# NUMERICAL METHODS FOR ASTROPHYSICS

*Michela Mapelli*[*]

*Padova University*



---

## CONTENTS

1 Introduction 6
- 1.1 Operating systems . . . 6
- 1.2 The Linux operating system . . . 7
- 1.3 Basic Linux commands . . . 9

2 Short summary of PYTHON for astrophysicists 14
- 2.1 Why python? . . . 14
- 2.2 Spyder . . . 15
- 2.3 Emacs . . . 15
- 2.4 Variables and assignments . . . 15
- 2.5 Output and input statements . . . 18

[*]michela.mapelli@unipd.it

2.6 Arithmetic . . . 19
2.7 Packages and modules . . . 21
2.8 Lists, tuples and arrays . . . 22
2.8.1 Lists . . . 22
2.8.2 Tuples . . . 24
2.8.3 Arrays . . . 25
2.9 A very important note on lists and numpy arrays . . . 27
2.10 Comments . . . 28
2.11 if statement . . . 28
2.12 while statement . . . 29
2.13 for loops . . . 30
2.14 break and continue statements . . . 31
2.15 Dictionaries . . . 32
2.16 Functions . . . 35
2.17 Reading from and writing to files . . . 40
2.18 Subprocess . . . 42
2.19 Regular expressions . . . 42

3 Visualization 46
3.1 Scatter plot . . . 46
3.2 Line plot . . . 49
3.3 Logarithmic axes . . . 50
3.4 Annotating text . . . 50
3.5 Legend . . . 50
3.6 Note on fontsize . . . 51
3.7 Colors in python . . . 52
3.8 Histogram . . . 57
3.9 Two-dimensional histograms . . . 62
3.10 Contour plots . . . 63
3.11 Subplots . . . 70
3.12 3D plots . . . 74

4 Accuracy and speed 78
4.1 Scientific notation . . . 78
4.2 Maximum size of a variable . . . 78
4.3 Rounding errors . . . 79
4.4 Speed . . . 82
4.5 Small scattered tips . . . 88

5 Solution of linear equations 90
5.1 Gauss elimination method . . . 91
5.2 Pivoting . . . 98
5.3 LU decomposition . . . 99
5.4 Gauss-Seidel method . . . 101
5.5 Pros and cons of the algorithms we discussed . . . 102

6 Solution of non-linear equations 105
6.1 The relaxation method . . . 105
6.2 Overrelaxation . . . 107
6.3 Bisection method . . . 108
6.4 Newton-Raphson method . . . 109
6.5 Newton-Raphson method for systems of non-linear equations . . . 112

7 Numerical derivatives 116
7.1 Forward, backward and central differences . . . 116
7.2 Second derivatives . . . 117
7.3 Partial derivatives . . . 117
7.4 Derivatives of noisy data . . . 117

8 Random numbers 119
8.1 Random generators . . . 119
8.2 Random number seeds . . . 121
8.3 Uniform and non-uniform random numbers . . . 122
8.4 Inverse random sampling . . . 123
8.5 Gaussian random numbers . . . 125
8.6 Rejection method . . . 128

9 Integration of functions 138
9.1 Trapezoidal rule . . . 138
9.2 Estimate of the errors . . . 139
9.3 Monte Carlo technique . . . 142
9.3.1 **The mean value method** . . . 144
9.3.2 **Integrals in many dimensions** . . . 146
9.3.3 **Importance sampling** . . . 148
9.4 Built-in functions to integrate in python . . . 152

10 Integration of ordinary differential equations (ODEs) 156
10.1 Euler scheme . . . 156
10.2 Second-order Runge-Kutta scheme . . . 157
10.3 Fourth-order Runge-Kutta scheme . . . 157
10.4 Systems of ordinary differential equations . . . 158
10.5 Second-order and higher-order ordinary differential equations . . . 160
10.6 The astrophysical N-body problem . . . 161
10.7 Astrophysical N-body problem with Euler's method . . . 162
10.8 Astrophysical N-body problem with midpoint scheme . . . 163
10.9 Leapfrog scheme . . . 167
10.10 Fourth-order predictor-corrector Hermite scheme . . . 169
10.11 Collisional vs collisionless N-body simulations . . . 172
10.12 Adaptive step size . . . 174

10.13 When adaptive time steps matter: evolution of a binary compact object by gravitational wave emission . . . . . 175
10.14 Modified mid-point method . . . . . 179
10.15 Bulirsch-Stoer method . . . . . 183
10.16 Initial value problems and boundary value problems . . . 188
10.17 The shooting method . . . . . 188

11 Partial differential equations 190
11.1 Boundary-value PDEs with finite difference methods . . . 190
11.2 Solve by iteration . . . . . 191
11.3 Initial-value PDEs with finite difference methods . . . . . 194
11.4 The forward-time centered-space (FTCS) method for initial-value PDEs . . . . . 195
11.5 Euler's method, again . . . . . 195

12 Interpolation and extrapolation 199
12.1 Linear interpolation . . . . . 200
12.2 Polynomial interpolation with Lagrange's method . . . . . 201
12.3 Polynomial interpolation with Newton's method . . . . . 203
12.4 Limitations of polynomial interpolation . . . . . 207
12.5 Two-dimensional interpolation . . . . . 208
12.6 Cubic spline interpolation . . . . . 210
12.7 Python modules to interpolate . . . . . 213

13 Fitting procedures 217
13.1 Linear fit with the least squares . . . . . 218
13.2 Linear fit by weighting of data . . . . . 222
13.3 General approach to least-square fitting . . . . . 223
13.4 Polynomial fitting . . . . . 224
13.5 Python functions for fits . . . . . 226
13.5.1 scipy.optimize.least_squares . . . . . 226
13.5.2 scipy.optimize.curve_fit . . . . . 228

14 Fourier transforms 237
14.1 Fourier series: a math summary . . . . . 237
14.2 The discrete Fourier transform (DFT) . . . . . 240
14.3 Fast Fourier transform (FFT) . . . . . 243
14.4 Physical interpretation of the Fourier transform . . . . . 246

15 Other useful packages and environments 255
15.1 The conda environment . . . . . 255
15.2 jupyter-notebook . . . . . 256
15.3 Using file formats other than plain text files . . . . . 257
15.4 Scipy . . . . . 266
15.5 Astropy . . . . . 266
15.6 Introduction to PANDAS . . . . . 276

15.6.1 Series . . . . . 277
15.6.2 DataFrame . . . . . 280
15.6.3 Reading/Writing DataFrame from/to files . . . . . 285
15.6.4 Visualization in pandas . . . . . 287

16 Notions of machine learning in astrophysics 289
16.1 What is Machine learning? . . . . . 289
16.2 Interpretability and Machine Learning . . . . . 291
16.3 Decision Tree . . . . . 291
16.4 Example: the iris flowers . . . . . 294

17 Sorting algorithms 299
17.1 Bubble Sort . . . . . 299
17.2 Selection Sort . . . . . 299
17.3 Quicksort . . . . . 300
17.4 Merge Sort . . . . . 301

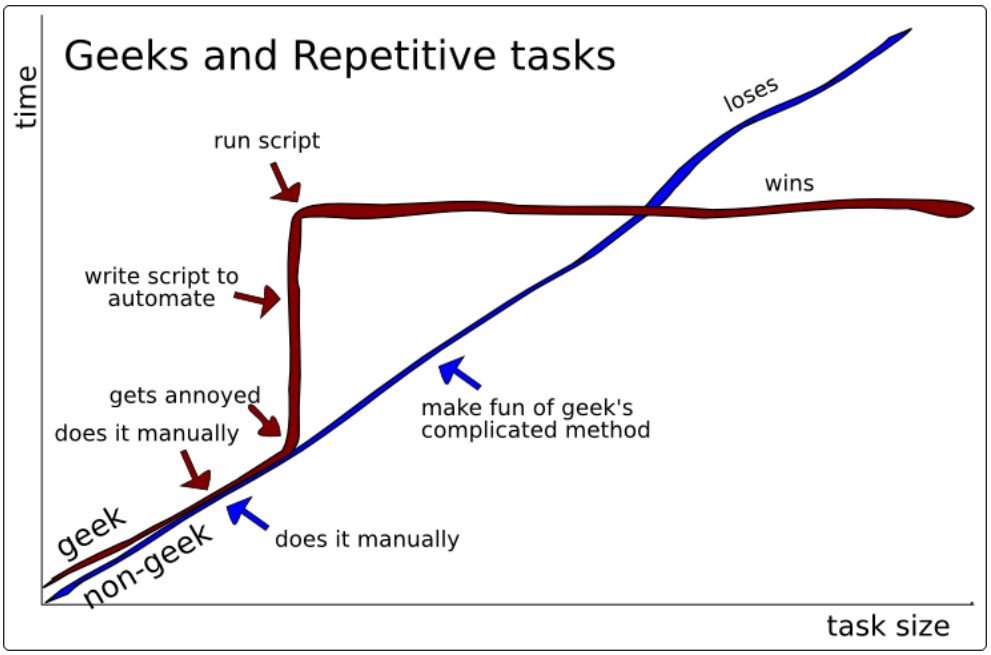


*Figure 1: A nice way to explain why it is better to know numerical methods. Credits: Bruno Oliveira.*

# 1 INTRODUCTION

We start this course with a good news and a bad news. The bad news is that numerical methods are a *must* if we want to do research in astrophysics nowadays (to better say, not just in astrophysics but in everything), the good news is that i) numerical methods are tremendously easy to learn and ii) are going to save your time (see Figure 1). Let's get started.

## 1.1 *Operating systems*

"*An operating system (OS) is system software that manages computer hardware, software resources, and provides common services for computer programs*" (source Wikipedia).

In most cases, the OS must work as an intermediary between computer hardware and user-launched applications, to allocate resources (e.g. memory) and to optimize the usage of the hardware. Currently, the most-common operating systems are Microsoft Windows (~82% of the desktops have Windows when they are sold), macOS (~ 13%) and Linux (~ 2%). Consider, however, that it is common practice among Linux users to buy a Windows desktop, because most of the machines offered on the market have Windows by default, and then to install Linux soon after.

**DISCLAIMER:** Before I start talking about Linux, let's make the following statement. You do not have to use Linux during this course. You can study python and do the proposed exercises with your preferred operating system

(Linux, mac OS or windows), but there are two good reasons to use Linux for this course. The practical reason is that the written exam will be in room P104 (unless covid-19 wins), where you will have Linux machines. The long-term, general reason is that this course is an opportunity to learn Linux, which is the most used operating system in high-performance computing (HPC) and in science.

If you do not have Linux on your machine but you want to try it, you have three options:

1. connect to a workstation of room P104 remotely;

```
ssh -p 5210 -Y username@labp104.fisica.unipd.it
```

2. download the **virtualmachine** prepared by the system managers (thank you Dr Loris Lazzaro and Dr Matteo Menguzzato) for this course from this url `https://drive.google.com/file/d/1dY6RzW3gePm9K2diPWOJnfNcs7Jy-oq0/view?usp=sharing`;

3. install **cygwin** (`https://www.cygwin.com/`, or other Linux emulator on windows), which allows you to "simulate" Linux on your windows workstation/laptop;

4. install Linux on your laptop.

I can help you with these; ask the system managers of lab P104 to set up your account. Now, let's move on.

## 1.2 *The Linux operating system*

**Linux** is an **open source operating system**, or, to better say, a family of open source operating systems based on the **Linux kernel**. We should rather call it **GNU/Linux operating system** because the Linux operating system is a combination of Linux kernel and supporting system software and libraries, most of which are provided by the **GNU project**.

There are different versions of the Linux software which are called **Linux distributions**. The most popular ones include **Ubuntu**, **Debian**, **Fedora** and **OpenSuse**.

**Open source** means that all the lines of the code are public: they can be seen and edited by everyone legally. It does not necessarily mean that the code is for free. A **free software** is an open source code that is also for free. There are distributions of the linux software (e.g., Red Hat and Suse) which are open source but are not free.

A **kernel** is literally the core of the operating system. It is a computer program at the core of the operating system which controls the system itself.

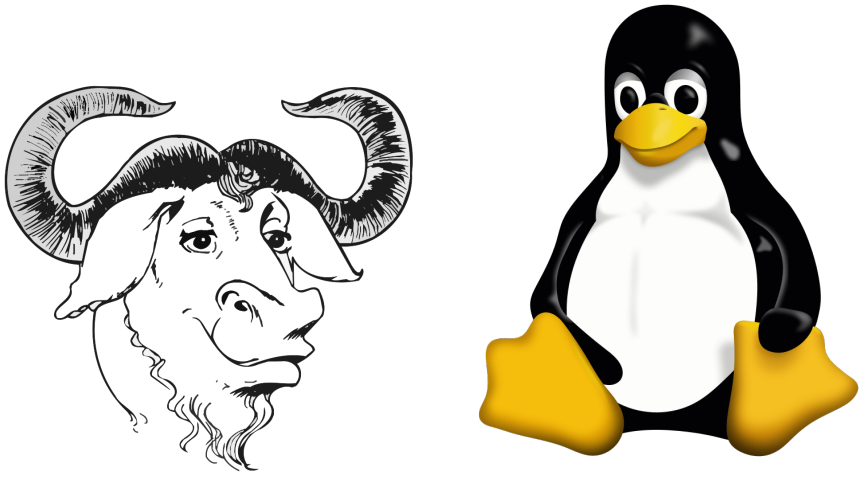

*Figure 2: Left: The original GNU logo, drawn by Etienne Suvasa. Right: Tux the penguin, mascotte of Linux.*

The GNU project is a **free software** initiative and an extensive collection of utility programs wholly free software. Most of it is licensed under the GNU Project's own General Public License (GPL). The GNU operating system was intended to use the Hurd kernel, but this remains at a state of development that is not ready for daily use. Instead GNU software is more usually used in combination with the Linux kernel, in which case it is commonly known as Linux. **Richard Stallman**, an employee of the MIT Artificial Intelligence Laboratory, founded the GNU project on September 27 1983. Soon after, he quit his job at the Lab so that they could not claim ownership or interfere with distributing GNU components as free software. The main idea of Stallman was to create a completely free software initiative that "can create community and social justice". Figure 2 shows the popular GNU and Linux mascottes.

The first version of the Linux kernel was released by **Linus Torvalds** on September 17 1991. Linux takes its original structure from Unix, which, after being born as open source, in 1984 became a proprietary product, where users were not legally allowed to modify software. Torvalds wanted to build its own kernel in a way that it was open source and compatible with Unix. Many open source developers agree that the Linux kernel was not designed, rather it evolved through Natural Selection. Many developers contributed to modify Linux since its early stages, and only the best modifications were accepted by the community, contributing to a natural selection of the software.

Linux started to become popular (not only among geeks) in the mid-1990s in the supercomputing community, where organizations such as NASA started to replace their increasingly expensive supercomputers with clusters of inexpensive computers running Linux. Commercial use began when Dell and IBM started offering Linux support to escape Microsoft's monopoly in the


*Figure 3: The Unity desktop environment with Ubuntu Linux.*

desktop operating system market.

Nowadays, Linux is not particularly common among desktop users (~ 2%, probably underestimated, see the discussion above), but is by far **the most common OS for servers and HPC facilities**: all the Top500 machines (i.e., the 500 most powerful computer clusters around the World) have Linux. Moreover, **android** adopts the Linux kernel, making linux the most common OS among mobile phones.

## 1.3 *Basic Linux commands*

The **user interface** (i.e. the way the user can communicate with the OS) is known as the **shell**. In the case of Linux, the most common shell is the **Bourne Again Shell (bash)**, originally developed for the GNU project. The shell is usually composed of both a **command-line interface (CLI)** and a **graphical user interface (GUI)**. On desktops, the GUI shell is packaged together with the desktop environment. The most common GUI shells and desktop environments are **KDE, Unity and GNOME**. Figure 3 shows the desktop you might have in front of you if you have a Ubuntu linux with a Unity desktop environment. On the left bar, you have a menu with different fundamental applications and a search tool. On the top right, you have some fundamental commands, such as the shut down button, the clock, the language setting, the connection information, the sound information, and several other system settings.

If you work on Linux, remember to use the **terminal**: the command line of the terminal can be more effective than the graphics interface (although, it might seem counter-intuitive to people used to work with Windows and Mac).

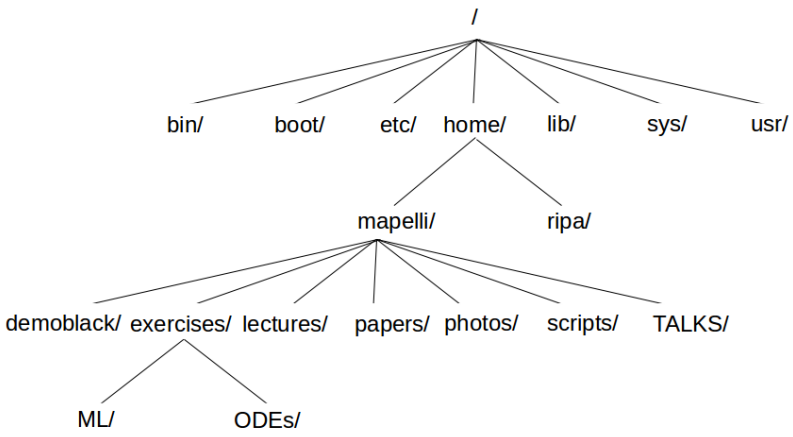


*Figure 4: An example of the Linux directory tree.*

When you open the terminal, you can first type

```
pwd
```

followed by an enter.

**pwd** tells me in which directory I am.

Usually, the default directory of Linux (i.e. the one in which I am when I log in) is my home:

```
/home/username/
```

*Warning: Directories in a computer are organized like a* **tree**, *with a common root (the / directory) and many sub-directories which are the branches and the leaves of the tree (see Figure 4). The / directory, the one on top of the directory tree, is usually the* **system** *directory, i.e. the place where the operating system was installed and lives. Be careful: don't mess up with the / directory.*

To see more options for pwd (or any other command), type

```
man pwd
```

This calls the Linux manual, which will allow you to see all possible arguments taken by that command.

Other important commands are the following.

**ls** Shows you what you can find in the specific directory where you are. I usually use it with the options below

**ls -lrt** The argument -l prints information on the file owner and last-change date. The argument -t allows you to sort files by modification time,

newest first, and the argument -r reverses the previous order: you see the newest last (usually you need to see the most recent files, rather than the oldest ones).

**mkdir** Creates new directory (if followed by the name of the new directory).

**cd** To change directory. **cd directoryname** moves you to the directory with name directoryname. If you type **cd** without any arguments, you are sent to the home.

*SUGGESTION: If you are not familiar with the terminal, use* **cd** *and* **ls** *to look around and to get familiar with the directory structure. It might seem clumsy at the beginning, but, when you become familiar with it, is way more efficient than through the* **Files** *graphical application (the one that looks like a windows or mac application).*

**rm** Removes a file or a directory. If you want to remove a directory, use **rm -r**. If you want to avoid removing something you did not want to remove use **rm -i filename** (for files) and **rm -ri directoryname** (for directories). You will be asked to confirm that you want to remove that file/directory.

**cp** Copies a file or a directory into something else. If you want to copy a directory, use **cp -r**. If you want to avoid overwriting something, use **cp -i filename newfilename** (for files) and **cp -ri directoryname newdirectoryname** (for directories). With the '-i' option, if there is already a file or directory with that name, you will be asked to confirm that you want to overwrite the file/directory.

**diff** filename1 filename2. Compares filename1 and filename2 and prints out the differences.

**ssh** Connect to another computer remotely. The best syntax is the following

```
ssh username@computername.domain -X
```

where '-X' is necessary to open pop-up windows.

**scp** Copies files from or to a remote computer. In particular,

```
scp username@computername.domain:filepath/filename pathtosendfile/filename
```

copies the file with name filename (that you can find on computer computername.domain when following the path filepath) from computername.domain to your local machine, at the location specified by pathtosendfile/filename.

```
scp filepath/filename username@computername.domain:pathtosendfile/filename
```

copies the file with name filename (that is on your computer at the path filepath) to a remote computer computername.domain

**rsync** Copies a file (or directory) from a directory to another directory or to a remote computer in a safe way (by checking the integrity of the file and by updating time and ownership). Use the following syntax:

```
rsync -a username@computername.domain:filepath/filename pathtosendfile/filename
```

if you want to copy the file filename from a remote computer to your computer, and

```
rsync -a filepath/filename username@computername.domain:pathtosendfile/filename
```

if you want to copy the file filename from your computer to a remote computer. Highly recommended with respect to scp.

**mv** Moves a file to some other file (i.e. it renames the file) or to some other place on your computer.

*SUGGESTION: do not use* mv, *just use* cp *or* rsync -a. *If you are moving a big file or directory and something happens (e.g. a black out) during your moving, you lose the file or directory forever. When you want to move a file, it is safer to make a copy of it and then remove the original.*

**du -sh** followed by a filename or a directoryname, tells you how large (in bytes) the file or directory is. The argument -sh means **human readable units** (which means that units of measure (k for kilobytes, M for megabytes, G for gigabytes, etc) are provided by the command. In contrast, -ks is only in kilobytes. If you use **du -sh** $*$, you see the size of all the files in that directory and of all the subdirectories in that directory. In general, $*$ means all the files and all the subdirectories in that directory.

**df -h** followed by nothing shows you the size of the disks mounted on the computer and how much of this disk space is used/available. df -h . gives you information only on the local disk.

**top** shows all the processes running on your computer (you can kill the unwanted ones with k).

**ps aux** also shows all processes running on your computer. You can use the command

**kill** followed by the identifier of the process to kill that process (if the process does not die use kill -KILL followed by the identifier of the process to kill).

**find -name** filename or 'file*' searches a file in the directory and in all the directories below the one where you type the command.

**grep** 'something' filename searches the string something within filename. It is very useful when you debug large codes (trust me!) or when you want to do a quick check over a file (e.g. see whether there are some NaN).

**wc -l** filename calculates the number of rows in fi.

**nl** filename prints the number of lines in filename followed by the content of each line (does not work well with binary files, of course).

**more** filename shows the content of the file filename (in a non-editable way).

**less** filename shows the content of the file filename (in a non-editable way). Very similar to more but more powerful (e.g. you can scroll up and down a file with the arrows on your keyboard).

**tar cvfz** filename.tgz followed by the list of files or directories to be archived produces an archive (= a container) of files and zips them. Here c means create the archive, v means verbose mode, f means use the archive tool, while z means zips it. If you do not want to zip the archive use just **tar cvf**.

**tar tvfz** filename.tgz allows you to see the names of the files and directories stored into the (already existing) archive filename.tgz without actually uncompressing it.

**tar xvfz** filename.tgz uncompresses the tar file filename.tgz. If you want to extract only one file or directory, add the name of this file or directory at the end of the command.

Remember that Linux command can be concatenated by using the pipe ('|') command. For example

```
ps aux | grep python | nl
```

ps aux prints the processes running on your computer, grep finds every row of filename where there is a 'python', then 'nl' calculates and prints the number of these rows.

# 2 SHORT SUMMARY OF PYTHON FOR ASTROPHYSICISTS

The examples corresponding to this section are in examples/python/.

The material needed for the exercises is in exercises/python/.

## 2.1 *Why python?*

Why choosing python as programming language for this course? No special reason. The important thing about programming is to understand and learn the **logic** of programming, in general. Once this is achieved, differences between programming languages are a minor thing to learn.

However, a feature of python that makes it particularly appealing for this kind of courses is that python is a **high-level programming language**: you do not need to write a code the way the computer understands it directly. Hence, you can write a code more the way YOU understand it, making your life easier and your coding faster[1].

Moreover, python is an **interpreted language**: you do not compile and execute python. Rather, you call an **interpreter** that interprets your code for you, i.e. makes it understandable to the computer, through pre-compiled python libraries[2].

To call the interpreter on linux you type

*python namecode.py*

and then you press enter, or simply

*python*

and then you press enter. In the former case, you want the interpreter to interpret (i.e. execute) your script *namecode.py*, while in the latter case, you enter the interpreter and then you can write your script interactively inside the interpreter. This latter option is not particularly smart for a long code, but is a very nice option to get a flexible scientific calculator (for free and always available on your computer).

[1] In contrast, low-level programming languages such as C and C++ need you to program more the way the computer thinks.

[2] In contrast, compiled languages (such as C and C++) need you to compile and execute them. Compilation consists in translating the code from the chosen programming language to computer language, i.e. assembly language (which depends on computer architecture; hence it is good practice to recompile your code when you want to use it on a different machine). Of course, you do not have to translate it 'by hand': every compiled language has several compilers that perform this task for you. For example, if you use GNU software, in C the typical compiler is the gcc. The result of compilation is a new file, written in assembly language, that is called the executable. To run your code you need to execute the executable. In Linux, usually you run the executable by typing ./executable_name.

Furthermore, it is extremely **easy to produce plots** with python (through the *matplotlib* package, for example) and in general python includes a lot of **mathematical libraries** that are very easy to call and use (mostly through the *numpy* and the *scipy* packages). In the following, we review the most important ingredients of python programmers for beginners.

## 2.2 *Spyder*

If you don't like running python directly from the terminal, you can use the **spyder** environment (spyder stands for Scientific Python Development Environment, `https://www.spyder-ide.org/`). Spyder can be installed quite easily (it is already provided on the workstations of lab P104). It provides many features, including an **editor** to display and edit your python scripts, a **graphical user interface to run** your python script (if you really don't like the command line from the terminal) and a **debugger**, to spot the main issues with your script. During these lectures, I will use the command line because I am a command-line enthusiast, but you are perfectly welcome to use spyder instead: it is nice and user friendly.

## 2.3 *Emacs*

I prefer to use the command line to run my python script and I prefer to use the emacs editor to write my scripts. During these lectures I will almost always use emacs.

What about you? If you are already familiar with another editor (e.g., the editor inside spyder, or gedit, or vim, or whatever else), of course you can keep using your preferred editor.

If you are not familiar with any editor or you want to try a different editor from the one you used before, I suggest either emacs or the editor inside spyder. Have a look at Figure 5 to have some nerdish fun about opinions of scientists on the best editor.

## 2.4 *Variables and assignments*

**Variables** are the minimum building blocks in coding. They contain information about scalar quantities, playing approximately the same role as they do in algebra. For example

```
x=1
```

means that I have defined a variable x and I have assigned it the value of 1. This is an **assignment statement**. Note that in many other languages (e.g. C and C++) the definition of a variable is independent from the assignment and both the definition and the assignment are mandatory. For example, in C

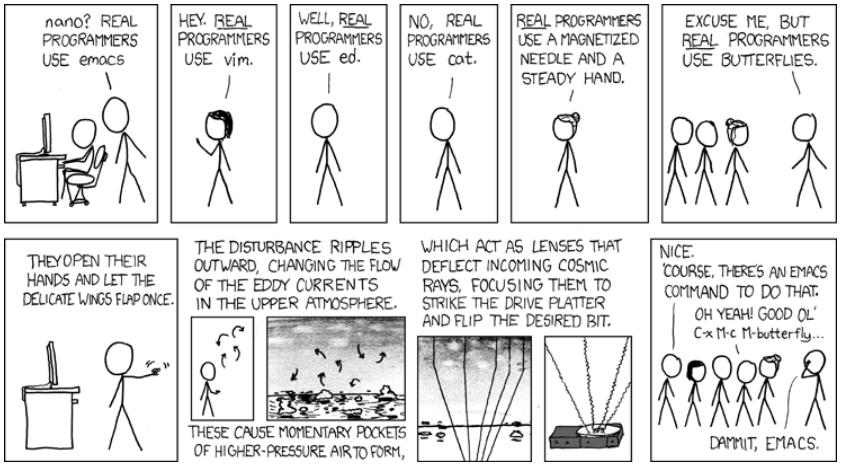


*Figure 5: A "debate" on the best editor for real programmers. Credits: xkcd (see* `https://xkcd.com/`*).*

```
int x;
x=1;
```

In contrast, in python you do not need to define a variable, you just assign it a value. This simplifies your coding but is *dangerous*, because you are allowed to use the same name for two different variables in the same script, even with two completely different assignments, possibly with catastrophic effects.

Variables have different **types**. The main types are

- **integer**: a variable is an integer if it is assigned an integer value, which can be positive or negative (e.g., -1,0,20002). Fractional values (e.g. 2.4) are not integers. In terms of programming, a number is an integer if and only if it is written without the dot (e.g. 1 is an integer, but 1.0 is not an integer for the computer);
- **float** (or floating-point): a variable that is assigned a real or fractional value (e.g. 1.345e-30, 1.5 or 1.0);
- **complex**: a variable that can take a complex value such as $1 + 2\,i$. Note that $i$ in the sense of $i^2 = -1$ in python cannot be written as $i$ but as $j$. Thus, the complex number $1 + 2\,i$ in python is written as $1 + 2\,j$.
- **string**: a variable that can be associated to a string of characters (or a mixed string of characters and numbers). For example "c" or "I am a string" or "I am a 123 string".

If I assign

```
x=1
```

or

```
x=int(1)
```

it means that x is a integer.

Instead, if I assign

```
x=1.0
```

or

```
x=float(1)
```

it means that x is a float.

Integers and float occupy different sizes of the memory of a computer. Integers in python are 32 bit[3] (in C they are only 16 bit)[4].

To know the exact value of the maximum python integer on your machine, inside the python2 interpreter type the following

```
import sys
sys.maxint
```

Note that the above command works only in python2. The command sys.maxint was removed in python3 (the default of our course) because python3 in principle does not have limits on the size of int. On the other hand, you can use sys.maxsize as an integer larger than any practical list or string index.

Floats in python are 64 bit (while in C they are just 32 bit; a 64 bit is a double precision variable in C). The maximum value a float can assume is 1.8e308. If we try to assign a larger number than approximately 1.8e308 to a variable, the variable is automatically re-assigned to "inf". Try for example

```
x=1.8e308
print(x)
```

To know the exact value of the maximum float on your machine, inside the python2 or python3 interpreter type the following

```
import sys
sys.float_info
```

If a quantity I want to use is an integer (for example the number of planets in the Solar System is intrinsically an integer), use an integer variable for it.

[3] A bit is a basic unit of information in information theory. It is the information that can be conveyed by a number which can assume only the values of 0 or 1, or a statement that can be either False or True.

[4] Larger integers can be defined through the *numpy* package. For example type *uint64* in *numpy* is a 64 bit unsigned integer, i.e. a 64 bit positive integer.

Besides using less memory, it is more accurate. Rounding errors make float intrinsically less accurate, for example 1.0 can be stored in the computer as 0.9999999999999999.

If I want to define and assign a string I do

```
x="a"
```

or

```
x=str("a")
```

In this case the string value is *a*. Or I can define, for example,

```
x="this is a string"
```

or

```
x=str("this is a string")
```

In this case the string is *this is a string*. Or I can define the following string

```
x="123.0"
```

or

```
x=str("123.0")
```

In this case 123.0 is interpreted as a string, not as a float: it will behave as a string.

Strings are important to assign the names of files, as we will see later. In python " and ' have the same meaning and usage.

Python is one of the few languages that allow to use the same variable to have two or more different types in the same function. This is allowed but is not advisable. For example, if you assign x=1 and few lines later you assign x=1.5 you might lose control of the actual type of x and produce disasters.

## 2.5 *Output and input statements*

An **output statement** is the way the code prints some results to the terminal. This is very simple in python:

```
x=1
print(x)
```

The function *print* writes the value of x. Note that the syntax of print has changed from python2 to python3. If you use python2 you can also type the following

```
x=1
print x
```

An **input statement** is the way we assign the value of a variable through the command line:

```
x= input("Enter the value of x: ")
```

The function *input* requires the user to assign the value of the variable by typing it into the command line. *input* tries to guess the type of the input but to be sure that the type of the input is what you want, you must explicitly choose it. For example

```
x= float(input("Enter the value of x: "))
```

tells that x will be a float,

```
x= int(input("Enter the value of x: "))
```

tells that x will be an integer and

```
x= str(input("Enter the value of x: "))
```

tells that x will be a string.

## 2.6 *Arithmetic*

The basic **arithmetic operators** are written as follows.

```
x+y    addition
x-y    subtraction
x*y    multiplication
x/y    division
x**y   power
```

In addition

```
x//y   x divided by y and result rounded to the nearest int
x%y    modulo (the remainder of x after being divided by y)
```

Note that the result of an arithmetic operation depends on the type of variables. For example

```
x=123.0
y=2.0
x+y
```

and

```
x="123.0"
y="2.0"
x+y
```

are completely different!

Similar to normal algebra, operations can be combined with each other. Multiplications and divisions are performed before additions and subtractions. If there are several multiplications (or divisions), they are carried from left to right. Powers are calculated before anything else. To change the order of operations, you can use round brackets ().

Note that these operations are **arithmetic assignments**, they are not equations, although they might look like equations. For example

```
x=0
x=x**2-2
print(x)
```

If this were a true equation, it would have admitted two results (2 and -1). Instead, the computer calculates the right term of the above operation using the previous assignments for variables ($x**2 - 2 = 0 - 2 = -2$) and then assigns the result to the variable on the left term. This is why print(x) gives $-2$.

Python admits the usage of **modifiers** (similar to C and C++):

```
x+=1 is equivalent to x=x+1
x-=2 is equivalent to x=x-2
x*=2.4 is equivalent to x=x*2.4
x/=7.0 is equivalent to x=x/7.0
x//=3.0 is equivalent to x=x//3.0
```

In python it is possible to assign two variables with a single statement:

```
x,y=2.2,1
```

is equivalent to

```
x=2.2
y=1
```

As a consequence

```
x,y=y,x
```

swaps the value of the two variables.

**EXERCISE:**

Use what you learned to calculate the distance covered in a (user provided) time $t$ by a ball falling from a tower of (user provided) height $h$. Furthermore, calculate at what time $t_2$ the ball reaches the ground (the gravity constant $g = 9.81$ m s$^{-2}$).

## 2.7 *Packages and modules*

From the above description of arithmetic operators, you can clearly spot some missing important arithmetic functions. For example, logarithms or trigonometric functions. These important operators as well as more sophisticated functions and tools are not present in the "default" python interpreter, but can be found in additional **packages** (i.e. collections of useful functions and constants, some of them coming with the default python, others needing to be installed separately). The way to load these additional packages, making them available to our calculations is to type "import namepackage". For example, the following command

```
import math
```

imports (i.e. makes available to the interpreter) the package *math*, that contains some important mathematical function such as

```
log             natural logarithm
log10           log base 10
exp             exponential
sin, cos, tan   sine, cosine, tangent (in radians)
asin, acos, atan arcsine, arccosine, arctan (in radians)
sqrt            square root (without importing math **0.5)
```

Each of the functions that are inside the package must be called as *packagename.functionname*. For example

```
import math
a=math.log10(100.)
```

If you want to import only a part of the entire package, you can do as follows

```
from math import log10
a=log10(100.)
```

which imports only the log10 function.

Some packages are so big that they contain multiple **modules**, i.e. sets of functions and other useful things. For example, we will often use the **numpy** package that contains several modules.

```
import numpy as np
```

imports the entire numpy package and assigns it the nickname np, which is much faster to type in the rest of the code.

If we are not interested in the entire numpy but only in one of its modules we should type, for example

```
from numpy import linalg as lin
```

The above command imports the module linalg (facilities for linear algebra and Fourier transformations) of the numpy package and gives it the nickname of lin.

Finally, if we are interested only in one function (or other content) inside the module, we can import just this function. For example

```
from numpy.linalg import inv
```

which imports only the function inv (which calculate the inverse of a matrix) from the entire linalg module (note the namepackage.namemodule formalism).

## 2.8 *Lists, tuples and arrays*

So far, we have used only variables, which are scalar quantities. But what happens if I want to use something that looks like a vector (for example the vector describing the position $\vec{r} = x, y, z$ of a point in space)? Or, what can I do if I have a set of data which I want to treat all in the same way (for example I want to multiply each of them for the same variable), even if they do not constitute a vector?

To satisfy these needs you can use the so called **containers** of python. In this course we will use mainly two kinds of containers: lists and arrays. We will briefly mention two additional types of containers: tuples and dictionaries.

### 2.8.1 Lists

A **list** in python is an ordered list of values, which are called its **elements**. The elements of a list can be of different types, i.e. you can have a list composed of several strings, several floats and several integers all together.

As we already discussed for variables, a list can be assigned in python without being defined, which is fast but dangerous. To define a list we type

```
r = [1.,15.,2,"sea",1e30]
```

where we have a list named r, whose elements are 1.,15.,2,sea and 1e30 (note that sea is a string, 2 is an integer, while the other elements are float). In our course, we will use almost exclusively lists whose elements share the same type.

You can assign a list also in the following way (through variables):

```
x=1.
y=15.
z=2*int(x)
a="sea"
b=1e30
r = [x,y,z,a,b]
print(r)
print(r[0])
print(r[-1])
```

The elements of a list are numbered from 0 to $n-1$, where $n$ is the number of elements of the list. So r[0] is the first element of the list (in the above example its value is 1.), r[1] is the second element of the list (in the above example, its value is 15.) and so on. The last element of a list can be called r[-1].

I can do several operations to the elements of a list, for example I can sum them

```
r = [1.,2.,10.,3.]
a=sum(r)
print(a)
```

I can even add elements to the end of a list or add/remove elements from the list with the functions append and insert/pop.

**append**

```
r = [1.,2.,10.,3.]
r.append(6.1)
print(r)
```

from the print I see that the new r is [1.0, 2.0, 10.0, 3.0, 6.1], with the new element (the float 6.1) at the end.

**pop**

```
r = [1.,2.,10.,3.]
r.pop(1)
print(r)
```

from the print I see that the new r is [1.0, 10.0, 3.0], i.e. pop has removed second element (r[1], which is 2.0).

**insert**

```
r = [1.,10.,3.]
r.insert(1,9.)
print(r)
```

with insert I have inserted a new element (9.) with index 1 (i.e. the second element of the list) and I have shifted all the other elements to the right. Hence, insert wants two arguments **insert(index,object)**: the first argument is the index of the element I want to add in the list, the second argument is the value of this element.

A very common way of producing a list is to start with an empty list and then building it with append. For example

```
r = []
print(r)
r.append(1.)
r.append(2.)
print(r)
```

where r=[] defines an empty list (list with no elements), while at the end of the example r=[1.0, 2.0] has two elements. For this usage, append is probably the most useful and common function applied onto lists.

### 2.8.2 Tuples

In general python, there exists another way to organize vector-like quantities: the **tuples**. Like a list, the elements of a tuple do not need to be of the same type. For example,

```
a=('word', 17.7, 2)
```

defines a tuple of three elements, which are a string, a float and an integer. From the example, you also see that tuples are defined with round brackets instead of square brackets. Operations on the tuples are the same as operations on the lists (e.g., if you sum two tuples, you simply concatenate them). The last thing to know about tuples is that the number of elements cannot be changed, so NO append on a tuple.

### 2.8.3 Arrays

Similar to lists, arrays are ordered lists of values, but they have important differences with respect to lists:

1. They exist only in the *numpy* package.
2. The number of elements of an array is fixed.
3. The elements of an array must be of the same type.

From the above enumeration it is apparent that arrays are way less flexible than lists. Then, why and when should we use them?

Here are few good reasons to use arrays:

1. Arrays can be two-dimensional as matrices.
2. Arrays behave like vectors and matrices in algebra. When you do operations with them (e.g. sums), the result is what you expect from arithmetic, while when you do operations with lists you get very different results (example below).
3. Arrays work faster than lists.

To use an array, you need to create it. There are several ways to proceed. For example

```
import numpy as np
a = np.zeros(4,float)
print(a)
```

This creates the array a with 4 elements that are float. All the elements are initially zero (the function np.zeros fills with zeros all the elements).

Or you can convert to an array a list you previously created

```
import numpy as np
a = [1.,2.]
b = np.array(a)
print(b)
```

where b is the list a transformed into an array.

To create a two dimensional float array with $m$ rows and $n$ columns (like a $m$x$n$ matrix) you use the command numpy.zeros([$m$, $n$], float). For example

```
import numpy as np
a = np.zeros([2,3],float)
print(a)
```

produces the following output

```
[[0.  0.  0.]
 [0.  0.  0.]]
```

I can call the elements of a matrix with a set of two indexes

```
import numpy as np
a = np.zeros([2,3],float)
a[0,1]=-1.0
a[1,2]=1.0
print(a)
```

produces the following output

```
[[0. -1.  0.]
 [0.  0.  1.]]
```

Here is an example of the different behaviour of arithmetic operators when applied to lists and arrays.

```
import numpy as np
a = [1.,2.]
a1 = np.array(a)
b = [3.,1.]
b1 = np.array(b)
c=a+b
print(c)
c1=a1+b1
print(c1)
```

The print(c) operation gives the following

```
[1.0, 2.0, 3.0, 1.0]
```

i.e. the sum of two lists is a new list that results from the concatenation of the two lists.

Instead, the print(c1) operation gives the following

```
[4.0, 3.0]
```

which is the arithmetic sum of two vectors.

An array can be sliced:

```
a = np.array([2.,3.,9.,5.,10.,1.])
slice = a[2:4]
print(slice)
```

gives as output

```
[9. 5.]
```

## 2.9 *A very important note on lists and numpy arrays*

A very important thing to know about python is that the assignment

```
x=np.zeros(4,float)
x2=x
```

where both x and x2 are numpy arrays (or lists), does not make a copy of x into x2. Instead, the assignment statement makes x and x2 both POINT to the same address in memory (if you are familiar with C and C++ you immediately understand what this means).

Hence, if (in the rest of the script) you change the elements of x2, you change also the elements of x, because x2[i] and x[i] are stored in the same memory address.

In older versions of python this was easily avoided with some trick, BUT in python3 this is always the case for both lists and arrays.

See `https://ecco-v4-python-tutorial.readthedocs.io/ECCO_v4_Operating_on_Numpy_Arrays.html`

You can check what happens with this example:

```
x=np.zeros(4,float)
x2=x
x[1]=1.0
print(x,x2)
```

where print gives as output

```
(array([ 0.,  1.,  0.,  0.]), array([ 0.,  1.,  0.,  0.]))
```

Both x[1] and x2[1] have changed!

If you want that x2 and x keep pointing two TWO DIFFERENT MEMORY ADDRESSES and you just want to make a copy of x into x2, you should use the function numpy.copy for numpy arrays (and copy.copy for the lists – note that copy.copy is a function of the package copy). Here below, you see an example for numpy arrays.

```
x=np.zeros(4,float)
x2=np.copy(x)
x[1]=1.0
print(x,x2)
```

where print gives as output

```
(array([ 0.,  1.,  0.,  0.]), array([ 0.,  0.,  0.,  0.]))
```

In this case, x2 is not affected by the changes on x.

With a more technical phrasing, this means that BOTH lists and numpy arrays (and in general objects in python) are stored by reference: you do not assign the value of x to x2, but a pointer to the object that x is pointing to.

## 2.10 *Comments*

Comments are parts of the code that are completely ignored by the computer. This does not mean they are useless. They are the only way for you to explain what you have done in a script. Imagine you read the same script, two years from now, without comments..Better to add comments, don't you think?

Here below an example.

```
import numpy as np #import numpy
a = [1.,2.] #define new list a
a1 = np.array(a) # transform a into an array a1
b = [3.,1.] #define new list b
b1 = np.array(b) # transform b into an array b1
c=a+b #sum the two lists
print(c) #print the sum of the two lists
c1=a1+b1 #sum the two arrays
print(c1) #print the sum of the two arrays
```

## 2.11 *if statement*

We use the **if** statement when we want to do something only if a given condition is met. For example

```
x=int(input("Enter an integer no greater than ten: "))
if(x>10):
     print("You entered an integer greater than ten.")
     print("Let me fix it for you.")
     x=10
print(x)
```

Note the usage of **indentation**, which is very strict in python. All the operations that must be performed only if x> 10 must be shifted to the right (through a **Tab** command) with respect to the line containing the if statement.

Here, we list some possible if conditions (but there are of course infinite conditions):

```
if(x==y):   checks if x is equal to y
if(x>y):    checks if x is larger than y
if(x>=y):   checks if x is larger than or equal to y
if(x<y):    checks if x is smaller than y
if(x<=y):   checks if x is smaller than or equal to y
if(x!=y):   checks if x is not equal to y
```

If you are familiar with C and C++, you immediately realised these symbols are identical in C and C++.

It may happen that we need an if statement to check more conditions. If we want all these conditions to be equally satisfied, we must connect these conditions with the *and* logical operator. If we want that at least one of these conditions is satisfied, we must connect them with the *or* logical operator. For example

```
x=4
if((x>1) and (x<3)):
     print("the if statement with and gives x=", x)
if((x>1) or (x<3)):
     print("the if statement with or gives x=", x)
```

We will not get the first print (because $x = 4$ satisfies $x > 1$ but does not satisfies $x < 3$), while we will get the second print in the form of

```
if statement with or gives x= 4
```

The symbol for logical and and logical or are different from C and C++ (which use && and ||).

## 2.12 *while statement*

Similarly to the if statement, the **while** statement checks if a condition is met. If it is met, the indented block is executed, otherwise it is skipped. Differently from the if statement, in the while statement, if the condition is met and the block is executed, the while statement loops back to the beginning of the statement and checks again if the condition is met. You cannot exit the **while** unless the condition is met. For example

```
x=10
if(x>2):
     x-=1
     print("We are inside the if, x=", x)
print("We are out of the if, x=", x)
x=10
while(x>2):
     x-=1
     print("We are inside the while, x=", x)
print("We are out of the while, x=", x)
```

gives as an output

```
  We are inside the if, x= 9
  We are out of the if, x= 9
  We are inside the while, x= 9
  We are inside the while, x= 8
  We are inside the while, x= 7
  We are inside the while, x= 6
  We are inside the while, x= 5
  We are inside the while, x= 4
  We are inside the while, x= 3
  We are inside the while, x= 2
  We are out of the while, x= 2
```

### 2.13 *for loops*

A for **loop** is the most commonly used loop in python (way more than while). It consists of a loop that runs though the elements of a list or array in turn. For example,

**Example 1**

```
r= [1., 3., 5.]
for n in r: #for loop over the element n in r
     print(n)
print("Loop ended")
```

gives as output

```
1.0
3.0
5.0
Loop ended
```

**Example 2**

```
r= [1., 3., 5.]
for i in range(len(r)): #for loop over the integer i from 0
# to the length of the list r (calculated with len())
     print(r[i])
print("Loop ended")
```

gives as output

```
1.0
3.0
5.0
Loop ended
```

## 2.14 *break and continue statements*

The **break** statement inside a loop allows to break out of a loop even if the condition in the while (or for) statement is not met. For example

```
x=10
while(x>2):
     x-=1
     print("We are inside the while, x=", x)
     if x==5:
          break
print("We are out of the while, x=", x)
```

gives output:

```
We are inside the while, x= 9
We are inside the while, x= 8
We are inside the while, x= 7
We are inside the while, x= 6
We are inside the while, x= 5
We are out of the while, x= 5
```

This statement is **nested** inside the while loop.

The **continue** statement inside a loop allows to skip the rest of the indented code in the loop and then goes back to the beginning of the loop. For example

```
x=10
while(x>2):
     x-=1
     print("We are inside the while, x=", x)
     if x==5:
          x-=1
          continue
     print("We are after the continue")
print("We are out of the while, x=", x)
```

gives output:

```
We are inside the while, x= 9
We are after the continue
We are inside the while, x= 8
We are after the continue
We are inside the while, x= 7
We are after the continue
We are inside the while, x= 6
We are after the continue
We are inside the while, x= 5
We are inside the while, x= 3
We are after the continue
We are inside the while, x= 2
We are after the continue
We are out of the while, x= 2
```

From the above example, we see that when the condition of the if is met and then we find the “continue” the loop goes back to the beginning without printing the line "We are after the continue”.

## 2.15 *Dictionaries*

A dictionary is a collection of information, which is unordered, changeable and indexed. It is the last kind of container that we see. We have not seen it before, because it is a bit more complicated to handle than lists and arrays. For people who already know C and C++, it is similar to a structure in C/C++. It is created with curly brackets, for example:

```
mycat = {
   "color": "red",
   "fur": "short",
   "spots": "tabby"
}
```

This dictionary defines my cat's properties. On the left, "color", "fur" and "spots" represent the **keys**, i.e. the categories which define my dictionary and to which we want to assign a value. On the right, after the colon, you have the **values** assigned to each key: my cat has red color, short fur and is a tabby because of the structure of his spots.

In this case all values are strings, but they can be any type of variables.

*Operations you can do with dictionaries:*

- you can print a dictionary with print(mycat);
- you can access the items of a dictionary by referring to their key names, as

```
x = mycat["color"]
```

  If you print the variable x, you see that its value has become 'red'. Alternatively, you can do the same by using the function get()

```
x = mycat.get("color")
```

- you can change the values of a specific item by doing, e.g.:

```
mycat["color"] = 'black'
```

  If you now print the dictionary mycat you see that mycat's color has become black;

- you can loop over the keys of a dictionary

```
for x in mycat:
   print(x)
```

  returns color, fur and spots;

- or you can loop over the values of a dictionary

```
for x in mycat.values():
   print(x)
```

  returns red, short and tabby;

- or you can loop over both keys and values:

```
for x,y in mycat.items():
   print(x, y)
```

- you can check whether a key exists in a dictionary:

```
if "color'' in mycat:
   print(mycat["color"])
```

- you can add a new key to an existing dictionary:

```
mycat["age"] = 7
print(mycat)
```

- and you can remove a key from a dictionary:

```
mycat.pop("age")
print(mycat)
```

- you can copy a dictionary into another dictionary:

```
  yourcat = mycat.copy()
```

  In principle, you can do also yourcat = mycat, but, as we already said for numpy arrays and lists, this does not make a copy; rather, it creates a new dictionary yourcat that points to the same memory address as mycat: the two of them change together since then;

- You can create a dictionary that contains several dictionaries (**nested dictionaries**):

```
mycats = {
   "ettore" : {
   "color" : "white",
   "fur" : "short",
   "age" : 10
   },
   "ezzelino" : {
   "color" : "red",
   "fur"   : "short",
   "age"   : 7
   }
}
```

It is also possible to create the two dictionaries separately and then assign them to the more general dictionary:

```
ettore = {
   "color" : "white",
   "fur" : "short",
   "age" : 10
}
ezzelino = {
   "color" : "red",
   "fur"   : "short",
   "age"   : 7
}
mycats = {
   "ettore" : ettore,
   "ezzelino" : ezzelino
}
```

- you can construct a dictionary using the dict() function:

```
mycat = dict(color="red", fur="short", spot="tabby", age=7)
```

Dictionaries are important to know if you plan to use **pandas** (see one of the later chapters in these lectures).

## 2.16 *Functions*

Functions in python (i.e. sets of instructions) are called **built-in functions** when they are always available with the default version of python, without the need for importing a package. For example, the *print* function is a built-in function.

Other functions live in **packages**. They need a package to be imported, in order to be ready for usage. For example, the *log* function is part of the *math* package and of the *numpy* package.

Finally, **user-provided functions** are explicitly defined by the programmer (possibly because they are custom functions available in no package), with the following syntax

```
#see examples/python/simple_def.py

#I define a function that calculates the squares
def simple(x):
    simplef=x**2
    return simplef

#The main
number=0.0
while(number<10.):
        power2=simple(number)
        print(power2) #square of first 10 integers
        number+=1
```

In the above example, I define the function simple(x), which takes one argument (x) and returns a float (simplef). This function calculates the square of the input argument. I then call simple in the main of my script.

Let us now consider the following much more complex example. We want to define and use a function that calculates the lookback time, i.e. (in a ΛCDM cosmology where the curvature parameter $\Omega_K = 0$):

$$t_{lb} = \frac{1}{H_0} \int_0^z \frac{dz'}{(1+z') \left[(1+z')^3 \Omega_M + \Omega_\Lambda\right]^{1/2}}, \tag{1}$$

where $H_0$ is the Hubble constant, $\Omega_M$ and $\Omega_\Lambda$ are the density parameter of (baryonic+dark) matter and the effective density parameter of dark energy, respectively, according to the ΛCDM model. The lookback time represents the time elapsed from now to redshift $z$ (that is lookback time corresponding to present time is 0, while lookback time corresponding to redshift $z = 1$ is approximately 8 Gyr ago).

The example script to calculate lookback time is the following.

```
#see examples/python/lookback.py

import scipy.integrate as integrate
#integrate is the module of scipy
#that performs numerical integration

def looktime(z): #definition of new function looktime.
# This function takes one argument (z)

    H0=67.2 #Hubble constant in km/s/Mpc
    convert=1e5/3.086e24*3.1536e7*1e9
    #converts from km/s/Mpc to Gyr
    OmegaM=0.2726   #omega matter, parameter from cosmology
    OmegaL= 0.7274  #omega lambda, parameter from cosmology

    integrand = lambda x: 1./((1.+x)*(OmegaM*(1.+x)**3.+ \
                OmegaL)**0.5)
    #integrand is the function I want to integrate
    #between 0 and z

    ltime=integrate.quad(integrand, 0.0, z)
    #ltime is an array of 2 elements:
    # ltime[0] = result of integral
    # ltime[1] = error

    look=ltime[0]
    look/=(H0*convert)

    return look #function looktime returns look, which is
    #the lookback time at redshift z in comoving framework


#the main

z=10.   #initial redshift
look=[] #look-back time list
redsh=[] #redshift list
while(z>0.0):
    look.append(looktime(z)) #call looktime and append
                             #result to the list look
    redsh.append(z) #store z into list redsh
    z-=0.1
for i in range(len(look)): #loop over the elements of look
    print(redsh[i], look[i]) #prints redshift and
                     #corresponding look back time list
```

Function looktime takes redshift $z$ as input and returns the lookback time (variable look) corresponding to $z$. Note that we jumped a bit ahead of schedule with this example, because

1. we used the package **scipy** that contains several scientific tools in python, for example tools for numerical integration in the module **scipy.integrate**;

2. from scipy.integrate we called the function **quad** that integrates functions numerically. In particular, quad uses a technique from the Fortran library QUADPACK, which consists in a Clenshaw-Curtis method which uses Chebyshev moments (see e.g. `https://en.wikipedia.org/wiki/Clenshaw-Curtis_quadrature`);

3. the function we want to integrate (integrand) with quad can be written as

```
lambda x: 1./((1.+x)*(OmegaM*(1.+x)**3.+OmegaL)**0.5)
```

in particular "lambda x:" must precede the function and indicates that we are defining a small function without using the def formalism. Hence the above notation with lambda x is almost equivalent to defining a new function as

```
def integrand(x):
    OmegaM=0.2726   #omega matter
    OmegaL= 0.7274  #omega lambda
    f=1./((1.+x)*(OmegaM*(1.+x)**3.+OmegaL)**0.5)
    return f
```

Note that "lambda" in the notation "lambda x:" tells us that x is the variable.

We can use lambda also to define functions with multiple variables (e.g. x and y), by separating them with a comma. For example "lambda x,y :" indicates that the function defined after the colon has two variables x and y.

The program would have worked even defining the function integrand with def (see the alternative example examples/python/lookback2.py). In this case, the syntax should have been:

```
def integrand(x):
    OmegaM=0.2726   #omega matter
    OmegaL= 0.7274  #omega lambda
    f=1./((1.+x)*(OmegaM*(1.+x)**3.+OmegaL)**0.5)
    return f


def looktime(z): #definition of new function looktime.
# This function takes one argument (z)

    H0=67.2 #Hubble constant in km/s/Mpc
    convert=1e5/3.086e24*3.1536e7*1e9
    #converts from km/s/Mpc to Gyr


    ltime=integrate.quad(integrand, 0.0, z)
    #ltime is an array of 2 elements. ltime[0]= result of integral
    #ltime[1] error

    look=ltime[0]
    look/=(H0*convert)
    return look #function looktime returns look, which
    #is the look back time at redshift z in comoving framework
```

Note that when integrate.quad calls integrand(x) we do not need to specify the variable. Even more: if we specify it we get an error.

See

`https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.quad.html`
for more details on integrate.quad.

**EXERCISE:**

Write a python script to calculate the comoving distance and the luminosity distance given the redshift. Use scipy.integrate.quad for the integration (as in the previous example).
Suggestion: the expression of the comoving distance (if $\Omega_K = 0$) is the following:

$$D_C(z) = \frac{c}{H_0} \int_0^z \frac{dz'}{\left[(1+z')^3\,\Omega_M + \Omega_\Lambda\right]^{1/2}}, \qquad (2)$$

The expression of the luminosity distance (if $\Omega_K = 0$) is the following:

$$D_L(z) = \frac{c}{H_0}\,(1+z) \int_0^z \frac{dz'}{\left[(1+z')^3\,\Omega_M + \Omega_\Lambda\right]^{1/2}} = (1+z)\,D_C(z) \qquad (3)$$

The final remark about user-provided functions is that if you want to use a function you have created in a script also in another script you can. You just import the name of the script where the function is. For example, if you want to import the function which calculates the lookback time, you can do as in lookback3.py (which contains only the defined functions, but not the main) and in lookback3_main.py (which imports the functions defined in lookback3.py). In particular, note the import statement in lookback3_main.py:

```
from lookback3 import *
```

With this statement you import all the functions (*) from lookback3.py.

This is a very nice practice to do if your script has a lot of user-provided functions that make it difficult to read (you have to scroll thousands of lines before you reach the main): just move all of your functions to another file and then import them.

The main caveat is that if you are very unlucky, you might call one of your functions in the same way as an already existing package, function or library in python. In this case, the user provided function overrides the existing package, function or library. For example, if I had called my lookback3.py file numpy.py (this is quite an extreme example..), the functions inside my numpy.py would have completely overridden the numpy package. Don't worry: the problem disappears when you cancel your own file numpy.py.

## 2.17 *Reading from and writing to files*

So far, we have seen how to read inputs from command line, but usually inputs must be read from files. Similarly, you might want to write your outputs in a file rather than printing them in the terminal. Here, we consider only the simple case of plain ASCII files.

The simplest way to read the content of an ascii file in python is with the numpy function **loadtxt** (or the similar function **genfromtxt**). Example of usage: I have to read a file (file name: mass_evol.txt, see the examples/python) with 20 columns but I am interested only in four of them (the 0th, the 1st, the 3rd and the 5th) and I want to copy them in arrays of type float. Moreover, I do not want to read lines starting with #, because I know these are comments to be skipped. Then I use

```
#see examples/python/read_file.py
import numpy as np

fname="mass_evol.txt"  #input the filename as a string

time,m,mHe,mCO = np.genfromtxt(fname,dtype="float", \
comments="#", usecols=(0,1,3,5), unpack=True)

print(m)
```

**fname** The first argument of genfromtxt is the name of the input file (defined as a string). If the filename extension is gz or bz2, the file is first decompressed.

**dtype="float"** Choose type of variables. Optional.

**comments="#"** The character used to indicate the start of a comment. All the characters occurring on a line after a comment are discarded. Optional.

**usecols**= (0, 1, 3, 5) Which columns to read (the other ones are ignored). Optional.

**unpack=True** Splits the output per columns, each column is assigned to the numpy arrays time, m, mHe ,mCO. Optional but very useful.

For more details on genfromtxt, see

`https://docs.scipy.org/doc/numpy/reference/generated/numpy.genfromtxt.html`.

Unfortunately, loadtxt and genfromtxt are terribly slow and memory consuming. If you have very large files, it is not smart to use them. As an alternative, you can write your own custom function to read an ascii file. For example, the following user-provided function does exactly the same as the former example of genfromtxt.

```
#see examples/python/read_file2.py
def readfast(fname, N):
    f=open(fname,"r")
    (time,m,mHe,mCO) = (
        np.zeros(N,dtype="float"),
        np.zeros(N,dtype="float"),
        np.zeros(N,dtype="float"),
        np.zeros(N,dtype="float"))
    i=0
    for linetext in f:
        if(linetext[0]==str("#")):
            continue
        word_list = linetext.split()
        #split splits a line in elements
        if(word_list[0]!=str("#")):
            time[i]=np.float(word_list[0])
            m[i]=np.float(word_list[1])
            mHe[i]=np.float(word_list[3])
            mCO[i]=np.float(word_list[5])
            i=i+1
    #end [ for linetext in f ]
    f.close()
    return (time, m, mHe,mCO)
```

In the main text, I call the above function as follows:

```
fname="mass_evol.txt"  #input the filename as a string
N0= 103 #number of lines of the file
(evoltime,mass,massHe,massCO) = readfast(fname,N0)
```

Of course, you can write your own version of this user provided function basically as you want.

The simplest way to write output in an ASCII file is to use the function **write**. For example, here I want to write a file that has four columns, each of them containing the arrays *evoltime*, *mass*, *massHe*, *massCO* of the above example. I also add a first line of comment.

```
#see examples/python/read_file2.py
fname="output.txt" #output filename as a string
f=open(fname,"w")  #I create file with filename
                   #the new file will be for writing (w)
f.write("# Time(Myr), Mass(Msun), He(Msun), CO(Msun) \n")
for i in range(len(evoltime)):
     f.write(str(evoltime[i])+" "+str(mass[i])+" "+ \
str(massHe[i])+" "+str(massCO[i])+"\n")
```

Note that write can take as arguments only strings.

## 2.18 *Subprocess*

Python allows you to call linux commands from python scripts, if needed. This happens through the **subprocess** module. By importing subprocess, you can call linux commands as if you were using a command line.

For example

```
import subprocess
subprocess.call(['df', '-h'])
```

is equivalent to the linux command df -h. I can do the same with basically every linux command.

## 2.19 *Regular expressions*

Often, the files you have to read are very simple: they are just tables with numbers ordered in rows and columns. Sometimes, you have to read files where some of the columns/rows are strings and some other columns/rows contain numbers. Finally, in a small but non-negligible number of cases, the files you are supposed to read look like a messy distribution of strings and numbers in a chaotic way (no regular rows and columns). Regular expressions is the best way to handle the last case and is a good option (in terms of memory and reading speed) for the other two cases.

A regular expression (or RE) specifies a string or a set of strings that you want to look for in a file. In python, the module **re** allows you to use regular expressions. Let's start from an example to make it clearer.

```
#see examples/python/regex.py
import os #This module provides a portable way of
          # using operating system dependent functionality
          # it is needed to call re
import re # regular expression module

m = re.compile('^The mass is 3 Msun')
# the string I want to look for: the mass is 3 Msun
# ^ means that the string should be at the start of a row

fname=str('file_name.txt')
f=open(fname)

for s in f: #s is a generic string in f
     tosearch = m.search(s) # I search for string m in s
     if(tosearch != None):
          print(tosearch)
```

If you run the above simple code with python3 you get something strange like

```
<re.Match object; span=(0, 18), match='The mass is 3 Msun'>
```

This is because tosearch is a new object. So far, this thing looks a bit awkward and useless, but now we get back to our example and we add a new feature:

```
#see examples/python/regex2.py
import os #This module provides a portable way
          # of using operating system dependent functionality
          # it is needed to call re
import re # regular expression module

m = re.compile('^The mass is (\d+) Msun')
# the string I want to look for:
# The mass is some integer number Msun


fname=str('file_name.txt')
f=open(fname)

for s in f: #s is a generic string in f
     tosearch = m.search(s) # I search for string m in s
     if(tosearch != None):
             mass = tosearch.group(1)
             print(mass)
```

Here I still find the same string as with the previous script ('The mass is 3 Msun'), but I could have also found other strings ('The mass is 5 Msun' or

'The mass is 7 Msun'), i.e. the command

```
m = re.compile('^The mass is (\d+) Msun')
```

in this new script allows me to search for strings with different numbers of the mass in Msun. The reason is that I have substituted '3' with '\d+'.

Here \ indicates that what follows (in this case 'd+') is not a string but a command of the regular expression. As a command of re, 'd' indicates any possible integer digit. The '+' symbol after 'd' indicates that I am searching only strings where there is AT LEAST ONE integer digit in that position of the string (but there might be many more). If instead of '\d+' I had indicated '\d', the script would have looked only for sentences where the integer is a single digit. If I had indicated '\d*', the script would have looked for sentences where there are integers (one or more digits) in that position but also for sentences where there are not integers in that position. Thus, what comes before the symbol '*' might or might not be there.

The regular expressions have many other similar symbols and we will not have time to discuss all of them. Just look here `https://docs.python.org/2/library/re.html` for more details.

Let's just mention one important thing. Everything in regular expressions breaks down to two symbols '\s' and '\S'. The former means 'space' the latter means 'non-space', that is every possible character (string, integer, float, mixture of them) which is not a space. Whatever possible string or number in the world can be written as a given sequence of spaces and non-spaces.

The next important feature of the example is that '\d+' is within round brackets '()'. When some part of the regular expression is put within round brackets, it means that it becomes a variable. To read this variable, we should then use the function 'group($i$)' where $i = 0$ means the entire string found by the regular expression, while $i = 1, 2, 3, ...$ refer to the first, second, third, etc variable put within brackets. This explains why

```
mass = tosearch.group(1)
print(mass)
```

in the above example produces as print '3'.

One last thing to be added about regular expressions is their possible usage as fast tool to read large ascii files. Suppose that we have a simple file with 5 columns separated by spaces. The problem is that the file is very long and genfromtxt/loadtxt would require too much memory to read it. One option is to open the file and read it using the regular expressions as (for example):

```
import os
import re

m = re.compile('^(\S+)\s+(\S+)\s+(\S+)\s+(\S+)\s+(\S+)')

fname=str('five_columns.txt')
f=open(fname)

col1=[]
col2=[]
col3=[]
col4=[]
col5=[]

for s in f:
     tosearch = m.search(s)
     if(tosearch != None):
             col1.append(tosearch.group(1))
             col2.append(tosearch.group(2))
             col3.append(tosearch.group(3))
             col4.append(tosearch.group(4))
             col5.append(tosearch.group(5))
```

# 3 VISUALIZATION

The examples corresponding to this section are in examples/python/.

The material needed for the exercises is in exercises/python/.

Python is a powerful tool for visualization. The main problem is that there are even too many tools in python for visualization and one can get lost among them. The first and most important suggestion is to start from the examples that can be browsed over the internet: python is very popular nowadays and basically all possible tools for visualization are already discussed in forums, blogs, userguides and whatever. Plus, you cannot remember the syntax of every python command by hart..Even if you have an elephant memory, there are better things to do than to memorize python commands.

Here I will provide just few essential examples, and I will leave to the students the task of finding more over the internet. My preferred (and possibly the most used) plotting package from python is **pyplot**, which is a sub-package of **matplotlib**. The following examples are drawn from `https://matplotlib.org/users/pyplot_tutorial.html`, where you can find many more details.

## 3.1 *Scatter plot*

The following script is an example of a very simple scatter plot

```
#see examples/python/simple_plot.py
import matplotlib.pyplot as plt
plt.plot([1,2,3,4], [1,4,9,16], color='r',marker='o')
plt.xlim(0, 6)
plt.ylim(0, 20)
plt.xlabel('x', fontsize=20)
plt.ylabel('y', fontsize=20)
plt.show()
```

This produces as an example Figure 6. In the above example, plot takes four arguments: the list of values on the $x$ axis, the list of values on the $y$ axis and the characteristics of the data points: the color **color='r'**, where 'r' stands for red and the marker type **marker='o'**, where 'o' stands for circles.

I can write the definition of color and marker in a more compact way as

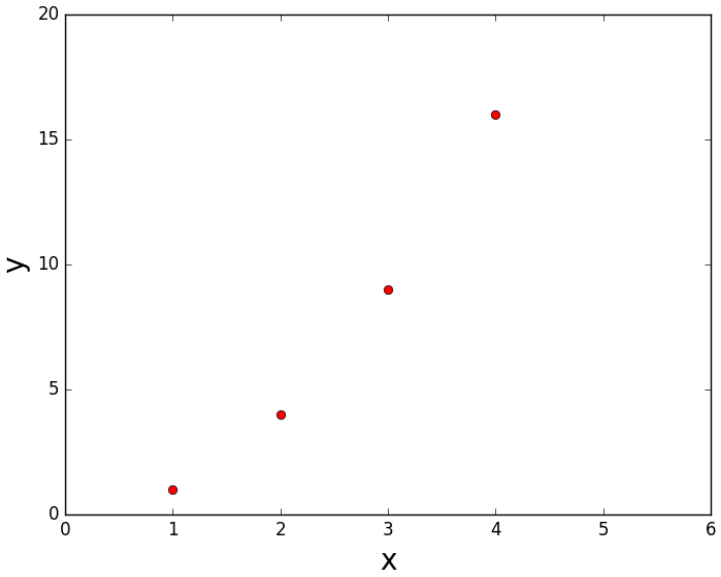


*Figure 6: The simplest example of a scatter plot.*

```
#see examples/python/simple_plot.py
import matplotlib.pyplot as plt
plt.plot([1,2,3,4], [1,4,9,16], 'ro')
plt.xlim(0, 6)
plt.ylim(0, 20)
plt.xlabel('x', fontsize=20)
plt.ylabel('y', fontsize=20)
plt.show()
```

Here, I have just 3 arguments of plot: the x values, the y values and the argument 'ro' where r stands for red and o indicates the markers. This plot is absolutely identical to the previous example.

In the following table, you can find other possible examples of markers that can be obtained with matplotlib.pyplot.plot:

| character | description |
|---|---|
| - | solid line style |
| -- | dashed line style |
| -. | dash-dot line style |
| : | dotted line style |
| . | point marker |
| , | pixel marker |
| o | circle marker |
| v | triangle-down marker |
| ^ | triangle-up marker |
| < | triangle-left marker |
| > | triangle-right marker |
| 1 | tri-down marker |
| 2 | tri-up marker |
| 3 | tri-left marker |
| 4 | tri-right marker |
| s | square marker |
| p | pentagon marker |
| * | star marker |
| h | hexagon1 marker |
| H | hexagon2 marker |
| + | plus marker |
| x | x marker |
| D | diamond marker |
| d | thin-diamond marker |
| \| | vline marker |
| _ | hline marker |

An alternative function to produce scatter plots is 'scatter' and works as in the example:

```
#see examples/python/simple_plot.py
import matplotlib.pyplot as plt
plt.scatter([1,2,3,4], [1,4,9,16], color='r',marker='o')
plt.xlim(0, 6)
plt.ylim(0, 20)
plt.xlabel('x', fontsize=20)
plt.ylabel('y', fontsize=20)
plt.show()
```

The first two arguments of scatter are the same as before. The main difference is that the color should be indicated after the command 'color=' and the marker after the command 'marker='. The symbols used to indicate colors and markers are the same as already indicated in table 3.1.

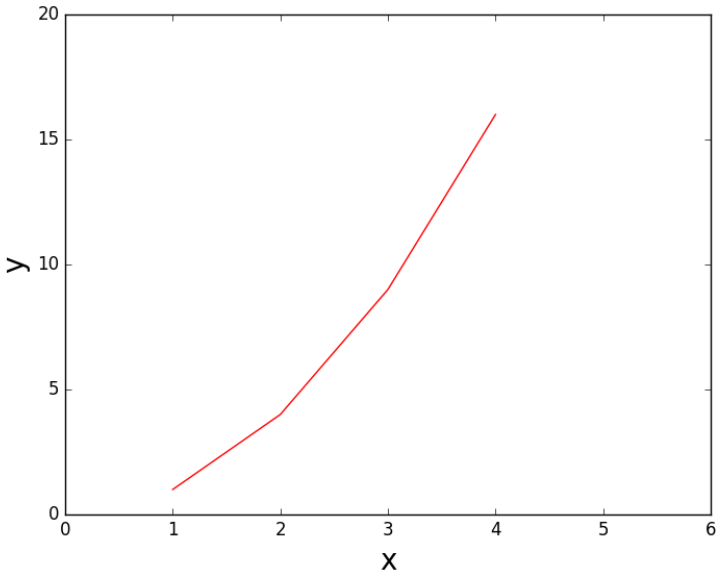


*Figure 7: The simplest example of a line plot.*

## 3.2 *Line plot*

The function 'plot' can be used even for line plots, by substituting the symbol of the marker with a symbol for a line type:

```
#see examples/python/simple_plot.py
import matplotlib.pyplot as plt
plt.plot([1,2,3,4], [1,4,9,16], 'r-')
plt.xlim(0, 6)
plt.ylim(0, 20)
plt.xlabel('x', fontsize=20)
plt.ylabel('y', fontsize=20)
plt.show()
```

In the above example (see Fig. 7) the circles have been substituted by a solid line (indicated by the symbol '-'). The other standard line types are the following.

| character | description |
|---|---|
| - | solid line style |
| -- | dashed line style |
| -. | dash-dot line style |
| : | dotted line style |

## 3.3 *Logarithmic axes*

If you want to make logarithmic axes simply add

```
plt.yscale('log')
```

for the y axis, and

```
plt.xscale('log')
```

for the x axis.

## 3.4 *Annotating text*

If you want to annotate some text you can use the function 'text'. For example

```
plt.text(1.8, 2.0, '$\mathrm{Log-log\,{}plot}$',fontsize=17)
```

The aforementioned command writes the sentence 'Log-log plot' starting from the point at $(1.8, 2.0)$ in the Cartesian plane of the aforementioned example. Hence, the first two entries of text are the x and y location of the annotated text, then you write the text you want to annotate as a string (between the inverted commas "), finally you might indicate the size of the annotated text.

In this specific example the text is between dollars ($$). **If you put some text between dollars, you can use almost the same commands as mathematical latex. This is particularly useful for symbols. For example the symbol of the solar mass is $\odot$ even in matplotlib. There are a few differences with math latex, which you will find out case by case.**

## 3.5 *Legend*

When multiple lines or data sets are present, it is always a good idea to add a legend. This is an example of how to set up a legend:

```
#see examples/python/simple_plot.py
import numpy as np
import matplotlib.pyplot as plt
x=np.zeros(1000,float)
y=np.zeros(1000,float)
y2=np.zeros(1000,float)

for i in range(1,len(x)):
    x[i]=x[i-1]+1.
    y[i]=np.sin(x[i])
    y2[i]=np.cos(x[i])

a= plt.scatter(x,y, color='b',marker='^')
b= plt.scatter(x,y2,color='r',marker='o')
plt.legend([a,b], ['sin(x)','cos(x)'],fontsize='20', \
    loc='upper right')

plt.show()
```

The first argument of legend is the set of graphs for which I need a legend entry, i.e. $[a, b]$. If you do not provide this argument, python looks for all the things that you are plotting and set up an entry in the legend for each of them. The second argument is the set of labels associated to legend entries, i.e. $['sin(x)','cos(x)']$. Then, I can add other optional arguments such as font size (very important!) and location (with loc). If you do not enter any location or you do not set the fontsize, python will choose a location and a fontsize based on its judgment (which is not always smart). For other possible arguments, see e.g. `https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.legend.html`.

The above script produces the plot shown in Figure 8.

## 3.6 *Note on fontsize*

By default, matplotlib chooses a fontsize (of the axis labels, of the numbers on the axis, of the legend, etc) that is too small for scientific literature: when your plot is included into the draft a scientific journal article (e.g. the Astrophysical Journal or the Monthly Notices of the Royal Academic Society), the labels are too small to be easily read. Then, it is important to increase the fontsize with respect to the default. You can always set a fontsize in each function of your plot – for example, a fontsize in the xlabel(), ylabel(), text() and legend() function. However, sometimes it is quite boring to add fontsize in each single function.

If you want to set up a basic fontsize in your plot, which is used by default in each function, you can use the command

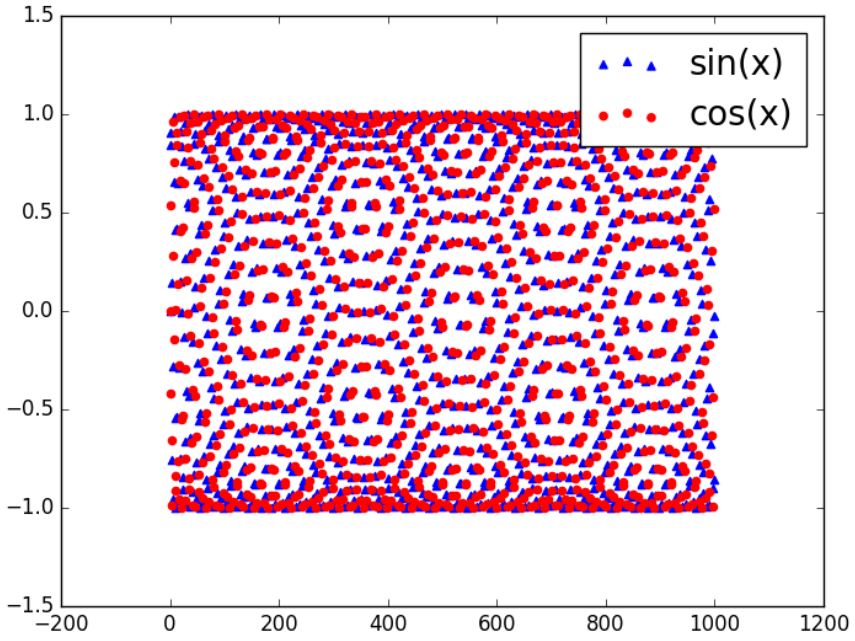


*Figure 8: A plot with a legend.*

```
plt.rcParams.update({'font.size': 20})
```

In general, rcParams.update() allows you to update several default parameters of matplotlib, such as the font type, the default text and background color. As you can recognize from the example, rcParams is nothing but a dictionary that contains the information on these parameters. The function .update() allows to change the value corresponding to the chosen key. In the example, you want that the key font.size contains the new value 20.

If you want to use a different fontsize for a specific function, for example if you want to use a smaller font in the legend with respect to the axis labels, you just write down a different fontsize among the arguments of a function – e.g., legend(fontsize='15'). Writing down the fontsize among the arguments of a single function overrides the rcParams for that specific function.

## 3.7 *Colors in python*

Colors in python can be used and defined in many (too many..) ways. The **default** color cycle is very simple: 'b' (blue), 'g' (green), 'r' (red), 'c' (cyan), 'm' (magenta), 'y' (yellow), 'k' (black).

Of course, you can define more colors. I will give just one example of how this can be done:

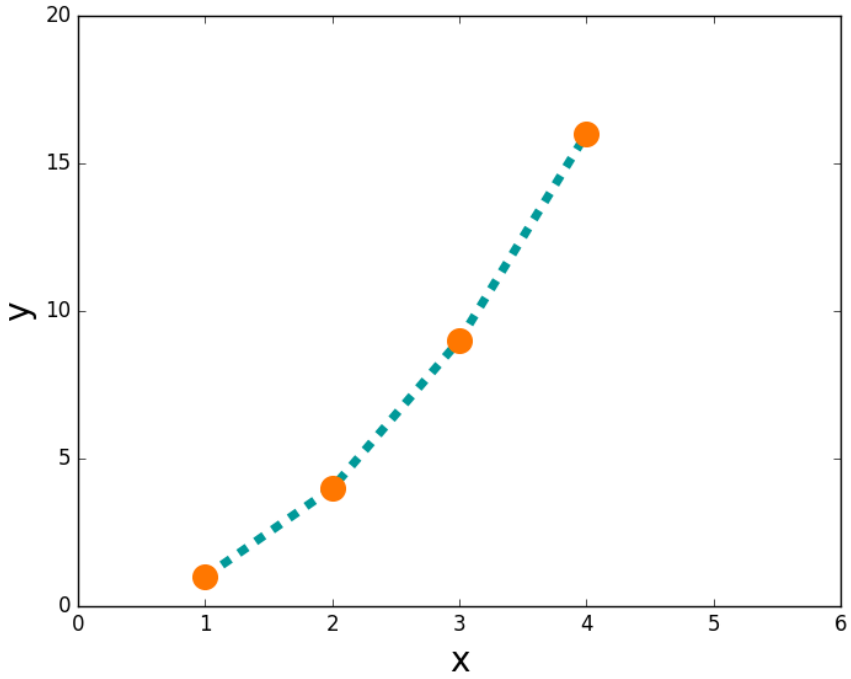


*Figure 9: A scatter and line plot with definition of new colors and some additional features (e.g. s=200 defines the size of the marker, zorder defines the order of overlaps between lines).*

```
#see examples/python/simple_plot.py
import matplotlib.pyplot as plt
plt.scatter([1,2,3,4], [1,4,9,16], color='#FF7700', \
marker='o', s=200, zorder=2)
plt.plot([1,2,3,4], [1,4,9,16], color='#009999',\
linestyle='--', linewidth='5', zorder=1)
plt.xlim(0, 6)
plt.ylim(0, 20)
plt.xlabel('x', fontsize=20)
plt.ylabel('y', fontsize=20)
plt.show()
```

The above script produces Fig. 9. This Figure, is based on the same data set as first two examples, but now the color of the circles is described by color='#FF7700' and the color of the line is described by color='#009999'. The two strings '#FF7700' and '#009999' define a new orange and a new sea blue. For a string to be interpreted as a color properly, it should be built in the following way:

- First the symbol # which means that we are building a color.

- The first two digits after the # refer to the amount of red in the RGB mixture (see the additive color theory): 00 means dark red (almost black), 99 means the most luminous red, FF means even more luminous than 99.

- The second two digits after the # refer to the amount of green in the RGB mixture (see the additive color theory): 00 means dark green (almost black), 99 means the most luminous green, FF means even more luminous than 99.

- The last two digits refer to the amount of blue in the RGB mixture (see the additive color theory): 00 means dark blue (almost black), 99 means the most luminous blue, FF means even more luminous than 99.

Hence, the mixture between red, green and blue channels should be done following the additive color theory. For example, '#000000' means black (subtraction of all colors with minimum luminosity), while '#FFFFFF' means white (sum of all the colors with the maximum luminosity). Is there a way to find out what colors correspond to a given combination of these six digits? Basically not, just try. If you find a nice color, keep memory of its definition.

If you use **matplotlib**, you have a nice set of pre-defined colors with names (as represented in Figure 10). Be careful, because the list might change from one version to the other of matplotlib (see the named_colors.py script).

One important color tool you will surely need is **color maps**, i.e. palettes of several colors that can be used, e.g., to produce two-dimensional histograms and contour plots. Matplotlib has a plethora of color maps and more maps can be added with customized libraries. For a list of color maps in recent versions of matplotlib see e.g. `https://matplotlib.org/3.1.0/tutorials/colors/colormaps.html`. One thing we should always check when choosing the color map for a plot is that the lightness of sequential colors in the colormap increases (or decreases) in a monotonic way and possibly in a linear way. This makes the plot readable also by color-blind people or other people with a different perception of colors. Figure 11 is a summary of the color maps that are perceptually uniform and of the color maps that are sequential (i.e. monotonic in their lightness). In contrast, Figure 12 is an example of maps that are strongly non sequential (among them, the famous jet map that is still used when Doppler shifts or metallicity are considered).

**EXERCISE:**

Write a script to plot comoving distance, luminosity distance and look-back time (derived from the previous exercise and example) as a function of redshift. The results should look like Figure 13.

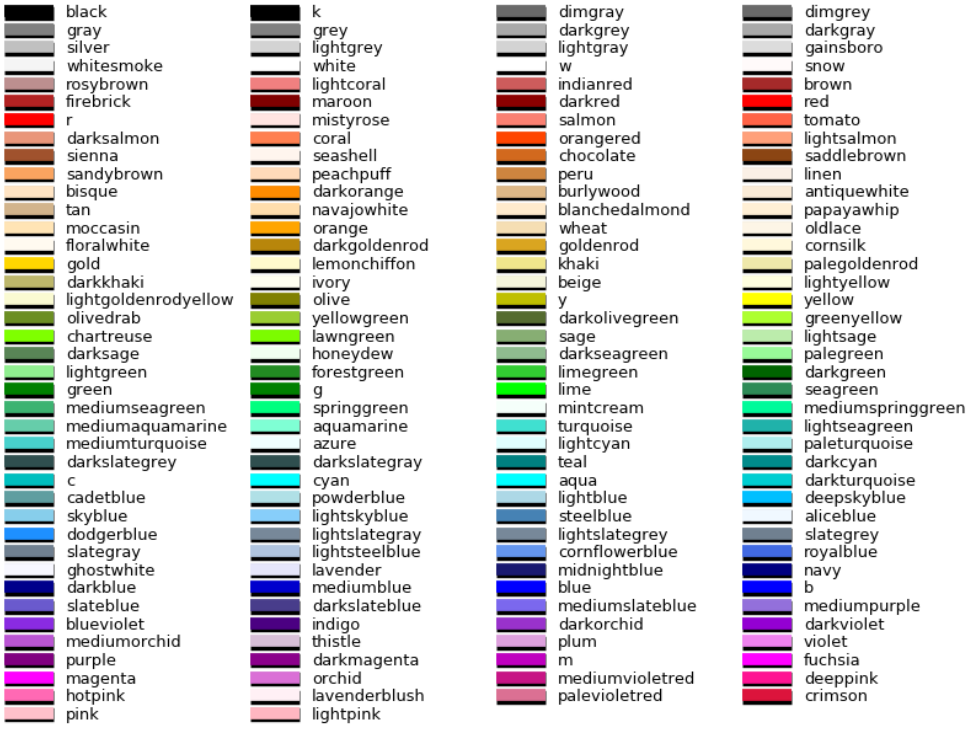


*Figure 10: Colors with names in one of the many versions of matplotlib: check yours.*

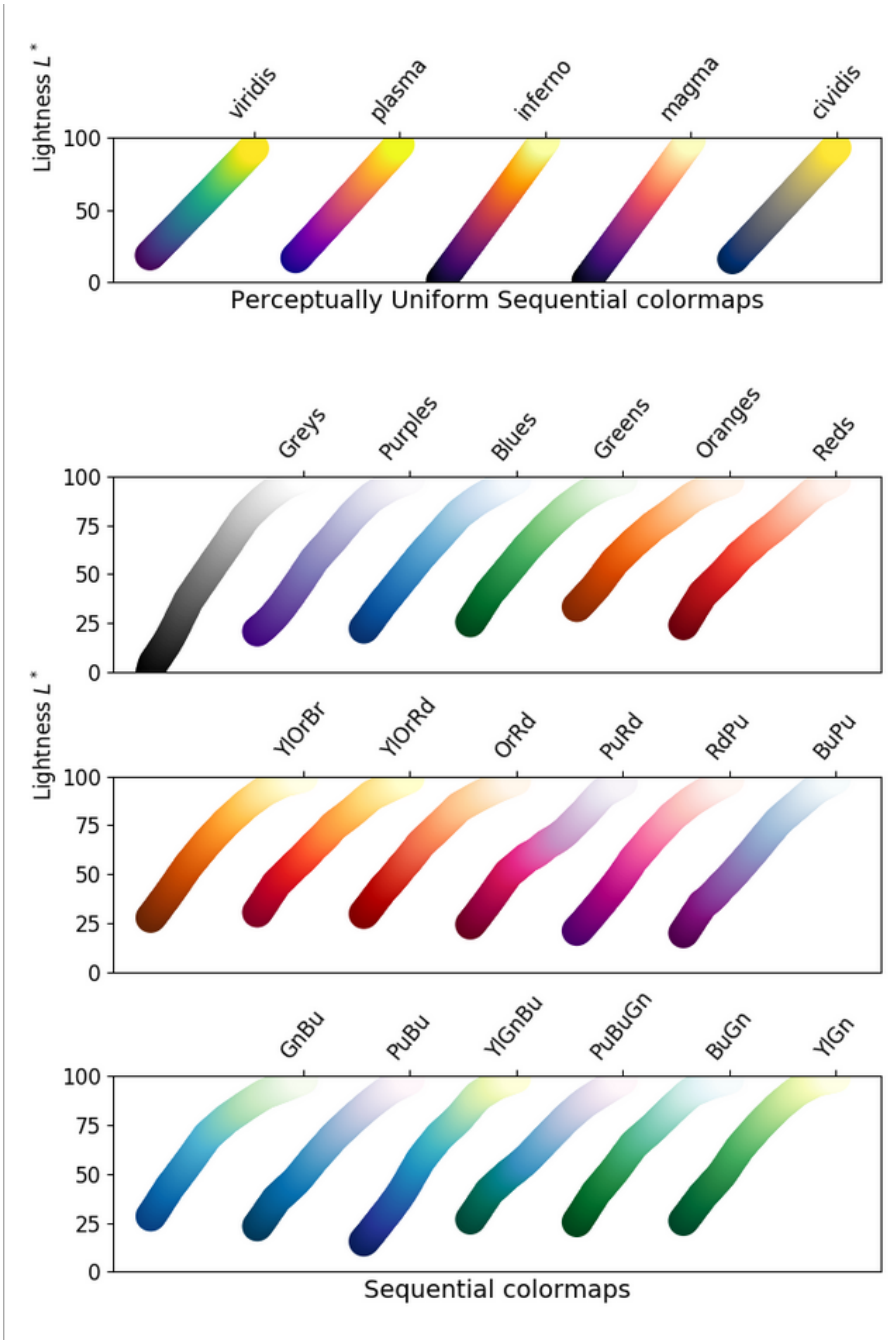


*Figure 11: Matplotlib color maps that are fine for color-blind people.*

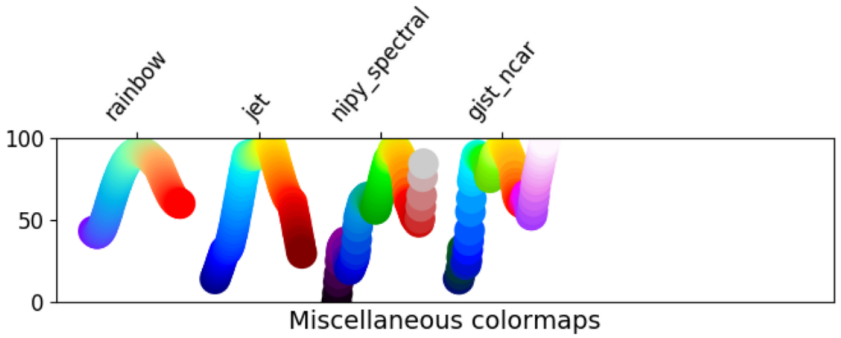


*Figure 12: Examples of matplotlib color maps that do not have a monotonic trend.*

## 3.8 *Histogram*

The main (but not the only) command to create an histogram is hist. See (and try to run) the example below:

```
#see examples/python/simple_plot.py
import numpy as np
import matplotlib.pyplot as plt
x=np.zeros(1000,float)
y=np.zeros(1000,float)

for i in range(1,len(x)):
    x[i]=x[i-1]+1.
    y[i]=np.sin(x[i])

plt.hist(y, bins=60, density='True',histtype='step')
plt.show()
```

The above script produces the plot shown in Figure 14.

In the example, we first create a sample of data points (stored in the array *y*) and then we call hist to produce an histogram. The only argument needed by histogram is the array (or list) of data for which we want to produce the histogram. Then, hist takes a number of optional arguments, some of which are explained below (for all the arguments see e.g. `https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.hist.html`).

bins: can be an integer or a sequence (array, list), optional. If an integer is given, bins + 1 bin edges are calculated and returned, consistent with numpy.histogram. If bins is a sequence, gives bin edges, including left edge of first bin and right edge of last bin. In this case, bins is returned unmodified.

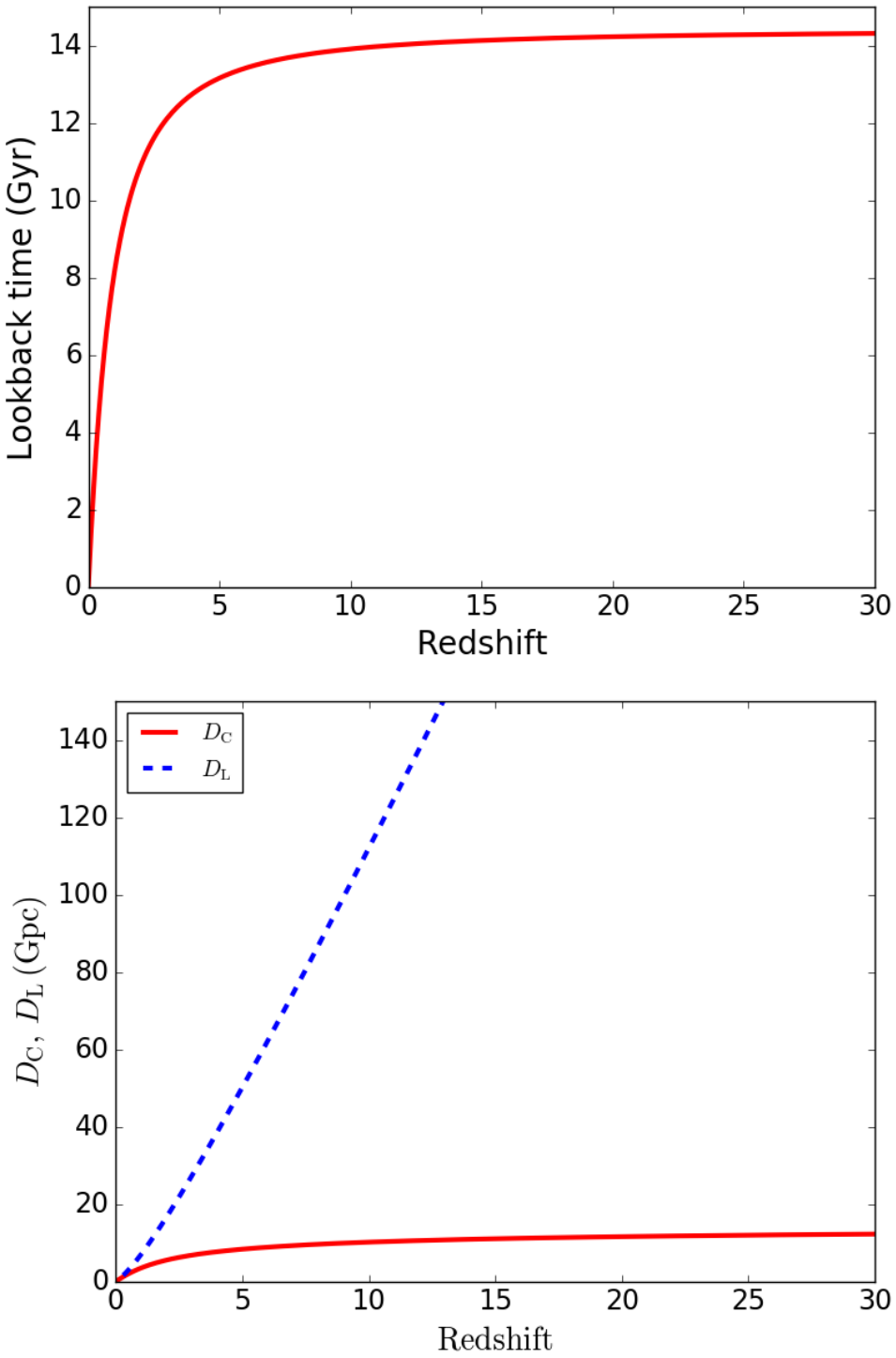


*Figure 13: Example of result from Exercise 1.4.*

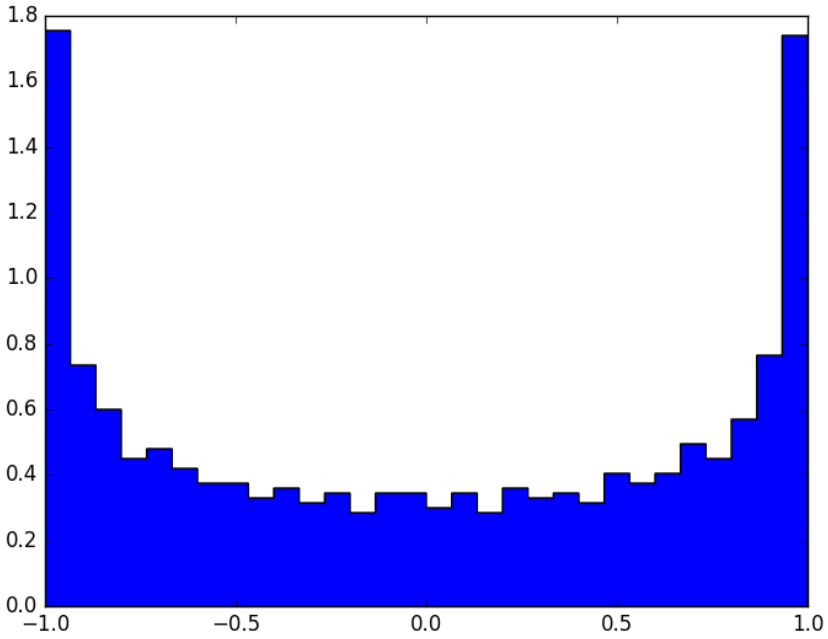


*Figure 14: The simplest example of a histogram.*

range: tuple or None, optional. The lower and upper range of the bins. Lower and upper outliers are ignored. If not provided, range is (x.min(), x.max()). Range has no effect if bins is a sequence.

density: bool, optional. If True, the first element of the return tuple will be the counts normalized to form a probability density, i.e., the area (or integral) under the histogram will sum to 1. This is achieved by dividing the count by the number of observations times the bin width and not dividing by the total number of observations. In oldish matplotlib versions, instead of density you have normed.

histtype: {'bar', 'barstacked', 'step', 'stepfilled'}, optional. The type of histogram to draw. (i) 'bar' is a traditional bar-type histogram. If multiple data are given the bars are arranged side by side. (ii) 'barstacked' is a bar-type histogram where multiple data are stacked on top of each other. (iii) 'step' generates a lineplot that is by default unfilled. (iv) 'stepfilled' generates a lineplot that is by default filled. Usually, astrophysicists choose step or stepfilled. Default is 'bar'.

align: 'left', 'mid', 'right', optional. Controls how the histogram is plotted. (i) 'left': bars are centered on the left bin edges. (ii) 'mid': bars are centered between the bin edges. (iii) 'right': bars are centered on the right bin edges. Default is 'mid', which is also what astrophysicists should use.

log: bool, optional. If True, the histogram axis will be set to a log scale. If log is True and x is a 1D array, empty bins will be filtered out and only the non-empty (n, bins, patches) will be returned. Default is False

color: color, optional. Default (None) uses the standard line color sequence.

*IMPORTANT NOTE: The histogram can be created plotting data in lin-lin (i.e. both axes linear), lin-log (i.e. x-axis linear, y-axis logarithmic), or log-log scale (both axes logarithmic). Note that the command log sets only the y-axis to logarithm, hence it produces a lin-log histogram. Log-log histograms are very useful, especially if you want to see "by eye" whether the distribution you are plotting scales as a power law.*

*A common mistake when producing a log-log histogram is to forget that the bins on the x-axis must be also set in logarithmic spacing. This means that it is not sufficient to impose plot.xscale('log'). This will produce the horrible example on the top panel of Figure 15 with bin size varying on the x-axis. For a correct log-log plot the bins should be defined not as $\Delta x$ but as $\Delta \log_{10}(x)$. A simple way to produce logarithmic bins is through the function np.logspace(a,b,num=25), which divided the interval between b and a (given as logarithmic quantities) into a number num (=25, in the example) of logarithmically spaced bins. The default is 10-based logarithm, which is perfect for scientific plots. The output of np.logspace is an array of logaritmically spaced bins. The example below produces the bottom panel of Figure 15.*

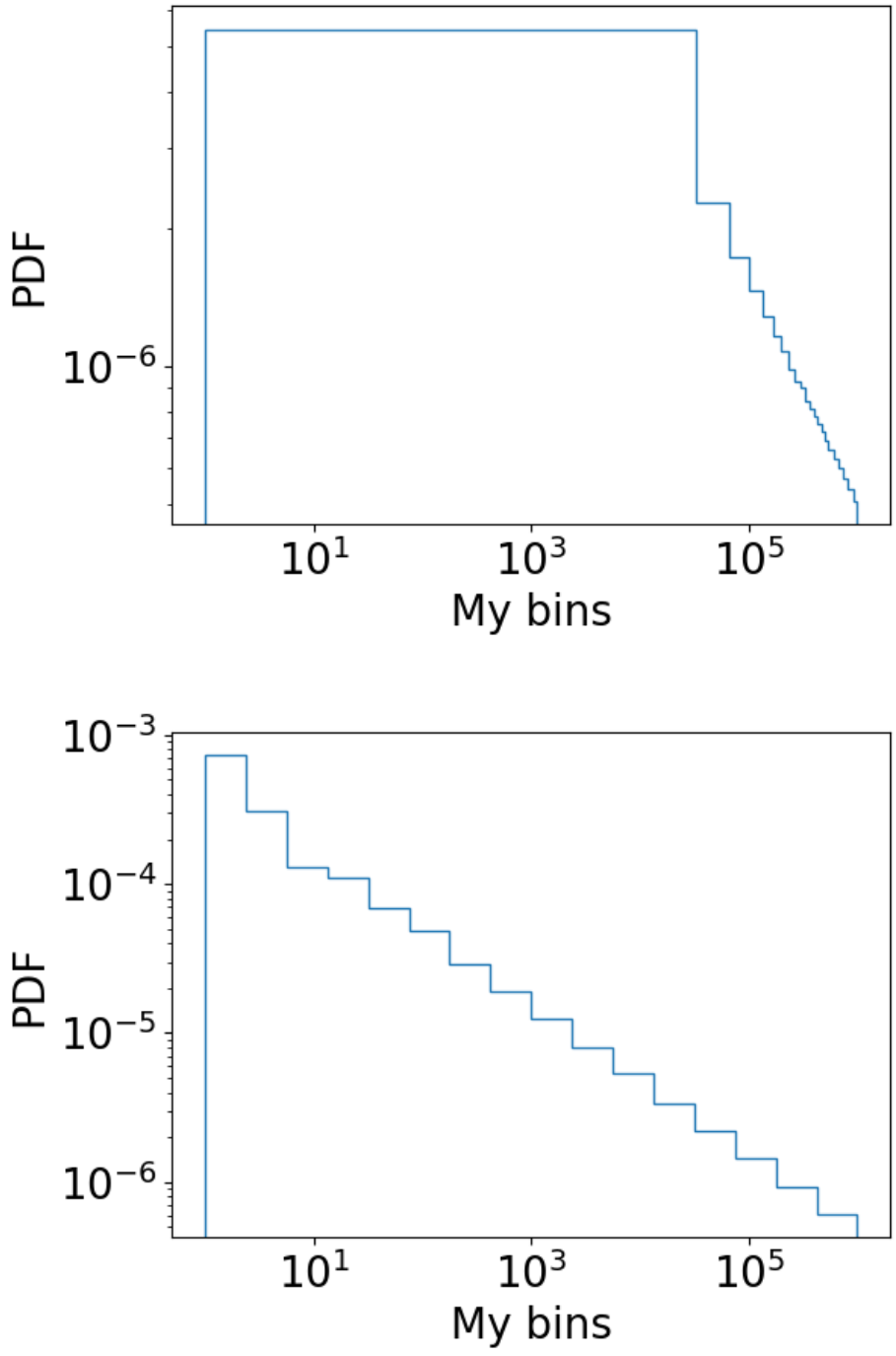


*Figure 15: Top panel: a wrong log-log histogram. Bottom panel: a correct log-log histogram, done with log_hist.py.*

```
#examples/python/log_hist.py generates proper log-log histograms
import numpy as np
import matplotlib.pyplot as plt

plt.rcParams.update({'font.size': 20})

x=np.zeros(1000,float)
y=np.zeros(1000,float)

x[0]=1.
y[0]=1.
for i in range(1,len(x)):
    x[i]=x[i-1]+1.
    y[i]=x[i]**2.

a=np.log10(min(y))
b=np.log10(max(y))

mybins=np.logspace(a,b,num=17)

plt.hist(y, bins=mybins, density='True',histtype='step',log=True)

plt.xscale("log")
plt.xlabel("My bins")
plt.ylabel("PDF")

plt.tight_layout()
plt.show()
```

## 3.9 *Two-dimensional histograms*

There is a very useful equivalent function to hist for two-dimensional histograms, which is **hist2d**. The following example uses hist2d to plot the eccentricity versus orbital period of Galactic binary neutron stars (i.e. binary neutron stars that populate the Milky Way) into one of our models.

```
#examples/python/BNS_plot.py
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as colors


plt.rcParams.update({'font.size': 15})


m1,m2,a,e=np.genfromtxt("binary_neutron_stars.txt", usecols=(2,3,5,6),unpack=True)

a*=1.49e13 #semimajor axis from AU to cm
m1*=1.989e33 #primary mass from Msun to g
m2*=1.989e33 #secondary mass from Msun to g
G=6.667e-8 #gravity constant in cgs
P=np.zeros(len(a)) #define numpy array for periods
P=np.log10(2.*np.pi*(a**3/(G*(m1+m2)))**0.5/3.1536e7) #calculate log period years
print(P)


plt.hist2d(P,e,bins=25,norm=colors.LogNorm())
cbar = plt.colorbar()
cbar.set_label('Number of BNSs per cell')

plt.xlim(-4.5,6.5)
plt.ylim(0.0,1.0)
plt.xlabel("$\log_{10}\mathrm{P/yr}$")
plt.ylabel("Eccentricity")

plt.tight_layout()
plt.show()
```

The second important function in this example is plt.colorbar, which is needed to generate a color bar for the contour plot. The result of this example is Figure 16.

## 3.10 *Contour plots*

In your academic life, you will surely need to draw a contour plot sooner or later. For the first example, we start again from the binary neutron stars in the Milky Way:

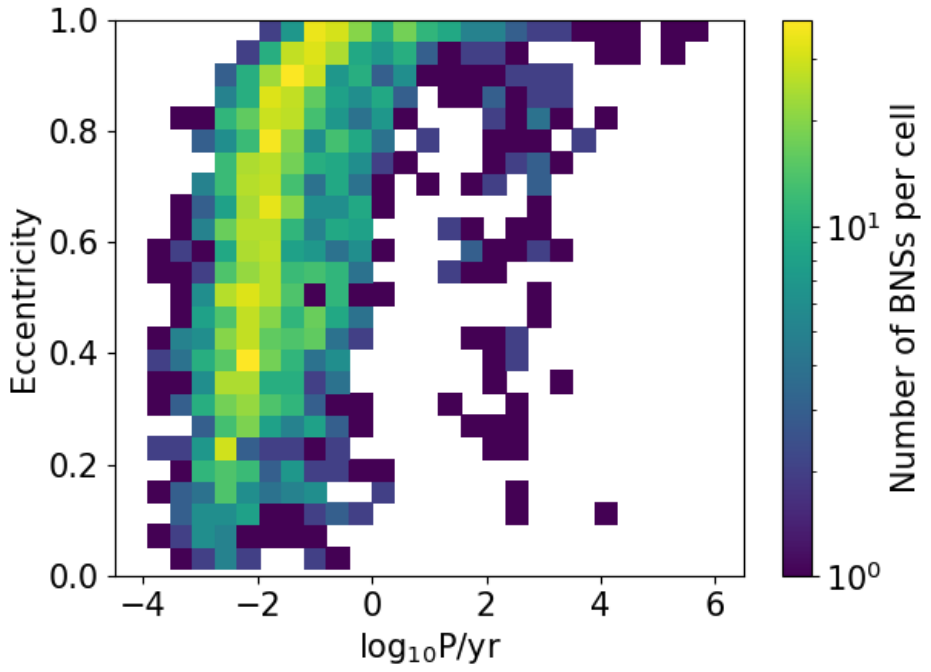


*Figure 16: Modelled distribution of the eccentricities and orbital periods of Galactic binary neutron stars. Credits: data obtained by C. Sgalletta with the* MOBSE *code [Mapelli et al., 2017].*

```
#examples/python/simple_contour.py
import numpy as np
import matplotlib.pyplot as plt


plt.rcParams.update({'font.size': 15})


m1,m2,a,e=np.genfromtxt("binary_neutron_stars.txt", usecols=(2,3,5,6),unpack=True)

a*=1.49e13 #semimajor axis from AU to cm
m1*=1.989e33 #primary mass from Msun to g
m2*=1.989e33 #secondary mass from Msun to g
G=6.667e-8 #gravity constant in cgs
P=np.zeros(len(a)) #define numpy array for periods
P=np.log10(2.*np.pi*(a**3/(G*(m1+m2)))**0.5/3.1536e7) #calculate log period years


Z,xedges,yedges=np.histogram2d(P,e,bins=25,density=False)
x=np.zeros(len(xedges)-1)
y=np.zeros(len(xedges)-1)
for i in range(len(xedges)-1):
    x[i]=(xedges[i]+xedges[i+1])/2.
    y[i]=(yedges[i]+yedges[i+1])/2.

# to have the matrix in the form needed by contourf you have to transpose
Z = np.transpose(Z)

cs=plt.contourf(x,y, Z, levels=10)

cbar = plt.colorbar(cs,orientation='vertical')
cbar.solids.set_edgecolor("face")
cbar.set_label('Number of BNSs per cell')

plt.xlim(-4.5,6.5)
plt.ylim(0.0,1.0)
plt.xlabel("$\log_{10}\mathrm{P/yr}$")
plt.ylabel("Eccentricity")

plt.tight_layout()
plt.show()
```

The above example produces Fig. 17.

The crucial function here is plt.contourf (where the final letter 'f' stands for 'filled', while plt.contour produces unfilled contour lines). A healthy usage of contourf requires at least three arguments:

z: a two dimensional array-like object of size $n \times m$. The values over which the contour is drawn.

x, y: two array-like objects, optional. x and y represent the coordinates of the values in z. x and y must both be 2 dimensional arrays with the same shape as z (e.g. created via numpy.meshgrid), or they must both be 1

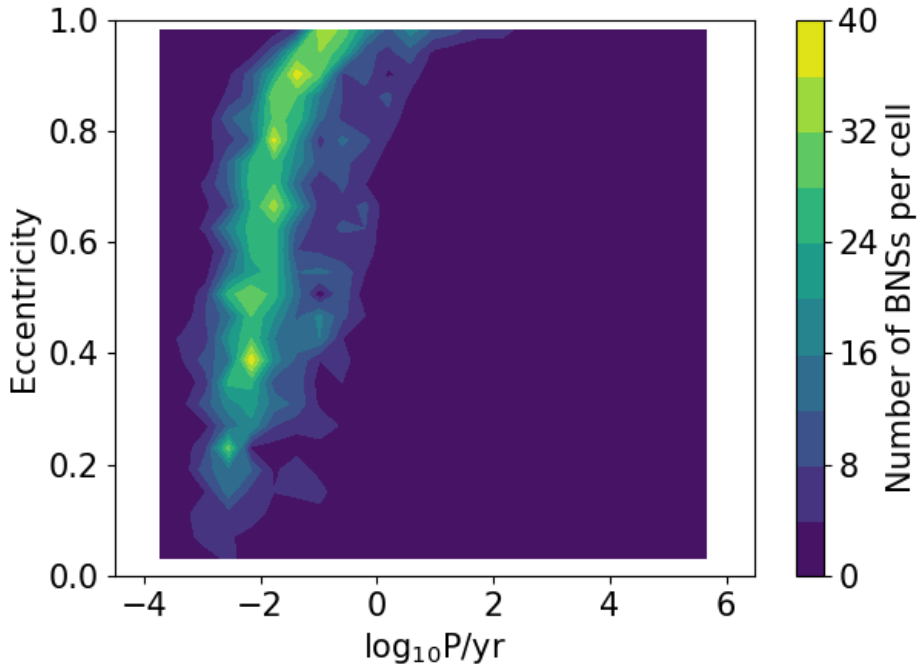


*Figure 17: Contour plot generated by the simple python script example starting from the data of the Galactic binary neutron stars.*

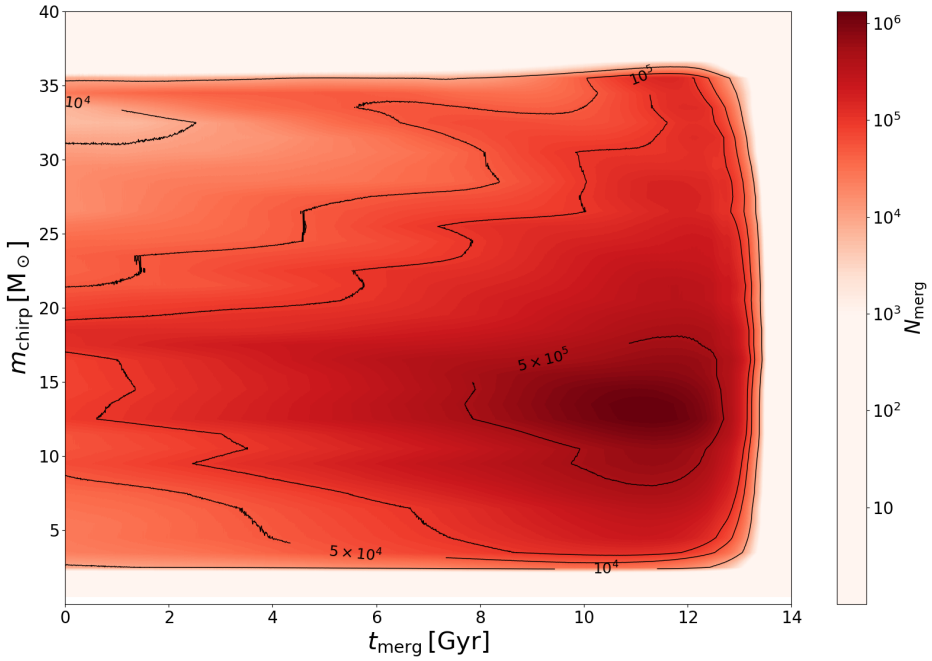


*Figure 18: Contour plot of the chirp mass ($m_{\mathrm{chirp}}$) versus time of merger ($t_{\mathrm{merg}}$, in look back time). Each cell contains the number of mergers with chirp mass between $m_{\mathrm{chirp}} - \delta m$ and $m_{\mathrm{chirp}} + \delta m$, where $\delta m = 0.5\ M_\odot$ occurring between $t_{\mathrm{merg}} - \delta t$ and $t_{\mathrm{merg}} + \delta t$, where $\delta t = 0.005$ Gyr.*

dimensional arrays such that $len(x) == m$ is the number of columns in z and $len(y) == n$ is the number of rows in z. If not given, they are assumed to be integer indices, i.e. $x = range(m)$, $y = range(n)$. In the example, x and y are 2 dimensional arrays generated with np.meshgrid, but I prefer to create my own one dimensional arrays.

cmap: string, optional. Sets the color map.

To produce the input for contourf we have used **numpy.histogram2d**, which works in a similar way as matplotlib.pyplot.hist2d: it produces a 2 dimensional histogram. The difference with hist2d is that histogram2d does not produce a plot: it returns the matrix with the values of the 2D histogram in each bin (in our example Z) and the edges of the bins in the x and in the y axis. Note that the matrix provided by numpy.histogram2d must be transposed (with numpy.transpose) to become the one accepted by contourf.

**EXERCISE:**

Produce a contour map like Fig. 18 with data files chirpmass_bin.dat (array of chirp masses, $M_\odot$), tmerg_bin.dat (array of merger times, Gyr), chirpmass_tmerg_tot.dat (complete matrix to produce the contours). The chirp mass is defined as $m_1^{3/5}\, m_2^{3/5}\, (m_1 + m_2)^{-1/5}$, where $m_1$ and $m_2$ are the masses of two compact objects in a binary system. This quantity is important for gravitational waves, because the frequency of gravitational waves changes as $\dot{f}_{\rm GW} \propto m_{\rm chirp}^{5/3}$ during inspiral. Hence, chirp mass can be directly derived from gravitational wave data, given frequency and frequency derivative with time. The merger time is the look-back time when a merger happened. These data come from a theoretical study on the cosmic merger rate of binary black holes [Mapelli et al., 2017].

**EXERCISE:**

Produce a new contour map of the mass of the secondary black hole (i.e. the lighter one) versus the mass of the primary black hole (i.e. the more massive one) considering a sample of theoretically generated binary black holes. Unlike the previous exercise, here you have to generate the matrix $z$ (containing the number of binary black holes in each cell with primary mass between $m_1 + \delta m$ and $m_1 - \delta m$ with $\delta m = 0.5\ M_\odot$ and with secondary mass between $m_2 + \delta m$ and $m_2 - \delta m$ with $\delta m = 0.5\ M_\odot$). The file you should start from is time_BHillustris1_30.dat (look at the comments in the first line to understand the meaning of the columns). Columns 7 and 8 are the masses of the two black holes. Note that the black hole in column 7 is not necessarily the most massive: you should swap the two black holes if the one in column 7 is lighter than the one in column 8. If you succeed, you should be able to recover a contour plot as the one in Figure 19.

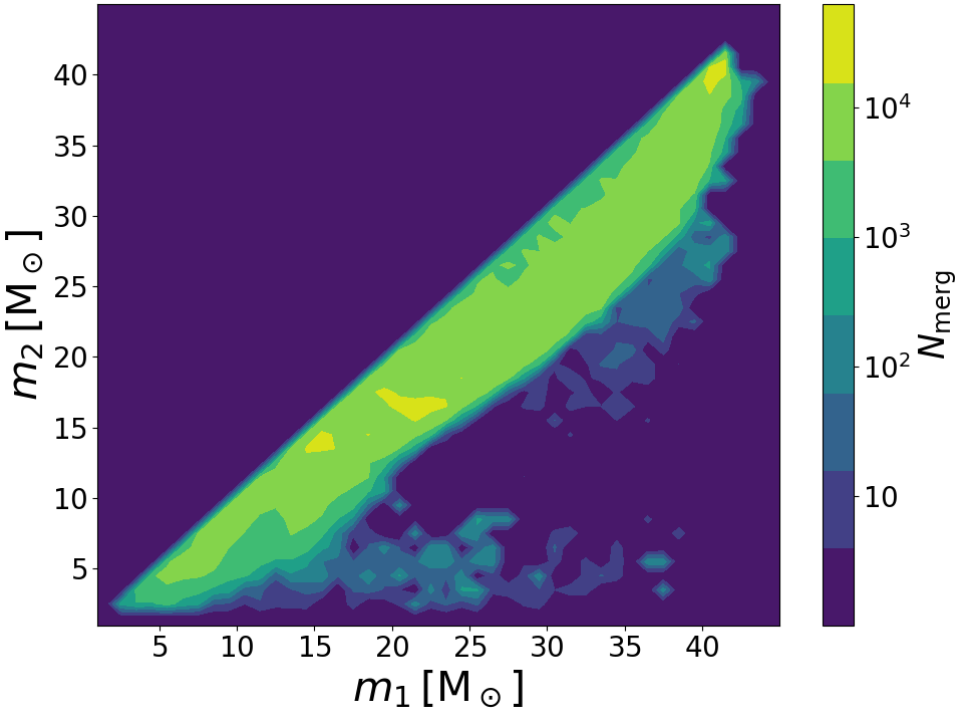


*Figure 19: Contour plot of the primary black hole mass ($m_1$) versus the secondary black hole mass ($m_2$) for a theoretically generated sample of binary black hole mergers.*

Suggestion: Note that time_BHillustris1_30.dat is quite a large file. If you read the arrays of $m_1$ and $m_2$ and then you do a couple of nested for loops to calculate the matrix $z$, your program will be very slow. To speed it up significantly you can use the following consideration.
I define the edges of the mass bins as

```
bin[0]=0
bin[1]=bin[0]+dm
bin[2]=bin[1]+dm
...
bin[n]=bin[n-1]+dm,
```

where dm = $(m_{\max} - m_{\min})$/N and N = number of bins.
Then I can assign the indexes as

```
index1 = int(mass1/dm)
index2 = int(mass2/dm)
```

Finally, I calculate the table as

```
nmerg[index2][index1] +=1
```

## 3.11 *Subplots*

The subplots function in the matplotlib.pyplot package is a very fast way to produce plots with multiple panels.

See the example below.

```
#examples/python/subplots.py

import numpy as np
import matplotlib.pyplot as plt

# Fixing random state for reproducibility
np.random.seed(19680801)

dt = 0.01
t = np.arange(0, 30, dt)
nse1 = np.random.randn(len(t))   # gaussian distributed white noise 1
nse2 = np.random.randn(len(t))   # gaussian distributed white noise 2

# Signal that evolves as a sinusoid with time
s0s = np.sin(2 * np.pi * 10. * t)

# Signal that evolves as a cosinusoid with time
s0c = np.cos(2 * np.pi * 10. * t)


# Adding white noise to the sinusoidal signal
s1 = s0s + nse1
s2 = s0s + nse2


# Adding white noise to the cosinusoidal signal
s3 = s0c + nse1
s4 = s0c + nse2


fig, axs = plt.subplots(3, 2)
#print(axs)
#print(fig)
axs[0][0].plot(t, s0s, color="black")
axs[0][0].set_xlim(0, 2)
axs[0][0].set_xlabel('time')
axs[0][0].set_ylabel('Sinusoidal signal')
axs[0][0].grid(True)


axs[1][0].plot(t, s1)
axs[1][0].plot(t, s2)
axs[1][0].legend(["s1","s2"],ncol=2)
axs[1][0].set_xlim(0, 2)
axs[1][0].set_xlabel('time')
axs[1][0].set_ylabel('Signal+Noise')
axs[1][0].grid(True)


axs[2][0].plot(t, s1-s2)
axs[2][0].set_xlim(0, 2)
axs[2][0].legend(["s1-s2"],ncol=2)
axs[2][0].set_xlabel('time')
axs[2][0].set_ylabel('Residuals')
axs[2][0].grid(True)

axs[0][1].plot(t, s0c, color="black")
axs[0][1].set_xlim(0, 2)
axs[0][1].set_xlabel('time')
axs[0][1].set_ylabel('Co-sinusoidal signal')
axs[0][1].grid(True)
```

```
axs[1][1].plot(t, s3)
axs[1][1].plot(t, s4)
axs[1][1].legend(["s3","s4"],ncol=2)
axs[1][1].set_xlim(0, 2)
axs[1][1].set_xlabel('time')
axs[1][1].set_ylabel('Signal+Noise')
axs[1][1].grid(True)



axs[2][1].plot(t, s3-s4)
axs[2][1].set_xlim(0, 2)
axs[2][1].legend(["s3-s4"],ncol=2)
axs[2][1].set_xlabel('time')
axs[2][1].set_ylabel('Residuals')
axs[2][1].grid(True)




fig.tight_layout()
plt.show()
```

The above example produces the plot shown in Figure 20.

In the first part we simply generate four fake sets of data (e.g. signals in arbitrary units versus time). Time is stored in the numpy array 't', while the two "perfect" signals (without noise) are assigned to s0s (the sinusoidal signal) and s0c (the cosinusoidal signal). To generate those we have used the trigonometric functions np.sin and np.cos for s0s and s0c, respectively.

We then generate numpy arrays s1 and s2, which are the same as the sinusoidal signal s0s plus two different sets of white noise nse1 and nse2. These are generated as random numbers drawn from a Gaussian distribution centered around zero (thanks to the function np.random.randn). We will talk about randn when we introduce random numbers. For now it is sufficient to know that this is used to add some random noise to the data.

s3 and s4 are the same, but starting from the co-sinusoidal signal s0c.

The subplots part starts with

```
fig, axs = plt.subplots(3, 2)
```

This function creates a figure (fig) and a set of subplots (axs).

The figure fig is a container for all the plot elements, see `https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.figure.Figure.html#matplotlib.figure.Figure`

axs is an array of Axes objects (see `https://matplotlib.org/3.1.1/api/axes_api.html#matplotlib.axes.Axes`). Each of the axes objects contains the elements which go into one of the panels of the figure (data, labels, axis information, text, etc).

The number of sub-plots (=panels of the figure) is decided by the two

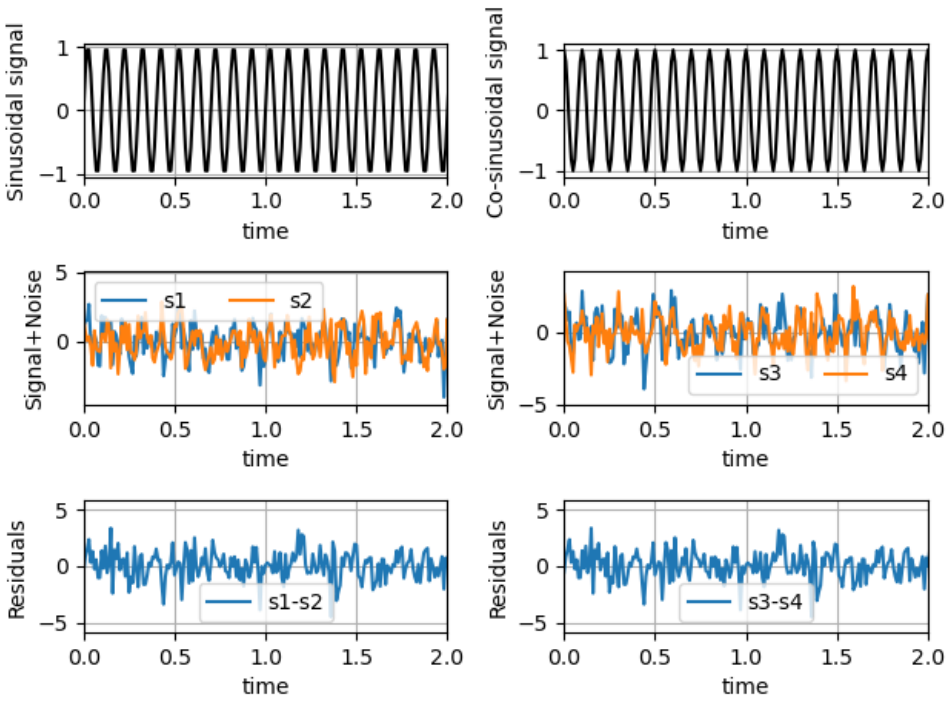


*Figure 20: Example of subplots. In the upper left-hand panel we simulate a perfect sinusoidal signal (without noise), while in the upper right-hand panel we simulate a perfect co-sinusoidal signal (without noise). In the central left-hand panel we show what we obtain if we sum the signal with some white noise. The blue and the orange lines are obtained with the same sinusoidal signal but two different random realizations of the noise. The central right-hand panel shows the same but for the co-sinusoidal signal. Lower left-hand panel: residual between the two noisy sinusoidal signals. Lower right-hand panels: the same but for the noisy co-sinusoidal signals.*

arguments of subplots. The first argument is the number of rows, the second argument the number of columns. Hence (3,2) means that we build a figure with three rows and two columns, i.e. three panels from top to bottom and two panels from left to right. axs will then be a matrix of objects with three rows and two columns: axs[0][0] refers to the top left-hand panel, axs[1][0] to the middle left-hand panel, axs[2][0] to the bottom left-hand panel, while axs[0][1] refers to the top right-hand panel, axs[1][1] to the middle right-hand panel and axs[2][1] to the bottom right-hand panel.

The lines of code following the call to subplots build the 6 panels of the sub-plot figure. For example

```
axs[0][0].plot(t, s0s, color="black")
axs[0][0].set_xlim(0, 2)
axs[0][0].set_xlabel('time')
axs[0][0].set_ylabel('Sinusoidal signal')
axs[0][0].grid(True)
```

contains all the commands needed to fill in the top left-hand panel (axs[0][0]): we plot s0s as a function of time, we decide the limits on the x axis, we define the labels of the x and y axis and finally we decide that we want to show a grid in the figures (grid function).

**EXERCISE:**

Come back to the file time_BHillustris1_30.dat you read to perform the previous EXERCISE and produce two subplots (just one row, two columns). In the first column, you report the contour plot of the mass of the secondary black hole (i.e. the lighter one) versus the mass of the primary black hole (i.e. the more massive one). In the second column, you should plot the chirp mass versus the total mass ($m_1 + m_2$). If you want to add other subplots, you are free to experiment with the quantities provided in the file time_BHillustris1_30.dat.

## 3.12 *3D plots*

One of the simplest ways to produce 3D plots in python is with the subplots feature, by importing the Axes3D package from mpl_toolkits.mplot3d. See the example below, which produces Figure 21.

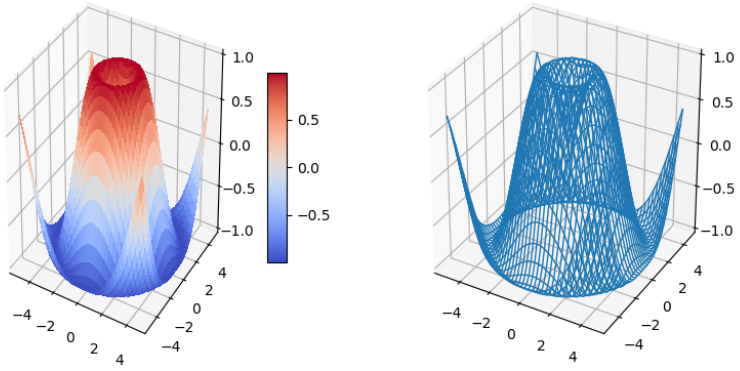


*Figure 21: Two examples of 3D plots done with matplotlib.pyplot. Produced with the example script plots3D.py.*

```
#see examples/python/plots3d.py
import matplotlib.pyplot as plt
from matplotlib import cm
import numpy as np

# This imports the 3D projection (projection='3d')
from mpl_toolkits.mplot3d import Axes3D


# set up a figure twice as wide as it is tall
fig = plt.figure(figsize=plt.figaspect(0.5))

#===============
#  First subplot
#===============
# set up the axes for the first plot
ax = fig.add_subplot(1, 2, 1, projection='3d')

# plot a 3D surface
X = np.arange(-5, 5, 0.25)
Y = np.arange(-5, 5, 0.25)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)
surf = ax.plot_surface(X, Y, Z, rstride=1, cstride=1, \
 cmap=cm.coolwarm, linewidth=0, antialiased=False)
ax.set_zlim(-1.01, 1.01)
fig.colorbar(surf, shrink=0.5, aspect=10)

#===============
# Second subplot
#===============
# set up the axes for the second plot
ax = fig.add_subplot(1, 2, 2, projection='3d')

# plot a 3D wireframe
ax.plot_wireframe(X, Y, Z, rstride=1, cstride=1, linewidth=1)
ax.set_zlim(-1.01, 1.01)

plt.show()
```

From this script we learn not only how to produce 3D plots, but also an alternative way of producing subplots (note that the subplots package presented in the previous section cannot be used with 3d projections). In this case, we use the figure module of matplotlib.pyplot and the add_subplot function to add subplots to the figure.

add_subplot here takes four arguments. The first one is the number of rows, the second one the number of columns of the entire Figure, the third one the number of the subplot (here we have 1 row × 2 columns, so two subplots in total). Finally, the fourth argument is **projection='3d'**. This comes from the Axes3D package and allows to do a 3D projection (with 3 axes).

The plot_surface function plots a 3D figure with color filling, while the plot_wireframe function plots a 3D wire frame (like a net). Note the **antialiased=False** in function plot_surface to remove transparency effects. **rstride** and **cstride** define the spacing between lines and/or colors. Find out the other features by yourselves, playing with the example script.

**EXERCISE:**

Come back to the file time_BHillustris1_30.dat and produce a 3D scatter plot.
Unlike the surface 3D plot presented in the main lecture, a 3D scatter plot needs the function **scatter** (instead of plot_surface or plot_wireframe). The function scatter reads data from three monodimensional arrays (or lists) that contain the three different quantities of the same object we want to plot on three different axes.
In this exercise you will consider column 4 of time_BHillustris1_30.dat (metallicity of the progenitor stars of the simulated black holes, in absolute values Z), column 7 and 8 (mass of the first and mass of the second black hole in $M_\odot$) and column 9 (delay time in Gyr, i.e. time elapsed from the formation of the progenitor stars to the merger of the two black holes). **IMPORTANT: please read only the first 10'000 lines of the file time_BHillustris1_30.dat, otherwise your plot becomes too crowded and heavy.**
You will produce a 3D scatter plot similar to the one in Figure 22, where the x-axis shows the total mass of the binary ($m_1 + m_2$), the y-axis shows the metallicity of the progenitor star (Z) and the z-axis shows the delay time. Finally, the Figure also shows the chirp mass of the system $(m_1\, m_2)^{3/5}\, (m_1 + m_2)^{-1/5}$ as colour gradient (see the colour map). You should be able to find out this feature of the scatter function by looking at the matplotlib manuals on the internet. Otherwise, just ask me.

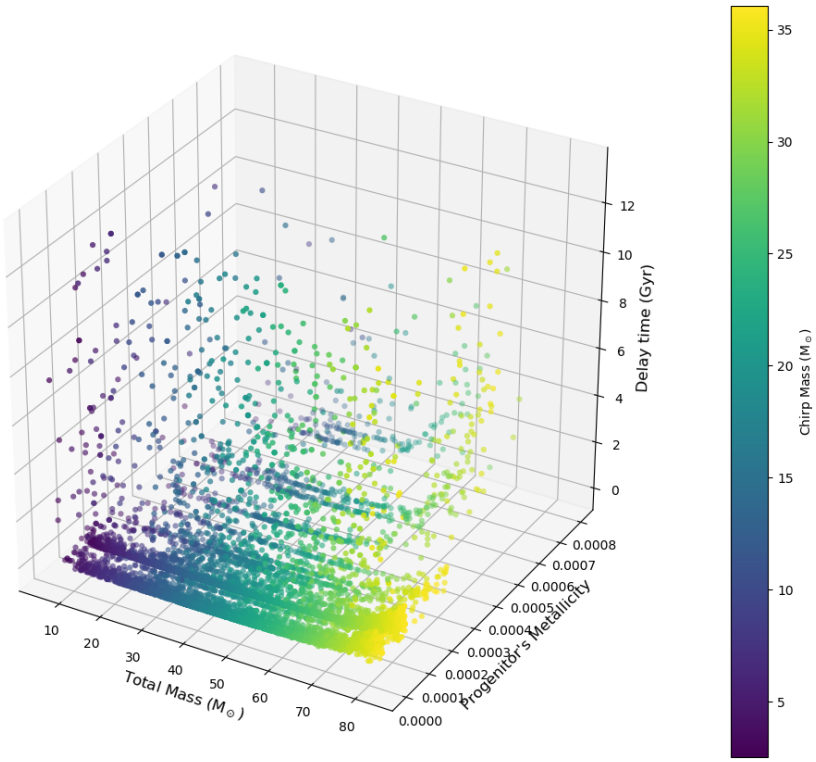


*Figure 22: Result of the above EXERCISE: 3D scatter plot using the data of time_BHillustris1_30.dat. x-axis: total mass of the two black holes; y-axis: metallicity of their progenitor stars; z-axis: delay time; color map: chirp mass of the two black holes.*

# 4 ACCURACY AND SPEED

This chapter is based on *Computational Physics* by Mark Newman `http://www-personal.umich.edu/~mejn/cp/`.

The examples corresponding to this section are in examples/rounding.

## 4.1 *Scientific notation*

Scientific notation consists in using an "e" to denote the exponent of a number, expressed in powers of 10. For example 1e19 means $10^{19}$, 1.602e20 means $1.602 \times 10^{20}$, 2e-10 means $2 \times 10^{-10}$.

Note that a variable expressed in scientific notation is always a float, even if it is intrinsically an integer (like 1e19).

Scientific notation is highly recommended, especially if you deal with very large or very small numbers.

A common badness is to use the power operator to simply express a large number. For example, both

```
x=2*10**20
```

and

```
x=2e20
```

assign to x the value of $2 \times 10^{20}$, but there is a big intrinsic difference between the two notations. Using the power operator, you ask python to compute the value and then you assign it to x. Instead, using the exponential notation, you simply assign a value to x, without computing anything. Computing a value is always more computationally expensive than assigning it; hence, especially inside a loop,

```
x=2*10**20
```

should be avoided.

## 4.2 *Maximum size of a variable*

Python variables (as well as variables in other programming languages) cannot hold numbers that are arbitrarily large. For example, the largest value of a floating-point variable is $2^{1024} \sim 1.79769 \times 10^{308}$.

If the value of a variable exceeds this limit, the variable **overflows**. When this happens, you usually (but not always) get a overflow warning.

Try the following:

```
x=1e308
y=10.*x
print(y)
```

Python does not complain (no overflow warning), but the print tells you that y is inf, that is infinity. For your computer, anything larger than the largest number you can store in a variable is infinity.

Similarly, if you pretend to use a value smaller than $2^{-1022} \sim 2.22507 \times 10^{-308}$, the calculation **underflows** and the computer just sets the number to zero.

This problem is partially solved by python for integers (and this is a special feature of python with respect to other programming languages). Python can represent integers of arbitrary size (=arbitrary number of digits), because it decides memory allocation based on the size of the integer. The limit is the memory of the computer. Plus, consider that operations on very large integer numbers require a lot of time, because of memory access time. For example, try to calculate

```
2**10000000
```

## 4.3 *Rounding errors*

Floating point numbers are represented on a computer to only a certain precision. In python, the standard level of precision is 16 digits. For example,

```
import numpy as np
np.pi #print the value of pi greek
```

You get 3.141592653589793, i.e. it stops at 16 digits.

The difference between the true value of a number and its value on the computer is called **ROUNDING ERROR**.

For example, we know that $3.3 - 1.1 = 2.2$, but python might print 2.1999999999999997.

A practical implication is that you should never use an if statement to check equality between two floats. For example

```
x=3.3-1.1
if(x==2.2):
    print(x)
```

does not print anything, because $x = 2.1999999999999997$, which is different from 2.2. If you want to do a check like this, choose a tolerance you can accept, and then do a comparison

```
x=3.3-1.1
epsilon=1e-2
if(abs(x-2.2)<epsilon):
     print(x)
```

We can regard at the rounding error as an **error of measurement in a lab experiment**. For example

```
from math import sqrt
x=sqrt(2)
```

The value of x will not be $x = \sqrt{2}$ but rather $x \pm \epsilon = \sqrt{2}$, where $\epsilon$ is the rounding error. Since x in our example is a float, the error will be of the order of $x/10^{16}$. It is usually a good assumption to consider the error to be distributed according to a Gaussian distribution with standard deviation $\sigma = C\,x$, where C is the **error constant** and $C = 10^{-16}$ in our case. When quoting the error on a calculation, we usually give the value of C, not that of $\epsilon$, because we do not know the exact value of $\epsilon$.

Suppose now that we sum two numbers $x_1$ and $x_2$, both affected by rounding errors with standard deviation $\sigma_1$ and $\sigma_2$. Then, the rounding error on their sum will be calculated starting from the **variance** $\sigma$ in experimental error theory:

$$\sigma = \sqrt{\sigma_1^2 + \sigma_2^2} = \sqrt{C^2\,x_1^2 + C^2\,x_2^2} = C\sqrt{x_1^2 + x_2^2} \tag{4}$$

Similarly, if we sum up N numbers:

$$\sigma^2 = \sum_{i=1}^{N} C^2\,x_i^2 = C^2\,N\,\bar{x^2}, \tag{5}$$

where $\bar{x^2}$ is the mean-square value of $x$. Hence, $\sigma$ increases with the number of operations but only as $\sqrt{N}$. Finally, the **fractional error** on $\sum_{i=1}^{N} x_i$ is

$$\frac{\sigma}{\sum_{i=1}^{N} x_i} = \frac{C\sqrt{N}\sqrt{\bar{x^2}}}{N\,\bar{x}} = \frac{C}{\sqrt{N}}\frac{\sqrt{\bar{x^2}}}{\bar{x}} \tag{6}$$

This means that the fractional error goes down if we add more numbers.

Now, we want to calculate a subtraction between two numbers which are very different in size, for example

```
x=int(1e15)
y=1000000000000001.23456789
print(y-x)
```

The result of this print is 1.2, because we hit the 16 digit rounding.

The situation gets even worse when we calculate the subtraction of two numbers that are nearly the same.

**EXERCISE:**

Consider the two numbers $x = 1$, $y = 1 + 10^{-14}\sqrt{2}$. We see that

$$10^{14}(y - x) = \sqrt{2}$$

Now, write a script that defines $x$ and $y$ as above and then prints $10^{14}(y - x)$. The result of the print will be 1.42108547152, while $\sqrt{2} = 1.41421356237$. Hence, the result is only accurate to the first decimal place.

**EXERCISE:**

Consider a quadratic equation $a\,x^2 + b\,x + c = 0$ that admits 2 real solutions.

a) Write a script that takes in input the three numbers $a$, $b$, $c$ and returns the solution $x$ using the standard formula

$$x = \frac{-b \pm \sqrt{b^2 - 4\,a\,c}}{2\,a} \tag{7}$$

Apply this to the solution of the equation where $a = c = 0.001$, $b = 1000$.

b) Write the solution $x$ in another way, by multiplying numerator and denominator by $-b \mp \sqrt{b^2 - 4\,a\,c}$:

$$x = \frac{2\,c}{-b \mp \sqrt{b^2 - 4\,a\,c}} \tag{8}$$

Apply this to the solution of the equation where $a = c = 0.001$, $b = 1000$. What do you see? How do you explain it?

c) Using what you learned, write a script that calculates both solutions accurately.

You can write a script like this

```
import math

a=c=0.001
b=1000.0

x1=-b+math.sqrt(b*b-4*a*c)
x1/=(2.*a)


x2=-b-math.sqrt(b*b-4*a*c)
x2/=(2.*a)

print(x1,x2)
print(a*x1*x1+b*x1+c,a*x2*x2+b*x2+c)

x1p=2*c
x1p/=(-b-math.sqrt(b*b-4*a*c))

x2p=2*c
x2p/=(-b+math.sqrt(b*b-4*a*c))


print(x1p,x2p)
print(a*x1p*x1p+b*x1p+c,a*x2p*x2p+b*x2p+c)
print((x1-x1p)/x1,(x1-x1p),(x2-x2p)/x2,(x2-x2p))
```

The result has relative error of $10^{-5}$ in both cases, but since solution x1 is small while solution x2 is big, the error on solution x2 looks larger.

If I fold the solutions back into the original quadratic equation, x1p is slightly better than x1 because I avoid the subtraction of two similar numbers. x2 is much better than x2p because I avoid the subtraction of two similar numbers. Hence, the best is to use x1p and x2.

## 4.4 *Speed*

The computer cannot be infinitely fast. Considering state-of-the-art laptops and workstations, it takes few seconds to do a million of simple operations. Billions of operations take minutes to hours. Hence, before you start coding, you might want to estimate whether your script will be executed in a reasonable time or not. The rule of thumb is that the calculation is doable if you have up to a billion of operations.

If you expect many more, either you find a smarter way to do things (see the example of building an histogram with indexes in the previous chapter) or you go for a supercomputer. Usually, there is a smarter way to do things.

In addition, not all operations are the same. Sum and subtraction are fast, while the power operator or more complex ones require more time.

The speed of a script can be improved with some simple tricks.

**EXERCISE :**

Consider for example the quantum simple harmonic oscillator with energy levels $E_n = \hbar\,\omega(n + \frac{1}{2})$, where $n = 0, 1, 2, ..., \infty$. According to Boltzmann and Gibbs, the average energy of a simple harmonic oscillator at temperature T is

$$\langle E \rangle = \frac{1}{Z} \sum_{n=0}^{\infty} E_n \exp(-\beta E_n), \tag{9}$$

where $\beta = 1/(k_B\,T)$, with $k_B$ being the Boltzmann constant and $Z = \sum_{n=0}^{\infty} \exp(-\beta E_n)$. Suppose we want to calculate the value of $\langle E \rangle$ for $k_B\,T = 100$. The worst term from the computational point of view is the sum. Let's consider first $n = 0, 1, ..., 10^6$. Let's work in units where $\hbar = \omega = 1$.

For example, the solution of this exercise can be worked out as:

```
#examples/rounding/harm_osc_0.py
import numpy as np

N=1000000
T=100.

Em=0.0
Z=0.0
beta=1./T

E=np.zeros(N,float)

for i in range(N):
    E[i]=i+0.5

for i in range(N):
    Z=Z+np.exp(-beta*E[i])

for i in range(N):
    Em=Em+E[i]*np.exp(-beta*E[i])

Em=Em/Z

print(Em)
```

The script above gives the correct result but is quite poor in performance because I used 3 loops when I can use just one and because I do twice the calculation of the exponential (which is consuming). On my laptop, it runs in $\sim$ 4 seconds.

The script can be greatly improved with few simple modifications, as follows:

```
#examples/rounding/harm_osc_1.py
import numpy as np

N=1000000
T=100.

E=np.zeros(N,float)
Em=0.0
Z=0.0
beta=1./T

for i in range(N):
    E[i]=i+0.5
    Eexp=np.exp(-beta*E[i])
    Z+=Eexp
    Em+=E[i]*Eexp

Em/=Z

print(Em)
```

On my laptop, it runs in $\sim$ 2.5 seconds.

The important highlights of the improved script are:

- Initialize all the constants at the beginning of the script. It will be easier to change them.

- Minimize the number of loops. Only one loop is sufficient here if we calculate both E and Z at the same time.

- The exponential term is the same for both Z and E. Calculate it only once per each term of the loop.

If you are a bit more familiar with python, you can make the script even faster, as follows:

```
#examples/rounding/harm_osc_2.py
import numpy as np

N=1000000
T=100.

integ=np.arange(0,N,1)
E=np.zeros(N,float)
Eexp=np.zeros(N,float)
Em=0.0
Z=0.0
beta=1./T

E[:]=integ[:]+0.5
Eexp=np.exp(-beta*E)
Z=sum(Eexp)
Em=sum(E*Eexp)

Em/=Z

print(Em)
```

In the above script, I have totally removed the loops and I have substituted them with the properties/operations of numpy arrays. On my laptop, it runs in $\sim$ 0.5 seconds: I have improved by a factor of 8 with respect to the first version and by a factor of 5 with respect to the second version.

Now try to change the value of num and plot the value of $\langle E \rangle$ for num= 1000, 1100, 1200, , 1400, $10^4$, $10^5$, $10^6$. This will probably give you a figure like Fig. 23 From the figure, it appears that the result tends asymptotically to the value of 100.00083333194436 when $N \geq 10^4$. Hence, $N = 10^4$ could be the best choice in terms of accuracy versus computational time.

**EXERCISE:**

Let's calculate the product between two $N \times N$ matrices. How long it takes to do the product if $N = 100$? And if $N = 1000$?

You can obtain a script like this, where I have added few pedagogical things:

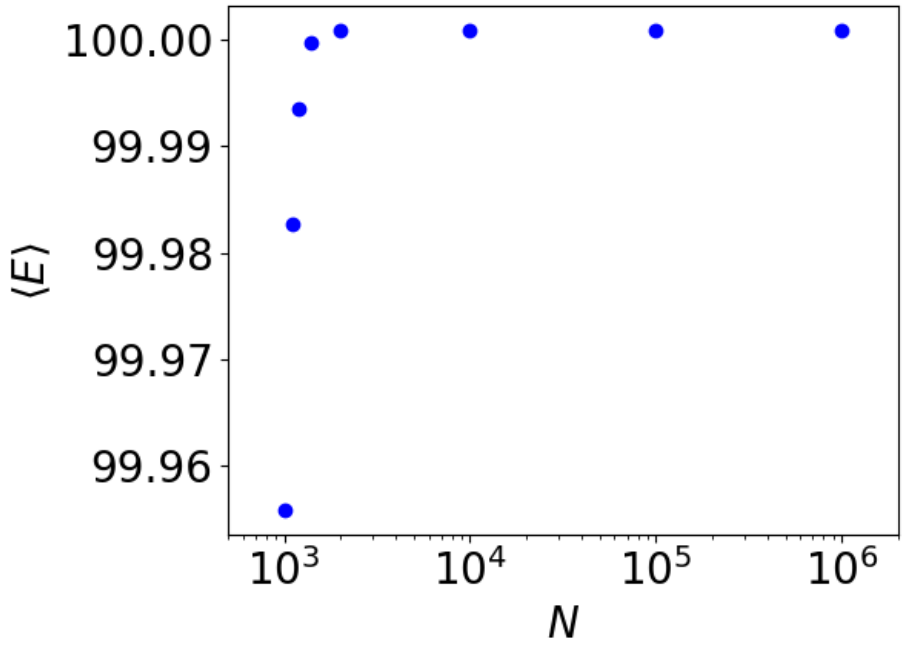


*Figure 23: Average energy of the quantum simple harmonic oscillator* ⟨E⟩ *versus the maximum number of levels* N *considered.*

```
import time
import numpy as np

N=1000
A=np.zeros([N,N],float)
B=np.zeros([N,N],float)
C=np.zeros([N,N],float)

A[:,:]=1.0
B[:,:]=2.0

start = time.time()
for i in range(N):
    for j in range(N):
        for k in range(N):
            C[i,j]+=A[i,k]*B[k,j]
end = time.time()
print(end-start)

start = time.time()
D=np.dot(A,B)
end = time.time()
print(end-start)

print(C[17,13],D[17,13])
```

The first pedagogical thing is the usage of the module time. As shown in the example, **time.time()** allows to calculate the time spent in a given chunk of the code.

The second pedagogical thing is the usage of **numpy.dot(matrix1,matrix2)**, which is a function of numpy that calculates the product of two matrices.

On my computer, for N = 100, the time requested by the user provided function is $\approx 0.6$ seconds, while the time requested by the numpy function to do the same calculation is $\approx 0.0006$ seconds. For N = 1000, the time requested by the user provided function is $\approx 600$ seconds, while the time requested by the numpy function to do the same calculation is $\approx 0.7$ seconds. **Bottom line: use the optimized functions python offers to you when you can and when you know exactly what they do**. They will be faster than your own function. If you are a very good programmer, or you are not completely sure that the python function does what you want, or you want to learn how to code, then go for your own function.

By looking at the script, we can see that in the user defined calculation we have 3 nested for loops, each of them with range N. In the innermost loop, for each element we have 2 operations (a multiplication and a sum). Hence,

the number of operations is 2 N in the innermost loop. If we consider the two outer loops, the total number of operations is $2\,N \times N \times N = 2\,N^3$. Hence, if $N = 1000$, we have to do of the order of a billion operations. It is advisable to avoid calculating the product of two matrices larger than $1000 \times 1000$ with your own function. Use numpy.dot().

### 4.5 *Small scattered tips*

- Unless you want a calculation to be with integers, please remember writing your numbers as floating point. For example

```
x=10.0
a1=x**(3/2)
a2=x**(3./2.)
print(a1,a2)
```

  If you are programming in python2 (or in many other languages), you will realize that a1=10.0 is different from a2=31.622776601683793. python2 interprets 3/2 as a division between integers[5], hence the result is 1 not 1.5.

- Use scientific exponential notation instead of powers.
- Try to minimize the usage of arrays. If there is no need to define a new array, just don't do it. Even better: sometimes you can just write with scalars. It will save RAM memory and it will make easier to code for you.
- Use **stack overflow** `https://stackoverflow.com/` or similar internet resources (start with a **google search**) to understand the errors you get from your code and to find better functions for the problem you want to tackle. You cannot learn all the python by heart.
- Try to minimize nested loops, because they really slow down your code. Sometimes, a single function of python can do the same without any loop. Please, look on the online python manuals and on slack overflow to find the best function for your case.
- Choose your units of measure to make it easier for the computer to do the calculations. If you want to calculate the mass of a galaxy cluster, it might be not-so-smart to use grams, because you will probably end up with something as big as $10^{46}$ g. Solar masses seem to be a better choice, otherwise the rounding errors might dominate your result.
- Check units of measure 100 of times in your code. Bugs hide very well in there.

[5] python3 instead interprets this as a division between floating points, because the other variable involved in the calculation, x, is a floating point.

- If some constants are used many times to a certain power or in a certain combination, define a new constant that calculates this combination to do the calculation just once. It will save time and accuracy. For example, we will see one gravitational-wave related example where G and $c$ (gravity constant and speed of light) always appear as $G^3/c^5$. Define the new constant e.g.

```
G3c5=G**3/c**5
```

  and use it through the script.

# 5 SOLUTION OF LINEAR EQUATIONS

This chapter is based on *Numerical Methods in Engineering with Python* by Jaan Kiusalaas and on *Computational Physics* by Mark Newman `http://www-personal.umich.edu/~mejn/cp/`.

We want to solve $n$ linear, algebraic equations in $n$ unknowns. A system of algebraic equations has the form

$$\begin{aligned} \mathrm{A}_{11}\,x_1 + \mathrm{A}_{12}\,x_2 + \ldots + \mathrm{A}_{1n}\,x_n &= b_1 \\ \mathrm{A}_{21}\,x_1 + \mathrm{A}_{22}\,x_2 + \ldots + \mathrm{A}_{2n}\,x_n &= b_2 \\ &\vdots \\ \mathrm{A}_{n1}\,x_1 + \mathrm{A}_{n2}\,x_2 + \ldots + \mathrm{A}_{nn}\,x_n &= b_n \end{aligned} \tag{10}$$

where the coefficients $\mathrm{A}_{ij}$ and $b_i$ are known, while the $x_i$ are the unknowns. In matrix notation, these equations are written as

$$\begin{bmatrix} \mathrm{A}_{11} & \mathrm{A}_{12} & \ldots & \mathrm{A}_{1n} \\ \mathrm{A}_{21} & \mathrm{A}_{22} & \ldots & \mathrm{A}_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ \mathrm{A}_{n1} & \mathrm{A}_{n2} & \ldots & \mathrm{A}_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \tag{11}$$

or in compact form $\mathbf{A}\,\mathbf{x} = \mathbf{b}$.

This system of linear equations has a unique solution, provided that the determinant of matrix $\mathbf{A}$ is NON-SINGULAR.

If the determinant is singular, the system may have no solutions or infinite solutions. For example

$$\begin{bmatrix} 2 & 1 \\ 4 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3 \\ 6 \end{bmatrix} \tag{12}$$

has infinite solutions because the second equation is equal to the first equation multiplied by 2: every solution of the first equation is solution of the second, too.

There are two different approaches to the solution of linear algebraic equations:

- **DIRECT METHODS:** transform the original equations into equivalent equations (i.e. equations that have the same solution) that can be solved more easily. The transformation is carried out by applying three operations (*elementary operations*) that do not change the solution but might affect the determinant of the coefficient matrix. These are i) exchanging two equations (changes the sign of the determinant); ii) multiplying an equation by a non-zero constant (multiplies the determinant by the

same constant); iii) multiplying an equation by a non-zero constant and then subtracting it from another equation (leaves the determinant unchanged). Examples of direct methods are the Gauss elimination method, the LU decomposition and the Gauss-Jordan elimination.

- **INDIRECT METHODS:** start with a guess (*Ansatz*) of the solution $\mathbf{x}$ and then repeatedly refine the solution until a certain convergence criterion is satisfied. Indirect methods are generally less efficient than direct methods but are most used, because they are simpler to implement. They are efficient if the matrix is large and sparse (has many 0) and they handle rounding errors quite well, because the rounding error of one iteration can be improved in the next one. The most serious drawback of indirect methods is that they do NOT always CONVERGE. An example of indirect methods is the Gauss-Seidel method (see below).

## 5.1 *Gauss elimination method*

Suppose we must solve a system of linear equations as the following one:

$$\begin{aligned} 2\,x_0 + x_1 + 4\,x_2 + x_3 &= -4 \\ 3\,x_0 + 4\,x_1 - x_2 - x_3 &= 3 \\ x_0 - 4\,x_1 + x_2 + 5\,x_3 &= 9 \\ 2\,x_0 - 2\,x_1 + x_2 + 3\,x_3 &= 7 \end{aligned} \tag{13}$$

In matrix form this looks like:

$$\begin{bmatrix} 2 & 1 & 4 & 1 \\ 3 & 4 & -1 & -1 \\ 1 & -4 & 1 & 5 \\ 2 & -2 & 1 & 3 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -4 \\ 3 \\ 9 \\ 7 \end{bmatrix} \tag{14}$$

and in compact notation $\mathbf{A}\,\mathbf{x} = \mathbf{b}$.

The way to solve this equation you probably learned in your math courses, is to find the inverse of the matrix $\mathbf{A}$, then multiply both sides by it to get the solution $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. This procedure is inefficient and difficult to implement numerically. Hence, we try an alternative procedure (called Gaussian elimination) as follows.

First of all, remember that we can use the following rules:

- We can **multiply any of our simultaneous equations by a constant** and it is still the same equation, in the sense that the solutions do not change. Hence, if we multiply any row of the matrix $\mathbf{A}$ by any constant and we multiply the corresponding row of the vector $\mathbf{b}$ by the same constant, then the solution does not change.

- We can take **any linear combination of two equations** to get another correct equation. This implies that if we add to or subtract from any row

of **A** a multiple of any other row, and we do the same for the vector **b**, then the solution does not change.

Our first goal in the Gaussian elimination method is to produce an **upper triangular** matrix (i.e. a matrix with all zeros below the diagonal) with all diagonal elements equal to 1. We reach this goal by using the above rules in the following way.

1. We divide the first row of **A** by the top-left element of the matrix ($a_{00} = 2$) and we obtain

$$\begin{aligned} x_0 + 0.5\,x_1 + 2\,x_2 + 0.5\,x_3 &= -2 \\ 3\,x_0 + 4\,x_1 - x_2 - x_3 &= 3 \\ x_0 - 4\,x_1 + x_2 + 5\,x_3 &= 9 \\ 2\,x_0 - 2\,x_1 + x_2 + 3\,x_3 &= 7 \end{aligned} \tag{15}$$

   In matrix form this looks like:

$$\begin{bmatrix} 1 & 0.5 & 2 & 0.5 \\ 3 & 4 & -1 & -1 \\ 1 & -4 & 1 & 5 \\ 2 & -2 & 1 & 3 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -2 \\ 3 \\ 9 \\ 7 \end{bmatrix} \tag{16}$$

2. We now subtract 3 times the first row from the second row, to zero the first element of the second row ($a_{10}$):

$$\begin{bmatrix} 1 & 0.5 & 2 & 0.5 \\ 0 & 2.5 & -7 & -2.5 \\ 1 & -4 & 1 & 5 \\ 2 & -2 & 1 & 3 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -2 \\ 9 \\ 9 \\ 7 \end{bmatrix} \tag{17}$$

3. We do a similar trick for the third and fourth row, so that the first column becomes 1,0,0,0:

$$\begin{bmatrix} 1 & 0.5 & 2 & 0.5 \\ 0 & 2.5 & -7 & -2.5 \\ 0 & -4.5 & -1 & 4.5 \\ 0 & -3 & -3 & 2 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -2 \\ 9 \\ 11 \\ 11 \end{bmatrix} \tag{18}$$

4. We want to do something similar with the other columns. Hence, we divide each element of the second row by $a_{11} = 2.5$:

$$\begin{bmatrix} 1 & 0.5 & 2 & 0.5 \\ 0 & 1 & -2.8 & -1 \\ 0 & -4.5 & -1 & 4.5 \\ 0 & -3 & -3 & 2 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -2 \\ 3.6 \\ 11 \\ 11 \end{bmatrix} \tag{19}$$

5. We subtract $-4.5$ times the second row from the third and $-3$ times the second row from the fourth:

$$\begin{bmatrix} 1 & 0.5 & 2 & 0.5 \\ 0 & 1 & -2.8 & -1 \\ 0 & 0 & -13.6 & 0 \\ 0 & 0 & -11.4 & -1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -2 \\ 3.6 \\ 27.2 \\ 21.8 \end{bmatrix} \tag{20}$$

6. We divide the third row by $a_{22} = -13.6$ to get $a_{22} = 1$:

$$\begin{bmatrix} 1 & 0.5 & 2 & 0.5 \\ 0 & 1 & -2.8 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -11.4 & -1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -2 \\ 3.6 \\ -2 \\ 21.8 \end{bmatrix} \tag{21}$$

7. subtract $-11.4$ times the third row from the fourth:

$$\begin{bmatrix} 1 & 0.5 & 2 & 0.5 \\ 0 & 1 & -2.8 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -2 \\ 3.6 \\ -2 \\ -1 \end{bmatrix} \tag{22}$$

8. Finally, we divide the fourth row by $a_{33} = -1$ to get

$$\begin{bmatrix} 1 & 0.5 & 2 & 0.5 \\ 0 & 1 & -2.8 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -2 \\ 3.6 \\ -2 \\ 1 \end{bmatrix} \tag{23}$$

Now we have an upper triangular matrix with diagonal elements equal to 1. In general form, our system of linear equations has become:

$$\begin{bmatrix} 1 & a_{01} & a_{02} & a_{03} \\ 0 & 1 & a_{12} & a_{13} \\ 0 & 0 & 1 & a_{23} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{24}$$

Which is equivalent to:

$$\begin{aligned} x_0 + a_{01}\, x_1 + a_{02}\, x_2 + a_{03}\, x_3 &= b_0 \\ x_1 + a_{12}\, x_2 + a_{13}\, x_3 &= b_1 \\ x_2 + a_{23}\, x_3 &= b_2 \\ x_3 &= b_3 \end{aligned} \tag{25}$$

This system of linear equations can be easily solved by **backsubstitution**, i.e.

by first solving the last equation $x_3 = b_3$, so that the second-last equation becomes $x_2 = b_2 - a_{23}\, x_3$, the third-last equation becomes $x_1 = b_1 - a_{12}\, x_2 - a_{13}\, x_3$, $x_0 = b_0 - a_{01}\, x_1 - a_{02}\, x_2 - a_{03}\, x_3$, which is very easy to implement numerically.

**EXERCISE:**

Produce a python script to implement the Gauss elimination method with backsubstitution and solve the system 14.
Solution of the exercise: [ 2. − 1. − 2. 1.]

Since this is the very first difficult exercise, let us study it together step by step. First thing to do for every numerical problem is to **decompose our problems into small sequential steps the computer can perform and write them down in the proper logical order**. This can be done with a **piece of paper** before you start actually writing the code, depending on your taste.

- **Step 0:** Import the packages we need, set up our input values and arrays

```
import numpy as np

a=np.array([[2.,1.,4.,1.],[3.,4.,-1.,-1.],[1.,-4.,1.,5.],[2.,-2.,1.,3.]])
b=np.array([-4.,3.,9.,7.])
x=np.zeros(4,float)
print(a,b)
n=len(b)
```

  **NOTE:** During the phase of development, do not forget to put **many prints** around your script. In the likely case the first version of the script does not work properly, these prints will help you understanding WHICH IS THE EXACT POINT WHERE your code starts not working properly. Here, I print the input $a, b$ as soon as I have set it up, to check that everything is ok.

- **Step 1:** Starting from row 0 (and then moving to the other rows) we cycle on the columns $j = 0, n-1$ to divide each element $a[i, j]$ of row $i$ by the diagonal value $a[i, i]$. In practice:

```
n=len(b)
for i in range(n):
    for j in range(n):
        a[i,j]=a[i,j]/a[i,i]
    b[i]=b[i]/a[i,i]
print(a,b)
```

  If I try performing this I immediately realize the value $a[i, i]$ is overridden by $a[i, i] = a[i, i]/a[i, i]$. Hence, I realise I need an auxiliary variable, where I can store the initial value of $a[i, i]$ before it is overridden:

```
n=len(b)
for i in range(n):
    temp=a[i,i]
    for j in range(n):
        a[i,j]=a[i,j]/temp
    b[i]=b[i]/temp
print(a,b)
```

- **Step 2:** Now I have to consider the second row and I have to do the following operation per each column $j$ of the second row:

```
temp=a[1,0]
for j in range(n):
    a[1,j]=a[1,j]-temp * a[0,j]
b[1]=b[1]-temp * b[0]
```

where note that I have defined an auxiliary temp not to override the initial value of $a[1,0]$. This expression subtracts from second row $a[1,0]$ times the first row.

- **Step 3:** Now, I can repeat the same operation to all the $k$ rows below the first one and I get (generalizing $1 \to k$ and cycling over $0 < k < n$):

```
for k in range(1,n):
    temp=a[k,0]
    for j in range(n):
        a[k,j]=a[k,j]-temp * a[0,j]
    b[k]=b[k]-temp * b[0]
```

- **Step 4:** Finally, I want to repeat this procedure for all the rows, not just for the first one, so, connecting to the first part of my script:

```
n=len(b)
for i in range(n):
    temp=a[i,i]
    for j in range(n):
        a[i,j]=a[i,j]/temp
    b[i]=b[i]/temp
    for k in range(i+1,n):
        temp=a[k,i]
        for j in range(n):
            a[k,j]=a[k,j]-temp * a[i,j]
        b[k]=b[k]-temp * b[i]
print(a,b)
```

where I have changed $0 \rightarrow i$ to loop over all $i$ rows with $0 \leq i < n$ and I have changed "for k in range(1,n)" to "for k in range(i+1,n)", to consider all the rows before the current one (which is the ith row).

The print should now produce the final elements of $a$ and $b$ and $a$ should be upper triangular:

```
[[ 1.   0.5  2.   0.5]
 [ 0.   1.  -2.8 -1. ]
 [-0.  -0.   1.  -0. ]
 [-0.  -0.  -0.   1. ]]

[-2.   3.6 -2.   1. ]
```

- **Step 5:** The last step is the back substitution. The more naive way is to write down all the operations:

```
x[n-1]=b[n-1]
x[n-2]=b[n-2]-x[n-1]*a[n-2][n-1]
  ....
x[0]=b[0]-x[1]*a[0][1]-x[2]*a[0][2]-...-x[n-1]*a[n-2][n-1]
```

The key point is to realize that this can be done more effectively (in terms of script readability and generalization) with two nested for loops, the outer one running over the $i$ going from $n-1$ to $-1$ ($-1$ not included) at steps of $-1$, the inner one going from $n-1$ to $i$ ($i$ not included) at steps of $-1$:

```
for i in range(n-1,-1,-1):
    x[i]=b[i]
    for j in range(n-1,i,-1):
        x[i]=x[i]-x[j]*a[i][j]
print(x)
```

Putting together our script, we can now write it in a complete form:

```
import numpy as np

a=np.array([[2.,1.,4.,1.],[3.,4.,-1.,-1.],[1.,-4.,1.,5.],[2.,-2.,1.,3.]])
b=np.array([-4.,3.,9.,7.])
x=np.zeros(4,float)
print(a,b)
n=len(b)

for i in range(n):
    temp=a[i,i]
    for j in range(n):
        a[i,j]=a[i,j]/temp
    b[i]=b[i]/temp
    for k in range(i+1,n):
        temp=a[k,i]
        for j in range(n):
            a[k,j]=a[k,j]-temp * a[i,j]
        b[k]=b[k]-temp * b[i]
print(a,b)

for i in range(n-1,-1,-1):
    x[i]=b[i]
    for j in range(n-1,i,-1):
        x[i]=x[i]-x[j]*a[i][j]
print(x)
```

The script as above is correct but there might be a couple of more "semi-aesthetic" changes I might want to do.

1. I can make the notation a little bit more synthetic using the feature of python (and c/c++ and many other languages) that $a = a/b$ and $a/ = b$ are exactly the same thing. This is not necessary, but I like it because it makes the equation more compact.

2. I can use a more compact writing for the simplest inner loops of my script, which is also a little be faster, i.e. I can write (for example)

   ```
   a[i,:]/=temp
   ```

   in place of

```
for j in range(n):
    a[i,j]/=temp
```

The result of these semi-aesthetic changes is the following script:

```
#gauss_eli.py
import numpy as np

a=np.array([[2.,1.,4.,1.],[3.,4.,-1.,-1.],[1.,-4.,1.,5.],[2.,-2.,1.,3.]])
b=np.array([-4.,3.,9.,7.])
x=np.zeros(4,float)
#print(a)
n=len(b)

for i in range(n):
    temp=a[i,i]
    #for j in range(n):
    #    a[i,j]/=temp
    a[i,:]/=temp
    b[i]/=temp
    #print(a)
    for k in range(i+1,n):
        temp=a[k,i]
        #for j in range(n):
        #    a[k,j]=a[k,j]-temp*a[i,j]
        a[k,:]=a[k,:]-temp*a[i,:]
        b[k]-=temp*b[i]

for i in range(n-1,-1,-1):
    x[i]=b[i]
    for j in range(n-1,i,-1):
        x[i]-=x[j]*a[i][j]

print(x)
```

## 5.2 *Pivoting*

Suppose we want to evaluate a matrix with **one or more zeroes on the diagonal**, for example:

$$\begin{bmatrix} 0 & 1 & 4 & 1 \\ 3 & 4 & -1 & -1 \\ 1 & -4 & 1 & 5 \\ 2 & -2 & 1 & 3 \end{bmatrix} \begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix} = \begin{bmatrix} -4 \\ 3 \\ 9 \\ 7 \end{bmatrix} \tag{26}$$

The Gauss elimination method clearly does not work with this system, because it would require to divide by zero. However, we can change the order of the equations in the system without changing the result of the equations. Thus we

can simply exchange the first row of **A** and the first element of **b** as follows:

$$\begin{bmatrix} 3 & 4 & -1 & -1 \\ 0 & 1 & 4 & 1 \\ 1 & -4 & 1 & 5 \\ 2 & -2 & 1 & 3 \end{bmatrix} \begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 3 \\ -4 \\ 9 \\ 7 \end{bmatrix} \tag{27}$$

This trick of changing the order of equations to avoid zeroes on the diagonal is called **pivoting**. Clearly pivoting does not work if the matrix is too **sparse**, i.e. if I have more zeroes on the diagonal than I can accommodate with line exchanges.

## 5.3 *LU decomposition*

The LU decomposition method is a slight modification of the Gaussian elimination method with pivoting. Suppose we want to solve many sets of equations in the form $\mathbf{Ax} = \mathbf{b}$, in which **A** is the same but **b** changes. In this case, it would be efficient to keep track of the transformations we make to **A**, in order to perform them on the different vectors **b** without repeating every time the Gauss elimination procedure. The LU decomposition is just this: a Gauss elimination procedure in which we keep track of the transformations we make to **A**.

Suppose we have a general matrix **A** (we write this as $4 \times 4$ matrix, but the procedure can be generalized to $N \times N$):

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \tag{28}$$

We now perform Gaussian elimination to transform this matrix in upper triangular, but we do it using general matrix notation:

$$\frac{1}{a_{00}} \begin{bmatrix} 1 & 0 & 0 & 0 \\ -a_{10} & a_{00} & 0 & 0 \\ -a_{20} & 0 & a_{00} & 0 \\ -a_{30} & 0 & 0 & a_{00} \end{bmatrix} \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 1 & b_{01} & b_{02} & b_{03} \\ 0 & b_{11} & b_{12} & b_{13} \\ 0 & b_{21} & b_{22} & b_{23} \\ 0 & b_{31} & b_{32} & b_{33} \end{bmatrix} \tag{29}$$

You can check with calculations that this is the same as the first step of Gauss elimination, i.e. the step in which we divide the first row of the matrix by $a_{00}$ and then we subtract the first row times $a_{10}$, $a_{20}$ and $a_{30}$ from the second, third and fourth rows, respectively.

We now call $\mathbf{L_0}$ the matrix we multiply by on the left (this matrix is lower

triangular):

$$\mathbf{L_0} = \frac{1}{a_{00}} \begin{bmatrix} 1 & 0 & 0 & 0 \\ -a_{10} & a_{00} & 0 & 0 \\ -a_{20} & 0 & a_{00} & 0 \\ -a_{30} & 0 & 0 & a_{00} \end{bmatrix} \tag{30}$$

It can be shown that the second step of the Gaussian elimination process can be written as

$$\frac{1}{b_{11}} \begin{bmatrix} b_{11} & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -b_{21} & b_{11} & 0 \\ 0 & -b_{31} & 0 & b_{11} \end{bmatrix} \begin{bmatrix} 1 & b_{01} & b_{02} & b_{03} \\ 0 & b_{11} & b_{12} & b_{13} \\ 0 & b_{21} & b_{22} & b_{23} \\ 0 & b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} 1 & c_{01} & c_{02} & c_{03} \\ 0 & 1 & c_{12} & c_{13} \\ 0 & 0 & c_{22} & c_{23} \\ 0 & 0 & c_{32} & c_{33} \end{bmatrix} \tag{31}$$

We now call $\mathbf{L_1}$ the matrix we multiply by on the left (this matrix is also lower triangular):

$$\mathbf{L_1} = \frac{1}{b_{11}} \begin{bmatrix} b_{11} & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -b_{21} & b_{11} & 0 \\ 0 & -b_{31} & 0 & b_{11} \end{bmatrix} \tag{32}$$

Finally, the two remaining steps of the Gauss elimination method can be done in the same way, defining the two new matrices $\mathrm{L}_2$ and $\mathrm{L}_3$ as

$$\mathbf{L_2} = \frac{1}{c_{22}} \begin{bmatrix} c_{22} & 0 & 0 & 0 \\ 0 & c_{22} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -c_{32} & c_{22} \end{bmatrix}, \qquad \mathbf{L_3} = \frac{1}{d_{33}} \begin{bmatrix} d_{33} & 0 & 0 & 0 \\ 0 & d_{33} & 0 & 0 \\ 0 & 0 & d_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{33}$$

Putting together these different steps, we find that Gauss elimination is equivalent to multiplying $\mathbf{A}$ in succession by $\mathbf{L_0}$, $\mathbf{L_1}$, $\mathbf{L_2}$ and $\mathbf{L_3}$. Thus, the Gaussian elimination method can be written as

$$\mathbf{L_3}\,\mathbf{L_2}\,\mathbf{L_1}\,\mathbf{L_0}\,\mathbf{A}\,\mathbf{x} = \mathbf{L_3}\,\mathbf{L_2}\,\mathbf{L_1}\,\mathbf{L_0}\;\mathbf{b} \tag{34}$$

From this procedure, we can evaluate $\mathbf{x}$ by backsubstitution.

From a technical point of view, the LU decomposition is implemented as follows. We define the matrix $\mathbf{L}$ as

$$\mathbf{L} = \mathbf{L_0}^{-1}\,\mathbf{L_1}^{-1}\,\mathbf{L_2}^{-1}\,\mathbf{L_3}^{-1} \tag{35}$$

where $\mathbf{L_0}^{-1}$, $\mathbf{L_1}^{-1}$, $\mathbf{L_2}^{-1}$ and $\mathbf{L_3}^{-1}$ are the inverses of $\mathbf{L_0}$, $\mathbf{L_1}$, $\mathbf{L_2}$ and $\mathbf{L_3}$.

We define the matrix $\mathbf{U}$ as

$$\mathbf{U} = \mathbf{L_3}\,\mathbf{L_2}\,\mathbf{L_1}\,\mathbf{L_0}\,\mathbf{A} \tag{36}$$

Hence the system $\mathbf{Ax} = \mathbf{b}$ can be written as

$$\mathbf{LUx} = \mathbf{b} \tag{37}$$

The nice thing here is that **U** us upper triangular and **L** is lower triangular. In fact, we can derive

$$\mathbf{L_0}^{-1} = \begin{bmatrix} a_{00} & 0 & 0 & 0 \\ a_{10} & 1 & 0 & 0 \\ a_{20} & 0 & 1 & 0 \\ a_{30} & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{L_1}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & b_{11} & 0 & 0 \\ 0 & b_{21} & 1 & 0 \\ 0 & 0 & b_{31} & 1 \end{bmatrix} \tag{38}$$

$$\mathbf{L_2}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & c_{22} & 0 \\ 0 & 0 & c_{32} & 1 \end{bmatrix} \quad \mathbf{L_3}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & d_{33} \end{bmatrix} \tag{39}$$

Multiplying them together, we find

$$\mathbf{L} = \begin{bmatrix} a_{00} & 0 & 0 & 0 \\ a_{10} & b_{11} & 0 & 0 \\ a_{20} & b_{21} & c_{22} & 0 \\ a_{30} & b_{31} & c_{32} & d_{33} \end{bmatrix} \tag{40}$$

This is something we can easily calculate from the Gaussian elimination formalism we just derived.

On the other hand, in python there is no need for writing this algorithm, because the **LU decomposition with backsubstitution** is exactly what is done by the function **solve** of the **numpy.linalg** package, that can be called in the following way:

```
from numpy.linalg import solve
x=solve(A,b)
```

*In general, mathematical functions in python are much better optimized than you can ever achieve by writing your own function. So, if there is a pre-defined function doing exactly what you need, use it without re-inventing the wheel. On the other hand, please avoid using python functions without knowing what they do exactly, i.e. which algorithms they implement and what are the validity limitations of these algorithms.*

## 5.4 *Gauss-Seidel method*

Gaussian elimination and LU decomposition are both direct methods. Now, let's see an example of indirect method, the Gauss-Seidel method. This method is particularly used for **sparse matrices**, when pivoting is not sufficient to avoid zeroes on the diagonal.

The equation $\mathbf{A}\,\mathbf{x} = \mathbf{b}$ can be written in scalar notation as

$$\sum_{j=1}^{n} \mathrm{A}_{ij}\, x_j = b_i \quad i = 1, 2, ..., n \tag{41}$$

Extracting the diagonal term $x_i$ the above equation becomes

$$\mathrm{A}_{ii}\, x_i \; + \sum_{j=1, j\neq i}^{n} \mathrm{A}_{ij}\, x_j = b_i \quad i = 1, 2, ..., n \tag{42}$$

Solving for $x_i$ we get

$$x_i = \frac{1}{\mathrm{A}_{ii}} \left( b_i - \sum_{j=1, j\neq i}^{n} \mathrm{A}_{ij}\, x_j \right) \quad i = 1, 2, ..., n \tag{43}$$

This suggests the following iterative scheme

$$x_i \leftarrow \frac{1}{\mathrm{A}_{ii}} \left( b_i - \sum_{j=1, j\neq i}^{n} \mathrm{A}_{ij}\, x_j \right) \quad i = 1, 2, ..., n \tag{44}$$

We start by choosing a value for the array **x**. We plug this value of **x** into the right-hand terms of equations 44 and we re-calculate the left-hand terms. Then we repeat this procedure a number of times. We stop when the value of **x** after a new iteration is 'sufficiently' similar to the value of **x** before the iteration.

## 5.5 *Pros and cons of the algorithms we discussed*

**LU decomposition and Gaussian elimination, PROS:**

- easy to implement;
- fast;
- exact solution.

**LU decomposition and Gaussian elimination, CONS:**

- not always possible to use (when zeroes on the diagonal even after pivoting).

**Gauss-Seidel, PROS:**

- even easier to implement than Gaussian elimination;
- fast;

- efficient in handling rounding errors.

**Gauss-Seidel, CONS:**

- approximate solution;
- might fail to converge.

**EXERCISE:**

Write a script to implement the Gauss-Seidel method and use it to solve the following equations.

$$\begin{bmatrix} 4 & -1 & 1 \\ -1 & 4 & -2 \\ 1 & -2 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 12 \\ -1 \\ 5 \end{bmatrix} \tag{45}$$

Solution of the exercise: [3.0, 1.0, 1.0].

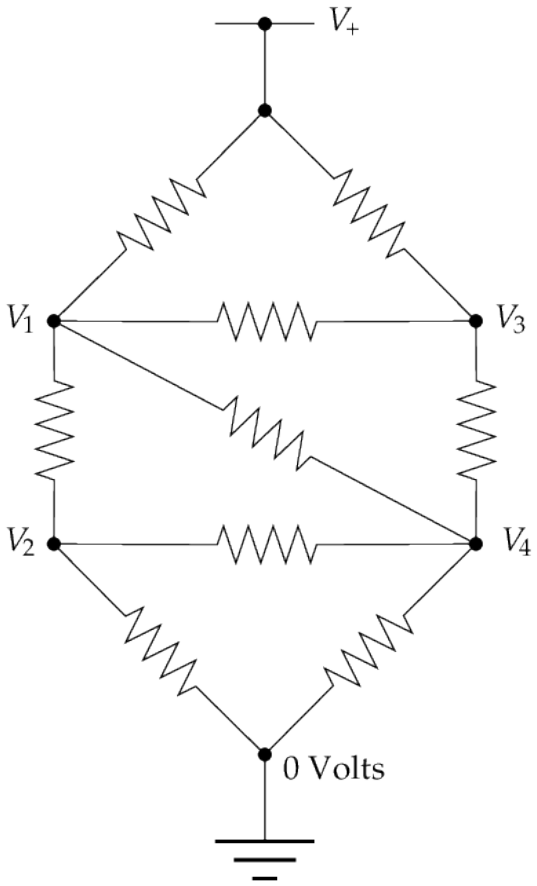


*Figure 24: Circuit of the following exercise (image credits: Mark Newman).*

**EXERCISE:**

Use the Gauss-Seidel method and the Gauss elimination method with backsubstitution (or the LU method with backsubstitution) to solve the circuit of resistors shown in Figure 24. All the resistors have the same resistance R. The power rail at the top is at voltage $V_+ = 5$ V. What are the other voltages $V_1$ and $V_4$?
Suggestion to solve the problem:
1. Write down the system of linear equations using Ohm's law and Kirchhoff current law.
For example, for the junction at Voltage $V_1$ we have

$$\frac{V_1 - V_2}{R} + \frac{V_1 - V_3}{R} + \frac{V_1 - V_4}{R} + \frac{V_1 - V_+}{R} = 0 \qquad (46)$$

2. Adapt your Gauss-Seidel and Gauss elimination scripts to solve the equations.

Solution of the exercise: [3., 1.6666667, 3.333333, 2].

# 6 SOLUTION OF NON-LINEAR EQUATIONS

This chapter is based on *Computational Physics* by Mark Newman `http://www-personal.umich.edu/~mejn/cp/`.

Many equations in physics and astrophysics are non-linear. Non-linear equations are generally harder to solve than linear equations, especially if we have systems of non-linear equations. In the following, we discuss the most known methods to solve non-linear equations: relaxation method, bisection method and Newton-Raphson method.

A non-linear equation for a single variable $x$ can be always written in the form $f(x) = 0$. Thus, finding a solution for such an equation is equivalent to **finding the zeros, or roots, of $f(x)$.** This is why several methods to solve non-linear equations (including the bisection and the Newton-Raphson methods we will discuss in this course) are called root finding methods.

## 6.1 *The relaxation method*

Suppose we have a single non-linear equation in a single variable such as

$$x = 2 - e^{-x} \tag{47}$$

The simplest approach to solve this equation (similar to what we have seen with the Gauss-Seidel method for systems of linear equations) is to **iterate the equation**. This approach is called **relaxation method**. Let's start with a guess for $x$ (e.g. $x = 1$) and plug this guess into the right-hand side of the above equation. We find a new $x = 1.63212055883$. Repeat the operation with the new guess, etc. A script doing this would look like

```
import numpy as np
tol=1e-6
x=1.0
xold=10.0
while(abs(x-xold)>tol):
    xold=x
    x=2.-np.exp(-x)
    print(x)
print("Converged to x=", x)
```

When it works, the relaxation method is a very good method, simple to program and fast. The main problems of the relaxation methods are the following.

First, it is not always easy/possible to write an equation in the simple form $x = f(x)$, where $f(x)$ is some known function. For example, we might try to solve something like $\log x + x^2 - 1 = 0$. The first thing we try to do in this case

is to arrange the equation in the form $x = f(x)$, for example

$$x = \exp(1 - x^2) \tag{48}$$

If we use the above script to solve this equation and we choose as starting guess $x = 0.5$, we can easily see that the method **DOES NOT CONVERGE**, jumping between $\sim 2.718$ and $\sim 0.00168$.

Before we give up, we can try to rearrange the equation, again, in a different way. For example

$$x = \sqrt{(1 - \log x)} \tag{49}$$

With the same initial guess ($x = 0.5$) this version of the equation converges in few iterations.

**When do we stop iterating?** A reasonable approach is to decide an **accuracy** (or **tolerance**) based on the specific problem we want to solve. Then, we impose that we exit the loop when (for example) $|x - xold| < \epsilon$, where $\epsilon$ is the imposed accuracy.

**Why the method does not always converge?** We can explain it mathematically. Assume that we have an equation in the form $x = f(x)$ that has a solution $x = x^*$. Performing a Taylor expansion around the true solution ($x^*$), the value $\tilde{x}$ after an iteration is given in terms of the previous value of $x$ by

$$\tilde{x} = f(x) = f(x^*) + (x - x^*) f'(x^*), \tag{50}$$

where we neglect the higher order terms. Since $x^*$ is the actual solution, i.e. $x^* = f(x^*)$, we have that

$$\tilde{x} - x^* = (x - x^*) f'(x^*) \tag{51}$$

This equation tells us that the distance between $x$ and the true solution $x^*$ gets multiplied by $f'(x^*)$ at every iteration. If the derivative $|f'(x^*)| < 1$, then the distance becomes smaller and smaller and the method converge. In contrast, if $|f'(x^*)| > 1$ the method does not converge.

However, if we have a function with derivative $|f'(x^*)| > 1$ and we invert the equation, the reciprocal of the derivative will be $|f'(x^*)^{-1}| < 1$. This is way the method did not converge for $x = \exp(1 - x^2)$, while it converged for $x = \sqrt{(1 - \log x)}$, which is the inverse.

Thus, by inverting the equation we could in principle solve the problems of the relaxation method. Unfortunately, not all equations can be inverted.

To summarize, the main **PROS** of the relaxation method are:

- trivial to implement;
- fast;

- might be used to solve systems of non-linear equations (but convergence is a major issue).

The main **CONS** are:

- often it does not converge;
- to force convergence, it might require re-arranging the equation or finding the inverse; this operation is not always possible.

**Basic reminder: Taylor series**

During these lectures, we will make use of Taylor series several times, both for estimating the approximation error and to build new algorithms (e.g., the algorithms for ordinary differential equations). Hence, let us refresh what is a Taylor series. It is the expansion of a real function $f(x)$ about a point $x_0$ and can be written as:

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2!} f''(x_0)(x - x_0)^2 + \\ + \frac{1}{3!} f^{(3)}(x_0)(x - x_0)^3 + \ldots + \frac{1}{n!} f^{(n)}(x_0)(x - x_0)^n + \ldots \quad (52)$$

## 6.2 *Overrelaxation*

In the relaxation method, we can try to speed up convergence by using **overrelaxation**. Let us rewrite $\tilde{x} = f(x)$ in the form $\tilde{x} = x + \Delta x$, where

$$\Delta x \equiv \tilde{x} - x = f(x) - x. \quad (53)$$

Let us now add a $(1 + \omega)$ term in the original equation

$$\tilde{x} = x + (1 + \omega)\,\Delta x, \quad (54)$$

where $\omega$ is a free parameter.

Then, the equation we want to iterate becomes

$$\tilde{x} = x + (1 + \omega)\,[f(x) - x] = (1 + \omega)\,f(x) - \omega\,x \quad (55)$$

Values of $\omega > 0$ imply that we take larger steps with respect to the iterations without $\omega$. This might help speeding up convergence but might also cause divergence.

There is no analytic way to choose the best $\omega$. Values of $\omega < 1$ are recommended.

## 6.3 *Bisection method*

The **bisection method** is the most robust method to solve a single non-linear equation. We start specifying an interval to search for a solution. If a solution exists in that interval, the bisection method will always find it.

The bisection method is a root finding algorithm. To use the bisection method, let's first re-arrange our non-linear equation so that it looks like $f(x) = 0$ (i.e. let's move all the non-zero terms of the equation to the left).

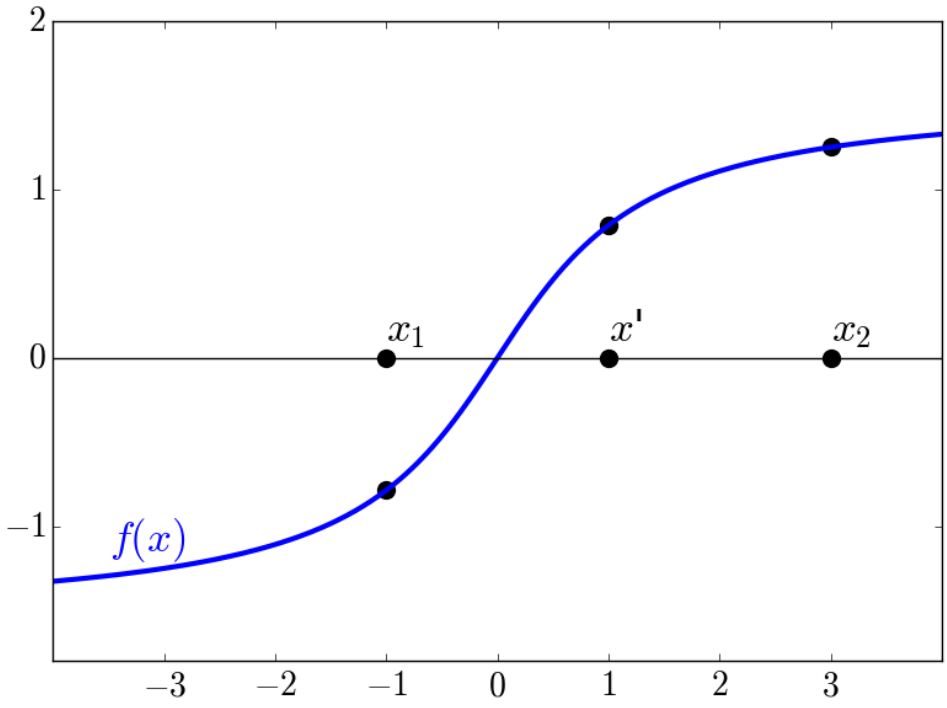


*Figure 25: Visualization of the bisection method.*

Suppose we want to find a root of $f(x)$ in the interval between $x_1$ and $x_2$, if such root exists (see figure 25). We calculate $f(x_1)$ and $f(x_2)$. If one of the two values is positive, while the other is negative, it means that there is at least one zero inside the interval $x_1$, $x_2$. Then, we calculate the mid-point of the $x_1$, $x_2$ interval, i.e. $x' = 0.5 * (x_1 + x_2)$ and we estimate $f(x')$. If $f(x_1)$ and $f(x')$ have different signs (e.g. in the figure $f(x_1) < 0$, $f(x') > 0$), then our new interval becomes $x_1, x'$. Otherwise, the interval becomes $x', x_2$. We repeat the procedure, finding the midpoint of our new interval, ans so forth.

At every step, we know the root with a factor of 2 better accuracy. If we want to find the root with an accuracy of $10^{-6}$, we simply stop when the interval is $\leq 10^{-6}$.

To summarize, the bisection procedure is the following.

1. Choose an initial interval $x_1$, $x_2$. Choose the minimum accuracy you

want $\epsilon$.

2. Check that $f(x_1)$ and $f(x_2)$ have opposite signs. If they don't, you need to choose another interval or another method.

3. Calculate the midpoint $x' = 0.5 * (x_1 + x_2)$.

4. If $f(x')$ has the same sign as $f(x_1)$, then define the new interval as $x', x_2$. Otherwise, define the new interval as $x_1, x'$.

5. If $|x_1 - x_2| > \epsilon$, repeat from step 3. Otherwise, calculate $0.5 * (x_1 + x_2)$ once more and this is the final estimate of the root.

Since the error decreases by a factor of 2 on each time-step, the accuracy improves exponentially. We can easily estimate the number N of steps we need to take to reach the desired accuracy $\epsilon$. If $\Delta$ is the initial size of the interval, then after N steps the size of the interval is $\Delta/2^N$. Thus, the number N of steps required to reach an accuracy $\epsilon$ (which is the size of the final interval), will be

$$N = \log_2(\Delta/\epsilon), \tag{56}$$

which means that this method is very fast.

Thus, the main **PROS** of the bisection methods are

- robust method: if a zero lies in the initial interval, it converges always;
- fast method.

The main **CONS** of the bisection method are the following:

- it might be that $f(x_1)$ and $f(x_2)$ have the same sign, because there is an even number of zeros in the $x_1, x_2$ interval. In this case, the bisection does not work;
- it cannot find even-order polynomial roots of functions, e.g, function $(1 - x^2)$. Functions with such roots only touch the horizontal axis at the position of the root but do not cross it;
- it does not allow to solve systems of non-linear equations.

## 6.4 *Newton-Raphson method*

Figure 27 shows a graphic representation of the Newton-Raphson method. Like the bisection method, it is a root finding algorithm: we start by re-writing the non-linear equation in the form $f(x) = 0$. We start with a guess $x$ for the solution and then we calculate the slope of the function at $x$ to refine our initial guess. The slope at $x$ is then used to extrapolate a new guess $x'$, which is usually better than the first guess, although, in some unlucky cases, it might be worse.

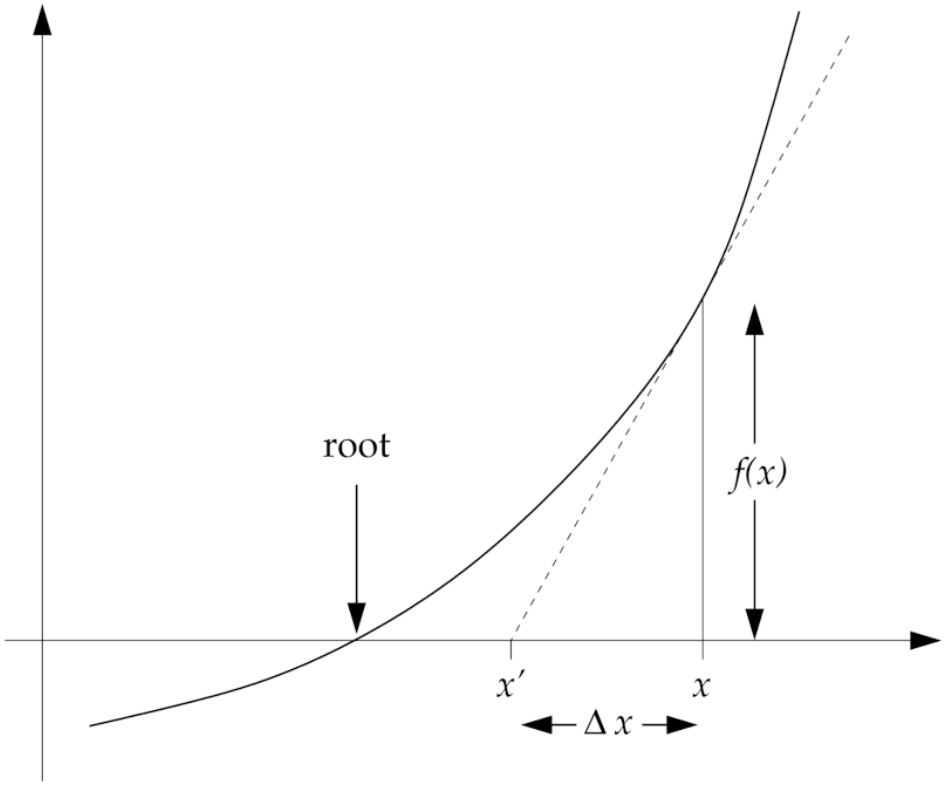


*Figure 26: Visualization of the Newton-Raphson method (credits: Mark Newman, Computational physics).*

The slope at $x$ is

$$f'(x) = \frac{f(x)}{\Delta x} \tag{57}$$

i.e. it is the **first derivative** of $f(x)$ in $x$.

Thus, the formula to find our new guess $x'$ for the root is

$$x' = x - \Delta x = x - \frac{f(x)}{f'(x)} \tag{58}$$

This algorithm requires us to know the derivative $f'(x)$ of $f(x)$.

For functions with more than one root, this method typically finds a root close to the starting value of $x$. Thus, to find different roots we can start from different guesses.

With Taylor expansion around the true value of the root, we can demonstrate that the error on our next estimate $x'$ is given by

$$\epsilon' = \left[\frac{f''(x)}{2\,f'(x)}\right]\epsilon^2, \tag{59}$$

where $\epsilon$ is the error on the previous estimate $(x)$ and $f''(x)$ is the second derivative of $f(x)$ in the previous estimate $x$. In other words, the error on the next step will scales as $\epsilon^2$ (where $\epsilon \ll 1$ is the error on the previous step). Thus, Newton-Raphson converges very quickly.

We can summarize the Newton-Raphson method as follows:

1. Choose a starting guess $x$ and an accuracy $\epsilon$ you want to achieve.
2. Calculate a new guess $x' = x - f(x)/f'(x)$.
3. If $|x' - x| > \epsilon$, repeat from point 2. If $|x' - x| \leq \epsilon$, then $x'$ is our root.

The main **PROS** of the Newton-Raphson method are:

- it is very fast;
- it is easy to implement;
- it can be used to solve systems of non-linear equations.

The main **CONS** of the Newton-Raphson method are:

- sometimes, it does not converge (less serious problem than for the relaxation algorithm);
- we need to know the derivative of $f(x)$; if we do not, we can try to use numerical derivatives (see Section 7).

We have seen three methods to solver non-linear equations: relaxation, bisection and Newton-Raphson. We have seen that all of them have pros and cons. In some cases, the bisection method will perform better than the Newton-Raphson and *vice versa*. Thus, we must be flexible and, when one algorithm fails to solve a problem, try another one.

Following Mark Newman's book: "*One of the keys to doing good computational (astro)physics is to have a range of methods at your disposal, so that if one works poorly for a particular problem you have others up your sleeve.*"

## 6.5 *Newton-Raphson method for systems of non-linear equations*

The Newton-Raphson method can be used to solve systems of non-linear equations. The first thing to do is to write these equations in the form

$$\begin{aligned} f_1(x_1, x_2, ..., x_N) &= 0, \\ f_2(x_1, x_2, ..., x_N) &= 0, \\ &\vdots \\ f_N(x_1, x_2, ..., x_N) &= 0. \end{aligned} \tag{60}$$

Suppose these equations have a root $x_1^*, x_2^*, ..., x_N^*$. Then, we can write the Taylor expansion around the $x_i$ as

$$f_i(x_1^*, x_2^*, ..., x_N^*) = f_i(x_1, x_2, ..., x_N) + \sum_j (x_j^* - x_j) \frac{\partial f_i}{\partial x_j} + ... \tag{61}$$

In vector notation, this becomes

$$\mathbf{f}(\mathbf{x}^*) = \mathbf{f}(\mathbf{x}) + \mathbf{J} \cdot (\mathbf{x}^* - \mathbf{x}) + ... \tag{62}$$

where $\mathbf{J}$ is the Jacobian matrix, the $N \times N$ matrix with elements $J_{ij} = \frac{\partial f_i}{\partial x_j}$.

Since $\mathbf{x}^*$ is a root of the equations, $\mathbf{f}(\mathbf{x}^*) = 0$. Neglecting higher order terms and setting $\Delta\mathbf{x} \equiv \mathbf{x} - \mathbf{x}^*$, we have

$$\mathbf{J} \cdot \Delta\mathbf{x} = \mathbf{f}(\mathbf{x}) \tag{63}$$

This is a set of linear equations of the form $\mathbf{A}\,\mathbf{x} = \mathbf{v}$, thus we could solve it using (e.g.) the Gaussian elimination method or the Gauss-Seidel method.

Once we have solved for $\Delta\mathbf{x}$, our new estimate $\mathbf{x}'$ of the position of the root is

$$\mathbf{x}' = \mathbf{x} - \Delta\mathbf{x} \tag{64}$$

Thus, applying the Newton-Raphson method for more than one variable

involves evaluating the Jacobian matrix **J** of first derivatives at the point $x$, then solving a system of linear equation for $\Delta x$, then using the result to calculate a new estimate of the root.

**EXERCISE:**

Write a script to implement relaxation method, bisection method and Newton-Rapshon. Use them to find the zeros of the following function of the eccentric anomaly E:

$$\mathcal{F} = \mathrm{E} - ecc * np.sin(\mathrm{E}) \tag{65}$$

In the above equation, $ecc$ is the eccentricity of a Keplerian binary system: it can take any value in the range $[0, 1)$. Please, try to solve the equation for three different values of $ecc = 0.1, 0.7, 0.9$.
The term $\mathcal{F} = 2\,\pi\,t_p/\mathrm{T}$ (where T is the orbital period of the binary and $t_p$ is the time elapsed from pericentre passage) is the mean anomaly and can take any values between 0 and $2\,\pi$. Please calculate the result for $\mathcal{F} = \pi$ and $\pi/3$.
Finally, E is the eccentric anomaly of a Keplerian binary system and is the unknown you should try to find with this exercise. For more information on the eccentric anomaly, see `https://en.wikipedia.org/wiki/Eccentric_anomaly` and the figure below.

Solutions of the exercise (with tolerance $10^{-6}$):
$\mathrm{E}(ecc = 0.1, \mathcal{F} = \pi/3) \sim 1.137976$
$\mathrm{E}(ecc = 0.7, \mathcal{F} = \pi/3) \sim 1.737494$
$\mathrm{E}(ecc = 0.9, \mathcal{F} = \pi/3) \sim 1.899123$
$\mathrm{E}(ecc = *, \mathcal{F} = \pi) = \pi$

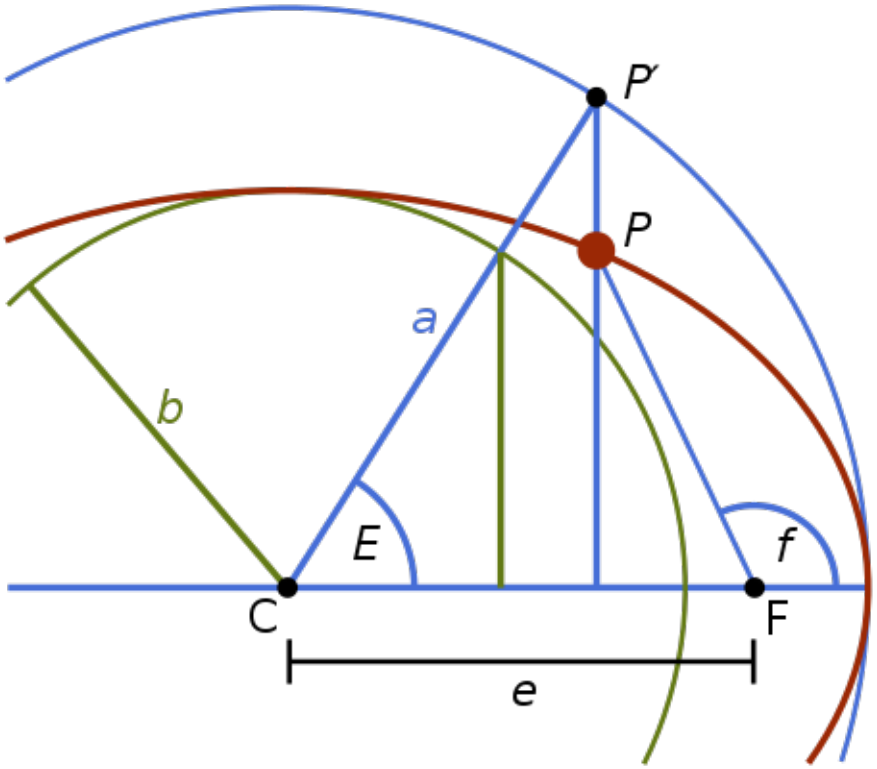


*Figure 27: Visualization of the eccentric anomaly (credits: CheCheDaWaff, Wikipedia).*

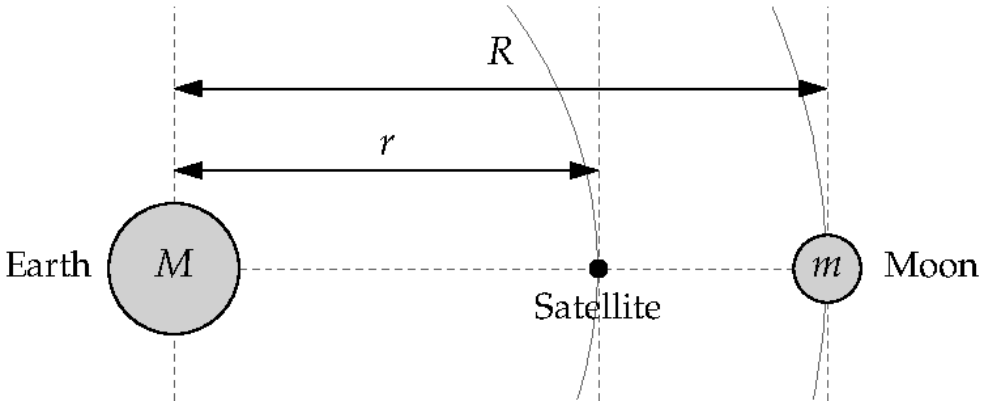


*Figure 28: Visualization of the $L_1$ Lagrangian point between Earth and Moon (credits: Mark Newman).*

**EXERCISE:**

The $L_1$ Lagrangian point between the Earth and the Moon is the point at which the inward pull of the Earth and the outward pull of the Moon balance, to give the centripetal force that keeps a satellite in a synchronous orbit with the Moon: a satellite in $L_1$ orbits the Earth in perfect synchrony with the Moon. See Figure 28 for a visual representation (from Mark Newman's book).
For simplicity, let's assume that the orbits are circular (zero eccentricity), that the mass of the satellity $m_{\text{Satellite}}$ is negligible with respect to both the mass of the Earth ($M_{\text{Earth}}$) and the mass of the Moon ($M_{\text{Moon}}$). Under such assumptions, you can easily find that the distance $r$ from the center of the Earth to the $L_1$ point satisfies

$$\frac{G\,M_{\text{Earth}}}{r^2} - \frac{G\,m_{\text{Moon}}}{(R-r)^2} = \omega^2\, r, \tag{66}$$

where $G = 6.674 \times 10^{-8}$ cm$^3$ g$^{-1}$ s$^{-2}$ is the gravity constant, $M_{\text{Earth}} = 5.974 \times 10^{27}$ g, , $m_{\text{Moon}} = 7.348 \times 10^{25}$ g, $R = 3.844 \times 10^{10}$ cm is the distance between Earth and Moon, $\omega = (G\,M_{\text{Earth}}/R^3)^{1/2}$ is the angular frequency of both the satellite and the moon.
Solve the above equations with the methods you know and estimate $r$.

# 7 NUMERICAL DERIVATIVES

This chapter is based on *Computational Physics* by Mark Newman `http://www-personal.umich.edu/~mejn/cp/`.

Textbooks on numerical methods often skip numerical derivatives, because

- they are (too) easy to understand;
- in many cases there is no need to calculate the derivative numerically, because we can do it analytically (but the Newton-Raphson method is an important example of a situation in which we may need to calculate derivatives numerically);
- there are significant problems with numerical derivative accuracy, when applied to noisy data (see below).

## 7.1 *Forward, backward and central differences*

A first derivative is defined as

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h} \tag{67}$$

Numerically, a first derivative can be calculated simply as follows

$$\frac{\mathrm{d}f}{\mathrm{d}x} \sim \frac{f(x+h) - f(x)}{h} \tag{68}$$

where $h$ is a "small" interval. The above equation is called **forward difference**, because the derivative is calculated over the interval following $x$.

In contrast, a **backward difference** is calculated over the interval preceding $x$:

$$\frac{\mathrm{d}f}{\mathrm{d}x} \sim \frac{f(x) - f(x-h)}{h} \tag{69}$$

Usually, the backward difference and the forward difference give similar answers, but their goodness depends on the behaviour of the function $f$ over the interval. In any case, it is always safer to calculate the **central difference**, i.e. the difference with respect to the mid point of the interval:

$$\frac{\mathrm{d}f}{\mathrm{d}x} \sim \frac{f(x+h/2) - f(x-h/2)}{h} \tag{70}$$

## 7.2 *Second derivatives*

In a similar way, we can estimate also the second derivatives of a function. Let's take the first derivatives calculated with the central difference in the following points:

$$f'(x+h/2) \sim \frac{f(x+h)-f(x)}{h}$$
$$f'(x-h/2) \sim \frac{f(x)-f(x-h)}{h} \tag{71}$$

Then, we apply again the central difference for the second derivative:

$$f''(x) \sim \frac{f'(x+h/2)-f'(x-h/2)}{h}$$
$$= \frac{[f(x+h)-f(x)]/h - [f(x)-f(x-h)]/h}{h} = \frac{f(x+h)-2f(x)+f(x-h)}{h^2} \tag{72}$$

## 7.3 *Partial derivatives*

If you have a function $f(x,y)$ of two variables, the central difference approximations to partial derivatives with respect to $x$ and $y$ are simply

$$\frac{\partial f}{\partial x} = \frac{f(x+h/2,y)-f(x-h/2,y)}{h}$$
$$\frac{\partial f}{\partial y} = \frac{f(x,y+h/2)-f(x,y-h/2)}{h} \tag{73}$$

## 7.4 *Derivatives of noisy data*

If the function we are sampling is actually a set of data with some noise (see e.g. figure 29), the numerical derivative of this function will be even noisier. The reason is that locally, the slope of the function in a given point might be dominated by the slope of the noise in that point.

There are three ways to improve numerical derivatives in this case: i) considering larger $h$, so that the contribution of noise cancels out (but often this is not sufficient); ii) fitting a smooth curve to the data and then calculating the derivative of the fit; iii) calculating the Fourier transforms of the data, which makes them smoother (see Section 14)

**EXERCISE:**

You have already written a script to perform root finding with the Newton-Raphson method. Now modify that script to include a numerical derivative (for the case of eccentric anomaly) with the central difference method.

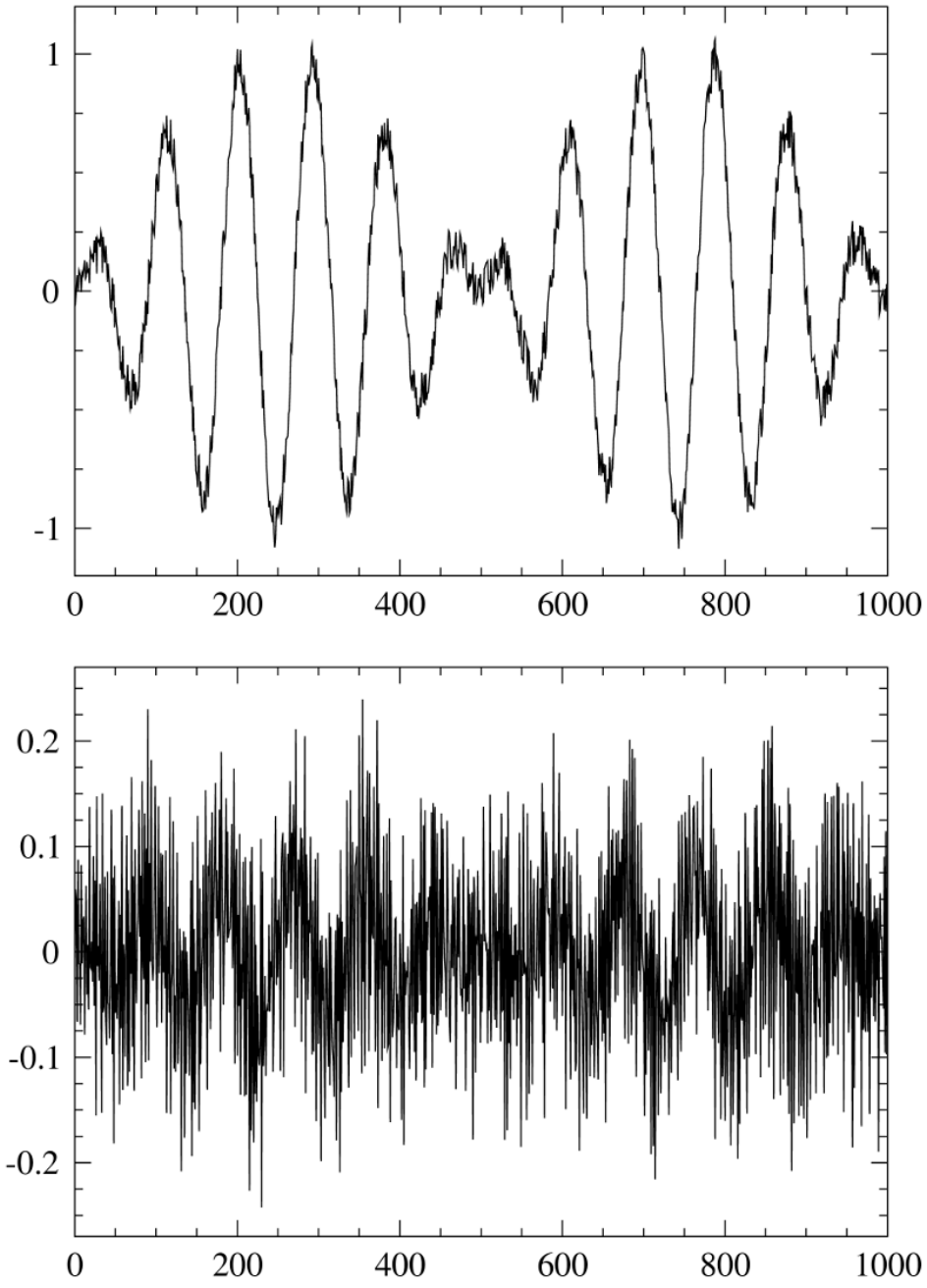


*Figure 29: Top panel (a): example of noisy data; bottom panel (b): numerical derivative of the noisy data (credits: Mark Newman, Computational physics).*

# 8 RANDOM NUMBERS

This chapter is based on personal notes, on *Computational Physics* by Mark Newman `http://www-personal.umich.edu/~mejn/cp/` and on *Numerical Recipes in C* by William H. Press and Saul A. Teukolsky `http://www.nrbook.com/a/bookcpdf.php`.

Some (astro)physical processes are random. For example radioactive decay. Quantum mechanics tells us the probability of decays per unit time, but the exact moment of a decay is random and cannot predicted.

Other (astro)physical processes are not intrinsically random, but we might need random numbers to represent them. For example, if we want to produce a mock sample of astrophysical data (e.g. magnitudes of individual stars in a star cluster), we might either produce this sample from a regular grid of values or randomly draw some quantities from a distribution function.

For example, in computational astrophysics, initial conditions of simulations are often drawn from random numbers. If we want to simulate a galaxy or a star clusters with a number of stellar particles, we need to first assign positions and velocities to these particles. If we perform this operation over a fixed grid, we will get initial conditions that look quite unnatural (stars are not evenly spaced in nature..). In contrast, if we randomly draw initial positions and velocities from distribution functions, we get something that looks more like "a real thing".

Thus, it is important to know how to produce a set of random numbers with a computer. How can a computer produce "genuine" random numbers? Indeed, it cannot. Computer-generated random numbers are "pseudo-random" numbers, because they are generated by a deterministic formula.

## 8.1 *Random generators*

Consider the following equation

$$x' = (a\,x + c) \mod m, \tag{74}$$

where $a$, $c$ and $m$ are integer constants chosen such that $m > 0$, $0 < a < m$, $0 \leq c < m$, and $x$ is an integer variable. Given a value for $x$, this equation calculates an integer value and assigns it to $x'$. Now suppose we take that new value of $x'$ and plug it back in on the right-hand side of the equation (replacing $x$) again and get another value, and so on, generating a series of integers.

With a python script, this can be implemented as follows

```
#file examples/random/rand_gen.py
N = 100
a = int(1664525)
c=int(1013904223)
m=int(4294967296)
x=1
results = []
for i in range(N):
     x = (a*x+c)%m
     results.append(x)
print(results)
```

This programs calculates the first 100 numbers in the sequence generated by equation 74 with $a = 1664525$, $c = 1013904223$ and $m = 4294967296$. This is a **linear congruential random number generator**. It generates a series of apparently random integers, by iterating the same linear equation over and over.

Note that $m$ must always be larger than the sequence of random numbers we want to generate. If we generate a sequence of N values with $N > m$ we end up repeating the same sequence from the beginning, that is the $m + 1$ number will be the same as the first number of the sequence, the $m + 2$ number will be the same as the second number of the sequence and so on. This is a terrible mistake when using random numbers.

Of course, if we generate the sequence again and again, we will obtain always the same sequence, unless we change the starting value of $x$. This is important because it guarantees **reproducibility** of the results of any possible code/paper adopting this script, and reproducibility is one of the main requirements in scientific experiments. Besides, reproducibility of a random number series is a blessing when debugging a code.

Here below, we summarize the possible functions that generate random numbers in python. There are at least two packages that contain functions for random numbers: **random** and **numpy.random** (the latter is clearly a sub-package of numpy).

From the **random** package, the function **random.random()** generates floating point random numbers between 0 and 1. To obtain a random number in a different range (for example between min=4. and max=100.), we can do the following

```
#file examples/random/use_random.py
import random

min=4.
max=100.

a=random.random()
b=a*(max-min)+min

print(a,b)
```

where b is a floating point random number between 4 and 100.

The function **random.randint(min,max)**, which is also part of the random package, generates integer random numbers between min and max.

From the **numpy.random** package, the function **numpy.random.rand()** generates floating point random numbers between 0 and 1. Even in this case, to generate a random number in a different range, we can do the following

```
#file examples/random/use_nprandom.py
import numpy as np

min=4.
max=100.

a=np.random.rand()
b=a*(max-min)+min

print(a,b)
```

where b is a floating point random number between 4 and 100.

## 8.2 *Random number seeds*

A **seed** is the input value that tells the random generator where to start its sequence (for example, in the linear congruential generator of equation 74 it is the very first value of x). The seed uniquely determines the entire series. Usually, when the user does not provide the seed manually, the seed is automatically chosen based on the date of the computer. Thus, if we want to ensure the reproducibility of our random series, it is better to choose the seed manually and record it.

To assign the seed with random.seed, we can proceed the following way:

```
#examples/random/use_seed.py
from random import random, seed

seed(42) #assign 42 as seed
for i in range(10):
    a=random()
    print(a)
```

To assign the seed with numpy.random.seed, we can proceed the following way:

```
#examples/random/use_npseed.py
from numpy import random

random.seed(42) #assign 42 as seed
for i in range(10):
    a=random.rand()
    print(a)
```

Note that with numpy.random you can avoid to use the for cycle and generate all the randoms you need, storing them into one array:

```
#examples/random/use_npseed.py
from numpy.random import random, seed

seed(42) #assign 42 as seed
for i in range(10): #calculate 10 random numbers with a loop
    a=random()
    print(a)

seed(42) #assign 42 as seed
b=random(10) #calculate 10 random numbers
             #with properties of numpy arrays
print(b)
```

## 8.3 *Uniform and non-uniform random numbers*

The random numbers generated by a random generator are **uniform deviates**: each of them has the same probability to be generated within a given range. Mathematically, this is equivalent to say that the probability distribution function is constant over the range:

$$p(x)\,\mathrm{d}x = \mathrm{const}\,\mathrm{d}x \tag{75}$$

where $p(x)$ is the probability distribution function, and $p(x)\,\mathrm{dx}$ is the probability to draw a random number between $x$ and $x + \mathrm{d}x$. For a uniform deviate $p(x) = \text{const}$, where const is a normalization constant (which guarantees that $\int p(x)\,\mathrm{d}x = 1$).

For a general (astro)physical problem, it is more likely that we need random numbers generated according to a specific non-constant probability distribution function. For example, if we want to produce a set of (mock) measurements with their statistical error, and we know that the statistical error is well reproduced by a Gaussian distribution, we need to draw random numbers following a Gaussian probability distribution function.

In this case, we proceed as follows. We first generate uniform deviates with the random generator and then we transform these uniform deviates into non-uniform deviates thanks to the laws of probability.

We will see two techniques:

- inverse random sampling;
- rejection method.

In addition, we will see the Box-Muller method to calculate Gaussian deviates, which is a sophisticated version of the inverse random sampling method.

## 8.4 *Inverse random sampling*

The fundamental transformation law of probabilities tells that the probability $p(y)\,\mathrm{dy}$ of generating a number between $y$ and $y + \mathrm{dy}$ is equal to the probability $q(x)\,\mathrm{dx}$ of generating a number between $x$ and $x + \mathrm{dx}$, that is

$$p(y)\,\mathrm{d}y = q(x)\,\mathrm{d}x \tag{76}$$

regardless of the functional form of $p(y)$ and $q(x)$ and of the ranges, provided that they are properly defined probabilities over their respective ranges (i.e. $\int p(y)\,\mathrm{d}y = 1$, $\int q(x)\,\mathrm{d}x = 1$). Thus, the value of $y$ is related to $x$ by a function $y = y(x)$.

If $q(x)\,\mathrm{d}x$ is the probability to extract a uniform deviate between 0 and 1, then $q(x) = 1$ and $\int_0^x \mathrm{d}x' = x$. This means that for a generic $p(y)$

$$\int_{y_{\min}}^{y(x)} p(y')\,\mathrm{d}y' = \int_0^x \mathrm{d}x' = x. \tag{77}$$

This means that if we derive a uniform deviate $x$ between 0 and 1 it is possible to derive a non-uniform deviate $y(x)$ as a function of $x$. Thus, it is possible to generate a non-uniform random deviate from a uniform random deviate. The necessary condition to extract $y$ from $x$ is that we can solve the left-hand term

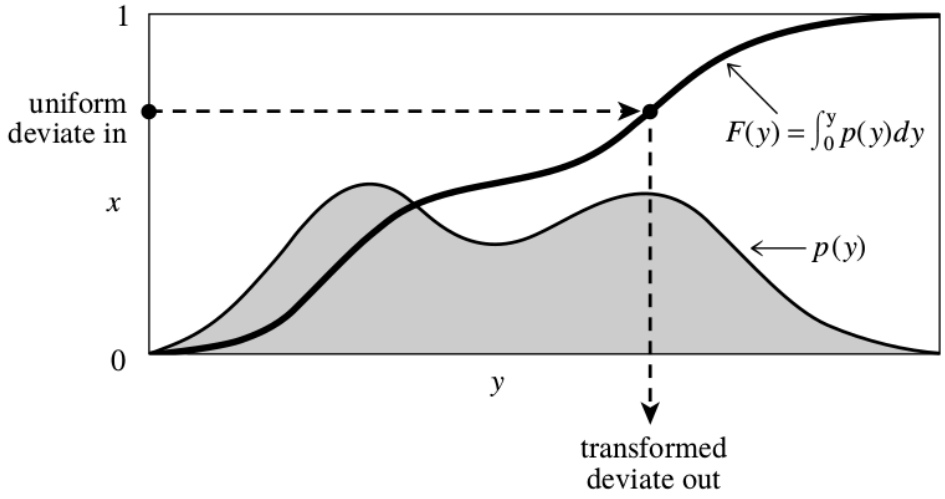


*Figure 30: Visualization of the inverse random sampling method. From Press et al., Numerical recipes in C, Second Edition, Cambridge University Press.*

of the integral 77 and that we can calculate $y(x)$. See Figure 30 for a visual representation.

For example, if $p(y) = 2\,y$ and $y_{\min} = 0$ $y_{\max} = 1$, then

$$\int_{y_{\min}}^{y(x)} 2\,y'\,\mathrm{d}y' = x, \tag{78}$$

which gives

$$y(x) = x^{0.5} \quad \text{if } y_{\min} = 0 \tag{79}$$

and

$$y(x) = \sqrt{x + y_{\min}^2} \tag{80}$$

in the general case.

This method is called **inverse random sampling**, because the function $y(x)$ must be defined an invertible to derive $y$ from $x$.

In practice, the steps we need to do to apply this method are the following.

1. Take a probability distribution function $p(y)$ of the quantity $y$ you want to sample.

2. Integrate $p(y)\,\mathrm{d}y$ over the range to obtain the cumulative probability distribution function $\mathrm{P}(y) = \int_{y_{\min}}^{y} p(y')\mathrm{d}y'$

3. P($y$) is monotonic and takes values from 0 to 1 by definition of probability.
4. Randomly sample the values $x = \mathrm{P}(y)$ of the cumulative distribution function between 0 and 1 (with a random generator).
5. Invert the function P($y$) to get $y = \mathrm{P}(y)^{-1}$.
6. Repeat steps 4 and 5 as many times as you need to get $y$ for N random numbers.

**EXERCISE:**

The Salpeter mass function [Salpeter, 1955] is one of the most popular initial mass functions for stars. It is defined as

$$p(m)\,\mathrm{d}m = \mathrm{const}\; m^{-\alpha}\,\mathrm{d}m, \tag{81}$$

where $\alpha = 2.3$. It means that, given a population of stars in the zero-age main sequence, the probability to have a star of mass $m$ in this population is $p(m) = \mathrm{const}\; m^{-\alpha}$. Massive stars are significantly less common than light stars.
Assuming that the minimum stellar mass is $m_{\mathrm{min}} = 0.1\ \mathrm{M}_{\odot}$ and the maximum stellar mass is $m_{\mathrm{max}} = 150\ \mathrm{M}_{\odot}$, randomly calculate the mass of $10^6$ stars distributed according to the Salpeter initial mass function by using the inverse random sampling technique. Plot the resulting population of stellar masses with an histogram. The result should look like Figure 31.

*Suggestion: First you have to calculate the normalization constant* const *and the analytic integral of* $\int_{m_{\mathrm{min}}}^{m} p(m')\,dm'$.

## 8.5 *Gaussian random numbers*

Many data samples in astrophysics follow a Gaussian distribution:

$$p(x) = \frac{1}{\sqrt{2\,\pi\sigma^2}}\,\exp\left(\frac{-x^2}{2\,\sigma^2}\right), \tag{82}$$

where $\sigma$ is the standard deviation and the factor in front of the exponential is a normalization. The cumulative probability distribution function associated with equation 84 is

$$\mathrm{P}(x) = \frac{1}{\sqrt{2\,\pi\sigma^2}}\int_{-\infty}^{x}\exp\left(\frac{-x'^2}{2\,\sigma^2}\right)\mathrm{d}x', \tag{83}$$

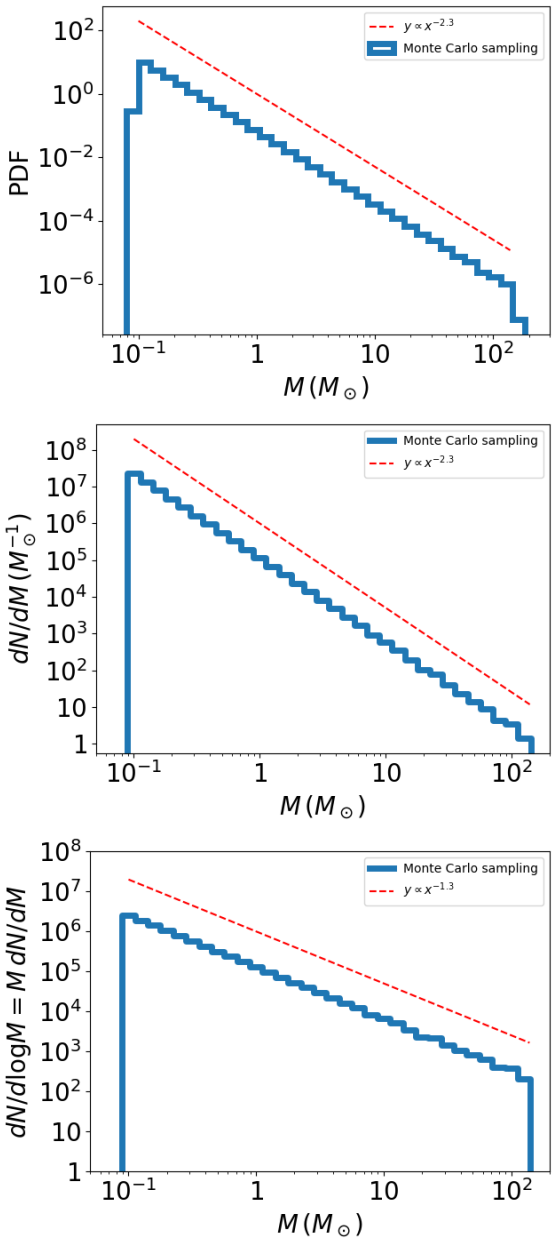


*Figure 31: Result of the exercise on Salpeter mass function, written in three different forms. Take home message: be careful with log–log histograms.* **Upper panel:** *Distribution of stellar masses randomly generated according to a Salpeter mass function between 0.1 and 150 $M_\odot$ (blue histogram). The red dashed line shows the slope $y \propto x^{-2.3}$. The y axis is a well defined probability distribution function (PDF), which I generated using the option* **density=True** *of matplotlib.pyplot.hist(). If you use python2 the option is normed=True.* **Central panel:** *Same as in the upper panel, but I build the histogram manually with the function matplotlib.pyplot.step(). Note that in order to obtain the values on the y–axis of this plot, you should take the number $d$N of elements per bin and divide it by $d$M, where $d$M is the* **linear** *size of the bin. If you use* **logarithmic** *bins $d\log$M, you have to calculate $d$M = M $d\log$M and then you get $d$N/$d$M = M$^{-1}$ $d$N/$d\log$M.* **Lower panel:** *the same as the central panel but here the y–axis reports the values of $d$N/$d\log$M = M $d$N/$d$M. Apart from a normalization constant, this is equivalent to just plotting $d$N on the y–axis. Hence, the red dashed line shows the slope $y \propto x^{-1.3}$. All the three panels are correct representations of the result. It is important that you know what you put on the y–axis. To make it simpler, just plot the probability distribution function (PDF) using the option density=True of hist, but I want you to be aware of those logical steps.*

which cannot be integrated. However, we can use the following trick.

Imagine we have two independent random numbers $x$ and $y$ both drawn from a Gaussian distribution with the same standard deviation $\sigma$. The probability that the point with position vector $(x, y)$ falls in some small element $\mathrm{d}x\,\mathrm{d}y$ of the Cartesian $xy$ plane is then

$$p(x)\mathrm{d}x\,p(y)\mathrm{d}y = \frac{1}{\sqrt{2\,\pi\sigma^2}}\,\exp\left(\frac{-x^2}{2\,\sigma^2}\right)\mathrm{d}x\,\frac{1}{\sqrt{2\,\pi\sigma^2}}\,\exp\left(\frac{-y^2}{2\,\sigma^2}\right)\mathrm{d}y$$

$$= \frac{1}{2\,\pi\sigma^2}\,\exp\left(-\frac{x^2+y^2}{2\,\sigma^2}\right)\mathrm{d}x\,\mathrm{d}y \tag{84}$$

Now we can rewrite the above formula in polar coordinates $(r, \theta)$, by using the well known coordinate transformation $x^2 + y^2 \to r^2$, $\mathrm{d}x\,\mathrm{d}y \to r\,\mathrm{d}r\,\mathrm{d}\theta$:

$$p(r, \theta)\,\mathrm{d}r\,\mathrm{d}\theta = \frac{1}{2\,\pi\sigma^2}\,\exp\left(-\frac{r^2}{2\,\sigma^2}\right)r\,\mathrm{d}r\,\mathrm{d}\theta$$

$$= \frac{r}{\sigma^2}\,\exp\left(-\frac{r^2}{2\,\sigma^2}\right)\mathrm{d}r\,\frac{\mathrm{d}\theta}{2\,\pi} \tag{85}$$

in the above equation, we can perfectly separate the term depending on $r$ and the term depending on $\theta$ and both of them have the correct normalization to be a probability distribution function. In fact,

$$p(r)\,\mathrm{d}r = \frac{r}{\sigma^2}\,\exp\left(-\frac{r^2}{2\,\sigma^2}\right)\mathrm{d}r \tag{86}$$

$$p(\theta)\,\mathrm{d}\theta = \frac{1}{2\,\pi}\,\mathrm{d}\theta \tag{87}$$

These distribution functions can be easily integrated and inverted:

$$\mathrm{P}(r) = \int_0^r \frac{r'}{\sigma^2}\,\exp\left(-\frac{r'^2}{2\,\sigma^2}\right)\mathrm{d}r' = 1 - \exp\left(-\frac{r^2}{2\,\sigma^2}\right) \tag{88}$$

$$\mathrm{P}(\theta) = \int_0^\theta \frac{1}{2\,\pi}\,\mathrm{d}\theta' = \frac{\theta}{2\,\pi} \tag{89}$$

We can now use the inverse sampling technique to generate $\theta$ and $r$, by generating two uniform random numbers $z_1$ and $z_2$ between 0 and 1, where $z_1 = \mathrm{P}(r)$ and $z_2 = \mathrm{P}(\theta)$, and we obtain

$$r = \sqrt{-2\,\sigma^2\,\ln(1 - z_1)} \tag{90}$$

$$\theta = 2\,\pi\,z_2 \tag{91}$$

Finally, we derive $x$ and $y$ by transforming back from polar to Cartesian

coordinates $x = r \cos\theta$, $y = r \sin\theta$. In this way, we have generated two random numbers ($x$ and $y$), each of them distributed according to a Gaussian centered in zero with standard deviation $\sigma$. We can use one of the two numbers and store the other one for the future. If we want a Gaussian with a different mean value, we can simply shift the random numbers by the desired value.

**Note: in python you can directly generate Gaussian deviates by using the function numpy.random.normal(loc=0.0,scale=2.0), where loc indicates the mean and scale indicates the $\sigma$, or the function random.gauss(0.0, 2.0), where the first and the second arguments are - again - the mean and the $\sigma$.**

**EXERCISE:**

Write a script to generate $N = 10^5$ Gaussian deviates with the Box-Muller method. Assume $\sigma = 2$ and that the Gaussian is centered on zero. The result should look like Figure 32.

It can be shown that a Maxwellian curve

$$p(v)\,dv = \sqrt{\frac{2}{\pi}}\,\frac{v^2}{\sigma^3}\,\exp\left(-\frac{v^2}{2\,\sigma^2}\right)dv \tag{92}$$

can be randomly sampled as $v = \sqrt{x^2 + y^2 + z^2}$, where $x$, $y$ and $z$ are random numbers distributed according to a Gaussian probability distribution function centered around zero and with the same value of $\sigma$. Several astrophysical processes follow a Maxwellian, too. For example, natal kicks of pulsars [Hobbs et al., 2005].

**EXERCISE:**

Write a script to generate $N = 10^5$ random numbers following a Maxwellian distribution with $\sigma = 265$ km s$^{-1}$. According to Hobbs et al. [2005], this is the distribution of natal kicks of neutron stars.

*Note: every compact object which forms from a supernova is thought to receive a kick at birth. The main reason is that linear momentum is conserved during a supernova and asymmetries in the ejecta (or in neutrino losses) push the compact object to move in the opposite direction with respect to the bulk of the ejecta.*

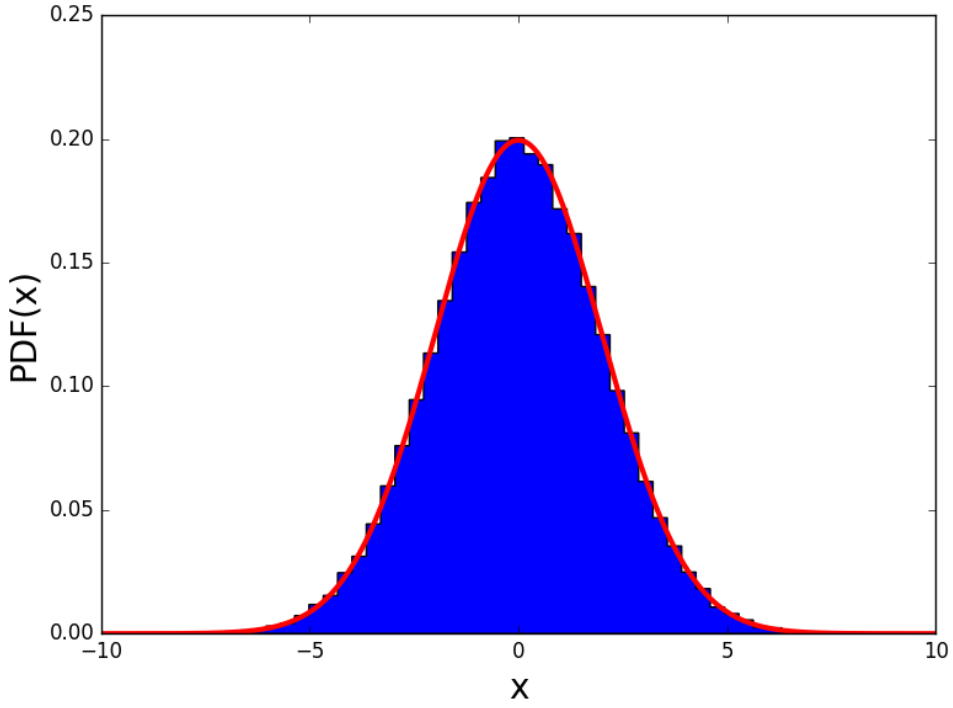


*Figure 32: Blue histogram: sample of* $10^5$ *random values distributed according to a Gaussian distribution. The values were generated with the Box-Muller method. Red line: analytic Gaussian distribution.*

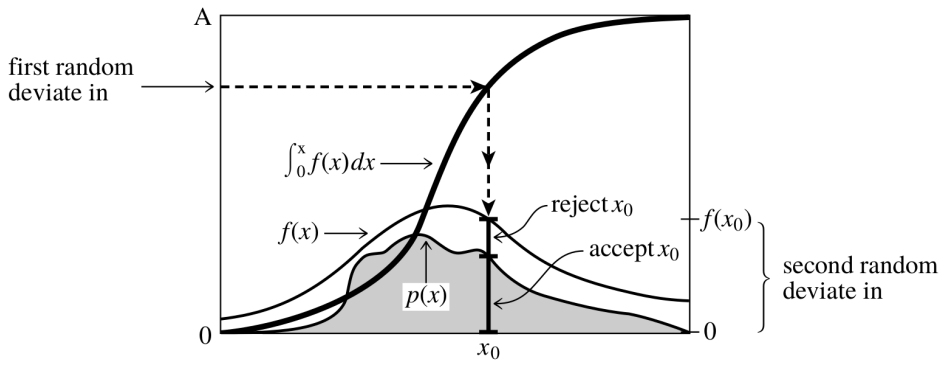


*Figure 33: Visualization of the rejection sampling method. From Press et al., Numerical recipes in C, Second Edition, Cambridge University Press.*

## 8.6 *Rejection method*

What can we do when the cumulative distribution function cannot be either calculated or inverted (easily)? We can follow the **rejection sampling approach**.

1. Take a probability distribution function $p(x)$ of the quantity $x$ you want to sample. But $p(x)$ is difficult/impossible to integrate (or it can be integrated but then the cumulative distribution cannot be inverted)!

2. Take a second function $f(x)$, with $f(x) > p(x)$ everywhere, that can be easily integrated and whose cumulative distribution function can be easily inverted, to obtain the cumulative distribution function $g(x) = \int_{x_{\min}}^{x} f(x')\mathrm{d}x'$ (note that g(x) is NOT a well defined probability).

3. Randomly sample $y = g(x)$ between min and max value.

4. Invert $g(x)$ to obtain $x$. $x$ is distributed according to $f(x)$.

5. Generate a second random number $m$ uniform between 0 and $f(x)$. Reject $x$ if $m > p(x)$ and accept $x$ if $m \leq p(x)$.

6. Repeat 3, 4 and 5 as many times as you need to get $x$ for N particles.

This procedure is visualized in figure 33.

**EXERCISE:**

Generate $10^4$ points distributed according to a Gaussian (with $\sigma = 2$ and centered around zero) with the rejection method, by using the distribution function $f(x) = 1$, uniform between $min = -50$ and $max = +50$. The result should look like Figure 34.

*Suggestion: once again, note that $f(x)$ is not a well defined probability distribution function – and it cannot be, because $f(x) > p(x)$ everywhere, where $p(x)$ is a well defined probability function (the Gaussian PDF in this case). Hence $y(x) = \int_{x_{\min}}^{x} f(x)\,dx$ is a uniform random number but does not necessarily lie in the interval between 0 and 1. You must first calculate*

$$y_{\max} = \int_{x_{\min}}^{x_{\max}} f(x)\,dx \tag{93}$$

*Hence, you should draw an uniform random number $y \in [0, y_{\max}]$.*

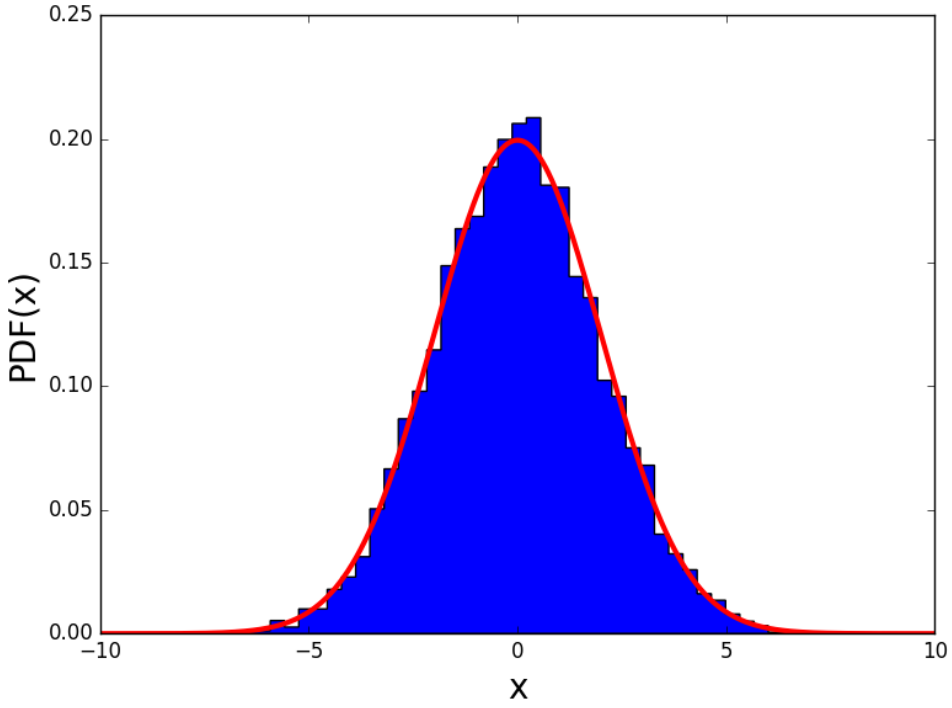


*Figure 34: Blue histogram: sample of $10^4$ random values distributed according to a Gaussian distribution. The values were generated with the rejection method. Red line: analytic Gaussian distribution.*

**EXERCISE:**

The Plummer sphere [Plummer, 1911] is a density distribution function expressed as

$$\rho(r) = \frac{3\,M}{4\,\pi a^3}\left(1+\frac{r^2}{a^2}\right)^{-5/2}, \tag{94}$$

where $\rho(r)$ is the mass density (g cm$^{-3}$), $a$ is a typical scale-length, $r$ is the radial coordinate (in spherical coordinates) and M is the total mass of the sphere. The Plummer sphere is used in astrophysics because it is the simplest distribution function that mimics the radial distribution of stars in a star cluster.
The goal of this exercise is to build an N-body model for a virialized star cluster, where stellar masses are randomly drawn from a Salpeter IMF, positions are randomly drawn from a Plummer sphere density distribution and stellar velocities are randomly drawn from a Maxwellian curve:

- stellar masses are drawn from a Salpeter mass function with $m_{\rm min} = 0.1\ M_\odot$ and $m_{\rm max} = 150\ M_\odot$;
- stellar positions are drawn from a Plummer spere with scale radius $a = 1$ pc;
- stellar velocities are drawn from a Maxwellian distribution;
- the star cluster is in virial equilibrium, i.e. $Q_{\rm vir} = 2\,K/|W| = 1$, where K and W are the kinetic and the potential energy of the cluster.

For simplicity, let us also assume that:

- the cluster has $10^3$ stars (you can try also $10^4$ after your script is debugged: it takes longer to run);
- the mass of a star does not depend on star's position inside the cluster (no initial mass segregation);
- the velocity of a star does not depend on star's position inside the cluster;
- the distribution of stellar positions and velocities are isotropic.

If you think you already understood what you need to do, stop reading here and start working. If you need some more suggestions, keep reading.

**EXERCISE, suggestion:**

The first step to do is to randomly draw stellar masses from the Salpeter IMF, as you have done in a previous exercise (just import your function or copy-and-paste it to the new script). As you can simply verify, drawing $10^3$ stars is equivalent to a mass of $\sim 300 - 500\ M_\odot$ (a part from stochastic fluctuations), because the average stellar mass is $\sim 0.3 - 0.5\ M_\odot$. Save the exact value of the total mass in M.

The second step is to randomly draw stellar positions according to a Plummer. On the other hand, equation 94 is a density distribution, while we need the mass distribution assuming the cluster is isotropic, hence:

$$dm = \rho\, dV = \rho(r)\, r^2 \sin\theta\, d\theta\, d\phi\, dr, \tag{95}$$

where I have used the definition of the volume element in spherical coordinates $dV = r^2 \sin\theta\, d\theta\, d\phi\, dr$.
The good news of assuming spatial isotropy is that the angular coordinates ($\theta$ and $\phi$) are independent from each other and from the radial coordinate $r$: we can draw $r$, $\theta$ and $\phi$ as three independent random numbers, from three different distributions.
Let's start with $\phi$, which is the simplest one. The cumulative probability distribution of $\phi$ is

$$P(\phi) = \frac{1}{2\pi} \int_0^\phi d\phi' = \frac{\phi}{2\pi} \tag{96}$$

Thus, we can draw a uniform random number $P(\phi)$ from 0 to 1 and then estimate $\phi$ as $\phi = 2\pi P(\phi)$.
Now, let's calculate $\theta$: the cumulative probability distribution of $\theta$ is

$$P(\theta) = \frac{1}{2} \int_0^\theta \sin\theta'\, d\theta' = \frac{1}{2}(1 - \cos\theta) \tag{97}$$

Thus, I can draw a uniform random number $P(\theta)$ between 0 and 1 and then I can extract $\theta$ from $P(\theta)$ by simply inverting the above equation $\theta = \arccos[1 - 2P(\theta)]$.

**EXERCISE, suggestion:**

To extract $r$ is a bit more complicated. The cumulative distribution function of mass is

$$\begin{aligned} M(r) = \int_V \rho \, dV &= \int_0^{2\pi} d\phi \int_0^{\pi} \sin\theta \, d\theta \int_0^{r} \rho(r) \, r^2 \, dr, \\ &= 4\pi \int_0^{r} \frac{3M}{4\pi a^3} \left(1 + \frac{r^2}{a^2}\right)^{-5/2} r^2 \, dr, \\ &= M \left(\frac{r}{a}\right)^3 \left(1 + \frac{r^2}{a^2}\right)^{-3/2} \end{aligned} \tag{98}$$

Hence, the probability cumulative distribution function is

$$P(r) = \left(\frac{r}{a}\right)^3 \left(1 + \frac{r^2}{a^2}\right)^{-3/2} \tag{99}$$

After few math. steps, we find that equation 99 can be inverted. We can draw a uniform random number $P(r)$ from 0 and 1 and then obtain $r = \sqrt{\frac{a^2}{P(r)^{-2/3} - 1}}$.
Finally, we randomly draw $10^3$ values for $\phi$, $\theta$, and $r$. It might be easier to plot our points in Cartesian coordinates, thus we simply make the conversion:

$$\begin{aligned} x &= r \sin\theta \cos\phi \\ y &= r \sin\theta \sin\phi \\ z &= r \cos\theta \end{aligned} \tag{100}$$

**EXERCISE, suggestion:**

Now, we must assign a velocity to each of the $10^3$ particles. Since stellar velocities do not depend on positions (according to our assumption), we do not need to worry about stellar position in the cluster (in real-life clusters we should). Moreover, since we assumed that velocities are isotropic, we can do the same as we did for positions and generate the modulus of velocity $v$, and the angles $\theta$ and $\phi$ independently of each other. For $\theta$ and $\phi$ we do exactly as in equations 97 and 96. To generate $v$, you can use the script you produced for the exercise on the Box-Muller method and extend it to generate Maxwellian deviates, knowing that $v_{\rm Maxwellian} = \sqrt{x^2 + y^2 + z^2}$, where $x$, $y$ and $z$ are random deviates distributed according to a Gaussian distribution function centered on zero (mean value equal to zero).
We need to assign a value to the standard deviation of the three Gaussian curves (they must have the same standard deviation to produce a Maxwellian curve, when combined together). Let's start from a guess value $\sigma = 0.5$ km s$^{-1}$. This value will be updated later to the correct one, when we impose the virial condition. Note that the standard deviation of the Gaussians will become the one-dimensional root mean square parameter $\sigma$ of the Maxwellian.
Once we have generated $10^3$ new values for $v$, $\theta$ and $\phi$, we can convert to Cartesian coordinates by using the following transformation:

$$\begin{aligned} v_x &= v\ \sin\theta\cos\phi \\ v_y &= v\ \sin\theta\sin\phi \\ v_z &= v\ \cos\theta \end{aligned} \tag{101}$$

The last step consists in enforcing the virial equilibrium. We need to calculate the total kinetic energy and the total potential energy of the star cluster we have just generated, as

$$\begin{aligned} \mathrm{K} &= \frac{1}{2}\sum_{i=1}^{\mathrm{N}} m_i\, v_i^2 \\ \mathrm{W} &= -\mathrm{G}\sum_{i=1}^{\mathrm{N}}\sum_{j>i}^{\mathrm{N}} \frac{m_i\, m_j}{|\vec{x}_i - \vec{x}_j|}. \end{aligned} \tag{102}$$

We calculate the virial ratio of our cluster as $\mathrm{Q_{vir}} = 2\,\mathrm{K}/|\mathrm{W}|$. If $\mathrm{Q_{vir}} \neq 1$, we have to impose the virial condition. The simplest way to do so, given the isotropy of positions and velocities, is to rescale the velocities by:

$$\begin{aligned} v_x &= v_x/\mathrm{Q}_{\rm vir}^{1/2} \\ v_y &= v_y/\mathrm{Q}_{\rm vir}^{1/2} \\ v_z &= v_z/\mathrm{Q}_{\rm vir}^{1/2}. \end{aligned} \tag{103}$$



The correct result should look like Figure 35.

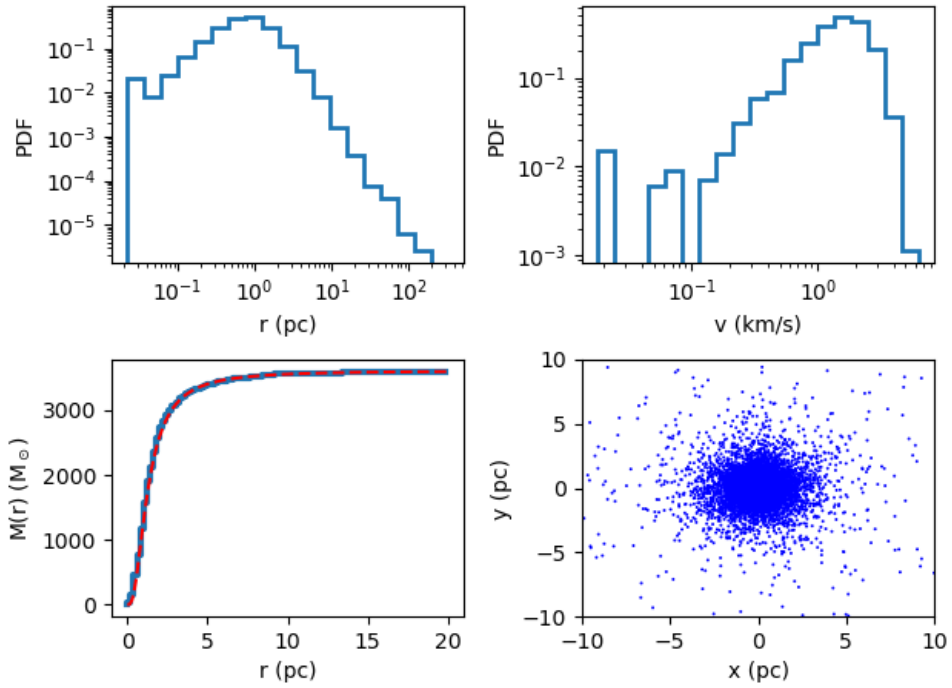


*Figure 35: Properties of the Plummer model. From top to bottom and from left to right: probability distribution function (PDF) of the stellar radii, PDF of the velocities, enclosed mass and scatter plot of the stellar positions in the xy plane.*

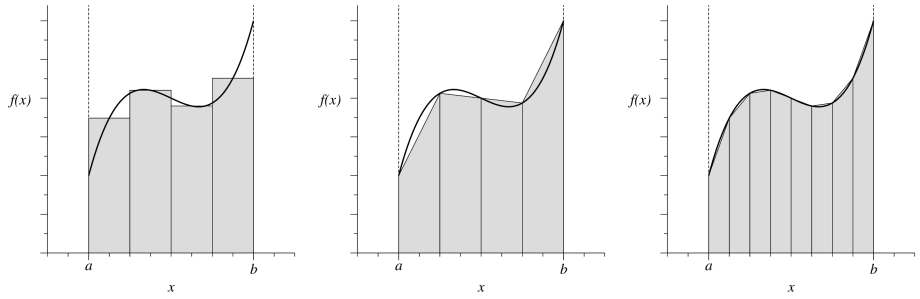


*Figure 36: Visualization of the trapezoidal rule (Credits: Mark Newman, Computational Physics.*

# 9 INTEGRATION OF FUNCTIONS

This chapter is based on *Computational Physics* by Mark Newman `http://www-personal.umich.edu/~mejn/cp/`.

Suppose we want to integrate a given function. Let's start from the simplest case: the integral of a function of a single variable over a finite range. There are many different methods to calculate the integral of a function numerically (for example: trapezoidal rule, Simpson's rule, Romberg integration, Gauss quadrature and Monte Carlo methods). Here we consider two of them: trapezoidal rule and Monte Carlo method.

## 9.1 *Trapezoidal rule*

Let's suppose we want to integrate function $f(x)$ between $x = a$ and $x = b$ (see Figure 36):

$$\mathrm{I}(a, b) = \int_a^b f(x)\,\mathrm{d}x. \tag{104}$$

This operation is equivalent to calculate the area under the curve $f(x)$ from $a$ to $b$. The simplest way to proceed is to divide this area into rectangular slices, calculate the area of each rectangle and then sum them up.

Instead of rectangles, we can use trapezoids and (as we see from the Figure) this is already a significant improvement. In practice, we divide the interval from $a$ to $b$ into N slices (or steps) so that each slice has a width $h = (b - a)/\mathrm{N}$. Then, the right-hand side of the $k - th$ slice falls at $a + k\,h$ and the left-hand side of the $k - th$ slice falls at $a + k\,h - h = a + (k - 1)\,h$. Thus, the area of the $k - th$ trapezoid is

$$\mathrm{I}_k = \frac{1}{2}\,h\,[f(a + (k - 1)\,h) + f(a + k\,h)], \tag{105}$$

which is called **trapezoidal rule**.

Now, the final integral will be written as the sum of the $\mathrm{I}_k$:

$$\mathrm{I}(a,b) \simeq \sum_{k=1}^{\mathrm{N}} \mathrm{I}_k = \frac{1}{2}\, h \sum_{k=1}^{\mathrm{N}} \left[ f(a+(k-1)\,h) + f(a+k\,h) \right],$$

$$= \frac{1}{2}\, h \left[ f(a) + 2\, f(a+h) + 2\, f(a+2\,h) + \ldots + 2\, f(a+(\mathrm{N}-1)\,h) + f(b) \right],$$

$$= h \left[ \frac{1}{2} f(a) + \frac{1}{2} f(b) + \sum_{k=1}^{\mathrm{N}-1} f(a+k\,h) \right] \quad (106)$$

## 9.2 *Estimate of the errors*

The numerical solution of an integral is an approximation. We want to estimate how good (or bad) this approximation is. Let's consider again our generic function $f(x)$ and let's make a Taylor expansion of $f(x)$ about the slice $k-1$ ($x_{k-1}$):

$$f(x) = f(x_{k-1}) + (x - x_{k-1})\, f'(x_{k-1}) + \frac{1}{2}(x - x_{k-1})^2\, f''(x_{k-1}) + \ldots \quad (107)$$

where $f'$ and $f''$ are the first and second derivatives of $f$, respectively. Integrating the above equation between $k-1$ and $k$:

$$\int_{x_{k-1}}^{x_k} f(x)\,\mathrm{d}x = f(x_{k-1}) \int_{x_{k-1}}^{x_k} \mathrm{d}x + f'(x_{k-1}) \int_{x_{k-1}}^{x_k} (x - x_{k-1})\,\mathrm{d}x + \frac{1}{2}\, f''(x_{k-1}) \int_{x_{k-1}}^{x_k} (x - x_{k-1})^2\,\mathrm{d}x + \ldots \quad (108)$$

If we substitute $u = x - x_{k-1}$ and we use the fact that $x_k - x_{k-1} = h$ (see the definition of $h$ for the trapezoidal rule):

$$\int_{x_{k-1}}^{x_k} f(x)\,\mathrm{d}x = f(x_{k-1}) \int_0^h \mathrm{d}u + f'(x_{k-1}) \int_0^h u\,\mathrm{d}u + \frac{1}{2}\, f''(x_{k-1}) \int_0^h u^2\,\mathrm{d}u + \ldots$$

$$= h\, f(x_{k-1}) + \frac{1}{2}\, h^2\, f'(x_{k-1}) + \frac{1}{6}\, h^3\, f''(x_{k-1}) + \mathcal{O}(h^4), \quad (109)$$

where $\mathcal{O}(h^4)$ denotes the rest of the terms of the series, that are smaller than $h^3$ and that we neglect.

We repeat the same Taylor expansion around $x_k$:

$$f(x) = f(x_k) + (x - x_k)\, f'(x_k) + \frac{1}{2}(x - x_k)^2\, f''(x_k) + \ldots \quad (110)$$

Then, we integrate from $x_{k-1}$ to $x_k$, using the same trick $u = x - x_{k-1}$ and

$h = x_k - x_{k-1}$. It is important to notice that $x_k = x_{k-1} + h$:

$$\int_{x_{k-1}}^{x_k} f(x)\,\mathrm{d}x = f(x_k)\int_{x_{k-1}}^{x_k}\mathrm{d}x + f'(x_k)\int_{x_{k-1}}^{x_k}(x - x_{k-1} - h)\,\mathrm{d}x + \frac{1}{2}f''(x_k)\int_{x_{k-1}}^{x_k}(x - x_{k-1} - h)^2\,\mathrm{d}x + \ldots$$

$$= f(x_k)\int_0^h \mathrm{d}u + f'(x_k)\int_0^h (u-h)\,\mathrm{d}u + \frac{1}{2}f''(x_k)\int_0^h (u-h)^2\,\mathrm{d}u + \ldots$$

$$= h\,f(x_k) + f'(x_k)\left[\int_0^h u\,\mathrm{d}u - h\int_0^h \mathrm{d}u\right] + \frac{1}{2}f''(x_k)\left[\int_0^h u^2\mathrm{d}u - 2\,h\int_0^h u\,\mathrm{d}u + h^2\int_0^h \mathrm{d}u\right] + \ldots$$

$$= h\,f(x_k) + f'(x_k)\left(\frac{1}{2}h^2 - h^2\right) + \frac{1}{2}f''(x_k)\left(\frac{1}{3}h^3 - h^3 + h^3\right) + \ldots$$

$$= h\,f(x_k) - \frac{1}{2}\,h^2\,f'(x_k) + \frac{1}{6}h^3\,f''(x_k) + \mathcal{O}(h^4) \quad (111)$$

Taking the average of equations 109 and 113, we get

$$\int_{x_{k-1}}^{x_k} f(x)\,\mathrm{d}x =$$

$$= \frac{1}{2}\,h\,[f(x_{k-1}) + f(x_k)] + \frac{1}{4}\,h^2\,[f'(x_{k-1}) - f'(x_k)] + \frac{1}{12}\,h^3\,[f''(x_{k-1}) + f''(x_k)] + \mathcal{O}(h^4) \quad (112)$$

Finally, we sum this expression over all the slices to get the integral from $a$ to $b$:

$$\int_a^b f(x)\,\mathrm{d}x = \sum_{k=1}^{\mathrm{N}}\int_{x_{k-1}}^{x_k} f(x)dx$$

$$= \frac{1}{2}\,h\,\sum_{k=1}^{\mathrm{N}}[f(x_{k-1}) + f(x_k)] + \frac{1}{4}\,h^2\,[f'(a) - f'(b)] + \frac{1}{12}\,h^3\,\sum_{k=1}^{\mathrm{N}}[f''(x_{k-1}) + f''(x_k)] + \mathcal{O}(h^4) \quad (113)$$

In the above equation, the term $\frac{1}{2}\,h\,\sum_{k=1}^{\mathrm{N}}[f(x_{k-1}) + f(x_k)]$ is identical to the trapezoidal rule. This means that when we apply the trapezoidal rule, we assume

$$\int_a^b f(x)\,\mathrm{d}x = \frac{1}{2}\,h\,\sum_{k=1}^{\mathrm{N}}[f(x_{k-1}) + f(x_k)] \quad (114)$$

and we discard all the other terms. The size of the terms we have discarded tells us what is the **approximation error** of the method. In principle, all $\mathcal{O}(h^2)$ terms are neglected in the trapezoidal method, which makes it a first-

order approximation method, i.e. it is accurate to $\mathcal{O}(h)$ and the leading-order approximation error is of the order of $h^2$.

This result has an intuitive and useful application: since the approximation error scales with $h^2$, we can make the error smaller by using a smaller value of $h$.

Numerical integration, as all the other numerical calculations, is subject also to **rounding errors**, which indicate the amount by which the finite computer's representation of the number is wrong. Thus, when approximation errors and the rounding errors become comparable, decreasing $h$ does not make any improvement in the result.

**EXERCISE:**

As we already discussed for random numbers, we can obtain the mass of an astrophysical system (galaxy, cluster of galaxies or stellar cluster) by computing the following integral

$$M = \int_0^{r_{max}} 4\,\pi\, r^2\, \rho(r)\, dr, \tag{115}$$

where $\rho(r)$ is the density distribution function (DDF) and $r_{max}$ is the maximum radius of the system (i.e. a cut-off radius).
Here below, you can find some examples of DDFs commonly used in astrophysics.
1. The **singular isothermal sphere** is possibly the simplest profile used to represent the density profile of an astrophysical system:

$$\rho(r) = \frac{\sigma^2}{2\,\pi\, G\, r^2}, \tag{116}$$

where $\sigma$ is the central velocity dispersion of the system and G is the gravity constant. The isothermal sphere is very simple but has two problems: the central divergence and the fact that mass never goes to zero at infinite distance from the centre (i.e. we must truncate it drastically). These problems can be cured by deriving more complex versions of this DDF (for example the lowered King model, [King, 1966]), but for the purpose of this exercise let's consider just the simple version of the isothermal sphere, given by equation 116.
Calculate the integral 115 for a singular isothermal sphere with the trapezoidal rule, assuming $\sigma = 10$ km s$^{-1}$ (typical of a star cluster) and $r_{max} = 10$ pc.

**EXERCISE, continued:**

2. The **Navarro-Frenk-White (NFW)** DDF is commonly used to describe simple spherical dark matter haloes (after Navarro et al. 1996):

$$\rho(r) = \frac{\rho_0}{\frac{r}{r_s}\left(1+\frac{r}{r_s}\right)^2}, \tag{117}$$

where $\rho_0$ is a normalization (in units of mass/volume) and $r_s$ is the scale radius (in units of length).
Calculate the integral 117 for a NFW profile with the trapezoidal rule, assuming $\rho_0 = 10^8\ M_\odot\ kpc^{-3}$, $r_s = 10$ kpc and $r_{max} = 100\, r_s$.

The integrals you just calculated numerically can be solved analytically (both for the isothermal sphere and for the NFW profile). Solve the integrals analytically (or google the correct forms, if you have problems with analytic integrals..). Compare the result of the numerical integration with the analytic value. How large are the errors? How much do they improve when reducing the step size in the integration?

Results:
Isothermal sphere: $M \sim 4.65 \times 10^5\ M_\odot$
NFW profile: $M \sim 4.56 \times 10^{12}\ M_\odot$

## 9.3 *Monte Carlo technique*

Random numbers can be used to integrate "challenging" functions, which vary too fast or require multi-dimensional integration.

Suppose we want to evaluate the integral

$$I = \int_0^2 \sin^2\left[\frac{1}{x\,(2-x)}\right] dx \tag{118}$$

The entire function $\sin^2\left[\frac{1}{x(2-x)}\right]$ for $x \in [0, 2]$ fits in the $2 \times 1$ rectangle (Figure 37). The area of the rectangle is $A = 2$. Thus, the value of the integral I (the area below the curve in Figure 37) is finite and must be less than $A = 2$.

**Let's use what we learned for the rejection method of random numbers.**

First, let's generate random points uniformly in the rectangle A. In order to do so, we have to generate two independent uniform random numbers, one for the $x$ and one for the $y$ Cartesian coordinates of the point. Obviously, the

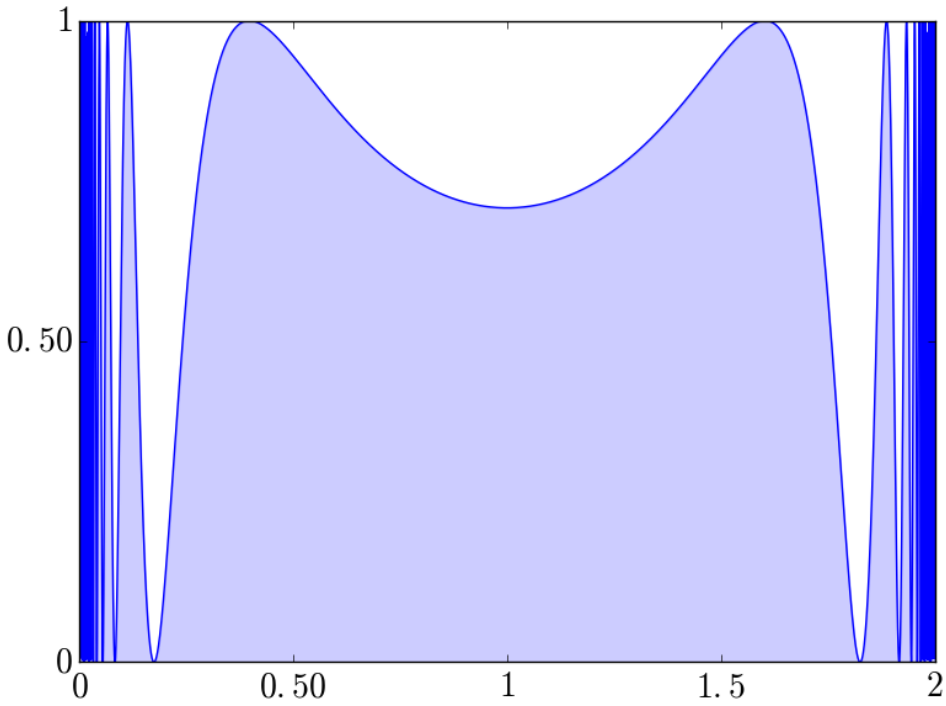


*Figure 37: The blue area is integral* I *of the function presented in equation 118. The entire rectangle defined by the x and y axes is* A.

$x$ coordinate will be generated uniformly in $[0,\ 2]$, while the $y$ coordinate will be generated uniformly in $[0,\ 1]$.

If we choose a point uniformly at random in the rectangle A (as detailed above), the probability that that point falls in the blue shaded region is $p = \mathrm{I/A}$. In other words, the probability that a random point falls under the curve rather than over is $p = \mathrm{I/A}$.

So, here is our scheme: we generate a large number N of random points in the bounding rectangle A, check each one to see if it is below the curve and keep a count of the number of points that are below the curve. Let's call this number $k$. This is equivalent to say that we take all the points below the curve and we reject (like in the rejection method) all the points above the curve.

The fraction of points below the curve $k/\mathrm{N}$ should be approximately equal to the probability $p$, that is

$$\mathrm{I} \simeq \frac{k\,\mathrm{A}}{\mathrm{N}} \tag{119}$$

**EXERCISE:**

Write a python script to perform the integration in equation 119 of the integral in equation 118. Calculate the integral with $\mathrm{N} = 10^3,\ 5\times10^3,\ 10^4,\ 5\times10^4,\ 10^5,\ 5\times10^5,\ 10^6,\ 5\times10^6$. Estimate and plot the difference you obtain by repeating this exercise with different values of N (the plot will be N on the $x$ axis and I on the $y$–axis). The result should look like Figure 38 ($\mathrm{I} \sim 1.45$).

*Note: we should generate two random numbers per each point: one along the x–axis and one along the y–axis.*

It can be shown (or proved experimentally, see the bottom panel of Figure 38) that the error scales with N as $\mathrm{N}^{-1/2}$: the accuracy improves the more random samples we take. This scaling is not particularly good. Even the trapezoidal rule did better. In fact, the error of the trapezoidal rule goes as $h^2$, $h = (b-a)/\mathrm{N}$ being the height of the trapezoids. Hence the error in the trapezoidal rule goes as $\mathrm{N}^{-2}$, much faster than with the Monte Carlo algorithm we just described. In the following, we see how we can improve it.

### 9.3.1 The mean value method

We want to evaluate the integral

$$\mathrm{I} = \int_a^b f(x)\,\mathrm{d}x \tag{120}$$

The average value of $f(x)$ from $a$ to $b$ is

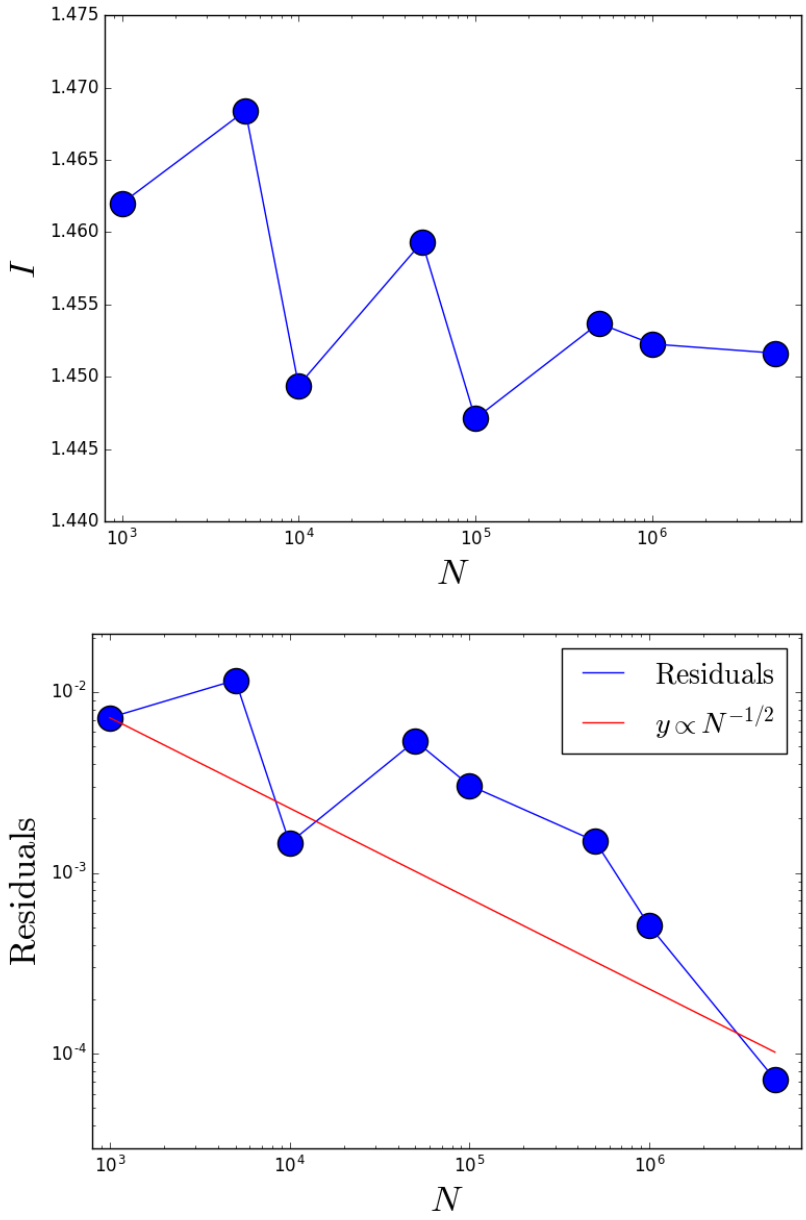


*Figure 38: Top panel: Result of the exercise on the simplest Monte Carlo integration. Bottom panel: Residuals (in blue), calculated as* $|\mathrm{I(N)} - \mathrm{I_{acc}}|/\mathrm{I_{acc}}$*, where* $\mathrm{I_{acc}} = 1.45152558$ *is a particularly good solution of the integral you can get with the mean value method (see Section 9.3.1). The red line is the scaling* $y \propto \mathrm{N}^{-1/2}$.

$$\langle f \rangle = \frac{\int_a^b f(x)\,\mathrm{d}x}{\int_a^b \mathrm{d}x} = \frac{1}{b-a}\int_a^b f(x)\,\mathrm{d}x \tag{121}$$

Thus, we can rewrite equation 120 as

$$\mathrm{I} = (b-a)\langle f \rangle \tag{122}$$

If we estimate $\langle f \rangle$ with sufficient accuracy, then we can estimate I. A simple way to estimate $\langle f \rangle$ is to measure $f(x)$ at N points $x_1$, $x_2$, ..., $x_\mathrm{N}$ chosen uniformly at random between $a$ and $b$ and calculate $\langle f \rangle = \mathrm{N}^{-1}\sum_{i=1}^{\mathrm{N}} f(x_i)$.

In this way, we get the fundamental formula for the **mean value method**, which is

$$\mathrm{I} = \frac{(b-a)}{\mathrm{N}}\sum_{i=1}^{\mathrm{N}} f(x_i) \tag{123}$$

It can be shown that even with this method the error goes as $\mathrm{N}^{-1/2}$, but, for a given N, the error is smaller than in the previous method (equation 119).

**EXERCISE:**

Write a python script to perform the integration with the mean value method (equation 123) of the integral in equation 118. Calculate the integral with $\mathrm{N} = 10^3$, $5\times10^3$, $10^4$, $5\times10^4$, $10^5$, $5\times10^5$, $10^6$, $5\times10^6$. Estimate and plot the difference you obtain by repeating this exercise with different values of N (the plot will be N on the $x$ axis and I on the $y$–axis). The results should look like Figure 39 ($\mathrm{I} \sim 1.452$).

**EXERCISE:**

Use the scripts you have produced to perform Monte Carlo integration with mean value to integrate the mass of a stellar system according to the singular isothermal sphere and to the NFW density profiles (as you already did for the trapezoidal rule).

### 9.3.2 Integrals in many dimensions

The mean value method can be used to calculate multi-dimensional integrals quite efficiently, much faster than other methods. The integral of a function $f(x)$ over a volume V in a high-dimensional space is given by the generalization of equation 123:

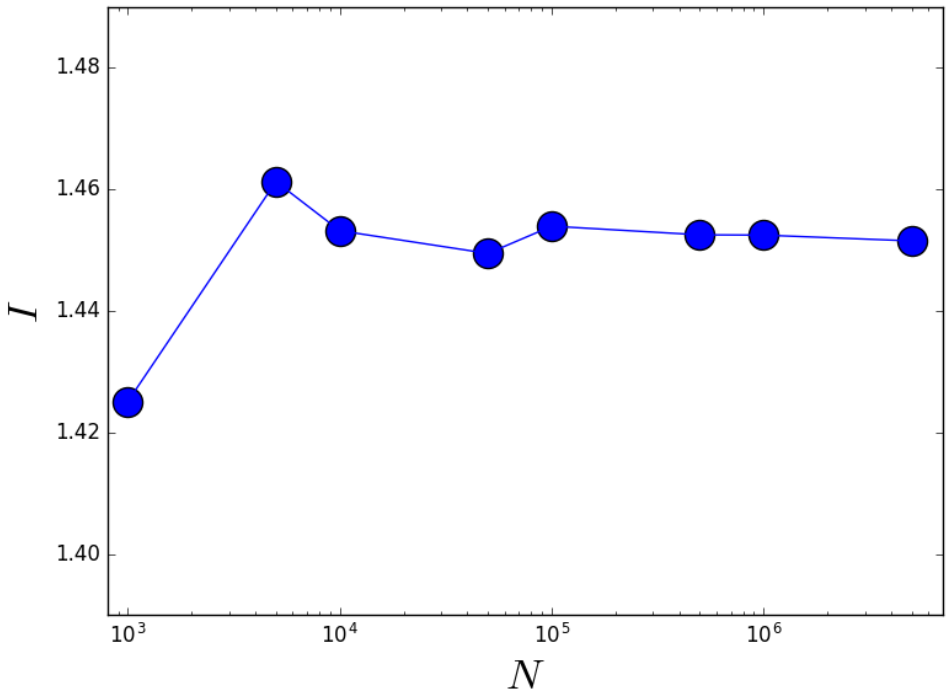


*Figure 39: Result of the exercise on the mean value method. If you compare this plot with the result of the exercise on the simplest Monte Carlo integration method, you see that the mean value method converges much better and faster than the oversimplified method presented in equation 119.*

$$\mathrm{I} \simeq \frac{\mathrm{V}}{\mathrm{N}} \sum_{i=1}^{\mathrm{N}} f(\mathbf{r}_i) \tag{124}$$

where the points $\mathbf{r}_i$ are picked uniformly at random from the volume V.

### 9.3.3 Importance sampling

The Monte Carlo methods presented with equations 123 and 124 are very efficient but perform rather poorly if the function to integrate is particularly pathological, for example if it contains a divergence. For example

$$\mathrm{I} = \int_0^1 \frac{x^{-1/2}}{e^x + 1}\,\mathrm{d}x \tag{125}$$

This equation (which by the way is used in the theory of Fermi gases) diverges at $x = 0$. The integral is finite in value but if you try to solve it with the mean value method you run into troubles because the bins around 0 give a very large contribution to the sum.

Then, you want to find a new Monte Carlo method able to understand where the function becomes pathological and to "get rid" of the pathology. The basic idea is that we use a similar algorithm to the mean value method but we substitute the uniformly drawn N random points $x_i$ with non-uniformly drawn random points. We proceed in the following way. For any general function $g(x)$ we can define a weighted average over the interval from $a$ to $b$ as

$$\langle g \rangle_w = \frac{\int_a^b w(x)\, g(x)\,\mathrm{d}x}{\int_a^b w(x)\,\mathrm{d}x} \tag{126}$$

where $w(x)$ is a generic function we choose (the weighting function).

Setting $g(x) = f(x)/w(x)$, the above equation becomes

$$\left\langle \frac{f(x)}{w(x)} \right\rangle_w = \frac{\int_a^b f(x)\,\mathrm{d}x}{\int_a^b w(x)\,\mathrm{d}x} \tag{127}$$

Let's consider again the generic integral of equation 120. From equation 120, we can re-write equation 128 as

$$\left\langle \frac{f(x)}{w(x)} \right\rangle_w = \frac{\mathrm{I}}{\int_a^b w(x)\,\mathrm{d}x} \tag{128}$$

Thus

$$\mathrm{I} = \left\langle \frac{f(x)}{w(x)} \right\rangle_w \int_a^b w(x)\,\mathrm{d}x \tag{129}$$

Equation 129 is analogous to the mean value method (equation 123) but contains a weighted mean instead of a simple uniform average.

How can we do a weighted mean? Let's define a probability density function as

$$p(x) = \frac{w(x)}{\int_a^b w(x)\,\mathrm{d}x} \tag{130}$$

which is the function $w(x)$ normalized so that its integral is 1. Let us sample N random points $x_i$ non uniformly with this probability density function. That is, the probability of generating a value in the interval between $x$ and $x + \mathrm{d}x$ is $p(x)\,\mathrm{d}x$. Thus, if I draw N random numbers, the average number of samples that fall in this interval is $\mathrm{N}\,p(x)\,\mathrm{d}x$ and so for any function $g(x)$

$$\sum_{i=1}^{\mathrm{N}} g(x_i) \simeq \int_a^b \mathrm{N}\,p(x)\,g(x)\,\mathrm{d}x \tag{131}$$

This approximation gets better if N gets larger.

Substituting equation 130 into equation 126 we get

$$\langle g \rangle_w = \int_a^b p(x)\,g(x)\,\mathrm{d}x \tag{132}$$

Finally, substituting equation 131 into the above equation, we find

$$\langle g \rangle_w = \int_a^b p(x)\,g(x)\,\mathrm{d}x \simeq \frac{1}{\mathrm{N}} \sum_{i=1}^{\mathrm{N}} g(x_i) \tag{133}$$

where the points $i$ are chosen from the distribution 130. Assuming now that $g(x) = f(x)/w(x)$, we can plug equation 133 into equation 129 and we get

$$\mathrm{I} = \frac{1}{\mathrm{N}} \sum_{i=1}^{\mathrm{N}} \frac{f(x_i)}{w(x_i)} \int_a^b w(x)\,\mathrm{d}x \tag{134}$$

which is the **fundamental equation of importance sampling**.

The main difference with respect to the mean value method is that we calculate the sum $\sum_{i=1}^{\mathrm{N}} \frac{f(x_i)}{w(x_i)}$ instead of the sum $\sum_{i=1}^{\mathrm{N}} f(x_i)$. This is important because it allows to choose a function $w(x)$ that gets rid of pathologies in the integrand $f(x)$. If $f(x)$ has a divergence, we can factor that divergence out and hence get a sum that is well behaved. The price we pay is that we have to draw our random numbers $x_i$ from a non-uniform distribution instead of a uniform one, which makes the programming more complex.

For example, in the case we started with (equation 125) a reasonable choice of $w(x)$ is $w(x) = x^{-1/2}$. In this case,

1. $f(x)/w(x) = (e^x + 1)^{-1}$, which has no singularity anymore;

2. the integral of $w(x)$ can be easily calculated as follows:

$$\int_a^b w(x)\,\mathrm{d}x = \int_0^1 x^{-1/2}\,\mathrm{d}x = 2\sqrt{x}\Big|_0^1 = 2; \tag{135}$$

3. we need to draw the random numbers $x_i$ according to the probability distribution function associated to $w(x)$, i.e.

$$p(x) = \frac{x^{-1/2}}{\int_0^1 x^{-1/2}\mathrm{d}x} = \frac{1}{2}\,x^{-1/2}. \tag{136}$$

Hence, with the inverse sampling method

$$y = \int_0^x p(x)\,\mathrm{d}x = \frac{1}{2}\int_0^x x^{-1/2}\,\mathrm{d}x = x^{1/2}$$

$$\rightarrow x = y^2 \tag{137}$$

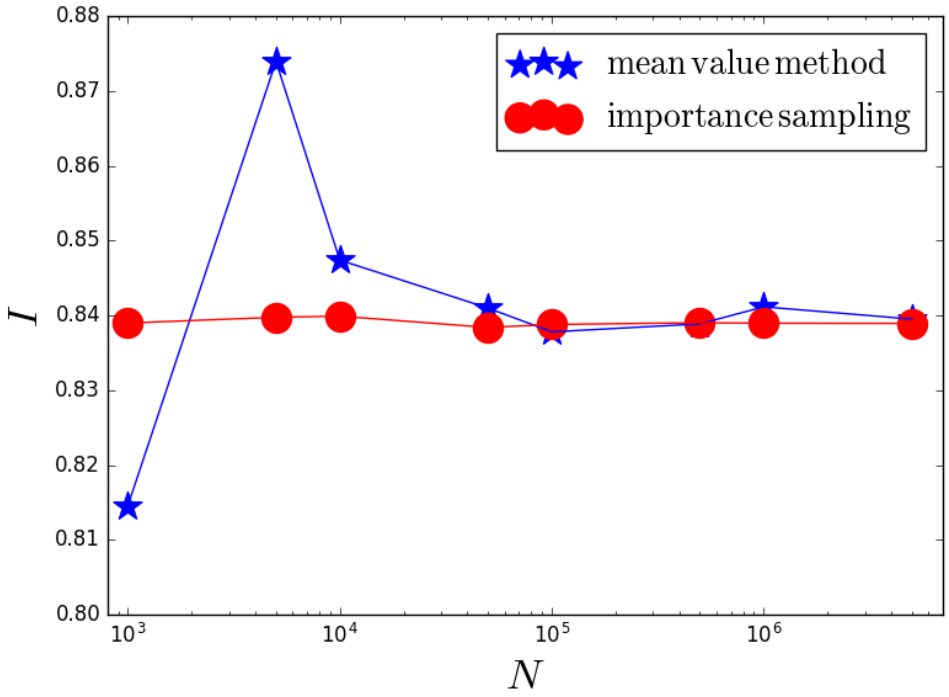


*Figure 40: Result of the exercise on the importance sampling technique.*

**EXERCISE:**

Write a python script to perform the integration in equation 125 with the importance sampling (I $\sim$ 0.839). Calculate the integral with N = $10^3$, $5 \times 10^3$, $10^4$, $5 \times 10^4$, $10^5$, $5 \times 10^5$, $10^6$, $5 \times 10^6$. Evaluate the same integration, for the same values of N, with your script that makes use of the mean value method. Compare the results. The result should look like Figure 40. The importance sampling performs very well already for a small value of N.

## 9.4 *Built-in functions to integrate in python*

As we have seen in the lookback time example (examples/python/loockback.py, Chapter 2), the package **scipy.integrate** contains options to integrate a function[6]. Let's discuss this in more detail.

**1. Methods for integrating functions given fixed samples.**

This is equivalent to the methods we have discussed in this Chapter. The python function takes as argument an array of points $f(x_i)$ (i.e. a fixed sample) and the corresponding array $x_i$ over which it performs the integration. The main functions in this family are

**trapz** Use trapezoidal rule to compute integral from samples.

**cumtrapz** Use trapezoidal rule to cumulatively compute integral.

**simps** Use Simpson's rule (we did not have time to discuss it during these lectures but you can find it on Mark Newman's book) to compute integral from samples.

**romb** Use Romberg Integration (we did not have time to discuss it during these lectures but you can find it on Mark Newman's book) to compute integral from $(2**k + 1)$ evenly-spaced samples.

Here below (and in the examples/integral/trapz.py), you can find an example of how to use scipy.integrate.trapz (a similar procedure applies to the other functions):

[6]The package scipy does not come with the default python. You must install it separately.

```
#how to use the trapz function of scipy.integrate
#applied to the NFW profile
#it calculates equation 102 of the lecture notes
#examples/integral/trapz.py


from scipy.integrate import trapz
import numpy as np

G=6.667e-8 #gravity constant in cgs
pc=3.086e18 # 1 pc in cgs
msun=1.989e33 #solar mass in cgs

rs=10.*1e3*pc #10 kpc in cgs
rmax=100.*rs
rho0=1e8*msun/(1e3*pc)**3 #1e8 Msun/kpc^3 in cgs

def NFW(r):
    x=r/rs
    rho=rho0/((1.+x)**2)
    mass=4.*np.pi*rs*r*rho
    return mass


b=rmax
a=0.0
intervallo=(b-a)
N=int(1e6)
h=intervallo/N

trapzx=[a]
trapzy=[NFW(a)]
for i in range(1,N-1,1):
    trapzx.append(a+i*h)
    trapzy.append(NFW(a+i*h))
trapzx.append(b)
trapzy.append(NFW(b))

II=trapz(trapzy,trapzx)
print("with scipy.integrate.trapz I= ",II/msun," Msun")
```

**2. Methods for integrating functions given function object.**

In this case, python takes the analytic form of the function you want to integrate as an argument, plus the integration range, and decides in which points $x_i$ to evaluate the function. This is exactly what we have done for the lookback time in the example examples/python/lookback.py of Chapter 2. The main functions in this family are

**quad** General purpose integration (uses a technique from the Fortran library QUADPACK: contains different algorithms to solve integrals).

**dblquad** General purpose integration in two dimensions (two variables).

**tplquad** General purpose integration in three dimensions (three variables).

**fixed_quad** Integrate func(x) using Gaussian quadrature of order n (we did not have time to discuss this method during these lectures but you can find it on Mark Newman's book).

**quadrature** Integrate with given tolerance using Gaussian quadrature

**romberg** Integrate func using Romberg integration (we did not have time to discuss this method during these lectures but you can find it on Mark Newman's book).

Let's take a better look at examples/integral/lookback.py (this is a new version of examples/python/lookback.py) to see how scipy.integrate.quad works (the other functions work in a similar way).

The basic commands are:

```
#examples/python/lookback.py
import scipy.integrate as integrate
```

[...]

```
ltime=integrate.quad(integrand, 0.0, z, epsrel=1.e-13)
```

The first argument of quad is the function that you want to integrate (in the example, **integrand**). The second and third argument are the two extremes of the interval over which you want to integrate (in the example, 0.0 and z). Then, there are other possible optional arguments. The only one shown in the example is very useful: **epsrel=** indicates the relative error tolerance (the default is 1.49e-8, which is already very good).

The function that you want to integrate (in the example, integrand) can be expressed in several ways.

You can define it with define:

```
OmegaM=0.2726   #omega matter, parameter from cosmology
OmegaL= 0.7274  #omega lambda, parameter from cosmology


def integrand(x):
    r=1./((1.+x)*(OmegaM*(1.+x)**3.+OmegaL)**0.5)
    return r
```

Or you can use the compact definition form:

```
integrand = lambda x: 1./((1.+x)*(OmegaM*(1.+x)**3.+OmegaL)**0.5)
```

where “lambda x:” specifies what is the variable.

# 10 INTEGRATION OF ORDINARY DIFFERENTIAL EQUATIONS (ODES)

This chapter is based on personal notes and on *Computational Physics* by Mark Newman `http://www-personal.umich.edu/~mejn/cp/`.

Differential equations are ubiquitous in astrophysics. For example, let's consider i) the equations of motion of a particle (e.g. a star) subject to Newton's gravity force, or ii) the equation of hydrostatic equilibrium of a stellar interior. Thus, it is particularly important to know how to solve them and we will treat this topic with great detail.

## 10.1 *Euler scheme*

The general form of a first-order one-variable ordinary differential equation is

$$\frac{\mathrm{d}x}{\mathrm{d}t} = f(x,t) \tag{138}$$

For a concrete example, let's assume that $t$ is the time and $x$ the position.

To integrate equation 138 numerically, the simplest way to proceed is with a Taylor expansion:

$$x(t+h) = x(t) + \frac{\mathrm{d}x}{\mathrm{d}t}\,h + \frac{1}{2}\,\frac{\mathrm{d}^2x}{\mathrm{d}t^2}\,h^2 + \ldots \tag{139}$$

which we can rewrite in the form

$$x(t+h) = x(t) + h\,f(x,t) + \mathcal{O}(h^2) \tag{140}$$

If we neglect terms of order higher than $h$, the equation becomes

$$x(t+h) = x(t) + h\,f(x,t) \tag{141}$$

The above equation tells us that if we know the value of $x$ at time $t$, then we can estimate the value of $x$ at time $t+h$. This approximation is particularly good if $h$ is small. Equation 141 defines the **Euler's method**, after its inventor, Leonhard Euler. To describe the evolution of $x$ over a time interval between $t = a$ and $t = b$, with $b - a \gg h$, we must repeat equation 141 for N time-steps, with $\mathrm{N} = (b-a)/h$.

By looking at equation 140, we immediately realize that the Euler method is a **first-order method**: errors scale as $h^2$. While we can improve the results by reducing $h$, we cannot make $h$ too small, otherwise the calculation becomes too slow. There are other methods with intrinsic higher accuracy that require less computational power.

## 10.2 *Second-order Runge-Kutta scheme*

The Euler method is also called first-order Runge-Kutta scheme, because it represents the first-order approximation version of the Runge-Kutta family. The second-order Runge-Kutta scheme, or **midpoint** method, is very simple and significantly more accurate than the Euler method.

The midpoint method consists in evaluating the slope $\mathrm{d}x/\mathrm{d}t$ of $x$ not at the end of the interval, but at the midpoint of the interval $h$. In mathematical terms, this corresponds to performing a Taylor expansion around $t + 1/2\,h$:

$$x(t+h) = x\left(t + \frac{1}{2}\,h\right) + \frac{1}{2}h\left(\frac{\mathrm{d}x}{\mathrm{d}t}\right)_{t+\frac{1}{2}h} + \frac{1}{8}h^2\left(\frac{\mathrm{d}^2x}{\mathrm{d}t^2}\right)_{t+\frac{1}{2}h} + \mathcal{O}(h^3)$$

$$x(t) = x\left(t + \frac{1}{2}\,h\right) - \frac{1}{2}h\left(\frac{\mathrm{d}x}{\mathrm{d}t}\right)_{t+\frac{1}{2}h} + \frac{1}{8}h^2\left(\frac{\mathrm{d}^2x}{\mathrm{d}t^2}\right)_{t+\frac{1}{2}h} + \mathcal{O}(h^3) \qquad (142)$$

Subtracting the second expression from the first, we get

$$x(t+h) = x(t) + h\left(\frac{\mathrm{d}x}{\mathrm{d}t}\right)_{t+\frac{1}{2}h} + \mathcal{O}(h^3)$$

$$= x(t) + h\,f\left(x\left(t + \frac{1}{2}h\right), t + \frac{1}{2}h\right) + \mathcal{O}(h^3) \qquad (143)$$

The error scales with $h^3$, i.e. the midpoint scheme is a second-order scheme.

The problem here is that equation 143 requires the knowledge of $x$ at the point $t + \frac{h}{2}$, which we do not have. This is a so-called **implicit scheme**: a scheme in which information is required on some quantities we still do not know. We have to calculate them in an approximate way with a trick.

We get around this problem by approximating $x\left(t + \frac{h}{2}\right)$ with Euler's method $\left[x\left(t + \frac{h}{2}\right) = x(t) + \frac{h}{2}\,f(x(t), t)\right]$ and then substituting it into the above equation. The complete midpoint scheme for a single step is thus:

$$k_1 = \frac{h}{2}\,f(x(t), t)$$

$$k_2 = h\,f\left(x(t) + k_1,\ t + \frac{h}{2}\right)$$

$$x(t+h) = x(t) + k_2 \qquad (144)$$

## 10.3 *Fourth-order Runge-Kutta scheme*

We can use the same formalism to improve the accuracy even more, cancelling out terms that scale with $h^3$ and $h^4$. When introducing these terms, the scheme becomes more complicated to implement. Usually, the fourth-order version of Runge-Kutta (with errors scaling as $h^5$) is considered the best match between

accuracy and not-too-complicated programming. The equations look like:

$$k_1 = \frac{1}{2} h f(x, t)$$

$$k_2 = \frac{1}{2} h f\left(x + k_1,\ t + \frac{1}{2}h\right)$$

$$k_3 = h f\left(x + k_2,\ t + \frac{1}{2}h\right)$$

$$k_4 = h f(x + k_3,\ t + h)$$

$$x(t + h) = x(t) + \frac{1}{6}(2\,k_1 + 4\,k_2 + 2\,k_3 + k_4) \tag{145}$$

We do not derive equation 145 because the algebra is quite tedious, but you can look at the "Numerical Recipes" book by Press et al., if you want to know more.

**EXERCISE:**

Write a python script to implement the Euler's method, the midpoint method and the fourth-order Runge-Kutta method. Use this script to integrate the following differential equation:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = -x^3 + \sin t \tag{146}$$

Compare the results. For a choice of initial time $t_0 = 0.0$, final time $t_{\mathrm{fin}} = 100$, initial position $x(t_0) = 0.0$ and step-size $h = 0.4$, you should obtain something similar to Figure 41.

## 10.4 *Systems of ordinary differential equations*

Solving systems of differential equations can be easily done with the same approach as we have seen in the previous sections, provided that the derivatives are with respect to a single variable (systems of ordinary differential equations). For example

$$\frac{\mathrm{d}x}{\mathrm{d}t} = f_1(x, y, t)$$

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f_2(x, y, t), \tag{147}$$

where we have only the derivatives with respect to $t$. Note that in our example, $f_1(x, y, t)$ and $f_2(x, y, t)$, i.e. $f_1$ depends not only on $x$ and $t$ but also on $y$ and $f_2$ depends not only on $y$ and $t$ but also on $x$. In this case, we only have to make sure that the two equations are integrated **in the same timestep**, to avoid mismatches between the value of $x(y, t)$ and the value of $y(x, t)$ at a given time $t$. We will see two examples later in this chapter: the evolution of semi-major

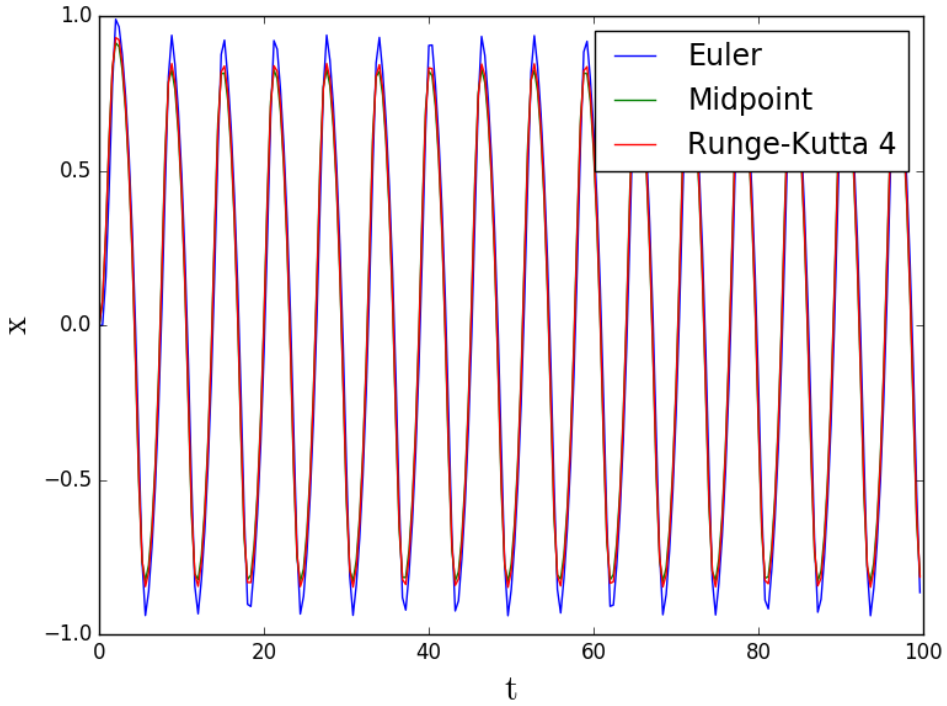


*Figure 41: Integration of the differential equation 146 for initial time $t_0 = 0.0$, final time $t_{\text{fin}} = 100$, initial position $x(t_0) = 0.0$ and step-size $h = 0.4$.*

axis $a$ and eccentricity $e$ of a binary system affected by gravitational wave emission, and the astrophysical N–body problem.

In contrast, systems like

$$\frac{\partial x}{\partial t} + \frac{\partial x}{\partial s} = f_1(x, y, t, s)$$
$$\frac{\partial y}{\partial t} + \frac{\partial y}{\partial s} = f_2(x, y, t, s), \tag{148}$$

are systems of partial differential equations, in which $x(t, s)$ and $y(t, s)$. These can be solved only with methods for partial differential equations (see next chapter).

## 10.5 *Second-order and higher-order ordinary differential equations*

Solving second-order (or higher-order) differential equations with just one variable is trivial once we know how to solve first-order differential equations. Imagine we have a second-order equation of the (general) kind:

$$\frac{\mathrm{d}^2 x}{\mathrm{d}t^2} = f\left(x, \frac{\mathrm{d}x}{\mathrm{d}t}, t\right) \tag{149}$$

This equation can be rewritten as a system of two first-order differential equations:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = y$$
$$\frac{\mathrm{d}y}{\mathrm{d}t} = f\,(x, y, t)\,. \tag{150}$$

We now can solve this system using the algorithms we learned for first-order differential equations. To solve higher order differential equations we simply repeat this trick till we have a system of first-order equations only.

The classical example for astrophysicists is the equation of motion of a star in a binary system (see the next section for a generalization):

$$\frac{\mathrm{d}^2 \mathbf{x}_i}{\mathrm{d}t^2} = -\mathrm{G}\, m_j \frac{\mathbf{x}_i - \mathbf{x}_j}{\left|\mathbf{x}_i - \mathbf{x}_j\right|^3}, \tag{151}$$

where $\mathbf{x}_i$ and $\mathbf{x}_j$ are the position vectors of the two members of the binary systems (star $i$ and star $j$), $m_j$ is the mass of star $j$ and $\mathrm{G} \sim 6.674 \times 10^{-8}$ $\mathrm{cm}^3\ \mathrm{g}^{-1}\ \mathrm{s}^{-2}$ is the gravity constant. Clearly, we have a second, symmetric equation for particle $j$. The above equation is second-order but can be rewritten as

the following system:

$$\frac{\mathrm{d}\mathbf{x}_i}{\mathrm{d}t} = \mathbf{v}_i$$
$$\frac{\mathrm{d}\mathbf{v}_i}{\mathrm{d}t} = -\mathrm{G}\, m_j \frac{\mathbf{x}_i - \mathbf{x}_j}{\left|\mathbf{x}_i - \mathbf{x}_j\right|^3}. \tag{152}$$

## 10.6 *The astrophysical N-body problem*

Astrophysicists need to solve ordinary differential equations for one of the most fundamental problems of Astrophysics: the so-called astrophysical N–body problem, i.e. the integration of the equations of motion for N bodies subject to Newton's gravity force (1687):

$$\frac{\mathrm{d}^2\mathbf{x}_i}{\mathrm{d}t^2} = -\mathrm{G} \sum_{j=1,\, j\neq i}^{\mathrm{N}} m_j \frac{\mathbf{x}_i - \mathbf{x}_j}{\left|\mathbf{x}_i - \mathbf{x}_j\right|^3}, \tag{153}$$

which is a ordinary differential equation of the second order with respect to time $t$. In the above equation, $\mathbf{x}_i$ is the position vector of the $i$th particle in a system of N particles (these can be stars or gas particles, or even dark matter particles), $m_j$ is the mass of the $j$th particle and $\mathrm{G} \sim 6.667 \times 10^{-8}$ cm$^3$ g$^{-1}$ s$^{-2}$ is the gravity constant.

The above equation can be split into a system of two first-order differential equations:

$$\frac{\mathrm{d}\mathbf{x}_i}{\mathrm{d}t} = \mathbf{v}_i$$
$$\frac{\mathrm{d}\mathbf{v}_i}{\mathrm{d}t} = -\mathrm{G} \sum_{j=1,\, j\neq i}^{\mathrm{N}} m_j \frac{\mathbf{x}_i - \mathbf{x}_j}{\left|\mathbf{x}_i - \mathbf{x}_j\right|^3}, \tag{154}$$

where $\mathbf{v}_i$ is apparently the velocity vector of the $i$th particle.

Clearly, this is a vector equation. So, to solve it, we have to solve per every

single component:

$$\frac{\mathrm{d}v_{x_i}}{\mathrm{d}t} = -\mathrm{G}\sum_{j=1,\, j\neq i}^{\mathrm{N}} m_j \frac{x_i - x_j}{\left|\mathbf{x}_i - \mathbf{x}_j\right|^3},$$

$$\frac{\mathrm{d}v_{y_i}}{\mathrm{d}t} = -\mathrm{G}\sum_{j=1,\, j\neq i}^{\mathrm{N}} m_j \frac{y_i - y_j}{\left|\mathbf{x}_i - \mathbf{x}_j\right|^3},$$

$$\frac{\mathrm{d}v_{z_i}}{\mathrm{d}t} = -\mathrm{G}\sum_{j=1,\, j\neq i}^{\mathrm{N}} m_j \frac{z_i - z_j}{\left|\mathbf{x}_i - \mathbf{x}_j\right|^3},$$

$$\frac{\mathrm{d}x_i}{\mathrm{d}t} = v_{x_i},$$

$$\frac{\mathrm{d}y_i}{\mathrm{d}t} = v_{y_i},$$

$$\frac{\mathrm{d}z_i}{\mathrm{d}t} = v_{z_i} \tag{155}$$

where $\mathbf{x}_i = (x_i,\, y_i,\, z_i)$ and $\mathbf{v}_i = (v_{x_i},\, v_{y_i},\, v_{z_i})$ in Cartesian coordinates.

Moreover, we have to solve these equations for each particle (gas particle or star) in the astrophysical system. Hence, we will have to **solve** $6 \times \mathrm{N}$ **simultaneous equations** to integrate the motion of the entire N-body system.

Equation 153 can be solved analytically for $\mathrm{N} = 2$ (Bernoulli solution, 1710). In 1885, a challenge was proposed, to be answered before January 21st 1889, in honour of the 60th birthday of King Oscar II of Sweden and Norway (`http://www.mittag-leffler.se/library/henri-poincare`):

*Given a system of arbitrarily many mass points that attract each according to Newton's law, under the assumption that no two points ever collide, try to find a representation of the coordinates of each point as a series in a variable that is some known function of time and for all of whose values the series converges uniformly.*

Nobody found the solution, although many participated (including Henry Poincaré). In 1991, the mathematician Qiudong Wang found the first convergent power series solution for a generic number of bodies. Thus, equation 153 can be considered as an equation that admits analytic solution (at least in the sense of a convergent series). However, the solution by Q. Wang is too difficult to implement and slow to converge. Thus, everybody solves equation 153 numerically for $\mathrm{N} \geq 3$.

## 10.7 *Astrophysical N-body problem with Euler's method*

To solve equation 153, we can use the Runge-Kutta methods (Euler's, midpoint and Runge-Kutta fourth orders included), but we can use also other methods (e.g. leapfrog scheme, Hermite scheme). The simplest example is (of course) the Euler's method. The Euler integration of the gravitational N–body problem

looks like this:

$$\mathbf{a}_i(t) = -\mathrm{G} \sum_{j=1,\, j\neq i}^{\mathrm{N}} m_j \frac{\mathbf{x}_i(t) - \mathbf{x}_j(t)}{\left|\mathbf{x}_i(t) - \mathbf{x}_j(t)\right|^3},$$
$$\mathbf{x}_i(t+h) = \mathbf{x}_i(t) + \mathbf{v}_i(t)\, h,$$
$$\mathbf{v}_i(t+h) = \mathbf{v}_i(t) + \mathbf{a}_i(t)\, h, \tag{156}$$

where $\mathbf{a}_i(t) = \frac{\mathrm{d}\mathbf{v}_i}{\mathrm{d}t}$ is the gravitational acceleration of the $i$th particle.

## 10.8 *Astrophysical N-body problem with midpoint scheme*

The general formulation of the midpoint scheme applied to an astrophysical N−body problem is:

$$k_{1,x} = \frac{h}{2} \frac{\mathrm{d}x_i(t)}{\mathrm{d}t} \tag{157}$$

$$k_{1,v} = \frac{h}{2} \frac{\mathrm{d}v_i(x_i(t), t)}{\mathrm{d}t} \tag{158}$$

$$k_{2,x} = h \frac{\mathrm{d}(x_i(t) + k_{1,x}, t + h/2)}{\mathrm{d}t} \tag{159}$$

$$k_{2,v} = h \frac{\mathrm{d}(v_i(t) + k_{1,v}, t + h/2)}{\mathrm{d}t} \tag{160}$$

$$x_i(t+h) = x_i(t) + k_{2,x} \tag{161}$$

$$v_i(t+h) = v_i(t) + k_{2,v} \tag{162}$$

Equations 157 and 158 can be rewritten explicitly as

$$k_{1,x} = \frac{h}{2} v_i(t) \tag{163}$$

$$k_{1,v} = \frac{h}{2} a_i(t) \tag{164}$$

Equations 159 and 160 require a little bit more of work. Let's first rewrite equation 159 as:

$$k_{2,x} = h \frac{\mathrm{d}}{\mathrm{d}t}\left(x_i(t) + \frac{h}{2} v_i(t)\right) = h\left[v_i(t) + \frac{h}{2} a_i(t)\right] \tag{165}$$

In contrast, the smartest thing to do for equation 160 is to consider that

$$k_{2,v} = h \frac{\mathrm{d}^2 x}{\mathrm{d}t^2}\Big|_{t+\frac{h}{2}} = h\, a_i(x_i(t + h/2), t + h/2) \tag{166}$$

but the value of the acceleration $a_i$ calculated at time $t + h/2$ can be easily obtained, because in the midpoint scheme we have already calculated the value of $x_i(t + h/2) = x_i(t) + h/2\, v_i(t)$ thanks to the Euler method. Hence, the equation 166 becomes:

$$k_{2,v} = h\, a_i(x_i(t) + h/2\, v_i(t)) \tag{167}$$

Finally, we can rewrite equations 165 and 167 using the definitions of $k_{1,x} = h/2\, v_i(t)$ and $k_{1,v} = h/2\, a_i(t)$:

$$k_{2,x} = h\, [v_i(t) + k_{1,v}] \tag{168}$$

$$k_{2,v} = h\, a_i(x_i(t) + k_{1,x}) \tag{169}$$

So, the final compact formula for the midpoint scheme applied to the astrophysical N-body problem is:

$$k_{1,x} = \frac{h}{2}\, v_i(t)$$

$$k_{1,v} = \frac{h}{2}\, a_i(t)$$

$$k_{2,x} = h\, [v_i(t) + k_{1,v}]$$

$$k_{2,v} = h\, a_i(x_i(t) + k_{1,x})$$

$$x_i(t + h) = x_i(t) + k_{2,x}$$

$$v_i(t + h) = v_i(t) + k_{2,v} \tag{170}$$

In a less elegant but maybe more understandable way, from a practical

point of view, the equations 170 can be rewritten also as:

$$
\begin{aligned}
a_i(t) &= -\mathrm{G} \sum_{j=1, j\neq i}^{n} \frac{m_j \left[x_i(t) - x_j(t)\right]}{\left|x_i(t) - x_j(t)\right|^3} \\
k_{1,x} &= \frac{h}{2} v_i(t) \\
k_{1,v} &= \frac{h}{2} a_i(t) \\
\\
x_i(t + h/2) &= x_i(t) + k_{1,x} \\
a_i(t + h/2) &= -\mathrm{G} \sum_{j=1, j\neq i}^{n} \frac{m_j \left[x_i(t + h/2) - x_j(t + h/2)\right]}{\left|x_i(t + h/2) - x_j(t + h/2)\right|^3} \\
k_{2,x} &= h \left[v_i(t) + k_{1,v}\right] \\
k_{2,v} &= h\, a_i(t + h/2) \\
\\
x_i(t + h) &= x_i(t) + k_{2,x} \\
v_i(t + h) &= v_i(t) + k_{2,v}
\end{aligned}
\tag{171}
$$

Here below, we discuss the leapfrog scheme and the fourth-order Hermite scheme, which are very popular among astrophysicists studying systems of N bodies.

**EXERCISE:**

Write a new script to implement Euler's method to evolve a system of two points in two dimensions ($xy$ plane), subject to gravity forces, with the following initial conditions. Initial positions of particles 1 and 2 (in the plane $xy$): $\mathbf{x} = (1.0, -1.0)$, $\mathbf{y} = (1.0, -1.0)$.
Initial velocities of particles 1 and 2 (in the plane $xy$): $\mathbf{v}_x = (-0.5, 0.5)$, $\mathbf{v}_y = (0.0, 0.0)$.
Let us assume that the masses are $m_1 = m_2 = 1$, and the gravity constant in our units is G = 1.
Let us assume $t_0 = 0$, $t_{\mathrm{fin}} = 300$ and $h = 0.01$. The result should look like the blue line in Figure 42.

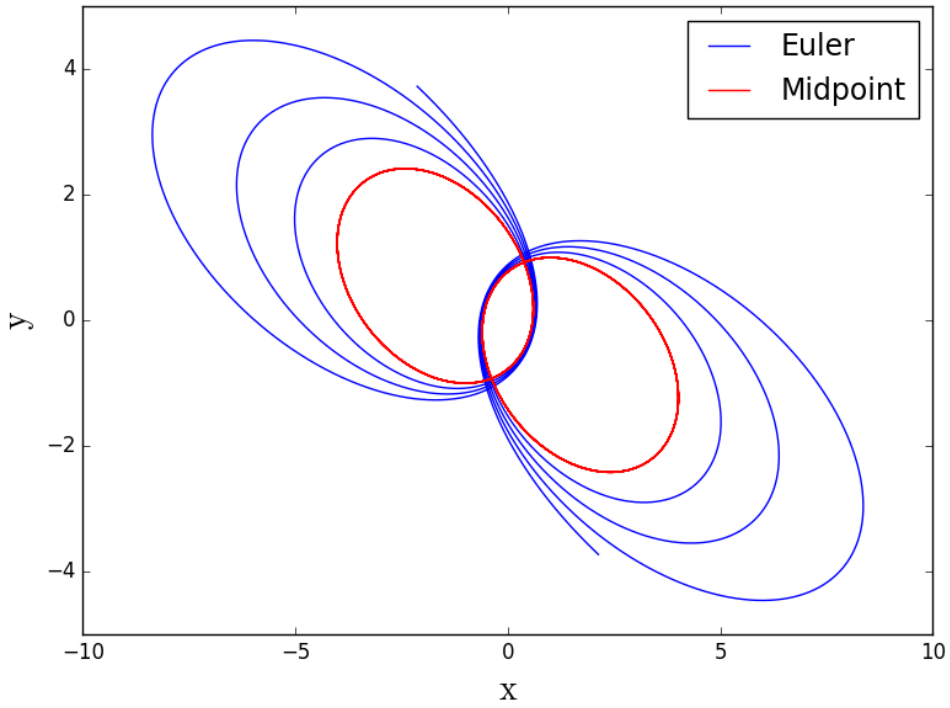


*Figure 42: Integration of a binary star with the Euler method, with the Midpoint scheme and with the Runge-Kutta 4th order scheme, for initial time $t_0 = 0.0$, final time $t_{\text{fin}} = 300$ and step-size $h = 0.01$. Blue line: orbit of the two particles integrated with the Euler's method. Red line: orbit of the two particles integrated with Midpoint or Runge-Kutta 4th order method.*

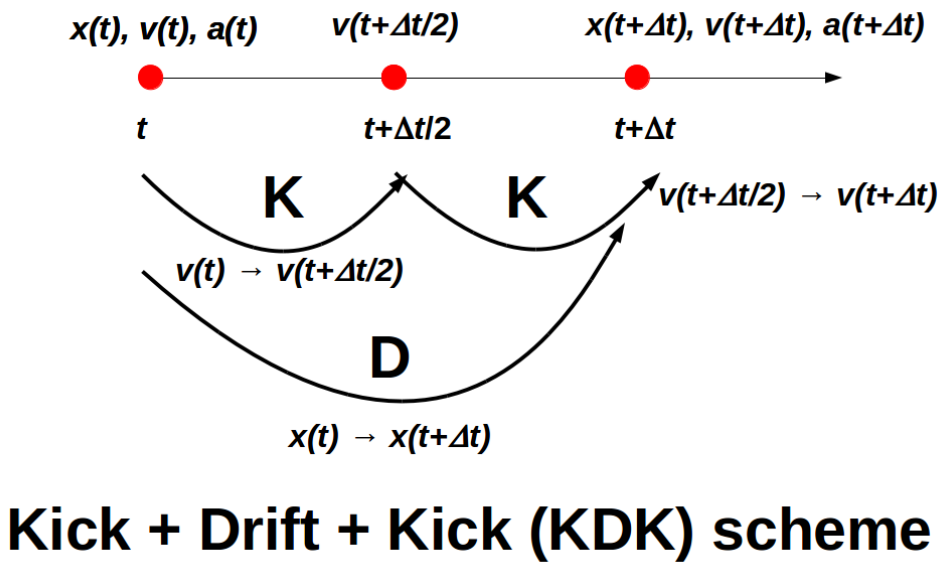


*Figure 43: Visualization of the leapfrog scheme in its kick-drift-kick (KDK) declination.*

**EXERCISE:**

Write a new script to implement the Midpoint method and/or the Runge-Kutta 4th order method to evolve a system of two points in two dimensions ($xy$ plane) described in the pevious exercise.
Let us assume $t_0 = 0$, $t_{\rm fin} = 300$ and $h = 0.01$. The result should look like the red line in Figure 42 (Midpoint and Runge-Kutta 4th order cannot be distinguished by eye in this case).

## 10.9 *Leapfrog scheme*

The leapfrog scheme is indeed a particular version of the midpoint scheme. Its name comes from the leapfrog play (in Italian: la cavallina). Indeed, the idea beneath the leapfrog is that of evaluating velocity and position by jumping like little happy frogs within a time-step.. It is the same as Euler's method but evaluated in between a time-step (see the cartoon 43).

The most common version of the leapfrog scheme is the kick-drift-kick (KDK), which consists in estimating the velocity in the midpoint of the time-step (kick), then estimating the position at the end of the time-step, based on the velocity in the midpoint (drift), and finally estimating the velocity at the

end of the time-step (kick). Mathematically, it is written as:

$$\mathbf{a}_i(t) = -\mathrm{G}\sum_{j=1,\, j\neq i}^{\mathrm{N}} m_j \frac{\mathbf{x}_i(t) - \mathbf{x}_j(t)}{\left|\mathbf{x}_i(t) - \mathbf{x}_j(t)\right|^3}, \tag{172}$$

$$\mathbf{v}_i\left(t + \frac{h}{2}\right) = \mathbf{v}_i(t) + \frac{h}{2}\mathbf{a}_i(t) \tag{173}$$

$$\mathbf{x}_i(t+h) = \mathbf{x}_i(t) + h\,\mathbf{v}_i\left(t + \frac{h}{2}\right) \tag{174}$$

$$\mathbf{a}_i(t+h) = -\mathrm{G}\sum_{j=1,\, j\neq i}^{\mathrm{N}} m_j \frac{\mathbf{x}_i(t+h) - \mathbf{x}_j(t+h)}{\left|\mathbf{x}_i(t+h) - \mathbf{x}_j(t+h)\right|^3}, \tag{175}$$

$$\mathbf{v}_i(t+h) = \mathbf{v}_i\left(t + \frac{h}{2}\right) + \frac{h}{2}\mathbf{a}_i(t+h) \tag{176}$$

In more compact form (by substituting equation 173 into equations 174 and 176), this becomes:

$$\mathbf{a}_i(t) = -\mathrm{G}\sum_{j=1,\, j\neq i}^{\mathrm{N}} m_j \frac{\mathbf{x}_i(t) - \mathbf{x}_j(t)}{\left|\mathbf{x}_i(t) - \mathbf{x}_j(t)\right|^3},$$

$$\mathbf{x}_i(t+h) = \mathbf{x}_i(t) + h\,\mathbf{v}_i(t) + \frac{h^2}{2}\mathbf{a}_i(t)$$

$$\mathbf{a}_i(t+h) = -\mathrm{G}\sum_{j=1,\, j\neq i}^{\mathrm{N}} m_j \frac{\mathbf{x}_i(t+h) - \mathbf{x}_j(t+h)}{\left|\mathbf{x}_i(t+h) - \mathbf{x}_j(t+h)\right|^3},$$

$$\mathbf{v}_i(t+h) = \mathbf{v}_i(t) + \frac{h}{2}\left[\mathbf{a}_i(t) + \mathbf{a}_i(t+h)\right] \tag{177}$$

The leapfrog scheme is surprisingly accurate for being just a (barely) second-order scheme. There is also an alternative version of drift-kick-drift (DKD) leapfrog scheme, in which position is evaluated at the midpoint $(t+h/2)$, then velocity is advanced to the end and finally position is recalculated to the end of the step. You can try to derive this one by yourself.

Another nice feature of the leapfrog is that (unlike Runge-Kutta) leapfrog is time-reversal symmetric: if we integrate the same problem forward in time and then backward in time with the same timestep $h$ we get exactly the same answer (within computer rounding errors). This has the pleasant consequence that the error on energy conservation does not grow with time.

**Important note: A nice way to estimate how well an integrator of celestial dynamics works is to calculate the conservation of total energy and total angular momentum as a function of time during the integration.**

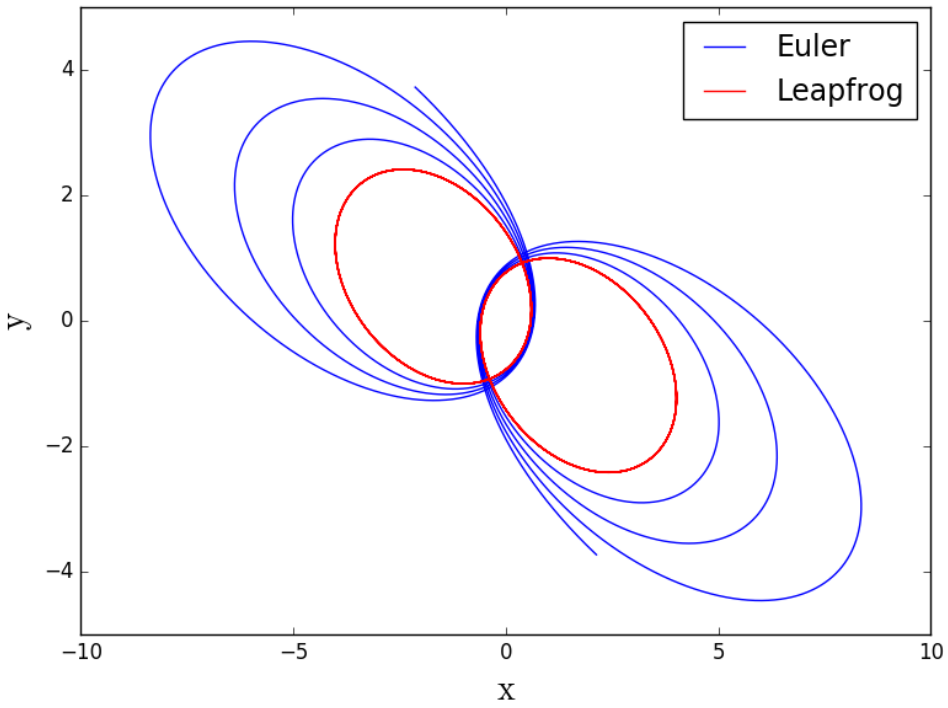


*Figure 44: Integration of a binary star with the leapfrog scheme for initial time $t_0 = 0.0$, final time $t_{\rm fin} = 300$ and step-size $h = 0.01$. Blue line: orbit of the two particles integrated with the Euler's method. Red line: orbit of the two particles integrated with the leapfrog method. The leapfrog is very simple to code and very fast, but it works as good as Midpoint and nearly as good as Runge-Kutta fourth order. In addition, leapfrog is time reversible, while Runge-Kutta is not. These are some of the reasons why leapfrog is so popular among N-body codes.*

**EXERCISE:**

Write a code to implement the leapfrog scheme. Integrate the binary star in the previous exercises with the leapfrog scheme. Compare Euler's method with the leapfrog scheme. Choose $t_0 = 0$, $t_{\rm fin} = 300$ and $h = 0.01$. The result should look like Figure 44. Leapfrog is much better, isn't it?

## 10.10 *Fourth-order predictor-corrector Hermite scheme*

The Hermite scheme can be implemented as a fourth-order or higher than fourth-order scheme. Here, we see the fourth-order version of this scheme. It is called a **predictor-corrector** scheme, because particle positions and velocities are first predicted at third-order level and then corrected to make the accuracy fourth-order.

To derive the Hermite scheme we first need to define the **jerk**, which is the

derivative of acceleration. For Newton's gravity equation, the jerk is

$$\mathbf{j}_i \equiv \frac{\mathrm{d}\mathbf{a}_i}{\mathrm{d}t} = -\mathrm{G} \sum_{j=1,\, j\neq i}^{\mathrm{N}} m_j \left[ \frac{\mathbf{v}_{ij}}{x_{ij}^3} - 3\frac{\left(\mathbf{x}_{ij}\cdot\mathbf{v}_{ij}\right)\mathbf{x}_{ij}}{x_{ij}^5} \right], \tag{178}$$

where $\mathbf{x}_{ij} \equiv \mathbf{x}_i - \mathbf{x}_j$, $\mathbf{v}_{ij} \equiv \mathbf{v}_i - \mathbf{v}_j$, $x_{ij} \equiv \left|\mathbf{x}_i - \mathbf{x}_j\right|$. We then indicate with $\mathbf{j}_i'$ and with $\mathbf{j}_i''$ the first and second derivative of the jerk, respectively (in the N-body slang, $\mathbf{j}_i'$ and $\mathbf{j}_i''$ are also referred to as snap and crackle, respectively).

Let us start from fourth-order Taylor expansions for positions and velocities, third-order for accelerations, second-order for jerks:

$$\mathbf{x}_i(t+h) = \mathbf{x}_i(t) + h\,\mathbf{v}_i(t) + \frac{h^2}{2}\mathbf{a}_i(t) + \frac{h^3}{6}\mathbf{j}_i(t) + \frac{h^4}{24}\mathbf{j}_i'(t) \tag{179}$$

$$\mathbf{v}_i(t+h) = \mathbf{v}_i(t) + h\,\mathbf{a}_i(t) + \frac{h^2}{2}\mathbf{j}_i(t) + \frac{h^3}{6}\mathbf{j}_i'(t) + \frac{h^4}{24}\mathbf{j}_i''(t) \tag{180}$$

$$\mathbf{a}_i(t+h) = \mathbf{a}_i(t) + h\,\mathbf{j}_i(t) + \frac{h^2}{2}\mathbf{j}_i'(t) + \frac{h^3}{6}\mathbf{j}_i''(t) \tag{181}$$

$$\mathbf{j}_i(t+h) = \mathbf{j}_i(t) + h\,\mathbf{j}_i'(t) + \frac{h^2}{2}\mathbf{j}_i''(t) \tag{182}$$

We can use equations 181 and 182 to eliminate the first and second derivative of jerk from equations 179 and 180. With this trick we obtain

$$\mathbf{x}_i(t+h) = \mathbf{x}_i(t) + \frac{h}{2}\left[\mathbf{v}_i(t) + \mathbf{v}_i(t+h)\right] + \frac{h^2}{12}\left[\mathbf{a}_i(t) - \mathbf{a}_i(t+h)\right] + \mathcal{O}(h^5) \tag{183}$$

$$\mathbf{v}_i(t+h) = \mathbf{v}_i(t) + \frac{h}{2}\left[\mathbf{a}_i(t) + \mathbf{a}_i(t+h)\right] + \frac{h^2}{12}\left[\mathbf{j}_i(t) - \mathbf{j}_i(t+h)\right] + \mathcal{O}(h^5) \tag{184}$$

The above equations are fourth order accuracy, with errors scaling as $h^5$, but the terms in $h^3$ and $h^4$ have cancelled out!

However, we have a problem here, because the equations 183 and 184 depend on $\mathbf{v}_i(t+h)$, $\mathbf{a}_i(t+h)$ and $\mathbf{j}_i(t+h)$, that we do not know yet. When a method depends on quantities that we still need to calculate (because they refer to the next step) it is called an **implicit method**, while methods depending only on quantities that we already know (because they refer to current step) are called **explicit** methods. Obviously, we cannot use an implicit method unless we have some trick to **predict** the quantities which refer to the next step (see the similar discussion for the midpoint method).

In the Hermite method, we predict positions and velocities of the next step by using a third-order Taylor expansion (one order less than fourth-order). Thus, the full predictor-corrector Hermite scheme has three steps, as follows.

1. **Predictor step:** we use the third order Taylor expansion to predict $\mathbf{x}_i(t+$

$h$) and $\mathbf{v}_i(t+h)$:

$$\mathbf{xp}_i(t+h) = \mathbf{x}_i(t) + h\,\mathbf{v}_i(t) + \frac{h^2}{2}\mathbf{a}_i(t) + \frac{h^3}{6}\mathbf{j}_i(t) \tag{185}$$

$$\mathbf{vp}_i(t+h) = \mathbf{v}_i(t) + h\,\mathbf{a}_i(t) + \frac{h^2}{2}\mathbf{j}_i(t) \tag{186}$$

Note that in the above equation we have used the $\mathbf{p}$ to indicate that these are not the final positions and velocities at time $t+h$ but just the predicted values $\mathbf{xp}_i(t+h)$ and $\mathbf{vp}_i(t+h)$.

2. **Force evaluation:** we use the above predictions (equations 185 and 186) to evaluate "predicted" acceleration and jerk at time $t+h$, i.e. $\mathbf{ap}_i(t+h)$ and $\mathbf{jp}_i(t+h)$:

$$\mathbf{ap}_i\,(t+h) = -\mathrm{G}\sum_{j=1,\,j\neq i}^{\mathrm{N}} m_j\,\frac{\mathbf{xp}_{ij}\,(t+h)}{xp_{ij}^3}, \tag{187}$$

$$\mathbf{jp}_i\,(t+h) = -\mathrm{G}\sum_{j=1,\,j\neq i}^{\mathrm{N}} m_j\left[\frac{\mathbf{vp}_{ij}}{xp_{ij}^3} - 3\frac{\left(\mathbf{xp}_{ij}\cdot\mathbf{vp}_{ij}\right)\mathbf{xp}_{ij}}{xp_{ij}^5}\right] \tag{188}$$

3. **Corrector step:** we substitute $\mathbf{ap}_i\,(t+h)$ and $\mathbf{jp}_i\,(t+h)$ into equations 183 and 184:

$$\mathbf{x}_i(t+h) = \mathbf{x}_i(t) + \frac{h}{2}\,[\mathbf{v}_i(t) + \mathbf{vp}_i(t+h)] + \frac{h^2}{12}\,[\mathbf{a}_i(t) - \mathbf{ap}_i(t+h)] \tag{189}$$

$$\mathbf{v}_i(t+h) = \mathbf{v}_i(t) + \frac{h}{2}\,[\mathbf{a}_i(t) + \mathbf{ap}_i(t+h)] + \frac{h^2}{12}\left[\mathbf{j}_i(t) - \mathbf{jp}_i(t+h)\right] \tag{190}$$

However, this result is only third-order in positions, because equation 189 contains the term $h\,\mathbf{vp}_i(t+h)$ which is third order.

We can use a smart trick to make it fourth order: we calculate $\mathbf{v}_i(t+h)$ first and then we use the result into $\mathbf{x}_i(t+h)$, as follows:

$$\mathbf{v}_i(t+h) = \mathbf{v}_i(t) + \frac{h}{2}\,[\mathbf{a}_i(t) + \mathbf{ap}_i(t+h)] + \frac{h^2}{12}\left[\mathbf{j}_i(t) - \mathbf{jp}_i(t+h)\right] \tag{191}$$

$$\mathbf{x}_i(t+h) = \mathbf{x}_i(t) + \frac{h}{2}\,[\mathbf{v}_i(t) + \mathbf{v}_i(t+h)] + \frac{h^2}{12}\,[\mathbf{a}_i(t) - \mathbf{ap}_i(t+h)] \tag{192}$$

Equations 191 and 192 are the simplest version of the fourth-order predictor-corrector Hermite scheme.

**EXERCISE:**

Write a code to implement the Hermite scheme. Re-do the previous exercise with the Hermite scheme. Use $t_0 = 0.0$, final time $t_{\rm fin} = 300$ and step-size $h = 0.01$. Compare the result of this exercise with the result of the leapfrog scheme. Figure 45 shows a comparison of Hermite and leapfrog also in terms of total energy conservation. The Hermite scheme performs better than the leapfrog scheme.
Please, don't do this exercise unless you are terribly bored by the simpler ones.
*Suggestion: to calculate energy conservation, use the reference frame of the reduced particle and neglect the kinetic energy of the center of mass. It simplifies your life. In this case, the total energy of the binary is*

$$\mathrm{E} = \frac{1}{2}\frac{m_1\, m_2}{(m_1 + m_2)} v_{ij}^2 - \frac{\mathrm{G}\, m_1\, m_2}{2\, x_{ij}}, \tag{193}$$

*where $m_1$ and $m_2$ are the masses of the two components of the binary star, $v_{ij} \equiv |\mathbf{v}_i - \mathbf{v}_j|$ is the modulus of the relative velocity and $x_{ij} \equiv |\mathbf{x}_i - \mathbf{x}_j|$ is the modulus of the relative distance between the two particles.*

## 10.11 *Collisional vs collisionless N-body simulations*

The leapfrog scheme is possibly the most used scheme for **collisionless N-body simulations**, while the Hermite scheme is certainly the most used scheme for **collisional N-body simulations**.

A system is collisionless when close encounters between its components are extremely rare and can be neglected. Quantitatively, systems whose two-body relaxation timescale is longer than the Hubble time can be treated as collisionless and can be described by the collisionless Boltzmann equation (jumps in phase space are not allowed).

We remind that the two-body relaxation timescale can be expressed, e.g., as [Binney and Tremaine, 1987]

$$t_{\rm rlx} = \frac{\mathrm{N}}{8\, \ln \mathrm{N}} \frac{\mathrm{R}}{v}, \tag{194}$$

where R is a characteristic radius of the system, $v$ is the typical velocity of stars in the system (for a non-rotating system in virial equilibrium, i.e. supported by velocity dispersion $\sigma$, we can assume $v = \sigma$), N is the number of particle in the system. The above equation is the simplest expression for the two-body relaxation timescale and is obtained assuming that we have a spherical system of radius R, in which all stars have the same mass $m$ and the density is uniform.

In the Universe, galaxies, clusters of galaxies and larger structures are

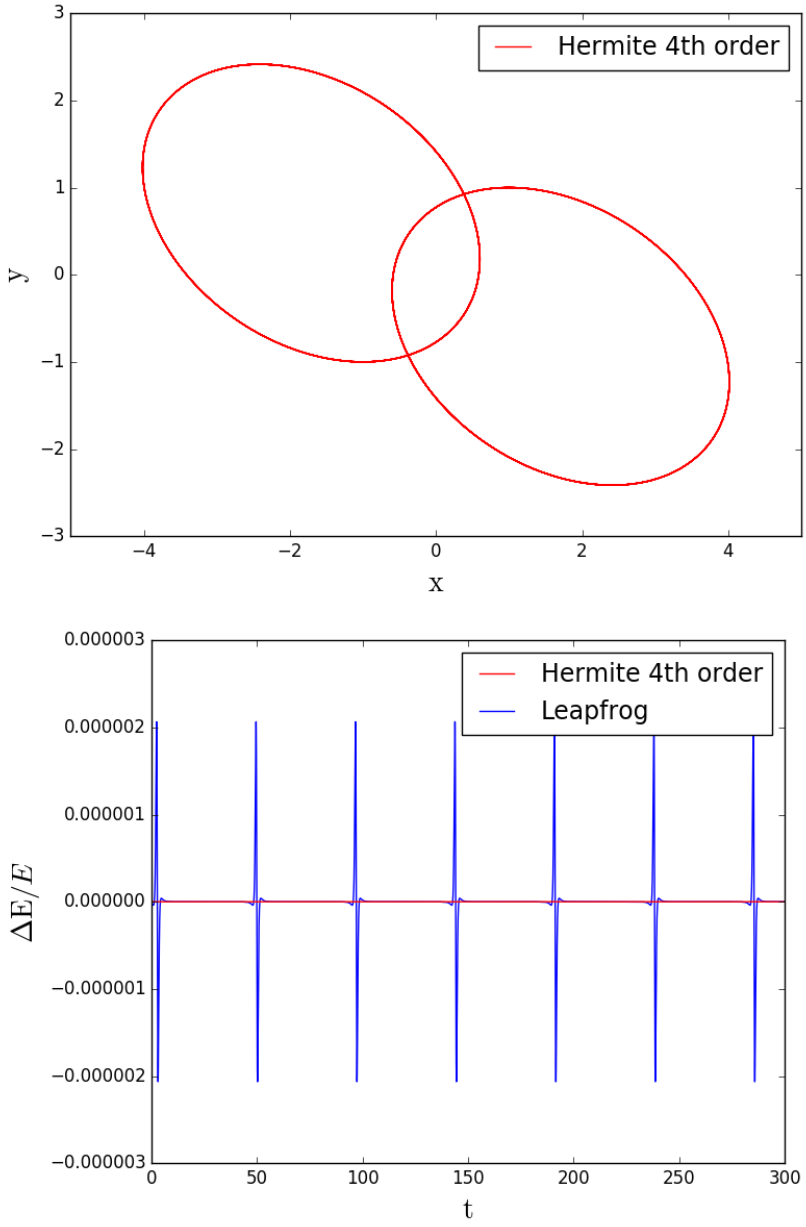


*Figure 45: Result of the integration of a binary star with the Hermite scheme for initial time $t_0 = 0.0$, final time $t_{\rm fin} = 300$ and step-size $h = 0.01$. Top panel, red line: orbit of the two particles integrated with the Hermite scheme. Bottom panel: energy conservation ($\Delta$E/E) as a function of time in the case of the Hermite scheme (red line) and of the leapfrog scheme (blue line).*

collisionless ($t_{\rm rlx} > t_{\rm H}$, where $t_{\rm H}$ is the Hubble time). N-body models of collisionless systems integrate gravity force with reasonable accuracy, to ensure energy and angular momentum conservation better than few per cent over the entire duration of the simulation. The leapfrog method is a very good scheme for these simulations, because it is extremely simple and fast and guarantees the required accuracy.

A system is collisional when close encounters between its components are frequent and can affect the overall evolution of the system. Quantitatively, collisional systems have a two-body relaxation timescale much shorter than the Hubble time. In the Universe, star clusters (globular clusters, open clusters, young star clusters and nuclear star clusters) are collisional systems. Collisional systems can be integrated only with direct N-body codes, i.e. with codes that solve Newton's gravity force directly for each single star in the star cluster, because we do not want to miss close encounters between two or more of these stars. The predictor-corrector Hermite scheme is very popular in direct N-body codes, for at least two reasons:

i) it is sufficiently high order (4th order or more), hence it guarantees energy and angular momentum conservation to $10^{-5}$ or better;

ii) the predictor-corrector scheme allows to distinguish between dynamically active and less dynamically active particles. Stars that, at a given time step, do not undergo a close encounter do not need to be integrated with fourth (or higher) order accuracy. Thus, their positions and velocities can be just predicted (without corrector step) saving a lot of computational time. In contrast, stars that, at a given time step, undergo a close encounter are integrated with the predictor and corrector step, reaching the required level of accuracy.

If you want to know more about N-body techniques and other computational techniques for astrophysics, you can follow the course on **Computational Astrophysics** (first semester next year): `https://en.didattica.unipd.it/off/2019/LM/SC/SC2443/000ZZ/SCP9087518/N0`.

## 10.12 *Adaptive step size*

So far, we have assumed that the step $h$ is constant. In most cases, the best choice is to have an **adaptive step**, i.e. a step that changes depending on the specific problem we want to solve. Let's see why.

In general, a smaller value of $h$ means higher accuracy, but an exceedingly small $h$ might result in a computational challenge, especially if we have to integrate a complex system (e.g. an astrophysical N-body problem with large N). Thus, we want to choose a value of $h$ that is a good compromise between accuracy and speed of calculation.

Unfortunately, there is no universal method to choose $h$. Usually, the most reasonable choice of $h$ is suggested by the problem we are integrating. For example, if we integrate an N-body system, we want to ensure a "reasonable" conservation of energy and angular momentum. Of course, it is not possible to achieve complete conservation, because we are doing the integration numerically, thus we cannot get rid of rounding errors and approximation errors. The term "reasonable" can be quantified depending on the degree of accuracy we need. For example, if we decide to conserve energy E within a timestep so that the maximum change is $\Delta E_{max}/E = 10^{-5}$, then we can write a program in which $h$ is reduced when the error on energy conservation after a timestep becomes $> 10^{-5}$.

Another good choice is to decide $h$ based on how fast the quantity we integrate changes. Let's take another example from the astrophysical N-body problem: many codes define $h = v/a$, where $v$ is the modulus of velocity and $a$ is the modulus of acceleration. If $a$ is large with respect to $v$, it means that a particle is changing its velocity very fast and thus we need a smaller $h$ to capture its motion with higher accuracy.

## 10.13 *When adaptive time steps matter: evolution of a binary compact object by gravitational wave emission*

We will now discuss an example, in which an adaptive time step is a condition *sine qua non* to have a good result: the evolution of a binary compact object by gravitational-wave emission.

Let us start with a short summary of **gravitational waves**, which is not part of this course, but might be useful to understand the exercise. As you have probably already seen in the general relativity course (otherwise please trust me..), gravitational waves are ripples of the space time that propagate as waves, at the speed of light. They are caused by the movement of an asymmetric distribution of masses, whenever the second time derivative of the **quadrupole moment of mass** is non-zero. The quadrupole moment of mass is defined as

$$I^{ij}(t) = \int_V dx^3 \, \rho(t, \mathbf{x}) \, x^i \, x^j \tag{195}$$

where $\rho(t, \mathbf{x})$ is the mass density and the index $i,\ j$ run over the three spatial coordinates. This is a general version of the **moment of inertia tensor** you met during your Physics 1 course.

Gravitational waves are a solution of **Einstein's field equations** under some assumptions. Einstein's field equations describe the evolution of the geometry of space-time in presence (or absence) of sources of the gravitational field (see Hartle 2003 for more information). They are highly non linear equations and do not have an analytic solution in their general formulation. However, they can be linearised if we are in the **weak-field approximation**,

i.e. if the metric tensor can be written as a Minkowski metric plus a small perturbation:

$$g_{\alpha\beta} = \eta_{\alpha\beta} + h_{\alpha\beta}, \tag{196}$$

where $\eta_{\alpha\beta}$ is the metric tensor in the Minkowski space, $h_{\alpha\beta}$ is a generic small perturbation of the Minkowski tensor and $g_{\alpha\beta}$ is the resulting metric. Analytic solutions can be derived for the linearised version of Einstein's equations, under specific Gauge choices. Using the so-called **Lorentz gauge**, the linearised Einstein's equations can be written as

$$\Box \bar{h}_{\alpha\beta} = \frac{16\,\pi\,\mathrm{G}}{c^4}\,\mathrm{T}_{\alpha\beta}, \tag{197}$$

where $\bar{h}_{\alpha\beta}$ is the trace-reversed tensor of the perturbation $h_{\alpha\beta}$ (i.e. the trace of $h_{\alpha\beta}$ is equal to minus the trace of $\bar{h}_{\alpha\beta}$), G is the gravity constant, $c$ is the speed of light and $\mathrm{T}_{\alpha\beta}$ is the stress-energy tensor, which encodes information on the sources of the gravitational field (again, see Hartle [2003] for details). This equation, containing the D'alembertian operator $\Box$, is clearly a **wave equation**, as you can remember from the solutions of Maxwell's equations in your Physics 2 course.

Under the assumption that the **sources of the gravity field** are **distant** (distant with respect to the observer) and **slowly moving** (slowly moving with respect to the speed of light), the solution of the linearised Einstein equations is

$$h^{ij}(t, \mathbf{x}) \sim \frac{2}{r}\,\frac{\mathrm{G}}{c^4}\,\frac{\mathrm{d}^2}{\mathrm{d}t^2}\,\mathrm{I}^{ij}(t - r/c), \tag{198}$$

where $h^{ij}$ is a $3 \times 3$ tensor (no time and mixed space-time components, only spatial components), $r$ is the distance of the observer from the source, G is the gravity constant, $c$ is the speed of light (note that the constant $\mathrm{G}/c^4 \sim 8 \times 10^{-50}\ \mathrm{s}^2\ \mathrm{cm}^{-1}\ \mathrm{g}^{-1}$, very small) and $t - r/c$ is the retarded time.

If the source of the gravity field is a binary system composed of two equal mass stars (or two equal mass compact objects) of mass $m$ orbiting about each other with a circular orbit, then it can be shown (see e.g. [Hartle, 2003]) that $h^{ij}$ is

$$h^{ij} \sim -\frac{16\,\mathrm{G}^2}{c^4}\,\frac{m^2}{r\,a}\begin{bmatrix} \cos\left(2\,\omega\,(t - r/c)\right) & \sin\left(2\,\omega\,(t - r/c)\right) & 0 \\ \sin\left(2\,\omega\,(t - r/c)\right) & -\cos\left(2\,\omega\,(t - r/c)\right) & 0 \\ 0 & 0 & 0 \end{bmatrix}, \tag{199}$$

where $a$ is the semi-major axis of the binary and $\omega = \sqrt{2\,\mathrm{G}\,m/a^3}$ is the angular frequency of the binary. Hence, the frequency of gravitational waves is just 2 times the orbital frequency.

Before merger, when the two bodies are still relatively far away from each other (technically this refers to the "inspiral" phase), the effects of gravitational

waves are basically two: i) to carry away energy from the system, reducing its internal energy, ii) to carry away angular momentum from the system, making it circularize.

The total internal energy of a Keplerian binary is $\mathrm{E_{int}} = -\frac{\mathrm{G}\, m_1\, m_2}{2\, a}$, where $m_1$ and $m_2$ are the masses of the two components. During the inspiral phase, transformation of mass to energy is negligible; hence the loss rate of $\mathrm{E_{int}}$ can be written as

$$\frac{\mathrm{dE_{int}}}{\mathrm{d}t} = \frac{\mathrm{G}\, m_1\, m_2}{2\, a^2} \frac{\mathrm{d}a}{\mathrm{d}t}, \tag{200}$$

i.e. the energy loss translates into a shrinking of the semi-major axis $a$ with time. For analogous reasons, circularization means loss of eccentricity $e$ ($\mathrm{d}e/\mathrm{d}t < 0$).

Peters [1964] has written down explicit expressions for $\mathrm{d}a/\mathrm{d}t$ and $\mathrm{d}e/\mathrm{d}t$, accounting for energy and angular momentum loss by gravitational waves:

$$\begin{aligned} \frac{\mathrm{d}a}{\mathrm{d}t} &= -\frac{64}{5} \frac{\mathrm{G}^3\, m_1\, m_2\, (m_1 + m_2)}{c^5\, a^3\, (1 - e^2)^{7/2}} \left(1 + \frac{73}{24}\, e^2 + \frac{37}{96}\, e^4\right) \\ \frac{\mathrm{d}e}{\mathrm{d}t} &= -\frac{304}{15}\, e \frac{\mathrm{G}^3\, m_1\, m_2\, (m_1 + m_2)}{c^5\, a^4\, (1 - e^2)^{5/2}} \left(1 + \frac{121}{304}\, e^2\right) \end{aligned} \tag{201}$$

The above equations are obviously the result of a series expansion. This looks like a nice system of two first-order ODEs. Then, let's start integrating them with your preferred algorithm. Euler is more than enough.

**EXERCISE:**

Use your Euler's script (or midpoint or RK 4th order or whatever you want) to integrate eqs. 201. Assume $m_1 = m_2 = 30\,\mathrm{M_\odot}$, $a(t = 0) = 1$ A.U., $e(t = 0) = 0.7$. Integrate this system till it reaches the last stable orbit of general relativity, defined as $a_{\mathrm{LSO}} = 3\, r_{\mathrm{sch}}$, where $r_{\mathrm{sch}} = \frac{2\,\mathrm{G}\,(m_1+m_2)}{c^2}$ is the Schwartzschild radius. As stopping condition for the main loop, use the following:

```
while((abs(a-aLS0)/aLS0>1e-1) and (iterate<1e5)):
```

this means that you stop either when the difference between the semi-major axis $a$ and the last stable orbit $a_{\mathrm{LSO}}$ is < 10% or when the number of iterations (iterate is a counter) is $\geq 10^5$. Start with a constant timestep $h = 10^3$ yr.

If you do so, you realize that either you have a very bad result (with negative $a$ and $e$ as soon as gravitational waves start being important) or a terribly slow integration (before you fuse your laptop, let me tell you that you need final timesteps of the order of few hours to get an acceptable result, with a constant timestep and RK4). The result of your constant timestep attempt should look like the blue dashed line in Figure 46 or even worse.

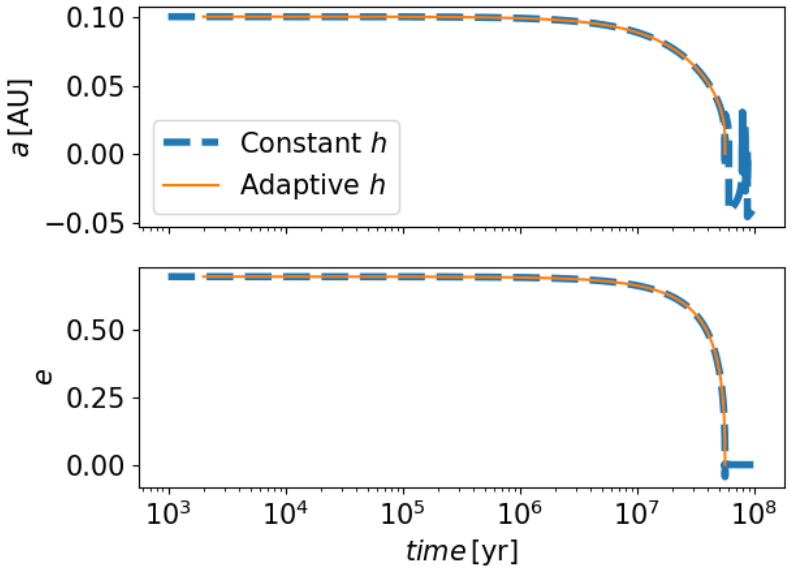


*Figure 46: Integration of the equations 201 from Peters (1964) with a Runge Kutta fourth order (RK4). Binary system with $m_1 = m_2 = 30\ M_\odot$, $a(t = 0) = 0.1$ AU, $e(t = 0) = 0.7$. Blue dashed line: constant timestep $h = 10^3$ yr. Red solid line: adaptive time step.*

The reason why this problem happens is that both $da/dt$ and $de/dt$ are very steep functions of $a$ and $e$, namely[7] $da/dt \propto a^{-3}(1-e^2)^{-7/2}$, $de/dt \propto a^{-4}(1-e^2)^{-5/2}$. Hence, a fixed timestep samples very well the first part of the integration, when $a$ and $e$ change slowly, and very badly the last part, when $a$ and $e$ change super-fast.

**How can we choose an adaptive timestep in this case? An option is that we require the relative variation of the semi-major axis to be nearly constant, or at least smaller than a chosen tolerance, during the integration.** In this way, the timestep adapts itself to the speed of changes of $a$ and consequently $e$. For example, try with

[7] Especially if you are using python2, please remember to write $(1-e^2)^{7/2}$ as (1.-e**2)**3.5, or at least as (1.-e**2)**(7./2.). Otherwise, you might have bad surprises, because (7/2) is interpreted as a division between integers by several programming languages. In this case, the result is not 3.5 but 3. My preferred option is (1.-e*e)**3.5, which gets rid of a power and substitutes it with a multiplication: slightly faster.

```
tol=1e-2
h=3.1536e10 #1e3 yr
while(a>=rth):
    anew,enew=runge4(m1,m2,a,e,h)

    if(abs(anew-a)/a<(0.1*tol)): #set adaptive timestep
        h=h*2.
        anew,enew=runge4(m1,m2,a,e,h)

    elif(abs(anew-a)/a>tol):
        while(abs(anew-a)/a>tol):
            h=h/10.
            anew,enew=runge4(m1,m2,a,e,h)
    a=anew
    e=enew
    t+=h
```

This makes the negative values to disappear and the script run much faster for a given accuracy. Any better idea?

## 10.14 *Modified mid-point method*

Let us now go back to the midpoint scheme to introduce another variation of it which is very popular, because of the Bulirsch-Stoer method (next section). The starting point of the midpoint scheme is the following equation

$$x(t+h) = x(t) + h\, f\left(x\left(t+\frac{h}{2}\right), t+\frac{h}{2}\right), \tag{202}$$

which is implicit because of $x\left(t+\frac{h}{2}\right)$. In the original version of the midpoint, we have used Euler's method to calculate $x\left(t+\frac{h}{2}\right) = x(t) + \frac{h}{2} f(x,t)$:

$$x(t+h) = x(t) + h\, f\left(x(t) + \frac{h}{2} f(x,t), t+\frac{h}{2}\right) \tag{203}$$

After this first step, in the usual midpoint scheme we calculate the midpoint of the next interval:

$$x\left(t+\frac{3}{2}h\right) = x(t+h) + \frac{h}{2} f(x(t+h), t+h) \tag{204}$$

An alternative to this (analog to the leapfrog scheme we have seen for the astrophysical N-body problem) is to evaluate directly the midpoint of the next timestep from the midpoint of the previous timestep, as:

$$x\left(t+\frac{3}{2}h\right) = x\left(t+\frac{h}{2}\right) + h\, f(x(t+h), t+h) \tag{205}$$

Once we have the above expression, we can calculate the full time step as

$$x(t+2\,h) = x(t+h) + h\, f\left(x\left(t+\frac{3}{2}h\right), t+\frac{3}{2}h\right) \tag{206}$$

and we repeat this process as many times as we want.

Hence, the general expression of this new scheme is:

$$x(t+h) = x(t) + h\, f\left(x(t) + \frac{h}{2}\, f(x,t), t+\frac{h}{2}\right)$$

$$x\left(t+\frac{3}{2}h\right) = \left[x(t) + \frac{h}{2}\, f(x,t)\right] + h\, f(x(t+h), t+h). \tag{207}$$

Suppose that we now want to solve an ODE from $t$ to $t+\mathrm{H}$ using $n$ steps of size $h = \mathrm{H}/n$. Let us write down the equations 207 in a still different form. The first half step is always:

$$\begin{aligned} x_0 &= x(t) \\ y_1 &= x_0 + \frac{1}{2} h\, f(x_0, t) \end{aligned} \tag{208}$$

Then, the next $n$ steps are:

$$\begin{aligned} x_1 &= x_0 + h\, f\left(y_1, t+\frac{1}{2}h\right) \\ y_2 &= y_1 + h\, f(x_1, t+h) \\ x_2 &= x_1 + h\, f\left(y_2, t+\frac{3}{2}h\right) \\ y_3 &= y_2 + h\, f(x_2, t+2\,h) \\ &\ldots \end{aligned} \tag{209}$$

This means that the $x_i$ terms are the solutions at integer multiples of $h$, while the $y_i$ are the solutions at half-integer multiples. Hence we can write the equations 209 in a more compact form as:

$$\begin{aligned} y_{m+1} &= y_m + h\, f(x_m, t+m\,h) \\ x_{m+1} &= x_m + h\, f\left(y_{m+1}, t+\left(m+\frac{1}{2}\right)h\right) \end{aligned} \tag{210}$$

Note that the final point is

$$x(t+\mathrm{H}) = x_n \tag{211}$$

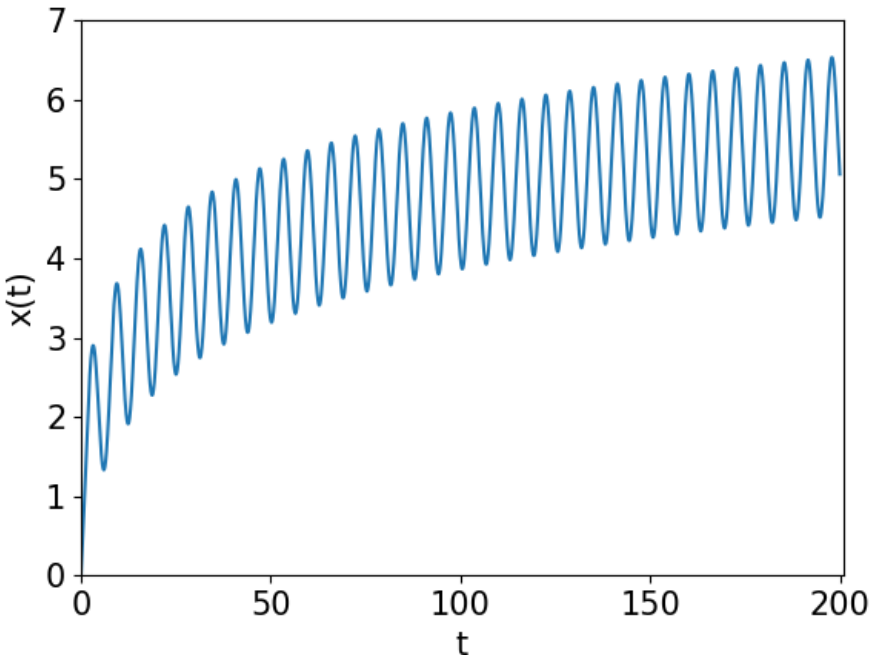


*Figure 47: Result of the modified-midpoint exercise.*

for our initial definition of H. Another possible way to calculate $x(t + \mathrm{H})$ is to start from $y_n$ and to advance by just a half-integer multiple:

$$x(t + \mathrm{H}) = y_n + \frac{1}{2} h\, f(x_n, t + \mathrm{H}) \tag{212}$$

Finally, the best way to estimate $x(t + \mathrm{H})$ is to take the average of equations 211 and 212:

$$x(t + \mathrm{H}) = \frac{1}{2}\left[x_n + y_n + \frac{1}{2} h\, f(x_n, t + \mathrm{H})\right] \tag{213}$$

It can be shown with some math that if we use the 213 all the error terms containing odd powers of $h$ that arise from the Euler's method step at the start of the calculation (eq. 208) cancel out, giving a total error that contains only even powers of $h$.

**Equations 208, 210 and 213 used together for an integration represent the modified midpoint scheme**. I repeat them here below for the sake of completeness:

$$
\begin{aligned}
x_0 &= x(t) \\
y_1 &= x_0 + \frac{1}{2} h\, f(x_0, t) \\
x_1 &= x_0 + h\, f\left(y_1, t + \frac{1}{2} h\right) \\
y_{m+1} &= y_m + h\, f(x_m, t + m\, h) \quad \forall 1 \leq m \leq (n-1) \\
x_{m+1} &= x_m + h\, f\left(y_{m+1}, t + \left(m + \frac{1}{2}\right) h\right) \quad \forall 1 \leq m \leq (n-1) \\
x(t + \mathrm{H}) &= \frac{1}{2}\left[x_n + y_n + \frac{1}{2} h\, f(x_n, t + \mathrm{H})\right] \qquad (214)
\end{aligned}
$$

**EXERCISE:**

Write a script to implement the modified midpoint method. Use it to evolve the following differential equation

$$\frac{dx}{dt} = \exp(-x) + \sin(t) \qquad (215)$$

Let us assume $t_0 = 0$, $t_{\mathrm{fin}} = 200$, $h = 0.01$ and $x_0 = 0$. The result should look like Figure 47.

## 10.15 *Bulirsch-Stoer method*

The Bulirsch-Stoer method exploits the advantages of the modified midpoint method to reach an accuracy as large as we want, at least in principle. Let's consider, for simplicity, a first-order ODE in a single variable: $dx/dt = f(x,t)$. Let's suppose we want to integrate it from time $t$ to time $t + h_1$.

We start calculating the solution i) with the modified midpoint method, ii) in just one single timestep of size $h_1$ equal to the entire range. This yields a solution $x(t + h_1) = \mathrm{R}_{1,1}$, which, of course, is a crude estimate.

Then, we split the time step in two: $h_2 = h_1/2$ and we repeat the calculation with the midpoint method, getting a solution $x(t + h_1) = \mathrm{R}_{2,1}$.

Since the error on the modified midpoint method is always and **even function of the stepsize**, we have that

$$x(t + h_1) = \mathrm{R}_{2,1} + c_1\, h_2^2 + \mathcal{O}(h_2^4) \tag{216}$$

and

$$x(t + h_1) = \mathrm{R}_{1,1} + c_1\, h_1^2 + \mathcal{O}(h_1^4) = \mathrm{R}_{1,1} + 4\, c_1\, h_2^2 + \mathcal{O}(h_2^4) \tag{217}$$

In the second equation above, we made use of the fact that $h_1 = 2\, h_2$. In both equations, $c_1$ is an unknown constant. Note that $c_1$ is the same in both equations, because of the relationship between the timesteps and the way the modified midpoint method work.

Now, we can use the equations 216 and 217 to calculate $c_1$:

$$c_1\, h_2^2 = \frac{1}{3}(\mathrm{R}_{2,1} - \mathrm{R}_{1,1}) \tag{218}$$

Substituting this back into equation 216, we find

$$x(t + h_1) = \mathrm{R}_{2,1} + \frac{1}{3}(\mathrm{R}_{2,1} - \mathrm{R}_{1,1}) + \mathcal{O}(h_2^4), \tag{219}$$

where we got rid of the terms in $h^2$. Our new equation 219 is now accurate to order $h^3$ and carries a $h^4$ error.

Let us call this new estimate $\mathrm{R}_{2,2}$:

$$\mathrm{R}_{2,2} = \mathrm{R}_{2,1} + \frac{1}{3}(\mathrm{R}_{2,1} - \mathrm{R}_{1,1}). \tag{220}$$

We can take this approach further. Let us now consider $h_3 = 1/3\, h_1$. The solution of the modified midpoint will be $x(t + h_1) = \mathrm{R}_{3,1}$. We can write down

$$x(t + h_1) = \mathrm{R}_{3,1} + c_1\, h_3^2 + \mathcal{O}(h_3^4) \tag{221}$$

$$x(t+h_1) = \mathrm{R}_{2,1} + c_1\, h_2^2 + \mathcal{O}(h_2^4) = \mathrm{R}_{2,1} + \frac{9}{4} c_1\, h_3^2 + \mathcal{O}(h_3^4) \tag{222}$$

Hence, we can evaluate $c_1$ as:

$$c_1\, h_3^2 = \frac{4}{5}(\mathrm{R}_{3,1} - \mathrm{R}_{2,1}) \tag{223}$$

and finally we can define:

$$\mathrm{R}_{3,2} = \mathrm{R}_{3,1} + \frac{4}{5}(\mathrm{R}_{3,1} - \mathrm{R}_{2,1}) \tag{224}$$

to obtain

$$x(t+h_1) = \mathrm{R}_{3,2} + \mathcal{O}(h_3^4) \tag{225}$$

which is a third order algorithm with fourth order errors.

We can now write down

$$x(t+h_1) = \mathrm{R}_{3,2} + c_2\, h_3^4 + \mathcal{O}(h_3^6) \tag{226}$$

Combining eq. 219 and eq. 220, we also have

$$x(t+h_1) = \mathrm{R}_{2,2} + c_2\, h_2^4 + \mathcal{O}(h_2^6) = \mathrm{R}_{2,2} + \left(\frac{3}{2}\right)^4 c_2\, h_3^4 + \mathcal{O}(h_3^6) \tag{227}$$

From equations 226 and 227 we can get the value of

$$c_2\, h_3^4 = \frac{16}{65}(\mathrm{R}_{3,2} - \mathrm{R}_{2,2}) \tag{228}$$

and substituting this into eq. 226 we get

$$x(t+h_1) = \mathrm{R}_{3,3} + \mathcal{O}(h_3^6), \tag{229}$$

where

$$\mathrm{R}_{3,3} = \mathrm{R}_{3,2} + \frac{16}{65}(\mathrm{R}_{3,2} - \mathrm{R}_{2,2}). \tag{230}$$

Now our error is of order $h^6$ and we have taken only 3 modified midpoint steps. **The power of this method is that it cancels out the error terms to higher and higher orders on successive steps.**

We can go to an arbitrarily high order. Suppose we denote the current

number of steps by $n$ and our modified midpoint estimate of $x(t + h_1)$ by $R_{n,1}$. Then we can generalize the above method with the formula:

$$x(t + h_1) = R_{n,m} + c_m\, h_n^{2m} + \mathcal{O}(h_n^{2m+2}), \tag{231}$$

where $c_m$ is the unknown constant.

The corresponding estimate at step $n - 1$ is

$$x(t + h_1) = R_{n-1,m} + c_m\, h_{n-1}^{2m} + \mathcal{O}(h_{n-1}^{2m+2}) \tag{232}$$

but $h_n = h_1/n$ and $h_{n-1} = h_1/(n - 1)$, so

$$h_{n-1} = \frac{n}{n-1} h_n \tag{233}$$

Substituting eq. 232 and 233 into eq. 231 and reshuffling to express in terms of $c_m$, we find

$$c_m\, h_n^{2m} = \frac{R_{n,m} - R_{n-1,m}}{[n/(n-1)]^{2m} - 1} \tag{234}$$

Finally, substituting equation 234 into eq. 232 we get the most general expression of the Bulirsch-Stoer:

$$x(t + h_1) = R_{n,m+1} + \mathcal{O}(h_n^{2m+2}), \tag{235}$$

where

$$R_{n,m+1} = R_{n,m} + \frac{R_{n,m} - R_{n-1,m}}{[n/(n-1)]^{2m} - 1} \tag{236}$$

Since this method is quite complex, we will see together the example of a binary star (our usual binary star).

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

G=1.0
m1=1.0
m2=1.0
Ttot= 300. #final time
N = 300 #number of big steps
h1 = Ttot/float(N) # size of big steps
delta = 1e-9 #position accuracy per unit time


def ode(r):
    num=4
    x=np.zeros(num,float)
    v=np.zeros(num,float)
    a=np.zeros(num,float)

    x=r[0:4]
    #x1 = r[0]
    #y1 = r[1]
    #x2 = r[2]
    #y2 = r[3]
    v=r[4:8]
    #vx1 = r[4]
    #vy1 = r[5]
    #vx2 = r[6]
    #vy2 = r[7]

    rij=((x[0]-x[2])**2.+(x[1]-x[3])**2.)**1.5
    for i in range(int(num/2)):
        a[i] = -G * m2 * (x[i]-x[i+2])/rij
    for i in range(int(num/2),num):
        a[i] = -G * m1 * (x[i]-x[i-2])/rij


    return np.array([v,a],float).ravel()



tp = np.arange(0.,Ttot,h1)
x1p = []
y1p = []
x2p = []
y2p = []

r = np.array([1.0,1.0,-1.0,-1.0,-0.5,0.0,0.5,0.0],float)
num=len(r)
```

```
#Do the big steps of size h1
for t in tp:
    x1p.append(r[0])
    y1p.append(r[1])
    x2p.append(r[2])
    y2p.append(r[3])

    #Do first big timestep
    n = 1
    r1 = r + 0.5 * h1 * ode(r)
    r2 = r + h1 * ode(r1)

    #The array R1 stores the first row of the extrapolation table,
    #which contains only the single modified midpoint estimate
    #at the end of the interval h1
    R1 = np.zeros([1,num],float) #8 columns because 8 variables x1y1x2y2vx1vy1vx2vy2
    #R1[0] = 0.5 * (r1 + r2 + 0.5 * h1 * ode(r1))
    R1[0] = 0.5 * (r1 + r2 + 0.5 * h1 * ode(r2))

    #Now increase n until the required accuracy is reached
    error = 2 * h1 * delta
    while(error > (h1 * delta)):
        n+=1
        h = h1/n

        #Modified midpoint method
        r1 = r + 0.5 * h * ode(r)
        r2 = r + h * ode(r1)
        for i in range(n-1):
            r1 += h * ode(r2)
            r2 += h * ode(r1)

        #Calculate extrapolation estimates.
        #Arrays R1 and R2 hold the two modst recent lines of the table
        R2=np.copy(R1)
        R1 = np.zeros([n,num],float)
        #R1[0] = 0.5 * (r1 + r2 + 0.5 * h * ode(r1))
        R1[0] = 0.5 * (r1 + r2 + 0.5 * h * ode(r2))
        for m in range(1,n):
            c = (R1[m-1]-R2[m-1])/((n/(n-1.))**(2.*m)-1.)
            R1[m] = R1[m-1] + c
        error = abs(c[0])
    #Set r equal to the most accurate estimate we have
    # before moving on to the next big step
    r = R1[n-1]


fig1, ax1 = plt.subplots()
#ax1.set_aspect('equal')
ax1.scatter(x1p,y1p)
ax1.scatter(x2p,y2p)
ax1.set_xlabel("x")
ax1.set_ylabel("y")
plt.show()
```

## 10.16 *Initial value problems and boundary value problems*

All the examples and exercises we have seen so far are **initial-value problems**: we solve differential equations given the initial values of the variables. There is another kind of problems, in which we do not know the initial values of the variables, but only the values of the variables at "some point/time".

For example, suppose we want to integrate the height above the ground of a ball thrown in the air and subject to gravity force (no friction from the air, for simplicity). Its motion is described by

$$\frac{\mathrm{d}^2 x}{\mathrm{d}t^2} = -g, \tag{237}$$

where $g = 9.81$ m s$^{-2}$ is the acceleration due to gravity.

This is an initial-value problem when we know the initial position and the initial velocity of the ball. In contrast, it becomes a boundary value problem, when we know (for example) its position at time $t = 0$ and its position at another time $t = t_1$ but we do not know its initial velocity. Boundary-value differential equations are usually harder to solve computationally.

## 10.17 *The shooting method*

A possibility to solve a boundary-value problem is to use the **shooting method**: we start with a guess of the initial values we do not know and then we improve our guess iteratively.

For example, if the problem is the ball thrown in the air and we know the initial position and the position at $t = t_1$, we start with a guess on the initial velocity, then we calculate the position $x_1$ at time $t = t_1$. If this position is different from the value we know, then we come back to the initial conditions and we try with a different velocity, namely with a larger (smaller) one if we find a value of $x_1$ smaller (larger) than the given one. We repeat this procedure till the difference between the given value of $x_1$ and the recovered one is smaller than the tolerance we have decided. This is essentially a problem of finding the zeros of a function: we can solve it with the bisection method we have learned.

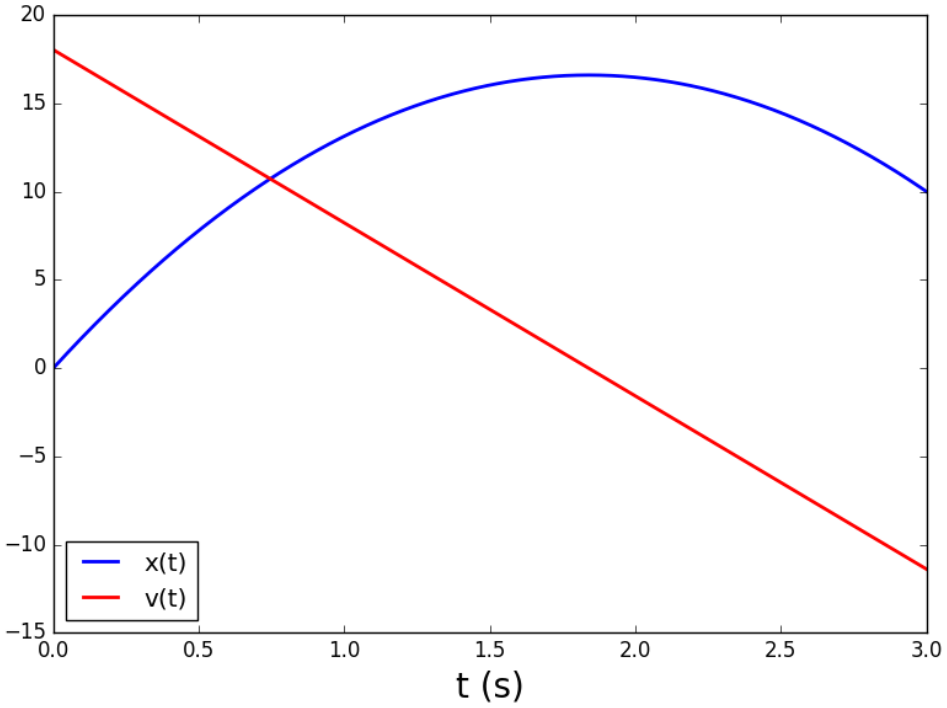


*Figure 48: Result of the exercise on the shooting method. Blue (Red) line: position (velocity) of the ball as a function of time. Position (Velocity) is shown in m (m/s).*

**EXERCISE:**

Solve the problem proposed in equation 237, i.e. integrate the motion of a ball vertically thrown in the air with initial position $x(t = 0) = 0$ and position $x(t = 3\,s) = 10$ m, at $t = 3$ s. Neglect the viscous friction from the air. Use the shooting method.

*Suggestion: use the Euler method to solve the ODEs. Take* $\mathrm{N} = 10^3$ *steps between* $t = 0$ *and* $t_f = 3$ *s. Assume a tolerance* $\epsilon = 10^{-3}$ *and stop the calculation when* $\left|x(t_f) - x_{\text{true}}(t_f)\right| < \epsilon$, *where* $x_{\text{true}}(t_f) = 10$ *m.*

The result should look like Figure 48. In particular, the correct initial velocity is $v_0 \sim 18.0$ m/s.

# 11 PARTIAL DIFFERENTIAL EQUATIONS

This chapter is based on *Computational Physics* by Mark Newman `http://www-personal.umich.edu/~mejn/cp/`.

A differential equation involving more than one independent variable is called a partial differential equation (PDE). Many problems in applied science, physics and engineering are modeled mathematically with PDEs. For example, the **wave** equation is a PDE:

$$-\frac{1}{c^2}\frac{\partial^2\psi}{\partial t^2} + \nabla^2\psi = 0, \tag{238}$$

where we remind that the Laplacian is

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}$$

$$\nabla^2 = \frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial}{\partial r}\right) + \frac{1}{r^2\,\sin\phi}\frac{\partial^2}{\partial\phi^2} + \frac{\partial^2}{\partial z^2}$$

$$\nabla^2 = \frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial}{\partial r}\right) + \frac{1}{r^2\,\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial}{\partial\theta}\right) + \frac{1}{r^2\,\sin^2\theta}\frac{\partial^2}{\partial\phi^2}, \tag{239}$$

in Cartesian, polar and spherical coordinates, respectively.

Similarly, the diffusion equation, the Laplace equation, the Maxwell's equations, the Schrödinger equation and the Poisson equations are PDEs.

## 11.1 *Boundary-value PDEs with finite difference methods*

As a simple example, let's start from the Laplace's equation:

$$\nabla^2\phi = 0, \tag{240}$$

where $\phi$ is the electrostatic potential in the absence of electric charges. Let's write it in two dimensions for simplicity

$$\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = 0 \tag{241}$$

For example, this could be the case of an empty box with conducting walls. The boundary conditions are that one wall is kept at voltage V, while the other three walls are insulated from it and are at 0 volts. This is an electrostatics problem, so we do not have time evolution here, but we want to calculate the potential $\phi(x, y)$ at each point in the 2D box.

First, we divide the box into a **grid** of points, each of them separated by a

small distance $a$ (in this example the grid is square and regular, but ideally we can have any possible type of grid: the problem itself suggests what kind of grid is better to use).

We can now use what we learned from the chapter on numerical derivatives (equation 72) and write each term of equation 241 as follows:

$$\frac{\partial^2 \phi}{\partial x^2} \simeq \frac{1}{a^2} \left[\phi(x+a, y) + \phi(x-a, y) - 2\,\phi(x, y)\right]$$
$$\frac{\partial^2 \phi}{\partial y^2} \simeq \frac{1}{a^2} \left[\phi(x, y+a) + \phi(x, y-a) - 2\,\phi(x, y)\right] \tag{242}$$

where $x + a$, $x$ and $x - a$ are adjacent points on the x-axis and $y + a$, $y$ and $y - a$ are adjacent points on the y-axis of the grid.

Substituting equation 242 into equation 241 we then get

$$\frac{1}{a^2} \left[\phi(x+a, y) + \phi(x-a, y) + \phi(x, y+a) + \phi(x, y-a) - 4\,\phi(x, y)\right] = 0 \tag{243}$$

In this way, we have transformed a second-order, double variable differential equation into a set of differences, evaluated over the grid points. We can now get rid of the $1/a^2$ and we can write the 243 as follows

$$\phi(x, y) = \frac{1}{4} \left[\phi(x+a, y) + \phi(x-a, y) + \phi(x, y+a) + \phi(x, y-a)\right] \tag{244}$$

Equation 244 states that we can derive the value of $\phi(x, y)$ at any point in the grid, provided that we solve **a set of linear equations**.

Hence, solving PDEs with a finite difference method ends up being equivalent to solving systems of linear equations: we can use the methods we have learned to solve sets of linear equations.

## 11.2 *Solve by iteration*

Since we have approximated our derivatives with finite differences, the solution of the PDEs will always be just an **approximated solution**. Hence, it is pointless to struggle with some very sophisticated algorithms. Moreover, since the accuracy of our final results will depend on how small is $a$, and thus on how many points we have in the grid, the system we want to solve will ideally have a very large number of linear equations: it is very important to choose a fast method to solve a **large system of linear equations**. For these two reasons, the best possible choice is to use a **fast approximate method** to solve our system of linear equations, rather than a slower exact method.

Equation 244 is immediately reminiscent of an iterative method to find the solution of equations (e.g. the relaxation method and the Gauss-Seidel method). Let's then set up this iterative method.

1) **Assign boundary values:**

   We must first assign the boundary values to the points of the grid that are at the boundaries (these are the only values of $\phi(x, y)$ that we already know). Thus, in our example, if we have a grid of M × M points: $\phi[0, :] = V$ (the first row is always V), $\phi[M-1, :] = \phi[1 : M-1, 0] = \phi[1 : M-1, M-1] = 0$.

2) **Start from a guess for the unknown values**

   Then, we assign to all the other points of the grid an *Ansatz* value (a guess). For example, $\phi[i, j] = 0$ for all $i \neq 0, (M-1)$ and $j \neq 0, (M-1)$. A PDE problem reduced to finite differences always translates into a quite simple (although very large) system of linear equations, thus we should not worry too much about convergence issues: whatever initial guess you star from should be fine.

3) **Iterate your calculation**

   Take the results of equation 244 and feed them back to the right side of the equation. Iterate your calculation over the whole grid as many times as you need to reach the requested accuracy. A good way to do so is to check for convergence. At each iteration, you calculate the norm of $\phi_{new} - \phi$, where $\phi_{new}$ is the current solution for the entire matrix, $\phi$ was the solution at the previous iteration for the entire matrix, and you check whether this difference is less than a requested accuracy.

Finally, it is recommended to use a **relaxation technique**, to make the procedure even faster. Let's see what relaxation means in this case. The relationship between the new guess of $\phi(x, y)$ (let's call it $\phi_{new}(x, y)$) and the old one (let's call it $\phi(x, y)$) from equation 244 can be written as

$$\Delta\phi(x, y) \equiv \phi_{new}(x, y) - \phi(x, y) \tag{245}$$

where $\Delta\phi(x, y)$ is the change of $\phi$ during this step.

Now, let's define a set of relaxed values $\phi_\omega(x, y)$ as

$$\phi_\omega(x, y) = \phi(x, y) + (1 + \omega)\,\Delta\phi(x, y), \tag{246}$$

where $\omega > 0$. In other words, we change each $\phi$ by a little more than we would in the un-relaxed case.

Substituting eq. 245 into the above expression for $\phi_\omega(x, y)$ we get

$$\begin{aligned}\phi_\omega(x, y) &= \phi(x, y) + (1 + \omega)\,[\phi_{new}(x, y) - \phi(x, y)], \\ &= (1 + \omega)\,\phi_{new}(x, y) - \omega\,\phi(x, y),\end{aligned} \tag{247}$$

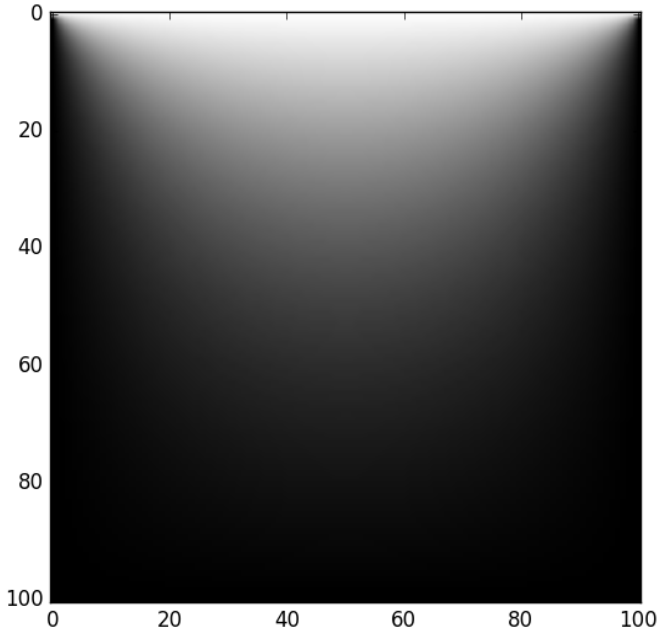


*Figure 49: Result of the electrostatics problem for* $\phi[0,:] = 1.0$ *volt and all the other walls at zero volts. Tolerance* $\epsilon = 10^{-3}$.

Using equation 247 (and assuming that $\phi_\omega(x, y)$ is the one we want to calculate), equation 244 becomes

$$\phi(x, y) = \frac{(1 + \omega)}{4} \left[\phi(x + a, y) + \phi(x - a, y) + \phi(x, y + a) + \phi(x, y - a)\right] - \omega\, \phi(x, y). \tag{248}$$

There is no clear rule to estimate $\omega$. In general, values of $\omega > 1$ become unstable, while values of $\omega < 1$ are always stable but slower. The rule-of-thumb is to try with $\omega = 0.9$ first, and then to reduce $\omega$ "by hand"if the solver is not stable.

**EXERCISE:**

Consider an empty 2D box with conducting walls. The boundary conditions are that one wall is kept at voltage V = 1 volt (for example, the first row of the matrix is kept at voltage V = 1 volt), while the other three walls are insulated from it and are at 0 volts. We want to calculate the potential $\phi(x, y)$ at each point in the 2D box. Plot the result by using the function matplotlib.pyplot.imshow().

*Suggestions: Require a tolerance delta* $= 10^{-3}$ *(smaller tolerances require a significant computing time). For the definition of tolerance you can use*

```
delta=numpy.linalg.norm(phi-phiold)
```

*where phi and phiold are the new and the old iteration of the matrix (this treats the two matrices as two vectors and calculates the norm of their difference).*
*Create a grid of* $M \times M$ *cells, with* $M = 100$. *Rows* 0 *and* $M - 1$ *and columns* 0 *and* $M - 1$ *are the walls of the box (i.e. the boundaries) and must be assigned the given voltage. The result should look like Figure 49 and should require* $\approx 3000$ *iterations to reach the required tolerance (for the above definition of tolerance).*
*To produce the plot use:*

```
import matplotlib.pyplot as plt
plt.imshow(phi) #where phi is the final matrix
plt.gray()
plt.show()
```

Now, solve the same exercise with relaxation, as proposed in equation 248.

*Suggestions: Choose* $\omega = 0.9$ *(higher* $\omega$ *might be unstable). With* $\omega = 0.9$ *you should reach a tolerance of delta* $= 10^{-3}$ *in* $\approx 200 - 300$ *iterations. The script is much faster with the* **relaxation**.

## 11.3 *Initial-value PDEs with finite difference methods*

We now consider the case of initial-value PDEs, i.e. PDEs for which we know the initial conditions and we must calculate the time evolution of one or more

variables.

As a simple example (which is also involved in stellar evolution and stellar dynamics), let's take the one-dimensional diffusion equation

$$\frac{\partial \phi}{\partial t} = \mathrm{D}\,\frac{\partial^2 \phi}{\partial x^2}, \tag{249}$$

where D is the diffusion coefficient. The variable $\phi(x, t)$ depends on both the spatial coordinate $x$ and time $t$. Thus, equation 249 is a PDE with two independent variables. However, it cannot be solved by simply creating a grid of points, because this is not a boundary-value problem but an initial-value problem: we know the initial value of $\phi(x, t)$, not the boundaries. We can proceed as follows.

## 11.4 *The forward-time centered-space (FTCS) method for initial-value PDEs*

First, we consider the spatial dependence of $\phi$ and write down the numerical derivative

$$\frac{\partial^2 \phi}{\partial x^2} = \frac{\phi(x + a, t) + \phi(x - a, t) - 2\,\phi(x, t)}{a^2}, \tag{250}$$

where we have chosen a spatial step $a$ and we have gridded our spatial domain into a grid of M points (if we were in two dimensions, then we would have used a grid of $\mathrm{M} \times \mathrm{M}$ points). Substituting the above numerical derivative into equation 249, the diffusion equation becomes

$$\frac{\partial \phi}{\partial t} = \frac{\mathrm{D}}{a^2}\,\left[\phi(x + a, t) + \phi(x - a, t) - 2\,\phi(x, t)\right], \tag{251}$$

where we have only one derivative left. Hence, we can substitute $\partial\phi/\partial t$ to $\mathrm{d}\phi/\mathrm{d}t$. Now, we can think of equation 251 as of a **set of** M **ordinary differential equations**, one per each point of the spatial grid.

Thus, we can solve it by applying one of the methods we learned for ordinary differential equations. The smartest method we can adopt here is simply the **Euler's method**. The Euler method is just a first-order method: very fast but quite inaccurate. On the other hand, it is not smart to use a more accurate method, because the numerical derivative in the right-hand term of equation 251 is first-order accuracy: it is pointless to look for a high-order accuracy in time when we are limited by first-order accuracy in space.

## 11.5 *Euler's method, again*

Euler's method is based on a first order Taylor expansion, thus for our function $\phi$:

$$\phi(x, t + h) \simeq \phi(x, t) + h\,\frac{\mathrm{d}\phi(x, t)}{\mathrm{d}t} \tag{252}$$

Substituting the value of $\mathrm{d}\phi(x,t)/\mathrm{d}t$ from equation 251 into the above equation, we get

$$\phi(x, t+h) \simeq \phi(x,t) + \frac{h\,\mathrm{D}}{a^2}\left[\phi(x+a,t) + \phi(x-a,t) - 2\,\phi(x,t)\right] \tag{253}$$

which is the basic equation of the forward-time centered-space (FTCS) method for initial-value PDEs, applied to diffusion equation.

**EXERCISE:**

Write a python script to solve the diffusion equation with the FTCS method for the following system.

The flat base of a container made of 1 cm thick stainless steel is initially at a uniform temperature of $\mathrm{T}_0 = 293$ K. The container is placed in a bath of cold water at $\mathrm{T}_{\mathrm{bath}} = 273$ K and is filled with hot water at $\mathrm{T}_{\mathrm{hot}} = 323$ K. Calculate the temperature profile of the steel as a function of distance $x$ from the hot side to the cold side (from 0 to 1 cm) and as a function of time. The system is shown in Figure 50.
Thermal conduction is described by the diffusion equation as

$$\frac{\partial \mathrm{T}}{\partial t} = \mathrm{D}\,\frac{\partial^2 \mathrm{T}}{\partial x^2}, \tag{254}$$

where $\mathrm{D} = 4.25 \times 10^{-2}$ cm$^2$ s$^{-1}$ for stainless steel. Use equation 253 to solve the problem.
Plot the temperature profile of the steel as a function of $x$ at four different times $t = 0.01$, 0.1, 1 and 10 s.

The result should look like Figure 51.

*Suggestion: use a spatial grid with M+1 point, where M=100. Choose h=0.001. What happens if you change h and why?*

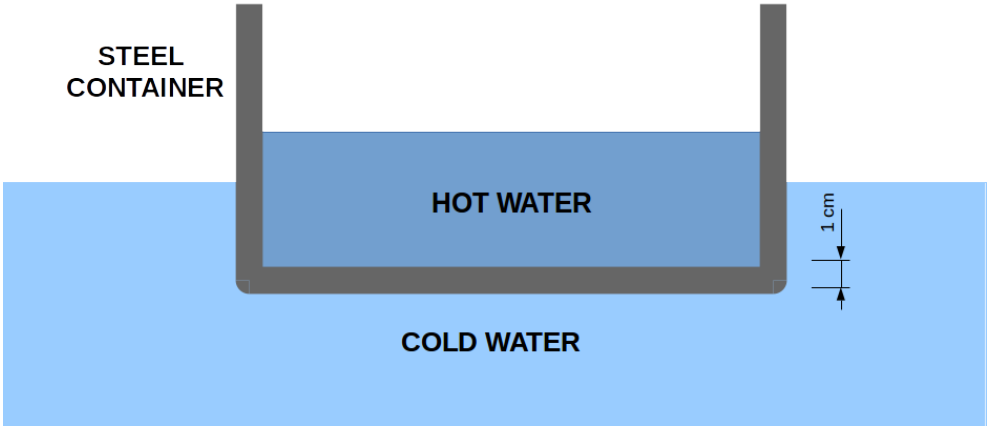


*Figure 50: System of exercise 9.2*

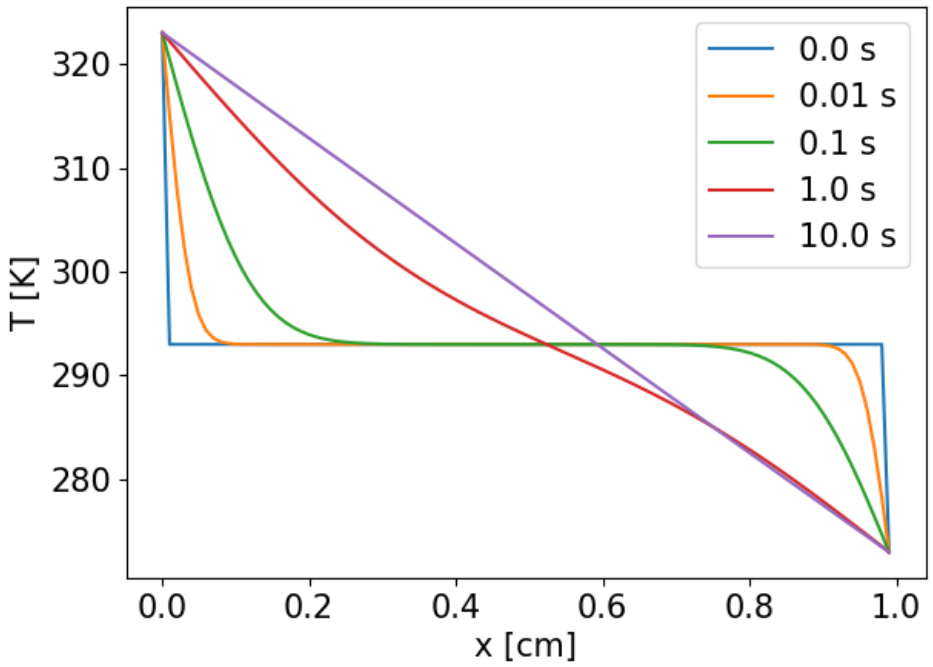


*Figure 51: Result of the initial value problem with $h = 0.001$. Temperature profile of the steel as a function of $x$ at times $t = 0$, 0.01, 0.1, 1 and 10 s.*

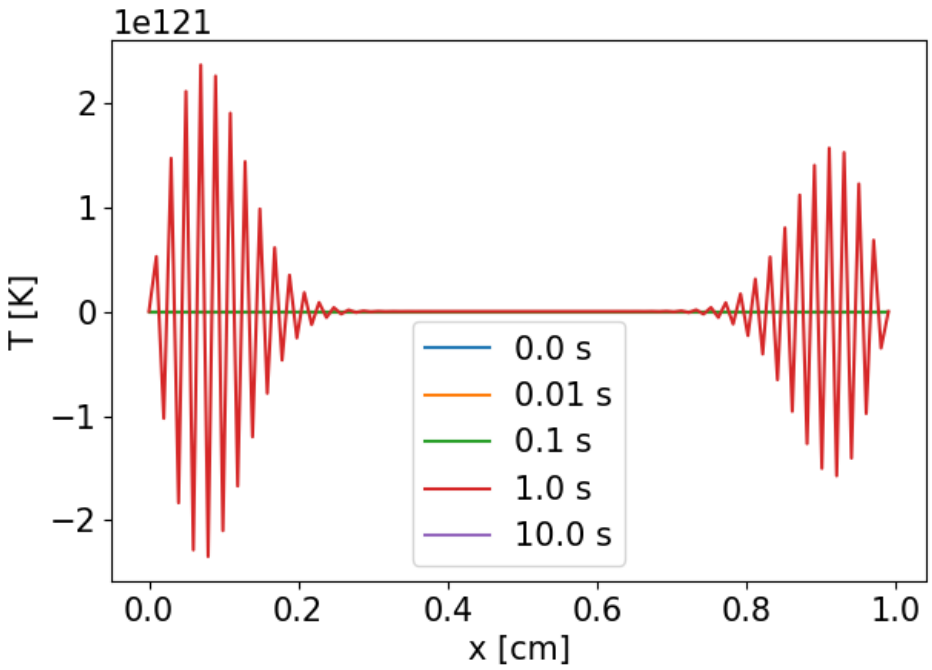


*Figure 52: Result of the initial value problem with $h = 0.01$. Temperature profile of the steel as a function of $x$ at times $t = 0$, 0.01, 0.1, 1 and 10 s. The solution diverges.*

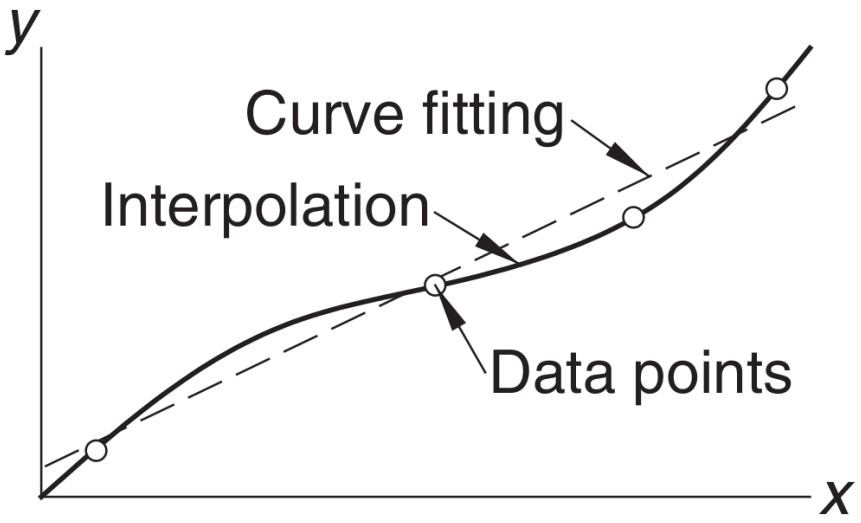


*Figure 53: Visualization of the difference between interpolation and curve fitting. Figure from Kiusalaas.*

## 12 INTERPOLATION AND EXTRAPOLATION

This chapter is based on *Numerical Methods in Engineering with Python* by Jaan Kiusalaas.

Interpolation and curve fitting should not be confused. In **interpolation**, we construct a curve passing *through* a discrete set of data points. In doing so, we make the implicit assumption that the data points are accurate and distinct.

**Curve fitting** is applied to data that contain scatter (noise), usually due to measurement errors. In this case, we want to find a smooth curve that approximates the data. Thus, the curve does not necessarily hit the data points. Figure 53 visualizes the difference between these two approaches. In practice, interpolation is used more often when we deal with theoretical data points (e.g. the outcome of discrete simulations, discrete numerical models), curve fitting is mostly done on a set of observational data.

In this chapter we will discuss interpolation, while in the next chapter we will talk about curve fitting.

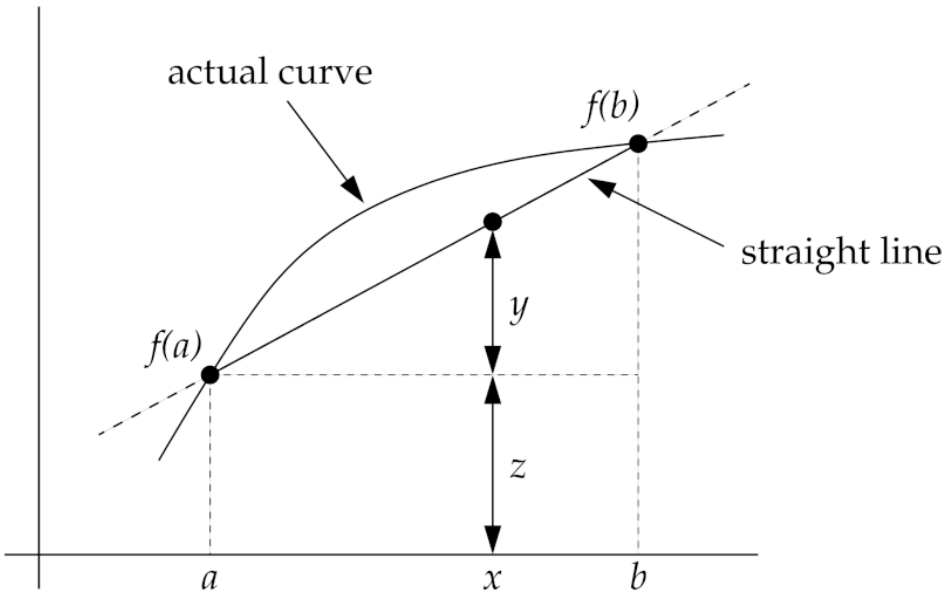


*Figure 54: Visualization of a linear interpolation. Figure from Newman.*

## 12.1 *Linear interpolation*

Suppose you are given the value of a function $f(x)$ at just two points $a$ and $b$ and you want to know the value of the function at another point $x$ in between. The simplest way to proceed (and often the BEST WAY to proceed) is to assume our function follows a straight line from $f(a)$ to $f(b)$.

In practice, the slope of the straight-line approximation is

$$m = \frac{f(b) - f(a)}{b - a}. \tag{255}$$

Thus, $f(x)$ will be simply given by

$$\begin{aligned} f(x) \simeq y + z &= \frac{f(b) - f(a)}{b - a}(x - a) + f(a) \\ &= \frac{(b - x)\, f(a) + (x - a)\, f(b)}{b - a} \end{aligned} \tag{256}$$

where $y$ and $z$ are the quantities shown in Figure 54.

In most astrophysical applications, we do not have a continuous function $f(x)$, but rather a sample of $n$ discrete data points in the form

| | | | | |
|---|---|---|---|---|
| $x_0$ | $x_1$ | $x_2$ | ... | $x_n$ |
| $y_0$ | $y_1$ | $y_2$ | ... | $y_n$. |

In the above example, the $x_i$ might be the discrete values of time at which

I have a measure, and $y_i$ might be the values of the measure at time $x_i$. For example, the $y_i$ could be fluxes of an astronomical source at time $x_i$.

The procedure of linear interpolation gives its best if we perform one linear interpolation every pair of data. This means that the linear interpolation formula becomes:

$$y(x) = \frac{(x_{i+1} - x)\, y_i + (x - x_i)\, y_{i+1}}{x_{i+1} - x_i} \tag{257}$$

This is often the best way to interpolate on a grid of data points. It could also be used for extrapolation, but with many grains of salt.

**EXERCISE:**

The file 120a300.fis contains the evolution of a 120 $M_\odot$ star at solar metallicity integrated with the stellar evolution code FRANEC [Limongi and Chieffi, 2018]. Columns 0 and 7 are the evolutionary time (in years) and the mass (in $M_\odot$). The file evol_120msun_scattered.dat shows a subsample of data points with respect to file 120a300.fis, with a much coarser time grid. Columns 0 and 1 are the evolutionary time (in Myr) and the mass (in $M_\odot$). In both files the comments are preceded by #. Using the linear interpolation, interpolate the values of the mass from file evol_120msun_scattered.dat in all the times of 120a300.fis. Then compare your interpolation with the values in the original file 120a300.fis. The plot should look like Figure 55.

## 12.2 *Polynomial interpolation with Lagrange's method*

The simplest form of an interpolant is a polynomial. It is always possible to construct an unique polynomial of degree $n$ that passes through $n + 1$ data points. Assume we have a sample of $n$ data points in the form $x_i$, $y_i$ as already discussed at the end of the previous section.

One means of obtaining this polynomial is the formula of Lagrange

$$\mathrm{P}_n(x) = \sum_{i=0}^{n} y_i\, l_i(x), \tag{258}$$

where the subscript $n$ demotes the degree of the polynomial and

$$l_i(x) = \prod_{j=0, j\neq i}^{n} \frac{x - x_j}{x_i - x_j}, \quad i = 0, 1, ..., n \tag{259}$$

are called the cardinal functions. For example, if $n = 1$, the interpolant is the

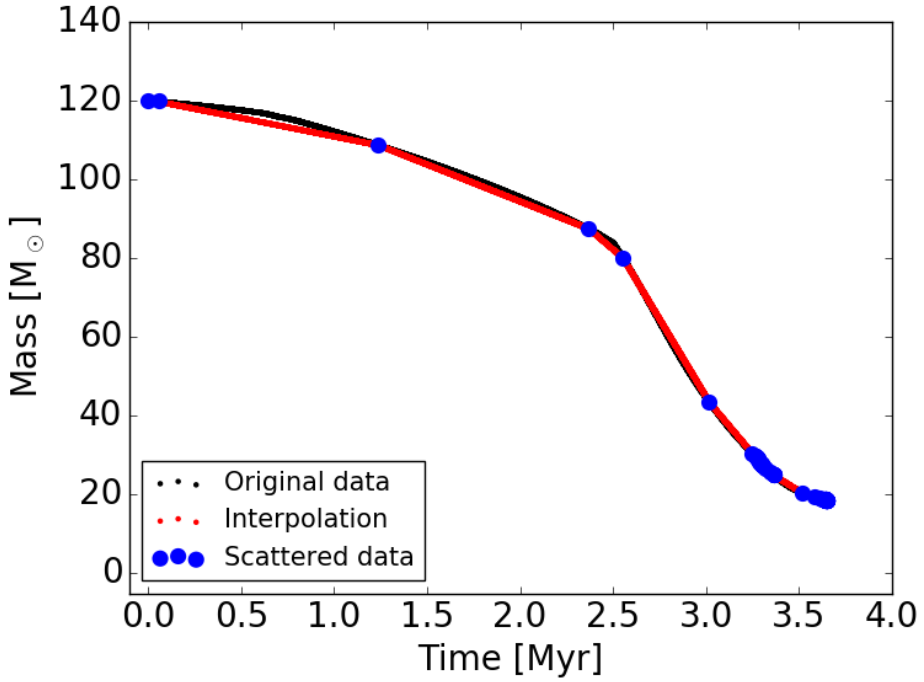


*Figure 55: Result of the exercise on linear interpolation.*

straight line $P_1(x) = y_0\, l_0(x) + y_1\, l_1(x)$, where

$$l_0(x) = \frac{x - x_1}{x_0 - x_1}$$
$$l_1(x) = \frac{x - x_0}{x_1 - x_0} \tag{260}$$

It can be easily found that the case with $n = 1$ is the linear interpolation we discussed in the previous section.

If $n = 2$ the interpolation is parabolic: $P_2(x) = y_0\, l_0(x) + y_1\, l_1(x) + y_2\, l_2(x)$ with

$$l_0(x) = \frac{x - x_1}{x_0 - x_1}\,\frac{x - x_2}{x_0 - x_2}$$
$$l_1(x) = \frac{x - x_0}{x_1 - x_0}\,\frac{x - x_2}{x_1 - x_2}$$
$$l_2(x) = \frac{x - x_0}{x_2 - x_0}\,\frac{x - x_1}{x_2 - x_1} \tag{261}$$

This method is conceptually simple but quite inefficient to implement numerically.

## 12.3 *Polynomial interpolation with Newton's method*

Newton's method is another polynomial interpolation method, similar to Lagrange's method but much better to implement numerically.

The interpolating polynomials are written as

$$\mathrm{P}_k(x) = a_{n-k} + (x - x_{n-k})\,\mathrm{P}_{k-1}(x), \tag{262}$$

where $n$ is the interpolation order and $k = 0,\ 1,\ 2,\ ...,\ n$.

For example, if $n = 1$

$$\begin{aligned} \mathrm{P}_0(x) &= a_1 \\ \mathrm{P}_1(x) &= a_0 + (x - x_0)\,\mathrm{P}_0(x) \end{aligned} \tag{263}$$

If $n = 2$,

$$\begin{aligned} \mathrm{P}_0(x) &= a_2 \\ \mathrm{P}_1(x) &= a_1 + (x - x_1)\,\mathrm{P}_0(x) \\ \mathrm{P}_2(x) &= a_0 + (x - x_0)\,\mathrm{P}_1(x) \end{aligned} \tag{264}$$

If $n = 3$

$$\begin{aligned} \mathrm{P}_0(x) &= a_3 \\ \mathrm{P}_1(x) &= a_2 + (x - x_2)\,\mathrm{P}_0(x) \\ \mathrm{P}_2(x) &= a_1 + (x - x_1)\,\mathrm{P}_1(x) \\ \mathrm{P}_3(x) &= a_0 + (x - x_0)\,\mathrm{P}_2(x) \end{aligned} \tag{265}$$

and so on. Hence, the $k = 3$ polynomial can be written as:

$$\mathrm{P}_3(x) = a_0 + (x - x_0)\,\{a_1 + (x - x_1)\,[a_2 + (x - x_2)\,a_3]\} \tag{266}$$

The coefficients $a_k$ are determined by forcing the polynomial to pass through each data point $y_i = \mathrm{P}_k(x_i)$ with $i = 0, 1, .., n$ and $k = n$. This yields

the simultaneous equations

$$\begin{aligned} y_0 &= a_0 \\ y_1 &= a_0 + (x_1 - x_0)\, a_1 \\ y_2 &= a_0 + (x_2 - x_0)\, a_1 + (x_2 - x_0)\,(x_2 - x_1)\, a_2 \\ &\vdots \\ y_n &= a_0 + (x_n - x_0)\, a_1 + (x_n - x_0)\,(x_n - x_1)\, a_2 + \ldots + (x_n - x_0)\ldots(x_n - x_{n-1})\, a_n \end{aligned} \tag{267}$$

If we write the divided differences as

$$\begin{aligned} \nabla y_i &= \frac{y_i - y_0}{x_i - x_0}, \quad i = 1, 2, \ldots, n \\ \nabla^2 y_i &= \frac{\nabla y_i - \nabla y_1}{x_i - x_1}, \quad i = 2, \ldots, n \\ \nabla^3 y_i &= \frac{\nabla^2 y_i - \nabla^2 y_2}{x_i - x_2}, \quad i = 3, \ldots, n \\ &\vdots \\ \nabla^n y_n &= \frac{\nabla^{n-1} y_i - \nabla^{n-1} y_{n-1}}{x_i - x_{n-1}} \end{aligned} \tag{268}$$

the solution of equations 267 becomes

$$a_0 = y_0, \quad a_1 = \nabla y_1, \quad a_2 = \nabla^2 y_2, \quad \ldots, \quad a_n = \nabla^n y_n \tag{269}$$

Here is an example of how to write the Newton's method in python efficiently

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

def coeffts(xData,yData):
    #Computes the coefficients of Newton polynomial.
    n = len(xData)
    # Number of data points108
    a = yData.copy()
    for k in range(1,n):
        for j in range(k,n):
            a[j] = (a[j]-a[k-1])/(xData[j] - xData[k-1])
        #or more simply
        #a[k:n] = (a[k:n] - a[k-1])/(xData[k:n] - xData[k-1])
    return a

def evalPoly(a,xData,x):
    n = len(xData) - 1
    # Degree of polynomial
    p = a[n]
    for k in range(1,n+1):
        p = a[n-k] + (x -xData[n-k])*p
    return p

#Evaluates Newton polynomial p at x. The coefficient
#vector { a } can be computed by the function coeffts.
def newtonPoly(xData,yData,x):
    a = coeffts(xData,yData)
    p = evalPoly(a,xData,x)
    return p

#main
xData=np.zeros(10,float)
yData=np.zeros(10,float)
x=np.zeros(100,float)
for i in range(len(xData)):
    xData[i]=float(i*0.5)
    yData[i]=float(xData[i]**2)
for i in range(len(x)):
    x[i]=float(i*0.5/10.)

y=newtonPoly(xData,yData,x)


fig, ax1= plt.subplots()
ax1.scatter(xData,yData, marker='o', edgecolor='black',\
 facecolors='black', s=100,zorder=1)
ax1.plot(x,y)
fig.tight_layout()
plt.show()
```

This example produces the interpolation in Figure 56.

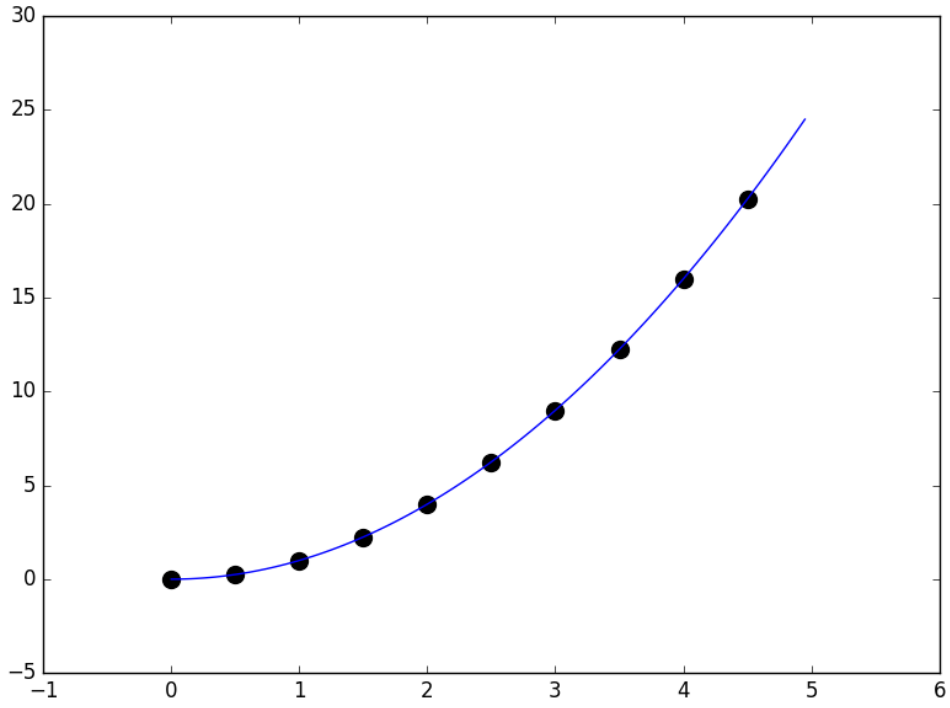


*Figure 56: Example of Newton's polynomial interpolation following the given script.*

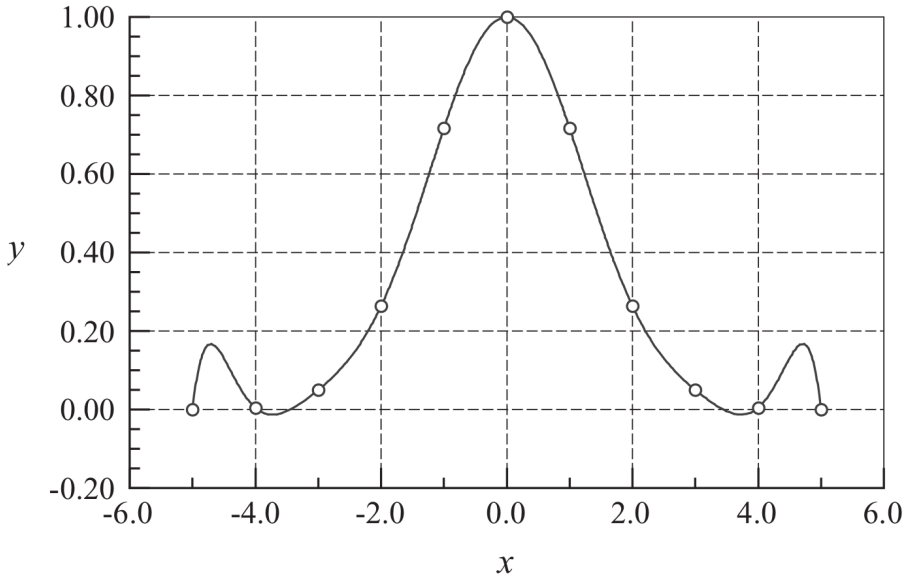


*Figure 57: Polynomial interpolant displaying oscillations.*

## 12.4 *Limitations of polynomial interpolation*

Polynomial interpolation should be carried out with the **fewest feasible number of data points**. Linear interpolation, using the nearest two points, is often sufficient if the data points are closely spaced. Three to six nearest-neighbor points produce good results in most cases. An interpolant intersecting more than six points must be viewed with suspicion. The reason is that the data points that are far from the point of interest do not contribute to the accuracy of the interpolant. As a matter of fact, they can be detrimental.

The danger of using too many points is illustrated in Figure 57. There are 11 equally spaced data points represented by the circles. The solid line is the interpolant, a polynomial of degree 10, that intersects all the points. As seen in the figure, a polynomial of such a high degree has a tendency to **oscillate** excessively between the data points. A much smoother result would be obtained by using a cubic interpolant spanning four nearest-neighbour points.

Polynomial extrapolation (i.e. interpolating outside the range of data points) is dangerous as well. As an example, Figure 58 shows a fifth-degree interpolating polynomial. The interpolant performs fine over the data, but produces a strange result when extrapolating. If extrapolation cannot be avoided, it should be done following some good practice:

- plot the data and visually verify that the extrapolated value makes sense in few test cases;
- use a low-order polynomial based on nearest-neighbour data points. A

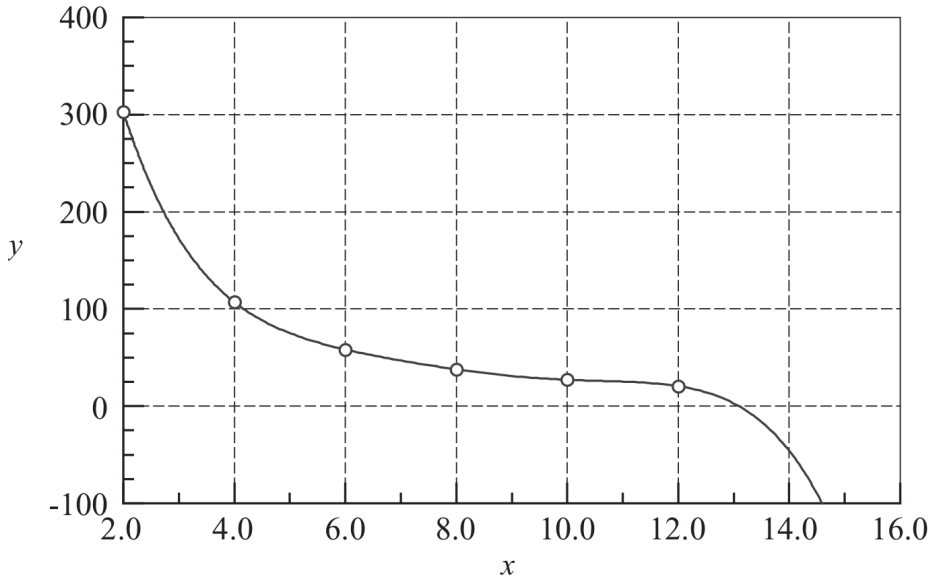


*Figure 58: Extrapolation might produce spurious trends.*

linear or quadratic interpolant should work fine;

- work with plots of log vs log, which is usually much smoother than the linear axes.

## 12.5 *Two-dimensional interpolation*

Often in astrophysics we have to interpolate not just in one dimension, but in two or more dimensions. Here, we discuss the case of **two-dimensional interpolation** focusing on linear interpolation only. When applied in two dimensions, linear interpolation is often called **bilinear interpolation**.

Suppose that we want to find the value of the unknown function $f(x, y)$ at the point $(x, y)$. It is assumed that we know the value of $f(x, y)$ at the four points Q11 = $(x1, y1)$, Q12 = $(x1, y2)$, Q21 = $(x2, y1)$, and Q22 = $(x2, y2)$, as shown in Fig. 59.

We first interpolate along the $x-$ direction. This yields

$$f(x, y1) \sim \frac{x2 - x}{x2 - x1} f(\mathrm{Q11}) + \frac{x - x1}{x2 - x1} f(\mathrm{Q21})$$
$$f(x, y2) \sim \frac{x2 - x}{x2 - x1} f(\mathrm{Q12}) + \frac{x - x1}{x2 - x1} f(\mathrm{Q22}) \qquad (270)$$

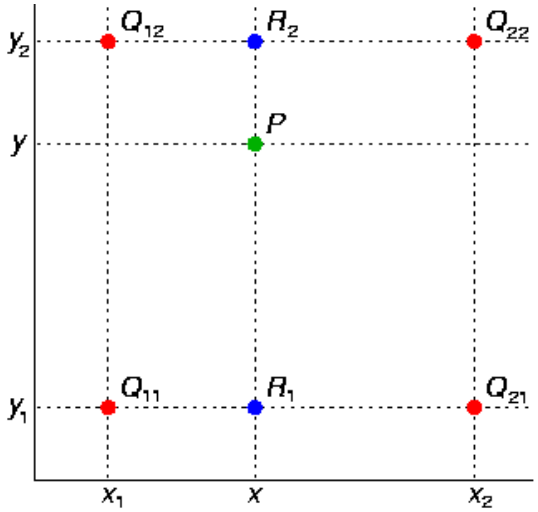


*Figure 59: Scheme for bilinear interpolation.*

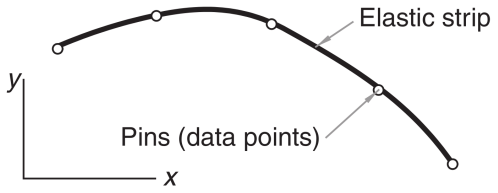


*Figure 60: Cartoon of a cubic spline, represented as an elastic beam. From Kiusalaas.*

We then interpolate along the $y-$ axis:

$$f(x,y) \sim \frac{y2-y}{y2-y1} f(x,y1) + \frac{y-y1}{y2-y1} f(x,y2)$$

$$= \frac{y2-y}{y2-y1}\left[\frac{x2-x}{x2-x1} f(\mathrm{Q11}) + \frac{x-x1}{x2-x1} f(\mathrm{Q21})\right] +$$

$$\frac{y-y1}{y2-y1}\left[\frac{x2-x}{x2-x1} f(\mathrm{Q12}) + \frac{x-x1}{x2-x1} f(\mathrm{Q22})\right]$$

$$= \frac{1}{(x2-x1)(y2-y1)}\{(y2-y)\,[(x2-x)\,f(\mathrm{Q11}) + (x-x1)\,f(\mathrm{Q21})] +$$

$$(y-y1)\,[(x2-x)\,f(\mathrm{Q12}) + (x-x1)\,f(\mathrm{Q22})]\} \quad (271)$$

We obtain the same result if we interpolate first along $y$ and then along $x$. Note that this algorithm works only if the grid points are uniformly spaced, which often is not the case.

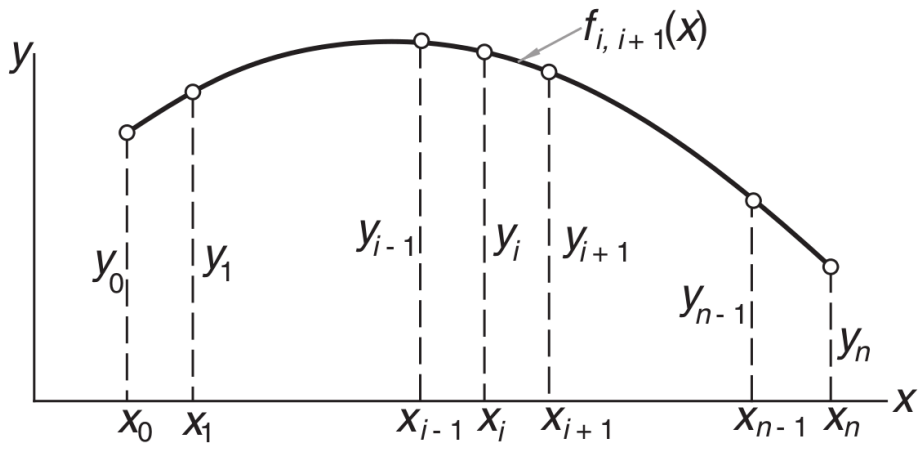


*Figure 61: Model of a cubic spline. From Kiusalaas.*

## 12.6 *Cubic spline interpolation*

The cubic spline is possibly the best interpolation method, especially if there are more than a few data points. The mechanical model of a cubic spline is a thin, **elastic beam** that is attached with pins to the data points (see Fig. 60). The pins, that is, the data points, are called the **knots** of the spline. Each segment of the spline curve is a cubic polynomial with smooth junctions: the first and second derivatives are continuous at the knots. This gives a "stiffer" behaviour than a polynomial, in the sense that the cubic spline has less tendency to oscillate between data points.

Figure 61 shows a cubic spline that spans $n + 1$ knots. We use the notation $f_{i,i+1}(x)$ for the cubic polynomial that spans the segment between knots $i$ and $i + 1$. Note that the spline is a **piecewise** cubic curve, put together from the $n$ cubics $f_{0,1}(x)$, $f_{1,2}(x)$, ..., $f_{n-1,n}(x)$, all of which have different coefficients. Denoting the second derivative of the spline at knot $i$ by $k_i$, continuity of second derivatives requires that

$$f''_{i-1,i}(x_i) = f''_{i,i+1}(x_i) = k_i. \tag{272}$$

At this stage, each $k$ is unknown, except for

$$k_0 = k_n = 0 \tag{273}$$

The starting point for computing the coefficients of $f_{i,i+1}(x)$ is the expression for $f''_{i,i+1}(x)$. Using Lagrange's two-point interpolation, we can write

$$f''_{i,i+1}(x) = k_i\, l_i(x) + k_{i+1}\, l_{i+1}(x), \tag{274}$$

where

$$l_i(x) = \frac{x - x_{i+1}}{x_i - x_{i+1}}, \quad l_{i+1}(x) = \frac{x - x_i}{x_{i+1} - x_i} \tag{275}$$

Hence

$$f''_{i,i+1}(x) = \frac{k_i\,(x - x_{i+1}) - k_{i+1}\,(x - x_i)}{x_i - x_{i+1}} \tag{276}$$

Integrating twice with respect to $x$, we obtain

$$f_{i,i+1}(x) = \frac{k_i\,(x - x_{i+1})^3 - k_{i+1}\,(x - x_i)^3}{6\,(x_i - x_{i+1})} + \mathrm{A}\,(x - x_{i+1}) - \mathrm{B}\,(x - x_i), \tag{277}$$

where A and B are constants of integration. Usually, the constants of integration are written in the form $\mathrm{C}\,x + \mathrm{D}$, but defining $\mathrm{C} = \mathrm{A} - \mathrm{B}$ and $\mathrm{D} = -\mathrm{A}\,x_{i+1} + \mathrm{B}\,x_i$ we end up with two terms that are more handy in the procedure that follows.

Simplifying the notation, we now write $y_i \equiv f_{i,i+1}(x_i)$ and $y_{i+1} \equiv f_{i,i+1}(x_{i+1})$. Therefore,

$$\begin{aligned} \frac{k_i\,(x_i - x_{i+1})^3}{6\,(x_i - x_{i+1})} + \mathrm{A}\,(x_i - x_{i+1}) &= y_i, \\ \frac{-k_{i+1}\,(x_{i+1} - x_i)^3}{6\,(x_i - x_{i+1})} - \mathrm{B}\,(x_{i+1} - x_i) &= y_{i+1}, \end{aligned} \tag{278}$$

which leads us to an expression for both A and B:

$$\begin{aligned} \mathrm{A} &= \frac{y_i}{(x_i - x_{i+1})} - \frac{k_i}{6}\,(x_i - x_{i+1}), \\ \mathrm{B} &= \frac{y_{i+1}}{(x_i - x_{i+1})} - \frac{k_{i+1}}{6}(x_i - x_{i+1}) \end{aligned} \tag{279}$$

Substituting back the values of A and B into equation 277 and rearranging, we find:

$$\begin{aligned} f_{i,i+1}(x) = \frac{k_i}{6}\left[\frac{(x - x_{i+1})^3}{(x_i - x_{i+1})} - (x_i - x_{i+1})\,(x - x_{i+1})\right] \\ -\frac{k_{i+1}}{6}\left[\frac{(x - x_i)^3}{(x_i - x_{i+1})} - (x_i - x_{i+1})\,(x - x_i)\right] \\ + \frac{y_i\,(x - x_{i+1}) - y_{i+1}\,(x - x_i)}{(x_i - x_{i+1})} \end{aligned} \tag{280}$$

The only missing terms in equation 280 are the second derivatives at the knots, $k_i$ and $k_{i+1}$. They can be obtained from the continuity condition on the slope, i.e. $f'_{i-1,i}(x_i) = f'_{i,i+1}(x_i)$. Calculating the derivative of eq. 280 in $i - 1, i$ and

$i, i+1$ we obtain

$$k_{i-1}\,(x_{i-1}-x_i)+2\,k_i\,(x_{i-1}-x_{i+1})+k_{i+1}\,(x_i-x_{i+1})=6\left(\frac{y_{i-1}-y_i}{x_{i-1}-x_i}-\frac{y_i-y_{i+1}}{x_i-x_{i+1}}\right) \tag{281}$$

for $i = 1, 2, ..., n-1$. This gives a system of linear equations that can be solved with one of the methods you already know, to get the $k_i$ terms. The simplest way to solve it is to observe that the matrix is tridiagonal, i.e. it has non zero elements only on the diagonal (terms $i, i$) and on the cells $i, i-1$ and $i, i+1$, that is

$$\begin{bmatrix} d_0 & e_0 & 0 & 0 & 0 & 0 \\ c_0 & d_1 & e_1 & 0 & 0 & 0 \\ 0 & c_1 & d_2 & e_2 & 0 & 0 \\ 0 & 0 & c_2 & d_3 & e_3 & 0 \\ 0 & 0 & 0 & c_3 & d_4 & e_4 \\ 0 & 0 & 0 & 0 & c_4 & d_5 \end{bmatrix}$$

Namely, our system is:

$$\begin{bmatrix} 2\,(x_0-x_2) & (x_1-x_2) & 0 & 0 & 0 & 0 \\ (x_1-x_2) & 2\,(x_1-x_3) & (x_2-x_3) & 0 & 0 & 0 \\ 0 & (x_2-x_3) & 2\,(x_2-x_4) & (x_3-x_4) & 0 & 0 \\ 0 & 0 & (x_3-x_4) & 2\,(x_3-x_5) & (x_4-x_5) & 0 \\ 0 & 0 & 0 & (x_4-x_5) & 2\,(x_4-x_6) & (x_5-x_6) \\ 0 & 0 & 0 & 0 & (x_5-x_6) & 2\,(x_5-x_7) \end{bmatrix} \begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \\ k_5 \\ k_6 \end{bmatrix} =$$

$$\begin{bmatrix} 6\left(\frac{y_0-y_1}{x_0-x_1}-\frac{y_1-y_2}{x_1-x_2}\right) \\ 6\left(\frac{y_1-y_2}{x_1-x_2}-\frac{y_2-y_3}{x_2-x_3}\right) \\ 6\left(\frac{y_2-y_3}{x_2-x_3}-\frac{y_3-y_4}{x_3-x_4}\right) \\ 6\left(\frac{y_3-y_4}{x_3-x_4}-\frac{y_4-y_5}{x_4-x_5}\right) \\ 6\left(\frac{y_4-y_5}{x_4-x_5}-\frac{y_5-y_6}{x_5-x_6}\right) \\ 6\left(\frac{y_5-y_6}{x_5-x_6}-\frac{y_6-y_7}{x_6-x_7}\right) \end{bmatrix} \tag{282}$$

Tridiagonal matrices are particularly easy to solve with the LU decomposition (see example below).

If the data are evenly spaced, i.e. $x_i - x_{i-1} = x_{i+1} - x_i = h$, equation 281 further simplifies to

$$k_{i-1}+4\,k_i+k_{i+1}=\frac{6}{h^2}\,(y_{i-1}-2\,y_i+y_{i+1})\,. \tag{283}$$

For the implementation of the cubic spline, we have to follow these steps:

1. Calculate the $k_i$ by solving the system of linear equations (remember that $k_0 = k_n = 0$ and you only have to calculate $k_i$ with $i = 1, 2, ..., n-1$);

2. Calculate the interpolant at $x$ from Eq. 280. Repeat for all the $x$ points you want;

**EXERCISE:**

Write a script to implement the cubic spline. To calculate the coefficients $k_i$ use the LU decomposition through numpy.linalg.solve. Apply it to the following sample of fake data.

```
import numpy as np
N=18
x=np.arange(0,18,1.)
y=np.sin(x)
```

The result should look as the upper panel of Fig. 62.
Now, apply this script to the file evol_120msun_scattered.dat and reconstruct the evolution of the 120 $M_\odot$ star with the spline. The result should look as the lower panel of Fig. 62.

## 12.7 *Python modules to interpolate*

- The function **numpy.interp** returns the one-dimensional piecewise linear interpolant to a function with given discrete data points $(xp, yp)$, evaluated at $x$. It is analogous to our linear interpolation script. Reliable, robust, highly recommended. Syntax:

```
y=numpy.interp(x,xp,yp)
```

- The module **scipy.interpolate** contains several different functions for interpolation. Let us just mention **scipy.interpolate.CubicSpline** that interpolates using a cubic spline, i.e. interpolates data with a piecewise cubic polynomial which is twice continuously differentiable. Example of usage of CubicSpline

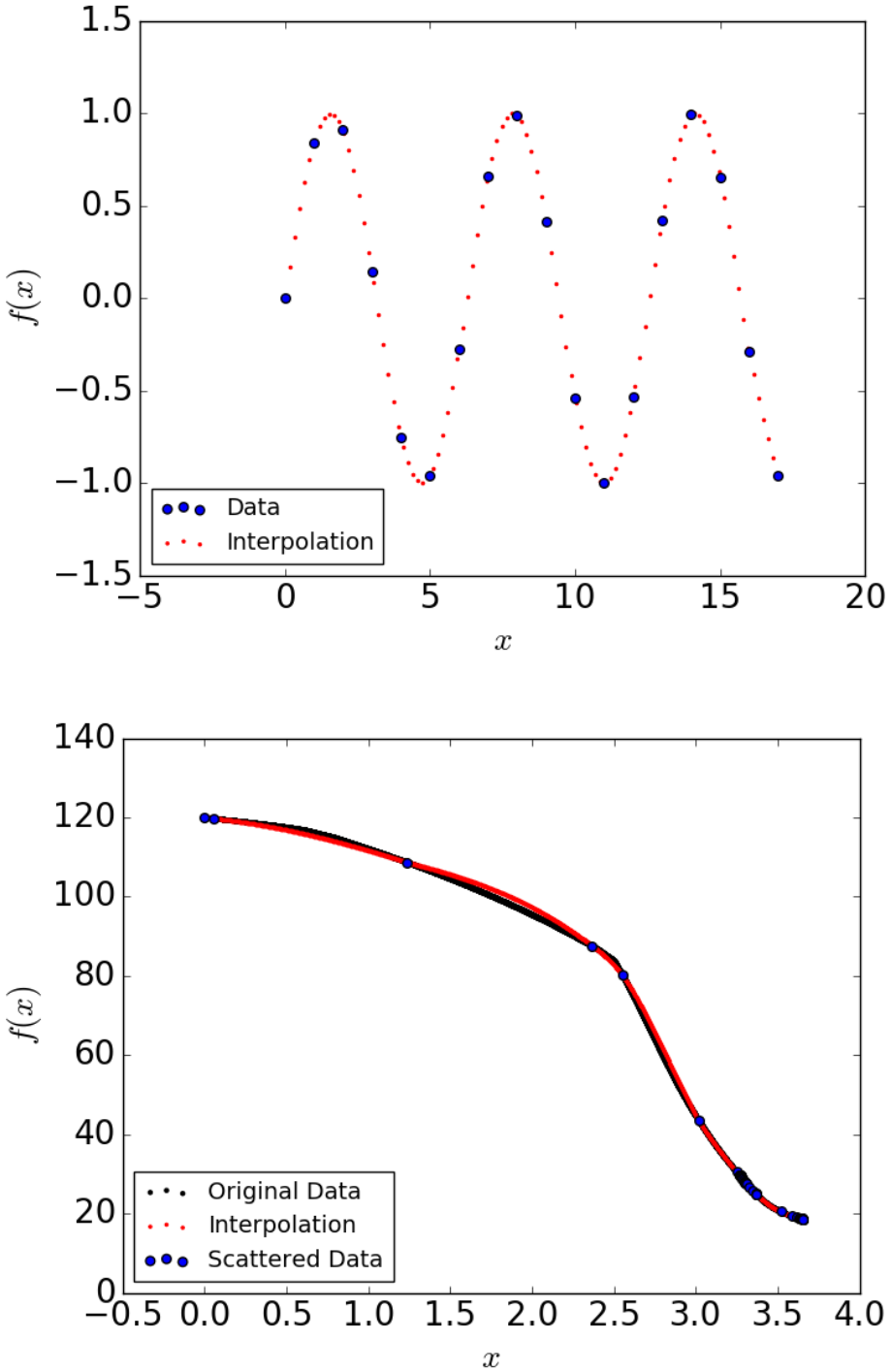


*Figure 62: Results of the cubic spline exercise. Top: sinusoidal set of data. Bottom: evolution of a 120 $M_\odot$ star.*

```
from scipy.interpolate import CubicSpline
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(10)

y = np.sin(x)

cs = CubicSpline(x, y) # x is the array of data,
#y is the array of values of the function in these data

xs = np.arange(-0.5, 9.6, 0.1)

fig, ax = plt.subplots(figsize=(6.5, 4))

ax.plot(x, y, 'o', label='data') #scattered data

ax.plot(xs, np.sin(xs), label='true') #true value of the function

ax.plot(xs, cs(xs), label="f") #interpolated value of the function

ax.plot(xs, cs(xs, 1), label="f'") #first derivative of f

ax.plot(xs, cs(xs, 2), label="f''")  #second derivative of f

ax.plot(xs, cs(xs, 3), label="f'''")  #third derivative of f

ax.set_xlim(-0.5, 9.5)

ax.legend(loc='lower left', ncol=2)

plt.show()
```

which produces the result in Figure 63.

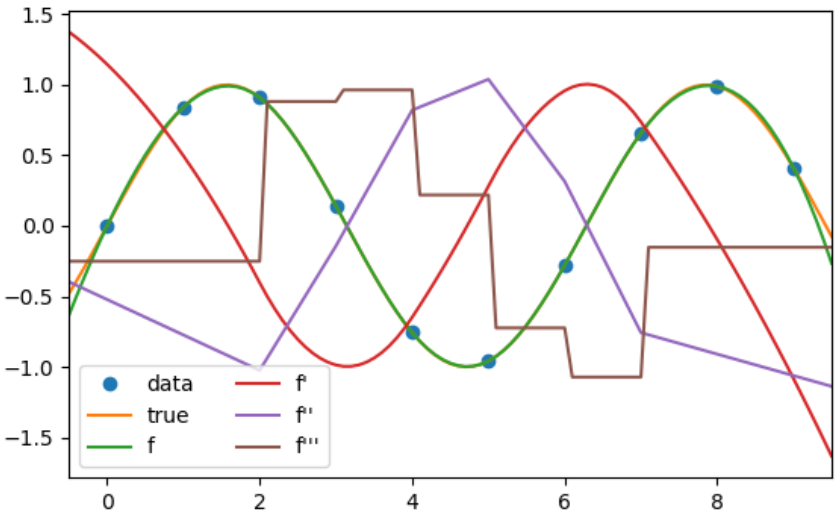


*Figure 63: Example of the function scipy.interpolate.CubicSpline.*

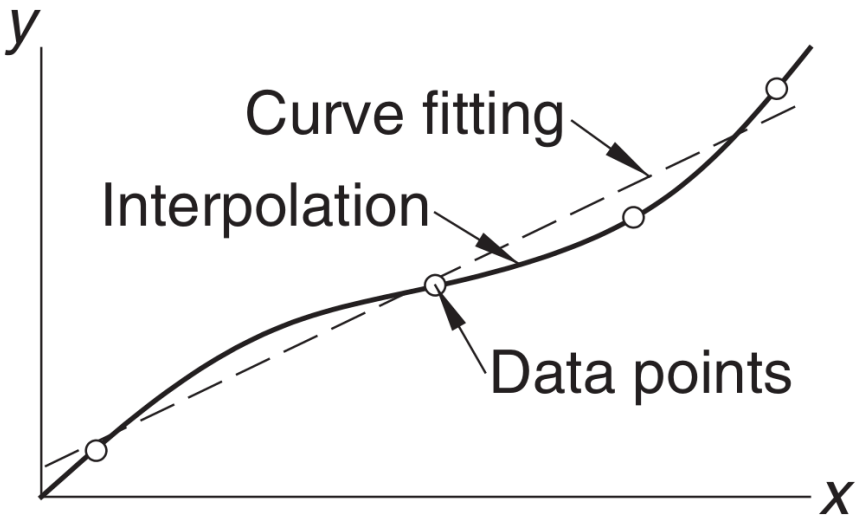


*Figure 64: Visualization of the difference between interpolation and curve fitting. Figure from Kiusalaas.*

## 13 FITTING PROCEDURES

This chapter is based on *Numerical Methods in Engineering with Python* by Jaan Kiusalaas.

We start this Chapter by quoting the beginning of previous chapter (important concept!):

"Interpolation and curve fitting should not be confused. In interpolation we construct a curve passing *through* a discrete set of data points. In doing so, we make the implicit assumption that the data points are accurate and distinct.

Curve fitting is applied to data that contain scatter (noise), usually due to measurement errors. In this case, we want to find a smooth curve that approximates the data. Thus, the curve does not necessarily hit the data points. Figure 64 visualizes the difference between these two approaches. In practice, interpolation is used more often when we deal with theoretical data points (e.g. the outcome of discrete simulations, discrete numerical models), curve fitting is mostly done on a set of observational data."

In this chapter we show some examples of curve fitting, starting from the simpler ones.

## 13.1 *Linear fit with the least squares*

The simplest approach is to assume that our data points can be fit with a **straight line**. For example, let $y_i$ be our set of measurements in the points $x_i$, where $i = 0, 1, .., \mathrm{N} - 1$. For example, the $y_i$ could be measurements of the flux of an astrophysical source at different times $x_i$.

Assuming that the $y_i$ and the $x_i$ are connected by a linear relationship is equivalent to say that, if we could measure $y_i$ with infinite accuracy and avoiding any kind of noise, the $y_i$ would perfectly obey the following relationship

$$y = \mathrm{A} + \mathrm{B}\, x, \tag{284}$$

where A and B are constant and the values $y$ and $x$ represent the "continuous version of $y_i$ and $x_i$.

Note that often $y_i$ and $x_i$ are not our data but the **base-10 logarithms of our data**. This is because many quantities in nature can be represented by **power laws**

$$\tilde{y} = 10^{\mathrm{A}}\, \tilde{x}^{\mathrm{B}}, \tag{285}$$

where $\tilde{y} \equiv 10^{y}$, $\tilde{x} \equiv 10^{x}$.

Of course, it will never be possible to measure the $y_i$ with infinite accuracy and avoiding any kind of noise. Hence, first of all we have to make an assumption about the underlying distribution of the errors of measure or, if we are lucky, we have an independent way to estimate such distribution. Very often, if we cannot do better, we assume that the errors of measure on the $y_i$ follow a **Gaussian distribution**, even if we cannot probe it.

Note that here we are also assuming that there is **no error of measurement on the** $x_i$ and we know them perfectly. This is always false for obvious reasons, but in general we can choose the $x_i$ so that the error of measurement on the $x_i$ is much smaller than the one on the $y_i$ and can be neglected. For example, in the case of the measurement of the flux $y_i$ from a source at time $x_i$, usually our measure of time is way more accurate than the measure of fluxes. If the uncertainties on the $x_i$ are comparable to the uncertainties on the $y_i$, the method we are about to discuss technically fails.

If the errors on $y_i$ are Gaussian distributed, then the measurement of $y_i$ is governed by a normal distribution centered on the true value $\mathrm{A} + \mathrm{B}\, x$ with standard deviation $\sigma_y$. Therefore, the probability of obtaining the observed value of $y_i$ is

$$\mathrm{P}_{\mathrm{A,B}}(y_i) \propto \frac{1}{\sigma_y}\, \exp - \frac{(y_i - \mathrm{A} - \mathrm{B}\, x_i)^2}{2\, \sigma_y^2} \tag{286}$$

The probability of obtaining our complete set of measurements $y_0, y_1, .., y_{\mathrm{N}-1}$ is then

$$P_{A,B}(y_0, y_1, .., y_{N-1}) = P_{A,B}(y_0)\, P_{A,B}(y_1) .. P_{A,B}(y_{N-1}) \propto \frac{1}{\sigma_y^N} \exp(-\chi^2/2), \quad (287)$$

where

$$\chi^2 = \sum_{i=0}^{N-1} \frac{(y_i - A - B\, x_i)^2}{\sigma_y^2}. \quad (288)$$

Now we can derive A and B from our set of data, as the values of A and B for which the probability 287 is maximum, i.e. the ones for which the $\chi^2$ is minimum. For this reason, this method is called **least squares fitting**.

To find A and B we differentiate $\chi^2$ with respect to A and B and set the derivatives equal to zero:

$$\frac{\partial \chi^2}{\partial A} = -\frac{2}{\sigma_y^2} \sum_{i=0}^{N-1} (y_i - A - B\, x_i) = 0$$

$$\frac{\partial \chi^2}{\partial B} = -\frac{2}{\sigma_y^2} \sum_{i=0}^{N-1} x_i\, (y_i - A - B\, x_i) = 0, \quad (289)$$

which can be simply rewritten as

$$A\, N + B \sum x_i = \sum y_i$$

$$A \sum x_i + B \sum x_i^2 = \sum x_i\, y_i, \quad (290)$$

where $\sum \equiv \sum_{i=0}^{N-1}$ for simplicity. After few steps, we can find that the solutions of the system 290 are

$$A = \frac{\sum x_i^2 \sum y_i - \sum x_i \sum x_i\, y_i}{\Delta}$$

$$B = \frac{N \sum x_i\, y_i - \sum x_i \sum y_i}{\Delta}, \quad (291)$$

where $\Delta \equiv N \sum x_i^2 - (\sum x_i)^2$.

Numerically, equations 291 for A and B are subject to large rounding errors, because the two terms of the subtraction in the numerators and in the denominators of both equations are similar numbers (in Chapter 4, we have seen that the subtraction has large rounding errors if it is done between two numbers that are both very large and very similar to each other). In your script, it is then better to use the following equivalent equations, which are

designed to reduce rounding errors.

$$\mathrm{B} = \frac{\sum y_i \ (x_i - \langle x \rangle)}{\sum x_i \ (x_i - \langle x \rangle)}$$
$$\mathrm{A} = \langle y \rangle - \langle x \rangle \, \mathrm{B}, \tag{292}$$

where

$$\langle x \rangle = \frac{\sum x_i}{\mathrm{N}}$$
$$\langle y \rangle = \frac{\sum y_i}{\mathrm{N}} \tag{293}$$

are the **mean values** of the $x_i$ and $y_i$ data.

Following the same line of reasoning as before, we can estimate the uncertainty on each $y_i$. Since we assumed that the errors of measurement on $y_i$ are normally distributed about the true value $\mathrm{A} + \mathrm{B}\, x_i$, the deviations $y_i - \mathrm{A} - \mathrm{B}\, x_i$ are also normally distributed, with a central value = 0 and the same standard deviation $\sigma_y$. Hence

$$\sigma_y = \sqrt{\frac{1}{(\mathrm{N}-1)} \sum_{i=0}^{\mathrm{N}-1} (y_i - \mathrm{A} - \mathrm{B}\, x_i)^2}, \tag{294}$$

where we have written N – 1 instead of N because the two constants A and B reduce the number of degrees of freedom of our system. The above equation yields the uncertainty on each single measurement.

Now we want to determine the uncertainty on our fitting constants, A and B. By simply performing error propagation of equations 291 in terms of the errors on $y_0, y_1, .., y_{\mathrm{N}-1}$ we find:

$$\sigma_{\mathrm{A}} = \sigma_y \sqrt{\frac{\sum x_i^2}{\Delta}}$$
$$\sigma_{\mathrm{B}} = \sigma_y \sqrt{\frac{\mathrm{N}}{\Delta}}. \tag{295}$$

The bigger the uncertainties $\sigma_{\mathrm{A}}$ and $\sigma_{\mathrm{B}}$, the worse is our assumption that the $y_i$ and the $x_i$ are connected by a linear relationship.

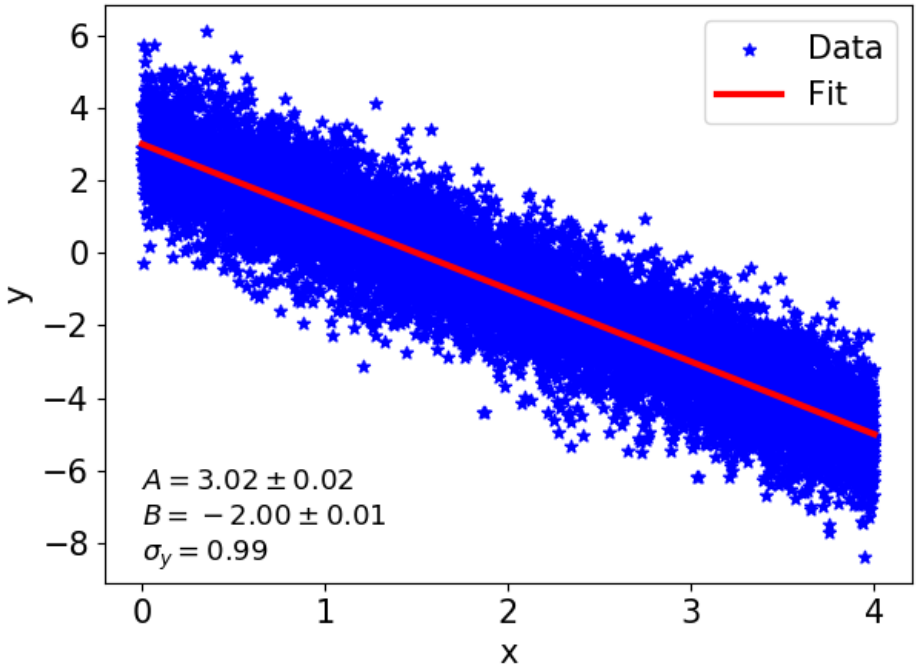


*Figure 65: Fit of the fake data set proposed in the exercise with the least squares method.*

**EXERCISE:**

Write a script to calculate the least square fit using the formulas 292 for A and B and the formulas 294 and 295 for the uncertainties. Construct your "fake" set of data as follows

```
def data_set():
    x=np.logspace(0,4,num=10000) #x data
    b=-2.0
    a=3.0
    y=10.**a * x**b #intrinsic relation y=10**a * x**b
    x=np.log10(x)
    y=np.log10(y)

    sigma=1.
    y=rnd.normal(y,sigma) #add gaussian noise
    return x,y
```

Use your script to fit the fake data. Your results should look like Figure 65.

## 13.2 *Linear fit by weighting of data*

The previous application of the least squares method does not include the possibility that the uncertainty is different for each single datum. The most common case is that the uncertainty does depend on the data. In this case, we might want to assign a **weight** to each single data point in the form

$$\chi^2 = \sum_{i=0}^{N-1} W_i \, (y_i - A - B \, x_i)^2, \tag{296}$$

where $W_i$ is the weight of the $i$th datum. From the comparison with equation 288, we see that usually $W_i \equiv 1/\sigma_{y_i}^2$, if we know $\sigma_{y_i}$.

The conditions for minimizing $\chi^2$ in this case are:

$$\frac{\partial \chi^2}{\partial A} = -2 \sum_{i=0}^{N-1} W_i \, (y_i - A - B \, x_i) = 0$$

$$\frac{\partial \chi^2}{\partial B} = -2 \sum_{i=0}^{N-1} W_i \, x_i \, (y_i - A - B \, x_i) = 0, \tag{297}$$

which finally yields

$$A = \frac{\sum W_i \, x_i^2 \, \sum W_i \, y_i - \sum W_i \, x_i \, \sum W_i \, x_i \, y_i}{\Delta}$$

$$B = \frac{\sum W_i \, \sum W_i \, x_i \, y_i - \sum W_i \, x_i \, \sum W_i \, y_i}{\Delta}, \tag{298}$$

where $\Delta \equiv \sum W_i \, \sum W_i \, x_i^2 - (\sum W_i \, x_i)^2$.

As we already noticed for the non-weighted least-square method, the above formulas for A and B can be affected by large rounding errors. Hence, we can re-write them in an equivalent, but more numerical code-friendly way as:

$$B = \frac{\sum W_i \, y_i \, (x_i - \langle x \rangle_w)}{\sum W_i \, x_i \, (x_i - \langle x \rangle_w)}$$

$$A = \langle y \rangle_w - B \, \langle x \rangle_w, \tag{299}$$

where

$$\langle x \rangle_w = \frac{\sum W_i \, x_i}{\sum W_i}$$

$$\langle y \rangle_w = \frac{\sum W_i \, y_i}{\sum W_i} \tag{300}$$

are the weighted averages of the $x_i$ and $y_i$ data.

The associated error (standard deviation) becomes then

$$\sigma_y = \sqrt{\frac{1}{(N-1)} \sum_{i=0}^{N-1} W_i \, (y_i - A - B \, x_i)^2}. \tag{301}$$

The resulting error on A and B can be found to be

$$\sigma_A = \sigma_y \sqrt{\frac{\Sigma \, W_i \, x_i^2}{\Delta_W}}$$

$$\sigma_B = \sigma_y \sqrt{\frac{\Sigma \, W_i}{\Delta_W}}, \tag{302}$$

with

$$\Delta_W \equiv \Sigma \, W_i \, \Sigma \, W_i \, x_i^2 - (\Sigma \, W_i \, x_i)^2 \, . \tag{303}$$

**EXERCISE:**

Redo the previous exercise with weights on the errors (equations 299). Considering the following set up:

```
x=np.logspace(0.,4,num=10000) #x data
b=-2.0
a=3.0
y=10.**a * x**b #intrinsic relation y=10**a * x**b
x=np.log10(x)
y=np.log10(y)

sigma=np.zeros(len(x),float)
for i in range(len(y)):
    sigma[i]=10.*rnd.random()
    y[i]=rnd.normal(y[i],sigma[i])
w=1./sigma**2
return x,y,w
```

You will find that the result is as accurate as before (or even better) with a much larger apparent scatter of the data. Expected errors on A and B of the order of $\mathcal{O}(10^{-3} - 10^{-2})$.

## 13.3 *General approach to least-square fitting*

So far, we have seen only the application of the least-square fit method to a straight line. Let us now generalize to a general function $f(x)$.

In most cases, the right approach to follow is to first assume that the data can be fitted with a straight line. If the fit we obtain with the straight line is

poor, then let us start considering some more complex functional form.

In some cases, we have some theoretical understanding of how the distribution of data should be. In that specific case, it is smart to start fitting the data with the function suggested by theory.

If, for the former or the latter reason, we have to consider a more complex function than a straight line, it is important that this function has a simple form (for example a polynomial) to avoid **overfitting** the data, i.e. to avoid that we start modeling not only the underlying distribution of the data but also the noise. Let's define this general function as

$$f(x) = f(x; a_0, a_1, ..., a_m), \tag{304}$$

where $x$ is the variable and $a_0, a_1, ..., a_m$ are the $m + 1$ parameters. If we want to fit this function to our data $(x_i, y_i)$ with $i = 0, 1, ..., \mathrm{N} - 1$, it is important that $(m + 1) < \mathrm{N}$ (better if $(m + 1) \ll \mathrm{N}$).

In this general case, the least-square fit is the one which minimizes the function

$$\mathrm{S}(a_0, a_1, .., a_m) = \sum_{i=0}^{\mathrm{N}-1} [y_i - f(x_i)]^2 \tag{305}$$

with respect to each $a_k$. Therefore, the optimal values of the parameters are given by the solution of the equations

$$\frac{\partial \mathrm{S}}{\partial a_k} = 0, \quad \forall k = 0, 1, ..., m \tag{306}$$

The above equation is clearly the generalization of equation 289 to a number $m + 1$ of parameters. Following this line of reasoning, the spread of the data about the fitting curve is quantified by the standard deviation, defined as

$$\sigma_y = \sqrt{\frac{\mathrm{S}}{\mathrm{N} - m}}, \tag{307}$$

where $\mathrm{N} - m$ is the number of degrees of freedom of the fit. In the linear case, we have two parameters A and B ($m + 1 = 2$), so the number of degrees of freedom is always $\mathrm{N} - 1$. Note that if $\mathrm{N} = m$ we have interpolation, not curve fitting. In this case both S and $\mathrm{N} - m$ are equal to zero, so $\sigma_y$ is indeterminate.

## 13.4 *Polynomial fitting*

Often the fitting function $f(x)$ is a **linear combination of functions**:

$$f(x) = a_0\, f_0(x) + a_1\, f_1(x) + ... + a_m\, f_m(x) \tag{308}$$

so that the equations 306 are linear. If the fitting function is a **polynomial**, then

$$\begin{aligned} f_0(x) &= 1 \\ f_1(x) &= x \\ f_2(x) &= x^2 \\ &... \\ f_m(x) &= x^m \end{aligned} \tag{309}$$

Here below, we discuss polynomial fitting, because it has a nice functional form and implementation. If the degree of the polynomial is $m$, then we have $f(x) = \sum_{j=0}^{m} a_j\, x^j$. Hence, the basis functions are

$$f_j(x) = a_j\, x^j \quad (j = 0, 1, 2, ..., m) \tag{310}$$

Equation 306 becomes

$$\frac{\partial \mathrm{S}}{\partial a_k} = -2 \left[ \sum_{i=0}^{\mathrm{N}-1} \left( y_i - \sum_{j=0}^{m} a_j\, x_i^j \right) x_i^k \right] = 0 \quad (k = 0, 1, ..., m) \tag{311}$$

Separating the terms that depend on $j$ from those who do not, and reshuffling a bit, we get:

$$\sum_{j=0}^{m} \left( \sum_{i=0}^{\mathrm{N}-1} x_i^j\, x_i^k \right) a_j = \sum_{i=0}^{\mathrm{N}-1} y_i\, x_i^k \tag{312}$$

Since the $a_j$ are our unknowns and we have a linear combination of the terms in $a_j$, the above equation can be rewritten as a **system of linear equations** $\mathrm{A}_{jk}\, a_j = b_k$, where

$$\begin{aligned} \mathrm{A}_{jk} &\equiv \sum_{i=0}^{\mathrm{N}-1} x_i^{j+k} \\ b_k &\equiv \sum_{i=0}^{\mathrm{N}-1} x_i^k\, y_i. \end{aligned} \tag{313}$$

The equations 313 can be implemented to perform our polynomial fit of degree $m$.

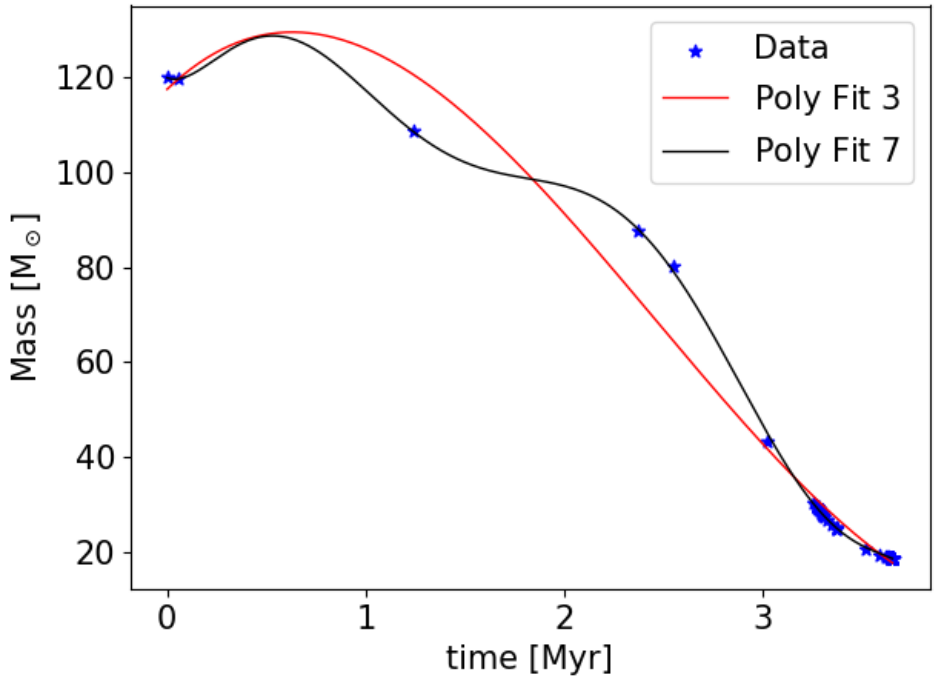


*Figure 66: Polynomial fit of the stellar track with $m = 3$ and $m = 7$.*

**EXERCISE:**

Write a script to perform a polynomial fit using equations 313. For the solution of the system of linear equations, use the method you prefer (I used LU decomposition).
Apply this method to fit the data of the file evol_120msun_scattered.dat (the stellar track of a 120 $M_\odot$ star) with a polynomial of the 3rd order. Repeat for a polynomial of 7th order. You should obtain something similar to Figure 66.

## 13.5 *Python functions for fits*

There are several possible functions in python that do least-square fitting, but possibly the most reliable are **scipy.optimize.least_squares** and **scipy.optimize.curve_fit**.

### 13.5.1 scipy.optimize.least_squares

**scipy.optimize.least_squares**: solves a general least-squares problem given the residuals $r(x)$.

I must start providing a functional form $r(x)$ of the residuals. For example, if I think may data are distributed according to a Gaussian function (e.g. I am

looking at an emission line in a spectrum), the residuals should be estimated as

$$f(x_i) = a_2 \exp\left[-(x_i - a_0)^2/(2. * a_1^2)\right]$$
$$r(x_i) = f(x_i) - y_i, \tag{314}$$

where $f(x_i)$ is a Gaussian function evaluated in $x_i$, $y_i$ is the value of the measure at $x_i$, and the three parameters $a_0$, $a_1$ and $a_2$ are the mean, standard deviation and normalization (note: here we are not talking of probability distribution functions, so $a_3$ can be whatever normalization).

Or, in python words:

```
def fun_res(a,x,y):
  gauss_res=a[2]*np.exp(-(x-a[0])**2/2./a[1]**2)-y
  return gauss_res
```

where vector $a$ contains the parameters of the function, while $(x,y)$ is a data point (in my example, the lambda and the intensity of the line at that lambda). Of course, the vector $a$ contains the parameters that I still do not know and I want to estimate, but I can start by providing a guess for them.

After I have defined fun_res, I can call least_squares as

```
from scipy import optimize as opt

lsq=opt.least_squares(fun_res,a,args=(x,y),xtol=1e-07,loss='cauchy')
```

where I call fun_res first (note that I must not specify the arguments of the function), then I provide my Ansatz for the parameters a (which should be defined somewhere before), then I input my whole array of data $(x, y)$. In addition, I have several other options like setting the tolerance xtol, or the loss function $g(z)$, which is used to "modulate" the cost function F$(x)$ we want to minimize:

$$\mathrm{F}(x) = 0.5\ \mathrm{sum}(g(r(x)^2)) \tag{315}$$

where $r(x)$ is the residual I defined before.

If I choose "loss=linear" (default), $g(r(x)^2) = r(x)^2$, which yields the standard definition of the least-squares problem in the previous pages. For example, if I choose "loss=cauchy', $g(r(x)^2) = \ln\left[1 + r(x)^2\right]$, which weakens the influence of outliers in the data (bad measurements) but might slow down the optimization (or even make it fail).

The output of opt.least_squares() is a quite complicated structure of data, defined in scipy. Here below, I summarize the most important outputs and how you can get them:

```
lsq.x
```

is the array of the $a_j$ after the optimization. In the case of the Gaussian fit, $a_0$, $a_1$ and $a_2$ will contain the best fit values of the mean, the standard deviation and the normalization.

```
lsq.success
```

The answer can be only True or False. False means that the fit failed, True that it succeeded.

**EXERCISE:**

Write a script to perform a least-square fit using **scipy.optimize.least_squares**. Apply it to this set of mock data

```
def data_set():
    a=0.0
    s=2.0
    h=1.0
    x=np.linspace(-5.,5.,num=500) #x data
    y=h*np.exp(-(x-a)**2/(2.*s**2))


    sigma=np.zeros(len(x),float)
    for i in range(len(y)):
        sigma[i]=0.1*rnd.random()
        y[i]=rnd.normal(y[i],sigma[i]) #add gaussian noise

    return x,y
```

Of course, these mock data correspond to a Gaussian with mean, sigma and height equal to 0, 2 and 1, respectively, and with some Gaussian noise. Plot the results. They should look as Figure 67

### 13.5.2 scipy.optimize.curve_fit

**scipy.optimize.curve_fit** solves a general least-squares problem to fit a function, $f(x)$, to the data. The main (but not the only) difference with respect to scipy.optimize.least_squares is that scipy.optimize.curve_fit requires the function in input, not the residuals.

For example, if I think may data are distributed according to a Gaussian function, the function can be defined as:

```
def func(x,a,s,h):
    gauss=h*np.exp(-(x-a)**2/2./s**2)
    return gauss
```

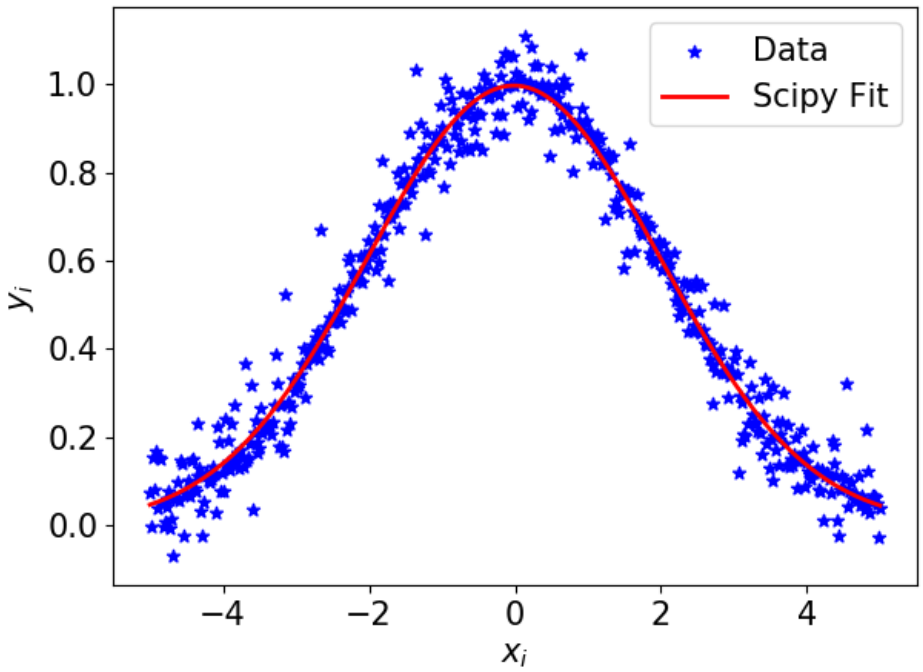


*Figure 67: Least-square fit with a Gaussian using scipi.optimize.least_squares.*

where a, s and h are the mean, the standard deviation and the height of the Gaussian.

After I have defined func, I can call curve_fit as

```
from scipy import optimize as opt

popt,pcov=opt.curve_fit(func,x,y,p0=(a,s,h))
```

the minimal arguments for curve_fit are the input function func (note that I must not specify the arguments of the function), the data set $x$, the data set $y$, the array $p0$ which contains the parameters needed by func, over which we want to do the optimization (in this case a, s and h).

Other additional possible input parameters are **sigma**: the array of uncertainties on the data $y$ (if we know it); **bounds**: the lower and upper bounds on parameters (default is no bounds); **methods**: the method used by the function to do the optimization.

The output of optimize.curve_fit is composed of two arrays:

popt: contains the optimal values for the parameters so that the sum of the squared residuals of $f(x, *popt) - y$ is minimized.

pcov: is the estimated covariance of popt. The diagonals provide the variance of the parameter estimate. To compute one standard deviation errors on

the parameters use perr = np.sqrt(np.diag(pcov)).

**EXERCISE:**

Write a script to perform a least-square fit using **scipy.optimize.curve_fit**. Apply it to this set of mock data

```
def data_set():
    a=0.
    s=1.
    h=10.

    x=np.linspace(-10.,10.,num=500) #x data
    y=h*np.exp(-(x-a)**2/2./s**2)

    sigma=np.zeros(len(x),float)
    for i in range(len(y)):
        sigma[i]=4.*rnd.random()
        y[i]=rnd.normal(y[i],sigma[i]) #add gaussian noise

    return x,y
```

Of course, these mock data correspond to a Gaussian with mean, sigma and height equal to 0, 2 and 1, respectively, and with some Gaussian noise. Plot the results. They should look like Figure 68

Note that **optimize.curve_fit** can be also used to do fits in multi-dimensions. For example, let us see now the example of a Gaussian in two dimensions. This is quite useful in observational astrophysics, because several point spread functions can be approximated with 2D Gaussians.

Here below, you find an example of this fit.

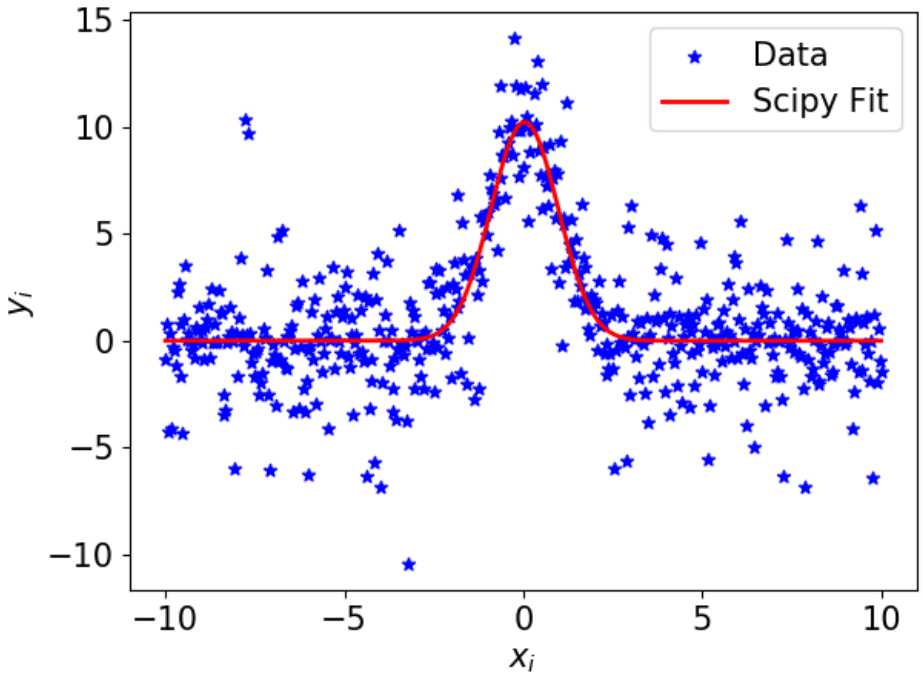


*Figure 68: Least-square fit with a Gaussian using scipi.optimize.curve_fit. In this case, the parameters with errors are:* $a = -0.03 \pm 0.05$, $s = 0.94 \pm 0.05$, $h = 10.0 \pm 0.4$.

```
#examples/fit/fit_2D_scipy.py
#fit a 2D gaussian with scipy.optimize.curve_fit
import numpy as np
import numpy.random as rnd
import matplotlib.pyplot as plt
from scipy import optimize as opt
from scipy.stats import norm
plt.rcParams.update({'font.size': 15})

def twoD_gauss(xy,x0,y0,sx,sy,h):
    z=h*np.exp(-((xy[0]-x0)**2/(2.*sx*sx)+(xy[1]-y0)**2/(2.*sy*sy)))
    return z.ravel()

def data_set():
    x=np.linspace(-5.,5.,num=500) #x data
    y=np.linspace(-5.,5.,num=500) #x data
    x,y=np.meshgrid(x,y)
    z=np.zeros([len(x),len(y)],float)
    x0=0.0
    y0=0.0
    sx=1.0
    sy=0.7
    h=1.0
    xy=(x,y)
    z=twoD_gauss(xy,x0,y0,sx,sy,h)
    z=z + 1e-1*np.random.normal(size=z.shape)
    return x,y,z

def sci_fit(x,y,x0,y0,sx,sy,h):
    xy=(x,y)
    popt,pcov=opt.curve_fit(twoD_gauss,xy,z,p0=(x0,y0,sx,sy,h))
    return popt,pcov

#main
x,y,z=data_set() #data points
x0=0. #Ansatz for parameters
y0=1.
sx=2.
sy=2.
h=2.
popt,pcov=sci_fit(x,y,x0,y0,sx,sy,h)
perr = np.sqrt(np.diag(pcov))

print("x0= ", popt[0], " err x0= ", perr[0])#x0, err on x0
print("y0= ",popt[1], " err y0= ",perr[1]) #y0, err on y0
print("sx= ",popt[2], " err sx= ",perr[3]) #sx, err on sx
print("sy= ",popt[3], " err sy= ",perr[3]) #sy, err on sy
print("h= ",popt[4]," err h= ",perr[4]) #h, err on h

xy=(x,y)
data_fitted = twoD_gauss(xy, *popt) #fitted data

figsize=plt.figaspect(1.0)
fig, ax = plt.subplots(1, 1, figsize=figsize)
ax.set_xlim(-3,3)
ax.set_ylim(-3,3)
ax.contourf(x,y,z.reshape(len(x),len(y)),100,cmap=plt.cm.jet)
ax.contour(x, y, data_fitted.reshape(len(x),len(y)), 8, colors='w')
ax.set_xlabel("x")
ax.set_ylabel("y")
plt.tight_layout()
plt.show()
```

Now, let us explain this example step by step.

First, let us write down an expression for a 2D Gaussian:

$$f(x,y) = h\,\exp\left\{-\left[\frac{(x-x_0)^2}{2\,\sigma_x^2}+\frac{(y-y_0)^2}{2\,\sigma_y^2}\right]\right\}, \tag{316}$$

where $h$ is the height, $x_0$ and $\sigma_x$ are the mean and the standard deviation in the $x$ direction, $y_0$ and $\sigma_y$ are the mean and the standard deviation in the $y$ direction. Here we assumed that the 2D Gaussian is oriented along the $x$ and $y$ axis. For a generic orientation, we should include also a rotation angle.

Let us write this in python:

```
def twoD_gauss(xy,x0,y0,sx,sy,h):
    z=h*np.exp(-((xy[0]-x0)**2/(2.*sx*sx)+(xy[1]-y0)**2/(2.*sy*sy)))
    return z
```

where the $x, y$ couples are stored in the $xy = (x, y)$ tuple.

Now, we can use the optimize.curve_fit as

```
def sci_fit(x,y,x0,y0,sx,sy,h):
    xy=(x,y)
    popt,pcov=opt.curve_fit(twoD_gauss,xy,z,p0=(x0,y0,sx,sy,h))
    return popt,pcov
```

Note however that if we do so for a reasonably large choice of $z$ (like a $500 \times 500$ matrix) we likely get the error

```
ValueError: object too deep for desired array
```

The reason is that $z$ is a matrix, while curve_fit only takes 1D arrays as argument. Hence, we must first convert $z$ to a 1D array. We can use for this the function ravel() of numpy, which acts on numpy arrays to transform them to 1D. For example

```
a=np.array([[2,1],[5,7]]) #defines a 2x2 np matrix
print(a)
a.ravel()
print(a)
```

The first print yields

```
array([[2, 1],
       [5, 7]])
```

The second print yields

```
array([2, 1, 5, 7])
```

i.e. the matrix has become a 1D array thanks to ravel().

Now, if we change our previous code to

```
def twoD_gauss(xy,x0,y0,sx,sy,h):
    z=h*np.exp(-((xy[0]-x0)**2/(2.*sx*sx)+(xy[1]-y0)**2/(2.*sy*sy)))
    return z.ravel()

def sci_fit(x,y,x0,y0,sx,sy,h):
    xy=(x,y)
    popt,pcov=opt.curve_fit(twoD_gauss,xy,z,p0=(x0,y0,sx,sy,h))
    return popt,pcov
```

we will see that python stop complaining.

Let us input our mock data now, for example

```
def data_set():
    x=np.linspace(-5.,5.,num=500) #x data
    y=np.linspace(-5.,5.,num=500) #x data
    x,y=np.meshgrid(x,y)
    z=np.zeros([len(x),len(y)],float)

    x0=0.0
    y0=0.0
    sx=1.0
    sy=0.7
    h=1.0

    xy=(x,y)
    z=twoD_gauss(xy,x0,y0,sx,sy,h)

    z=z + 1e-1*np.random.normal(size=z.shape)


    return x,y,z
```

Note here two things. First, the usage of **np.linspace** to produce a set of 500 numbers linearly spaced from −5 to 5 (which has the equivalent function for logspaced numbers, np.logspace).

Second, note the usage of **np.meshgrid** to transform $x$ and $y$ from 1D arrays to 2D matrices the way we need to build a Cartesian grid. For example

```
x=np.array([2,1])
y=np.array([0,3])
x,y=np.meshgrid(x,y)
print(x)
print(y)
```

where the first print yields

```
array([[2, 1],
       [2, 1]])
```

and the second yields

```
array([[0, 0],
       [3, 3]])
```

Now, assuming that the fit is successful, we have to produce our fitted values of z as a function of xy. Note that we do it with

```
data_fitted = twoD_gauss(xy, *popt) #fitted data
```

In the above piece of code we have called again the function twoD_gauss in the points xy and we have assigned it also the parameters of the best fit as *popt. The asterisk here has the meaning of calling a pointer to the array popt, which contains the best fitted parameters. This is a pointer like in C and C++. This does not exist in python, just in scipy and is not commonly used (but is used in this case). If you do not know the C and C++ and you do not know what is a pointer, do not worry. Just think that this is a funny notation to call the array of the optimized parameters and that you will encounter it only a bunch of times and only with scipy.

At the end, we plot our results, but notice that $z$ is no longer a matrix. We have to transform it back to a matrix and to do so we use the command reshape(N,N), which transforms it back to a N,N matrix:

```
ax.contourf(x,y,z.reshape(len(x),len(y)),100,cmap=plt.cm.jet)
```

The example should be more clear now. And the final result should be like Figure 69 and should held parameters like $x_0 = -0.0007 \pm 0.0008$, $y_0 = 0.0007 \pm 0.0006$, $\sigma_x = 1.0000 \pm 0.0007$, $\sigma_y = 0.7000 \pm 0.0005$.

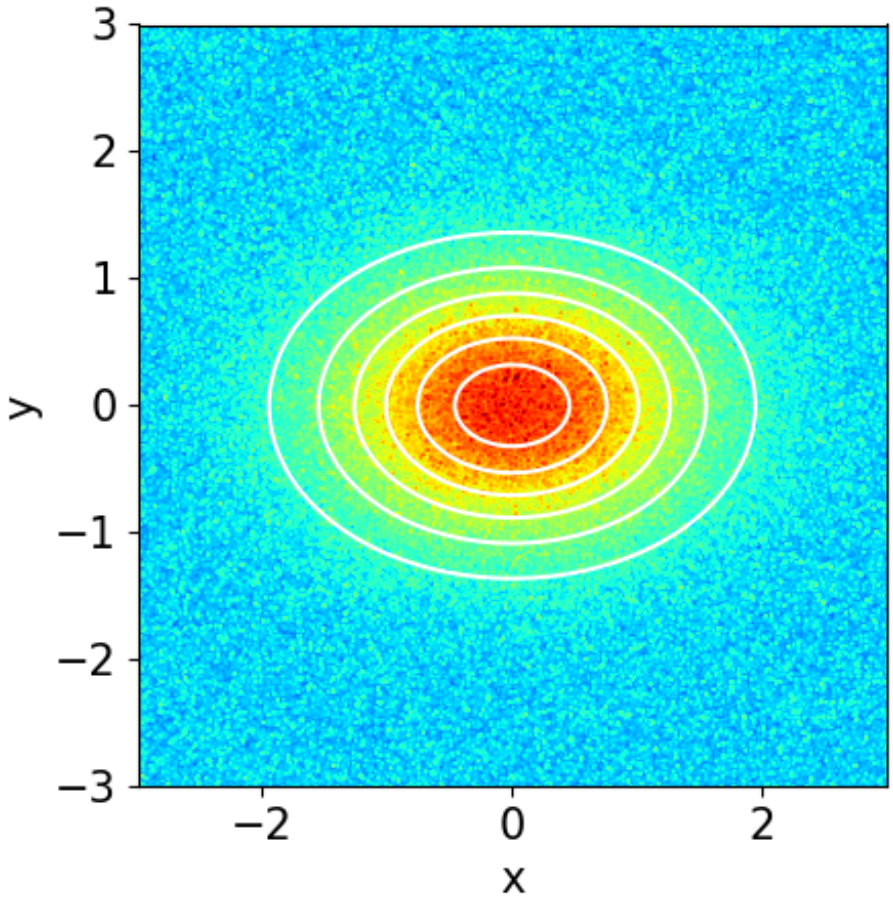


*Figure 69: Least-square fit with a 2D Gaussian using scipi.optimize.curve_fit. Color map: mock data. White contour levels: 2D Gaussian best fit.*

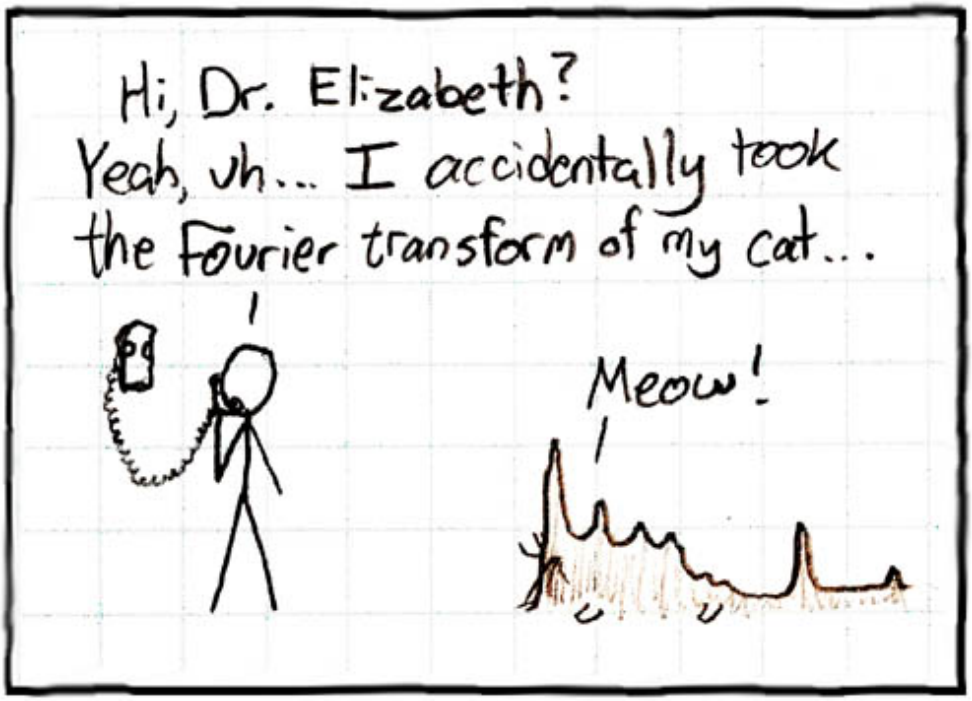


*Figure 70: No need for caption. Credits: xkcd (see* `https://xkcd.com/`*). See also* `https://www.youtube.com/watch?v=UGRhqZ2ooIw`*.*

## 14 FOURIER TRANSFORMS

This chapter is based on *Computational Physics* by Mark Newman `http://www-personal.umich.edu/~mejn/cp/`.

Fourier transforms are ubiquitous in physics, astrophysics and possibly in every-day life (e.g. they are behind the mp3 file and the jpg file technique).

### 14.1 *Fourier series: a math summary*

As you have learned in your math courses, a **periodic function** $f(x)$ defined on a finite interval $0 \leq x < \mathrm{L}$ can be written as a Fourier series, provided that it is bounded and has at most a finite number of discontinuities and extrema.

If the function is **even** (i.e. symmetric) about the midpoint at $x = \mathrm{L}/2$ then we can use a cosine series:

$$f(x) = \sum_{k=0}^{\infty} \alpha_k \cos\left(\frac{2\pi k x}{\mathrm{L}}\right), \tag{317}$$

where the $\alpha_k$ are a set of coefficients whose values depend on the shape of

the function.

If the function is **odd** (i.e. anti-symmetric) about the midpoint, then we can use a sine series:

$$f(x) = \sum_{k=1}^{\infty} \beta_k \, \sin\left(\frac{2\,\pi\,k\,x}{\mathrm{L}}\right), \tag{318}$$

where the $\beta_k$ are another set of coefficients whose values depend on the shape of the function. Note that the sum of the sine series starts from $k = 1$ only because the term with $k = 0$ is always zero.

For a **general** periodic function, with no special symmetry, we can write:

$$f(x) = \sum_{k=0}^{\infty} \alpha_k \, \cos\left(\frac{2\,\pi\,k\,x}{\mathrm{L}}\right) + \sum_{k=1}^{\infty} \beta_k \, \sin\left(\frac{2\,\pi\,k\,x}{\mathrm{L}}\right). \tag{319}$$

An alternative version of writing down this series is by using the complex notation $\cos\theta = \frac{1}{2}\,(\exp(-i\,\theta) + \exp(i\,\theta))$ and $\sin\theta = \frac{1}{2}\,i\,\,(\exp(-i\,\theta) - \exp(i\,\theta))$. Substituting these formulas into equation 319, we get

$$\begin{aligned} f(x) = \frac{1}{2} \sum_{k=0}^{\infty} \alpha_k \left[\exp\left(-i\,\frac{2\,\pi\,k\,x}{\mathrm{L}}\right) + \exp\left(i\,\frac{2\,\pi\,k\,x}{\mathrm{L}}\right)\right] \\ + \frac{i}{2} \sum_{k=1}^{\infty} \beta_k \left[\exp\left(-i\,\frac{2\,\pi\,k\,x}{\mathrm{L}}\right) - \exp\left(i\,\frac{2\,\pi\,k\,x}{\mathrm{L}}\right)\right] \end{aligned} \tag{320}$$

Collecting terms and generalizing for $k \in (-\infty, \infty)$, the above equation can be rewritten as

$$f(x) = \sum_{k=-\infty}^{\infty} \gamma_k \, \exp\left(i\,\frac{2\,\pi\,k\,x}{\mathrm{L}}\right), \tag{321}$$

where

$$\gamma_k = \begin{cases} \frac{1}{2}\,(\alpha_{-k} + i\,\beta_{-k}) & \text{if } k < 0 \\ \alpha_0 & \text{if } k = 0 \\ \frac{1}{2}\,(\alpha_k - i\,\beta_k) & \text{if } k > 0 \end{cases} \tag{322}$$

If we can calculate the coefficients $\gamma_k$, then we are able to reconstruct the function $f(x)$ by using its Fourier series.

Note that Fourier series can be used only for periodic functions, meaning that the function in the interval $0 \le x < \mathrm{L}$ is repeated over and over again all the way out to infinity in both the positive and negative directions. Most of the functions we deal with **are not periodic**. Luckily, if we are interested in a portion of a non-periodic function over a finite interval 0, L, it is always possible to take that portion and just repeat it to create a periodic function,

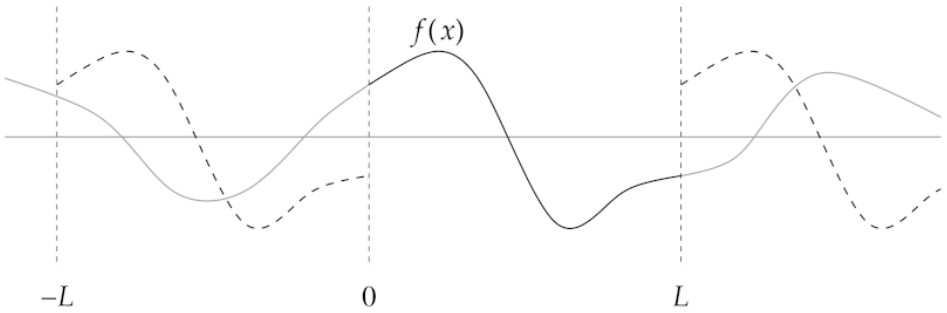


*Figure 71: Creating a periodic function from a non-periodic one. The function $f(x)$ we are interested in is non-periodic. We can calculate a Fourier series of $f(x)$ in a finite interval $0 \leq x < \mathrm{L}$ by modifying the function as shown. We discard the parts of the function represented by the gray curves that fall outside the region of interest and replace them with an infinite set of periodic repetitions of the portion from 0 to* L *(dashed black curves). Credits: Mark Newman, Computational Physics.*

as shown in Figure 71. Then the Fourier series formulas given above are the correct representation of the function in the interval 0, L. Outside of that interval, they give an incorrect answer, but this does not matter since we are interested only in the interval 0, L.

The coefficients $\gamma_k$ in equation 322 are, in general, complex. The standard way to calculate them is to evaluate the integral

$$\int_0^{\mathrm{L}} f(x)\, \exp\left(-i\, \frac{2\,\pi\, k\, x}{\mathrm{L}}\right) \mathrm{d}x = \sum_{k'=-\infty}^{\infty} \gamma_{k'} \int_0^{\mathrm{L}} \exp\left(i \frac{2\,\pi\,(k'-k)\,x}{\mathrm{L}}\right) \mathrm{d}x, \qquad (323)$$

where we have used equation 321 and we have multiplied both terms by $\exp\left(-i\, \frac{2\,\pi\, k\, x}{\mathrm{L}}\right)$. The integral on the right is easy to calculate. When $k' \neq k$, the integral $\int_0^{\mathrm{L}} \exp\left(i \frac{2\,\pi\,(k'-k)\,x}{\mathrm{L}}\right) \mathrm{d}x$ is always equal to zero for the following reasons. From your math courses you might remember (otherwise it is easy to derive) that, for any complex $z = a + i\, b$, we have

$$\int e^{z\,x} \mathrm{d}x = \frac{e^{z\,x}}{z} \qquad (324)$$

Hence, the right-hand term of equation 323 if $k' \neq k$ becomes

$$\int_0^{\mathrm{L}} \exp\left(i\,\frac{2\,\pi\,(k'-k)\,x}{\mathrm{L}}\right)\mathrm{d}x = \frac{\mathrm{L}}{i\,2\,\pi\,(k'-k)}\left[\exp\left(i\,\frac{2\,\pi\,(k'-k)\,x}{\mathrm{L}}\right)\right]_0^{\mathrm{L}} =$$

$$\frac{\mathrm{L}}{i\,2\,\pi\,(k'-k)}\left\{\exp\left[i\,2\,\pi\,(k'-k)\right]-1\right\}=0, \quad (325)$$

where we used the fact that $\exp(i\,2\,\pi\,n)=1$ for any integer $n$ (and $k'-k$ is an integer by definition).

The only non-zero case for the right-hand term of equation 323 is for $k'=k$, when the integral is equal to L. Thus, equation 323 can be rewritten as

$$\int_0^{\mathrm{L}} f(x)\,\exp\left(-i\,\frac{2\,\pi\,k\,x}{\mathrm{L}}\right)\mathrm{d}x = \gamma_k\,\mathrm{L} \quad (326)$$

or, rewriting in terms of $\gamma_k$

$$\gamma_k = \frac{1}{\mathrm{L}}\int_0^{\mathrm{L}} f(x)\,\exp\left(-i\,\frac{2\,\pi\,k\,x}{\mathrm{L}}\right)\mathrm{d}x \quad (327)$$

Thus, given the function $f(x)$, we can find the Fourier coefficients $\gamma_k$, or, vice versa, given the coefficients $\gamma_k$ we can reconstruct the function $f(x)$.

## 14.2 *The discrete Fourier transform (DFT)*

The integral in equation 327 can be calculated analytically only for some functions. In many cases, the integral does not admit an analytic solution, or the analytic solution is too complicated, or we do not know the functional form of $f(x)$ but only the value of $f(x)$ in a set of points (for example because we have a set of experimental data). In such cases, we can calculate the coefficients $\gamma_k$ numerically.

Let us simply use the **trapezoidal rule** to solve the integral in equation 327 with N slices of width $h = \mathrm{L}/\mathrm{N}$. Applying the trapezoidal rule (equation 106), we get

$$\gamma_k = \frac{1}{\mathrm{L}}\frac{\mathrm{L}}{\mathrm{N}}\left[\frac{1}{2}f(0)+\frac{1}{2}f(\mathrm{L})+\sum_{n=1}^{\mathrm{N}-1} f(x_n)\,\exp\left(-i\,\frac{2\,\pi\,k\,x_n}{\mathrm{L}}\right)\right], \quad (328)$$

where the positions $x_n$ of the sample points for the integral are

$$x_n = \frac{n}{\mathrm{N}}\,\mathrm{L} \quad (329)$$

Since $f(x)$ is by hypothesis periodic with period L, then $f(0) = f(\mathrm{L})$ and equation 328 simplifies to

$$\gamma_k = \frac{1}{\mathrm{N}} \sum_{n=0}^{\mathrm{N}-1} f(x_n) \exp\left(-i \frac{2\,\pi\,k\,x_n}{\mathrm{L}}\right). \tag{330}$$

The equation 330 can be used to evaluate the coefficients $\gamma_k$ numerically.

If the function $f(x)$ is evenly sampled over an interval 0, L, i.e. if the $x_n$ are equally spaced, we can further simplify equation 330 by defining $y_n = f(x_n)$ and by using equation 329:

$$\gamma_k = \frac{1}{\mathrm{N}} \sum_{n=0}^{\mathrm{N}-1} y_n \exp\left(-i \frac{2\,\pi\,k\,n}{\mathrm{N}}\right). \tag{331}$$

In this form, we only need to know the sample values $y_n$ and the total number of samples N.

By convention, the formula 331 is written in a slightly different way (by removing the $\frac{1}{\mathrm{N}}$ term):

$$c_k = \sum_{n=0}^{\mathrm{N}-1} y_n \exp\left(-i \frac{2\,\pi\,k\,n}{\mathrm{N}}\right), \tag{332}$$

where the coefficients $c_k$ are nothing but $c_k = \gamma_k\,\mathrm{N}$. Equation 332 is known as the **discrete Fourier transform** (**DFT**) of the samples $y_n$.

We have derived equation 332 by using the trapezoidal rule, which is an approximated method to solve the integral. However, with few mathematical steps we can show that the DFT is, in a certain sense, exact. In fact (see the book by Mark Newman for the calculations), by starting from equation 332 and adopting some mathematical tricks we can show that

$$y_n = \frac{1}{\mathrm{N}} \sum_{k=0}^{\mathrm{N}-1} c_k \exp\left(i \frac{2\,\pi\,k\,n}{\mathrm{N}}\right), \tag{333}$$

which is the **inverse discrete Fourier transform** (or **inverse DFT**). Equation 333 tells us that, given the coefficients $c_k$ we can derive from equation 332, we can recover the values of the samples $y_n$ that they come from *exactly* (except for rounding errors).

This is an amazing result: even though we thought our Fourier coefficients were only approximate, it turns out that they are actually exact in the sense that **we can completely recover the original samples from them**. Thus, we can freely move back and fourth from the coefficients to the samples and vice versa **WITHOUT LOSING ANY INFORMATION** in our data: both the samples and the Fourier coefficients give us a complete representation of the

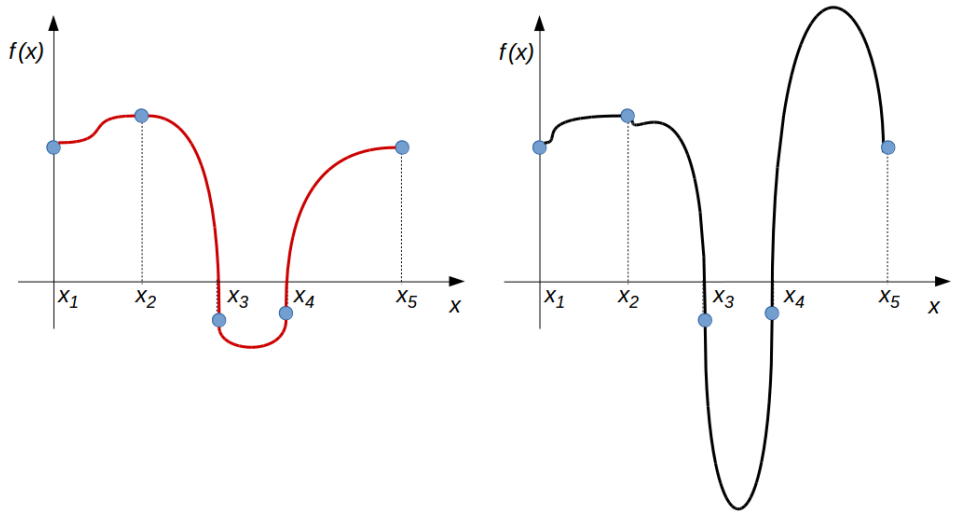


*Figure 72: The function on the left and the function on the right assume the same values a the five sample points (blue circles). Hence, they have the same DFT, but they look dramatically different in between points.*

original data.

Notice that we only need the Fourier coefficients $c_k$ up to $k = \mathrm{N} - 1$ to recover the samples, so we need to evaluate equation 332 only for $0 \leq k < \mathrm{N}$ (no need to calculate an infinite sum!).

It is important to note that, unlike the original Fourier series (e.g. equation 321), the inverse DFT equation 333 only gives us the sample values $y_n = f(x_n)$: **it tells us nothing about the value of $f(x)$ between sample points**. The original function could do anything it wanted between two sample points and we would never know it. In other words, any two functions that have the same values at the sample points will have the same DFT, no matter what they do between the sample points, like the two functions in Figure 72.

Still, if a function is reasonably smooth and we sample enough points, knowing the values at the sample points is enough to get a picture of the function's general shape. Moreover, we cannot do better than this when we do not know the functional form of the function, but we have only a sample of discrete experimental data.

The results discussed before apply for both complex and real functions. If we are interested only in real functions (which is mostly the case, in physics), we can do some further simplification. Suppose all the $y_n$ are real and consider the value $c_k$ for $\mathrm{N}/2 < k < \mathrm{N}$, which we write as $k = \mathrm{N} - r$ where $1 \leq r \leq \mathrm{N}/2$.

Then

$$c_{\mathrm{N}-r} = \sum_{n=0}^{\mathrm{N}-1} y_n \, \exp\left(-i \, \frac{2\,\pi\,(\mathrm{N}-r)\,n}{\mathrm{N}}\right)$$

$$= \sum_{n=0}^{\mathrm{N}-1} y_n \, \exp\left(-i\,2\,\pi\,n\right) \exp\left(i \, \frac{2\,\pi\,r\,n}{\mathrm{N}}\right)$$

$$= \sum_{n=0}^{\mathrm{N}-1} y_n \, \exp\left(i \, \frac{2\,\pi\,r\,n}{\mathrm{N}}\right) = c_r^*, \tag{334}$$

where $c_r^*$ is the complex conjugate of $c_r$. Thus, $c_{\mathrm{N}-1} = c_1^*$, $c_{\mathrm{N}-2} = c_2^*$, etc. This means that, if the function is real, we have to calculate only the coefficients $c_k$ with $0 \leq k \leq \mathrm{N}/2$. If N is even, we must calculate $\mathrm{N}/2 + 1$ coefficients, if N is odd, we must calculate $(\mathrm{N} + 1)/2$ coefficients. In python notation, this means that we should calculate

```
N//2+1
```

coefficients, where '//' is the integer division operator.

## 14.3 *Fast Fourier transform (FFT)*

The computational cost of the DFT method described in the first section scales approximately as $\mathrm{N}^2$, because we need to calculate $\sim \mathrm{N}/2 + 1$ coefficients and for each coefficient we need to loop over N sampled points (hence, the DFT method scales as $\mathrm{N}\,(\mathrm{N}/2 + 1)$). This is computationally heavy. Can we do better with a different algorithm?

The **fast Fourier transform** (or **FFT**) algorithm is a much more clever scheme to calculate a Fourier transform. It was discovered by Carl F. Gauss in 1805, then almost forgotten, then independently re-discovered by James Cooley and John Tukey in 1965. The simplest FFT transform assumes that the samples is a power of two, hence $\mathrm{N} = 2^m$, with $m$ = integer.

Divide the terms of the sum in the DFT formula (equation 332) into two equally sized groups. Let the first group consists of the terms with $n$ even and the second the terms with $n$ odd. The group of the even terms, i.e. the terms with $n = 2\,r$ with $r = 0, 1, \ldots \mathrm{N}/2 - 1$ is

$$\mathrm{E}_k = \sum_{r=0}^{\frac{\mathrm{N}}{2}-1} y_{2\,r} \, \exp\left(-i \, \frac{2\,\pi k\,(2\,r)}{\mathrm{N}}\right) = \sum_{r=0}^{\frac{\mathrm{N}}{2}-1} y_{2\,r} \, \exp\left(-i \, \frac{2\,\pi k\,r}{\frac{1}{2}\,\mathrm{N}}\right), \tag{335}$$

which is another Fourier transform, with number of samples equal to N/2.

Similarly, the odd terms (with $n = 2\,r + 1$) is

$$\sum_{r=0}^{\frac{\mathrm{N}}{2}-1} y_{2\,r+1}\ \exp\left(-i\,\frac{2\,\pi k\,(2\,r+1)}{\mathrm{N}}\right) = \exp\left(-i\,\frac{2\,\pi k}{\mathrm{N}}\right) \sum_{r=0}^{\frac{\mathrm{N}}{2}-1} y_{2\,r+1}\ \exp\left(-i\,\frac{2\,\pi k\,r}{\frac{1}{2}\,\mathrm{N}}\right) = \exp\left(-i\,\frac{2\,\pi k}{\mathrm{N}}\right) \mathrm{O}_k,$$

where $\mathrm{O}_k$ is another Fourier transform with N/2 samples.

We can thus rewrite the Fourier coefficients as

$$c_k = \mathrm{E}_k + \exp\left(-i\,\frac{2\,\pi k}{\mathrm{N}}\right) \mathrm{O}_k \tag{336}$$

Hence, the Fourier coefficients in the DFT method can be written as the sum of two terms $\mathrm{E}_k$ and $\mathrm{O}_k$ that are each DFTs of the same function $f(x)$ but with half as many points spaced twice as far apart, plus an extra-factor $\exp\left(-i\,\frac{2\,\pi k}{\mathrm{N}}\right)$, called **twiddle factor**.

We now repeat the splitting process onto each of the two Fourier transforms, i.e. $\mathrm{E}_k$ and $\mathrm{O}_k$. Each of them can be divided in its even and its odd terms. We repeat the splitting, until eventually we get to the point where each transform is the transform of just a single sample. But the Fourier transform of a single sample has only a single Fourier coefficient $c_0$, which, putting $k = 0$ and $\mathrm{N} = 1$ is equal to

$$c_0 = \sum_{n=0}^{0} y_n\ \exp(0) = y_0, \tag{337}$$

i.e. if we have a single sample the Fourier coefficient is the sample itself.

The actual calculation of the FFT is the reverse of the process outlined above. We start from the individual samples (which are their own Fourier transforms) and we combine them in pairs, then we combine the pairs into fours, the fours into eights, and so on, creating larger and larger Fourier transforms, until we have reconstructed the full transform of the complete set of samples.

The power of this approach is its speed. At the first round of the calculation we have N samples, at the second N/2 transforms with 2 coefficients each (i.e. N operations), at the third N/4 transforms with 4 coefficients each (i.e. N operations) and so on. We have to do N operations per each level.

We have to calculate this for $m$ levels, where $m = \log_2(\mathrm{N})$, by definition (since $\mathrm{N} = 2^m$). Thus, the FFT method scales as $\mathrm{N}\,\log_2(\mathrm{N})$, much better than the DFT method ($\propto \mathrm{N}^2$).

We will not see the formulas of the FFT method, because they are quite tedious (but see the book by Mark Newman for details) and because there is already a super-efficient module for FFTs in numpy: **numpy.fft**. This module contains several functions.

1. The function **numpy.fft.rfft** calculates the FFT for a sample of **real** numbers ('r' in rfft stands for real). It works as follows.

```
from numpy.fft import rfft
c=rfft(y)
```

   where $y$ is a real numpy array, while $c$ is a complex numpy array. Note that $c$ in the above example has N/2 + 1 coefficients, because of the property we discussed in equation 334.

   To generate a complex numpy array, you should type something like

```
import numpy as np
c=np.array(p,complex)
```

   where $p$ is the number of elements and complex is the type of the array.

2. The function **numpy.fft.irfft** calculates the **inverse FFT**, i.e. takes as an argument the Fourier coefficients $c$ and recovers the value of the samples:

```
from numpy.fft import rfft,irfft
c=rfft(y)
y2=irfft(c)
print(y2)
```

   Note that the real numpy.array $y2$ has N elements, i.e. irfft reconstructs all the N samples from the N/2 + 1 coefficients.

3. The function **numpy.fft.rfft2** calculates the **FFT** for a generic number of real samples in two dimensions, i.e. for samples $y_{mn}$ and with coefficients $c_{ml}$.

4. The function **numpy.fft.irfft2** calculates the **inverse FFT** for a generic number of real Fourier coefficients in two dimensions.

5. The function **numpy.fft.fft** calculates the **FFT** for a generic number of complex samples.

6. The function **numpy.fft.ifft** calculates the **inverse FFT** for a generic number of complex Fourier coefficients.

7. The function **numpy.fft.fft2** calculates the **FFT** for a generic number of complex samples in two dimensions.

8. The function **numpy.fft.ifft2** calculates the **inverse FFT** for a generic number of complex Fourier coefficients in two dimensions.

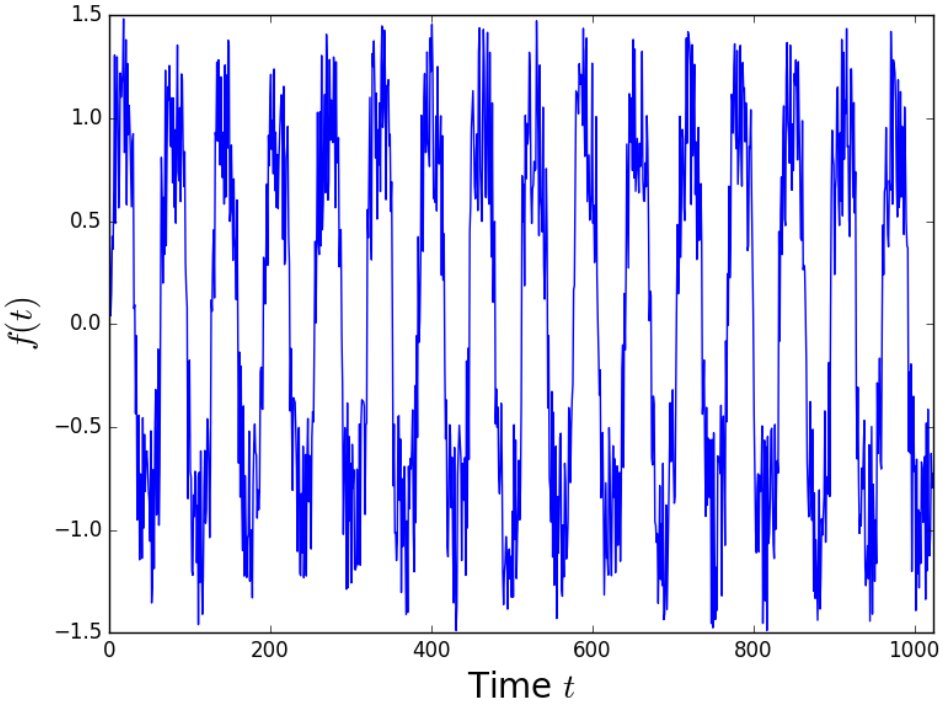


*Figure 73: Example signal presented in Figure 7.3 of Mark Newman, Computational Physics. This plot can be reconstructed from the file* `http://www-personal.umich.edu/~mejn/cp/data/pitch.txt`. *See examples/fourier/. X-axis: time $t$, y-axis: signal $f(t)$. This signal has an overall wave-like shape wit a well-defined frequency ($\nu = k_{\max}/\mathrm{N} = 16/1024 \sim 0.0156$) but also contains some noise.*

## 14.4 *Physical interpretation of the Fourier transform*

The Fourier transform represents a function via a set of real or complex sinusoidal waves. Each term in the sum in equation 333 represents a single **wave** with its own well-defined **frequency** $\nu = k/\mathrm{N}$ and **period** $\mathrm{T} = \nu^{-1} = \mathrm{N}/k$.

If the function $f(x)$ is a function in space, then we will have spatial frequencies, if it is a function in time then we will have temporal frequencies.

Saying that any function can be expressed as a Fourier transform is equivalent to saying that any function can be represented as a **sum of waves of given frequencies** and the **coefficients of the Fourier transform tell us how much power is associated with each frequency, i.e. how big is the contribution of each frequency to the sum.**

The frequency associated with the largest coefficient different from the $k = 0$ coefficient is then the most important frequency associated with the data we are considering.

For example, let us consider the signal shown in Figure 73. This signal consists of a basic wave with a well-defined frequency and period, but also has some noise, visible as smaller wiggles in the line. If this signal were a

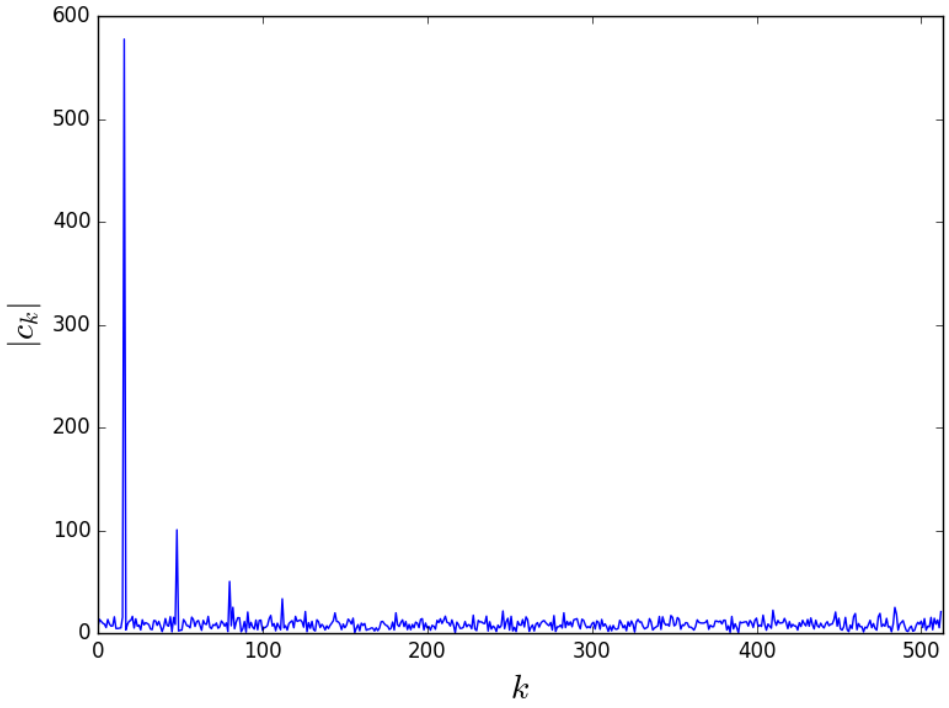


*Figure 74: Values of the Fourier coefficients $|c_k|$ versus k for the example signal shown in Figure 73. The main spike corresponds to a frequency $\nu = k_{\max}/\mathrm{N} = 16/1024 \sim 0.0156$.*

sound, one could hear a constant note at the frequency of the main wave, plus a background hiss that comes from the noise.

We can calculate the Fourier transform of this signal as shown in examples/fourier/. The file fft.py contains the following lines

```
#examples/fourier/fft.py
#uses FFT to calculate Fourier transform of pitch.txt
import matplotlib.pyplot as plt
import numpy as np
from cmath import exp,pi
from numpy.fft import rfft,ifft

fname="pitch.txt"
y=np.genfromtxt(fname,unpack=True)
m=range(len(y))

# plot Figure 73
plt.plot(m,y)
plt.xlabel("Time $t$",fontsize=20)
plt.ylabel("$f(t)$",fontsize=20)
plt.xlim(0,1024)
plt.tight_layout()
plt.show()

# calculates Fourier transform with FFT
N=len(y)
c=rfft(y)
cabs=(abs(c))
kk=[]
maxc=0.0
maxk=0

# calculated frequency k/N corresponding to the max c_k
for k in range(N//2+1):
    if((cabs[k]>maxc) and (k>0)):
        maxc=cabs[k]
        maxk=k/float(N)
    kk.append(k)

#plots |ck| versus k (Figure 74)
plt.plot(kk,cabs)
plt.xlabel("$k$",fontsize=20)
plt.ylabel("$|c_k|$",fontsize=20)
plt.ylim(0,600)
plt.xlim(0,513)
plt.tight_layout()
plt.show()

#prints frequency corresponding to the max c_k
print(maxk,maxc)
```

It first reads the file pitch.txt and plots Figure 73. Then it calculates the Fourier transform calling the function numpy.fft.rfft. Finally, it plots the value of the absolute value of the coefficients $|c_k|$ versus $k$ (Figure 74) and it calculates

what is the frequency associated with the largest $|c_k|$.

Figure 74 shows $|c_k|$ versus $k$. We remind that $k$ is proportional to the frequency of the waves ($\nu = k/\mathrm{N}$). There are a number of significant spikes in the plot, corresponding to the largest values of $|c_k|$. The highest spike corresponds to $k = 16$, hence a frequency $\nu \sim 0.0156$ and a period $\mathrm{T} = 1/\nu = 64$. These are the main frequency and periodicity visible in Figure 73.

The remaining spikes in Figure 74 are **harmonics** of the first one: multiples of the main frequency that tell us that the wave in the original data was not a pure sine wave (a pure sine wave is represented with just a single term of the appropriate frequency in the Fourier series, while any other wave requires some additional terms). Finally, the plot shows some small, apparently random values of $|c_k|$ creating a low-level background. These non-zero but very small $|c_k|$ are produced by some noise in the signal, which can be considered a **white noise**, because it does not show any dependence with $k$.

If we want to get rid of the noise, we might decide to reconstruct the initial samples **without considering the smallest $|c_k|$.** Hence, the simplest way to **de-noise** a signal, is to **reconstruct the original samples by doing the inverse FFT after zeroing all the values of $c_k$ with $|c_k|$ smaller than a given threshold**. In our example, $|c_k| \sim 25$ seems to be a reasonable assumption for the largest $|c_k|$ produced by the noise. If we then complete the previous script as following

```
#DENOISE DATA BY REMOVING THE SMALLEST |c_k|
threshold=25. #maximum value of |c_k| that
#I consider associated with white noise

for k in range(N//2+1):
    if(cabs[k]<threshold):
        c[k]=0.0

y2=irfft(c)
#plots the reconstructed function "without noise"
a,=plt.plot(m,y)
b,=plt.plot(m,y2,color="red",linewidth=4)
plt.xlabel("Time $t$",fontsize=20)
plt.ylabel("$f(t)$",fontsize=20)
plt.xlim(0,1024)
plt.legend([a,b],["Signal","De-noised signal"],fontsize=15)
plt.tight_layout()
plt.show()
```

we obtain Figure 75, where the red line shows the reconstructed signal after removing the noise-like coefficients. Of course, this procedure works only if the noise is perfectly white and the final result is sensitive to the choice of the threshold. This discussion is not a tutorial on de-noising but just a very

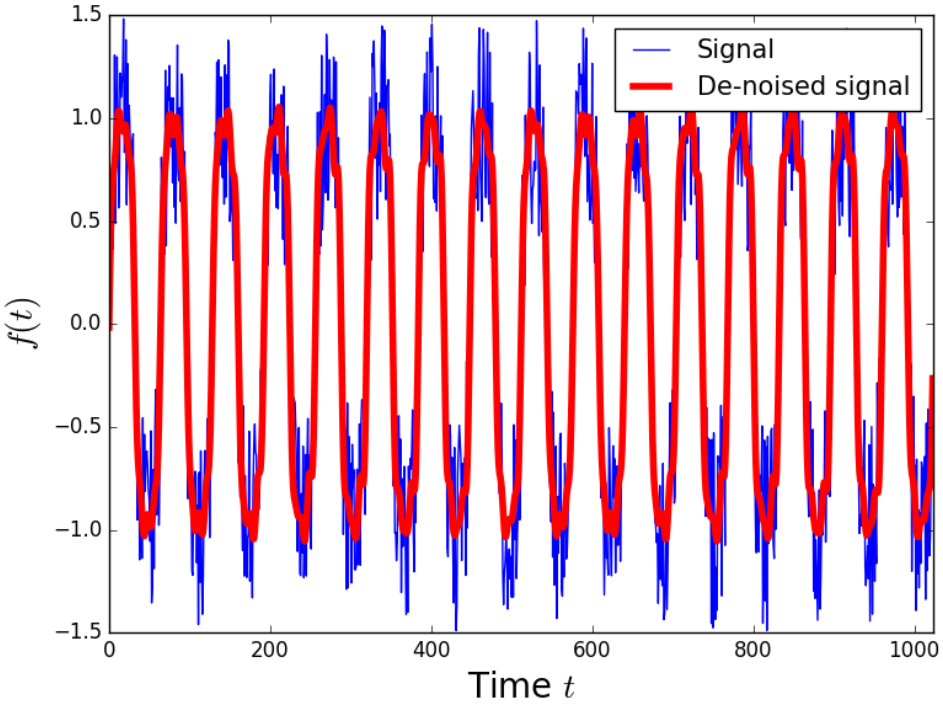


*Figure 75: Blue line: same signal $f(t)$ as a function of $t$ presented in Figure 73. Red thick line: de-noised signal obtained by doing the inverse FFT after zeroing all the coefficients $c_k$ with $|c_k| < 25$.*

simplified example.

After we have reconstructed the initial samples, we have a smoother version of the signal which, for example, can be used to calculate less noisy numerical derivatives (see the discussion in Section 7.4).

**EXERCISE:**

The file exercises/fourier/sunspots.txt (or, if you prefer to use gitlab: the file exercises/fourier/sunspots.txt) contains the observed number of sunspots on the Sun for each month since January 1749. The file contains two columns: column 0 is the month and column 1 is the number of sunspots per each month.

1. Plot the number of sunspots as a function of the month. The result should look like Figure 76.
2. Write a script to perform the discrete Fourier transform (DFT) and the fast Fourier transform (FFT) of the number of sunspots as a function of the month. For the DFT, use equation 332. For the FFT, use the numpy.fft.rfft function. Compare the different performances. The DFT should be significantly slower than the FFT.
3. Plot the Fourier coefficients $|c_k|$ as a function of $k$. The result should look like Figure 77.
4. Calculate the period T = N/$k$ associated with the largest $|c_k|$. This gives you the sunspot periodicity ($t \sim 10.9$ years).
5. Try to denoise the data assuming that there is just white noise. Plot the denoised signal. The result should look like Figure 78. Note: for the inverse Fourier transform it is sufficient that you use the corresponding numpy function (no need to write an indirect DFT).

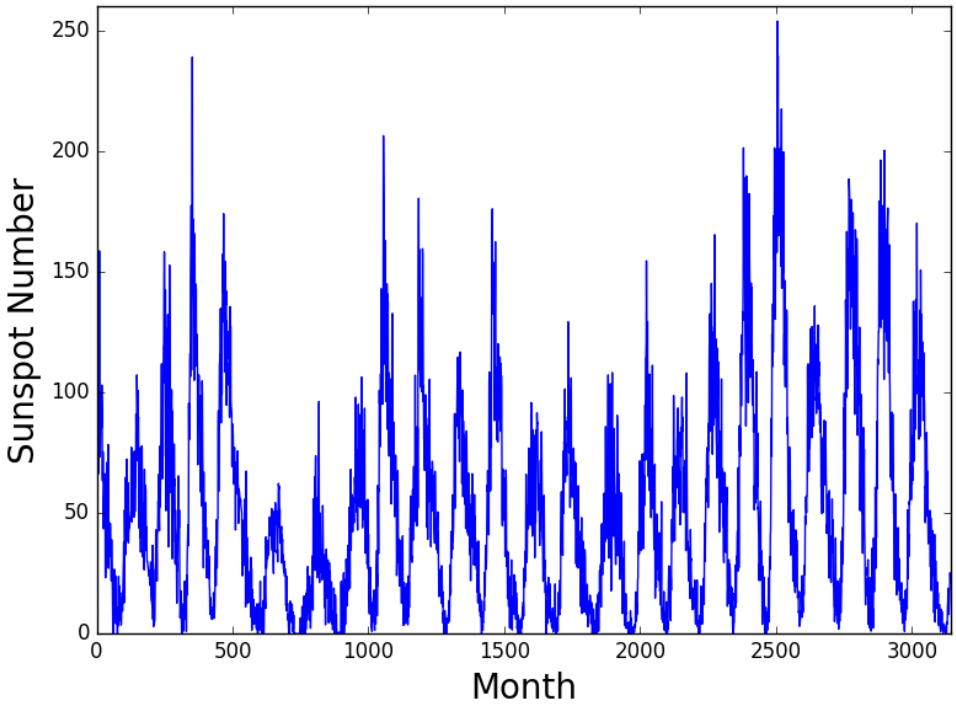


*Figure 76: Number of sunspots as a function of the month. Result of the exercise.*

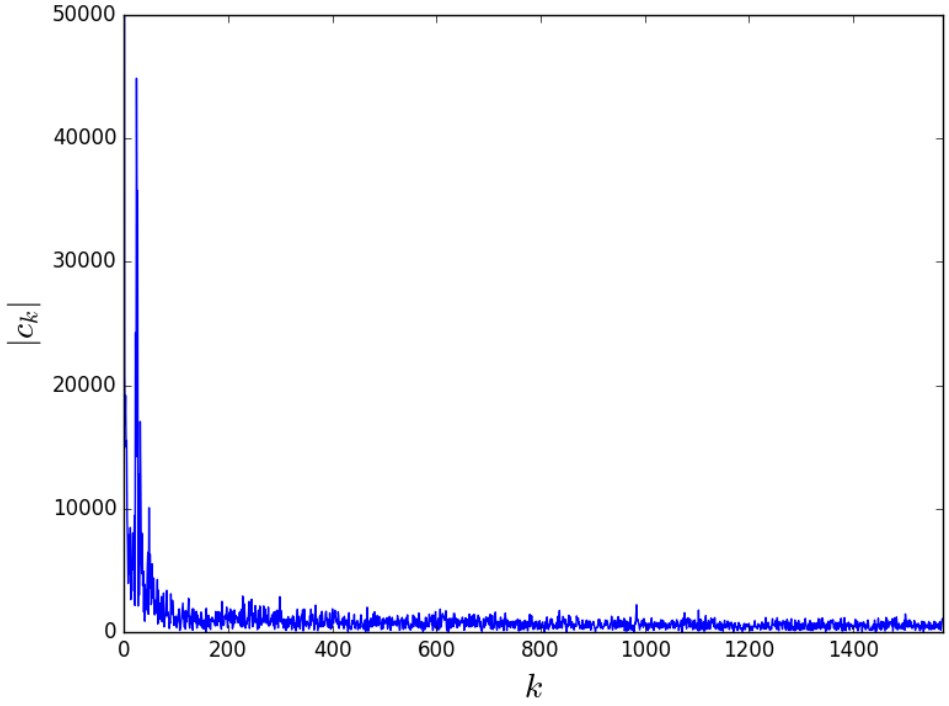


*Figure 77: Fourier coefficients $|c_k|$ as a function of k in the data of the sunspot exercise.*

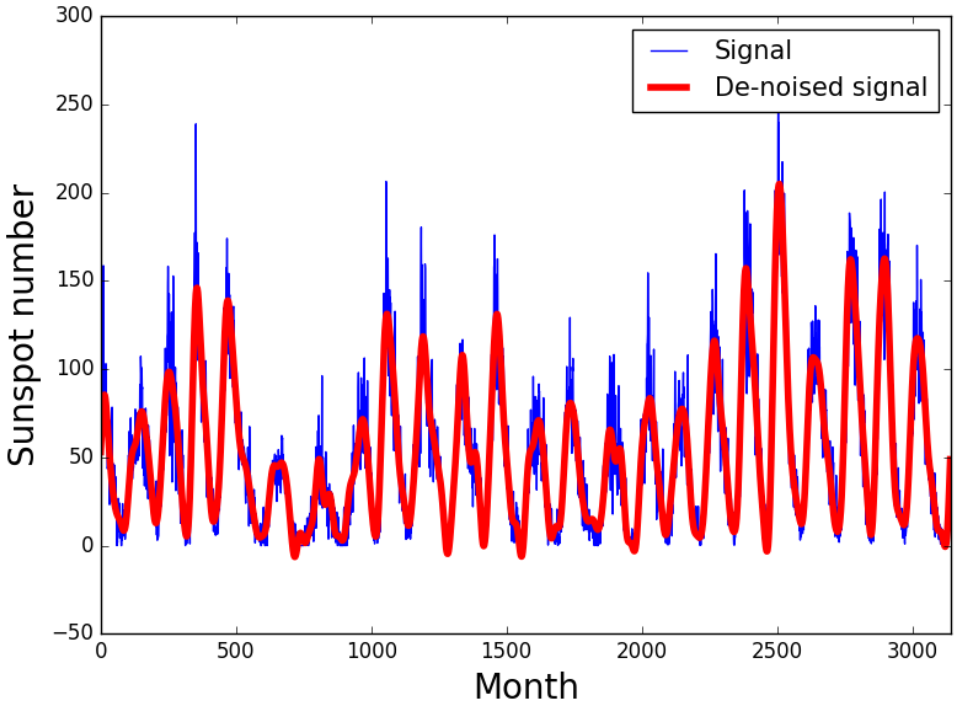


*Figure 78: Blue line: Number of sunspots as a function of the month. Red line: de-noised version of the data.*

# 15 OTHER USEFUL PACKAGES AND ENVIRONMENTS

## 15.1 *The conda environment*

Let's briefly introduce **CONDA**. You can download the conda package and learn more about it from `https://docs.conda.io/projects/conda/en/latest/user-guide/index.html`. Conda is an open and free software, of course. If you are using the virtual machine or the workstations at the lab P104, you do not need to download it, because it is already installed and running (even if you do not see it).

Conda is an 'open source package management system and environment management system that runs on Windows, macOS and Linux. Conda quickly installs, runs and updates packages and their dependencies. Conda easily creates, saves, loads and switches between environments on your local computer. It was created for Python programs, but it can package and distribute software for any language.' (the quotes indicate that these sentences were taken from the conda website).

Translated into simpler and less celebratory words: conda is a software package with which you can create one or more **environments**, i.e. "virtual spaces" where you can install software that is different from the one you have on your computer, without the risk of conflicts with other programs installed on your computer. The rest of the software on your computer is not affected by conda, it does not know anything about conda.

Basically, the main advantage of using conda is that you can install and un-install software without the risk of screwing up your computer operating system and other installed software[8].

The packages that you install through conda work only inside conda do not affect your overall system. Hence, you can install through conda packages that otherwise would be **conflicting** with your system.

You can set up different environments with different kind of softwares depending on what you need (e.g. one with python3 and one with python2) and running them at the same time without conflicts.

It is extremely user friendly and safe. You can install software through conda by simply typing:

```
conda install software_name
```

What conda does not do:

- it is not a virtual machine (but you can install it inside a virtual machine);
- it does not support an operating system different from the one you have on your computer. E.g., if you are a windows user you must install conda

[8] Note, for example, that if you screw up when you install python on ubuntu, you might screw up your entire computer, because ubuntu is basically built through python. If you do the installation inside conda, this does not affect your operating system

for windows, if you are a mac user you must install the version for mac OS, if you are a linux user the one for linux.

There are two different ways you get conda:

- ANACONDA: complete version, needs disk space (∼ 3 Gb initially, but it can easily grow to ∼ 10 Gb if you install more software later on);
- MINICONDA: lighter version, much less space (∼ 400 Mb) but less features

The name "CONDA" refers only to the **package manager** (the package which allows you to install software in a new environment) and is included in both the anaconda and miniconda software distributions. I suggest full anaconda if you have space

To install conda on your laptop, go here `https://docs.conda.io/en/latest/` and follow the instructions for your operating system. If you have problems, ask me. After you install conda, you have to activate it by typing

```
conda activate
```

To exit conda

```
conda deactivate
```

If you use the virtual machine or the workstations at the lab P104 you do not need to install conda: it is already there even if you do not see it.

If you install conda, all the packages we will be talking about in the next sections are automatically installed together with conda. Hence, it is quite a good investment to spend ∼ 15 minutes of your life installing conda.

## 15.2 *jupyter-notebook*

Project jupyter **https://jupyter.org/** includes several other features besides the notebook, but the notebook is definitely the most used and famous. Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text.

If you have conda, jupyter-notebook should be already installed inside conda and can be simply started by typing **jupyter-notebook** on your command line. It launches a web browser interface that allows to enter and run blocks of code.

If you have conda, but jupyter-notebook is not installed (quite strange), install it doing

```
conda install jupyter
```

Let's look at the example file **examples/jupyter/hermite.ipynb**.

Run it with

```
jupyter-notebook hermite.ipynb
```

**My warning:** it is terribly easy to forget re-running some blocks, even if you change them. Please be careful.

**My suggestion:** notebook is very user-friendly when you are creating a new script (especially for plotting), but it might be less useful when you go to production (e.g., if you want to launch it on a remote server with many cores but limited options for internet access and graphics interface). Once you are sure your script works with the notebook, you can save it as a plain python script and run it as a plain python script.

**Very popular:** jupyter notebook is becoming increasingly popular in the scientific community. For example, all the LIGO–Virgo code is done with notebooks.

## 15.3 *Using file formats other than plain text files*

Broadly speaking, all kind of files can be divided into just two major families: text files and binary files.

**Text files:**

A text file is a file that contains plain, human-readable text (with numbers and/or words). When I open a text file with a text editor (e.g. emacs, gedit), I can read its content directly. For example, ASCII files and CSV files are text files. At the beginning of the course we have already discussed how we can open, read and write plain text files.

**Binary files:**

A binary file is a computer file that is not a text file. "Binary files are usually thought of as being a sequence of bytes, which means the binary digits (bits) are grouped in eights. Binary files typically contain bytes that are intended to be interpreted as something other than text characters" (source: Wikipedia). In the following, we will see one example of a binary file which is the HDF5 file format.

Python can open a generic binary file, for reading or writing, with the function **open**, just by adding the "b" option. For example

```
f=open("mybinfile","rb")
f2=open("mybinfile2","wb")
```

which opens file mybinfile for reading and mybinfile2 for writing.

The main problem is how to actually read and interpret the information inside the binary file after I have opened it with the function open. To read

a binary file, I need more information on how the content of the binary file is organized. This information can be, at least partially, retrieved inside the binary file itself. See the specific example of the HDF5 file below.

The correct reading of some (but not all!) binary file formats might be limited by the **endiannes** of the machine where the file was generated. Endianness is the ordering of bytes in computer memory storage or during transmission. Endiannes can be either big-endian or little-endian. A **big-endian** system stores the most significant byte of a word at the smallest memory address and the least significant byte at the largest. Vice versa, a **small-endian** system stores the most significant byte of a word at the largest memory address and the least significant byte at the smallest. Big-endianness is the dominant ordering in networking protocols (the most significant byte is transmitted first). Conversely, little-endianness is the dominant ordering for processor architectures (e.g., x86).

Since binary files are just sequences of bytes, a binary file produced by a small-endian system has an internal ordering that is totally different from a binary file produced by a big-endian system and vice versa. This might have disruptive impact on your analysis.

In many contemporary binary file formats (e.g. HDF5) this issue is cured by information stored inside the file: the user does not need to care about endianness, because the file handles it on its own. When you have to work with a binary file format you have never used before, just check how it handles endianness.

### ASCII files

An ASCII File is a file that contains unformatted ASCII text: just **plain, human-readable text using only ASCII characters**. Hence an ascii file is a specific type of **text file** that contains only ASCII characters. ASCII file is often used as a synonym of text file.

Technically speaking, ASCII (American Standard Code for Information Interchange) is a character encoding standard for electronic communication. In particular, ASCII is an encoding representing each typed character by a number between 0 and 255 (i.e., a number that can be stored into a byte). It was published in 1963 and reviewed in 1967. It might be soon replaced by UTF-8, which stores each character into up to 4 bytes, allowing to include all the characters from, e.g., Greek, Cyrillic and Arabic, and most characters from, e.g., Chinese, Japanese and Korean.

### Encoding for Dummies

Let's open a short parenthesis on character representation in computers. As you know already from previous courses (hopefully), the smallest unit of storage of data into your computer is the **bit**. The bit stores just a 0 or a 1. Groups of 8 bits are called **bytes**.

Numbers on a computer are represented as **binary numbers** (e.g., $100101 = 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 37$). Hence, representing a number on a computer usually requires more than one bit (only 0 and 1 require just one bit).

**Characters** other than numbers are stored in a computer by associating to each of them a given number. The correspondence between a given character and a given number is established by the **character encoding standard** we choose.

ASCII is one of such character encoding standards that establish a one-to-one correspondence between characters and numbers. In particularly, the ASCII standard allocates up to **one byte per each character**. Since the maximum number that can be stored into one byte is 255 (because $256 = 2^8 = 100000000$), ASCII can contain only up to 256 characters. Actually, it contains only 128 characters by convention. Hence, the ASCII is a **7-bit encoding**.

Figure 79 shows the current ASCII table, with the correspondence between characters and numbers.

ASCII allows to represent most of the characters present in Latin-based alphabets, but misses several alphabets.

The 8-bit Unicode Transformation Format (UTF-8) uses up to 4 bytes to store characters and includes all the characters present in Latin-based alphabets, Greek, Cyrillic, Arabic, and several other alphabets. UTF-8 also includes most characters from, e.g., Chinese, Japanese and Korean. For this reason, UTF-8 is increasingly popular and might replace ASCII as standard.

# ASCII TABLE

| Decimal | Hexadecimal | Binary | Octal | Char | Decimal | Hexadecimal | Binary | Octal | Char | Decimal | Hexadecimal | Binary | Octal | Char |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | *[NULL]* | 48 | 30 | 110000 | 60 | **0** | 96 | 60 | 1100000 | 140 | ` |
| 1 | 1 | 1 | 1 | *[START OF HEADING]* | 49 | 31 | 110001 | 61 | **1** | 97 | 61 | 1100001 | 141 | **a** |
| 2 | 2 | 10 | 2 | *[START OF TEXT]* | 50 | 32 | 110010 | 62 | **2** | 98 | 62 | 1100010 | 142 | **b** |
| 3 | 3 | 11 | 3 | *[END OF TEXT]* | 51 | 33 | 110011 | 63 | **3** | 99 | 63 | 1100011 | 143 | **c** |
| 4 | 4 | 100 | 4 | *[END OF TRANSMISSION]* | 52 | 34 | 110100 | 64 | **4** | 100 | 64 | 1100100 | 144 | **d** |
| 5 | 5 | 101 | 5 | *[ENQUIRY]* | 53 | 35 | 110101 | 65 | **5** | 101 | 65 | 1100101 | 145 | **e** |
| 6 | 6 | 110 | 6 | *[ACKNOWLEDGE]* | 54 | 36 | 110110 | 66 | **6** | 102 | 66 | 1100110 | 146 | **f** |
| 7 | 7 | 111 | 7 | *[BELL]* | 55 | 37 | 110111 | 67 | **7** | 103 | 67 | 1100111 | 147 | **g** |
| 8 | 8 | 1000 | 10 | *[BACKSPACE]* | 56 | 38 | 111000 | 70 | **8** | 104 | 68 | 1101000 | 150 | **h** |
| 9 | 9 | 1001 | 11 | *[HORIZONTAL TAB]* | 57 | 39 | 111001 | 71 | **9** | 105 | 69 | 1101001 | 151 | **i** |
| 10 | A | 1010 | 12 | *[LINE FEED]* | 58 | 3A | 111010 | 72 | **:** | 106 | 6A | 1101010 | 152 | **j** |
| 11 | B | 1011 | 13 | *[VERTICAL TAB]* | 59 | 3B | 111011 | 73 | **;** | 107 | 6B | 1101011 | 153 | **k** |
| 12 | C | 1100 | 14 | *[FORM FEED]* | 60 | 3C | 111100 | 74 | **<** | 108 | 6C | 1101100 | 154 | **l** |
| 13 | D | 1101 | 15 | *[CARRIAGE RETURN]* | 61 | 3D | 111101 | 75 | **=** | 109 | 6D | 1101101 | 155 | **m** |
| 14 | E | 1110 | 16 | *[SHIFT OUT]* | 62 | 3E | 111110 | 76 | **>** | 110 | 6E | 1101110 | 156 | **n** |
| 15 | F | 1111 | 17 | *[SHIFT IN]* | 63 | 3F | 111111 | 77 | **?** | 111 | 6F | 1101111 | 157 | **o** |
| 16 | 10 | 10000 | 20 | *[DATA LINK ESCAPE]* | 64 | 40 | 1000000 | 100 | **@** | 112 | 70 | 1110000 | 160 | **p** |
| 17 | 11 | 10001 | 21 | *[DEVICE CONTROL 1]* | 65 | 41 | 1000001 | 101 | **A** | 113 | 71 | 1110001 | 161 | **q** |
| 18 | 12 | 10010 | 22 | *[DEVICE CONTROL 2]* | 66 | 42 | 1000010 | 102 | **B** | 114 | 72 | 1110010 | 162 | **r** |
| 19 | 13 | 10011 | 23 | *[DEVICE CONTROL 3]* | 67 | 43 | 1000011 | 103 | **C** | 115 | 73 | 1110011 | 163 | **s** |
| 20 | 14 | 10100 | 24 | *[DEVICE CONTROL 4]* | 68 | 44 | 1000100 | 104 | **D** | 116 | 74 | 1110100 | 164 | **t** |
| 21 | 15 | 10101 | 25 | *[NEGATIVE ACKNOWLEDGE]* | 69 | 45 | 1000101 | 105 | **E** | 117 | 75 | 1110101 | 165 | **u** |
| 22 | 16 | 10110 | 26 | *[SYNCHRONOUS IDLE]* | 70 | 46 | 1000110 | 106 | **F** | 118 | 76 | 1110110 | 166 | **v** |
| 23 | 17 | 10111 | 27 | *[ENG OF TRANS. BLOCK]* | 71 | 47 | 1000111 | 107 | **G** | 119 | 77 | 1110111 | 167 | **w** |
| 24 | 18 | 11000 | 30 | *[CANCEL]* | 72 | 48 | 1001000 | 110 | **H** | 120 | 78 | 1111000 | 170 | **x** |
| 25 | 19 | 11001 | 31 | *[END OF MEDIUM]* | 73 | 49 | 1001001 | 111 | **I** | 121 | 79 | 1111001 | 171 | **y** |
| 26 | 1A | 11010 | 32 | *[SUBSTITUTE]* | 74 | 4A | 1001010 | 112 | **J** | 122 | 7A | 1111010 | 172 | **z** |
| 27 | 1B | 11011 | 33 | *[ESCAPE]* | 75 | 4B | 1001011 | 113 | **K** | 123 | 7B | 1111011 | 173 | **{** |
| 28 | 1C | 11100 | 34 | *[FILE SEPARATOR]* | 76 | 4C | 1001100 | 114 | **L** | 124 | 7C | 1111100 | 174 | **\|** |
| 29 | 1D | 11101 | 35 | *[GROUP SEPARATOR]* | 77 | 4D | 1001101 | 115 | **M** | 125 | 7D | 1111101 | 175 | **}** |
| 30 | 1E | 11110 | 36 | *[RECORD SEPARATOR]* | 78 | 4E | 1001110 | 116 | **N** | 126 | 7E | 1111110 | 176 | **~** |
| 31 | 1F | 11111 | 37 | *[UNIT SEPARATOR]* | 79 | 4F | 1001111 | 117 | **O** | 127 | 7F | 1111111 | 177 | *[DEL]* |
| 32 | 20 | 100000 | 40 | *[SPACE]* | 80 | 50 | 1010000 | 120 | **P** | | | | | |
| 33 | 21 | 100001 | 41 | **!** | 81 | 51 | 1010001 | 121 | **Q** | | | | | |
| 34 | 22 | 100010 | 42 | **"** | 82 | 52 | 1010010 | 122 | **R** | | | | | |
| 35 | 23 | 100011 | 43 | **#** | 83 | 53 | 1010011 | 123 | **S** | | | | | |
| 36 | 24 | 100100 | 44 | **$** | 84 | 54 | 1010100 | 124 | **T** | | | | | |
| 37 | 25 | 100101 | 45 | **%** | 85 | 55 | 1010101 | 125 | **U** | | | | | |
| 38 | 26 | 100110 | 46 | **&** | 86 | 56 | 1010110 | 126 | **V** | | | | | |
| 39 | 27 | 100111 | 47 | **'** | 87 | 57 | 1010111 | 127 | **W** | | | | | |
| 40 | 28 | 101000 | 50 | **(** | 88 | 58 | 1011000 | 130 | **X** | | | | | |
| 41 | 29 | 101001 | 51 | **)** | 89 | 59 | 1011001 | 131 | **Y** | | | | | |
| 42 | 2A | 101010 | 52 | ***** | 90 | 5A | 1011010 | 132 | **Z** | | | | | |
| 43 | 2B | 101011 | 53 | **+** | 91 | 5B | 1011011 | 133 | **[** | | | | | |
| 44 | 2C | 101100 | 54 | **,** | 92 | 5C | 1011100 | 134 | **** | | | | | |
| 45 | 2D | 101101 | 55 | **-** | 93 | 5D | 1011101 | 135 | **]** | | | | | |
| 46 | 2E | 101110 | 56 | **.** | 94 | 5E | 1011110 | 136 | **^** | | | | | |
| 47 | 2F | 101111 | 57 | **/** | 95 | 5F | 1011111 | 137 | **_** | | | | | |

*Figure 79: ASCII table.*

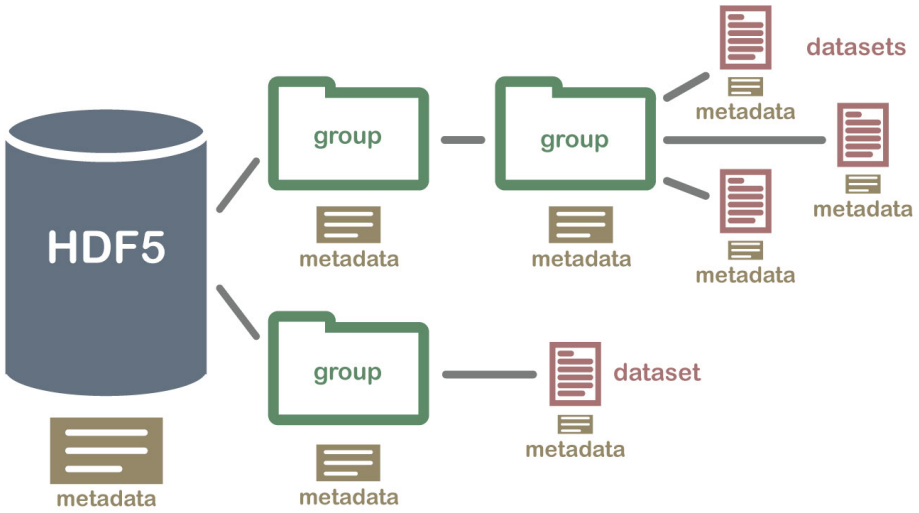


*Figure 80: Cartoon of the structure of an HDF5 file.*

**HDF5 file format:**

HDF5 (Hierarchical Data Format version 5) is a powerful **open-source binary file format for big data**. It is increasingly used for astrophysical data (e.g., LOFAR data) and for astrophysical simulations (e.g., the EAGLE simulation project, Schaye et al. 2015). Python has a user-friendly package to read HDF5, called **h5py** (`https://docs.h5py.org/en/stable/quick.html`). It is already installed in conda, otherwise you can install it with

```
conda install h5py
```

An HDF5 file is a container for two kinds of objects: datasets, which are array-like collections of data, and groups, which are folder-like containers that hold datasets and other groups. The most fundamental thing to remember when using h5py is: **Groups work like dictionaries, and datasets work like numPy arrays**.

You can think of a HDF5 file as a well-organized bundle of directories and files contained in these directories, all stored in the same file. The groups of the HDF5 file have the same role as the directories, while the datasets are equivalent to the files. To summarize (see also Figure 80):

- **Group:** A folder like element within an HDF5 file that might contain other groups OR datasets within it.
- **Dataset:** The actual data contained within the HDF5 file. Datasets are often (but don't have to be) stored within groups in the file.

In practice, the first thing to do is to open a file for reading

```
import h5py
f = h5py.File('mytestfile.hdf5', 'r')
```

The File object is your starting point. What is stored in this file?

Check the lecture on python dictionaries. Groups are like Python dictionaries, thus we can check the keys,

```
key = list(f.keys())
print(key)
```

This command prints a list whose elements are the keys of the dictionary. We can use them to access the values of the dictionary, which are our dataset object:

```
a = list(f.keys())[0]
dset = f[a]
```

In the following example (**examples/hdf5/read_hdf5.py**), we read one of the HDF5 files of the EAGLE cosmological simulation (`http://icc.dur.ac.uk/Eagle/`). Note that if the dictionary is particularly nested / complex, the most effective way to proceed is to know what is the organization of the keys (e.g., in this case you can know it by reading the documentation of the EAGLE simulation).

```
#examples/hdf5/read_hdf5.py
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as colors
import h5py as h5

#read a snapshot of a galaxy from EAGLE
Gnr = 7  #group
Sgrn = 0 #subgroup

fname="GalaxySample_Gnr"+str(Gnr)+"Sgrn"+str(Sgrn)+"_ce1515R0.5_GW_DNS_25Mpc_merge28.h5"

f = h5.File(fname, "r")


print(list(f.keys()))

# the first key is the galaxy label
Galaxy_label = list(f.keys())[0]
#Galaxy_label = "Group_"+str(Gnr)+"SubGroup_"+str(Sgrn)


# the further keys are the properties of the galaxy
# here we specify the coordinates of each star and the mass
coord = "Coordinates_stars"
mass = "mstars"

#print(f[Galaxy_label][mass])

m_stars = np.asarray(f[Galaxy_label][mass]) #Mass in 10^10Msunh
pos_stars = np.asarray(f[Galaxy_label][coord]) #Coordinates in Mpc/h
x_stars = pos_stars[:,0]
y_stars = pos_stars[:,1]
z_stars = pos_stars[:,2]

mtot=0.0
xcm=ycm=zcm=0.0
mtot=sum(m_stars)
xcm=sum(x_stars*m_stars)/mtot
ycm=sum(y_stars*m_stars)/mtot
zcm=sum(z_stars*m_stars)/mtot

x_stars-=xcm
y_stars-=ycm
z_stars-=zcm

print(xcm)
print(x_stars)
#plt.scatter(x_stars,y_stars)
lims=[[-0.5,0.5],[-0.23,0.3]]

plt.hist2d(x_stars,y_stars,range=lims,normed=True,bins=200,norm=colors.LogNorm())
plt.xlabel("x [Mpc/h]")
plt.ylabel("y [Mpc/h]")
plt.show()
```

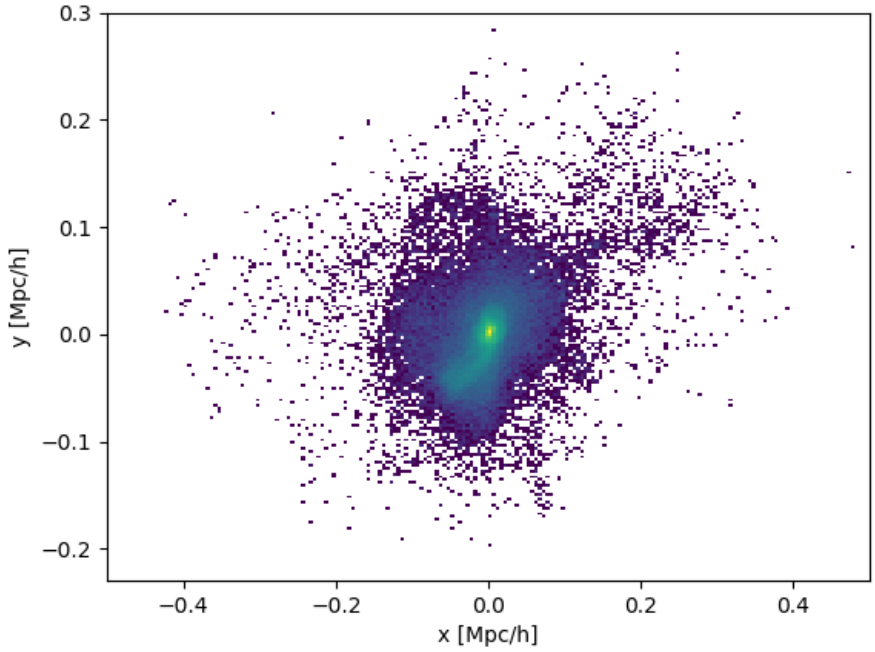


*Figure 81: Projected positions of the stellar particles of a Milky-Way galaxy at $z = 0$ from the* EAGLE *simulation.*

This script produces Figure 81.

**JSON file format:**

JSON stands for JavaScript Object Notation, hence it is based on the JavaScript programming language and is particularly optimized for data exchanges between client and server, i.e. through the web. It is a **text file**. Its format is independent of Java, so that JSON can be read and written also in other programming languages. Overall, a JSON is similar to a python **dictionary**. It is used by astrophysicists when large databases are shared via web, for example by the ILLUSTRIS cosmological simulation project [Vogelsberger et al., 2014].

A JSON file can be read in python with the **json package**, which basically converts the JSON to a dictionary. In order to read a JSON file, you need to know the keys of the "dictionary": this depends on your specific case. In contrast, converting a python dictionary to a json file is trivial: you define a dictionary and then convert it with the function **json.dumps(dictionaryname)**. For example (see the file **examples/json/json_write_read.py**),

```
#examples/json/json_write_read.py
import json

##python dictionary
mycat = {
  "name": "Ettore",
  "age": 10.0,
  "color": "white-read",
  "spots": "tabby"
}

#transform python dict into json (without writing file)
print(json.dumps(mycat))

#transform python dict into json and writes file
f=open('mycat.json','w',encoding='utf-8')
json.dump(mycat,f, ensure_ascii=False, indent=4)
f.close()

#reads from previously created file
f=open('mycat.json')
data = json.load(f)
f.close()

print(data['name']) #print the value associated with key name
print(data['age']) #print the value associated with key age
```

**CSV file format:**

CSV stands for **comma-separated values**. A CSV is a **text file** that uses commas to separate values. It is commonly adopted for tabular data with numbers and text. The ideal way to read a CSV file in python is via **pandas**. Hence, we will talk more about CSV reading and writing when we discuss pandas.

**FITS file format:**

The Flexible Image Transport System (FITS) is an open-source file format useful for storage, transmission and processing of data: formatted as multi-dimensional arrays (for example a 2D image), or tables. FITS is the most commonly used digital file format in astronomy.

Every FITS is composed of one or more **headers**, containing metadata, and of one or more **data objects**. The header is usually a human readable ASCII text. A FITS file may contain several extensions, and each of these may contain a data object. For example, it is possible to store x-ray and infrared exposures in the same file. FITS is also often used to store non-image data, such as spectra, photon lists, data cubes, or structured data such as multi-table databases. The astropy package **astropy.io.fits** is warmly suggested to read

and create FITS files in python.

## 15.4 *Scipy*

Scipy (`https://www.scipy.org/`) is a python based ecosystem of open-source software for mathematics, science, and engineering. Some of the core packages are:

- numpy: no need to add more;
- matplotlib: no need to add more;
- scipy library: it provides many user-friendly and efficient numerical routines, such as routines for numerical integration, interpolation, optimization, linear algebra, and statistics. We have seen many of them during this course. See the documentation: `https://docs.scipy.org/doc/`;
- IPython: an architecture for interactive computing (`http://ipython.org/`). It initially included jupyter, which is now on its own;
- SymPy: a python library for symbolic mathematics (`https://www.sympy.org/`). It aims to become a full-featured computer algebra system (CAS) while keeping the code as simple as possible in order to be comprehensible and easily extensible. SymPy is used, e.g., by Spyder;
- pandas: an open source data analysis and manipulation tool, built on top of the Python programming language (`https://pandas.pydata.org/`). We will talk about pandas later in this chapter.

Out of this list of packages, the **scipy library** is what we usually refer to, when we talk about scipy. We have already seen many examples of scipy, including scipy.optimize (to do fits), scipy.interpolate (to interpolate functions), scipy.integrate (to integrate functions). It is definitely my preferred python package after numpy and matplotlib. It is impossible to discuss all the features of scipy here: just browse them when you need some new functions to do some operation/analysis.

## 15.5 *Astropy*

The package astropy `https://www.astropy.org/` collects several different sub-packages intended for the usage of professional astrophysicists. The sub-packages do very different tasks. An example of the most commonly used sub-packages can be found here below.

- **astropy.cosmology:** contains classes for representing cosmologies and utility functions for calculating commonly used quantities that depend on a cosmological model. This includes distances (proper, comoving and angular distances), ages, and lookback times corresponding to a measured redshift.

| Name | Reference | $H_0$ [km $s^{-1}$ $Mpc^{-1}$] | $\Omega_M$ | Flat |
|---|---|---|---|---|
| WMAP5 | Komatsu et al. 2009 | 70.2 | 0.277 | Yes |
| WMAP7 | Komatsu et al. 2011 | 70.4 | 0.272 | Yes |
| WMAP9 | Hinshaw et al. 2013 | 69.3 | 0.287 | Yes |
| Planck13 | Planck Collab 2013, Paper XVI | 67.8 | 0.307 | Yes |
| Planck15 | Planck Collab 2015, Paper XIII | 67.7 | 0.307 | Yes |

The cosmological parameters are currently provided for five different sets of measure as you can see from **cosmology.parameters.available** and from Table 15.5. Of course, using the most recent cosmology is the better choice, unless you are working with something that was obtained with a different cosmology (e.g., the outputs of a cosmological simulation). For example, if I need to analyze the data from the ILLUSTRIS cosmological simulations [Vogelsberger et al., 2014], I have to use WMAP9 for consistency, because this simulation was run with WMAP9 cosmology.

Here is a simple example of usage of this package (see **examples/astropy/simple_cosmology.py**):

```
>>> from astropy.cosmology import Planck15 as cosmo
>>> cosmo.H(0.0) # prints H_0
<Quantity 67.74 km / (Mpc s)>
>>> cosmo.comoving_distance([0.5, 1.0, 1.5])
>>> #comoving distance at z=0.5,1.0,1.5
<Quantity [1945.56126208, 3395.90531198, 4479.04518283] Mpc>
>>> cosmo.luminosity_distance(4.)  #luminosity distance at z=4
 <Quantity 36697.036387 Mpc>
>>> cosmo.age(0) # age of the Universe at z=0
<Quantity 13.7976159 Gyr>
>>> cosmo.age(6.) # age of the Universe at z=6
<Quantity 0.93139078 Gyr>
cosmo.lookback_time(0.8) # loockback time at z=0.8
<Quantity 7.02634875 Gyr>
```

- **astropy.constants** contains the definition of some of the most important astrophysical quantities. For example (see **examples/astropy/simple_constants.py**):

```
>>> from astropy import constants as const

>>> const.G
<<class 'astropy.constants.codata2018.CODATA2018'>
name='Gravitational constant' value=6.6743e-11
uncertainty=1.5e-15 unit='m3 / (kg s2)' reference='CODATA 2018'>

>>> const.G.cgs
<Quantity 6.6743e-08 cm3 / (g s2)>

>>> const.c
<<class 'astropy.constants.codata2018.CODATA2018'>
name='Speed of light in vacuum' value=299792458.0
uncertainty=0.0 unit='m / s' reference='CODATA 2018'>

>>> const.M_sun
<<class 'astropy.constants.iau2015.IAU2015'> name='Solar mass'
value=1.988409870698051e+30 uncertainty=4.468805426856864e+25
unit='kg' reference='IAU 2015 Resolution B 3 + CODATA 2018'>

>>> const.L_sun
<<class 'astropy.constants.iau2015.IAU2015'>
name='Nominal solar luminosity' value=3.828e+26 uncertainty=0.0
unit='W' reference='IAU 2015 Resolution B 3'>

>>> const.e
<<class 'astropy.constants.codata2018.EMCODATA2018'>
name='Electron charge' value=1.602176634e-19 uncertainty=0.0
unit='C' reference='CODATA 2018'>

>>> const.h
<<class 'astropy.constants.codata2018.CODATA2018'> name='Planck constant'
value=6.62607015e-34 uncertainty=0.0 unit='J s' reference='CODATA 2018'>
```

I can know the value and the units of a constant by typing .value and .unit. For example,

```
>>> cons.G.cgs.value
6.674299999999999e-08
>>> cons.G.cgs.unit
Unit("cm3 / (g s2)")
```

On the astropy webpage you can find a complete list of the constants included in this sub-package (`https://docs.astropy.org/en/stable/constants/index.html`).

- **astropy.units** handles some common units and unit conversions.

  Here below, you see an example of how to associate a quantity with units (see **examples/astropy/simple_units.py**):

```
>>> import numpy as np
>>> from astropy import units as u
>>> 42.0 * u.meter
<Quantity  42. m>
>>> [1., 2., 3.] * u.m
<Quantity [1., 2., 3.] m>
>>> np.array([1., 2., 3.]) * u.m
<Quantity [1., 2., 3.] m>
```

  You can get the unit and value using the unit and value commands:

```
>>> l = 42.0 * u.meter
>>> l.value #value of l
42.0
>>> l.unit #units of l
Unit("m")
```

  Unit conversions are done using the **.to()** function:

```
>>> x = 1.0 * u.parsec
>>> x.to(u.cm)
<Quantity 3.08567758e+18 cm>
```

  In several cases, especially if you are working with N−body simulations, you want to work with **dimensionless quantities**. To make a quantity dimensionless you have to use the dimensionless_unscaled unit, e.g.:

```
>>> l = 1.0 * u.dimensionless_unscaled
>>> l.unit
Unit(dimensionless)
```

  note that there is no other way in astropy to get rid of units after you assign them.

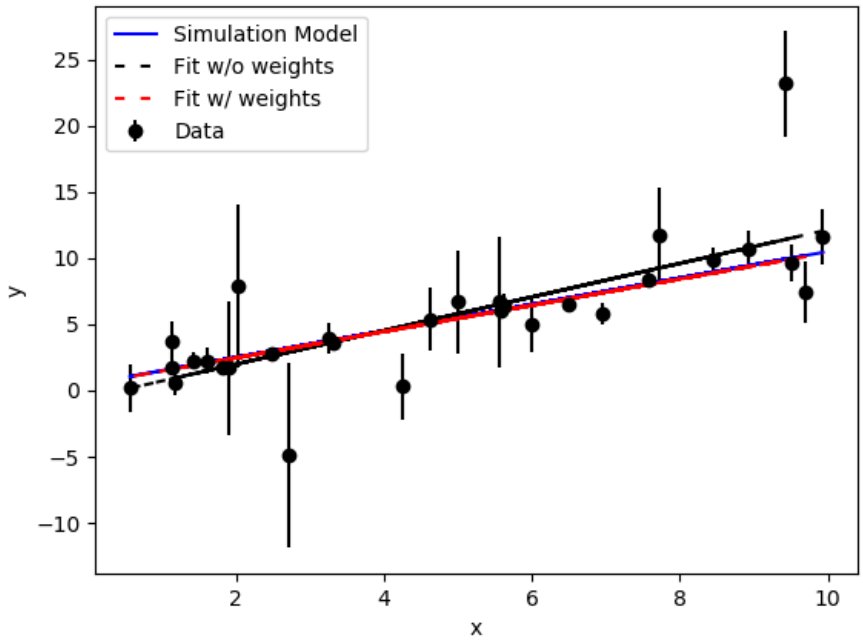


*Figure 82: Least square fit with astropy.modeling.*

- **astropy.io.fits:** astropy.io takes care of reading from files and printing to files (IO stands for input output, as usual). The .fits package is designed to read FITS files, which are particularly common among observers. The basic commands to open a FITS file with astropy are:

```
from astropy.io import fits

fits_image_filename = fits.util.get_testdata_filepath('test0.fits')
mydata = fits.open(fits_image_filename)
mydata.info()
```

  where .info() displays the header of the FITS file. See the example: **examples/astropy/simple_fits.py**.

- **astropy.modeling** provides several tools to do fits. For example, you can see below an alternative to the least-square fit methods of scipy: the function **astropy.modeling.fitting.LinearLSQFitter()** that can be used with or without weights. The example below is adapted from the astropy web page. You can find it also in **examples/astropy/fit_astropy.py**. The result is shown in Fig. 82.

```
import numpy as np
import matplotlib.pyplot as plt
from astropy.modeling import models, fitting

# define a model for a line
line_orig = models.Linear1D(slope=1.0, intercept=0.5)

# generate x, y data non-uniformly spaced in x
# add noise to y measurements
npts = 30
x = np.random.uniform(0.0, 10.0, npts)
y = line_orig(x)
yunc=np.zeros(npts,float) # error array

np.random.seed(10)
yunc[0:int(npts/2)] = np.absolute(np.random.normal(0.5, 1.5, int(npts/2)))
#half of the data have "good" measurement errors
np.random.seed(17)
yunc[int(npts/2):npts] = np.absolute(np.random.normal(0.5, 3., int(npts/2)))
#half of the data have "bad" measurement errors

y += np.random.normal(0.0, yunc, npts)

# initialize a linear fitter
fit = fitting.LinearLSQFitter()

# initialize a linear model
line_init = models.Linear1D()

# fit the data with the fitter without weights
fitted_line1 = fit(line_init, x, y)

# fit the data with the fitter with weights
fitted_line2 = fit(line_init, x, y, weights=1.0/yunc)

# plot
plt.figure()
plt.errorbar(x, y, yerr=yunc, fmt='ko', label='Data')
plt.plot(x, line_orig(x), 'b-', label='Simulation Model')
plt.plot(x, fitted_line1(x), 'k',linestyle='--', dashes=(3, 5), \
label='Fit w/o weights')
plt.plot(x, fitted_line2(x), 'r',linestyle='--', dashes=(3, 5), \
label='Fit w/ weights')
plt.xlabel('x')
plt.ylabel('y')
plt.legend(loc="upper left")
plt.show()
```

You can also do **sigma clipping:** when fitting, there may be data that are outliers from the fit that can significantly bias the fitting. These outliers can be identified and removed from the fitting iteratively. Note that the iterative sigma clipping assumes all the data have the same uncertainties for the sigma clipping decision. The example below is

taken from astropy web page. Fig. 83 shows the result. You can find this file in **examples/astropy/fit_astropy_sigmaclip.py**

```
import numpy as np
import matplotlib.pyplot as plt
from astropy.stats import sigma_clip
from astropy.modeling import models, fitting

# define a model for a line
line_orig = models.Linear1D(slope=1.0, intercept=0.5)

# generate x, y data non-uniformly spaced in x
# add noise to y measurements
npts = 30
np.random.seed(10)
x = np.random.uniform(0.0, 10.0, npts)
y = line_orig(x)
yunc = np.absolute(np.random.normal(0.5, 2.5, npts))
y += np.random.normal(0.0, yunc, npts)

# make true outliers
y[3] = line_orig(x[3]) + 6 * yunc[3]
y[10] = line_orig(x[10]) - 4 * yunc[10]

# initialize a linear fitter
fit = fitting.LinearLSQFitter()

# initialize the outlier removal fitter
or_fit = fitting.FittingWithOutlierRemoval(fit, sigma_clip, niter=3, sigma=3.0)

# initialize a linear model
line_init = models.Linear1D()

# fit the data with the fitter
fitted_line2 = fit(line_init, x, y)
fitted_line, mask = or_fit(line_init, x, y, weights=1.0/yunc)
filtered_data = np.ma.masked_array(y, mask=mask)

# plot
plt.figure()
plt.errorbar(x, y, yerr=yunc, fmt="ko", fillstyle="none", \
label="Clipped Data")
plt.plot(x, filtered_data, "ko", label="Fitted Data")
plt.plot(x, line_orig(x), 'b-', label='Simulation Model')
plt.plot(x, fitted_line(x), 'k',linestyle='--', dashes=(3, 5), \
label='Fitted Model')
plt.plot(x, fitted_line2(x), 'r',linestyle='--', dashes=(3, 5), \
label='Fitted Model w/ sigmaclip')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```

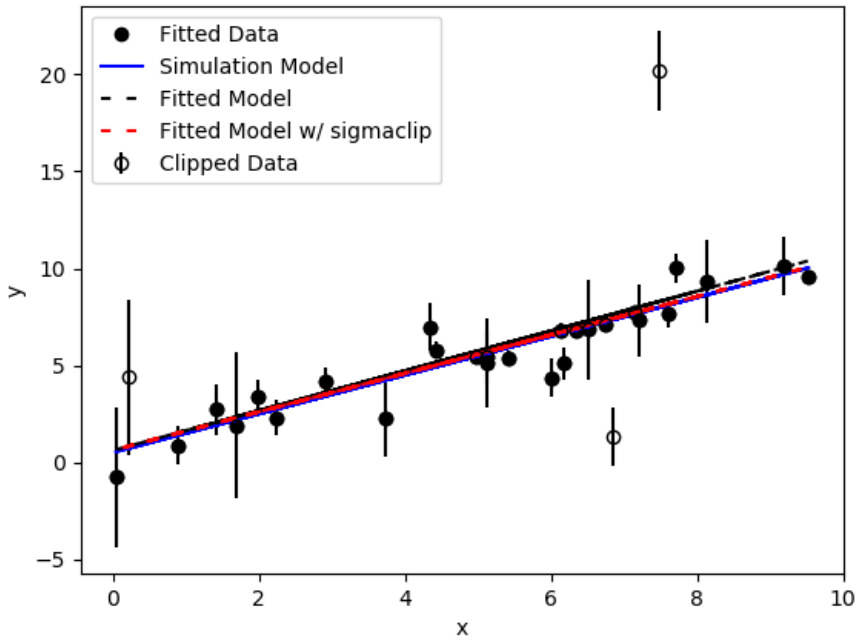


*Figure 83: Least square fit with astropy.modeling and sigma clipping.*

- **astropy.convolution** provides convolution functions and kernels that offer improvements compared to the SciPy scipy.signal convolution routines, especially when astrophysical images are concerned.

  Convolution here means masking some pixels of an image that are "wrong" and substituting their value with a new value interpolated between neighboring pixels with a kernel function. A pixel could be "wrong" when, e.g., it is affected by a cosmic ray, or when it is saturated because of a very bright star, or when that specific pixel is dead on the CCD.

  The improvements of astropy.convolution with respect to scipy.signal are:

  - Proper treatment of NaN values (ignoring them during convolution and replacing NaN pixels with interpolated values)
  - A single function for 1D, 2D, and 3D convolution
  - Improved options for the treatment of edges
  - Both direct and Fast Fourier Transform (FFT) versions
  - Built-in kernels that are commonly used in Astronomy

  The following thumbnails show the difference between scipy.signal.convolve and astropy.convolution functions on an astronomical image that con-

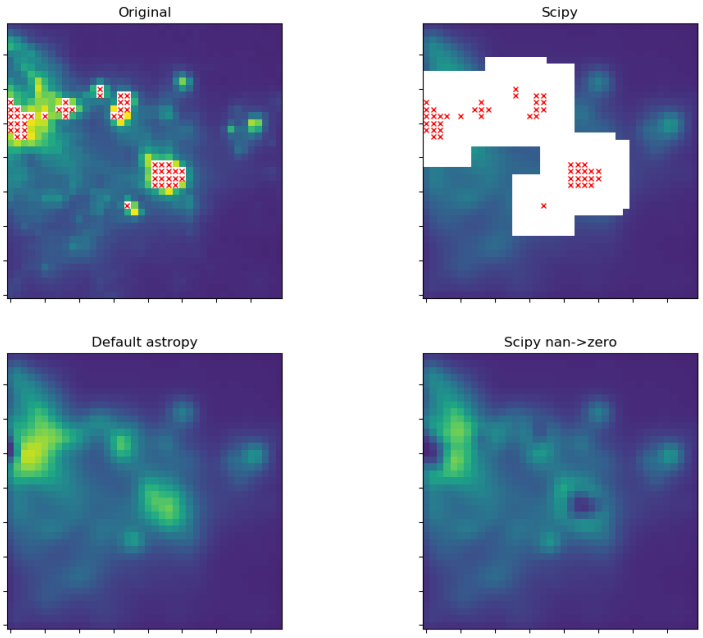


*Figure 84: Data reconstruction with astropy.convolution. Upper left: original image. The red crosses indicate pixels that contain a NaN value. Lower left: reconstruction of the image with astropy.convolution, which masks NaN pixels and substitutes them with an interpolation of the values of the neighboring pixels (in this case with a 2D Gaussian kernel density). Upper right: reconstruction of the image with scipy.signal.colvolve. Lower right: reconstruction of the image with scipy.signal.convolve after the pixels with value NaN are replaced with pixels of value 0.*

tains NaN values. scipy's function essentially returns NaN for all pixels that are within a kernel of any NaN value, which is often not the desired result. This example is taken from `https://docs.astropy.org/en/stable/convolution/index.html`. More details on how to perform convolution can be found at the aforementioned link. Figure 84 shows the outputs.

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

from astropy.io import fits
from astropy.utils.data import get_pkg_data_filename
from astropy.convolution import Gaussian2DKernel
from scipy.signal import convolve as scipy_convolve
from astropy.convolution import convolve


# Load the data from data.astropy.org
filename = get_pkg_data_filename('galactic_center/gc_msx_e.fits')
hdu = fits.open(filename)[0]

# Scale the file to have reasonable numbers
# (this is mostly so that colorbars do not have too many digits)
# Also, we crop it so you can see individual pixels
img = hdu.data[50:90, 60:100] * 1e5

# This example is intended to demonstrate how astropy.convolve and
# scipy.convolve handle missing data, so we start by setting the
# brightest pixels to NaN to simulate a "saturated" data set
img[img > 2e1] = np.nan

# We also create a copy of the data and set those NaNs to zero.  We will
# use this for the scipy convolution
img_zerod = img.copy()
img_zerod[np.isnan(img)] = 0

# We smooth with a Gaussian kernel with x_stddev=1 (and y_stddev=1)
# It is a 9x9 array
kernel = Gaussian2DKernel(x_stddev=1)

# Convolution: scipy's direct convolution mode spreads out NaNs (see
# panel 2 below)
scipy_conv = scipy_convolve(img, kernel, mode='same', method='direct')

# scipy's direct convolution mode run on the 'zero'd' image will not
# have NaNs, but will have some very low value zones where the NaNs were
# (see panel 3 below)
scipy_conv_zerod = scipy_convolve(img_zerod, kernel, mode='same',
                                  method='direct')

# astropy's convolution replaces the NaN pixels with a kernel-weighted
# interpolation from their neighbors
astropy_conv = convolve(img, kernel)

# Now we do a bunch of plots.  In the first two plots, the originally masked
# values are marked with red X's
plt.figure(1, figsize=(12, 12)).clf()
ax1 = plt.subplot(2, 2, 1)
im = ax1.imshow(img, vmin=-2., vmax=2.e1, origin='lower',
                interpolation='nearest', cmap='viridis')
y, x = np.where(np.isnan(img))
ax1.set_autoscale_on(False)
ax1.plot(x, y, 'rx', markersize=4)
ax1.set_title("Original")
ax1.set_xticklabels([])
ax1.set_yticklabels([])
```

```
ax2 = plt.subplot(2, 2, 2)
im = ax2.imshow(scipy_conv, vmin=-2., vmax=2.e1, origin='lower',
                interpolation='nearest', cmap='viridis')
ax2.set_autoscale_on(False)
ax2.plot(x, y, 'rx', markersize=4)
ax2.set_title("Scipy")
ax2.set_xticklabels([])
ax2.set_yticklabels([])

ax4 = plt.subplot(2, 2, 3)
im = ax4.imshow(astropy_conv, vmin=-2., vmax=2.e1, origin='lower',
                interpolation='nearest', cmap='viridis')
ax4.set_title("Default astropy")
ax4.set_xticklabels([])
ax4.set_yticklabels([])

ax3 = plt.subplot(2, 2, 4)
im = ax3.imshow(scipy_conv_zerod, vmin=-2., vmax=2.e1, origin='lower',
                interpolation='nearest', cmap='viridis')
ax3.set_title("Scipy nan->zero")
ax3.set_xticklabels([])
ax3.set_yticklabels([])

plt.show()
```

The above example can be found in **examples/astropy/astropy_convolution.py**.

- **astropy.coordinates** provides tools to use the main astrophysical coordinate systems (RA and DEC in different flavours).

## 15.6 *Introduction to PANDAS*

Pandas (`https://pandas.pydata.org/`) is a python package. It is particularly designed for (big) data analysis. The name **pandas** is derived from the term "panel data", an econometrics term for data sets that include observations over multiple time periods for the same individuals" [source Wes McKinney (2011): "pandas: a Foundational Python Library for Data Analysis and Statistics", `https://www.dlr.de/sc/Portaldata/15/Resources/dokumente/pyhpc2011/submissions/pyhpc2011_submission_9.pdf`].

Pandas should be installed by default with anaconda, but if your conda installation does not include pandas you can type

```
conda install pandas
```

The two primary components of pandas are the **Series** and **DataFrame**. A Series is essentially a single column, and a DataFrame is a multi-dimensional table made up of a collection of Series.

The simple lines of code that we will comment in the following Sections can

be found in **examples/pandas/simple_pandas.py**, or, if you prefer jupyter-notebook, in **examples/pandas/simple_pandas.ipynb**.

### 15.6.1 Series

You can create a Series starting **from a python dictionary, from a list or from a numpy array**. In the list case

```
import pandas as pd

a=[17.,13.,11.,0.]
s1 = pd.Series(a)
print(s1)
```

The output is

```
0    17.0
1    13.0
2    11.0
3     0.0
dtype: float64
```

The second column of the output contains the values from the list: these are the values of the series now. The first column contains the indexes of the list values. In the default version, the indexes are integers from 0 on. The last row specifies the data type: our list was a list of floats hence the data type is "float64". If I had a list of mixed types (e.g. b=[17.,13.,'ciao','meow']), the Series would have a 'dtype: object'.

I can decide the indexes as I want, for example:

```
import pandas as pd

a=[17.,13.,11.,0.]
ind=['a','b','c','d']
s1 = pd.Series(a,index=ind)
print(s1)
```

Alternatively, we can create a series from a dictionary (see the properties of dictionaries from the chapter about python):

```
import pandas as pd

mycat = {
   "color": "red",
   "fur": "short",
   "spots": "tabby"
}

s2=pd.Series(mycat)
print(s2)
```

This print returns

```
color      red
fur      short
spots    tabby
dtype: object
```

that is the keys of the dictionary are now indexes and the values of the dictionary are values of the series.

I can change the indexes of the series as many times as I want with the .index command.

```
s2.index= ["qui","quo","qua"]
print(s2)
```

which yields

```
qui      red
quo    short
qua    tabby
dtype: object
```

We can slice a Series in the same exact way as a numpy array:

```
import pandas as pd

a=[17.,13.,11.,0.]
s1 = pd.Series(a)
b= s1[:3]
print(b)
```

The output is

```
0    17.0
1    13.0
2    11.0
dtype: float64
```

We can use .append() as we did for lists:

```
import pandas as pd

a=[17.,13.,11.,0.]
b=[0.,2.,5.]
s1 = pd.Series(a)
s2 = pd.Series(b)
s3 = s1.append(s2)
print(s3)
```

The output is

```
0    17.0
1    13.0
2    11.0
3     0.0
0     0.0
1     2.0
2     5.0
dtype: float64
```

And I can remove elements from the Series with .drop

```
import pandas as pd

a=[17.,13.,11.,0.]
s1 = pd.Series(a,index=["a","b","c","d"])
s1 = s1.drop("d")
print(s1)
```

The output is

```
a    17.0
b    13.0
c    11.0
dtype: float64
```

I can do **operations on Series**.

- **.add()** sums up two series

```
a=[17.,13.,11.,0.]
s1 = pd.Series(a)
b=[2.,3.,4.,5.,6.,7.,9.]
s2=pd.Series(b)
s3=s1.add(s2)
print(s3)
```

```
0    19.0
1    16.0
2    15.0
3     5.0
4     NaN
5     NaN
6     NaN
dtype: float64
```

I can do a sum between Series, and the result of the sum looks like a vector sum, similar to numpy arrays, but, unlike numpy arrays, I can sum two series with a different length and the elements for which the sum is not defined are labeled as NaN. Very dangerous.

- **.sub()** subtracts one series from the other;
- **.mul()** multiplies one series by the other;
- **.div()** divides one series by the other;
- **.median()** calculates the median value of a series:

```
s3.median()
15.5
```

Note that median as well as all the other operators **ignores the NaN elements (or the inf elements, if there were some): again, this is dangerous**.

- **.max()** finds the maximum of the series;
- **.min()**finds the minimum of the series.

### 15.6.2 DataFrame

First, how do I create a DataFrame? I need a matrix (e.g. from numpy array) and (if I want) a set of indexes for each row and a set of indexes for each column:

```
>>> import pandas as pd
>>> import numpy as np
>>> index=np.arange(0,3,1)
>>> index
array([0, 1, 2])
>>> columns=["A","B","C","D","E"]
>>> num_arr=np.random.randn(3,5) #creates a 3x5 matrix of random numbers
>>> num_arr
array([[-0.46259011,  0.38687888, -0.91409853, -0.80747588,  2.15977694],
       [ 1.71789861,  0.21272667, -0.42692957,  0.54132848,  0.125578  ],
       [-0.64931233, -0.02896975, -1.79764273,  1.05789862, -0.28482786]])
>>> df1 = pd.DataFrame(num_arr, index=index, columns=columns)
>>> df1
          A         B         C         D         E
0 -0.462590  0.386879 -0.914099 -0.807476  2.159777
1  1.717899  0.212727 -0.426930  0.541328  0.125578
2 -0.649312 -0.028970 -1.797643  1.057899 -0.284828
```

The print looks like a ordered, nice table. It is almost the same as a data sheet. It looks even better if you use jupyter-notebook.

The matrix I obtained above from numpy can be provided also with a dictionary. The dictionary automatically provides the names of the columns, e.g.:

```
data = {'animal' : ['cat','cat','dog','rabbit','cat','dog'],
   'age'    : ['7', '10', '4', '2', '3', '3'],
   'visit day': ['yesterday','today','today','today','tomorrow','tomorrow']}
names = ['Molly','Ben','Rustie','Bax','Vampire','Tornado']
df2 = pd.DataFrame(data,index=names)
```

This yields the dataframe:

```
         animal age  visit day
Molly       cat   7  yesterday
Ben         cat  10      today
Rustie      dog   4      today
Bax      rabbit   2      today
Vampire     cat   3   tomorrow
Tornado     dog   3   tomorrow
```

Let us have a quick overview of the main functions and features of DataFrames.

- The command **.head(n)** allows me to see the first n rows of my dataframe:

  ```
  df2.head(2)
  ```

  prints the rows of Molly and Ben.

- Similarly **.tail(n)** shows the last n rows of my dataframe.

- The commands

```
df2.index
df2.columns
df2.values
```

  give the indexes of the rows, those of the columns and the values of the dataframe. The format of the values is basically that of a numpy array.

- Minimal statistical information can be obtained with **df2.describe()**.

- If you want to transpose the dataframe you can do **df2.T**

- If you want to sort the data, based on one of their values, use **df2.sort_values()**:

```
df2.sort_values(by="age")
```

  where you realize that the sorting is alphabetic and not numeric, because the column age is assumed to be made of strings.

- You can slice a DataFrame as a Series and a numpy array:

```
df2[1:3]
```

  which prints only the first two rows.

- Alternatively, you can use the **.iloc** command to slice a DataFrame:

```
df2.iloc[1:3]
```

  which gives exactly the same output as df2[1:3].

  **.iloc** allows you to pick up also a specific element:

```
df2.iloc[0,1]
```

  gives the zeroth row, first column element (starting from zero, of course), which in our example is 7.

- Or you can slice based on column names:

```
df2[['animal','age']]
```

  which returns only the two requested columns.

  We can use this to make operations on one entire column (or more). For example

```
>>> df2['animal']=df2['age']+df2['visit day']
>>> df2['animal']
Molly      7yesterday
Ben           10today
Rustie         4today
Bax            2today
Vampire     3tomorrow
Tornado     3tomorrow
Name: animal, dtype: object
```

In this case the sum is the sum of strings, because I have no floats or integers, but if I do the same with a numeric column, I can get the algebraic sum.

I can use the same command to generate new columns, for example:

```
>>> df2['New col']=df2['animal']+df2['age']
>>> df2
         animal age  visit day  New col
Molly       cat   7  yesterday     cat7
Ben         cat  10      today    cat10
Rustie      dog   4      today     dog4
Bax      rabbit   2      today  rabbit2
Vampire     cat   3   tomorrow     cat3
Tornado     dog   3   tomorrow     dog3
```

- **.drop(columns=['New col'])** allows me to remove one column:

```
>>> df2 = df2.drop(columns=['New col'])
>>> df2
         animal age  visit day
Molly       cat   7  yesterday
Ben         cat  10      today
Rustie      dog   4      today
Bax      rabbit   2      today
Vampire     cat   3   tomorrow
Tornado     dog   3   tomorrow
```

- You can make a copy of a DataFrame with **.copy()**:

```
df3 = df2.copy()
```

- With **.loc[]**, we can find some specific elements

```
>>> df2.loc[df2["age"] == "3"]
        animal age visit day
Vampire    cat   3  tomorrow
Tornado    dog   3  tomorrow
```

We can combine conditions:

```
>>> df2.loc[(df2["age"] == "3") & (df2["animal"] == "cat")]
        animal age visit day
Vampire    cat   3  tomorrow
```

Note that the & stands for the usual python *and* (&& of C and C++). Similarly, the or condition is given by | instead of the usual python *or* (|| of C and C++).

Moreover, we can actually change the value of a datum in the DataFrame. For example

```
>>> df2.loc['Bax','age']=1.5
>>> df2
         animal  age  visit day
Molly       cat    7  yesterday
Ben         cat   10      today
Rustie      dog    4      today
Bax      rabbit  1.5      today
Vampire     cat    3   tomorrow
Tornado     dog    3   tomorrow
```

- The **.str.contains()** help us finding specific strings in the columns.

```
>>> df2.loc[df2['animal'].str.contains('cat')]
        animal age  visit day
Molly      cat   7  yesterday
Ben        cat  10      today
Vampire    cat   3   tomorrow
```

With the ~ symbol we look only in these values that do not contain the searched string:

```
df2.loc[~df2['animal'].str.contains('cat')]
         animal age visit day
Rustie      dog   4     today
Bax      rabbit   2     today
Tornado     dog   3  tomorrow
```

- The **.median()** calculates the median. For example

```
df2[['age']].median()
age    3.5
dtype: float64
```

- The **.isnull()** function looks for no values, like a NaN.
- **Conditional changes:** based on a condition, I might want to change one or more columns.

```
>>> df2.loc[df2['animal'] == 'cat', 'visit day'] = 'today'
>>> df2
         animal age visit day
Molly       cat   7     today
Ben         cat  10     today
Rustie      dog   4     today
Bax      rabbit   2     today
Vampire     cat   3     today
Tornado     dog   3  tomorrow
```

- **.groupby()** allows to divide data by categories and analyze them separately. E.g.,

```
>>> df2.groupby(['animal']).mean()
             age
animal
cat     6.666667
dog     3.500000
rabbit  2.000000
```

It averages the only quantity it can (ages, which are now seen as floats), divided by every type of animal. In the same way I can do .median(), .sum() and many other operations.

Operations like this are particularly useful if I have a big data set and I do not want to analyze it all: I can just choose a subset.

### 15.6.3 Reading/Writing DataFrame from/to files

- **.to_csv('namefile.csv')** saves my DataFrame to a new CSV file called namefile.csv:

```
df2.to_csv('vet_visits.csv')
```

- **pd.read_csv('namefile.csv')** reads from the CSV file and already arranges the data into a DataFrame:

```
df_vet = pd.read_csv('vet_visits.csv')
print(df_vet)
```

Prints

```
  Unnamed: 0  animal  age  visit day
0      Molly     cat    7  yesterday
1        Ben     cat   10      today
2     Rustie     dog    4      today
3        Bax  rabbit    2      today
4    Vampire     cat    3   tomorrow
5    Tornado     dog    3   tomorrow
```

Note, however, that our original indexes (Molly, Ben, Rustie, Bax, Vampire, Tornado) are now considered part of the values and new default indexes (0,1,2,3,4,5) have been added.

If we are working with really big data we might want to read the input not all together but in multiple chunks, e.g.

```
df = pd.read_csv("saved.csv",chunksize=10000)
```

where after chunksize I indicate the number of rows each chunk should contain

```
>>> for df in pd.read_csv('vet_temp.csv',chunksize=2):
...     print(df)
...
  Unnamed: 0 animal  age  visit day
0      Molly    cat    7  yesterday
1        Ben    cat   10      today
  Unnamed: 0  animal  age visit day
2     Rustie     dog    4     today
3        Bax  rabbit    2     today
  Unnamed: 0 animal  age visit day
4    Vampire    cat    3  tomorrow
5    Tornado    dog    3  tomorrow
```

There are several similar commands to read/write into other file formats, for example

- **.to_excel(namefile)** saves the DataFrame to an excel data sheet

- **pd.read_excel(namefile)** reads an excel data sheet to a DataFrame
- **pd.read_json(namefile)** reads a JSON file (which is essentially a stored Python dictionary)

### 15.6.4 Visualization in pandas

Pandas uses matplotlib. You do not have to import matplotlib because it is already included in pandas. I am not a big fan of this. I suggest using matplotlib out of pandas.

**EXERCISE:**

The file "examples/pandas/Skyserver_SQL2_27_2018 6_51_39 PM.csv" contains the first 10000 entries from the Sloan Digital Sky Survey DR14 (`https://skyserver.sdss.org/dr14/en/help/docs/introduction.aspx`). Using a python notebook, read this to a DataFrame. Select the g and r columns and make a color magnitude diagram g versus g–r. Use hist2d to plot it reasonably (because there are too many points for a scatter plot: it would be too crowded). The result should look like Figure 85. This example is shown in **examples/pandas/SDSS.py** or, if you prefer jupyter-notebook, in **examples/pandas/SDSS.ipynb**.

Sources of for pandas:

- `https://www.learndatasci.com/tutorials/python-pandas-tutorial-complete-introduction-for-beginners/`
- `https://www.youtube.com/watch?v=vmEHCJofslg`
- `https://www.youtube.com/watch?v=PfVxFV1ZPnk`

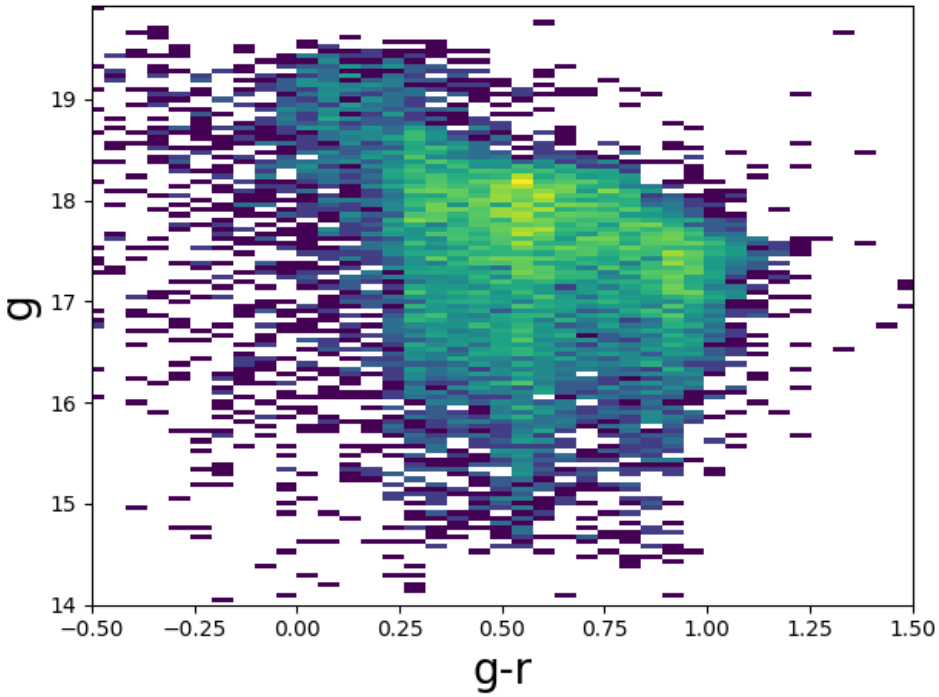


*Figure 85: Color magnitude diagram from the SDSS sample.*

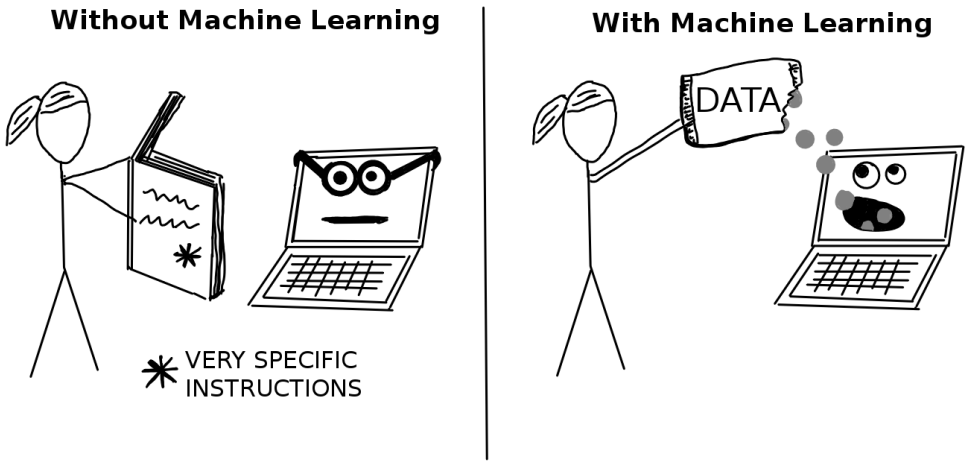


*Figure 86: Some cartoons from xkcd are just immortal* `https://xkcd.com/`.

# 16 NOTIONS OF MACHINE LEARNING IN ASTROPHYSICS

Most of this chapter is based on *Interpretable Machine Learning: A Guide for Making Black Box Models Explainable*, by Christoph Molnar `https://christophm.github.io/interpretable-ml-book/` (Creative Commons License).

## 16.1 *What is Machine learning?*

*Machine Learning is a set of methods that allow computers to learn from data, in order to make predictions.* For example, to predict the value of a house, the computer would learn patterns from past house sales.

Machine learning is a paradigm shift from "normal programming" where all instructions must be explicitly given to the computer to "indirect programming" that takes place through providing data (Fig. 86).

We will focus on **supervised** machine learning, which covers all prediction problems where we have a **training dataset**, i.e. a dataset for which we already know the outcome of interest (e.g. past house prices) and want to learn to predict the outcome for new data.

The goal of supervised learning is to learn a predictive model that maps **features of the data** (e.g. house size, location, floor type, ...) to an output (e.g. house price). If the output is categorical (e.g., dog or cat, yes or not, this flower is a daisy or an iris or a rose), the task is called **classification**, and if it is numerical, it is called **regression** (e.g., the house price is 200k EUR, the black hole mass is $10^6$ $M_\odot$).

The machine learning algorithm learns a model by estimating **parameters** (like weights) or **learning structures** (like trees). The algorithm is guided by a **score or loss function that is minimized**. In the house value example, the machine minimizes the difference between the estimated house price and the predicted price.

The typical steps in machine learning are the following.

- **Step 1:** Data collection. The more, the better. This will be the training sample.
- **Step 2:** Enter this information into a machine learning algorithm that generates a prediction.
- **Step 3:** Use model with new data. Try to predict some properties of the new data based on what you have learned from the training sample.

Pros of this procedure: Machines are faster than humans at repetitive taskes (and often cheaper);
Cons: Insights about the data and the task the machine solves is hidden in increasingly complex models. The procedure might be opaque.

Let us see now some basic definitions necessary to understand the next sections.

- A **Machine Learning Model** is the program that maps inputs to predictions.
- A **Black Box Model** is a system that does not reveal its internal mechanisms. In machine learning, "black box" describes models that cannot be understood by looking at their parameters (e.g. a neural network). The opposite of a black box is sometimes referred to as White Box or **interpretable model**. Interpretable Machine Learning refers to methods and models that make the behavior and predictions of machine learning systems understandable to humans.
- A **Dataset** is a table with the data from which the machine learns. The dataset contains the features and the target to predict. When used to induce a model, the dataset is called training data.
- An **Instance** is a row in the dataset. An instance consists of the feature values $x(i)$ and, if known, the target outcome $y_i$.

- The **Features** are the inputs used for prediction or classification. A feature is a column in the dataset. The matrix with all features is called X. The vector of a single feature for all instances is $x_j$ and the value for the feature $j$ and instance $i$ is $x_j^{(i)}$. In the example of house prices, the features are, e.g., house size, location, floor type.
- The **Target** (or the **Outcome**) is the information the machine learns to predict. In mathematical formulas, the target is usually called $y_i$ for a single instance. In the example of house prices, the target is the price of the house, which is known for the training sample and unknown for the new data.
- A **Machine Learning Task** is the combination of a dataset with features and a target. Depending on the type of the target, the task can be for example classification, regression, survival analysis, clustering, or outlier detection.
- The **Prediction** is what the machine learning model "guesses" what the target value should be based on the given features. The model prediction will be denoted by $\hat{f}(x^{(i)})$ or $y$.

## 16.2 *Interpretability and Machine Learning*

There is no mathematical definition of interpretability. Non-mathematical definitions include *"Interpretability is the degree to which a human can understand the cause of a decision"*[9] and *"Interpretability is the degree to which a human can consistently predict the model's result"*[10].

Machine learning techniques have some degree of interpretability if, for example, they provide a **summary statistics** for each feature (for example the importance of a feature to reach the final prediction). In some cases, the summary statistics can be visualized.

## 16.3 *Decision Tree*

A decision tree is currently one of the most popular algorithms for supervised machine learning, especially because of its interpretability.

Tree based models **split the data multiple times according to certain cut-off values in the features**. Through splitting, different subsets of the dataset are created, with each instance belonging to ONLY one subset. The final subsets are called terminal or **leaf nodes** and the intermediate subsets are called internal nodes or split nodes. To predict the outcome in each leaf node, the **average outcome of the training data** in this node is used. Trees can be used for classification and regression.

There are various algorithms that can grow a tree. They differ in the possible structure of the tree (e.g. number of splits per node), the criteria

[9] https://arxiv.org/abs/1706.07269

[10] https://papers.nips.cc/paper/6300-examples-are-not-enough-learn-to-criticize-criticism-for-interpretabil

how to find the splits, when to stop splitting and how to estimate the simple models within the leaf nodes.

The classification and regression trees (**CART**) algorithm is probably the most popular algorithm for tree induction. We will use CART because there is already a nice set of functions to do decision trees in the **scikit-learn** package of python and CART is used in this package: `https://scikit-learn.org/stable/modules/tree.html#tree-algorithms`

Inside conda, you can easily install scikit-learn by typing, as usual

```
conda install scikit-learn
```

Figure 87 is a simple schematic representation of a decision tree. Instances with a value greater than 3 for feature_x1 end up in the lower-right leaf on. For all other instances, an additional feature (feature_x2) is evaluated. If feature_x2 $\leq 1$ they are assigned to the lower-left leaf, otherwise to the third remaning leaf. The final leaves contain 88, 93 and 179 instances, respectively.

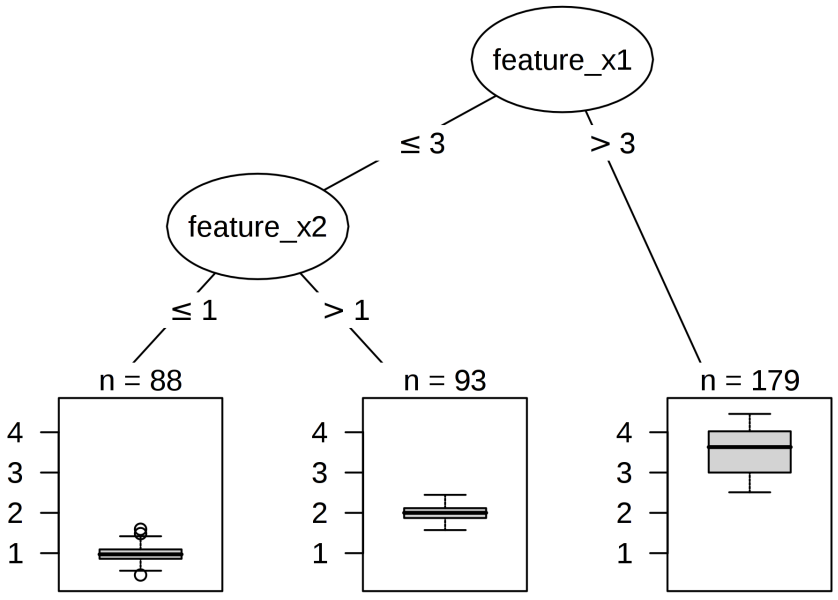


*Figure 87: Example of a simple decision tree (from Christoph Molnar's book).*

The following formula describes the relationship between the outcome $y$ and features $x$.

$$y = f(x) = \sum_{m=1}^{\mathrm{M}} c_m \, \mathrm{I}\{x \in \mathrm{R}_m\} \tag{338}$$

Each instance $x$ falls into exactly one leaf node $\mathrm{R}_m$. $\mathrm{I}\{x \in \mathrm{R}_m\}$ is the identity function that returns 1 if $x$ belongs to the subset $\mathrm{R}_m$ and 0 otherwise. If an instance $x_i$ falls into the leaf node $\mathrm{R}_l$, the predicted outcome is $y_i = c_l$, where $c_l$ is the average of all training instances in leaf node $\mathrm{R}_l$, i.e.

$$c_l = \frac{1}{\mathrm{N}} \sum_{i=1}^{\mathrm{N}} \tilde{c}(x_i), \tag{339}$$

where N is the number of instances in subset $\mathrm{R}_l$ and the $\tilde{c}(x_i)$ is the value of the outcome of instance $x_i$.

But where do the subsets come from? CART takes a feature and determines **which cut-off point minimizes the variance of $y$ for a regression task or the Gini index of the class distribution of $y$ for classification tasks**. The variance tells us how much the $y$ values in a node are spread around their mean value. The Gini index tells us how "impure" a node is, e.g. if all classes have the same frequency, the node is impure, if only one class is present, it is maximally pure.

The Gini index is given by

$$\mathrm{G} = \frac{\sum_{i=1}^{\mathrm{N}} \sum_{j=1}^{\mathrm{N}} |\tilde{c}_i - \tilde{c}_j|}{2\,\mathrm{N} \sum_{i=1}^{\mathrm{N}} \tilde{c}_i} \tag{340}$$

If all the differences $|\tilde{c}_i - \tilde{c}_j|$ are close to zero, then the Gini index is close to zero, meaning that there is no much difference between the value of the target of each instance in the leaf node. In this case, the node is maximally pure.

The algorithm tries to create subsets by trying different groupings of categories. After the best cutoff per feature has been determined, the algorithm selects the feature for splitting that would result in the best partition in terms of the variance or Gini index and adds this split to the tree.

The algorithm continues this search-and-split recursively in both new nodes until a stop criterion is reached. Possible criteria for stop are: A minimum number of instances that have to be in a node before the split, or the minimum number of instances that have to be in a terminal node.

- **Interpretability:** The interpretation is simple: Starting from the root

node, you go to the next nodes and at each bifurcation you know which subsets of data you are looking at. Once you reach the leaf node, the node tells you the predicted outcome. All the bifurcations are connected by 'AND'.

- **Feature importance:** The overall importance of a feature in a decision tree can be computed in the following way: Go through all the splits for which the feature was used and measure how much it has reduced the variance or Gini index compared to the parent node. The sum of all importances is scaled to 100. This means that each importance can be interpreted as share of the overall model importance.

## 16.4 *Example: the iris flowers*

scikit-learn contains several test examples. Let's take the iris sample, which contains the main features of 150 iris flowers. Each flower has 4 features: sepal length in cm, sepal width in cm, petal length in cm, petal width in cm. The target is the variety of the iris, i.e. setosa, versicolor or virginica.

We can apply a decision tree to this sample with the scope of performing a **classification**: classify an iris into one of the three above varieties (setosa, versicolor or virginica), based on the aforementioned four features.

We can do as follows

```
from sklearn.datasets import load_iris
from sklearn import tree
import matplotlib.pyplot as plt
import graphviz


iris = load_iris()
print(iris.data)
print(iris.target)
print(iris.feature_names)
print(iris.target_names)


clf = tree.DecisionTreeClassifier(max_depth=4)
clf = clf.fit(iris.data,iris.target)
tree.plot_tree(clf)

plt.show()

dot_data = tree.export_graphviz(clf, out_file=None,
                      feature_names=iris.feature_names,
                      class_names=iris.target_names,
                      filled=True, rounded=True,
                      special_characters=True)
graph = graphviz.Source(dot_data)
graph.render("iris")
```

This script imports the iris dataset and the CART tree from scikit-learn (which is called sklearn), then calls the tree by assuming that the maximum number of splitting will be four (tree.DecisionTreeClassifier(max_depth=4)) and performs the tree algorithm onto the iris sample (clf.fit). Finally, the function tree.plot_tree plots the result. As an alternative, if we want a more understandable tree (where labels are included), we can use the package graphviz (also available within conda) and obtain the tree that is shown in Figure 88.

But now we have a problem: we have used all our iris dataset as a training set, to allow our computer to learn the classification. How can we test whether the computer has learned in a smart way or not?

For example, we can do a **cross validation** (`https://scikit-learn.org/stable/modules/cross_validation.html#cross-validation`), i.e. we can remove a part of the data, perform the tree classification on the remaining data and then use the part we removed as a test to check the goodness of the algorithm. Since we already know the targets of the withheld part, we can compare the outcomes we get from the tree with the targets that we already

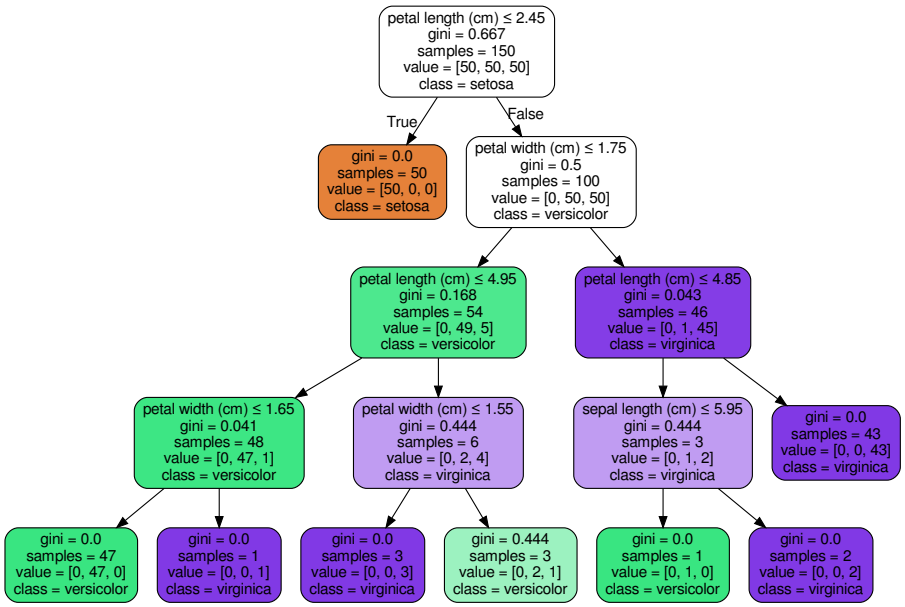


*Figure 88: The iris classification tree.*

know, to see how good our algorithm was. Actually, cross validation repeats this procedure for each subset of data.

In the basic approach, called $k$–fold cross-validation, the training set is split into $k$ smaller sets. The following procedure is followed for each of the $k$ "folds":

- A model is trained using $k - 1$ of the folds as training data;
- the resulting model is validated on the remaining part of the data (i.e., it is used as a test set to compute a performance measure such as accuracy).

The performance measure reported by $k$–fold cross-validation is then the average of the values computed in the loop. This approach can be computationally expensive, but does not waste too much data (as is the case when fixing an arbitrary validation set), which is a major advantage in problems such as inverse inference where the number of samples is very small. See Figure 89 for a schematic representation.

We can introduce cross validation into our iris example as follows.

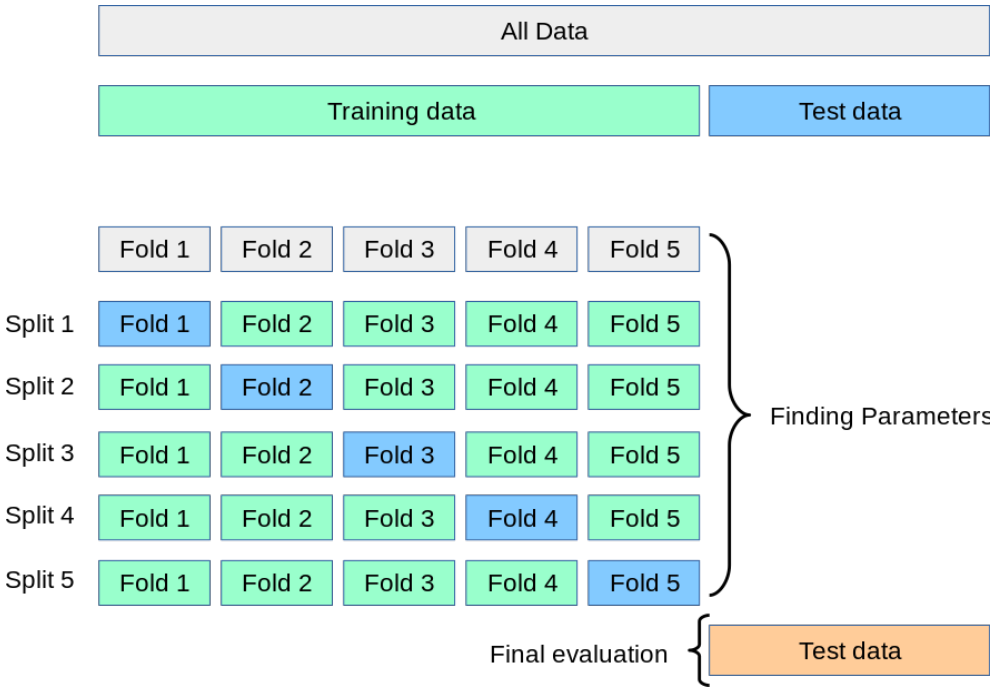


*Figure 89: Schematic representation of the k–fold cross-validation.*

```
from sklearn.datasets import load_iris
from sklearn import tree
from sklearn.model_selection import train_test_split, cross_val_score
import matplotlib.pyplot as plt


X, y = load_iris(return_X_y=True)


X_train, X_test, y_train, y_test = train_test_split(X, y,
                     test_size=0.4, random_state=0)


clf = tree.DecisionTreeClassifier(max_depth=4)
clf = clf.fit(X_train,y_train)
tree.plot_tree(clf)

plt.show()

a=clf.score(X_test, y_test)
print(a)
```

The output is $a = 0.95$ (the higher this number the better).

Now, if you have a sample of iris, you can measure their sepals, petals and apply your tree to a totally new set of data. Enjoy.

# 17 SORTING ALGORITHMS

Sorting refers to arranging the elements of an array (or list) in a descending or ascending order. It might seems useless for astrophysics, but instead it is a must for a fast treatment of data, whenever the data are BIG (e.g. files with millions or billions of rows).

For example, suppose that we have a file with a catalogue of a billion of stars. Each star is uniquely identified by its ID (for example this ID is a uint64). Suppose that we want to search for a specific star in the array, let say the one with ID 16409309430. If the stellar catalogue was SORTED by stellar IDs our search can be significantly sped up, thanks to a smart usage of file reading and "continue" or "break" (see the exercise).

Herebelow, we discuss bubble sort, selection sort, quicksort and merge sort which are four of the most used sorting algorithms. Other popular sorting algorithms include insertion sort and count sort (see `https://www.hackerearth.com/practice/notes/sorting-code-monk/` for more details).

## 17.1 *Bubble Sort*

This algorithm is based on the idea of repeatedly comparing pairs of adjacent elements and then switching their positions if they exist in the wrong order.

See Figure 90 to understand how it works **Complexity:** The complexity of

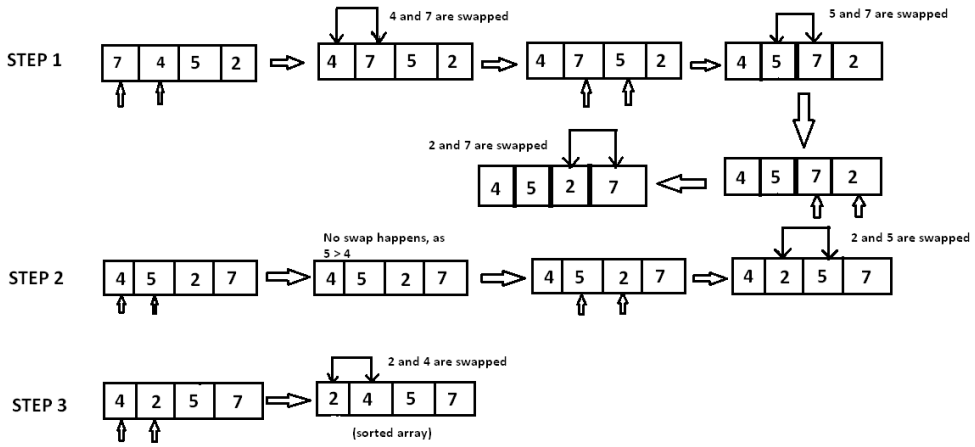


*Figure 90: Visualization of the bubble sort algorithm.*

bubble sort is $\mathcal{O}(N^2)$, because for every element we iterate over the the entire array each time.

## 17.2 *Selection Sort*

This algorithm is based on the idea of finding the minimum or maximum element in the unsorted array and then putting it in its correct position for a sorted array.

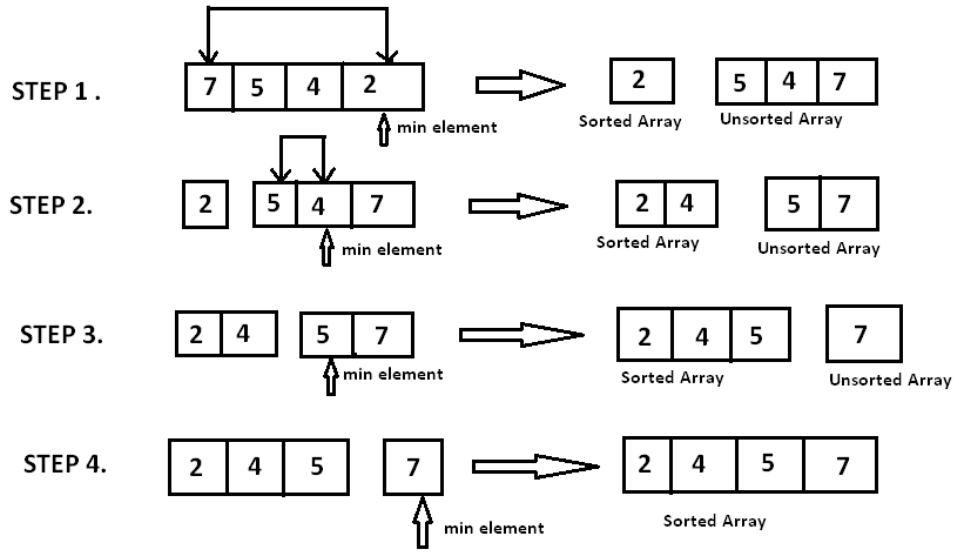


*Figure 91: Visualization of the selection sort algorithm.*

We have our list $a = [7, 5, 4, 2]$ and we need to sort it in ascending order. Let's find the minimum element in the array i.e., 2 and then replace it with the first position's element, i.e., 7. Now we find the second smallest element in the remaining unsorted array and put it at the second position and so on. See figure 91 to see how it works.

**Complexity:** The complexity of bubble sort is $\mathcal{O}(N^2)$, because for every element we iterate almost over the the entire array each time.

## 17.3 *Quicksort*

Quicksort is a **divide and conquer** algorithm. Divide and conquer algorithms consist in recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem (see `https://www.techiedelight.com/divide-and-conquer-interview-questions/` for more details).

Like all divide and conquer algorithms, it first divides a large array into two smaller sub-arrays and then recursively sorts the sub-arrays. It works through 3 steps:

- **Pivot selection:** pick an element, called a pivot, from the array (usually the leftmost or the rightmost element of the partition.

- **Partitioning:** reorder the array so that all elements $\leq$ ($\geq$ ) pivot come before (after) the pivot. After this partitioning the pivot is in its final location.

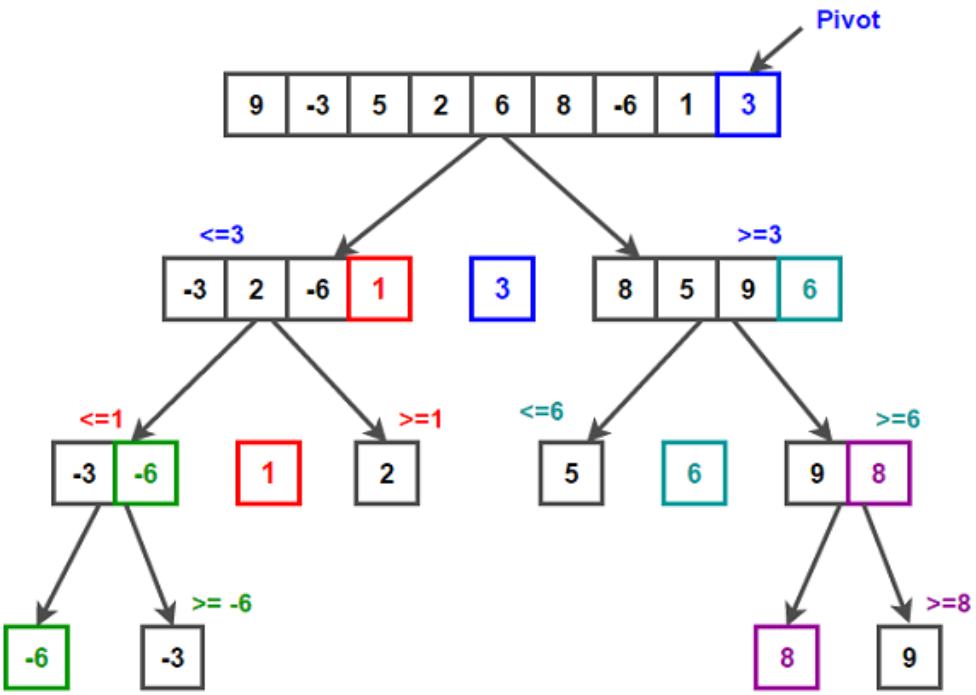


*Figure 92: Visualization of the quick sort algorithm.*

- **Recur:** recursively apply the above steps separately to the sub-array of elements with smaller values and greater values than the pivot.

This procedure is summarized in Figure 92. See `https://www.techiedelight.com/quicksort/` for more details.

**Complexity:** quicksort scales as fast as $\mathcal{O}(N \log N)$ at its best and as slow as $\mathcal{O}(N^2)$ at its worst.

## 17.4 *Merge Sort*

Merge sort is a **divide and conquer** algorithm, too. Like all divide and conquer algorithms, merge sort divides a large array into two smaller sub-arrays and then recursively sorts the subarrays. The procedure consists in two steps:

- divide the unsorted array into n subarrays, each of size 1 (an array of size 1 is considered sorted);

- repeatedly merge subarrays to produce new sorted subarrays until only 1 array is left (the one array is sorted).

This procedure is summarized in Figure 93. See `https://www.techiedelight.com/merge-sort/` for more details.

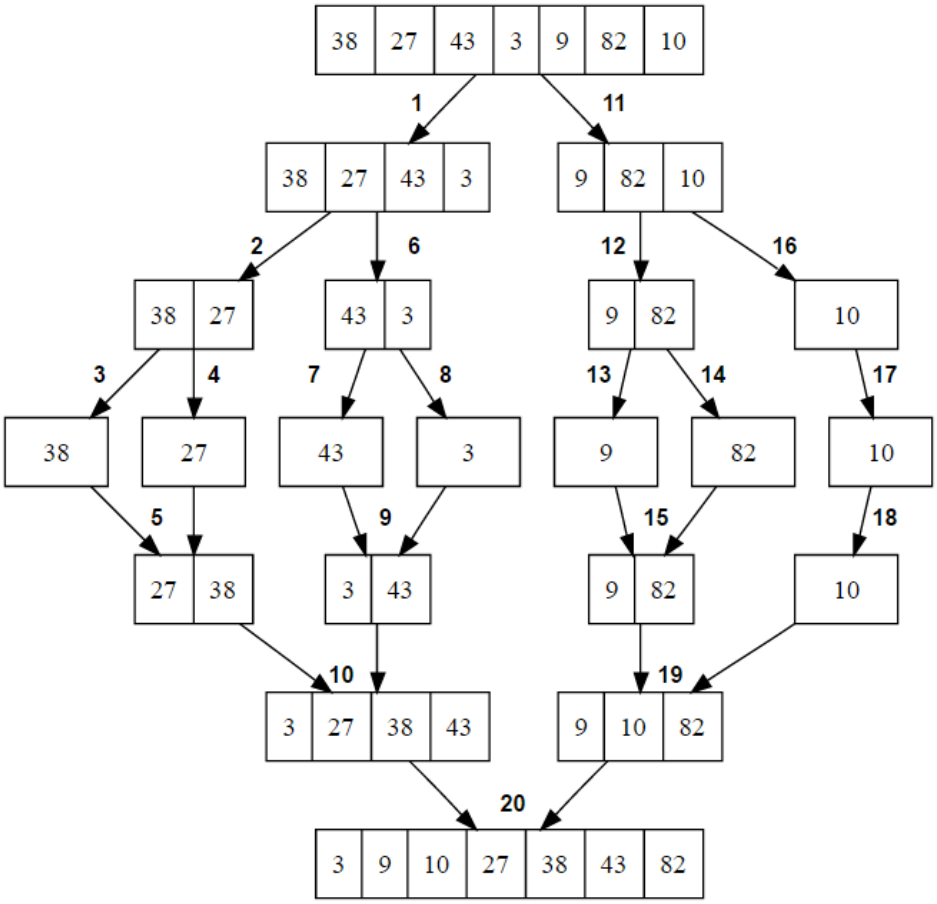


*Figure 93: Visualization of the merge sort sort algorithm.*

**EXERCISE:**

Write your own sorting script (in python) **for at least two** of the aforementioned sorting algorithms. Use them to order this embarassingly simple list a=[9,-3,5,2,6,8,-6,1,3].
Suggestion for the merge sort and quick sort algorithms: the ordering function calls itself.

**EXERCISE:**

The file illustris3_135.dat contains a list of stellar particles from the Illustris cosmological simulation [Vogelsberger et al., 2014]. The first column is the ID of each particle (the ID is a uint64). Use your preferred sorting algorithm to order the IDs.
Now, redo the sorting exercise using the function **sorted** already built-in python. The built-in function is probably much faster than your sorting algorithm (to estimate execution time, run the script with 'time python namefile.py').
Syntax: sorted(array), where array is the array to be sorted. The function sorted uses the timsort algorithm, which is a bit more complex than the simple ones we presented in these lectures (see `https://en.wikipedia.org/wiki/Timsort`).
To give you an idea, the script using sorted should take $\sim$ 15 seconds with a decent quad-core laptop, while a script with a good home-made quicksort algorithm should take $\sim$ 50 seconds on the same laptop (i.e. 3 – 4 times more). If your home-made algorithm is much slower than that you might try to improve it (but optimization is not requested for the exam).

## REFERENCES

J. Binney and S. Tremaine. *Galactic dynamics*. 1987.

J. B. Hartle. *Gravity : an introduction to Einstein's general relativity*. 2003.

G. Hobbs, D. R. Lorimer, A. G. Lyne, and M. Kramer. A statistical study of 233 pulsar proper motions. *Monthly Notices of the Royal Astronomical Society*, 360:974–992, July 2005. doi: 10.1111/j.1365-2966.2005.09087.x.

I. R. King. The structure of star clusters. III. Some simple dynamical models. *Astronomical Journal*, 71:64, Feb 1966. doi: 10.1086/109857.

M. Limongi and A. Chieffi. Presupernova Evolution and Explosive Nucleosynthesis of Rotating Massive Stars in the Metallicity Range -3 $\leq$ [Fe/H] $\leq$ 0. *Astrophysical Journal Supplement*, 237(1):13, July 2018. doi: 10.3847/1538-4365/aacb24.

M. Mapelli, N. Giacobbo, E. Ripamonti, and M. Spera. The cosmic merger rate of stellar black hole binaries from the Illustris simulation. *MNRAS*, 472(2): 2422–2435, Dec 2017. doi: 10.1093/mnras/stx2123.

J. F. Navarro, C. S. Frenk, and S. D. M. White. The Structure of Cold Dark Matter Halos. *Astrophysical Journal*, 462:563, May 1996. doi: 10.1086/177173.

P. C. Peters. Gravitational radiation and the motion of two point masses. *Phys. Rev.*, 136:B1224–B1232, Nov 1964. doi: 10.1103/PhysRev.136.B1224. URL `https://link.aps.org/doi/10.1103/PhysRev.136.B1224`.

H. C. Plummer. On the problem of distribution in globular star clusters. *MNRAS*, 71:460–470, Mar. 1911. doi: 10.1093/mnras/71.5.460.

E. E. Salpeter. The Luminosity Function and Stellar Evolution. *Astrophysical Journal*, 121:161, Jan 1955. doi: 10.1086/145971.

J. Schaye, R. A. Crain, R. G. Bower, M. Furlong, M. Schaller, T. Theuns, C. Dalla Vecchia, C. S. Frenk, I. G. McCarthy, J. C. Helly, A. Jenkins, Y. M. Rosas-Guevara, S. D. M. White, M. Baes, C. M. Booth, P. Camps, J. F. Navarro, Y. Qu, A. Rahmati, T. Sawala, P. A. Thomas, and J. Trayford. The EAGLE project: simulating the evolution and assembly of galaxies and their environments. *MNRAS*, 446(1):521–554, Jan. 2015. doi: 10.1093/mnras/stu2058.

M. Vogelsberger, S. Genel, V. Springel, P. Torrey, D. Sijacki, D. Xu, G. Snyder, D. Nelson, and L. Hernquist. Introducing the Illustris Project: simulating the coevolution of dark and visible matter in the Universe. *MNRAS*, 444(2): 1518–1547, Oct 2014. doi: 10.1093/mnras/stu1536.