot(U,np.dot(np.diag(np.exp(Lambda*dT)),np.linalg.inv(U)))
```

```
20
21 # loop over the different time steps
22 for i in range(1,nT):
23         C0 = np.dot(eKdt,np.array(C0))
```

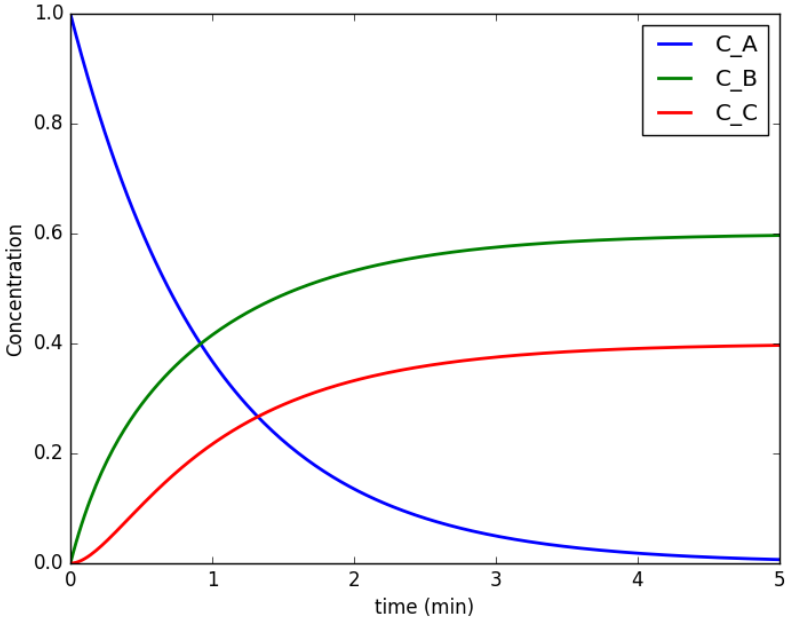


Figure 5.1: Concentration profiles of the a chemical reaction model by the equation 5.31.

5.3 Non-linear ODEs : initial value problem

It is not necessarily possible to write the sytem of ODE in a matrix form as the one in eq. 5.31. This is for example the case if the equations contain nonlinear terms. Many numerical methods have been developed to solve such non-linear equations. We suppose that the system of equation is in a canonical form

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(t, \mathbf{y}) \tag{5.34}$$

and that we know the initial conditions:

$$\mathbf{y}(t_0) = \mathbf{y}_0 \tag{5.35}$$

We first treat the case where only one equation defines the system to illustrate different approach. The case of multiple simultaneous nonlinear ODEs are treated in the next section. The numerical resolution of nonlinear ODE is usually done by dividing the independent variable (here t) in a series of intervals: $t = [t_0, t_1, ...t_i, ...t_N]$. The solution is then given as a series of values $y = [y_0, y_1, ...y_i, ...y_N]$. The solution of this equation can then be obtained by rearranging the terms in equation 5.34 and integrating on both sides

$$\int_{y_i}^{y_{i+1}} dy = \int_{t_i}^{t_{i+1}} f(t,y)dt \tag{5.36}$$

$$y_{i+1} - y_i = \int_{t_i}^{t_{i+1}} f(t,y)dt \tag{5.37}$$

$$y_{i+1} = y_i + \int_{t_i}^{t_{i+1}} f(t,y)dt \tag{5.38}$$

Assuming we know y_i, we only have to integrate the function $f(t,y)$ over the interval to obtain the value of y_{i+1}. The different approximation for this integrals presented in the previous chapter can then be used to solve ODEs.

5.3.1 The forward Euler method : rectangle approximation

As a reminder, the simplest approximation of an integral between two points, the so called rectangle approximation, reads:

$$\int_{t_i}^{t_{i+1}} f(t,y)dt = hf(t_i,y_i) \tag{5.39}$$

Introducing this expression in 5.38 leads to the forward Euler method for the solution of ODE

$$y_{i+1} = y_i + hf(t_i,y_i) \tag{5.40}$$

This equation simply means that in order to obtain y_{i+1} we compute $\frac{dy}{dt}$ at the i-th point (which is given by $f(y_i,t_i)$) and make a step of length h in the direction of the derivative. The accuracy of the Euler method is of the same order than the rectangle approximation for the integral, i.e. $O(h)$. The Euler equation is an *explicit* equation, meaning that y_{i+1} appears only on the left-hand side of the equation.

5.3.2 The Modified Euler Method: trapezoidal approximation

The trapezoidal approximation can also be used to calculate the integral in eq. 5.38. As a reminder the trapeze method reads

$$\int_{t_i}^{t_{i+1}} f(t,y)dt = \frac{h}{2}\left(f(t_i,y_i) + f(t_{i+1},y_{i+1})\right) \tag{5.41}$$

Introducing this approximation in eq. 5.38, we obtain the equation for the so-called modified Euler method

$$y_{i+1} = y_i + \frac{h}{2}\left(f(t_i,y_i) + f(t_{i+1},y_{i+1})\right) \tag{5.42}$$

This equation is *implicit* as y_{i+1} appears on both side of the equation. We can make this equation *explicit* by making a *prediction* for the term y_{i+1} that appears in the right hand side of eq 5.42 with the forward Euler approach given by eq. 5.40.

$$(y_{i+1})_{Pr} = y_i + hf(t_i,y_i) \tag{5.43}$$

We can the use this prediction to obtain a corrected value for y_{i+1}

$$(y_{i+1})_{Cor} = y_i + \frac{h}{2}\left(f(t_i,y_i) + f\left(t_{i+1},(y_{i+1})_{Pr}\right)\right) \tag{5.44}$$

5.3.3 The Runge-Kutta methods

The most widely used methods to solve ODEs are a series of methods called the *Runge-Kutta* second , third fourth plus a few other variants. The RK methods are an extension of the modified Euler method presented above. Let's rewrite Eq. 5.44 as

$$\begin{aligned} y_{i+1} &= y_i + \frac{h}{2} f(t_i, y_i) + \frac{h}{2} f(t_{i+1}, y_{i+1}) && (5.45)\\ &= y_i + w_1 k_1 + w_2 k_2 && (5.46) \end{aligned}$$

where in this case:

$$w_1 = \frac{1}{2} \qquad k_1 = hf(t_i, y_i) \tag{5.47}$$

$$w_2 = \frac{1}{2} \qquad k_2 = hf(t_i + h, y_i + k_1) \tag{5.48}$$

since y_{i+1} is approximated by $y_i + hf(t_i, y_i)$ in the prediction stage of the modified Euler method. The equation presented above define the second-order Runge-Kutta methods. This approach can be generalized to

$$y_{i+1} = y_i + \sum_{i=1}^{m} w_i k_i \tag{5.49}$$

Similarly to the second-order RK methods the k_i are defined by :

$$\begin{aligned} k_1 &= hf(t_i, y_i) && (5.50)\\ k_2 &= hf(t_i + c_2 h, y_i + a_{21} k_1) && (5.51)\\ k_3 &= hf(t_i + c_3 h, y_i + a_{31} k_1 + a_{32} k_2) && (5.52)\\ \dots & && (5.53)\\ k_m &= hf(t_i + c_m h, y_i + a_{m1} k_1 + a_{m2} k_2 + ... + a_{m,m-1} k_{m-1}) && (5.54) \end{aligned}$$

or in a more compact notation

$$k_p = hf(t_i + c_p h, y_i + \sum_{l=1}^{p-1} a_{pl} k_l) \tag{5.55}$$

The number of terms, m, in the expansion defined the degree of complexity of the methods. Each degree of complexity is entirely defined by the series of coefficients c_i and a_{ij}.

The derivation of these coefficients is difficult but has been performed for many orders. For example the *fourth-order* RK method is defined by the equations :

$$y_{i+1} = y_i + \frac{1}{6}(k_1 + 2k_2 + 2k_4 + k_4) \tag{5.56}$$

with :

$$\begin{aligned} k_1 &= hf(t_i, y_i) && (5.57)\\ k_2 &= hf\left(t_i + {}^{h}/_{2}, y_i + {}^{k_1}/_{2}\right) && (5.58)\\ k_3 &= hf\left(t_i + {}^{h}/_{2}, y_i + {}^{k_2}/_{2}\right) && (5.59)\\ k_4 &= hf\left(t_i + h, y_i + k_3\right) && (5.60) \end{aligned}$$

This method is maybe the most used technique to solve ODEs.

5.3.4 Numerical VS Exact Methods

To illustrate the performances of the forward and modified Euler methods, let's study the simplest chemical reaction possible

$$\mathrm{A} \xrightarrow{\mathrm{k}} \mathrm{B}$$

The differential equation that models the concentration of A and B are then given by:

$$\frac{d}{dt}C_A = -kC_A \qquad \frac{d}{dt}C_B = kC_A \tag{5.61}$$

Under the initial condition $C_A(0) = 1$ and $C_B(0) = 0$, the exact solutions of these ODEs can be readily derived

$$C_A(t) = e^{-kt} \qquad C_B(t) = (1 - e^{-kt}) \tag{5.62}$$

We can therefore assess the accuracy of the Euler methods by comparing their results to the analytic solution shown above. The snippet of code below, implement the different methods see above and compare their results with the exact solution

```
import numpy as np

# the euler method
def forward_euler(func, yi, ti, dt):
        return yi + dt*func(yi,ti)

# the modified euler method
def modified_euler(func, yi, ti, dt):
        y_pr = yi + dt*func(yi,ti)
        return yi + 0.5*dT*(func(yi,ti)+func(y_pr,ti+dt))

# 4th order RK
def RK4(func, yi, ti, dt):
        k1 = dt*func(yi,ti)
        k2 = dt*func(yi+0.5*k1,ti+0.5*dt)
        k3 = dt*func(yi+0.5*k2,ti+0.5*dt)
        k4 = dt*func(yi+k3,ti+dt)
        return yi + 1./6*( k1+2*k2+2*k3+k4 )

# variation of CA
def df(ca,t):
        return -k*ca
```

5.3.5 ODEs solver within *scipy*

Several methods have been implemented in the module *scipy* to solve ODEs. This can be done with the function *scipy.integrate.ode*(). The snippet of code presented below shows how to use this function.

```
import numpy as np
from scipy.integrate import ode

# variation of CA
```

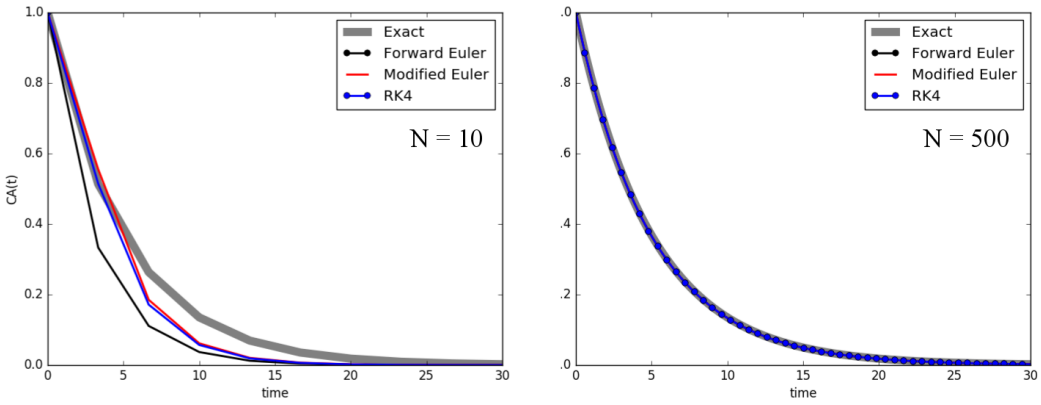


Figure 5.2: Comparison of the numerical and exact solution for the concentration of A during the simple chemical reaction $A \rightarrow B$. The solution were obtained for 2 different number of points along the time axis $N = 10$ and $N = 500$.

```
5  def df(t,y,k):
6          return -k*y # return the value
7
8  # time argument
9  tmax,nT = 30.0,500
10 T = np.linspace(0,tmax,nT)
11 dT = tmax/(nT-1)
12
13 # rate constant
14 k = 0.2
15
16 # initial condition
17 C0, T0 = 1.0, 0.0
18
19 # storage
20 C = np.zeros(nT)
21 C[0]  = C0
22
23 # initialize the solver
24 r = ode(df).set_integrator('dopri5')
25 r.set_initial_value(C0, T0).set_f_params(k)
26
27 # loop over time
28 iT = 1
29 while r.successful() and r.t<tmax:
30         C[iT] = r.integrate(r.t+dT)
31         iT += 1
```

We used here the method referred to as 'dopri5' which is a Runge-Kutta method. As you can read in the documentation of the *scipy.integrate.ode* function other methods are also available.

5.3.6 Simultaneous differential equations

As mentioned at the beginning of the section the methods developed here can be applied to the case of simultaneous ODEs. To illustrate this let's consider the following set of n ODEs

$$\frac{dy_1}{dt} = f_1(t, y_1, y_2, ...y_n) \tag{5.63}$$
$$\frac{dy_2}{dt} = f_2(t, y_1, y_2, ...y_n) \tag{5.64}$$
$$\dots \tag{5.65}$$
$$\frac{dy_n}{dt} = f_n(t, y_1, y_2, ...y_n) \tag{5.66}$$

that can be written in a vector form as $\frac{d}{dt}\mathbf{Y} = \mathbf{F}(t, \mathbf{Y})$. Suppose we want to apply the forward Euler method to solve this system starting from the initial condition $\mathbf{Y}_0$ that contains all the initial values of the y_i at t=0. We can directly use eq. 5.40 and write

$$\mathbf{Y}_{i+1} = \mathbf{Y}_i + h\mathbf{F}(t, \mathbf{Y}_i) \tag{5.67}$$

Any other methods presented above for a single ODE can be used to solve a set of simultaneous ODEs.

5.3.7 The predator-prey model

To illustrate the resolution of simultaneous ODEs, let's consider one of the most famous initial value problem: the *predator-prey* problem also called the Lotka-Volterra equations. In this problem the population of two species, a predator (A) and a prey (B), are studied. The population change of predator is given by

$$\frac{dA}{dt} = \alpha AB - \beta A \tag{5.68}$$

where α is the growth rate due to prey consumption and β the death rate by overpopulation. The population change of preys is given by

$$\frac{dB}{dt} = \gamma B - \mu AB \tag{5.69}$$

where γ is the birthrate and μ the death rate due to the predator. We can use the methods presented above to study the dynamics of these populations. The snippet of code shows how to do that using the different methods we've seen in this section to solve the predator-prey problem. The illustration of the populations are shown in Fig. 5.3.

```
import numpy as np
from scipy.integrate import ode

# 4th order RK
def RK4(func, yi, ti, dt):
        k1 = dt*func(ti,yi)
        k2 = dt*func(ti+0.5*dt,yi+0.5*k1)
        k3 = dt*func(ti+0.5*dt,yi+0.5*k2)
        k4 = dt*func(ti+dt,yi+k3)
        return yi + 1./6*( k1+2*k2+2*k3+k4 )

# population change
def df(t,Y):
```

```
        Yp = np.zeros(2)
        Yp[0] = alpha*Y[0]*Y[1] - beta*Y[0]
        Yp[1] = gamma*Y[1] - mu*Y[0]*Y[1]
        return Yp

# time
tmax, nT = 10.0,500
T = np.linspace(0,tmax,nT)
dT = T[1]-T[0]
# parameters
alpha, beta, gamma, mu = 1.5,2.0,2.0,1.0
# initial population
A0, B0 = 2.0, 1.0
T0 = 0.0

# storage
POP = np.zeros((nT,2))
POP[0,:] = [A0,B0]

# compute the population dynamics
for iT in range(1,nT):
        POP[iT,:] = RK4(df, POP[iT-1,:], T[iT-1], dT)

# initialize the solver
r = ode(df).set_integrator('vode')
r.set_initial_value(np.array([A0,B0]), T0)
POP_SCIPY = np.zeros((nT,2))
# loop over time
iT = 0
while r.successful() and r.t<tmax:
        POP_SCIPY[iT,:] = r.y
        r.integrate(r.t+dT)
        iT += 1
```

5.4 Non-linear ODEs : boundary value problem

As initial value problems, boundary problem are described by a set of simultaneous ODEs. However in contrast with initial value problem, the values variables are here given for different values of the independent variable. A boundary value problem of two unknown can therefore be written as

$$\begin{aligned} \frac{d}{dx}y_1 &= f_1(x, y_1, y_2) \\ \frac{d}{dx}y_2 &= f_2(x, y_1, y_2) \end{aligned} \tag{5.70}$$

for $x \in [x_0, x_f]$ The boundary conditions are then given by

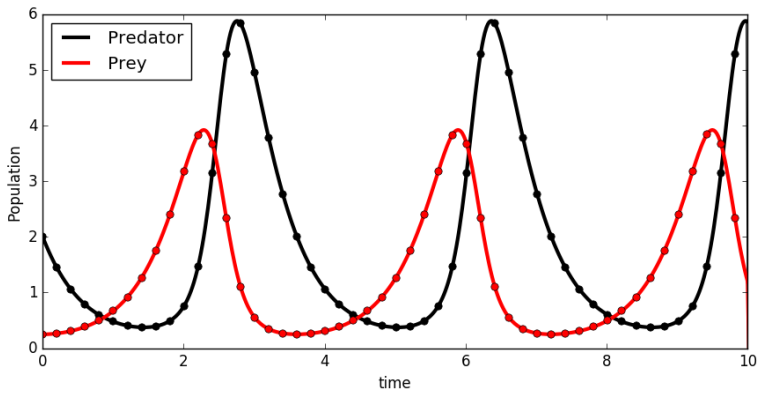


Figure 5.3: Solution of the predator prey model. The population of both species present a cyclic evolution: if there is two many predators they starve until the population of prey is restored and so on.

$$\begin{aligned} y_1(x = x_0) &= y_{1,0} \\ y_2(x = x_f) &= y_{2,f} \end{aligned} \tag{5.71}$$

Boundary values problem occurs in many branches of chemical engineering. Many methods of different degrees of complexity have been developed to solve such problem. In this section we will consider two: the shooting method and the finite difference methods.

5.4.1 The shooting method

The shooting method basically treats a boundary value problem as an initial value problem to take advantage of the efficient methods that have been developed to solve them. Do to so, the shooting methods propose to make a guess for the missing initial conditions, i.e. the values of the variables that are not know for $x = x_0$, solve the system under these guessed conditions and then optimize the values of the guessed condition until the boundary conditions are satisfied.

Let's consider the boundary value problem eq. 5.70 with the condition 5.71. To solve this problem with the shooting method we first guess that

$$y_2(x_0) = \gamma \tag{5.72}$$

We can then solve the equations 5.70 as an initial value problem with the initial condition $y_1(x_0) = y_{1,0}$ and $y_2(x_0) = \gamma$. This can be done using any methods presented in the previous section. However since the value of $y_2(x_0)$ was a guess it misses the target at $x = x_f$ and the solution does not satisfy the boundary condition $y_2(x = x_f) = y_{2,f}$. We therefore have to optimize our guess until we obtain the correct boundary condition for $y_2(x_f)$. We can define a cost function

$$\Phi(\gamma) = y_2(x_f, \gamma) - y_{2,f} \tag{5.73}$$

that quantifies how far the solution obtained with the guessed value of $y_2(x_0)$ is from the boundary condition. The cost function can be expressed as a Taylor serie

$$\Phi(\gamma + \Delta\gamma) = \Phi(\gamma) + \frac{\partial \Phi}{\partial \gamma}\Delta\gamma + O(\Delta\gamma^2) \tag{5.74}$$

When the value of $y_2(x_f, \gamma)$ is equal to the boundary condition we have $\Phi(\gamma + \Delta\gamma) = 0$ and therefore after truncating the high order terms

$$0 = \Phi(\gamma) + \frac{\partial \Phi}{\partial \gamma}\Delta\gamma \quad \longrightarrow \quad \Delta\gamma = \frac{-\Phi(\gamma)}{\partial\Phi/\partial\gamma} \tag{5.75}$$

Since $\partial\Phi/\partial\gamma = \partial y_2(x_f,\gamma)/\partial\gamma$ we finally obtain

$$\Delta\gamma = -\Big(y_2(x_f,\gamma) - y_{2,f}\Big)\left(\frac{\partial y_2(x_f,\gamma)}{\partial\gamma}\right)^{-1} \tag{5.76}$$

The value $\Delta\gamma$ is a correction to the previous values of γ used to solve the system. We can therefore obtain a new value of γ from the old ones until convergence is reached. This leads to a series of values for γ, noted γ_n and obtained via the recurrence relationship

$$\gamma_{n+1} = \gamma_n + \rho\Delta\gamma \qquad 0 < \rho \leq 1 \tag{5.77}$$

where the factor ρ is introduced to facilitate the convergence. We can keep updating the value of γ until $|\Delta\gamma| < \epsilon$. Note that the partial derivative in equation 5.76 can be approximated by its numerical value

$$\frac{\partial y_2(x_f,\gamma)}{\partial\gamma} = \frac{y_2(x_f,\gamma_{n+1}) - y_2(x_f,\gamma_n)}{\gamma_{n+1} - \gamma_n} \tag{5.78}$$

As we will see in the example below this require to make a guess not only for γ_0 but also for γ_1.

5.4.2 Application of the shooting method

It is easier to understand the shooting method with a simple problem. We want to solve the boundary value problem

$$\frac{d^2y}{dx^2} = 4(y - x) \tag{5.79}$$

in the domain $x \in [0,1]$ and with the boundary condition $y(0) = 0$ and $y(1) = 2$. The first thing we have to do is to write this system in its canonical form with the substitution $y_1 = y$, $y_2 = {}^{dy_1}/_{dx}$, which leads to:

$$\frac{dy_1}{dx} = y_2 \tag{5.80}$$

$$\frac{dy_2}{dx} = 4(y_1 - x) \tag{5.81}$$

The initial condition is available for the first equation ($y_1(0) = 0$) but not for the second one. We also have the boundary condition $y_1(1) = 2$ that the system must respect. Let's therefore make a guess and say: $y_2(0) = \gamma_0 = 1.0$. We can now solve the system of equation above as an initial value problem (for example with *scipy.integrate.ode*) and check if the boundary condition $y_1(1) = 2$ is respected or not. As seen on figure 5.4, or initial guess misses the target and hit $y_1(1, \gamma_0) = 1$ instead.

We therefore have to improve our guess. However at this stage we only know 1 value γ_0 and therefore we cannot use eq. 5.78 to evaluate the partial derivative in eq. 5.76. We therefore make a second guess and pose $y_2(0) = \gamma_1 = 0.0$. As shown in Fig. 5.4 the solution obtained with this initial condition misses the target even more and leads to $y_1(1, \gamma_1) = -0.813$. It's not going well. However

we do have now a robust method to estimate how to update our value of γ. Indeed we can use eq. 5.76 and write

$$\Delta\gamma = -\left(y_1(x_f,\gamma_1) - y_{1,f}\right)\left(\frac{y_1(x_f,\gamma_1) - y_1(x_f,\gamma_0)}{\gamma_1 - \gamma_0}\right)^{-1} \tag{5.82}$$

and obtain a new value for γ as $\gamma_2 = \gamma_1 + \Delta\gamma$. We have set here $\rho = 1$ for simplicity. The procedure can then be repeated until we obtain $y_1(1,\gamma_n) = 2 \pm \epsilon$ or equivalently $|\Delta\gamma| < \epsilon$. The solution is illustrated in Fig. 5.4 we see that after only 3 iterations the shooting method converges toward the correct boundary condition. This is the case for any linear boundary value problem. For non-linear problem few more iterations are generally required.

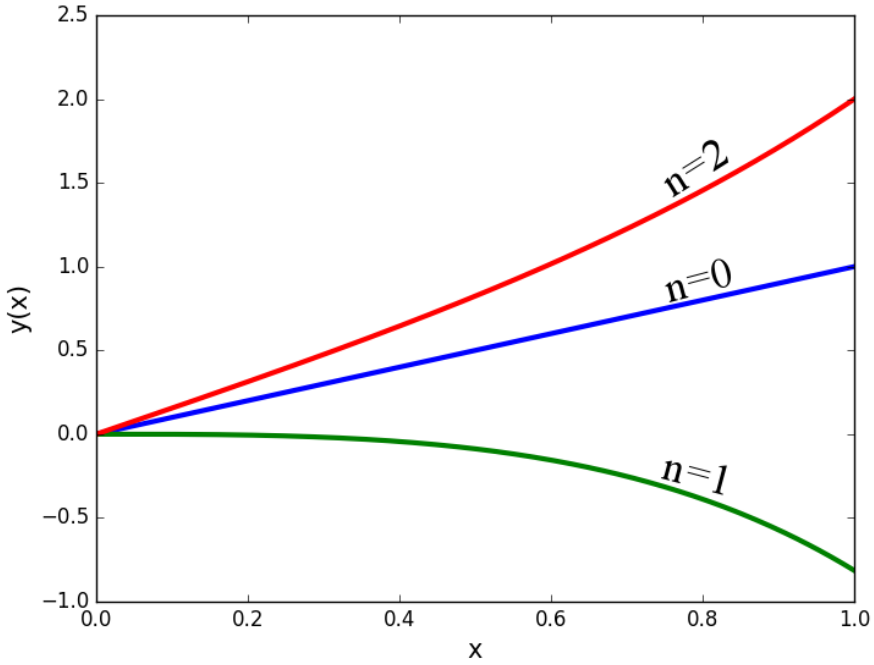


Figure 5.4: Resolution of the boundary value problem (5.70) with the shooting method. Only 3 iterations are necessary

5.4.3 Generalization to N simultaneous equations

It is possible to generalize the shooting method presented here to the case of N simultaneous equations

$$\frac{dy_i}{dx} = f(x, y_1, y_2, ...y_N) \qquad x \in [x, x_f] \tag{5.83}$$

The boundary conditions are split between initial condition for r equations and at least $N - r$ final conditions.

$$y_j(x_0) = y_{j,0} \qquad j = 1, 2, ...r \tag{5.84}$$
$$y_j(x_f) = y_{j,f} \qquad j = r+1, ...N \tag{5.85}$$

Following the principle behind the shooting method we're going to make an initial guess for the missing initial conditions

$$y_j(x_0) = \gamma_j^{(0)} \qquad j = r+1, ...N \tag{5.86}$$

The system of equation 5.83 can then be solved as an initial condition problem with any method available to do that. This hence provide some values for the $y_j(x_f)$ with $j = r+1, ...N$ that most likely do not respect the boundary conditions (5.85). To improve our guess we therefore compute the derivative of these incorrect solution with respect of our guess. This is performed by the calculation of the *Jacobian* matrix

$$\mathcal{J}(x_f, \boldsymbol{\gamma}) = \begin{pmatrix} \frac{\partial y_{r+1}}{\partial \gamma_{r+1}} & \frac{\partial y_{r+1}}{\partial \gamma_{r+2}} & \cdots & \frac{\partial y_{r+1}}{\partial \gamma_N} \\ \frac{\partial y_{r+2}}{\partial \gamma_{r+1}} & \frac{\partial y_{r+2}}{\partial \gamma_{r+2}} & \cdots & \frac{\partial y_{r+2}}{\partial \gamma_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial y_N}{\partial \gamma_{r+1}} & \frac{\partial y_N}{\partial \gamma_{r+2}} & \cdots & \frac{\partial y_N}{\partial \gamma_N} \end{pmatrix} \tag{5.87}$$

We can then compute the correction to our value for $\boldsymbol{\gamma}$ with

$$\Delta\boldsymbol{\gamma} = \left[\mathcal{J}(x_f, \boldsymbol{\gamma})\right]^{-1} \cdot \begin{pmatrix} y_{r+1}(x_f, \gamma_{r+1}) \\ y_{r+1}(x_f, \gamma_{r+2}) \\ \cdots \\ y_N(x_f, \gamma_N) \end{pmatrix} \tag{5.88}$$

and finally update the values of γ

$$\boldsymbol{\gamma}^{(n+1)} = \boldsymbol{\gamma}^{(n)} + \rho\Delta\boldsymbol{\gamma} \tag{5.89}$$

5.4.4 The finite difference method

We present here another technique to solve BVP using finite difference (FD). As a reminder a BVP is for example given by the differential equation

$$\frac{d^2y}{dx^2} + \alpha(x)\frac{dy}{dx} + y\beta(x) = f(x) \tag{5.90}$$

on the domain $a \leq x \leq b$ and with boundary condition given at a an b, i.e. $y(a) = y_a$ and $y(b) = y_b$. The FD method is based on the idea of replacing the derivative in this equation by algebraic approximations at different points on the x axis. The FD methods starts by dividing the x-axis in a series of evenly spaced points, called node and marked as x_i, with the interval between two points being given by $x_{i+1} - x_i = h$. If we have $N+1$ nodes numbered from 0 to N we have N intervals. In the $[a, b]$ interval the location of each node is given by $x_i = a + ih$.

As we have seen in Chapter 4, the first and second derivatives in eq. 5.90 can be approximated by the expressions

$$\frac{dy}{dx} = \frac{y_{i+1} - y_{i-1}}{2h} \tag{5.91}$$

$$\frac{d^2y}{dx^2} = \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} \tag{5.92}$$

Using these approximations we can rewrite eq 5.90 as

$$(y_{i+1} - 2y_i + y_{i-1}) + \frac{h}{2}\alpha(x_i)(y_{i+1} - y_{i-1}) + h^2\beta(x_i)y_i = h^2 f(x_i) \tag{5.93}$$

By writing this equation for each node and accounting for the boundary conditions we obtain a system of linear equations given by:

$$y_0 = y_a \tag{5.94}$$
$$y_{i+1}(1 + \frac{h}{2}\alpha(x_i)) + y_i(h^2\beta(x_i) - 2) + y_{i-1}(1 - \frac{h}{2}\alpha(x_i)) = h^2 f(x_i) \tag{5.95}$$
$$y_N = y_b \tag{5.96}$$

This linear system of equations can be written is a matrix form

$$\begin{pmatrix} a_{00} & a_{01} & \dots & a_{0N} \\ a_{10} & a_{11} & \dots & a_{1N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N0} & a_{N1} & \dots & a_{NN} \end{pmatrix} \cdot \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_N \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_N \end{pmatrix} \tag{5.97}$$

or more compactly

$$\mathbf{A} \cdot \mathbf{Y} = \mathbf{B} \tag{5.98}$$

The solution of this matrix equation (for example with *np.linalg.solve()*) gives all the values of the y along the x-axis with the appropriate boundary conditions. The FD method therefore transform a BVP of ODE into a linear system of equations easily solvable.

As an example, the snippet of code below uses the FD method to solve the BVP

$$\frac{d^2y}{dx^2} + y = 0 \tag{5.99}$$

on the domain $0 \leq x \leq \pi/2$ with the Dirichlet boundary condition $y(0) = 1.0$ and $y(\pi/2) = 1.0$. The results of the program is plotted in Fig. 5.5.

```
import numpy as np
import matplotlib.pyplot as plt
from numpy.linalg import solve

# Number of nodes
N = 10
X = np.linspace(0,np.pi/2.,N+1)
h = X[1]-X[0]

# define the matrices
A = np.zeros((N+1,N+1))
b = np.zeros(N+1)

# boundary condition at x=0
A[0,0] = 1.0
b[0] = 1.0

# boundary condition at x = pi/2
A[-1,-1] = 1.0
b[-1] = 1.0

```

```
# all the other nodes
for i in range(1,N):
        A[i,i-1] = 1.0
        A[i,i] = -2.0 + h**2
        A[i,i+1] = 1.0

Y = solve(A,b)

plt.plot(X,Y,marker='o',color='black')
plt.plot(X,np.cos(X)+np.sin(X),linewidth=2,color='blue')
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```

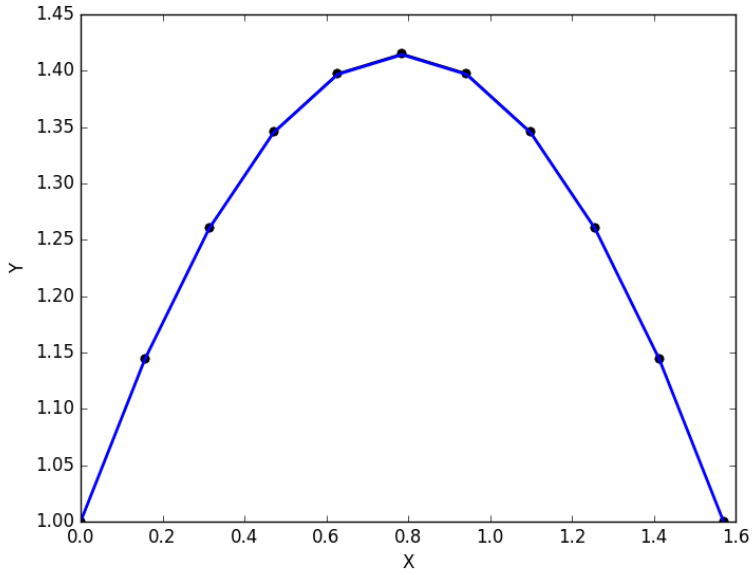


Figure 5.5: Example of numerical solution of a BVP using the FD method. The figure shows the numerical solution of $\frac{d^2y}{dx^2} + y = 0$ with the boundary conditions $x(0) = 1.0$ and $x(\pi/2) = 1.0$. (dots) The plain line shows the analytical solution of this equation

5.5 Exercise : Initial value problems

5.5.1 The Forward Euler method

In this first exercise you will implement the forward Euler method for the solution of ODEs. As a reminder, the forward euler method allows finding the solution of the differential equation

$$\frac{dy}{dt} = f(t, y) \tag{5.100}$$

with the initial solution $y(t_0) = y_0$ by the recurrence

$$y_{i+1} = y_i + hf(t_i, y_i) \tag{5.101}$$

To implement this approach you will write a function, for example called `forwardEuler()` that returns the value of y_{i+1} from the value of y_i. You will then call this function in a for loop running over all the time steps desired. To test your program you will compute the solution of the simple reaction

$$A \xrightarrow{k} B \tag{5.102}$$

with the initial condition $C_A(0) = 1$ and $C_B(0) = 0$ (where $C_X(t)$ is the concentration of X at time t) and a value of $k = 0.2\ \text{min}^{-1}$. You will simulate the dynamics of this reaction over a period of 30 min.

The differential equations for the evolution of C_A and C_B can be easily derived and solved analytically. You can therefore test your program by comparing the solution it provides with the exact solution that you will have derived analytically. Vary the number of time step you use in the simulations of the reaction with the forward Euler method. Use between 10 and 500 time steps. Comment on the accuracy of the method.

5.5.2 The Runge-Kutta 4-th order method

In this follow up exercise you will update your program by implementing the RK4 method. The equation defining the RK-4 methods are given by

$$y_{i+1} = y_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{5.103}$$

with :

$$\begin{aligned} k_1 &= hf(t_i, y_i) && (5.104)\\ k_2 &= hf\left(t_i + {}^{h}/_{2}, y_i + {}^{k_1}/_{2}\right) && (5.105)\\ k_3 &= hf\left(t_i + {}^{h}/_{2}, y_i + {}^{k_2}/_{2}\right) && (5.106)\\ k_4 &= hf\left(t_i + h, y_i + k_3\right) && (5.107) \end{aligned}$$

As previously you will write a function that returns the value of y_{i+1} from y_i and uses this function in a for loop to obtain the solution of the ODEs. You will once again use your program to compute the concentrations $C_A(t)$ and $C_B(t)$ of the reactions $A \xrightarrow{k} B$ under the initial condition $C_A(0) = 1$ and $C_B(0) = 0$. Vary the number of time step you use in the simulations of the reaction with the forward Euler method. Use between 10 and 500 time steps. Comment on the accuracy of the method.

5.5.3 Epidemic propagation

You've been hired by the World Health Organization to estimate the impact of a disease outbreak on a city of 50.000 people. The WHO has developed a model, called the SIR model to simulate the propagation of the infection. Let's S be the number of healthy people that can contract the virus, (S stands for susceptible here) I is the number of infected people and R the number of people who have recovered from the infection and are immune to it. The evolution of the population on S, I and R is then given by

$$\frac{dS}{dt} = -\gamma \cdot S \cdot I \tag{5.108}$$
$$\frac{dI}{dt} = \gamma \cdot S \cdot I - \alpha \cdot I \tag{5.109}$$
$$\frac{dR}{dt} = \alpha \cdot I \tag{5.110}$$

where γ is the transmission rate between sick and healthy people and α the recovery rate. For the present case the initial conditions are $S(0) = 50000$, $I(0) = 2$ and $R(0) = 0$. The transmission rate (γ) has been estimated at 10^{-4} day^{-1} while the recovery rate is 0.5 day^{-1}. Simulate the dynamics of the three different populations over a period of 30 days. Report the maximum number of infected people that can be expected and when this will happen. To limit the effect of the outbreak, the WHO has proposed to quarantine the population. They estimate that it will decreases the transmission rate to 0.5 10^{-4} day^{-1} but will slow down recovery to $\alpha = 0.1$ day^{-1}. Before implementing their plan they asked you to forecast the impact of the quarantine on the maximum number of infected people. Comment on the efficiency of this this quarantine ?

HINT : To solve the equation you can either use the routines you've written in the last exercise or use the scipy.integrate module.

5.6 Exercise : Boundary value problem

5.6.1 The shooting method

In this first exercise you will implement the shooting method for the solution of boundary value problems as described in section 5.4.1. As a reminder, the shooting methods treats a boundary value problem as an initial value problem. To this end we therefore make a guess of the initial values that are not provided, solve the equations and compare the solution obtained with the boundary conditions provided. This allows us to make a better guess for the missing initial condition. We can therefore converge toward the correct solution by iterating the process. A detailed description of the method can be found in section 5.4.1 and 5.4.2.

To test your implementation you will compute the solution of the BVP given by eq. 5.79, i.e.

$$\frac{d^2y}{dx^2} = 4(y - x) \tag{5.111}$$

for $x \in [0, 1]$ and with the BVC $y(0) = 0$ and $y(1) = 2$. Since this system contains a second derivative we need to transform it first to a canonical form leading to

$$\frac{dy_1}{dx} = y_2 \tag{5.112}$$
$$\frac{dy_2}{dx} = 4(y_1 - x) \tag{5.113}$$

We now know the initial value for the first equation ($y_1(0) = 0$) but not for the second. We therefore have to guess it and improve our guess following the procedure introduced in section 5.4.1. As shown in section 5.4.2, the shooting method should converge rapidly toward the solution as represented in Fig. 5.4.

HINT : To help you in the implementation we provide below a possible pseudo code for the shooting method

```
from scipy.integrate import odeint

# function that retrurns
# the systems of equation
def df(Y,x):
        dy1 = ...
        dy2 = ...
        return np.array([dy1,dy2])

....

# boundary condition
y1_0,y1_f = 0.0, 2.0

# guesses for gamma
# remember we need two
gamma0, gamma1 = 1.0, 0.0
....

# solve the system for the first guess
Y = odeint(....)
...

# error obtained with the first guess
error = ....

# loop until convergence
while error >  EPS and niter < MAXITER:

        # solve for the new value of gamma
        Y = odeint(....)
        ...

        # compute the error
        error = ...

        # compute Delta gamma
        dG = ....

        # update the gammaXs
        gamma0, gamma1 = gamma1, dG
```

5.6.2 Drug diffusion

You've been hired by a pharmaceutical company to analyze the potential of a new drug for treating vitreous hemorrhage in the eye. The drug is delivered by an eye drop and then diffuses to the retina where it is immediately taken away by the blood flow. The diffusion and decay of the drug

concentration (c) are described by the equation

$$\frac{d^2c}{dx^2} - k \cdot c = 0 \tag{5.114}$$

The boundary condition are $c = 1.0$ at $x = 0.0$ cm and $c = 0.0$ at $x = 2.0$ cm. The independent variable x represent here the distance between the cornea and the retina. The decay constant of the current version of the drug is $k = 20$ cm^{-2}. Solve this boundary problem using the program that you've implemented in the last exercise. Determine the concentration at the hemorrhage site that is located at $x = 1.0$ cm. Is this concentration superior to 0.2 that is the minimum effective concentration ? The company would also like you to estimate the optimal decay rate k that yield to a concentration of $c = 0.2$ at $x = 1.0$ cm

5.7 Exercise Bonus

5.7.1 Transport problem in 1D with finite difference

We consider a Newtonian fluid undergoing laminar pressure-driven flow between two parallel and infinite plates. The plates are separated by a distance B. The bottom plate is fixed while the top plate moves in the y direction at a constant speed V_{up}. This system is represented in Fig. 5.6a. We know the value of the constant dynamic pressure gradient $\frac{\Delta P}{\Delta x}$ and we want to compute the velocity profile $v(y)$. If we suppose that the transport occurs only in the x direction the Navier-Stokes equation reduces to

$$\mu \frac{d^2v}{dy^2} = \frac{\Delta P}{\Delta x} \equiv G \tag{5.115}$$

with the boundary conditions $v(y = 0) = 0$ and $v(y = B) = V_{up}$. The goal of this exercise is to compute the profile $v(y)$ using the expression for the numerical evaluation of the second derivative and the techniques used to solve linear systems of equations. Using the finite difference approach we can rewrite the eq. 5.115 as

$$\mu \frac{v_{i+1} - 2v_i + v_{i-1}}{h^2} = G \tag{5.116}$$

You can therefore rewrite the problem as a system of linear equations

$$\begin{pmatrix} a_{00} & a_{02} & \dots & a_{0N} \\ a_{10} & a_{12} & \dots & a_{1N} \\ \vdots & & \ddots & \vdots \\ a_{N0} & a_{N2} & \dots & a_{NN} \end{pmatrix} \cdot \begin{pmatrix} v_0 \\ v_1 \\ \vdots \\ v_N \end{pmatrix} = \begin{pmatrix} s_0 \\ s_1 \\ \vdots \\ s_N \end{pmatrix} \tag{5.117}$$

In this equation v_i is the velocity profile at the i-th point. You must determine the expression for the a_{ii} and s_i. Once you have established this linear systems, use one of the techniques presented in the chapter 3 to solve it and obtain the values of the v_i. Compare your numerical results to the analytic expression given by

$$v(y) = V_{up}\frac{y}{B} + \frac{1}{2\mu}\frac{\Delta P}{\Delta x}(y^2 - yB) \tag{5.118}$$

During the calculation you will take : $B = 5 \cdot 10^{-3}$ m, $V_{up} = (1/6) \cdot 10^{-4}$ m/s, $\mu = 10^{-3}$. You will vary the dynamic pressure gradient from 0.0 to -0.06 Pa/m (take 5 points linearly spaced between these two values). Take about 50 to 100 points along the y axis. You will then compare your numerical results to the analytic solution. You should find variations of the velocity profile similar to the ones represented in Fig. 5.6b.

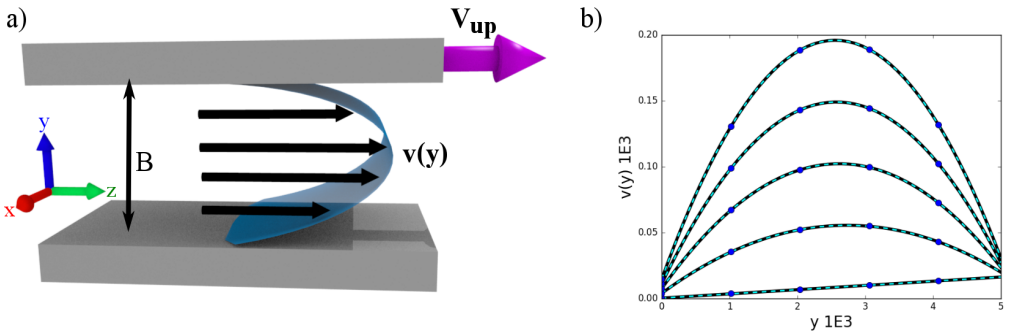


Figure 5.6: Laminar flow between two parallel plate. a) Representation of the problem the top plate is moving at a speed V_{up} while the bottom one is fixed. This creates a flow of the liquid present between the two plates with a well defined profile $v(y)$. b) Velocity profile of the liquid obtained via the exact solution (black line) or a numerical approach (blue line and dots)

5.8 Appendix

5.8.1 Demonstration of eq. 5.26

The exponential of a matrix is given by

$$e^{\mathbf{K}t} = \mathbf{I} + \mathbf{K}t + \frac{\mathbf{K}^2t^2}{2!} + \frac{\mathbf{K}^3t^3}{3!} + ... \tag{5.119}$$

We can use this expression to demonstrate that eq. 5.26 is a valid solution of eq. 5.23

$$\begin{aligned}
\frac{d}{dt}\mathbf{C} &= \frac{d}{dt}e^{\mathbf{K}t}\mathbf{C}_0 && (5.120)\\
&= \frac{d}{dt}\left(\mathbf{I} + \mathbf{K}t + \frac{\mathbf{K}^2t^2}{2!} + \frac{\mathbf{K}^3t^3}{3!} + ...\right)\mathbf{C}_0 && (5.121)\\
&= \left(\mathbf{K} + \mathbf{K}^2t + \frac{\mathbf{K}^3t^2}{2!} + ...\right)\mathbf{C}_0 && (5.122)\\
&= \mathbf{K}\left(\mathbf{I} + \mathbf{K}t + \frac{\mathbf{K}^2t^2}{2!} + ...\right)\mathbf{C}_0 && (5.123)\\
&= \mathbf{K}e^{\mathbf{K}t}\mathbf{C}_0 = \mathbf{K}\mathbf{C} && (5.124)
\end{aligned}$$

5.8.2 Exponential of a matrix

We have seen in chapter III that a nonsingular matrix of order n has n eigenvalues and n eigenvectors. Hence such matrix respect

$$A = U\Lambda U^{-1} \tag{5.125}$$

where the i-th column of U contains the i-th eigenvector and where Λ is a diagonal matrix containing the eigenvalues of A. The characteristic equations $AX = X\Lambda$ gives by right-multiplying by U^{-1}

$$AXX^{-1} = A = U\Lambda U^{-1} \tag{5.126}$$

Any power of the matrix A^n can then be expressed as

$$A^n = [U\Lambda U^{-1}]^n = U\Lambda^n U^{-1} \tag{5.127}$$

The exponential of the matrix then reads

$$\begin{aligned} e^A &= I + U\Lambda U^{-1} + \frac{1}{2!}U\Lambda^2 U^{-1} + ... && (5.128)\\ &= U\left[I + \Lambda + \frac{1}{2!}\Lambda^2 + ...\right]U^{-1} && (5.129)\\ &= Ue^{\Lambda}U^{-1} && (5.130) \end{aligned}$$

Chapter 6
Partial Differential Equations
The Finite Difference Method

In this chapter we study how to solve partial differential equation (PDEs) using the finite difference method (FD). We have already introduced the FD method in the last chapter where we used it to solve BVPs. We extend here this method to more complicated problems where we seek to compute the distribution in space and variation in time of physical quantities such as heat, velocity profiles, waves-packet .... that are defined by PDEs.

6.1 Introduction to PDEs

The solution of partial differential equation (PDE) are crucial in many fields of chemical engineering. As their name indicates PDE are equations that involve partial differential of a function that depends on several variables. One of the most studies PDE, the Laplace equation reads

$$\frac{\partial^2 \mathbf{c}(x,y)}{\partial x^2} + \frac{\partial^2 \mathbf{c}(x,y)}{\partial y^2} = f(x,y) \tag{6.1}$$

Solving this equation requires the determination of a function $\mathbf{u}(x,y)$ such that its derivative w.r.t to x plus its derivative w.r.t y is equal a given function $f(x,y)$ for each possible value of x, y. The Laplace equation descries for example the diffusion through solid and biological tissue. Other famous PDEs include the heat equation

$$\alpha \frac{\partial^2 \mathbf{T}}{\partial x^2} = \frac{\partial \mathbf{T}}{\partial t} \tag{6.2}$$

and the wave equation

$$\alpha \frac{\partial^2 \mathbf{w}}{\partial x^2} = \frac{\partial^2 \mathbf{w}}{\partial t^2} \tag{6.3}$$

As for the ODE, the PDE can be classified according to the order (first, second, ... derivative) and linearity. Second-order linear PDE of two variables are further classified into elliptic, parabolic and hyperbolic canonical forms depending on the coefficients in front of the derivatives. We will not enter the details of the zoology of PDEs.

Different initial and boundary conditions can be used for the solution of PDEs. To illustrate these conditions we will use the one-dimensional unsteady-state heat conduction equation 6.2. This equation describes how the temperature varies in time along a solid slab (e.g. the wall of a furnace) where the

heat transfer occurs in the x direction. If we specify the the value of **T** at $t = 0$ and for fixed values of x we have the so-called **Dirichlet** conditions.

$$\text{Dirichlet} \qquad T = f(x) \qquad \text{at} t = 0 \text{ and } 0 < x < 1 \tag{6.4}$$

On the other hand if we specify the value of the derivative of **T** as a function of x we have **Neumann** conditions

$$\text{Neumann} \qquad \frac{\partial T}{\partial x} = 0 \qquad \text{at} x = 0 \text{ and } t \leq 0 \tag{6.5}$$

A problem that combines Dirichlet and Neumann conditions is called a Cauchy problem. If the value of **T** is know for all boundaries (i.e. $t = 0$, $t = \infty$, $x = 0$ and $x = x_f$) then the problem is called a boundary problem. If any of these conditions is missing the problem is called an initial problem.

6.2 Finite difference method for PDE

We have seen in Chapter 4 numerical expressions for the partial derivatives of multivariable functions. In this section we will used these expressions to derive simple linear systems able to solve PDEs following the finite difference method. To illustrate the method let's consider the Laplace equation

$$\frac{\partial^2 c}{\partial x^2} + \frac{\partial^2 c}{\partial y^2} = 0 \tag{6.6}$$

were we have adopted the simplified notation $\mathbf{c}(x, y) = c$. We want to solve this equation for $0 \leq x \leq 1$ and $0 \leq y \leq 1$ with the following boundary conditions $c = 0$ on all boundaries except for $x = 0$ where $c = \sin(y)$. To solve this PDE we define a mesh of point that spans the x, y space. This mesh leads to a series of *nodes* (i, j) regularly spaced forming a square lattice as represented in Fig. XX. We can use this grid to express the partial derivatives with their centered finite difference expression. This yields a discrete version of the Laplace equation:

$$\frac{c_{i+1,j} - 2c_{i,j} + c_{i-1,j}}{h^2} + \frac{c_{i,j+1} - 2c_{i,j} + c_{i,j-1}}{h^2} = 0 \tag{6.7}$$

We can rearrange the equation to obtain

$$\frac{1}{h^2}\left(c_{i+1,j} + c_{i-1,j} + c_{i,j+1} + c_{i,j-1} - 4c_{i,j}\right) = 0 \tag{6.8}$$

This equation is valid at each i, j-node. The assembly of all these equations taken for all the possible values of i and j defines a matrix linear problem. To see that clearly let's define a vector C as

$$C = [c_{0,0}, c_{0,1}, ...c_{0,N}, c_{1,0}, c_{1,1}, ..., c_{1,N}, ..., ...c_{N,0}, ..., c_{N,N}] \tag{6.9}$$

We can rewrite eq 6.8 as

$$A \cdot C = 0 \tag{6.10}$$

where A is a $N^2 \times N^2$ matrix containing simple coefficients. Using finite difference we have therefore transformed the PDEs in a simple linear system that can be solved with any techniques seen in the Chapter 3.

6.2.1 Node mapping

To solve the linear systems 6.10, we first need to develop a mechanism to map each node (i, j) in a unique position labeled IJ in the C and A. This mapping is given by the following equation

$$IJ \longleftarrow (i, j) \tag{6.11}$$
$$IJ = i * N + j \tag{6.12}$$

where $i = [0, 1, ...M]$ and $j = [0, 1, .., N]$. Hence $(i = 0, j = 0) \rightarrow IJ = 0$, $(i = 0, j = 1) \rightarrow IJ = 1$, $(i = 1, j = 0) \rightarrow IJ = N$, ... It is now possible to construct all the matrices and vectors required to solve the equations (6.10) following this scheme. Then A is a $N^2 \times N^2$ whose elements $A[IJ, KL]$ can be computed with the finite difference method presented above.

6.2.2 Boundary conditions

Handling the boundary condition require a bit of attention. We present here how to incorporate these conditions using finite difference.

Dirichlet conditions The Dirichlet condition simply fix the value of c at the boundary of the domain. As an example Let's assume that one boundary condition of the problem is

$$c(x = 0, y) = \sin(\pi y) \tag{6.13}$$

To implement such condition we must modify the matrix A such as if $i = 0$

$$A[IJ, IJ] = 1 \tag{6.14}$$
$$A[IJ, KL \neq IJ] = 0 \tag{6.15}$$
$$b[IJ] = \sin(\pi y_j) \tag{6.16}$$
$$\tag{6.17}$$

with $IJ = i * N + j$. Setting the diagonal element $A[IJ, IJ]$ to 1 will enforce the Dirichlet boundary condition automatically.

Neumann conditions The Neumann condition fixes the value of the derivative of the c at the border of the domain. Let's assume here that one of the boundary condition of the problem reads

$$\frac{\partial c}{\partial y} = \beta \qquad \text{at } y = 0 \text{ and all } x \tag{6.18}$$

This boundary condition can be incorporated in the simulation to compute $c_{i,0}$. This is done replacing the above equation with the forward difference. Depending on the accuracy desired two equations can be used

$$\frac{1}{\Delta y}(c_{i,1} - c_{i,0}) = \beta \tag{6.19}$$
$$\text{or} \qquad \frac{1}{2\Delta y}(-c_{i,2} + 4c_{i,1} - 3c_{i,0}) = \beta \tag{6.20}$$

These two equations leads to the explicit formula

$$c_{i,1} - c_{i,0} = \Delta y \beta \tag{6.21}$$

$$\text{or} \qquad -3c_{i,0} + 4c_{i,1} - c_{i,2} = 2\beta\Delta y \tag{6.22}$$

These two equations can easily be incorporated in the definition of the matrix A. Hence using the second equation we have if $j = 0$

$$A[IJ, IJ] = -3 \tag{6.23}$$

$$A[IJ, I(J+1)] = 4 \tag{6.24}$$

$$A[IJ, I(J+2)] = -1 \tag{6.25}$$

$$b[IJ] = 2\beta\Delta y \tag{6.26}$$

Here $I(J+1) = i * N + (j+1)$. Note that if the Neumann condition is specified on the x-axis indexes such as $(I+1)J = (i+1) * N + j$ would appear. Note as well that if the Neumann condition is specified at the end of the domain, i.e.

$$\frac{\partial c}{\partial y} = \beta \qquad \text{at } y = y_{max} \text{ and all } x \tag{6.27}$$

the backward difference should be used instead of the forward difference. The backward difference leads then to the equations

$$c_{i,N_y} - c_{i,N_y-1} = \Delta y \beta \tag{6.28}$$

$$\text{or} \qquad -3c_{i,N_y} + 4c_{i,N_y-1} - c_{i,N_y-2} = 2\beta\Delta y \tag{6.29}$$

where N_y is the last node on the y-axis. These equations can be easily included in the definition of A.

6.2.3 Example : the Laplace equation

The snippet of code below does exactly that. After defining the constant of the problem the program uses two nested loops that go through all the nodes of the system and define the matrix A and the right-hand side b. Note that the boundary conditions (here all Dirichlet conditions) are directly introduced in the b vectors that defines the right-hand side.

```
1  import numpy as np
2  from scipy.linalg import solve
3  from scipy.sparse.linalg import spsolve
4  import matplotlib.pyplot as plt
5  # define the space
6  x0,xf,nX = 0.0,1.0,10
7  y0,yf,nY = x0,xf,nX
8
9  # vectors
10 X = np.linspace(x0,xf,nX)
11 Y = np.linspace(y0,yf,nY)
12
13 # increment
14 h = X[1]-X[0]
15 h2 = 1./h/h
```

```
16
17 # total size of the system and matrices
18 Ntot = nX*nY
19 A = np.zeros((Ntot,Ntot))
20 b = np.zeros(Ntot)
21
22 # nested loop for the mapping
23 for i in range(nX):
24         for j in range(nY):
25
26                 # mapping relation
27                 IJ = i*nY + j
28
29                 # neighbours
30                 IJp1 = i*nY + (j+1)
31                 IJm1 = i*nY + (j-1)
32                 Ip1J = (i+1)*nY + j
33                 Im1J = (i-1)*nY + j
34
35                 # check the boundaries
36                 # and fill up the matrices
37                 if i == 0:
38                         A[IJ,IJ] = 1.0
39                         b[IJ] = np.sin(np.pi*Y[j])
40                 elif (i == nX-1):
41                         A[IJ,IJ] = 1.0
42                         b[IJ] = 0.0
43                 elif (j == 0):
44                         A[IJ,IJ] = 1.0
45                         b[IJ] = 0.0
46                 elif (j == nY-1):
47                         A[IJ,IJ] = 1.0
48                         b[IJ] = 0.0
49                 else:
50                         A[IJ,IJ] = -4*h2
51                         A[IJ,IJp1] = h2
52                         A[IJ,IJm1] = h2
53                         A[IJ,Ip1J] = h2
54                         A[IJ,Im1J] = h2
55                         b[IJ] = 0.0
```

We can then solve and plot the solution of the system. We recommend here to use the *scipy.linalg.solve*() module. This function returns a 1D vectors that has to be reshaped before plotting. This can be done with the function *.reshape*() as shown in the snippet below

```
1 z = solve(A,b)
2 Z = z.reshape(nX,nY)
3 plt.contour(X,Y,Z)
4 plt.colorbar()
5 plt.xlabel('x')
```

```
plt.ylabel('y')
plt.show()
```

This code gives the distribution of temperature in the x, y plane according to the boundary conditions given above. This distribution is represented in Fig. 6.1.

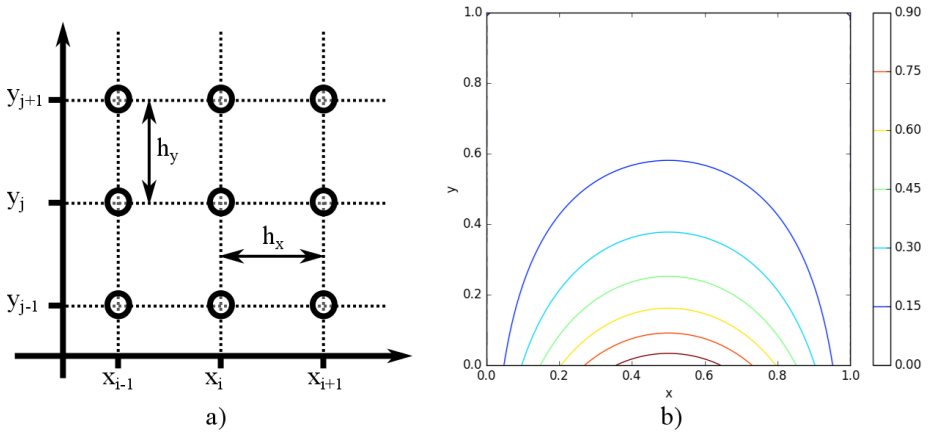


Figure 6.1: a) Grid used in the finite difference method to solve PDEs. b) Solution of the Laplace equation

6.2.4 Sparse matrices

As you can see the matrix A contains a lot of elements that are null. Such matrix is called a *sparse* matrix. Different methods have been developed to make the handling of sparse matrices more efficient. You can imagine that using normal matrix-matrix multiplication, to multiply two sparse matrices, the computer spend most of his time multiplying 0 by 0 to obtain more 0s. A widely used approach to deal with sparse matrices is to store only the elements that are not null and only use these elements when handling these matrices.

Different ways of storing and handling sparse matrices have been implemented in the module *scipy.sparse*. We will use two of these formats: the LInked List format or *LIL* and then Compressed sparse row format or *CSR*. The LIL format allows for an easy definition of sparse matrices that is similar to the *numpy* arrays. On the other hand the CSR format is extremely well suited for matrix-vector multiplication. The conversion from LIL to CSR is made really easy in scipy. Finally the submodule *scipy.sparse.linalg* contains different routines to perform linear algebra operations on sparse matrices. For example the function *scipy.sparse.linalg.spsolve*() allows solving system of linear equations $A \cdot x = b$, where A is a CSR sparse matrix. The snippet of code below present how to use sparse matrices to solve the Laplace equation has presented in the previous section

```
import numpy as np
import scipy.sparse as sp
from scipy.sparse.linalg import spsolve

# define the space
# ....
# ....

# total size of the system and matrices
```

```
10 Ntot = nX*nY
11 A = sp.lil_matrix((Ntot,Ntot))
12 b = np.zeros(Ntot)
13
14 # nested loop for the mapping
15 for i in range(nX):
16         for j in range(nY):
17
18                 # ...
19                 # ...
20
21 # convert the LIL matrix to a CSR martix
22 A = A.tocsr()
23
24 # solve using spsolve
25 z = spsolve(A,b)
```

If you compare this code snippet to the one presented in the previous section you'll see that very little has changed. The only differences is that the matrix A is now declared as a lil_matrix instead of a regular numpy array (line 20). We then transform this LIL matrix into a CSR matrix (line 59) and use the routine $spsolve()$ instead of *solve* to solve the system of linear equations. If the code changes only a little its performance are drastically improved. The calculation time obtained using normal and sparse matrices are reported for different mesh sizes in Fig. 6.2. As you can see on this figure several order of magnitude can be gained by using sparse matrices to solve system of linear equation. This is only possible if the matrix A is really sparse. Using sparse routines on non-sparse matrices will increase the calculation time dramatically.

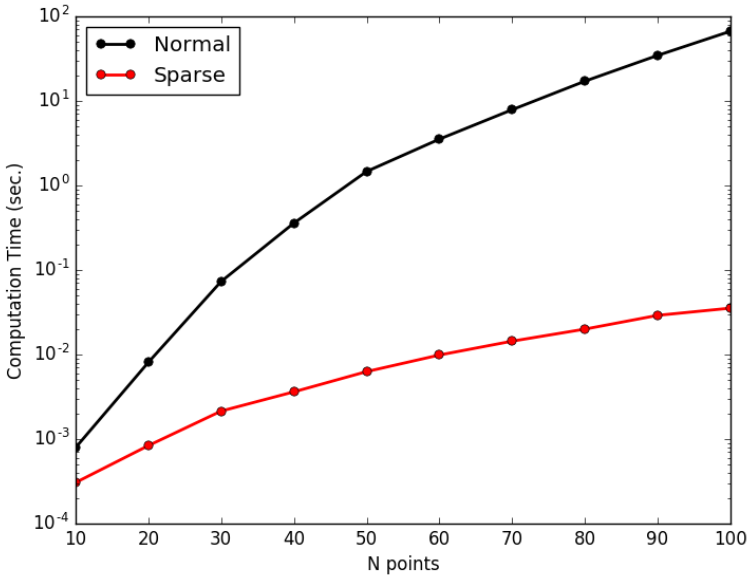


Figure 6.2: Computation time required to solve the Laplace problem using normal and sparse matrices with different mesh size

6.2.5 PDEs in 3-dimensions

The generalization of the FD method to the 3D case is straightforward. Consider for example the equations

$$\frac{\partial^2 c}{\partial x^2} + \frac{\partial^2 c}{\partial y^2} + \frac{\partial^2 c}{\partial z^2} = 0 \tag{6.30}$$

which is nothing else than the Laplace equations but in 3D. As for the 2D case we will start by creating a regular mesh of nodes that span the x, y, z space. Each node is then characterized by 3 numbers i, j and k that refers to its positions along the x,y, and z axis. The finite difference expressions of the partial derivatives are used to transform the PDE shown above to a set of linear equations. The main issue in the establishment of these linear system consists in mapping correctly each node (i, j, k) to a single index IJK. Similarly to the 2D case the equation supporting this mapping is given by

$$IJK \longleftarrow i \times (N_j \times N_k) + j \times N_k + k \tag{6.31}$$

with $i = [0, ..., N_i]$, $j = [0, ..., N_j]$ and $k = [0, ..., N_k]$. The finite difference method applied to the 3D case leads to an equation similar to eq. 6.8

$$\frac{1}{h^2}\left(c_{i+1,j,k} + c_{i-1,j,k} + c_{i,j+1,k} + c_{i,j-1,k} + c_{i,j,k+1} + c_{i,j,k-1} - 6c_{i,j,k}\right) = 0 \tag{6.32}$$

This equation can easily be written in a matrix format $A \cdot = b$ using the node mapping approach presented above. Once the system of equations set, the *scipy.linalg.sparse.spsolve*() can be used to obtain the value of the temperature at each point in the domain. The implementation of the 3D Laplace equation is a straightforward extension of the 2D case studied above. We present in Fig. 6.3 the results of this equation with the boundary condition

$$c(x, y, z = 0) = \sin(\pi x)\sin(\pi y) \tag{6.33}$$

and $c = 0$ on all the other boundaries. To plot these results we cannot simply plot a point at each vertex of the 3D mesh. However we can compute the 'contour' of the 3D data. Such contour are called isosurface. An isosurface is a surface that marks the points in space for which a given function $f(x, y, z)$ is constant. Computing such isosurface can be done with the so-called *marching cube* algorithm. This algorithm has been implemented in the a python module called *skimage* that we use in the following snippet of code.

6.3 Finite difference for time-dependent PDEs

In the PDEs presented above all the variables were spatial variable and these equations were used to describe the change of a given quantity in 1D, 2D or 3D. It happens often that the PDE also contains temporal derivative. This is for example the case for the heat conduction equation

$$\frac{\partial T}{\partial t} = \alpha\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2}\right) \tag{6.34}$$

These equations are often called *parabolic* equations. We present here two methods, one explicit and one implicit, that can be used to solve them.

6.3.1 Explicit Method : forward Euler in 1D

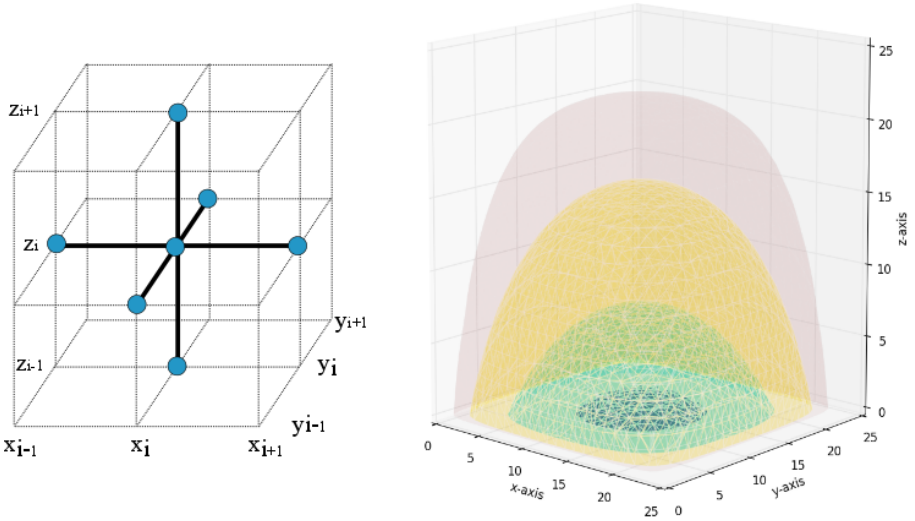


Figure 6.3: a) Regular mesh used to solve PDE in 3D. b) Solution of the Laplace equation (6.30) with the boundary condition (6.33). 4 isosurface are represented $c(x, y, z) = 0.75$ (blue), 0.25 (green), 0.05 (yellow) and 0.01 (pink).

To illustrate the forward Euler method let's consider the simple 1D parabolic PDE

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} \tag{6.35}$$

A simple explicit method to solve parabolic PDE can be obtained by replacing the derivatives with their numerical approximations. As before we use the centered difference for all spatial derivative

$$\frac{\partial^2 u}{\partial x^2} = \frac{1}{\Delta x^2}(u_{i+1,n} - 2u_{i,n} + u_{i-1,n}) \tag{6.36}$$

In the notation used above $u_{i,n}$ is the value of u on the i-th node on the x-axis, i.e. at $x = i \times \Delta x$ and at time $t = n \times \Delta t$. Similarly we can use finite difference to express the temporal derivative. As you may have guessed from the name of the method, we use here the forward Euler expression of the first-order time derivative

$$\frac{\partial u}{\partial t} = \frac{1}{\Delta t}(u_{i,n+1} - u_{i,n}) \tag{6.37}$$

Using these expressions, the eq. 6.35 can be written as

$$u_{i,n+1} = \frac{\alpha \Delta t}{\Delta x^2}(u_{i+1,n} + u_{i-1,n}) + \left(1 - 2\frac{\alpha \Delta t}{\Delta x^2}\right) u_{i,n} \tag{6.38}$$

This equation is explicit since the value of u at $t = (n+1) \times \Delta t$ only depends on the values of the function at $t = n \times \Delta t$.

6.3.2 Explicit Method : Stability of the 2D forward Euler method

The Euler method can be easily generalized in the case where we have several dimensional variables. Consider for example the 2D diffusion equation

$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \tag{6.39}$$

Using finite difference expressions this PDE can be written as

$$\begin{aligned} u_{i,n+1} &= \frac{\alpha\Delta t}{\Delta x^2}(u_{i+1,j,n} + u_{i-1,j,n}) \\ &+ \frac{\alpha\Delta t}{\Delta y^2}(u_{i,j+1,n} + u_{i,j-1,n}) \\ &+ \left(1 - 2\frac{\alpha\Delta t}{\Delta x^2} - 2\frac{\alpha\Delta t}{\Delta y^2}\right) u_{i,j,n} \end{aligned} \tag{6.40}$$

where u depends now on a new index j that refers to $y = j \times \Delta y$. Explicit formula such as the one above can however be *unstable* meaning that their solution rapidly diverges from the real value of the variable. A rule of thumb is that if all coefficients on the right-hand side are positive, then the scheme is stable. For the equation above this means

$$1 - 2\frac{\alpha\Delta t}{\Delta x^2} - 2\frac{\alpha\Delta t}{\Delta y^2} \geq 0 \tag{6.41}$$

6.3.3 Implicit Method : the Crank-Nicolson approach

A more accurate approach for the solution of PDEs such as

$$\frac{\partial u}{\partial t} = \alpha\frac{\partial^2 u}{\partial x^2} \tag{6.42}$$

relies on the central difference expressed at the half point $i, n + 1/2$ that lies between the points i, n and $i, n + 1$. The temporal derivative is then expressed as

$$\frac{\partial u}{\partial t}|_{i,n+1/2} = \frac{1}{\Delta t}(u_{i,n+1} - u_{i,n}) \tag{6.43}$$

In addition the spatial derivative is here expressed at the half point as the average of the central difference at $(i, n + 1)$ and (i, n). This leads to

$$\begin{aligned} \frac{\partial^2 u}{\partial x^2}|_{i,n+1/2} &= \frac{1}{2}\left(\frac{\partial^2 u}{\partial x^2}|_{i,n+1} + \frac{\partial^2 u}{\partial x^2}|_{i,n}\right) && (6.44) \\ &= \frac{1}{2\Delta x^2}(u_{i+1,n+1} - 2u_{i,n+1} + u_{i-1,n+1}) \\ &+ \frac{1}{2\Delta x^2}(u_{i+1,n} - 2u_{i,n} + u_{i-1,n}) && (6.45) \end{aligned}$$

We can hence write the eq 6.42 as

$$\begin{aligned} -\left(\frac{\alpha\Delta t}{\Delta x^2}\right)(u_{i-1,n+1} + u_{i+1,n+1}) + 2\left(1 + \left(\frac{\alpha\Delta t}{\Delta x^2}\right)\right) u_{i,n+1} \\ = \left(\frac{\alpha\Delta t}{\Delta x^2}\right)(u_{i-1,n} + u_{i+1,n}) + 2\left(1 - \left(\frac{\alpha\Delta t}{\Delta x^2}\right)\right) u_{i,n} \end{aligned} \tag{6.46}$$

This last equation is called the Crank-Nicolson implicit formula for the solution of parabolic PDEs. As one can see from the previous equation, the Crank-Nicolson formula when written for the entire grid generates a set of simultaneous algebraic linear equations whose matrix is usually trigonal. This system relates the values of $u_{i\pm1,n+1}$ to the values at the previous time step, i.e. $u_{i\pm1,n}$. Hence starting form the initial condition $u_{[0,\ldots,N_X],n=0}$ one can compute the values at $u_{[0,\ldots,N_X],n=1}$ and so on. Such system can be solved using direct or iterative methods seen in Chapter 3. The implicit formula have been found to be unconditionally stable.

6.3.4 Example: drug release

To illustrate the use of the Euler and Crank-Nicolson method for the solution of parabolic PDEs, let's study the diffusion of a pharmaceutical drug in the body of a patient over time. The device that delivers the drug is embedded in a polymer matrix that slowly diffuses the drug over time (see Fig. 6.4). The diffusion of the drug can then be simulated with the equation

$$\frac{\partial c}{\partial t} = D\frac{\partial^2 c}{\partial x^2} \tag{6.47}$$

where D is the diffusion coefficient and c is the drug concentration. The distance between the surface of the device ($x = 0$) to its center ($x = 1$) is equal to 1 (dimensionless). We know that the concentration at the surface is 0 ($c(x = 0) = 0$) as any drug reaching the surface is immediately swept away. We also know that since the device is symmetric we must have $\frac{\partial c(x=1.0)}{\partial x} = 0$. Finally we assume that the initial concentration is 1.0 everywhere except at the surface where it's null, i.e. $c(t = 0.0 < x \leq 1) = 1.0$ and $c(t = 0, x = 0) = 0.0$.

This problem is therefore a 1D parabolic PDE with a mix of Dirichlet boundary condition, $c(x = 0) = 0$, and Neumann boundary condition, $\frac{\partial c(x=1.0)}{\partial x} = 0$. Such boundary problem are usually referred to as Robin boundary condition. We furthermore have the initial condition $c(t = 0.0 < x \leq 1) = 1.0$ and $c(t = 0, x = 0) = 0.0$.

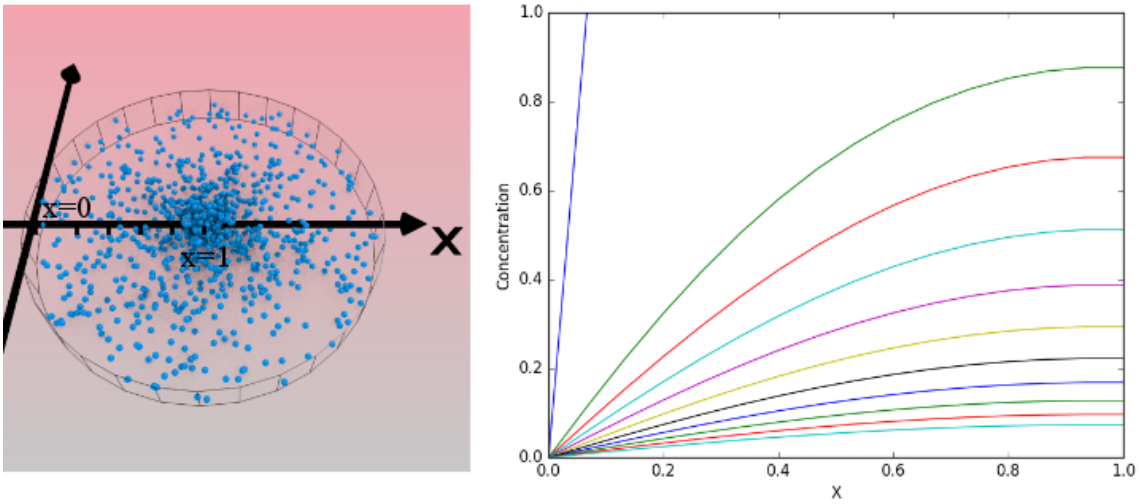


Figure 6.4: a) Representation of the drug delivery device. We assume here that diffusion is isotropic in the x and y direction b) Solution obtained with the forward Euler method. The different lines represent different time starting from $t = 0.0$ (blue) to $t = 50$ (cyan). A time interval of $\Delta t = 0.25$ was chosen to solve the PDE ($N_t = 200$).

Handling boundary conditions

Both for the Euler and Crank-Nicolson method we will use a grid made of M_X points on the x-axis regularly spaced with an interval h. Similarly we'll take N_T points during the time of the diffusion process with an interval of Δt. With this setup in mind, the Dirichlet boundary condition ($c(x = 0) = 0$)

can be translated into

$$c_{i=0,n+1} = 0.0 \tag{6.48}$$

for any value of n. This condition is identical to the ones encountered in the previous PDEs problem. We however have here a Neumann condition $\frac{\partial c(x=1.0)}{\partial x} = 0$. To understand how to implement this condition in our numerical procedure, let's write it as a finite difference

$$\frac{\partial c}{\partial x}_{x=1.0} = \frac{c_{M_X,n} - c_{M_X-1,n}}{h} = 0 \tag{6.49}$$

where M_X is the the number of points along the x-axis. The Neumann condition therefore reads

$$c_{M_X,n} = c_{M_X-1,n} \tag{6.50}$$

To enforce the Neuman boundary condition we therefore have to equate the last element of c to the second to last.

Euler Method

The application of the Euler method is straightforward and leads to the equation

$$c_{i,n+1} = c_{i,n} + \Delta t \frac{D}{h^2}(c_{i+1,n} - 2c_{i,n} + c_{i-1,n}) \tag{6.51}$$

This equation can be solved easily with the boundary conditions presented above and the initial condition $c_{x,0} = [0, 1, ..., 1]$. The snippet of code below illustrates the implementation of the Euler method to solve this diffusion problem.

```
import numpy as np

# size of the grids
MX = 16
nT = 201

# spatial coordinate
X = np.linspace(0,1,MX)
h = X[1]-X[0]

#temporal coordinate
t0, tmax = 0.0, 50.0
T = np.linspace(t0,tmax,nT)
dT = T[1]-T[0]

# diffusion coefficient
D = 1*1E-2
Dh2 = D/(h**2)

# initial condition
c0 = np.ones(MX)
c0[0] = 0.0

# declare the solution arrat
c1 = np.zeros(MX)
```

```
26
27 # loop over the time
28 for n in range(1,nT):
29
30         # Dirichlet BC
31         c1[0] = 0.0
32
33         for iX in range(1,MX-1):
34                 c1[iX] = c0[iX] + dT*Dh2 * (c0[iX+1]-2.0*c0[iX]+c0[iX-1])
35
36         # Neumann BC
37         c1[-1] = c1[-2]
38
39         # iterate
40         c0 = c1
```

Crank-Nicolson

To ease the writing of the Crank-Nicolson equation let's pose

$$R = \left(\frac{\alpha \Delta t}{h^2}\right) \qquad K = 2(1+R) \qquad J = 2(1-R) \tag{6.52}$$

The Crank-Nicolson equation applied to our drug diffusion problem then reads

$$-R(c_{i-1,n+1} + c_{i+1,n+1}) + Kc_{i,n+1} = R(c_{i-1,n} + c_{i+1,n}) + Jc_{i,n} \tag{6.53}$$

This system of equation can be written as a matrix equation $\mathbf{A} \cdot \mathbf{c} = \mathbf{b}$ following

$$\begin{pmatrix} \ddots & \ddots & \ddots & & & & \\ & -R & K & -R & & & \\ & & -R & K & -R & & \\ & & & -R & K & -R & \\ & & & & \ddots & \ddots & \ddots \end{pmatrix} \cdot \begin{pmatrix} \vdots \\ c_{i-1,n+1} \\ c_{i,n+1} \\ c_{i+1,n+1} \\ \vdots \end{pmatrix} = \begin{pmatrix} \vdots \\ \vdots \\ R(c_{i-1,n} + c_{i+1,n}) + Jc_{i,n} \\ \vdots \\ \vdots \end{pmatrix} \tag{6.54}$$

The boundary conditions give then the first and last row of the matrix. The Dirichlet condition impose that the first row of $\mathbf{A}$ contains a 1 at the first position and 0 elsewhere and that the first element of $\mathbf{b}$ is set to 0. This leads to the equation $c_{0,n+1} = 0$. The Neumann boundary condition then impose that $c_{M_X,n+1} = c_{M_X-1,n+1}$. To implement this condition we need to set the last row of $\mathbf{A}$ to $[0, 0, ..., -1, 1]$ and the impose that the last element of $\mathbf{b}$ equals 0. The snippet of code below shows the implementation of the Crank-Nicolson method for the solution of the diffusion problem. Note that even though we have a simple 1D system we use here sparse matrices to reduce the computation time.

```
1 import numpy as np
2 import scipy.sparse as sp
3 from scipy.sparse.linalg import spsolve
4
5 # size of the grids
6 MX = 16
7 nT = 201
8
9 # spatial coordinate
```

```
X = np.linspace(0,1,MX)
h = X[1]-X[0]

#temporal coordinate
t0, tmax = 0.0, 50.0
T = np.linspace(t0,tmax,nT)
dT = T[1]-T[0]

# diffusion coefficient
D = 1*1E-2
r = D*dT/(h**2)

# initial condition
c0 = np.ones(MX)
c0[0] = 0.0

# declare the solution arrat
c1 = np.zeros(MX)

# Matrix for the linear system
A = sp.lil_matrix((MX,MX))
b = np.zeros(MX)

# Dirichlet BC
A[0,0] = 1.0
b[0] = 0.0

# Neuman BC
A[-1,-2] = -1
A[-1,-1] = 1
b[-1] = 0.0

# form the matrix
for iX in range(1,MX-1):
        A[iX,iX-1] = -r
        A[iX,iX]   = 2.*(1+r)
        A[iX,iX+1] = -r
A = A.tocsr()

# loop over the time
for n in range(1,nT):

        # form the right-hand side
        for iX in range(1,MX-1):
                b[iX] = r*c0[iX-1] + 2*(1.0-r)*c0[iX] + r*c0[iX+1]

        # solve the linear system of equation
        c1 = spsolve(A,b)

```

```
59          # iterate
60          c0 = c1
```

The results of the Euler and Crank-Nicolson approach are reported on Fig. 6.5 for different number of time steps. If the number of time step is large enough the two methods yield identical results. However the advantage of the Crank-Nicolson method is evident for the case were only 40 points along the time axis are considered. In this case the Euler method is unstable and gives nonsensical results. In contrast the Crank Nicolson remains stable and give correct results.

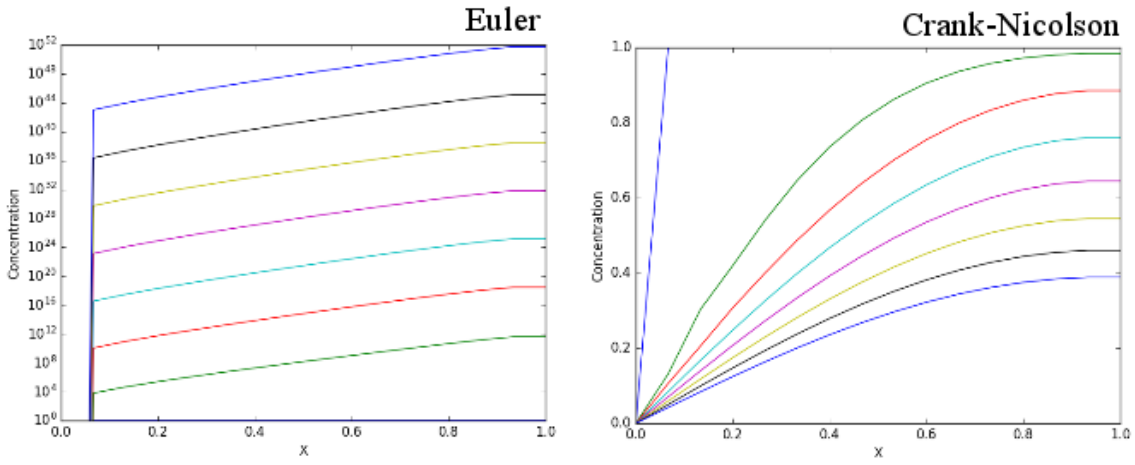


Figure 6.5: Comparison of the results obtained with the Euler and Crank-Nicolson method for a time interval of $\Delta t = 1.25$. While the Euler method diverges and yield wrong results and Crank-Nicolson approach remains valid.

6.4 The wave equation

The wave equation is a so-called 'hyperbolic' PDEs that reads in 2D:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \tag{6.55}$$

where c is a fixed constant. the This equations describes the time evolution of a quantity $u(x, y, t)$ in a 2D space and appears in many fields of chemistry and physics such as acoustic, water wave dynamics, fluid dynamics, electromagnetic etc ... This equation is subjected to the initial condition

$$u_{x,y,t=0} = f(x, y) \qquad \text{and} \qquad \frac{\partial u(x, y, t = 0)}{\partial t} = g(x, y) \tag{6.56}$$

that provides the initial value of u and of its time derivative in the 2D space. Boundary conditions must also be specified to solve the wave equation. We suppose here that

$$\frac{\partial u}{\partial x} = 0 \qquad \text{and} \qquad \frac{\partial u}{\partial y} = 0 \tag{6.57}$$

at the boundary of the 2D space.

We present here how to use the Euler method solve the wave equation. Using finite difference we can rewrite the wave equation as

$$\frac{u_{i,j}^{k+1} - 2u_{i,j}^{k} + u_{i,j}^{k-1}}{\Delta t^2} = c^2 \frac{u_{i,j+1}^{k} - 2u_{i,j}^{k} + u_{i,j-1}^{k}}{\Delta x^2} + c^2 \frac{u_{i+1,j}^{k} - 2u_{i,j}^{k} + u_{i-1,j}^{k}}{\Delta y^2} \tag{6.58}$$

At the difference of the previous PDEs we have seen the wave equation present a second-prder time derivative that needs a particular attention. As a consequence a special step has to be taken for the first iteration of the solution.

First iteration: We can see here that to obtain $u_{i,j}^{k+1=1}$, i.e. the the value of u at the first time step we need to know $u_{ij}^{k=0}$ and $u_{ij}^{k-1=-1}$. The value of u_{ij}^{0} is known since it is the initial condition. We have therefore

$$u_{ij}^{0} = f_{ij} \tag{6.59}$$

We can obtain the value of u_{ij}^{-1} with the initial value of $\partial u/\partial t$:

$$\frac{\partial u(x,y,t=0)}{\partial t} = g(x,y) \quad \longrightarrow \quad \frac{u_{ij}^{1} - u_{ij}^{-1}}{2\Delta t} = g_{ij} \quad \longrightarrow \quad u_{ij}^{-1} = u_{ij}^{1} - 2\Delta t g_{ij} \tag{6.60}$$

By introducing these terms, in eq 6.58 we obtain

$$\frac{2u_{i,j}^{1} - 2f_{i,j} - 2\Delta t g_{i,j}}{\Delta t^2} = c^2 \frac{f_{i,j+1} - 2f_{i,j} + f_{i,j-1}}{\Delta x^2} + c^2 \frac{f_{i+1,j} - 2f_{i,j} + f_{i-1,j}}{\Delta y^2} \tag{6.61}$$

To facilitate the writing of this equation we can define the Courant numbers

$$\gamma_x = \frac{c\Delta t}{\Delta x} \qquad \gamma_y = \frac{c\Delta t}{\Delta y} \qquad \gamma = 2(1 - \gamma_x^2 - \gamma_y^2) \tag{6.62}$$

and write the explicit equations of the first iteration time step

$$u_{ij}^{1} = \frac{\gamma}{2} f_{ij} + \frac{\gamma_x^2}{2}(f_{i,j+1} + f_{i,j-1}) + \frac{\gamma_y^2}{2}(f_{i+1,j} + f_{i-1,j}) + \Delta t g_{ij} \tag{6.63}$$

Subsequent iterations: The subsequent iterations can directly be computed from the values of u on the two previous ones. Hence we obtain the explicit equation

$$u_{ij}^{k+1} = \frac{\gamma}{2} u_{ij}^{k} + \gamma_x^2 (u_{i,j+1}^{k} + u_{i,j-1}^{k}) + \gamma_y^2 (u_{i+1,j}^{k} + u_{i-1,j}^{k}) - u_{ij}^{k-1} \tag{6.64}$$

Boundary conditions: Finally the boundary conditions $\partial u/\partial x = 0$ and $\partial u/\partial y = 0$ on the border of the domain forces us to impose

$$u_{0,j}^{k+1} = u_{1,j}^{k+1} \qquad u_{N_x,j}^{k+1} = u_{N_x-1,j}^{k+1} \tag{6.65}$$

$$u_{i,0}^{k+1} = u_{i,1}^{k+1} \qquad u_{i,N_y}^{k+1} = u_{i,N_y-1}^{k+1} \tag{6.66}$$

The snippet of code below shows how to solve the wave equation as outlined above using a simple Euler scheme. The results of this code can be visualized in Fig. 6.6

```
1 import numpy as np
2
3 # size of the system
4 x0,xmax,NX = -1.5,1.5,100
5 y0,ymax,NY = x0,xmax,NX
```

```
X = np.linspace(x0,xmax,NX)
Y = X

# evolution time
t0,tmax,NT=0,4,400
T = np.linspace(t0,tmax,NT)
dT = T[1]-T[0]

# initial condition
XX,YY = np.meshgrid(X,Y)
f = np.exp(-10*(XX**2+YY**2))
G = np.zeros((NX,NY))

# speed constant
c = 1.0

# Courant constant
dX, dY, dT = X[1]-X[0], Y[1]-Y[0], T[1]-T[0]
gx, gy = c*dT/dX, c*dT/dY
gx2, gy2 = gx**2, gy**2
g = 2.0*(1.0 - gx2 - gy2)

# declare the solution
u = np.zeros((NY,NX,NT+1))

# initial condition
u[:,:,0] = f

# compute the first time step
u[:,:,1] = g/2.0*u[:,:,0] + dT*G
u[1:-1,1:-1,1] += gx2/2.*(f[1:-1,2:]+f[1:-1,:-2])
u[1:-1,1:-1,1] += gy2/2.*(f[2:,1:-1]+f[:-2,1:-1])

for k in range(1,NT):

        # all the terms have those ones
        u[:,:,k+1] = g*u[:,:,k] - u[:,:,k-1]

        # interior point
        u[1:-1,1:-1,k+1] += gx2*(u[1:-1,2:,k]+u[1:-1,:-2,k])
        u[1:-1,1:-1,k+1] += gy2*(u[2:,1:-1,k]+u[:-2,1:-1,k])

        # boundary conditions (corners)
        u[0,0,k+1] = u[1,1,k+1]
        u[-1,0,k+1] = u[-2,1,k+1]
        u[0,-1,k+1] = u[1,-2,k+1]
        u[-1,-1,k+1] = u[-2,-2,k+1]

        # boundary conditions (sides)
```

```
        u[1:-1,0,k+1] = u[1:-1,1,k+1]
        u[1:-1,-1,k+1] = u[1:-1,-2,k+1]
        u[0,1:-1,k+1] = u[1,1:-1,k+1]
        u[-1,1:-1,k+1] = u[-2,1:-1,k+1]
```

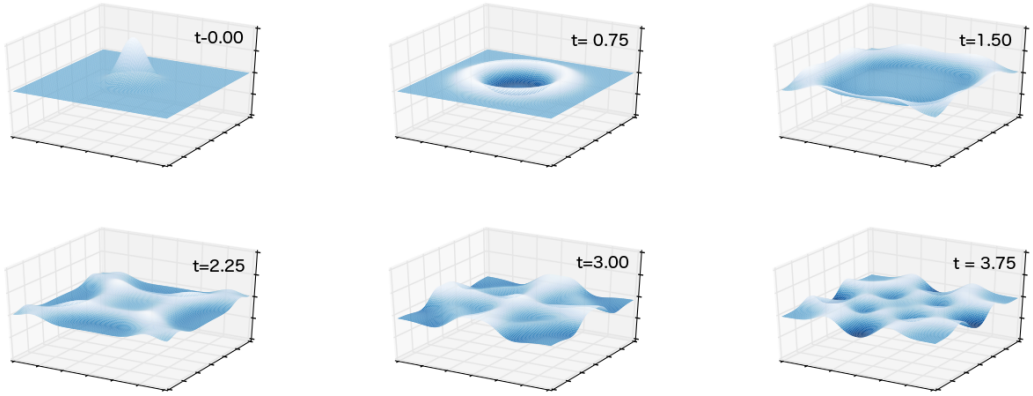


Figure 6.6: Solution of the wave equation at different time

6.5 Exercise

6.5.1 Unsteady-state velocity profile in a pipe

A fluid of constant density ρ and viscosity μ is contained in a long horizontal pipe of length L and radius R. Initially the fluid is at rest. At $t = 0$, a pressure difference of $\Delta P = (P_0 - P_L)/L$ is imposed on the system. The differential equation that governs the dynamics reads

$$\rho \frac{\partial v}{\partial t} = \frac{\Delta P}{L} + \mu \frac{1}{x} \frac{\partial}{\partial x} \left(x \frac{\partial v}{\partial x} \right) \tag{6.67}$$

with $-R \leq x \leq R$ and $t \geq 0$. This 1D parabolic PDEs is subjected to the initial condition

$$v = 0 \qquad \text{at } t = 0, \qquad \text{for } 0 \leq x \leq R \tag{6.68}$$

that states that the fluid is at rest at $t = 0$. We also have the Dirichlet boundary condition

$$v = 0 \qquad \text{at } x = R, \qquad \text{for } t \geq 0 \tag{6.69}$$

that states that the velocity at the border of the pipe is null. Finally we have the Neumann boundary condition

$$\frac{\partial v}{\partial x} = 0 \qquad \text{at } x = 0, \qquad \text{for } t \geq 0 \tag{6.70}$$

since the the velocity profile is supposed to be symmetric within the tube.

A - Use the finite difference method to obtain the steady state velocity profile within the tube. As a reminder, the steady state solution can be obtained by setting

$$\frac{\partial v}{\partial t} = 0 \tag{6.71}$$

in the PDE 6.67. By applying the finite difference approach to this equation you should obtain a linear system of equation that can be solve using for example $numpy.linalg.solve()$. This should remind you of an exercise we've done in chapter 4.

B - Using the techniques we've seen in this chapter, we can solve the full equation 6.67 and obtain the time dependent velocity profile along the pipe. To this end, use either the Euler or the Crank-Nicolson to solve this equation.

6.5.2 Metallic plate subjected to different heat sources

You've been hired by an metallurgic company to predict the heat distribution inside a metallic plate of 1m $\times$ 1m that is subjected to different heat sources. The heat distribution within the plate can be simulated with the 2D elliptic PDE

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = f \tag{6.72}$$

where $f = -Q/k$ is assumed to be a constant. This constant is null if the there is no internal heat source in the plate and is different form o otherwise. Q is the heat generated per unit of volume and k the thermal conductivity of the material

A - Your first task is to determine the temperature profile within the plates if its four edges are maintain at constant temperature. The Dirichet boundary conditions of the problem are therefore given by

$$\begin{aligned} T(0,y) &= 250\ ^oC & T(1,y) &= 100\ ^oC \\ T(x,0) &= 500\ ^oC & T(x,1) &= 25\ ^oC \end{aligned}$$

B - Perfect insulation is then installed at the right and top edges of the plate while the two other edges are still maintained at constant temperature. Modify your program to account for the following set of Dirichlet and Neumann conditions

$$T(0,y) = 250\ ^oC \qquad T(x,0) = 500\ ^oC \tag{6.73}$$

$$\frac{\partial T}{\partial x}|_{1,y} = 0 \qquad \frac{\partial T}{\partial y}|_{x,1} = 0 \tag{6.74}$$

and evaluate the new temperature profile in the plate.

C - The plate is subjected to an electric current that creates an uniform heat source within the plate. The amount of heat generated is $Q = 100$ kW/m^3 and the thermal conductivity of the plate is $k = 16$ W/m.K. The edges of the plate are in contact with a cooling fluid maintained at 25 ^{o}C. This set up leads to the Robbins boundary conditions

$$\frac{\partial T}{\partial x}|_{0,y} = 5(T(0,y) - 25) \tag{6.75}$$

$$\frac{\partial T}{\partial x}|_{1,y} = 5(25 - T(1,y)) \tag{6.76}$$

$$\frac{\partial T}{\partial x}|_{x,0} = 5(T(x,0) - 25) \tag{6.77}$$

$$\frac{\partial T}{\partial x}|_{x,1} = 5(25 - T(x,1)) \tag{6.78}$$

Compute the temperature profile within the plate to assess if the alloy will melt or not. The alloy has a melting point of 800 ^{0}C. Comment on your results and propose a solution.

D - To reduce the heat within the plate it has been proposed to cut a square in inside the plate as represented in Fig. 6.7. This enable the to place a cooling fluid at the center of the plate to reduce its temperature.

$$\frac{\partial T}{\partial t} = \alpha \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right) \tag{6.79}$$

where T is the temperature in Kelvin, t the time in s and α the thermal conductivity m^2/s. In its normal condition, the fluid at the center canno be kept at room temperature and therefore the edge at the center of the piece are fixed at a temperature of T = 85 C while the outside is at T = 25 C. We consider here Dirichlet boundary conditions. You can solve this problem by considering only half a side of the hollow plate as represented in Fig. 6.7. Because of the symmetry of the problem we know that

$T_{1,6} = T_{2,5}$ $T_{2,7} = T_{3,6}$ $T_{3,7} = T_{4,7}$
$T_{2,2} = T_{2,0}$ $T_{3,2} = T_{3,0}$ $T_{4,2} = T_{4,0}$

Compute the temperature in the plate when the same electric current that in **C** flows through the plate (i.e $Q = 100$ kW/m^3 and $k = 16$ W/m.K).

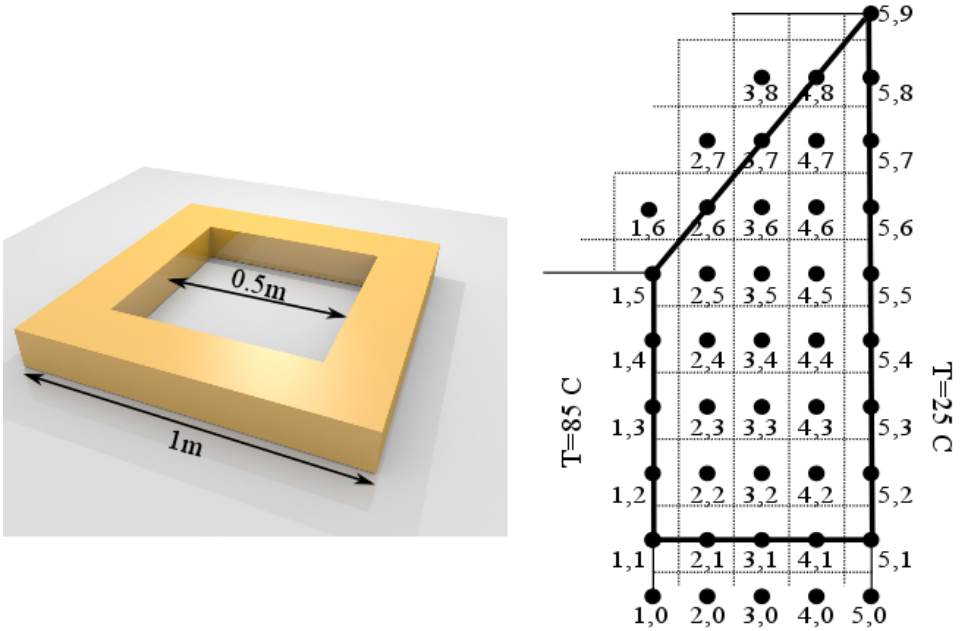


Figure 6.7: Right - Hollow square plate where the inside and outside edges are kept at 85 C and 25 C respectively. Left - Grid pattern used to solve this problem.

6.5.3 The shallow wave equation

The simulation of water wave propagation is most often done with the shallow wave equation (SWE) The SWE is a wave-like equations that relates the height of the waves (h)to their velocity in the x direction (u) and in the y direction (v). In its simplest form the set of equation reads:

$$\frac{\partial u}{\partial t} = -g\frac{\partial h}{\partial x} - bu \tag{6.80}$$
$$\frac{\partial v}{\partial t} = -g\frac{\partial h}{\partial y} - bv \tag{6.81}$$
$$\frac{\partial h}{\partial t} = -H(\frac{\partial u}{\partial x} + \frac{\partial h}{\partial y}) \tag{6.82}$$

where g is the acceleration due to gravity, b the viscous drag coefficient and H the mean height of the surface. These equations are used to simulate tsunamis, predict the weather etc .... We propose here to solve the SWE and visualize the wave propagation. We'll use Dirichlet condition for the velocities

$$u = 0 \qquad v = 0 \qquad \text{at the boundary of the domain} \tag{6.83}$$

and Neumann condition for the height

$$\frac{\partial h}{\partial x} = 0 \qquad \frac{\partial h}{\partial y} = 0 \qquad \text{at the boundary of the domain} \tag{6.84}$$

We'll use a regular grid ranging from 0 to 1 in both direction with 101 points in x and y direction with $\Delta x = \Delta y = 1E-2$. Use a time step of $\Delta t = \Delta x/100$. Use the values $g = 1$, $H = 1$ and $b = 0$. Different initial conditions can be used. We'll start by using a Gaussian initial height

$$h(x, y, t = 0) = 0.5\exp\left(-\frac{(x - x_0)^2}{\sigma}\right)\exp\left(-\frac{(y - y_0)^2}{\sigma}\right) \tag{6.85}$$

with $x_0 = y_0 = 0.5$ and $\sigma = 0.005$. The initial velocities are here given by $u(x, y, 0) = v(x, y, 0) = 0$. This defines a central water elevation with no initial momenta. Experiment with different initial condition. You can for example gives the wave an initial velocity by using $u(x, y, 0) = 0.5h(x, y, 0)$. Change the value of the drag coefficient b and comment on its impact on the wave propagation. You should obtain waves similar to the ones reported in Fig. 6.8.

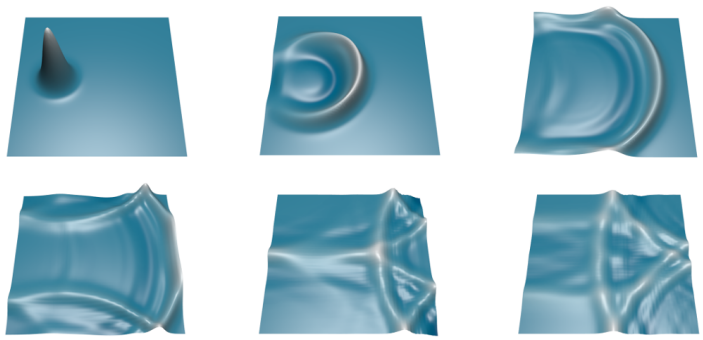


Figure 6.8: Evolution of the water surface with the shallow wave equation

Chapter 7
Monte Carlo Schemes & Molecular Dynamics Simulations

The previous chapter dealt with the calculation of physical quantities (heat, velocities) that were assumed to be continuous variables of the spatial direction. For example the velocity profile in a water channel was assumed to be a function of the lateral dimension of the pipe $v(z)$. In reality this profile is defined by the individual velocities of the water molecules that compose the fluid moving through the pipe. Similarly the heat-distribution calculated in the last chapter was assumed to be a continuous function in the $x-y$ plane. However at the microscopic scale the heat of a material is directly given by the motion of the atoms composing the system. We have therefore in the previous chapter calculated these various physical quantities over a large domain where the microscopic details are averaged.

In this chapter we focus on the description of matter at the microscopic scale and present two classes of techniques that allows to compute the values of different physical quantities of a many-particle system: molecular dynamics and Monte-Carlo simulations. The term 'particle' is here used loosely and can refer to individual atoms, molecules or clusters. Molecular dynamics simulations allows to follow in time the evolution of the system composed of interacting particles by solving the equation of motion. In contrast the Monte-Carlo schemes are based on random number and are therefore not deterministic simulations. Both methods make use of random number generators that we briefly discuss here first.

7.1 Random Number Generators

As their name stands for, random number generators produces a series of numbers that a randomly distributed according to a well defined distribution. The module *numpy.random* contains different generators. For example the function

```
1 r = numpy.random.rand()
```

create one random number in the open interval $[0,1[$. The generation of random number is done by deterministic algorithm and are therefore not really random. However good random number generators lack any patterns in the series of number generated and can be seen as truly random. It is sometimes useful when coding to obtain the exact same sequence of random number. This can be done by imposing a given seed to the generator with the function

```
1 r = numpy.random.seed(13)
```

The seed (here equals to 13) is an integer that entirely determine the full sequence of random number generated. For example when using

```
numpy.random.seed(13)           # seed the generator
r1 = numpy.random.rand(10)   # creates 10 random numbers
r2 = numpy.random.rand(10)   # creates another 10 random numbers

numpy.random.seed(13)             # re-seed the generator
r3 = numpy.random.rand(10)   # creates 10 random numbers
```

$tr1$ and $r3$ will be identical. If you do not specify a seed, Python will create one for you based on the current time. Hence each time you ll execute a program you'll obtain a different series of random numbers.

Different distribution have been implemented in numpy. We'll use here only the uniform distribution introduced above and the normal (Gaussian) distribution

$$p(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \tag{7.1}$$

A series of random numbers distributed according to a normal distribution can easily be generated using the function

```
X = numpy.random.randn(100)
```

that draws here 100 random numbers following a normal distribution. We can visualize and compare the uniform and normal distribution with the snippet of code below:

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

N = 1E4
XYu = np.random.rand(N,2)
XYn = np.random.randn(N,2)

plt.plot(XYu[:,0],XYu[:,1],'o',color='#007FFF')
plt.hist(XYu[:,0],20,color='#007FFF')

plt.plot(XYn[:,0],XYn[:,1],'o',color='#007FFF')
plt.hist(XYn[:,0],20,color='#007FFF')
```

This code generates two arrays containing the position of $N = 1E4$ points that are distributed either uniformly of normally in the x, y plane. This points are visualized both by plotting them directly and by computing the histogram of their x coordinate. The results of this simple code is show in Fig. 7.1.

Other distribution can be useful for example the command

```
X = numpy.random.uniform(min,max,N)
```

creates N points randomly distributed between min and max instead of 0 and 1 for the command $random.rand()$. Similarly the command

```
X = numpy.random.normal(loc = mu, scale = sigma, N=100)
```

creates N points that are distributed according to a normal distribution centered around loc and with a deviation controlled by $scale$. This command corresponds to the distribution

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{7.2}$$

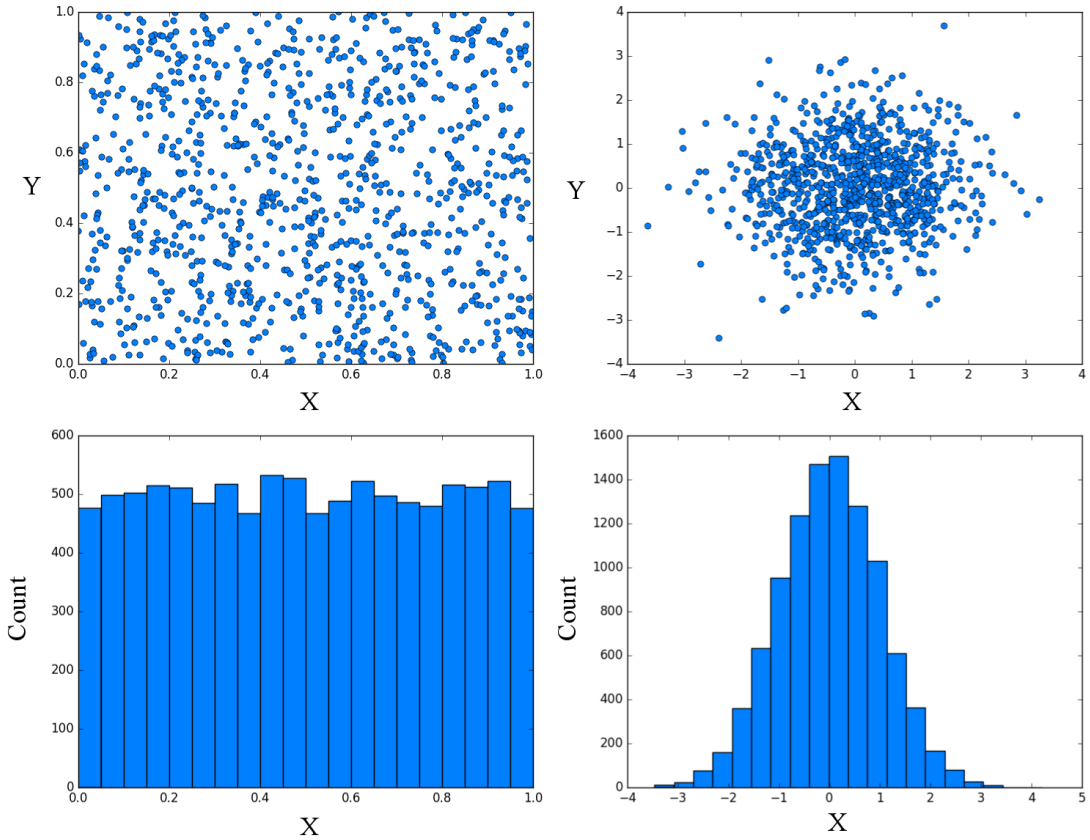


Figure 7.1: Random number generators. Top panels show the position of the points in the $x - y$ plane. A clear distinction can be seen between the two distribution. The bottom panels show the histogram of the x coordinate of the points.

Finally a random series of integers can be generated with the commands

```
X = numpy.random.random_integers(low,high,N)
```

This command creates N random integers in the interval $[low, high]$. A similar command

```
X = numpy.random.randint(low,high,N)
```

also creates N random integers but in the interval $[low, high[$ (i.e. the high value is never generated). In addition to the random generators contained in *numpy* the module *scipy.stats* gives access to a large number of distribution that are useful for physical system. For example the commands

```
from scipy.stats import maxwell
r = maxwell.rvs(size=1000)
```

will create 1000 random numbers following a Maxwell distribution that can be used to sample the velocities of atoms in a gas.

7.2 Monte-Carlo Integration

One of the first use of the random generator was the calculation of integrals, which might sounds funny as such calculation is purely deterministic. We'll see here two solution to compute definite integrals with the help of random numbers

7.2.1 Standard Monte-Carlo Integration

The standard Monte-Carlo integration proposed to evaluate the integral

$$I = \int_a^b f(x)dx \tag{7.3}$$

by the numerical sum:

$$I = \frac{b-a}{N}\sum_{i=1}^{N} f(x_i) \tag{7.4}$$

where the points x_i are chosen randomly instead of using a regular grid. As surprising as it is, this approach is much more accurate for high-dimension integrals than the methods we've seen in Chapter 4.

To illustrate Monte-Carlo integration let's compute the integral $\int_1^2 (2+3x)dx$ whose exact value is 6.5. The little code below does just that

```
import numpy as np

def f(x):
        return 2.+3.*x

def MCInt(a,b,f,N):
        X = np.random.uniform(a,b,N)
        I = (float(b-a)/N) * np.sum(f(X))
        return I

a,b,N = 1.,2.,1E3
```

```
I = MCInt(a,b,f,N)
print I
```

By executing this code we find a value of $\approx 6.5 \pm 0.05$ with only 100 points. Of course the exact value change each time we execute the program as the sampling point are chosen randomly. We can add more points to get a better estimate but for such simple integrals the Monte-Carlo integration converges slowly and 1E6 points are required to get an error below 10^{-3}. Therefore for simple integrals the methods introduce din Chapter 4 are much more efficient. However for functions of many variables the Monte-Carlo integration clearly outperform any other methods.

7.2.2 Computing areas by throwing some darts

Another approach of Monte-Carlo Integration is illustrated n Fig. 7.2. In this approach N points are randomly generated in a square where the function $f(x)$ is defined. N_0 of these points will fall below the curve. We know the area of the square $A = y_0 * (b - a)$. We can estimate the value of the integral by the fraction of points below the curve times the total area

$$y = \int_a^b f(x)dx \simeq \frac{N_0}{N} \times [y_0(b-a)] \tag{7.5}$$

The snippet of code below uses this method to compute the integral

$$I = \int_0^4 4 + \sin(2x)\exp(x/2) \tag{7.6}$$

```
import numpy as np

def f(x):
        return 4 + np.sin(2*x)*np.exp(0.5*x)

def MCInt(a,b,y0,f,N):
        X = np.random.uniform(a,b,N)
        Y = np.random.uniform(0,y0,N)
        N0 = Y[Y<=f(X)].size
        return float(N0)/N*y0*(b-a)

a,b,y0,N = 0.,4.,15.,1E4
I = MCInt(a,b,y0,f,N)
```

For 100 points this approach leads to $I \simeq 17.9$ where the quadrature method implemented in *scipy* leads to $I = 17.8365$. You will use this method in one of the exercise of this session to determine the value of π.

7.3 Random walks: Direct Monte-Carlo

As the name stands for a random walker is a particle whose motion is dictated by random forces. Hence at each time step the particle can change direction randomly to continue its path. Such motion is very useful to describe systems whose dynamics is not entirely deterministic and where randomness plays a crucial role. This is for example the case of the motion of particle molecules in liquid or gas phase, the motion of a particle in dust cloud, or even the evolution of stock prices.

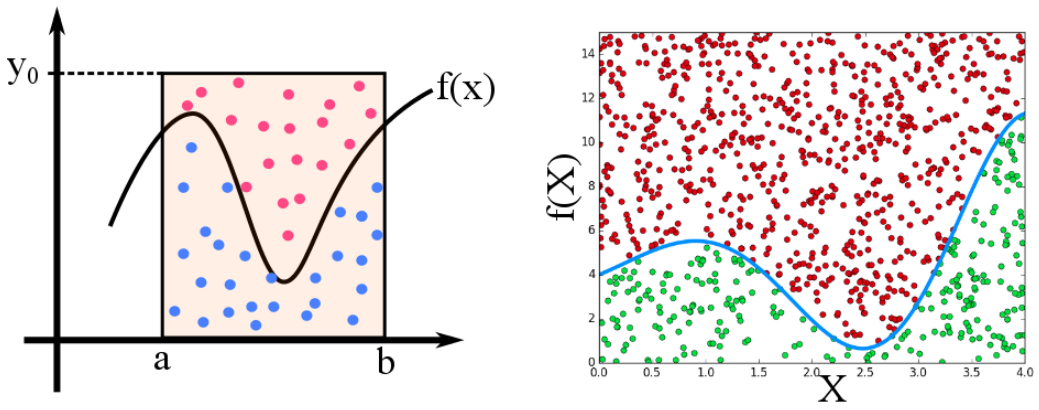


Figure 7.2: Illustration of the hit or miss monte-carlo integration of the function $f(x)$, represented by the solid line, between a and b. N random points are generated and we count the ones that fall below the curve.

7.3.1 Random Walk in 1D

The simulation of a random walk in 1D is then straightforward using the uniform generator. Starting from $x = 0$, we can generate a random number ϵ ranging between 0 and 1. The walker then move to the left if $\epsilon < 0.5$ and to the right if $\epsilon > 0.5$. We can let the walker walk for a while to see where he's going. Of course if we restart the walk we will obtain a different walk as each step is chosen randomly. Hence it is very important to average random walk (and in general stochastic processes) to obtain meaningful information. The snippet of code below show how to implement a 1D random walk and average the distance walked by the walker to extract meaningful quantities. The results are shown in Fig. 7.3.

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


# number of walker
M = 1000

# size and number of step
dX, NT = 2.,1000

# positions
X = np.zeros((NT,M))

# loop over time
for iT in range(1,NT):

        # generate random number
        EPS = np.random.rand(M)

        # propagate the walkers
        X[iT,EPS<=0.5] = X[iT-1,EPS<=0.5] - dX
```

```
        X[iT,EPS>0.5] = X[iT-1,EPS>0.5] + dX


#plt.plot(np.arange(NT),X)
#plt.show()

plt.plot(np.arange(NT),np.sqrt(np.mean(X**2,1)),color='blue',linewidth=2)
plt.plot(np.arange(NT),np.mean(X**1,1),color='black',linewidth=2)
plt.plot(np.arange(NT),np.sqrt(np.linspace(0.,NT,NT))*dX,color='#007FFF',linewidth=2)
plt.plot(np.arange(NT),np.zeros(M),color='#5F5F6F',linewidth=2)
plt.show()
```

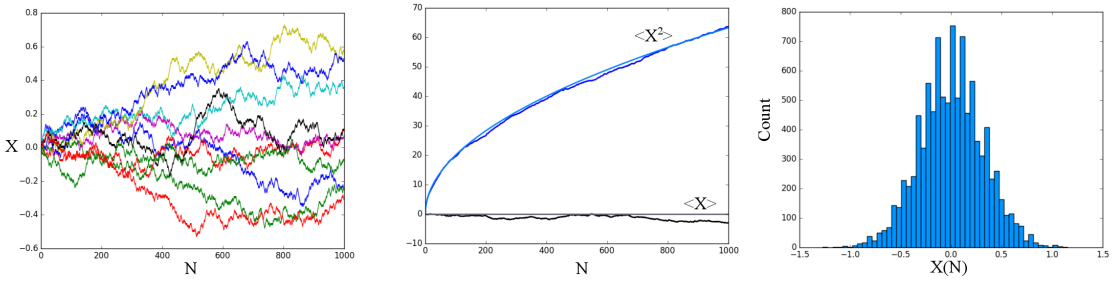


Figure 7.3: Evolution of random walker in 1D. Left - 10 examples of random walk Middle - Average value of the position and displacement of 1000 walkers. Right - Distribution of the final position of 1000 walkers

As we can see on the first panel of Fig. 7.3, every walker take a very different trajectory than than the other ones. Some go left (negative position), some right (positive values). However the mean value of the position ($< X >$) is null over time since the probability to go left equals the one to go right. However it can be shown that the average of the square distance, i.e. $< X^2 >$ increase linearly with the number of steps N. Taking the square root of $< X^2 >$ we obtain the root-mean-squared distance which represent the average positive distance away from the 0. We easily see that this distance increases with the square-root of N:

$$\sqrt{< X^2 >}(N) = \sqrt{N}\Delta X \tag{7.7}$$

As we can see our numerical simulations here with 1000 walkers gives agrees very well with this theoretical results. Running the simulation with even more walkers would lead to an even better agreement. Finally we can note from the last panel that the distribution of final positions follows a Gaussian distribution centered around 0. That also agree perfectly with analytical analysis that are however beyond the scope of this chapter.

7.3.2 Random walk as a diffusion equation

The random walk presented here can be directly related to PDEs as the ones studied in the previous sections. Let's assume that the walker arrives at the positions m after $N+1$ steps. This implies that at time N the walker either came from the position $m-1$ with a probability p or from $m+1$ with a probability q. We can therefore write

$$P(m, N+1) = p \times P(m-1, N) + q \times P(m+1, N) \tag{7.8}$$

In our case $p = q = 0.5$ as the walker is truly random. In the equation above $P(m, N)$ means that at time N the walker is at the position m. We can subtract $P(m, N)$ from both side, divide by Δt and introduce a few Δx to obtain

$$\frac{P(m, N+1) - P(m, N)}{\Delta t} = -p\frac{\Delta x^2}{\Delta t}\frac{P(m, N) - P(m-1, N)}{\Delta x^2} + q\frac{\Delta x^2}{\Delta t}\frac{P(m+1, N) - P(m, N)}{\Delta x^2} \tag{7.9}$$

If we now take $p = q = \alpha$ and we gather all the terms together we obtain

$$\frac{P(m, N+1) - P(m, N)}{\Delta t} = \alpha\frac{\Delta x^2}{\Delta t}\frac{P(m+1, N) - 2P(m, N) + P(m-1, N)}{\Delta x^2} \tag{7.10}$$

We recognize here the finite difference expression of the partial derivative and can therefore write

$$\frac{\partial}{\partial t}P(x, t) = D\frac{\partial^2}{\partial x^2}P(x, t) \tag{7.11}$$

We therefore arrive to the diffusion equation studied in chapter 6 for the diffusion of drug through the body of a patient. The parameter D is here called the diffusion coefficient. The random walk techniques constitutes therefore a statistical approach of solving the diffusion equation by tracking the trajectory of a collection of 'particles'. The more particle we follow the better the agreement with the continuous diffusion equation.

7.3.3 Example: Evolution of stock prices

1D Random walks play a central role in many fields including finance. A common mathematical mode for the evolution of stock prices is given by the equation

$$x_n = x_{n-1} + \Delta t \mu x_{n-1} + \sigma x_{n-1} r_{n-1}\sqrt{\Delta t} \tag{7.12}$$

where x_n is the stock price at time t_n, Δt the time interval, μ is the growth rate of the stock price, σ its volatility and $r_0, r_1, \ldots r_{n-1}$ are normally distributed random numbers with means 0 and unit standard deviation. A common technique to evaluate the expected price of the stock is to simulate N realization of the equation above providing x_0, μ, σ and Δt. A few realizations, simulated with $x_0 = 100$, $\Delta T = 0.1$, $\mu = \sigma = 0.01$, are shown in Fig. 7.4.

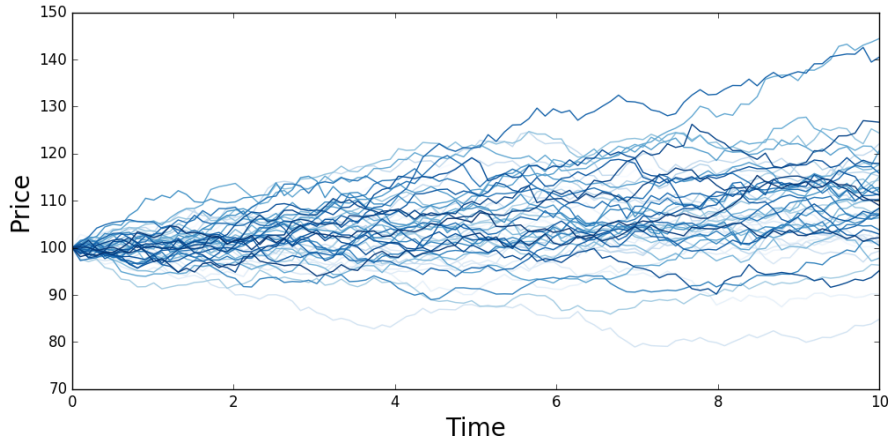


Figure 7.4: Evolution of the stock price following eq. 7.12.

From these realizations alone it is rather hard to make any prediction. Running a large number of these simulations allows to get a better idea of the price evolution. Three cases are reported in Fig. 7.5. As you can see there depending on the growth rate and volatility it is a more or less good idea to invest in the stock.

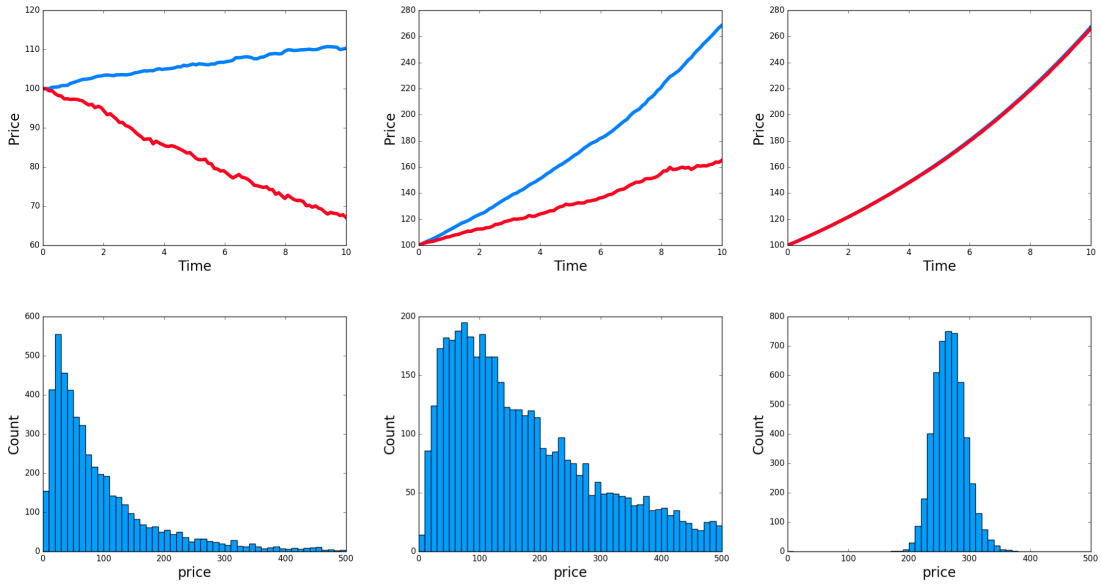


Figure 7.5: Top Panels : Mean (blue) and median (red) value of the stock price. Bottom Panel : Distribution of the final stock price. Three cases are shown: Left $\mu = 0.01$, $\sigma = 0.1$; Middle $\mu = 0.1$, $\sigma = 0.1$; Right $\mu = 0.1$, $\sigma = 0.01$.

7.3.4 Random Walk in 2D

Random walks can be generalized to the 2D. The process is very similar to the 1D cases except that each walker can not only go left or right but also up or down. These four directions are equiprobable with a probability of $1/4$. As an example one trajectory representing the random walk of 5 different particles is represented in Fig. 7.6. During one of the exercises of the session you will implement yourself a 2D random walk process and study the properties of this particular system.

7.4 Metropolis Monte-Carlo

Consider the the 2D random walk presented above. During the simulation the motion of each particle is uncorrelated with those of the others particles. The probability for one particle to move in a given direction is independent of the position of the other ones. As a consequence all the possible configurations of N particles are equiprobable. This is of course not the case when the particles interact with each other. In presence of interactions the particle will tend to adopt preferential configurations that minimize the potential energy of the entire system. The simulation of such system can be done with the Metropolis Monte-Carlo algorithm presented here.

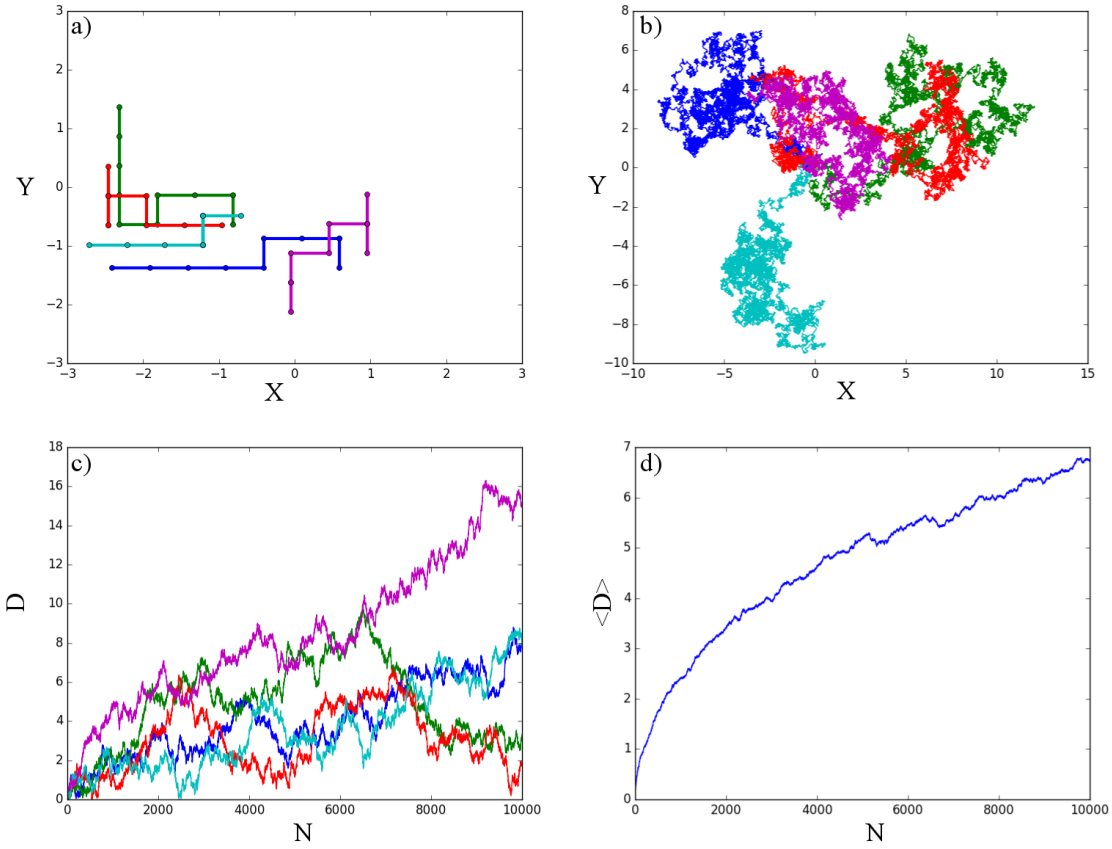


Figure 7.6: 2D random walks. a) Ten steps for 5 walkers on a regular grid. b) Many more small steps. c) Evolution of the distance walked by 5 walker. d) Average distance walked by 100 walkers.

7.4.1 Lattice Metropolis

To illustrate the metropolis algorithm let's consider two types of 'atoms' labeled A and B and that can diffuse on a rigid 2D lattice as represented in Fig. 7.7. To be clear each atoms can switch its position with one of its neighbor. If the the atom switches position with a vacancy the move is equivalent to a diffusion. Otherwise we have exchange between tow atoms. We assume that each atom only interacts with its nearest neighbor. Furthermore we assume that each A atoms prefers to have B neighbors and inversely. We therefore write the potential interaction

$$U_{AA} = U_{BB} = E_{\text{rep}} \qquad U_{AB} = E_{\text{att}} \tag{7.13}$$

For a given configuration of the atoms the total potential energy of the system can be obtained by summing up all the interactions between the pairs of atoms. We assume that in their initial configuration the atoms A and B occupied each half of the domain as represented in Fig. 7.7. We want to obtain the equilibrium configuration for a given temperature T. The metropolis algorithm then proceed as follows:

(1) Consider one particle with a randomly chosen position on the lattice i, j

(2) Randomly chose one move for the atom at i, j and perform that move

(3) Compute the energy change ΔE that is induced by this trial movement

(4) if $\Delta E < 0$ the move is accepted and we return to **(2)**

(5) if $\Delta E > 0$, a random number r is generated between 0 and 1.

(6) if $r < \exp(-\Delta E / k_B T)$, the move is accepted. Otherwise the move is rejected. In both case we return to **(2)**.

This very simple algorithm ensure that the final configuration (providing enough Monte-Carlo moves have been performed) respect a Boltzman distribution, i.e. is in equilibrium with the temperature imposed to the system. We have reported in Fig. 7.7 two final configurations obtained after 10000 moves for two different temperature. During the simulations we have set $E_{\text{rep}} = -E_{\text{att}} = 1.0$. The thermal energy was then $k_B T = 0.5$ (low temperature) and $k_B T = 10.0$ (high temperature). As you can see on the corresponding configuration the a very ordered phase is obtained at low temperature as a very small number of unfavorable moves (i.e. move that increases the total potential energy) are performed. On the contrary at high temperature many unfavorable move are performed resulting in a much more disordered phase.

7.4.2 Off-lattice Metropolis

The Metropolis algorithm can be generalized to the case where the particles are not fixed to a specific grid. To illustrate that method, let's consider an ensemble of identical particles that are allowed to move in a two dimensional space. We assume that the particles interact with each other via a Lennard-Jones potential

$$U(r) = 4\epsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^{6} \right] \tag{7.14}$$

where r is the distance between two particles, ϵ the interaction strength and σ the optimum distance between the particle. We want to simulate how these particles aggregates with each other. The Metropolis Monte Carlo method allows finding the solution of this problem for a large number of particles and at finite temperature. The algorithm proceeds as follows:

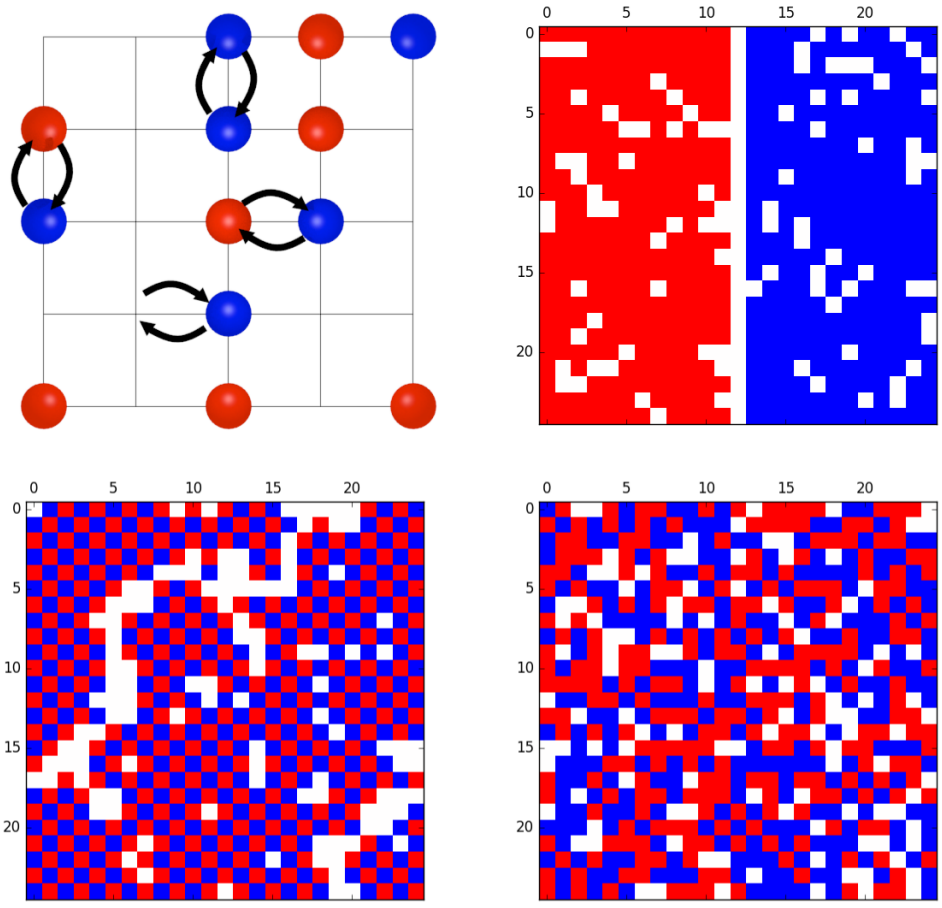


Figure 7.7: Lattice Monte-Carlo for atomic diffusion with two types of atoms (A red and B blue). Top-Left: Illustration of the possible moves. Top-Right: Initial configuration of the atoms. Bottom-Left: Configuration obtained for 10000 Monte-Carlo moves and at low temperature. Bottom-Right: Configuration obtained for 10000 Monte-Carlo moves and at high temperature.

(1) Generate an initial configuration of N particles distributed in the 2D space. To avoid difficulty when particles are very close and therefore U is very large, the particles are usually distributed along a regular mesh.

(2) Consider one particle with a randomly chosen index i and randomly change its position to $\mathbf{r'}_i = \mathbf{r}_i + \delta$

(3) Compute the energy change ΔE that is induced by this trial movement

(4) if $\Delta E < 0$ the move is accepted and we return to **(2)**

(5) if $\Delta E > 0$, a random number r is generated between 0 and 1.

(6) if $r < \exp(-\Delta E/k_B T)$, the move is accepted. Otherwise the move is rejected. In both case we return to **(2)**.

As you can see the algorithm is almost identical to the previous case. The principal difficulties arises in the calculation of the total energy that can be extremely time consuming if all the pair wise interaction are calculated at each Monte-Carlo moves. Efficient tricks have been developed to avoid that computational load and are presented in the next section in the framework of molecular dynamics simulations.

7.5 Molecular Dynamics Simulations

In the random walk presented above the 'time' is not a real time as the simulations is not deterministic. At the difference of Monte Carlo simulation,s molecular dynamics simulations propose to simulate the dynamics of a system of N particles in real time following deterministic equations. Molecular Dynamics (MD) simulations examine the dynamics of a system composed of N interacting particles such as the one represented in Fig. 7.8a. We assume that the interaction forces between the particle can be written as a sum over the particle pairs $F_{ij}(r_{ij})$ that depends on the distance between the particles. Hence the total force on the i-th atoms is given by

$$\mathbf{F}_i = \sum_{j=1, j\neq i}^{N} F_{ij}(r_{ij}) \tag{7.15}$$

External forces such as gravity can also be included in the simulations if necessary. Ignoring these forces for the moment, the total force F_i can be used to write the equation of motion for the i-th particle

$$\frac{d^2\mathbf{r}_i(t)}{dt^2} = \frac{\mathbf{F}_i}{m_i} \tag{7.16}$$

where m_i is the mass of the i-th particle and $\mathbf{r}_i$ a vector defining its position. Molecular dynamics refers then to technique in which these equations are solved numerically for a large number of particles. The main restrictions require to use MD simulations is that the forces on the different particle must be know and well defined by classical approximations. In addition we can only run MD simulations for a finite number of particle and finite evolution time. Hence the simulations of macroscopic material is prohibited by the computational time.

7.5.1 Periodic boundary conditions

To overcome this intrinsic limitation of computer programs MD simulation usually use the so-called periodic boundary conditions (PBC). In PBC, the system of interest is surrounded by fictitious system that are in the exact same configurations. These conditions are represented in Fig. 7.8.b. As a consequence each particle in the system of interest not only interact with the the other particles in this system but also with all the other particles in the neighboring cells. The interaction then reads

$$\mathbf{F}_i = \sum_{j=1, j\neq i}^{N} F_{ij}(r_{ij}) + \sum_{\text{copies}} \sum_{j=1}^{N} F_{ij}(r_{i,j_{\text{copies}}}) \tag{7.17}$$

The calculation of the sum over the infinite sum of copies can be extremely time consuming. However the interactions between the particles are usually short ranged and therefore accounting only for the first nearest-neighbor cells is usually sufficient. We wnat to stress that we do not wish to physically copy each particle in their respective copies. If we have N particles in the system of interest physically copying each particle will lead to $27 \times N$ only with the nearest neighboring copies. Since each particle and its copies are identical and only translated in one direction it is enough to compute the dynamics of the N original particles and account for the extra terms in the definition of the force. This also means that when one particle leaves the original cell on one side it must be reintroduced at the opposite side.

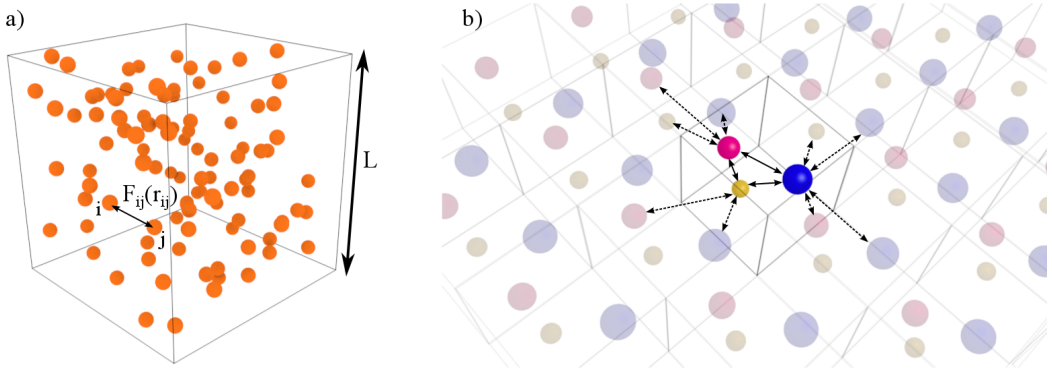


Figure 7.8: a) Representation of a system of N particles in a cubic box of size $L \times L \times L$. The particle i and j interact with each other via a force $F_{ij}(r_{ij})$. b) Periodic boundary conditions for a system of 3 particles. The principal system is surrounded by copies that are here semi-transparent. A few of the forces between the principal system and the copies are shown with dashed lines

7.5.2 Force calculations

Even when only accounting for a reduced number of copies for each particle, the calculation of the forces between the particle can be extremely time consuming. As already stated, most interactions are short range and are therefore negligible after a certain distance between particle pairs. We can therefore define a cutoff r_{cutoff} such as if the distance between two particles is larger than r_{cutoff} we do set these forces to 0. We can then limit the number of forces to be calculated. To do so we can keep a list of particle whose separation is smaller than a predefined distance r_{max} and update this list every 10 or 20 steps of the dynamics. The radius r_{max} must be larger than r_{cutoff} and chosen such that between two update of the list, its unlikely for a pair not in the list to become closer than r_{cutoff} . This is illustrated in Fig. 7.9 for the interactions between the orange particle with the central blue particle. This procedure allows computing only a small fraction of the different inter particle interactions without reducing the accuracy of the simulations.

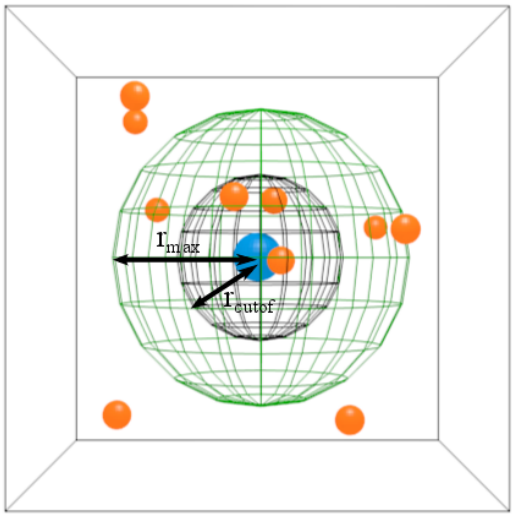


Figure 7.9: Definition of the cutoff and maximum distance for the force calculations. The interactions with the blue particle are only calculated for the particle within a sphere of radius $r_{\max}$. However for all the particles that are further away than r_{cutoff} from the blue one than these interactions are set to 0.

7.5.3 The Verlet algorithm: MD at constant energy

The Verlet algorithm is used most often to integrate the equation of motion during MD simulations. We describe here the most popular version of the algorithm i.e. the velocity-Verlet algorithm. Other approach such as the $leap-frog$ are also possibl ebut generally less stable numerically. The Verlet algorithm is easily derived by expanding the coordinate in a Taylor expansion at $t \pm h$ around t leading to

$$\mathbf{r}(t+h) = \mathbf{r}(t) + h\frac{d\mathbf{r}(t)}{dt} + \frac{h^2}{2}\frac{d^2\mathbf{r}(t)}{dt^2} + \frac{h^3}{6}\frac{d^3\mathbf{r}(t)}{dt^3} + ... \tag{7.18}$$

$$\mathbf{r}(t-h) = \mathbf{r}(t) - h\frac{d\mathbf{r}(t)}{dt} + \frac{h^2}{2}\frac{d^2\mathbf{r}(t)}{dt^2} - \frac{h^3}{6}\frac{d^3\mathbf{r}(t)}{dt^3} - ... \tag{7.19}$$

By adding these two equations one obtains the Verlet equation

$$\mathbf{r}(t+h) = 2\mathbf{r}(t) - \mathbf{r}(t-h) + h^2\mathbf{F}(t) \tag{7.20}$$

where we have used the identity

$$\frac{d^2\mathbf{r}(t)}{dt^2} = \mathbf{F}(t) \tag{7.21}$$

where we have set $m = 1$. The Verlet algorithm allows predicting the positions $\mathbf{r}(t+h)$ knowing $\mathbf{r}(t)$ and $\mathbf{r}(t-h)$. However at the start of the simulation we only know $\mathbf{r}(0)$ and the corresponding velocities $\mathbf{v}(0)$. To obtain the positions and velocities at $t = h$ we use the relation

$$\mathbf{r}(h) = \mathbf{r}(0) + h\mathbf{v}(0) + \frac{h^2}{2}\mathbf{F}(t=0) \tag{7.22}$$

Using eq. 7.20 requires to store the position at two different times, which can be memory demand-

ing and introduce numerical instabilities. However we can use the definition of the velocities

$$\mathbf{v}(t) = \frac{\mathbf{r}(t+h) - \mathbf{r}(t-h)}{2h} \tag{7.23}$$

to overcome this issue. Using this equation we can write

$$\begin{aligned} \mathbf{r}(t+h) &= \mathbf{r}(t) + h\mathbf{v}(t) + \frac{h^2}{2}\mathbf{F}(t) & (7.24)\\ \mathbf{v}(t+h) &= \mathbf{v}(t) + h[\mathbf{F}(t) + \mathbf{F}(t+h)]/2 & (7.25) \end{aligned}$$

We do not need to store two positions array but we do need now to store the values of the forces at two times. To circumvent this we can evaluate the different quantities (velocities, positions and forces) following the algorithm:

$$\tilde{\mathbf{v}}(t) = \mathbf{v}(t) + {}^{h}/_{2}\mathbf{F}(t) \tag{7.26}$$

$$\mathbf{r}(t+h) = \mathbf{r}(t) + h\tilde{\mathbf{v}}(t) \tag{7.27}$$

$$\mathbf{v}(t+h) = \tilde{\mathbf{v}}(t) + {}^{h}/_{2}\mathbf{F}(t+h) \tag{7.28}$$

These equations define the velocity-Verlet and allows avoiding multiple copies of the velocities, positions of forces at different time steps.

7.5.4 Constant temperature simulations: velocity rescaling

The simulation described so far is done at constant energy. The total energy of the system given by the initial positions and velocities of the system, is conserved through the dynamics. In experimental situation the total energy is not often well conserved but the temperature is. It is therefore interesting to develop MD algorithm that are able to perform constant temperature calculations. A simple approach to impose a constant temperature is to rescale the velocities of the particle according to

$$\mathbf{v}_i(t) \longrightarrow \lambda \mathbf{v}_i(t) \tag{7.29}$$

for all the particles $i = 1, ...N$. The rescaling factor is chosen so that the temperature of the system will converge toward the desired temperature T_D. This leads to :

$$\lambda = \sqrt{\frac{(N-1)3k_B T_D}{\sum_{i=1}^{N} m v_i^2}} \tag{7.30}$$

This rescaling is performed after every integration steps until the desired temperature is obtained. More elaborated methods, for example based on frictional forces, have been developed to obtain a constant temperature simulation but are beyond the scope of this introduction.

7.5.5 Example: Argon gas

To illustrate molecular dynamics solution we present here the simulation of an argon gas. The interactions between particles can be model with high accuracy by a Lennard-Jones potential

$$U(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \tag{7.31}$$

with the values $\epsilon/k_B = 119.8$ K and $\sigma = 3.405$ Å. Using this potential we want to simulate the evolution of N particles in a box of size $L \times L \times L$. We therefore have to defined the initial positions and velocities of these N particles. We could define these quantities by hand but it is more practical to use random

number generator. Similarly if we want the initial velocities to follow a Maxwell distribution we can use the module *scipy.stats.maxwell*. The snippet of code below shows how to generate the momentum of the particles according to a Maxwell distribution. However the velocities can have different (and random) direction in space.

```
import scipy.stats as stats
import scipy.constants
import numpy as np

kB = scipy.constants.k      # boltzman constant
T = 300.0                    # temperature
m = 6.6*1E-26               # mass of argon
a = np.sqrt(kB*T/m)

maxwell = stats.maxwell
P = maxwell.rvs(loc=0,scale=a,size=(100))
```

The random number generator will also play an important role for the Monte Carlo simulations. Writing a MD code for argon can easily be done following the procedure presented here. This is the focus of the first exercise of this chapter. The positions of the atoms can be exported at each time step to visualize the trajectory of the simulations. Three snapshots of such an MD calculations are represented in Fig. 7.10.

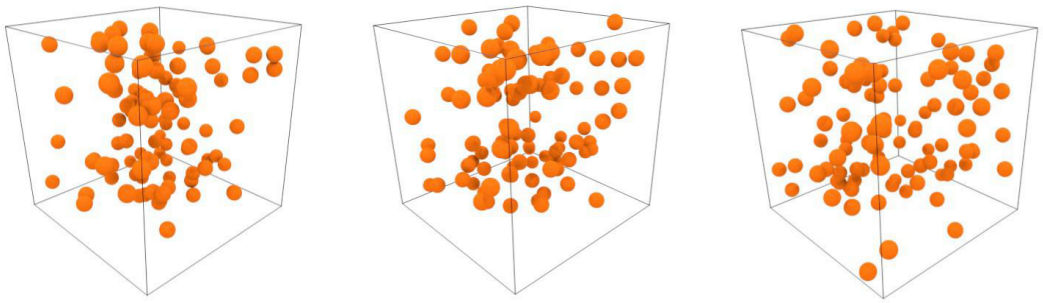


Figure 7.10: Snapshots of a molecular dynamics simulation of 100 argon atoms in a 100 $Å^3$ box with periodic boundary conditions.

7.5.6 Computing Physical quantities

Once the desired temperature is obtained (i.e. the system is in equilibrium) one can compute physical quantities from the positions and velocities of the particles. These quantities are calculated as time average

$$\overline{A} = \frac{1}{n}\sum_{\nu=0}^{N} A_\nu \tag{7.32}$$

where ν is the time step index, and n the total number of time step performed after the equilibrium was reached. It is for example important to keep track of the evolution of the kinetic and potential

energy as well as the virial given by:

$$E_{\text{Kin}} = \sum_i m_i v_i^2 \qquad E_{\text{Pot}} = \sum_{i,j>i} U(r_i j) \qquad \Lambda = \sum_{ij} = r_{ij} F(r_{ij}) \tag{7.33}$$

Note that the calculation of the potential energy is here limited to the particle pair whose separation is smaller that r_{cutoff}. To improve the calculation of the E_{Pot} one can use the the pair correlation function $g(r)$ To determine $g(r)$ we must keep track of the number of particle pair, $n(i\Delta r)$, whose separation lies between $i\Delta r$ and $(i+1)\Delta r$. Doing these for all possible values of i up to a certain limit gives an histogram $n(r)$ as a function of their separation r. The correlation function is then given by

$$g(r) = \frac{2V}{N(N-1)} \left(\frac{\overline{n(r)}}{4\pi r^2 \Delta r} \right) \tag{7.34}$$

It can been shown that a more accurate value of the potential energy can be obtained via

$$\overline{E}_{\text{Pot}} = (\overline{E}_{\text{Pot}})_{\text{cutoff}} + 2\pi \frac{N(N-1)}{V} \int_{r_{\text{cutoff}}}^{\infty} r^2 dr U(r) g(r) \tag{7.35}$$

Macroscopic quantities can also been calculated from the MD simulations. If all our simulations have been performed at constant volume, the pressure and temperature of the system are however fluctuating. These quantities can be obtained via:

$$T = \frac{2}{3} \frac{1}{k_B} \overline{1/2 m v^2} \qquad \frac{\beta P}{n} = 1 - \frac{\beta}{3N} \sum_{i=1}^{N} \mathbf{r}_i \nabla_i E_{\text{pot}} \tag{7.36}$$

7.5.7 MD for molecules

We have seen here how to perform MD simulations for atomic systems. MD simulations can of course be performed for molecular systems. However in this case one has to account of the internal degrees of freedom of the molecule such as the stretching of the chemical bonds of the bending of diehdral angles etc ... An illustration of these different terms is represented in Fig. 7.11. The different terms in the definition of the energy takes simple funcitonal form. For example the stretching energy between two atoms is usually given by

$$U_{\text{stretch}} = \frac{1}{2} K_s (r_{\alpha\beta} - r_0)^2 \tag{7.37}$$

This equations replace the complex chemical interactions that bind two atoms together by a simple classical spring with a spring constant K_b and an relaxed length of r_0. Similar expressions exists for the bond bending between three atoms the dihedral rotation between 4 atoms and so on. One exercise below proposes to use this approach to compute the vibration modes of a simple C_2H_4 molecule and visualize them.

7.6 Exercises : Algorithm

7.6.1 Approximating π with Monte-Carlo

In this first exercise, we propose to evaluate π with the help of a MonteCarlo technique. Consider the figure 7.12 where a circle of radius r is embedded in a square of length $2r$. The circle and the square have an area of πr^2 and $4r^2$ respectively. The ratio between the two is therefore of $p = \pi/4$. Let's focus on the upper right quarter of the figure. To evaluate pi, generate a series of points (x,y)

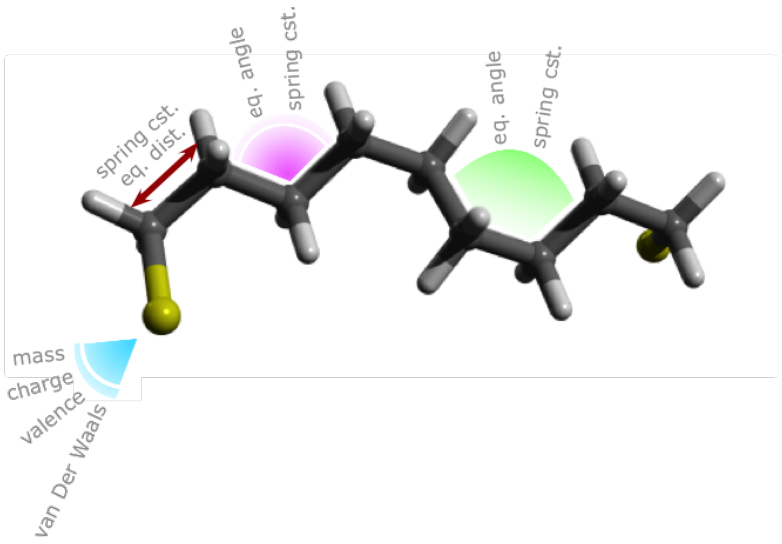


Figure 7.11: Illustration of the different terms present in the force field required for MD simulations on molecules.

randomly distributed (with an uniform distribution) in the upper right quarter of the figure. For each point decide if it belongs to the circle, i.e. $x^2 + y^2 \leq r^2$. The ratio of points inside the circle to all the points generated during the procedure should equal $\pi/4$. Compare the accuracy of this approach to the case where a regular mesh of points is used. Make sure to use the same number of points in both calculations.

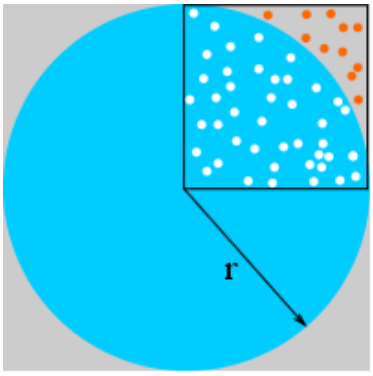


Figure 7.12: Integration with the Monte Carlo technique. One can determine the value of π using random numbers

7.6.2 Mixing of particles

We here consider a box divided in two equal size part by a wall. One half of the box contains N particles that are uniformly distributed in a random fashion. We now remove suddenly the wall and we want to study how the molecule fills in the entire box. We can simulate this process using a 2D random walk in a box. We set the dimension of the box to be $L_X = 1$ ad $L_Y = 1$ for simplicity.

To initiate the simulations you will randomly place N particles (N=100 is a good way to start) in the region defined by the boundary $[0; 1/2] \times [0; 1]$. Implement a 2D random walk dynamics to simulate

the trajectory of the particles. The particles cannot escape from the box. Hence a particle at one edge of the domains cannot go in all 4 directions. Visualize the results as an animation.

7.6.3 Stock Price

You've been hired as a consultant to guide client in their investment choices. One of your client wants to invest $1 million but hesitates between the companies listed in the table below.

	x_0	μ	σ
A	100	0.5	2.0
B	10	0.01	0.05
C	25	0.25	1

Simulate the evolution of the stock price using eq. 7.12 for a 1 year period and assuming an interval Δt of 1 hour. Advise your client on which stock to buy. Justify your answer.

7.7 Exercises : Application

7.7.1 Metropolis Monte-Carlo of a gas mixture

During this exercise you will perform simple Metropolis Monte Carlo simulations similar to the ones presented in section 7.4.1. Writing such code is technical and we therefore give you a module *MonteCarloModule.py* that contains most of the routines required to run such simulation. You just need to complete the loop inside the function *main*() that iterate over the Monte-Carlo moves. Complete that function following the steps given in section 7.4.1 and run some simulation for different temperatures and initial conditions.

7.7.2 Molecular dynamics of an argon gas

During this exercise you will perform simple molecular dynamics simulation of an argon gas. Writing MD code, even rudimentary is time consuming a technical. We therefore give you with all the routines required to run such simulation. You can find these functions in the module *molecularDynamicsModule.py*. All the functions are complete but one: the function called *velocityVerlet()* that update the positions and velocities of the atoms according to the velocity Verlet algorithm. Finish writing this function and run some MD simulations.

The program should write a file called *trajectory.xyz* that contains the positions of all the atoms during the MD simulations. This file can be opened with different module. We will use here Jmol. Visualize the trajectories obtained for different temperatures.

7.7.3 Vibration mode of an ethylene molecule

We consider an ethylene like 2D molecule as represented in Fig. 7.6. This molecule is composed of 6 atoms represented by balls that interact with each other via spring. We want here to compute and visualize the molecular vibration mode of this molecule. We consider only two types of interactions. The first ones are the stretching of the bonds whose potential energy reads

$$U^{XY}_{\text{stretch}}(r_{\alpha\beta}) = \frac{1}{2} K^{XY}_s (r_{\alpha\beta} - r^{XY}_0)^2 \tag{7.38}$$

Different spring constant and equilibrium distance must be used depending if the springs link two carbon atoms $K_s^{CC} = 200$ and $r_0^{CC} = 1.5$ or one carbon and on hydrogen atom $K_s^{CH} = 100$ and $r_0^{CH} = 1.0$. Similarly we consider the the bending angle between three atoms whose potential energy reads

$$U_{\text{bend}}^{XY}(\theta_{\alpha\beta\gamma}) = \frac{1}{2} K_b^{XYZ}(\theta_{\alpha\beta\gamma} - \theta_0^{XYZ})^2 \tag{7.39}$$

We set here $K_b^{XYZ} = 1$ and $\theta_0^{XYZ} = \pi/3$ regardless of the types of the three different atoms. To simplify the notation let's introduce the vector

$$\mathbf{q} = [x_1\, y_1\, z_1\, x_2\, y_2\, z_2\, x_3\, y_3\, z_3\, x_4\, y_4\, z_4\, x_5\, y_5\, z_5\, x_6\, y_6\, z_6] \tag{7.40}$$

that stores the positions of the the 6 atoms contained in the molecule. We refer to q_m for the m-th element of $\mathbf{q}$ and δq_m is a small increment of q_m.

A- You will first determine the positions of the atoms such that all the bond length and angles take their optimal values. HINT place the one of the two carbon at [0,0,0] and determine all the other positions from there. We refer to this positions as $\mathbf{q}_0$

B- To compute the vibration modes of this molecule you will first write a function that returns the value of the total potential energy ($U(\mathbf{q})$) of the system for a given value of the atomic positions $\mathbf{q}$. HINT: There is 5 stretching terms and 6 bending terms in the expression of the total potential energy.

C- The vibration modes of the molecules can be directly obtained with the diagonalization of the Hessian matrix

$$H(\mathbf{q}_0) = \begin{pmatrix} \frac{\partial^2 U}{\partial q_1 \partial q_1} & \frac{\partial^2 U}{\partial q_1 \partial q_2} & \cdots & \frac{\partial^2 U}{\partial q_1 \partial q_N} \\ \frac{\partial^2 U}{\partial q_2 \partial q_1} & \frac{\partial^2 U}{\partial q_2 \partial q_2} & \cdots & \frac{\partial^2 U}{\partial q_2 \partial q_N} \\ \vdots & \vdots & \ddots & \cdots \\ \frac{\partial^2 U}{\partial q_N \partial q_1} & \frac{\partial^2 U}{\partial q_N \partial q_2} & \cdots & \frac{\partial^2 U}{\partial q_N \partial q_N} \end{pmatrix} |_{\mathbf{q}_0} \tag{7.41}$$

The element of the Hessian can be calculated with the finite difference approach

$$[H(\mathbf{q}_0)]_{mn} = \frac{\partial}{\partial q_m}\left(\frac{\partial U}{\partial q_n}\right) \approx \frac{(\partial U/\partial q_n)_{\mathbf{q}_0 + \delta q_m} - (\partial U/\partial q_n)_{\mathbf{q}_0 - \delta q_m}}{2\delta q_m} \tag{7.42}$$

Developing the remaining differential with finite difference we can write

$$[H(\mathbf{q}_0)]_{mn} = \frac{U(\delta q_m, \delta q_n) - U(\delta q_m, -\delta q_n) - U(-\delta q_m, \delta q_n) + U(-\delta q_m, -\delta q_n)}{4\delta q_m \delta q_n} \tag{7.43}$$

where

$$U(\delta q_m, \delta q_n) \equiv U(\mathbf{q}_0 + \delta q_m + \delta q_n) \tag{7.44}$$

Using the function written in **B** compute the Hessian of the molecule. HINT you can avoid computing almost half of the term as $[H(\mathbf{q}_0)]_{mn} = [H(\mathbf{q}_0)]_{nm}$.

C - Diagonalize the Hessian using scipy. The eigenvalues of the Hessian (λ_i) are the vibration frequencies of the molecule and the eigenvectors the corresponding vibration modes of the molecules. NOTE you will obtain zero frequency modes that corresponds to rigid translation and rotation. Discard these modes. Compute and plot the spectrum given by

$$S(x) = \sum_i \exp\left(-\frac{(x - \lambda_i)^2}{2\sigma^2}\right) \tag{7.45}$$

with $\sigma = 0.1$. This spectrum corresponds to the peaks observed in IR or Raman spectroscopy.

D - The eigenvectors contains the displacement vectors of each vibrations modes. Hence consider the eigenvector

$$\mathbf{d} = [d_{x_1}\, d_{y_1}\, d_{z_1}\, d_{x_2}\, d_{y_2}\, d_{z_2}\, d_{x_3}\, d_{y_3}\, d_{z_3}\, d_{x_4}\, d_{y_4}\, d_{z_4}\, d_{x_5}\, d_{y_5}\, d_{z_5}\, d_{x_6}\, d_{y_6}\, d_{z_6}] \tag{7.46}$$

We can create an animation of the corresponding molecular motion by computing several snapshots of the positions

$$\mathbf{q}(t) = \mathbf{q}_0 + \mathbf{d} \times sin(t) \tag{7.47}$$

for $t = 0 \rightarrow 2\pi$. Exporting these positions in an XYZ file as presented in the previous exercises allows visualizing the molecular motion with most of the molecular visualization software (JMol, PyMOL, Blender-XYZ)

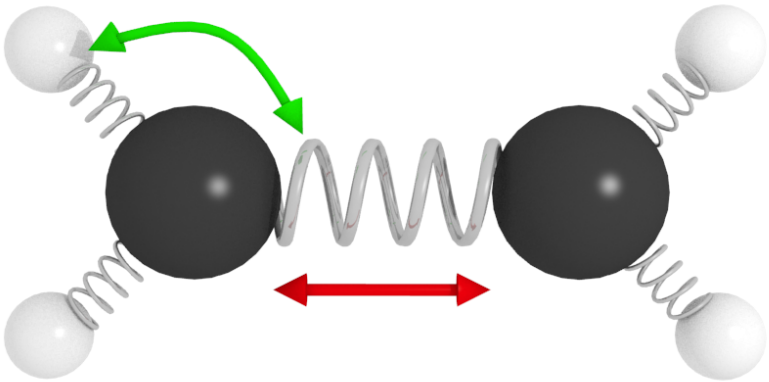


Figure 7.13: Ethylene molecule represented as a series of balls and springs.