Computer Modelling
Course Notes

2023 – 2024

Contents

1. Officiae

1.1. Course coordinates

This course runs across both semesters, and finishes by week 10 of semester 2.

Introductory Lecture: Week 1 in semester 1, 14:30–16:00, in Lecture Theatre B, JCMB.
Workshops: Tuesdays, 14:10 to 17:00 in JCMB 4325D. Weeks 2–11 in semester 1, and weeks 1–10, in semester 2.

Alternate week participation: You should attend workshops every second week. This should be allocated quickly and at random via the timetabling tool. Please get in touch in case of timetabling clashes.

Working Platform: You can work either on the lab computers, or your own laptop. If using your own machine, ensure you can install the Anaconda Python Distribution as soon as possible. This is available for Windows, Linux, and Mac.

Semester 1 submissions are individual, as is the final report. We aim for the project code development in semester 2 to be done in pairs, but in the event of further COVID issues we may have to change this.

Lecturer team: You can reach the lecturers as follows:

- Joe Zuntz (Course Organizer until February). Office: R7 (ROE). Email: joe.zuntz@ed.ac.uk
- Adam Carnall (Lecturer and Course Organizer from February). Office: Stobie 2 (ROE). Email: adamc@roe.ac.uk
- Alex Hall (Lecturer from February). Office: East Tower (ROE). Email: alex.hall@ed.ac.uk

Direct Entry Chemical Physics should write to the Course Organiser ASAP

2. Introduction

2.1. Synopsis

The course is a practical introduction to computational simulation techniques in physics, using the Python3 programming language. The rationale behind computer simulation will be introduced and the relationship between simulation, theory and experiment discussed. The course introduces good software development techniques, the algorithm/code design process and how to analyse/understand the results of simulations.

Assessment is by a series of smaller exercises in semester 1 that lead to a project to write a full simulation code in semester 2.

The final assessment is a report not too dissimilar from an experimental lab report, except data will be obtained from your very own code. All material is available through Learn. All marked assessments should be uploaded through Learn, to be marked by the lecturers and TA's.

You can find full details on the assessments on Learn.

Background. The course assumes students are familiar, and reasonably comfortable, with the Python introduction seen in *Practical Physics* or *Programming and Data Analysis*. However, we are aware those courses are often the first step into programming for many students, and that many students have not programmed at all for the last 8 months. Thus, the course builds up gradually, with simpler tasks in the first weeks. Should you need them, the Python notes from last year's course are available here..

Plagiarism. All submitted work should be your own. Submission links will only be available after you accept an *"own work declaration"*. We understand TA's will help you and, during traditional lab sessions, that you may also receive minor help from your colleagues, or be inspired by their comments.

YOU SHOULD NOT, UNDER ANY CIRCUMSTANCES, COPY CODE. Even if you rename variables, change spacing and comments and scramble methods, it will still be plagiarised code. All final submissions will be checked automatically for plagiarism, and reports will undergo a similarity test via Turnitin. In the tone of Terry Pratchett's DEATH:

NEVER COPY CODE THAT IS NOT YOUR OWN.

2.2. Conventions

Through these notes, and the course slides, we will use code snippets and commands. These will generally belong to three basic types: basic python, interactive python and, finally, terminal commands (the ones on Linux or Mac computers). These types of commands will show slightly different highlighting and symbols for the prompt.

Basic python is what you would type into a file, probably composed of several lines. You are likely to write it via emacs, Vim, Spyder, Atom, or similar. Basic python will be shown here inside a box, with no other prompt symbols:

```
#!/bin/env python3
"""
   This is
   a
   multiline comment
"""
import math
twopi = 2*math.pi
print("Hello, World!")
```

Python also has a builtin interactive console. It comes very handy to test code snippets, or a quick loop. Spyder has, by default, an iPython console in the bottom right of the window that allows you to type interactive python. We will denote interactive python commands with three *"bigger than"* symbols, or three dots if not the first line

```
>>> print("Hello, World!")
Hello, World!
>>> for i in range(3):
...     print(i)
...
0
1
2
```

Use interactive python to explore how scripted python would work. Finally, terminal commands similar to what you would type on a School Linux machine or a Mac will be denoted with a $ prompt:

```
$ whoami
miguel
$ date
Sun 19 Sep 2021 21:18:20 BST
```

2.3. Computational simulation and the scientific method

This course is primarily aimed at introducing the concepts of *computational modelling* and the skills you need in order to write your own models and run them on computers. As the course is taught in Python 3, all of the examples in the notes will be presented in Python; however, the concepts you will learn are equally applicable to programming in any object-oriented language (for example, Java or C++).

You will be familiar with the concepts of theory and experiment as part of the scientific method and how they are employed in physics to help us understand how the physical universe works. Since the advent of digital computers in the 1950's, computational simulation has become more and more important in scientific research. In fact, much scientific research would now be impossible without the support of simulations. This is particularly true in physics where our interpretation of the results from the largest experiments (LHC, distributed telescopes, diffraction methods) depends critically on having supporting simulation data.

It is also true at the other end of the spectrum. As well as supporting experimental research, simulation also allows us to explore the behaviour of our theories in physical regimes that are not available experimentally (due to scale, cost, danger, etc.) and to drive the direction of new experiments that have not yet been planned. Computers can even be used to help us prove mathematical theorems.

However, it is a mistake to see Computer Modelling as merely a tool to process large experimental data, or to get numbers out of a theory. The finest computational physicists are able to bridge both parts, and provide some unique hindsight.

Figure 2.1 illustrates the relationship between simulation, theory and experiment in the scientific method.

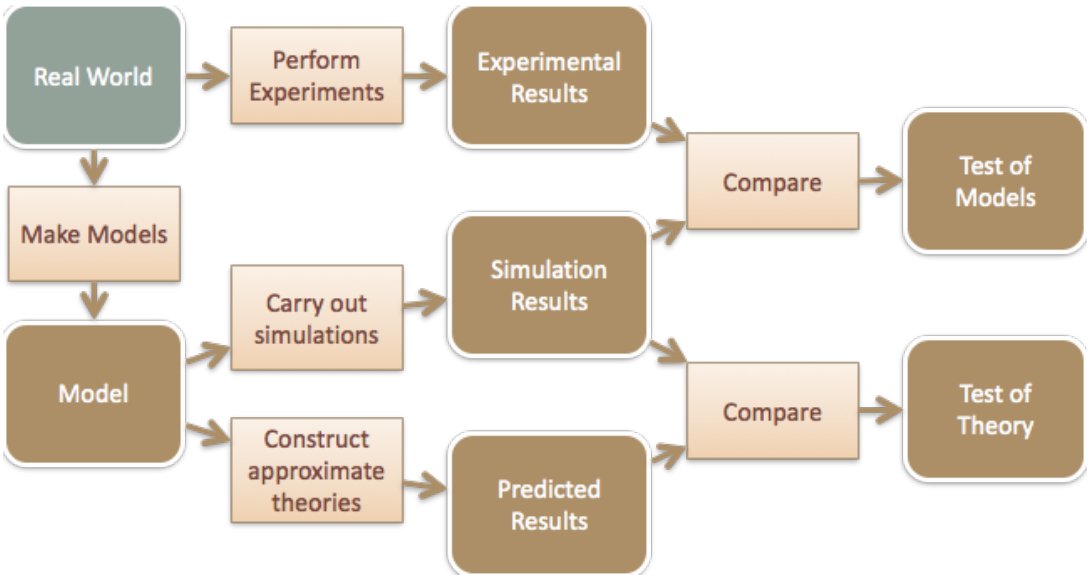

Figure 2.1.: How simulation and modelling fit into the scientific method.

3. Beyond SciProg

3.1. Procedural programming in Python

Every modern programming language has a set of similar features. All come with a set of data types: ways that certain data will be represented inside a computer. These representations would distinguish integer numbers from float numbers, or from strings of characters. Data types come with primitive operations: basic ways to manipulate data that are directly available to the user. These can range from very simple (e.g., shifting bits in registers, as done in Assembly) to rather complex (e.g., matrix multiplication, as done in Matlab). They also have mechanisms to combine these primitives to form complex expressions: through combinations of primitives in long arithmetic or logical expression, or definitions of external functions.

Much of the basic functionality in Python has been covered in the Scientific Programming section of the Programming and Data Analysis course in year 2. We recap here some of the essentials, but you should consult the course notes of PDA to re-familiarise yourself with details.

3.1.1. Data types

Every variable in Python has a type. Those types are set *implicitly* by the Python interpreter, i.e. they don't have to be specified when a variable is initialized. Variable types can also change upon re-assignment of a variable. These aspects differ from many other high-level programming languages and make Python code faster to write, but also more error-prone – be careful to ensure your variables have the correct type.

Native simple data types in Python include

- Numbers. Python distinguishes integers (data types `int` and `long`), floats (`float`) and complex floats (`complex`).
- Strings (`str`).
- Booleans (`bool`), or logical expressions.

The implicit assumption of data types extends to how Python interprets operations on variables. Observe in the following example how the "+"-sign is used for addition in numbers and in strings:

```
>>> a=3
>>> b=4
>>> print(a+b)
```

```
7
>>> a="3"
>>> b="4"
>>> print(a+b)
34
```

The simple data types in Python can be combined in different *composite* data structures. Python supports composites called lists, tuples, and dictionaries.

Lists

Lists are the most versatile Python data type. A *list* consists of a series of variables. These can be of any simple (or composite) Python data type, e.g. you can create lists of lists of integers. List elements can be selectively added and deleted and the values of list entries can be changed.

Lists are initialised using square brackets, with elements separated by comma. Elements are addressed by their position inside the list, which is an integer index between 0 and $N-1$, where N is the number of list entries. Here is an example list declaration, with subsequent output and element-wise manipulation:

```
>>> my_list = [0, 1, 3.14, "pie", a, b]
>>> print(my_list[2])
3.14
>>> my_list[0] = my_list[1] + my_list[2]
>>> print(my_list[0])
4.140000000000001
```

Tuples

Tuples are similar to lists, but not mutable: once created, entries can not be changed. You can think of them as "read-only lists". Tuples are initialised with round brackets, and their elements are also addressed by index:

```
>>> my_tuple = (1.0, "42", "pie", 5)
>>> print(my_tuple[2])
pie
>>> my_tuple[0] = 2.0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

An attempt at changing an entry in a tuple will result in an error message, as shown above.

Dictionaries

Dictionaries are unsorted collections of key/value pairs. Elements in a dictionary are addressed by their key, which can be used to extract or manipulate the corresponding value. Dictionary entries do not have a numerical order, so they can not be addressed by a sorting index. Keys are usually strings, but this is not necessary. Dictionaries are initialised with curly brackets:

```
>>> my_dict = {'name':'George', 'zip':'SW1A 1AA', 'pi':3.14}
>>> print(my_dict['name'])
George
>>> my_dict['pi']=3.1415
>>> print(my_dict['pi'])
3.1415
>>> print(my_dict.keys())
dict_keys(['pi', 'zip', 'name'])
```

Assignment operators

The most common assignment operator, `=`, just assigns an object in memory to a variable. However, some assignments are very common. Python simplifies the life of programmers by enabling convenient assignment operators such as `+=` or `-=`. Some examples include:

Operator	Example	Equivalent to?
`+=`	`a+=1`	`a = a+1`
`-=`	`a-=1`	`a = a-1`
`*=`	`a*=2`	`a = a*2`
`/=`	`a/=5`	`a = a/5`

We will showcase the simplest, and perhaps most common case, +=1:

```
>>> a = 0.5
>>> a += 1
>>> a
1.5
```

Convenience assignment operators also work with `numpy` arrays, which will be introduced in Section 3.2.

3.1.2. Data manipulation

In the previous subsection, and in the preceding code snippets throughout, we have already used different ways to manipulate data. Note that the same operators can have different effects depending on which data types they are applied to. As seen in section3.1.1, the "+"-sign applies to numbers in the usual way, as arithmetic operation, while it is used for concatenation if applied to string variables. If applied to Booleans, the "+"-operator

will act as numerical addition, after the two Booleans have been converted to integers: 1 to represent `True`, and 0 to represent `False`. This *operator overloading* is a feature of Python designed to make writing code simple: only relatively few operators need to be known, and they are applied in the most sensible way to different data structures. However, the onus is on the programmer to ensure that they are used correctly.

3.1.3. Control flow

Any programming language will provide functionality to break from purely sequential execution of code. This is absolutely necessary to progress beyond the most simple programs. Python is no different, and has language constructs to allow conditional sections, branching, and looping of code.

Python's `if-else` statements allow for selective execution of code: certain parts of the program are only executed if certain conditions are met. A simple example is

```
# Print whether or not a variable has the value 5
if n==5:
    print("n is 5.")
else:
    print("n is not 5.")
```

An `if`-statement begins with the protected word `if`, followed by a conditional statement, and a colon. The *conditional statement* must return a Boolean value, `True` or `False`; it can be as complicated as necessary. If the statement is `True`, the subsequent code snippet will be executed; this section is marked by an *indentation* of the code by four spaces – a particular convention of Python. Indentations are cumulative, for instance if inside an `if`-statement another `if`-statement appears. If the conditional statement is `False`, the Python interpreter will look for a `else:` statement following the `if`-branch. If such a statement exists, the subsequent indented code section will be executed. Otherwise, code execution will continue with the next statement.

Note that `if` and `else` sections are executed at most once.

A simple extension of `if`-statements that allows multiple execution is the `while`-loop:

```
# Print an increasing number of squares of integers
n=0
while n<10:
    print("n^2 is " + str(n**2))
    n = n+1
```

A `while`-loop begins with the protected word `while`, followed by a conditional statement and a colon. The subsequent indented code section will be executed as long as the conditional statement is true; that is, after every execution the conditional statement will be revisited. That means, to avoid an infinite number of executions, that the variables included in the conditional statement should be modified inside the code segment. In the example above, the counter variable `n` is increased, and after 10 iterations the conditional

statement will revert to `False`. Code execution then resumes at the next statement after the loop. A `while`-loop is thus executed anywhere between zero and infinite number of times.

An alternative to `while`-loops are `for`-loops. Instead of testing a conditional statement, a `for`-loop is executed for a certain range of inputs. These inputs can be elements of a list, keys of a dictionary, or (most common) integer counts to a certain maximum value. Simple examples are:

```
# Print squares of entries of an explicit list:
for n in [2, 3, 5, 8, 13]:
    print(n**2)
# Print inverse entries of a list variable:
my_list = [1.0, 2.0, 4.0]
for n in my_list:
    print(1/n)
# Print keys and values of a dictionary:
for keys in my_dict.keys():
    print(key, my_dict[key])
# Print squares of integers:
for n in range(10):
    print(n**2)
```

Every `for`-loop begins with the protected word `for`, followed by a counting variable, the word `in`, a sequence, and a colon. The sequence can be given explicitly (as a list), or implicitly (as a list variable, through the `range()` function, keys of a dictionary, etc.). The following indented code section is executed once for every entry in the sequence, where the counting variable assumes the value of said entry. Thus, a `for`-loop can be executed any number of times, including zero.

The `range()` function

You may be familiar with `range()` as used above, with one single argument denoting the end of the range. The function is, however, more flexible, and may take up to three arguments, as documented here. These allow you to begin the range in non-zero values, and to have a step different than 1. This is best showcased with two examples:

```
>>> list(range(5)) # End
[0, 1, 2, 3, 4]
>>> list(range(1,5)) # Start, End
[1, 2, 3, 4]
>>> list(range(1,5,3)) # Start, End, Step
[1, 4]
```

Advanced Iteration

A previous code box threw a curve ball that might have gone unnoticed:

```
my_list = [1.0, 2.0, 4.0]
for n in my_list:
    print(1/n)
```

These lines deserve further explanation. But first, I encourage you to run them in an interactive python session. Indeed, you get 1, 0.5, and 0.25. What is going on here?

Here, python detects that `my_list` is an iterable object (something that has an index to access objects within), and then will execute the next lines in the loop by going through every item within `my_list`. The beauty of this is that it will work no matter what. Numbers in a list, characters in a string, objects, anything. And, as it turns out, this is a faster way to deal with iterables in python than going by the index!

Sometimes, however, you need at the same time both the index of the item, as well as quick access to the item itself. The needed functionality is provided by `enumerate()`. This function will trawl through an iterable and, at every step, it returns the index as well as the item. Let us see this in action:

```
>>> import random
>>> mylist = [random.random() for i in range(3)] # look elsewhere...
>>> for index, item in enumerate(mylist):
...     print(f"The {index}-th item is {item}")
...
The 0-th item is 0.44126217504717713
The 1-th item is 0.007542479388990775
The 2-th item is 0.9052225380307908
```

Where the particular string syntax is explained in the next section.

This can be further exploited using `zip()`. A `zip` is an object that combines two or more lists, ignoring additional items if the lengths don't match, into a list of tuples of sorts for easy iteration. This is best shown in an example that prints the elements of two lists in sync:

```
>>> list1 = [1, 2, 3]
>>> list2 = ['a','b','c', 8, 9, 'lalala'] # longer than list1!
>>> for item1, item2 in zip(list1, list2):
...     print(item1, item2)
...
1 a
2 b
3 c
```

Note that `list2` has more items, but that `zip()` knows where to stop.

3.1.4. Input and output

Python provides a variety of interfaces to allow user interaction with a program. These are essential to allow us to execute the same code, but for instance with different initial conditions. Most commonly, input and output happens through the Terminal where the code is run, or through files stored on the computer hard disk in the same directory as the Python code.

Terminal interaction

Output to the Terminal is achieved using the `print()` command, as already used in the previous subsections. User input from the Terminal can be processed using the `input()` command.An example code section would be

```
# Ask for a user's name and print it back out
user_name = input("What is your name? ")
print("Hello, " + user_name + "!")
```

The above is a somewhat cumbersome way to print the value of a variable. F-strings, available since Python 3.6, offer a neat alternative. An f-string is created by prepending `f` to the string, an allows variables and operations by surrounding them in curly braces. We used an f-string in the `enumerate` example in the previous section. Here's another example:

```
>>> pi = 3.14
>>> num = 125
>>> print(f"N = {num}, Pi = {pi}")
N = 125, Pi = 3.14
```

Sometimes, however, strings must be formatted neater, taking a predefined amount of characters, or with certain precision. This can be achieved, for any string containing variables, via the `.format()` method. This may be useful for crafting easy to read output or in cases where a format is strictly specified. Let us show first an example, using `pi` and `num` from the previous case:

```
>>> print("N = {0:4d}, Pi = {1:10.6f}".format(num,pi))
N =  125, Pi =   3.140000
```

Here we have a string, in between quotation marks, with two variables denoted by the curly braces.

- `{0:4d}` will be the 0–th argument inside format, it will take 4 characters, and `d` means it will be a right-aligned integer.
- `{1:10.6f}` will be the 1–th argument, and it will be `f`, a floating point number, taking 10 characters of which 6 will be taken after the period.

While we are using `print()`, we could be writing to a file just as well, or even just crafting a string for further usage.

In general, each set of curly braces references an argument inside `.format()` with a number, and, optionally, is followed by a colon and a format descriptor or identifier. The variable type should be identified correctly to avoid errors. Arguments within `.format()` may be valid expressions. Another example:

```
>>> print("atan(1) = {0:f}".format(pi/4))
atan(1) = 0.785398
>>> print("atan(1) = {0:10.6f}".format(pi/4))
atan(1) =   0.785398
>>> print("atan(1) = {0:10.8f}".format(pi/4))
atan(1) = 0.78539816
>>> print("atan(1) = {0:10s}".format(pi/4))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Unknown format code 's' for object of type 'float'
```

A few common format identifiers are:

- `d` for decimal integers;
- `f` for decimal floating point;
- `e` for exponential floating point (scientific notation); and
- `s` for strings.

There are more identifiers, and fancy possibilities, such as aligning on the left or on the right. These are better described in the official documentation.

Run-time command line options

In some situations, explicit user interaction is not desired, but code should still be run with unique input parameters; for example, a Python code should be run for a given set of input parameters. Then, command line arguments can be passed to the Python program at the moment of execution. These are then available inside the program as a specific list, `sys.argv[]`. Here is an example of a Python code snippet:

```
# Python module newton.py
import sys
print(sys.argv[])
```

This particular Python code might be called from the command line as follows:

```
user@cplab001 $ python3 newton.py 9.81 apple tree
['newton.py', '9.81', 'apple', 'tree']
user@cplab001 $
```

Inside the Python program `newton.py`, the list `sys.argv[]` is well-defined and can be manipulated or printed like any other list. There are several important aspects of `sys.argv`:

1. `argv` will take the command line written to run the program after `python3`, and split it in between the spaces
2. The number of spaces in between arguments is irrelevant
3. The first entry, `sys.argv[0]`, is the name of the main Python program being executed. In the case above, `newton.py`
4. All items are strings, be them `'apple'` or `'9,81'`. If an item is meant to be a different kind, such as a float, or a boolean, it must be recast, e.g.

   ```
   g0 = float(sys.argv[1])
   ```

The idea behind using arguments is to set e.g. input or output file names, or some important parameters for your program, in a way that it is much faster and efficient than having to type these factors every time.

How can it be much faster, if you still have to type the information? Well, the command line typically keeps track of the command history, so accessing previously used lines is as easy as pressing the "up" arrow. Linux and Mac terminals even allow searching in the history.

If you are using an IDE such as Spyder, you can also benefit from these arguments, as you can set them per-file. In the Spyder window, this is done by activating the relevant python file. Then, on the top menu bar, click on "Run", then "Configuration per file", and then activating the "command line options" box. The necessary arguments would then be written into the text input field. For this particular case, that would be
`9.81 apple tree`

Consider exploiting `sys.argv[]` when passing this type of information to your code. It truly is nicer than writing the same arguments over and over again.

File I/O

Almost every meaningful computer program will interact with files on the hard disk, whether to read input information, or write results. The opening, reading/writing, and closing of files are thus essential. In Python, this is done in a very straightforward and easy way. Consider the following code section:

```
# Open input file for reading
input_file = open("input.dat", "r")
# Read in the first line, then close the file
```

```
line = input_file.readline()
input_file.close()
# Print the first word of the input file
items = line.split()
print(items[0])
```

The command `open()` is used to open files (specified by name as string) for reading (with option `"r"`) or writing (`"w"`). `open()` returns a so-called *file handle*, named `input_file` in the example above. The file handle is Python's internal gateway to access the information written in the file on the hard disk. One particular function that can be applied to file handles is `readline()`, which reads a complete line from the file, and moves the reading cursor to the next line; repeated application of `readline()` can therefore be used to read in the entire file. `readline()` returns a string, which can then be chopped into individual words using `split()`, and these can then be processed further.

Writing data to file is arguably even easier. A simple file output process would be

```
# Open input file for writing
output_file = open("output.dat", "w")
# Write variable value to file
output_file.write("Value: {0:12.8f}".format(num))
output_file.close()
```

The `write()` command, applied to a file handle variable, will write to the file, following the formatting specifications given afterwards. `write()` can work with the same options as the `print()` command.

Note that input and output files need to be closed after all reading or writing operations have been completed. Use the `close()` command on each file handle to do this; if you don't, you could leave the status of files in a limbo after the Python program has finished its execution, and compromise data on the hard disk.

Forgetting to close open files is a typical beginner error. During the course, if your code uses `matplotlib` for graphs, data loss will be averted only because Python is waiting for user input. However, once these `matplotlib` calls are removed, output files suddenly become incomplete. Avoid these surprises by proactively closing open files.

I/O in one (protected) go

Often times, especially when reading the input, or when writing the final results of a calculation, we open and close the files in sequential order. Python offers a syntax that is robust, helps with potential I/O issues, and makes sure that whatever file we open, it is closed by the end of the loop. This functionality is accessed via the `with...as` protected keywords.

For example, if we wanted to read and split the first input line as in the example above, we would do:

```
with open("input.dat","r"), as f:
    line = f.readline()
    items = line.split()
    print(items[0])
```

There are several bits to highlight here:

1. the `open()` syntax (read, write, append...) in the `with` line is the same as before.
2. We are naming the file handle `f`, but normal variable naming is allowed.
3. All operations on the file handle (again, `f`) happen within the `with` block, and the file handle will disappear once the block is over.

Again, the main advantage of this approach is that, under non-catastrophic circumstances, python will make sure input.dat is properly closed. Even if the code crashes. Even if the file is open for writing data. Finally, think how you would use this syntax to write the values in two lists of floats, such as `time` and `energy`, to "outfile.dat".

3.1.5. Error–checking

Sooner or later, sadly, something will go wrong. Many times, especially during input, it will be our fault: speed is too large, Earth's mass is negative, position is a series of Booleans, or maybe the input is just poorly formatted. Whenever this happens, if the code was not written with error-checking in mind, we may encounter one of the following issues:

1. The code crashes straight away (bad)
2. The code crashes later, while doing something unexpected (worse)
3. The code runs, and runs, and runs, and much later on produces garbage (worst)

In the worst case, garbage results will look okay, and require serious knowledge to realise something might have gone wrong. In the case of user input, the best way to avoid such a situation is to mistrust the user (likely your future self in this course) and proactively check input for potential pitfalls.

Pythons offers powerful syntax to try certain commands, and then produce meaningful messages without crashing. These messages can even vary depending on the type of error encountered. This functionality is accessed via the `try...except` syntax:

```
try:
    [code block 0]
except Exception1:
    [code block 1]
except (Exception2, Exception3):
    [code block 3]
```

```
else:
    [code block 4]
finally:
    [code block 5]
```

Here, Python would first try to run the code under the `try` block. If it failed, rather than crashing out, Python would execute one of the following blocks depending on the *exception* (or type of error) raised. The `else` block is run if `try` finds an exception not accounted to before. The `finally` block is executed in any case. You can find more information on the Python documentation Let us see how this works in practice.

Let us consider the case of a user-defined mass. It must be a positive float:

```
user_mass = input("Please input particle mass: ")
try:
    my_mass = float(user_mass)
except ValueError:
    print("Mass should be a floating point number. Exiting.")
    sys.exit()
if my_mass <= 0:
    print("Mass should be positive. Exiting.")
    sys.exit()
```

Note that the sign check could have been done within the `try` block. It is good practice to keep these minimal, though. Here we are explicitly stating to act upon the (`ValueError`) exception. It is not necessary to include it, though (try it!). The types of exception to catch are available in the documentation, although you can explore these in an interactive python session:

```
>>> int('test')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'test'
>>> f = open("nonexistent_file","r")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory:
'nonexistent_file'
```

Where `ValueError` is the exception raised if python fails to convert a string into an integer (or a float!), and `FileNotFoundError` is the exception raised if the file you are trying to read was not found. Exercise 0 has another `try...except` example in section 2.2.1.

You are encouraged to check the Python documentation on handling exceptions

3.1.6. Functions, multiple returns, and default argument values

There are times where the same set of operations is performed over the same objects over and over again, such as computing $|\mathbf{r}_j - \mathbf{r}_i|$ in an electrostatics calculation. When that happens, we can simplify the code by creating a *function* (or method). Functions are callable objects that act on other objects[1] in a predetermined way. For example:

```
def distance(ri, rj):
    r_ij = []
    # We forego error-checking, and assume 3D
    for n in range(3):
        r_ij.append( rj[n]-ri[n])
    mod_sq = r_ij[0]**2 + r_ij[1]**2 + r_ij[2]**2
    return math.sqrt(mod_sq)
```

So now, every time we want to check the distance between two 3D vectors, represented as lists, all we need to do is call `distance()`. And if we wanted improvements or bugs, we only need to focus on these lines.

Sometimes, as part of the calculation inside a function, we produce more than one result of interest. For example, the code above computes $\mathbf{r}_{ij}$ in our way to get r_{ij}. Python allows multiple returns in a single function, via `return a, b, c, ...`. Such syntax will return a tuple in which each element corresponds to the returned object.

Consider instead the following modification of `distance`:

```
def distance2(ri, rj):
    #    [These lines as before]
    return r_ij, math.sqrt(mod_sq)
```

We would obtain the difference vector and its modulus the following way:

```
>>> a = [0, 1, 0]
>>> b = [1, 1, 1]
>>> vector, modulus = distance2(a, b)
>>> vector
[1, 0, 1]
>>> modulus
1.4142135623730951
```

Finally, we will cover default argument values. Many programming languages offer so-called *"method overload"*, that is, we can define identically named methods as long as the number of arguments is different. Python does not. Only one name per method or function is allowed. However, we can still define default values:

[1] In Python, pretty much anything goes, including other functions

```
def ideal_volume(press, mol=1, temp=273.15):
    R0 = 22.41396954
    return R0*mol*temp/(press*273.15)
```

This would return the volume in l of an ideal gas for a given pressure in atm, assuming the default is 1 mol and $T = 273.15$ K. We could take advantage of this several ways:

```
>>> ideal_volume(1) # Use defaults
22.41396954
>>> ideal_volume(1,1,500) # Use all values
41.02868303130149
>>> ideal_volume(1,temp=500) # Naming one option allows omitting others
41.02868303130149
>>> ideal_volume(1,temp=500, mol=0.1) # Re-ordering named options
4.102868303130149
>>> ideal_volume(1,mol=0.1)
2.2413969540000003
```

There are two things we cannot change: first, all non-optional values must come first in the function definition. And second, calls to such methods must also keep the non-optional arguments first, even though we can reorder the default variables provided we name them. Let us showcase this as would appear in interactive sessions:

```
>>> def wrong_args(mol=1, press, temp=273.15):
...     R0 = 22.41396954E-3
...     return R0*mol*temp/(press*273.15)
...
  File "<stdin>", line 1
SyntaxError: non-default argument follows default argument
```

```
>>> ideal_volume(mol=0.1,2)
  File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
```

ADVANCED. This section must conclude with a warning for advanced users. *You must NEVER EVER set default values directly to function calls.* This is allowed, but python will evaluate that function once during the first syntax-checking run, at runtime, and remain as is. The proper way to assign a function call to a default variable is to, instead, set those variables to `None`, and then use a logical check for evaluation:

```
def my_function(var1, var2, var3=None):
    if var3 is None:
        var3 = call_function()
    ...
```

3.1.7. Constants

One feature that Python lacks is that of constants, that is, defining a *"variable"* to a value such that it cannot be changed. However, as physicists, we are extremely likely to use constants in our codes such as the speed of light, Böhr's radius and so on. The pythonic way to define these is to declare variables named in ALL_CAPS at the top of the files where these are needed, right after importing modules:

```
import math
import numpy as np

# CODATA 2010 Constants
G0 = 6.67384E-11              # m^3/kg/s^2
C0 = 299792458.0              # m/s
BOHR = 0.52917721092E-10      # m
```

You should never write the values of constants explicitly, and within the code. This makes it easier to introduce hard-to-find bugs. Furthermore, it is important to avoid having same constants taking different values throughout the code. This would also allow you to easily change the units of your program should you wish to do so. Similarly, constants should not be user-defined parameters without a very good reason for doing so.

When specifying constants the way above, cutting the number of significant figures is false economy; for the computer it is the same effort to multiply by 3×10^8 and the exact value of c, and you may be wasting valuable precision. NIST maintains a database of constants, updated every 4 years or so, with the best values known to date, named CODATA. It includes exact values where available (e.g. the speed of light in vacuum). The latest version, CODATA 2018, was published in May 2019 and is available in https://physics.nist.gov/cuu/Constants/index.html

3.2. Numpy

While lists are flexible, changeable in size, searchable and iterable, not to mention native to python, lists are a loose fit when it comes to scientific and numerical data usage. For example, a 2×2 matrix could be defined somewhat cumbersomely as a list of lists:

```
>>> sigma_x = [[0,1],[1,0]]
>>> sigma_x
[[0, 1], [1, 0]]
>>> sigma_x[0][0]
0
```

Other programming languages, such as Fortran, C or Java, have types better suited for such concepts, traditionally named *"arrays"*. Some even have element-wise operations to make programmers' lives easier. While python does not have native array support, this can be accessed via `numpy`, a third-party (e.g. not officially developed by the main people behind python) module. Nowadays, `numpy` forms, together with `scipy` and `matplotlib` the core of scientific python. Even if it is not an official part of python, `numpy` is often part of typical python distributions. Learning the basics of `numpy` is one of the expected outcomes of this course.

3.2.1. Loading numpy and creating arrays

In order to access `numpy` it must first be imported, like `math`, `sys`, or `random`. It is typical to call it `np`, and that is what we suggest for this course:

```
>>> import numpy as np
```

From now on, **all examples in this section assume numpy has been loaded this way**. Once numpy is loaded, we can now create arrays calling `numpy.array`, or `np.array`, given our alias. Items can be accessed via indices, just like in a list. Mirroring the example above:

```
>>> sigma_x = np.array([[0,1][1,0]])
>>> sigma_x
array([[0, 1],
       [1, 0]])
>>> sigma_x[0,0]
0
```

The i-th, j-th element of an array `a` can be accessed as `a[i,j]` or `a[i][j]`. *The former version is preferred.* Indices can also be variables, and this can be used to easily iterate over elements. We can also operate with items the regular way:

```
>>> for i in range(2):
...     for j in range(2):
...         sigma_x[i,j]*(-2)
```

The `type` python call shows what sort of object `sigma_x` is, and a new call, `.shape` returns a tuple with the size of each index of the array:

```
>>> type(sigma_x)
<class 'numpy.ndarray'>
>>> type(sigma_x[0,0])
<class 'numpy.int64'>
>>> sigma_x.shape
(2, 2)
```

This brings the first restriction of `numpy`: unlike a list, all elements in a `numpy` array must belong to the same type. Indeed, had we created the original array with just a single float, all other integers would have been cast as floats.

```
>>> new_sigma = np.array([[0,1],[1.0,0]])
>>> new_sigma[0,0]
0.0
>>> type(new_sigma[0,0])
<class 'numpy.float64'>
```

Conversely, if we try to assign a float to an element of an integer array, as all elements must be integer, `numpy` will try to cast it into an integer. For example:

```
>>> sigma_x = np.array([[0,1],[1,0]])
>>> sigma_x[1,0] = 1.2345
>>> sigma_x
array([[0, 1],
       [1, 0]])
```

Thus, dealing with arrays requires a bit of extra care. We can, of course, initialise arrays to the required type, by specifying it at the end of the call:

```
>>> my_array = np.array([['10','11',False],[0,1,2.5]],int)
>>> my_array
array([[10, 11,  0],
       [ 0,  1,  2]])
>>> array2 = np.array([[-3.14,0],[float('inf'),np.pi]],float)
>>> array2
array([[-3.14      ,  0.        ],
       [        inf,  3.14159265]])
```

In the first example above, note how the string `'10'` is converted succesfully to an integer 10, and how the Boolean `False` is converted to integer 0. The second example is showcasing how to set a floating point variable to ∞, and how `np.pi()` is the value you expect. In general, you should define the type of the array whenever there may be

potential ambiguity. Finally, should you need it it is possible to use other types, including non-native objects. This will not be covered here.

Other than explicit assignation of values, `numpy` provides calls to easily initialise arrays to a set of values, be it 0's, 1's, random numbers, or identity matrices. The size and shape of these arrays is determined by call arguments. All these are float arrays by default, unless specified otherwise, as shown below:

```
>>> np.zeros(3)
array([0., 0., 0.])
>>> np.ones([2,2],int)
array([[1, 1],
       [1, 1]])
>>> np.random.random([2,3])
array([[0.48446651, 0.73237817, 0.08738375],
       [0.63295284, 0.52610693, 0.41627254]])
>>> np.eye(3)
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

Equispaced arrays can be created using `np.arange([start=0], stop, [step=1])` and `np.linspace(start,stop,[num=50])`, with brackets denoting optional arguments.

3.2.2. Maths with arrays

Now we are already familiar with how to create arrays and how to access particular elements, so we can start benefitting from `numpy`. The most immediate advantage is gaining mathematical operations between arrays. `numpy` uses *operator overloading* (See section 3.3.4) to enable the use of the arithmetic operators (`+`, `-`, `*`, `/`...) to enable element-wise operations between arrays:

```
>>> a = np.array([3,0,1],float)
>>> b = np.array([np.pi,np.e,2**2],float)
>>> 2*a
array([6., 0., 2.])
>>> a + b
array([ 6.14159265,  2.71828183,  5.        ])
>>> a - b
array([-0.14159265, -2.71828183, -3.        ])
>>> a * b
array([ 9.42477796,  0.        ,  4.        ])
>>> a / b
array([ 0.95492966,  0.        ,  0.25      ])
>>> np.eye(2) + np.eye(2)
array([[2., 0.],
       [0., 2.]])
```

This is a *new* feature not available with lists, as the defined arithmetic operators on lists have little to do with the above. Try adding two lists together, as a counterexample.

Since the above shows `*` is the element-wise product, this raises the following question: *if my arrays represent matrices, how can I multiply them the usual way?*. For this we need to use a `numpy` method, `dot` or, since Python 3.5, the `@` operator. We will show this using Pauli matrices σ_x and σ_z:

```
>>> sigma_x = np.array([[0,1],[1,0]])
>>> sigma_z = np.array([[1,0],[0,-1]])
>>> sigma_x*sigma_z  # element-wise product
array([[0, 0],
       [0, 0]])
>>> sigma_x @ sigma_z  # Matrix product, w. operator
array([[ 0, -1],
       [ 1,  0]])
>>> np.dot(sigma_x, sigma_z)  # Idem, w. numpy call
array([[ 0, -1],
       [ 1,  0]])
```

On top of matrix multiplication, `numpy` also offers a series of vector operations: `np.inner` (dot product) and `np.cross` (cross product). Be aware that both `inner` and `dot` return the same value for 1D arrays, but **NOT** for 2D arrays (which could represent, say, the position of N particles in 3D space). Therefore, as always, you should trial the implementation when non-obvious cases emerge.

```
>>> v1 = np.array([1,0,0],float)
>>> v2 = np.array([0,1,0],float)
>>> v1*v2
array([0., 0., 0.])
>>> np.dot(v1,v2)
0.0
>>> np.inner(v1,v2)
0.0
>>> np.cross(v1,v2)
array([0., 0., 1.])
```

Basic linear algebra routines can be accessed via the `np.linalg` module (documentation). These even access system-wide optimised libraries for performance reasons. Should you need further routines, `scipy` (designed to complement `numpy` by the same developers) has additional linear algebra routines, and more (such as Fast Fourier Transforms).

Finally, all the arithmetic operations described above require arrays to have commensurate sizes. If the array dimensions mismatch, the operation attempted becomes meaningless, and `numpy` will complain accordingly:

```
>>> np.eye(2) + np.eye(3)
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with
shapes (2,2) (3,3)
```

Universal Functions

Universal functions (ufunc in short) are another immediately useful feature of `numpy`. Ufuncs are traditional mathematical functions, following the conventional naming scheme, that have been extended to act element-wise on arrays (with some extra hidden features that will not be covered here). With ufuncs, computing the exponential of all elements of an array becomes:

```
>>> a = np.array([0,1,2],float)
>>> np.exp(a)  # We call a numpy function
array([ 1.        ,  2.71828183,  7.3890561 ])
```

There are currently over 60 ufuncs available, listed at the official documentation. Basic ones you are likely to find useful this year include: abs, exp, log, log10, sin, cos, tan, arcsin, arccos, arctan, arctan2. Two important reminders about these:

1. `log` is the natural logarithm, base e
2. trigonometric functions *always* use radians

Special mention should go to `arctan2`. In computing, `atan2` is a function accepting two arguments, y and x, and which returns $\arctan(y/x)$ with knowledge of the quadrant, in the range $(-\pi, \pi)$

You should resort to `numpy` operations and ufuncs when possible, if only because these are checked and maintained by several people. You should *only* avoid the numpy way whenever your task explicitly instructs you not to.

3.2.3. Reshaping, Slicing

Before we continue, there is another important limitation of `numpy` to be aware of: size. One of the goals of `numpy` is high performance, which is why, under its nice python wrapping, many calls have C underneath. One of the consequences, as we just saw, is that all elements are the same type. Another one is that elements must be ordered and contiguous in memory. Consequently, and again unlike lists, arrays cannot change in size (easily). We can, however, *"reshape"* them, or extract sections of an array (*"slicing"*). Let us illustrate this, by first reshaping a 1D array into a 2D, and then a 3D array:

```
>>> a = np.arange(12)
>>> b = a.reshape(3,4)
>>> b
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
```

```
       [ 8,  9, 10, 11]])
>>> c = b.reshape(3,2,2)
>>> c
array([[[ 0,  1],
        [ 2,  3]],

       [[ 4,  5],
        [ 6,  7]],

       [[ 8,  9],
        [10, 11]]])
```

Here, `a.reshape(3,2,2)` would have produced the same numerical result. The process followed is to create a (3,2,2) array, and then run the indices in both source and target arrays, *from right to left*, filling the new data. This ordering of data in Python follows that of C, and is called *"Row-major"* order [2].

The next step to take is extracting a row or a column from `b`. This is the simplest form of slicing. We can do this by specifying a combination of the desired index and a colon. A single colon denotes a full range of indices. Continuing with the arrays from above:

```
>>> b
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> b[1]
array([4, 5, 6, 7])
>>> b[1,:]
array([4, 5, 6, 7])
>>> b[:,1]
array([1, 5, 9])
```

The first two slices denote the 1-th row, while the final slice returns the 1-th column. Note how both `b[1]` and `b[1,:]` return identical 1D arrays: python will add additional indices to the right as needed. Note also how all slices, row and column, return flat 1D arrays. If, for whatever reason, we needed a proper row, we could reshape the result:

```
>>> b[:,1].reshape(3,1)
array([[1],
       [5],
       [9]])
```

As you may remember from Practical Physics, colons in lists can be used to denote a range: `a:b` denotes from a until, but not including, b. Therefore, `my_list[4:7]` would

[2] Other programming languages, most notably FORTRAN, follow Column-major order. More info in Wikipedia

return a sub-list containing the elements 4, 5 *and* 6 of `my_list`. We can exploit this to get more complex slices out of arrays:

```
>>> b
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> b[1:3, 0:2 ]
array([[4, 5],
       [8, 9]])
```

Here, the syntax of the slice can be interpreted as taking all the values of b, conserving the shape, from the 1-th up to, but not including, the 3-rd row, and from the 0-th up to, but not including, the 2-nd column.

Martin McBride has written a very visual guide to slicing, including 3D arrays, worth visiting if only for the colourful diagrams. It is available here.

3.2.4. Broadcasting

We have seen until now how `numpy` extends typical mathematical operations to arrays and how it implements matrix operations. We have also seen `numpy` complains whenever dimensionality issues are found. Hidden in an example above is a product of an array by a scalar, which is interpreted the expected way (e.g. if $\mathbf{B} = \lambda\mathbf{A}$, then $b_{ij} = \lambda a_{ij}$). This is different from element-wise multiplication, as the scalar is a single element. One can then wonder: are there other circumstances in which `numpy` will perform arithmetic operations, despite an apparent mismatch of dimensions or shape? This is where *"broadcasting"* comes.

Broadcasting is the set of rules that `numpy` follows while evaluating arithmetic expressions between arrays of different shapes. When evaluating compatibility between arrays, `numpy` checks the trailing dimensions and then moves forward. Two dimensions are compatible when:

- they are equal, or
- one of them is 1.

If dimensions are equal, operations follow as usual. If one of the dimensions is 1, then that array will be replicated along that index until dimensions are equal. This can be illustrated by *"adding"* a vector and a scalar, or a vector and a matrix (despite this being **nonsensical** outside of a computer!!!):

```
>>> 2 + np.ones(3)
array([3., 3., 3.])
>>> np.ones(3) + 0.5*np.eye(3,3)
array([[1.5, 1. , 1. ],
       [1. , 1.5, 1. ],
       [1. , 1. , 1.5]])
```

Essentially, what is happening in the first line is

$$2 \longrightarrow \begin{pmatrix} 2 \\ 2 \\ 2 \end{pmatrix}$$
$$\begin{pmatrix} 2 \\ 2 \\ 2 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 3 \\ 3 \\ 3 \end{pmatrix},$$

as well as the following in the second:

$$\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \longrightarrow \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$
$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} + \begin{pmatrix} 0.5 & & \\ & 0.5 & \\ & & 0.5 \end{pmatrix} = \begin{pmatrix} 1.5 & 1 & 1 \\ 1 & 1.5 & 1 \\ 1 & 1 & 1.5 \end{pmatrix}.$$

On the other hand, if `numpy` is indeed unable to perform the instruction even with broadcasting, the code will crash with an error we have seen before:
`ValueError: operands could not be broadcast together with shapes`.

All in all, broadcasting is a very useful feature, but requires understanding what the underlying code does to avoid nasty surprises. The `numpy` documentation contains further examples that you would do well to check.

WARNING: If your code depends on broadcasting, and it does so in an unelegant way, or it looks likely to be a fluke, your marking stile will be downgraded accordingly.

3.2.5. Copying: a precautionary tale

As has been said many times before, a `numpy` array is a set of ordered values of the same kind lying contiguously in memory. This has some peculiar implications when it comes to copying variables. Let us play with some integers first:

```
>>> a = 2
>>> b = a
>>> a = 3
```

So here we first assign 2 to a, then copy that to b, and then reassign 3 to a. Can you guess the value of b? We haven't modified it, so it should be 2, right? Right?

```
>>> b
2
```

Right! Let's see what happens now with a simple `numpy` array:

```
>>> a = np.arange(3, dtype=float)
>>> b = a
>>> a[0] = np.pi
>>> b
???
```

So here, we create an array of floats, `[0., 1., 2.]`. We assign that object in memory to `a`, and then to `b`. But then we change `a[0]` to π. Surely, just like above, `b` is unchanged, right? Right?

```
>>> b
array([3.14159265, 1.        , 2.        ])
```

Wrong! The difference in behaviour we are seeing here happens because the `numpy` array is *mutable*, while the `int`'s before are *immutable*. You may remember that tuples are immutable. In python, `float`, `int`, `str` and `bool` are also immutable, and reassignment will follow the same approach as for the integers.

For mutable objects, such as arrays or lists, python is copying the memory address to b. Due to performance reasons, we are modifying the contents of that memory address when we change `a[0]` and so, we also indirectly modify `b`. If we require a new variable copying the contents of an old one, it is best to copy the object:

```
>>> a = np.arange(3, dtype=float)
>>> b = a.copy()
>>> a[0] = np.pi
>>> b
array([0., 1., 2.])
```

3.2.6. Per-axis operations

Many `numpy` functions, by default, act on all elements of an array. For example, `np.sum` will sum all elements of an array into a single value. However, there are times when we want the sum to be performed along a certain axis. For example, consider a (5,3) array, where the i-th row represents the linear momentum of particle i:

```
>>> all_p = np.random.random([5,3]) - 0.5
>>> all_p
array([[-0.10140604,  0.43189527,  0.18478626],
       [-0.01098867, -0.18221693,  0.04060135],
       [ 0.39563603,  0.16145504, -0.13275971],
       [-0.19709927, -0.31850395,  0.3465035 ],
       [-0.10437787, -0.33789963, -0.00953503]])
```

where the subtraction of 0.5 takes advantage of broadcasting, and is there to have an average closer to zero. The total momentum of these particles would be the sum along each column. Rather than doing this manually, we're best doing this using the `axis` option, remembering we want to sum along the i-th index, or axis 0:

```
>>> my_p = np.sum(total_p, axis=0)
>>> my_p
array([-0.01823582, -0.24527019,  0.42959638])
```

In this course, you may benefit from using this option in the call to compute the norm of a vector, `np.linalg.norm`. For example, if we wanted to compute the norm along the j-th index:

```
>>> np.linalg.norm(all_p,axis=1)
array([0.48058576, 0.18700863, 0.44746033, 0.51025246, 0.35378216])
```

where the i-th element of that array would represent the modulus of the linear momentum of particle i.

3.2.7. Performance

As we have seen, `numpy` features a new object that us useful for scientific programming, it extends basic arithmetics to arrays and on top of that implements useful functions, not the least linear algebra. But there is more! Python is not particularly fast, as its goal is to enable fast coding. One would typically seek performance later, with more specialised code covering performance-critical areas. This may be even written in C or C++. As it turns out, because many bits of `numpy` are written in such languages, and because `numpy` and `scipy` hook to performance libraries such as OpenBLAS or FFTW, `numpy` can offer large performance gains. Let us showcase this with a simple example:

```
>>> from math import sin
>>> import random
>>> import numpy as np
>>> my_list = [random.uniform(-np.pi/2,np.pi/2) for i in range(100)]
>>> my_array = np.array(my_list)
>>> def sin_list( some_list ):
...     return [sin(item) for item in some_list]
...
>>> %timeit sin_list(my_list)
6.68 µs ± 14.4 ns per loop [...]
>>> %timeit np.sin
1.05 µs ± 10.7 ns per loop [...]
```

Here, `%timeit` is a special iPython keyword to perform simple benchmarking. Note how, despite the list call using list comprehension, and despite importing `math.sin` directly

to avoid extra time searching through `math`, `numpy` is 6 times faster. The advantage grows with array size. If you have ipython3 installed, you could try this example with random lists of 1000 elements. Conversely, the smaller the array, the smaller the numpy advantage. Eventually, the overhead of calling `numpy` dominates. The above example is faster with plain python for lists or arrays of 10 elements.

In general, the more we traverse through indices, the bigger penalty we pay. If available, calling pure numpy functions will save a substantial amount of time. For example, a trivial matrix multiplication algorithm would have *three* `for` loops. The simple `numpy` call `A@B` can be as much as 100 times faster than it for 10×10 matrices.

The take home message from this section should be:

- If you need to manipulate non-small[3] datasets, consider going full `numpy`.
- If there is a way to write your code using pure `numpy`, do so.

But remember, write clear code at first! *"Early optimization is the root of all evil".*

[3]something you wouldn't do on pen and paper

3.3. Object-oriented Programming

Object-oriented programming, or OOP in short, is a method of programming that uses a new kind of variable, named *"object"*, to provide the functionality required. Objects are software building blocks that have well-defined properties and ways of working together to complete the function of the program. Objects can model:

- tangible things — a car, a building, a computer file;
- conceptual things — a date, a vector, a particle;
- processes — sorting a deck of cards, simulating a liquid.

An object is a collection of *properties*, things that define a particular instance of an object, and *methods* (or functions), which define the capabilities of the object.

For example, a given circle in $\mathbb{R}^2$ may have the properties: (x, y) coordinates of the centre, and radius; as well as methods such as: compute the circumference, compute the area or compute the shortest distance to a given point. It would be easy to store all properties of a circle in a dictionary. However, OOP offers blocks with the properties, and easy ways to solve related questions to one, or even more circles (e.g. do these two circles oveerlap?). This makes OOP powerful when building complex programs.

The above properties and methods are all meaningful for an arbitrary circle. In programming, the set of properties and methods of an object is referred to as an *instance* of the object.

What follows is a basic introduction to OOP in Python3. You may also find it useful to consult the Computer Simulation notes that Prof. Judy Hardy kindly made available.

3.3.1. Designing the object

When writing OOP code you should first sit down and think about what properties and methods are needed. This is known as designing the object.

For example, if we are designing an object to represent a complex number z we could imagine that we would need:

- Properties:
  - the real component
  - the imaginary component
- Methods:
  - Get the polar coordinates of z
  - Compute the modulus of z
  - Sum/subtract two complex numbers
  - Multiply/divide two complex numbers

Note how the last two methods above are inherently different: they involve more than one complex number. This may require a slightly different treatment, as we will see. Of course, the list above is by no means exhaustive. For example, you might want to compute the complex conjugate, or more complicated expressions, such as the complex logarithm.

An example `Complex` class is provided as part of exercise 2. Note that this class is somewhat superfluous, as Python can inherently handle complex numbers, but it should help you understand what classes are and how they are constructed. The `Complex` class is conceptually not very different from any of the classes that are used under the hood by Python itself.

3.3.2. Public and Private

One of the core concepts of OOP is that an object should only expose a well-defined public *interface* to any outside objects. All of the inner workings of the object are private and can be changed without affecting the public interface. This allows objects to be extremely reusable as any program can use an object and be confident that the public interface will not change even if the internal functioning of the object is changed or improved. Many OOP programming languages showcase this by making object properties private by default: these cannot be accessed directly from outwith the object. Instead, one defines public methods that either change or retrieve properties.

Python does not do this: object properties are by default public, and can be edited directly. This makes for easier access and less overhead in writing methods for a new class, but might lead to issues if object properties are changed without proper care.

For the reference, the usual approach in Python is to flag variables and functions for internal use with single or double pre underscores (e.g. `_var1`, `__var2`). This has wider implications that will not be covered here, so you should not use this unless familiar with the consequences.

3.3.3. Methods

Methods are functions that belong to an object. Methods in objects can take one of three forms: *instance methods*, *static methods* and *class methods*. In this course we will only cover the first two. The default in Python is that methods are instance methods.

Instance methods

Generally, an instance method operates on the particular instance of the object it is called on. To continue our complex number object example, the method to compute the modulus would be an instance method, because it returns the modulus of the complex number z being represented by the object instance it is called on.

Most objects also have special instance methods called *constructors* that are used when you create new instances of an object. Constructors are discussed in more detail below.

When designing your class, as a general rule of a thumb, if the method called is only related to a single instance, it should be a static method. If other instances are involved, but it can be clearly justified that the method is mainly related to one particular instance, then it could also be an instance method.

Syntax and usage will be covered later on but, briefly, the way to invoke instance methods is via `my_instance.method()`. Instance methods are the default; nothing needs to be added to your code.

Static methods

Static methods operate on objects of the class, but are not called on a particular instance of an object. They are often used to encapsulate functionality that is associated with an object but not dependent on a particular instance of the object, or for functionality that is associated with more than one instance of an object where defining as an instance methods would result in awkward syntax.

For example, all the methods in the `math` class in Python are static methods as you do not create an instance of the `math` class to be able to use them (in fact, `math` is a fully-static class).

Another example: if we want a method to sum two complex number objects we could create this as a static method of the complex number object as it is not dependent on a single instance of a complex number object. We could also create it as an instance method but the syntax for calling it would then look a little strange. Think about these two examples once we have covered the Python syntax for instance and static methods below.

Syntax and usage will be covered later on but, briefly, the way to invoke static methods is via `my_class.method()`. Class methods are marked by the *"decorator"* `@staticmethod` in the line right before the method definition.

3.3.4. Objects in Python

In Python, the implementation of an object is called a *class*. Usually, only one class is defined in a source code file, but this is not necessary.

However, all the source code for a class must be defined in a single code block.

For example, to create a class to represent a complex number called `Complex`[4]we could create a file called `complex.py`[5] containing:

```
class Complex(object):
    # All property and method definitions must be in this block.
```

The bracketed term "`(object)`" means that the class `Complex` does not build upon any other Python class, instead it builds on the generic `object` class. If you would like to construct classes that depend on (and extend) other classes, this entry should reflect that; you can then re-use properties and methods from the dependent class.

[4]Classes should be named following the CapWord convention

[5]Modules, that is, python source files with classes, functions and constants, should be named in lowercase, perhaps with underscores

Properties

There are two kinds of properties: *class* and *instance* properties. Class properties are defined within the `class` block, but outside any other method. It would be typical to define those early in the class. Class properties are identical for all instances of the class, and may be accessed as `ClassName.variable`.

Conversely, instance properties are only linked to one particular instance of the class. Instance properties are also usually defined towards the beginning of the class. Because each instance is likely to be defined by its properties, these will be initialised in the *"constructor method"*. Constructor methods are special instance methods that create new instances of a class, hopefully with well-defined variables. Being so special within a class, constructor methods often follow particular syntax.

In Python, we must define constructors as `__init__()`,[6] followed by standard Python variable definition statements. Properties can be primitive types or other objects. The `__init__()` method is a special instance method; it will be called whenever a new instance of the class is *instantiated*

For example, our `Complex` class will have two real number properties that hold the real and imaginary parts of a given instance.

```
class Complex(object):

    # Initialise an instance
    def __init__(self, a, b):
        self.real = a
        self.imag = b
```

Every instance method must use as first argument the instance of the object itself. It is convention to call this argument `self`. The `__init__()` method above then takes two proper arguments, `a` and `b`, and assigns them to properties of the object:`real` and `imag`. This, implicitly, defines the properties of each instance of the `Complex` class. The choice is not unique: another option would have `__init__` take no arguments, and initialises the new complex number to $(0, 0)$.

Creating new instances

We would now be ready to use the Complex class in our code. To do this, we would add `from <filename> import <Class>` wherever `Class` is used. In our case, after carefully considering the capitalization, that would be:

```
from complex import Complex
```

Initialising a new instance does not require explicitly calling `__init__`. Python understands this method (or `self`!) is special, and we should omit `__init__` when creating a

[6] Note the double pre- and post- underscores of `__init__`. These are important, and mark `__init__` as the first *"magic method"* we see in the course.

new instance. To use the `__init__()` method and create a new instance of our class, and to access and modify its real and imaginary parts, we would write:

```
>>> a = Complex(1.0, 2.0) # Initialise a new complex number
>>> a.real
1.0
>>> a.imag = 5.1
>>> a.imag
5.1
```

Instance methods

We should augment our class with useful methods that act on instances of the class. Below, an excerpt of the class is shown that implements a complex conjugate and a modulus method:

```
class Complex(object):
    # Defines a class for complex numbers

    # Initialise an instance
    # ...

    # Complex conjugate
    def conj(self):
        return Complex(self.real, -self.imag)

    # Modulus
    def norm(self):
        norm_sq = self.real**2 + self.imag**2
        return math.sqrt(norm_sq)
```

Note both instance methods return a value when called upon. The type of that return variable is (as common in Python) not specified. But we can see that the `conj()` method uses the `__init__()` method to initialise a new `Complex` instance, and that the `norm()` method returns a float, provided `real` and `imag` were floats. Instance methods are called by giving the *instance* name, followed by a period `.`, and then the method name. For example:

```
# Create a new complex number
a = Complex(1.0, 1.0)
# Get the modulus
mod = a.norm()
```

One neat thing of OOP is the ability to concatenate calls. Since `conj` returns a `Complex` instance, it will have all properties and methods a regular instance has. For example:

```
>>> a = Complex(1.0, 2.0)  # New instance
>>> a.conj().norm()        # Same as |a|
2.3259406699226015
>>> a.imag + a.conj().imag # Identically zero
0.0
```

A particular instance method is the `__str__()` method[7], which instructs Python how to print an instance of the class, or how to convert it to a string. For the `Complex` class, a sensible option could be:

```
class Complex(object):
    # Defines a class for complex numbers

    # Initialise an instance
    # ...

    # Convert to printable string
    def __str__(self):
        if self.imag>= 0.0:
            return "{0:f} + {1:f} i".format(self.real, self.imag)
        else:
            return "{0:f} - {1:f} i".format(self.real,-self.imag)
```

If a `__str__()` method is defined in a class, its instances can be printed using the standard `print()` command:

```
>>> c = Complex(1.0, -3.0)
>>> print(c)
1.000000 - 3.000000 i
```

Static Methods

Unlike the previous methods, there are times where a given method acts on more than one instance, and it is not obvious which of these is the main instance. For example, if we do $a+b$, it is not immediately obvious which one is the main instance. This functionality is covered by *"static methods"*.

Unlike instance methods, static methods do not need to include `self` as the first argument in the method definition. Instead, static methods should include the `@staticmethod` *"decorator"*. Let us demonstrate this with a method that sums two `Complex` numbers.

```
class Complex(object):
```

[7] Another magic method

```
    # ...Init method...

    # ...Instance methods...

    # Static methods
    @staticmethod
    def sum(a, b):
        sum_real = a.real + b.real
        sum_imag = a.imag + b.imag
        # Return a new complex number
        return Complex(sum_real, sum_imag)
```

Static methods are called by giving the *class name* and then the method name (separated by a period "`.`"). For example:

```
# Create new complex numbers
a = Complex(1.0, 1.0)
b = Complex(1.0, 2.0)
# Get the sum
c = Complex.sum(a, b)
```

You now recognise that the methods of the `math` class, e.g. `math.sqrt()`, or the properties of the `sys` class, such as `sys.argv[]`, follow the exact same "`.`" notation as introduced here.

Magic Methods

By now you might have realised a pattern: Python treats method names that have double pre- and post- underscores differently. This happens with a set of keywords, and exposes special functionality to classes other than native objects. This is in part what allows `numpy` to act so seamlessly. There are way more magic methods than we have time or interest, but you can get a taste of that by typing:

```
>>> dir(int)
```

This will print all methods for the `int` class. Magic methods are the ones with double pre- and post- underscores.

There are a few magic methods that might have caught your eye, methods such as `__add__` or `__mult__`. These do as suggested, and tell python how to use the arithmetic operators (`+`,`-`,`*`,`/`...) for user-defined classes. This technique is called *"operator overloading"*. Bear in mind magic methods are instance methods.

The following example shows how to use the magic method `__add__` to overload `+` and get seamless, sensible addition of complex numbers:

```
class Complex(object):

   # ...Init method...

   # ...Instance methods...

   # Operator overloading
   def __add__(a, b):
        new_real = a.real + b.real
        new_imag = a.imag + b.imag
        return Complex(new_real, new_imag)
```

Operator overloading requires using protected keywords (`__add__()` is associated with the "+" sign), and once defined can be used as follows:

```
# Create new complex numbers
a = Complex(1.0, 1.0)
b = Complex(1.0, 2.0)
# Get the sum
c = a + b
```

Should you be interested in implementing magic methods, here's an equivalence table:

Method	Operator
`__add__`	+
`__sub__`	-
`__mul__`	*
`__truediv__`	/
`__mod__`	%
`__pow__`	**
`__floordiv__`	//

4. Modelling

4.1. The simulation process

When constructing a simulation we want to model the behaviour of the real world and there are a number of steps involved (see Figure 4.1):

1. We create a mathematical model of the real world processes we wish to simulate. For example, in astrophysics we may model the effect of gravity using Newtonian mechanics – or, if necessary, using Einstein's general relativity. The mathematical model is thus based on a physical theory that must exist prior to the simulation process.

2. We convert our mathematical model into a numerical algorithm. This usually involves a discretization of the physical laws used in the previous step (for instance in the time or spatial dimensions), and also involves thought about the simulation program design and layout.

3. A program that implements the algorithm is then written which can perform the simulation based on some set of input data. Here we rely on a particular programming language, and make choices about data structures, organisation of input/output of data, etc.

4. The simulation is run with various inputs, and analyses of the output inform future simulations or edits to the program.

In this section we will outline the common numerical models that are found in scientific simulation codes and also mention parallel programming techniques which are key to the exploitation of modern supercomputing resources.

4.1.1. Analytical and numerical solutions

From theoretical physics you will be used to seeing theories and models written down in mathematical terms. An exact solution of the mathematical model is known as an *analytical* solution.

The vast majority of real problems which we wish to model are too complex for an analytical solution.

For example, two bodies interacting through Newtonian gravity is a model that can be solved analytically so we can obtain an exact solution to the model (within the hardware precision limitations of the computer). Note that this is not an exact solution of the physical problem but an exact solution to our *model* of the physical world.

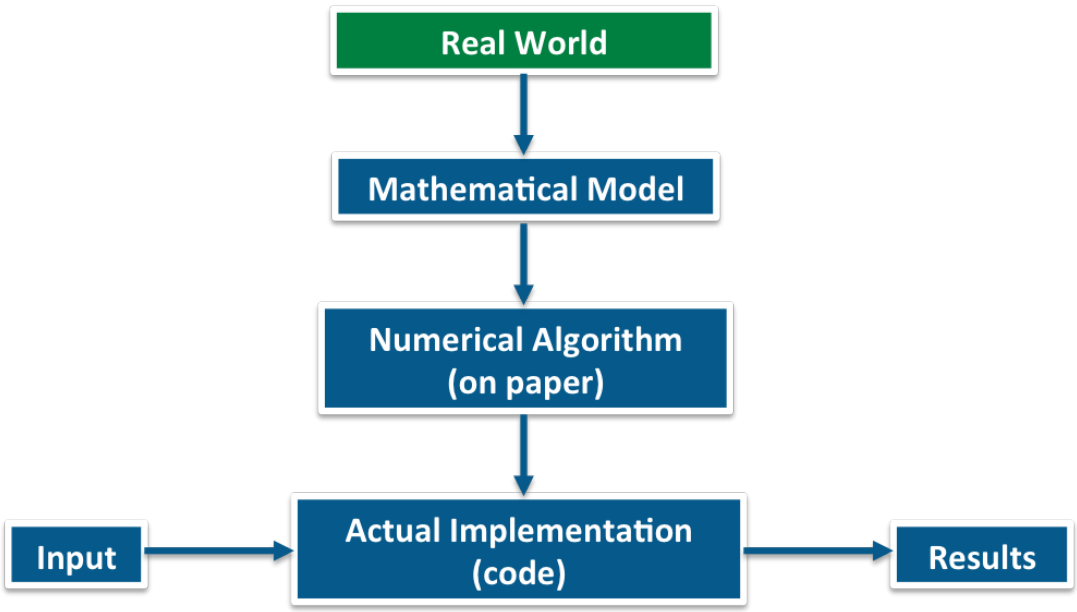

Figure 4.1.: The simulation process.

If we introduce a third body into our model then we are generally not able to use mathematical techniques to find an analytical solution – instead we must rely on *numerical techniques* which approximate a solution.

Examples of numerical techniques include numerical integration, eigensolver routines and discrete Fourier transforms.

4.1.2. Common numerical models in scientific simulations

The vast majority of time used for simulation is spent in *floating-point operations* (FLOPs) and data access. Floating-point numbers are discussed in more detail in section 4.4.2 but FLOPs essentially correspond to manipulations of real numbers.

There are a small number of numerical techniques used in scientific simulation that consume the majority of time (either because they are FLOP-heavy and/or place a large load on memory access). They are

1. Dense linear algebra
2. Sparse linear algebra
3. Spectral methods
4. N-body methods
5. Structured grids
6. Unstructured grids
7. Monte-carlo methods

These have been called the *Seven Dwarves of High Performance Computing* (where a dwarf is an algorithmic pattern, see below for a discussion of algorithms).

Many of the operations in these algorithmic classes are used so commonly that they have been encoded into standard *numerical libraries* that are optimised for each computer platform they are installed on. The benefit for the programmer is that they only have to learn how to *interface* to the routines in those libraries, and need not worry about the internal workings of the specific numerical routines. Updates to the library routines (to enhance functionality or increase speed) can be performed on the compute platform without requiring every programmer to change their code as well.

Dense linear algebra

Vector-vector, vector-matrix and matrix-matrix operations including singlular value decomposition, Eigenproblems and systems of linear equations. Data is almost always organised as contiguous arrays within memory.

Most modern processors perform very well on this type of numerical method, as floating-point performance is usually measured using the LINPACK benchmark – which is essentially a dense linear algebra algorithm. So, to get high performance figures for advertising, processor vendors optimise the processor architecture for dense linear algebra.

Modern processors contain vector instructions that can operate on multiple floating-point numbers simultaneously with the same operation (for example, multiplication, addition). These vector instructions rely on the floating point numbers to be stored contiguously in memory (*i.e.* be a continuous part of an array). These vector instructions are known as SIMD (Single-Instruction, Multiple-Data) instructions.

Sparse linear algebra

When matrices have large numbers of zero values it becomes advantageous to store only the non-zero entries and their indices. Benefits are reduced storage requirements and the possibility to use fewer FLOPs for a given linear algebra operation.

Given that you store only the non-zero values in an array and their indices the algorithms for performing linear algebra operation are changed from the dense versions above: memory access is usually non-contiguous. As processors are generally designed to perform well for contiguous data, these algorithms can suffer from memory-related performance problems.

Spectral methods

In many scientific and engineering models the data is operated on in both a *spectral* (or frequency) domain which is transformed from and/or to a spatial (or time) domain. For example, it can often be useful to operate on a periodically repeating function in the spectral domain and then convert to the spatial domain for further operations.

The most common numerical method used to transform between the two domains is a Fast Fourier Transform (FFT). FFTs are extremely efficient and can make excellent use of modern processor architectures.

A large number of quantum-mechanical programs that compute the electronic structure of condensed-phase materials rely on FFT algorithms. These programs are the biggest user of CPU-cycles on UK HPC machines.

N-body methods

These algorithms involve calculations that depend on interactions between discrete points (or particles). In the most basic implementation every point interacts with all others leading to a calculation cost that varies with the square of the number of particles (typically denoted $\mathcal{O}(N^2)$). More complex schemes combine the interactions from multiple points to reduce the computational cost. Interactions between points usually depend on the separation of the points and some property of the individual points (*e.g.* mass, charge).

These algorithms are widely used in both materials science and astrophysical simulations. The codes you write in this course lie in this category. The N-body calculation itself is usually combined with some form of integration method (time-based or stochastic) to propagate the system and sample the available *phase-space*. More information on time-integration methods is included below.

In materials science each point often corresponds to a single atom which can interact with other atoms in the simulation by a number of different interaction potential types. The collection of interaction potential definitions is known as the *forcefield*. Instead of representing atoms the points can be chosen to represent a distinct group of atoms or even virtual sites within the material. The parameters for any of these forcefields are usually fitted to data from experiments or from first-principles simulations.

In astrophysical simulations the points can represent different things depending on the type of simulation being performed. For example, when simulating a solar system the points may represent objects such as planets, moons and comets. When simulating galaxial dynamics, we may use the points to represent stars. In contrast to materials science, the only interaction potential usually considered is that due to gravity (although this treatment can be classical or relativistic).

Structured grids

In a structured grid data is arranged on regular multidimensional grid with the calculation proceeding through a sequence of grid update steps. At each step, each point is updated using the values from grid points in a small neighbourhood around the point. The pattern of the set of neighbouring particles (or cells) used in the update calculation is sometimes known as the *stencil*.

These algorithms are usually well suited to current computing architectures as the memory-access is extremely regular.

Examples of simulations that use structured grids are usually from the engineering or climate simulation communities. Computational fluid dynamics or grid-based whole Earth simulation codes use the data from neighbouring simulation cells to determine the update to the current cell.

A more complex version of the scheme is known as adaptive mesh refinement where higher resolution grids are overlayed on top of the the original grid in regions of interest

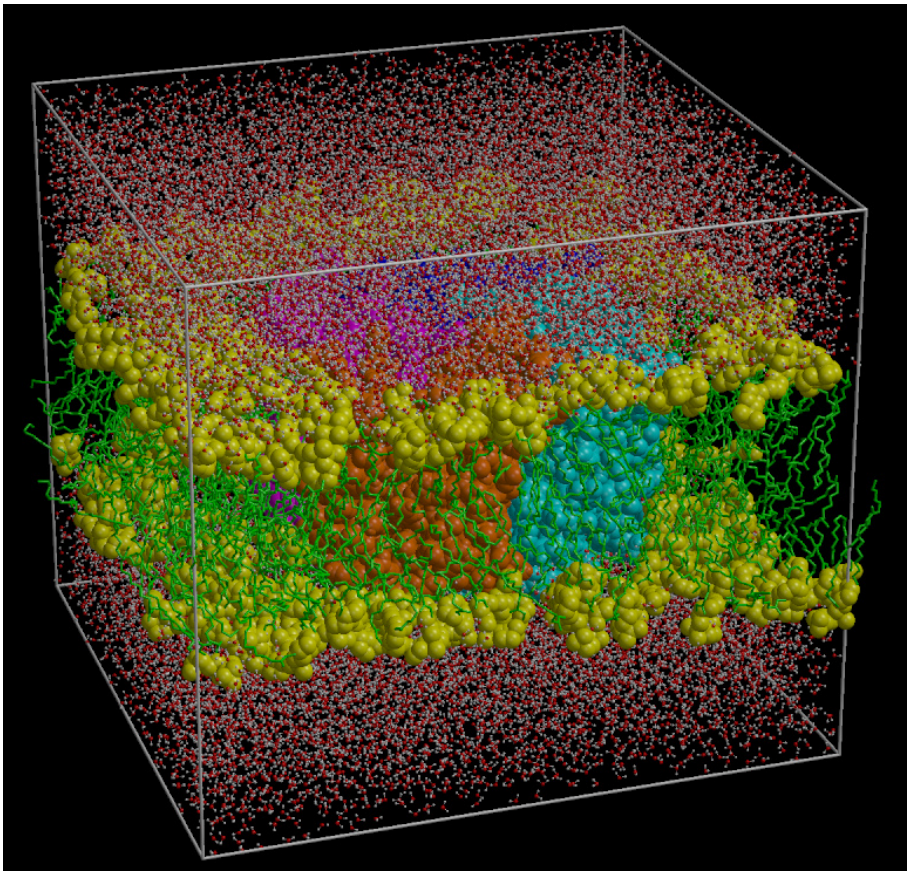

Figure 4.2.: Image of a simulation of the Aquaporin-1 transmembrane protein in a lipid bilayer.

where additonal spatial resolution is required.

Unstructured grids

An unstructured grid algorithm is similar in concept to the structured grids described above with the additional complication that the grid (or mesh) can now be irregular.

Many modelling problems involve objects with irregular geometric definitions and often the natural way to describe such a problem is to define a mesh that covers the surfaces and volumes of the objects being simulated. All the entities in the mesh - points, edges, faces, volumes - must be explicitly described in the simulation and the connection information stored. Different regions of the simulation may have very different resolutions depending upon the information required leading to comlpex mesh descriptions.

These algorithms can contain many irregular memory access and can prove challenging to run efficiently on modern computer architectures which are designed for regular memory access.

This type of simulation is extremely common and is has a wide range of applications from astrophysical simulations (dark matter structure of the universe), through oceanographic modelling and biomechanical simulations to engineering simulations.

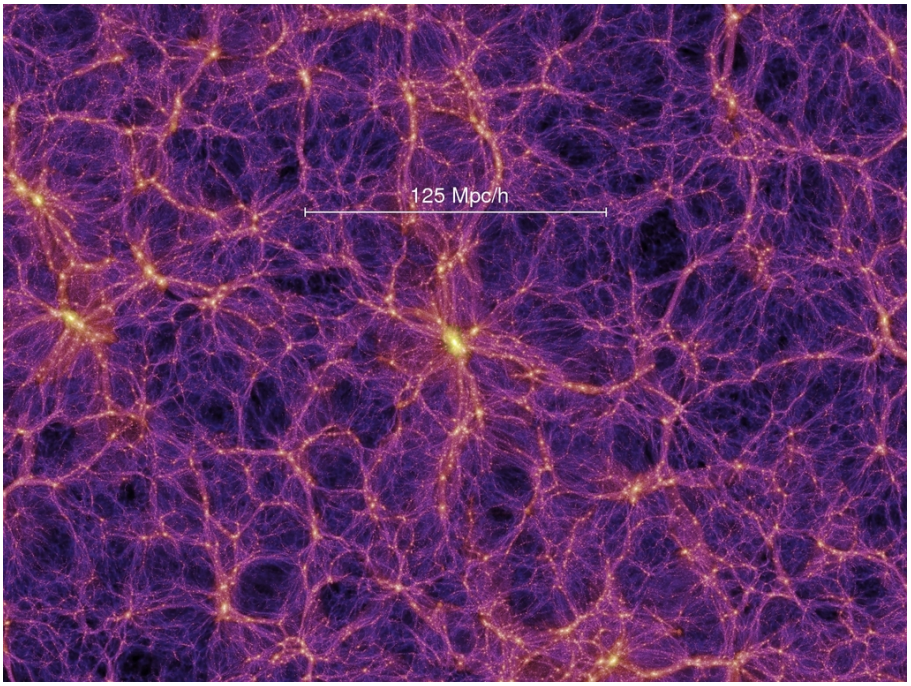

Figure 4.3.: Image of dark matter in full-universe N-body simulation.

Monte Carlo methods

A Monte Carlo algorithm uses repeated random trials to statistically sample the space of the problem being studied. Each of the trials should be statistically independent. The computation involved usually depends on the problem being treated and can be one of the algorithmic patterns described above (for example, Monte Carlo methods are often combined with N-body methods in materials science problems). Monte Carlo algorithms are an example of a *stochastic* method.

One of the simplest examples of a Monte Carlo algorithm is to randomly throw virtual darts at a circle inscribed into a square board and estimate the value of Pi based on the fraction of those that land inside and outside the circle.

4.1.3. Parallel programming

Although many problems of scientific interest can be treated on a single workstation, large numbers of simulations require more processing power and memory than can be made available from a single piece of hardware such as this. The solution is to link large numbers of processors (and associated memory) together using some form of *interconnect* to create a *massively parallel processing* (MPP) resource. On these types of computers a large number of computing units (*nodes*) work in a coordinated fashion to run the simulation.

Parallel programming techniques are used to exploit these resources and require an extra level of complexity over standard serial programming. All of the algorithmic patterns described above have parallel analogues and the design of efficient parallel algorithms is

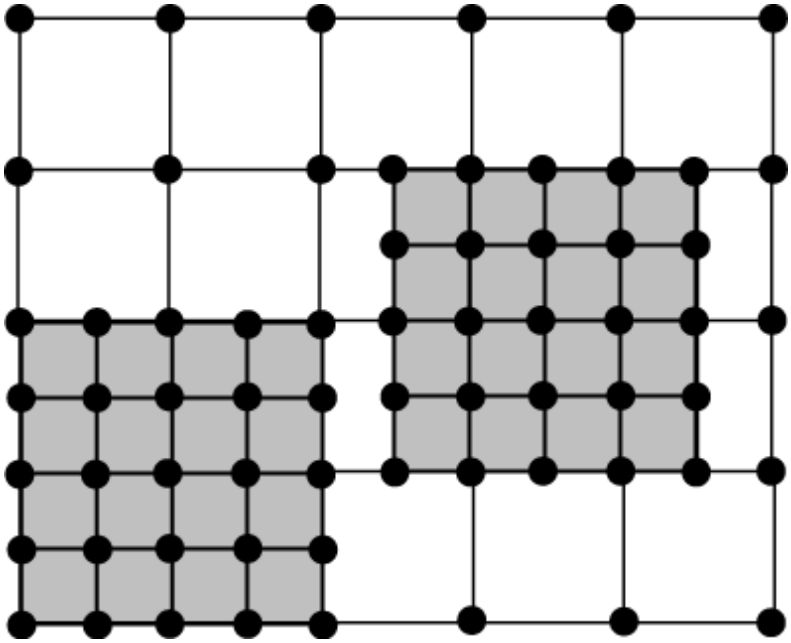

Figure 4.4.: A 2D structured grid with areas of higher resolution grids.

an extremely active area of current research.

4.2. Algorithms

A model will almost always be represented in some mathematical way. When writing a computer code to perform a simulation we first transform our mathematical description of the model into an *algorithm*: a step-by-step guide that tells the computer how to run the simulation.

4.2.1. Converting a mathematical representation to an algorithm

For example, the mathematical representation of the gravitational potential energy between several bodies (assuming $G = 1$) is:

$$U(\mathbf{r}) = -\frac{1}{2}\sum_{i \neq j} \frac{m_i m_j}{r_{ij}} \tag{4.1}$$

How do we break this down into an algorithm for calculating the gravitational potential energy? There are a number of ways we could do this but one such algorithm would be:

1. Choose a pair of particles (i.e. loop over particle indices)
2. Compute the separation between the two bodies, r_{ij} and store this value
3. Compute the product of the masses of the two bodies, and divide it by their separation,
4. add the previous value to a temporary variable used to keep track of the potential energy,
5. and go back to step 1 if there are any particle pairs not considered yet.

This is, of course, a very simple example. Most simulations will consist of a number of different steps with different algorithms which need to be performed in a certain sequence.

4.2.2. Supporting algorithms

As well as the mathematics that go into the final full simulation there are also a number of supporting algorithms that are needed to use a simulation in practice. For example, reading input data in from a file (in the example above we would need to know the position vectors and masses of the two particles).

This type of algorithm is not usually represented mathematically but will often look something like:

1. Check that specified input file exists on filesystem
   - If it does exist, go to step 2
   - If it does not exist, print an error message and stop
2. Read masses and positions from the file
3. Close the file

4.2.3. Converting an algorithm to actual code

You can hopefully see that once we have produced the algorithm it should be relatively simple to convert it into actual computer code. For the example above, a pseudo-Python code fragment to compute the gravitational energy between particles in 3-dimensional Cartesian space would be:

```
# Reset accumulating variable
potential = 0
# loop over all particle pairs
for pair in all_possible_pairs:
    # Separation between particles
    r = math.sqrt( (xj-xi)**2 + (yj-yi)**2 + (zj-zi)**2 )
    # Gravitational energy
    g_energy = -mi*mj/r
    # Scaling by gravitational constant
    potential += g_energy
```

How would you iterate over particle indices to account for all particle pairs? How would you avoid the case $i = j$? And double-counting, so we don't need the $\frac{1}{2}$ factor and, more importantly, avoid half the calculations? *Hint: the* `range()` *function can have more than one argument*

4.2.4. A more complicated example

A more complex example would be an algorithm to perform a 2D matrix multiplication:

$$C = AB \tag{4.2}$$

The algorithm could look like:

1. Loop over number of rows in A with i as index
2. Loop over number of columns in B with j as index
3. Zero running total
4. Loop over number of rows in B with k as index
5. Compute $a_{i,k} * b_{k,j}$ and add to running total
6. Finish k loop
7. Set $c_{i,j}$ equal to running total
8. Finish j loop
9. Finish i loop

Translated to Python code, this algorithm looks like:

```
for i in range(nrowa):
    for j in range(ncolb):
        sum = 0.0
        for k in range(nrowb):
            sum = sum + a[i][k]*b[k][j]
        c[i][j]=sum
```

Note here where the loops begin and end. Since Python loops are denoted by changes in indentation level, you should be very careful, and aim for consistent spacing and grouping of code lines throughout your code.

4.2.5. What about parallel programs?

This becomes much more complicated as the algorithm now has to take account of how data is shared between the different parallel tasks. The design of parallel algorithms is one of the key components of writing parallel code that can scale effectively on modern supercomputers and also of extracting the best performance from multicore processors.

4.3. Time Integration

Often we want to simulate the change in the state of a system over time. Time integration algorithms give us a method of propagating the system through time using the properties of the current state to compute how the state should change.

In all these methods we divide time into discrete chunks with the size of the chunks specified by the *time-step*. The larger the time-step we use, the bigger each jump in time in the simulation leading to a longer total amount of time simulated. As we will see below, however, larger time-steps can lead to numerical inaccuracies which invalidate our model. The goal is usually to choose a time-integration algorithm which allows you to pick the largest time-step possible while maintaining numerical accuracy.

In this course we will discuss time integration in terms of N-body methods where the interaction between two particles is computed in terms of some pair-wise interaction potential (for example, gravity).

We will also assume that the motion of the particles is governed by Newton's Laws of Motion - *i.e.* we are using a system of classical mechanics. Newton's Laws state:

$$\vec{F} = m\vec{a} \tag{4.3}$$

$$\vec{a} = \frac{d\vec{v}}{dt} \tag{4.4}$$

$$\vec{v} = \frac{d\vec{x}}{dt} \tag{4.5}$$

so given:

- a set of initial positions for our particles;
- an interaction potential between them (*i.e.* a force calculation);

we can calculate the force, use this to calculate the velocity of a particular particle and then use this velocity to update the position. Here, we have created an algorithm based on the mathematical expression of Newton's Laws of motion. This algorithm looks like:

1. Compute the total force vector for a particle due to its interaction with all other particles.
2. Calculate the particle's acceleration vector based on this force.
3. Calculate the particle's velocity vector based on this acceleration.
4. Update the particle's position vector based on this velocity.

If we repeat this for all the particles in the system then we will produce a new *configuration*. We can now repeat the algorithm starting again at step 1 (as the force vector depends on the position vectors which have now changed) and we will be simulating the motion of the particles over time.

4.3.1. Taylor's expansion

For the above algorithm to work we need to be able to express our continuous derivatives in a numerical form that can be evaluated on the computer. Taylor's expansion gives us a way of doing this. It shows that we can represent the function at a single point as an infinite series of its derivatives evaluated at that point. *i.e.*

$$f(x+\delta x) = \sum_{n=0}^{\infty} \frac{d^n f(x)}{dx^n} \frac{\delta x^n}{n!} \tag{4.6}$$

4.3.2. Euler integration scheme

The simplest implementation is to truncate the Taylor expansion at the first order:

$$f(x+\delta x) = f(x) + f'(x)\delta x,\ f'(x) \equiv df(x)/dx. \tag{4.7}$$

In the case of time integration we are interested in the positions $\vec{x}(t)$ and velocities, $\vec{v}(t)$ as a function of time, t. The time step we use in the numerical inegration is given by δt. This corresponds to similar updates to both the velocity and the position:

$$\vec{x}(t+\delta t) = \vec{x}(t) + \vec{v}(t)\delta t \quad (\text{noting } \vec{v}(t) = d\vec{x}/dt) \tag{4.8}$$

$$\vec{v}(t+\delta t) = \vec{v}(t) + \vec{a}(t)\delta t \quad (\text{noting } \vec{a}(t) = d\vec{v}/dt). \tag{4.9}$$

This is known as the Euler scheme (after Leonhard Euler) and is illustrated in Figure 4.5.

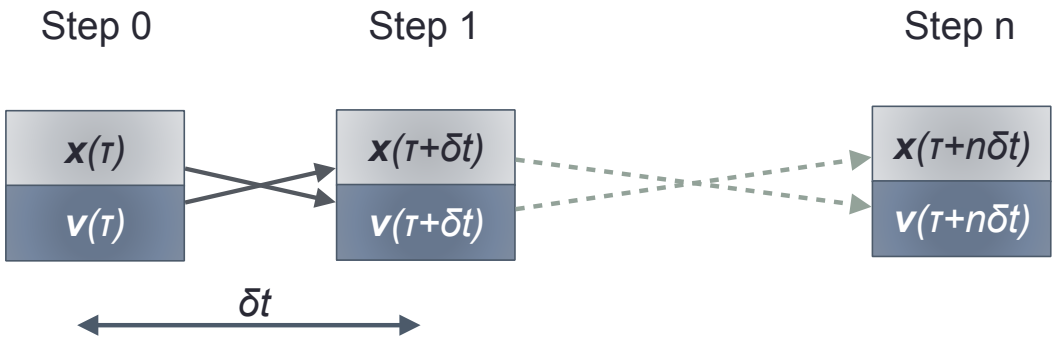

Figure 4.5.: The Euler time integration scheme.

4.3.3. Symplectic Euler integration scheme

The Euler scheme is not very good. The quasi-simultaneous updates of positions and velocities can lead to large errors accumulating over the course of the simulation. A simple yet effective improvement is the symplectic Euler scheme. There, at every time-step, we first update the particle positions based on their velocities, then use the *updated* positions to retrieve updated forces and thus velocities.

$$\vec{x}(t+\delta t) = \vec{x}(t) + \vec{v}(t)\delta t \tag{4.10}$$
$$\vec{v}(t+\delta t) = \vec{v}(t) + \vec{a}(t+\delta t)\delta t. \tag{4.11}$$

The symplectic Euler scheme is illustrated in Figure 4.6. It is the simplest of so-called "symplectic" time integration schemes, which conserve a Hamiltonian slightly perturbed from the original, and thus have superior long-term stability of simulations.

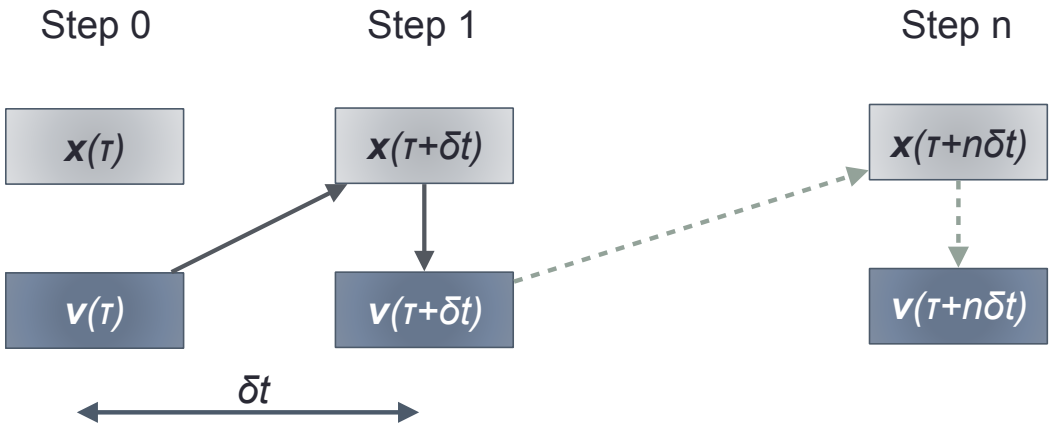

Figure 4.6.: The symplectic Euler time integration scheme.

4.3.4. Velocity Verlet integration scheme

Newton's Laws are second-order differential equations. We can therefore expand the positions $x(t)$ up to second order as both the first derivative, $v(t)$, and the second derivative, $a(t)$, are well-defined. We can not do a second-order expansion of the velocities $v(t)$, because their second derivative, $\dot{a}(t)$, is not defined through the physical laws. However, we can improve any first-order expansion by taking a weighted average of the first derivatives from a series of previous time steps. The simplest integration scheme exploiting these options is the velocity Verlet integration scheme, one of the most widely used integration schemes in molecular dynamics and other particle time evolution integrations.

The velocity Verlet integration scheme combines second-order Taylor expansions of the particle positions and time-averaged first-order expansions of their velocities, in the following way:

$$\vec{x}(t+\delta t) = \vec{x}(t) + \vec{v}(t)\delta t + \frac{1}{2}a(t)\delta t^2 \tag{4.12}$$
$$\vec{v}(t+\delta t) = \vec{v}(t) + \frac{1}{2}(\vec{a}(t) + \vec{a}(t+\delta t))\delta t. \tag{4.13}$$

The velocity Verlet scheme is illustrated in Figure 4.7.

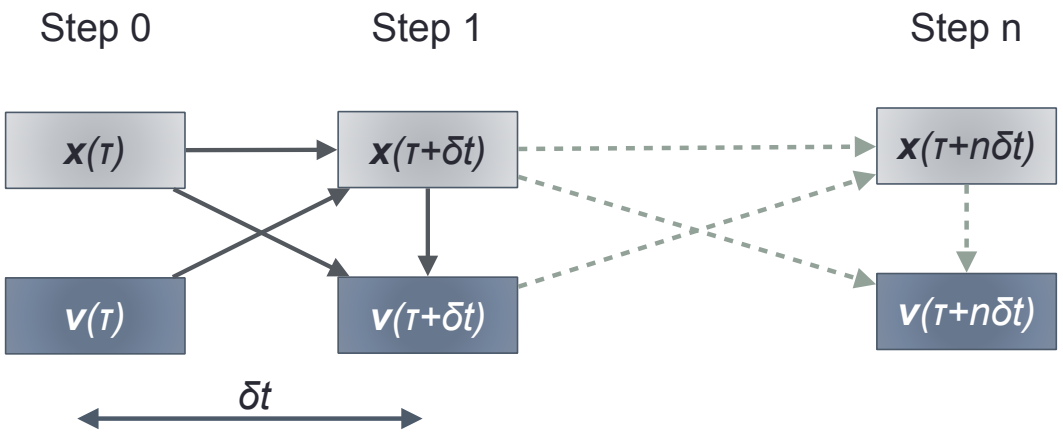

Figure 4.7.: The velocity Verlet time integration scheme.

4.3.5. Synchronicity

Independent of which integrator we use, we need to make sure we adhere to synchronicity throughout the simulation. This can be an issue if some section of the integration scheme relies on updated values, as done in both the symplectic Euler and velocity Verlet integrators. In both schemes, we need to determine the accelerations $\vec{a}(t+\delta t)$ on the particles, which we obtain, using Newton's first law, from the forces $\vec{F}(t+\delta t)$. In a N-body system, to obtain the correct force $\vec{F}_i(t+\delta t)$ on particle i, we need to use the updated positions $\{\vec{x}_j(t+\delta t)\}$ of *all* particles. Thus, at every time step, we need to update *all* particle positions before updating *all* particle velocities, using the respective equations given above.

4.3.6. Verification and validation

One of the biggest problems for many scientific simulation programs is *verifying* that it operating correctly and *validating* that it is giving the correct answer (for the model):

- Verification is solving the equations right;
- Validation is solving the right equations.

Sources of uncertainty

To be able to verify and validate our program we need to be aware of the uncertainties present in the our program - these generally arise from three different sources:

1. Construction of the conceptual model - these are uncertainties from the model of the physical world that we are basing our simulations on.
2. Formulation of the mathematical model - these are uncertainties from the mathematical techniques we are using to represent our model.

3. Computation of the simulation results - these are uncertainties from the way the mathematics are transferred to an algorithm on the computer or from the limitations of the computer hardware itself.

In this course you are given the model that we are working within so we will not be so concerned with the first source of uncertainties but it is worth getting into the habit of thinking about possible uncertainties and limitations that arise from any theoretical framework you are using. For example, in atomic simulations using classical mechanics we would not expect to be able to model any processes that depend on a quantum-mechanical description of the underlying physics.

Likewise, we will not be too concerned with the second source of uncertainties as you will usually be told the mathematical technique we are using for a particular problem. We do investigate this in Exercise 3 where we look at the different levels of accuracy between the symplectic Euler and the velocity Verlet methods for performing time integration. The different uncertainties associated with the two methods are fundamental properties of the different mathematical formulations.

You should be aware of uncertainties arising from the third source. For example, we detail in section 4.4 why it can be difficult to compare two floating-point numbers for equivalence in a program. You need to be aware of these uncertainties when *verifying* that your program is functioning correctly.

Verifying your program is working

There are many different ways of verifying that your program is working correctly - some more sophisticated than others. The standard technique used for simple programs (such as the majority produced on this course) is to pick a simple example that you can calculate the answer for by hand and compare the output from your program to this known solution. If your program gets the answer correct then you can have more confidence that it will produce the correct answer for cases which you cannot compute by hand.

We use this technique in Exercise 2 where we use a number of known vector identities to verify that our vector methods are functioning correctly. You should use this technique as a matter of course as you are writing your programs.

The write-up for the projects will expect you to provide evidence that your simulation program is performing correctly.

If you suspect that your program is producing incorrect output then you should make sure you can verify the individual steps in the calculation to track down the problem. Examples of using this process are given in section 4.5.

4.4. Representing values on a computer

In order to appreciate some of the issues surrounding writing modelling software it is necessary to understand a little of what goes on 'under-the hood' of computers when you run a program.

As you probably already know, all values on a computer - be they numbers, characters, boolean - are stored in a digital representation.

4.4.1. Representing integer numbers

Signed integer numbers are usually stored using 32 bits (each bit can either 0 or 1) or 4 bytes (8 bits in a byte). With negative numbers represented by their 2's complement For example, the decimal integer 13 would be:

$$00000000000000000000000000001101 \tag{4.14}$$

that is:

$$2^0 + 2^2 + 2^3 = 13 \tag{4.15}$$

and the decimal integer -13 would be:

$$11111111111111111111111111110011 \tag{4.16}$$

that is:

$$-2^{31} + 2^{30} + 2^{29} + \cdots + 2^4 + 2^1 + 2^0 = -13 \tag{4.17}$$

So all integer numbers are represented **exactly** in a binary representation. Although the number of bits places a limit on the minimum and maximum integers that can be represented. For a 32-bit integer the mimimum number that can be represented is -2,147,483,648 ($-(2^{31})$) and the maximum is 2,147,483,647 ($2^{31} - 1$).

The situation becomes more complex for non-integer numbers and their representation is dicussed below in the section 4.4.2.

Integer division

One common trap that programmers of all abilities and experience fall into quite often is a result of the fact that any operation involving two integer numbers may result in an integer result - in which case any fractional part will be lost. This can cause unexpected results when dividing two numbers. For example, the following statements:

```
>>> i = 1
>>> j = 3
>>> a = i / j
```

could result in the variable 'a' containing the value 0, or the value $0.3333\ldots$ – depending on whether Python interprets the '/' operator as integer or real division.

Confusingly, this interpretation has changed between Python versions 2.x and 3.x. In 'old' Python (versions 2.x), the '/' operator refers to integer division, and in the example above, $a = 0$. In 'new' Python (versions 3.x), it refers to real division, and thus $a = 0.3333\ldots$. In Python 3.x, the operator '//' is instead used for integer division. To get a representation of the result as a non-integer in Python 2.x we must convert (or *recast*) the integer numbers to a floating-point representation before we perform the calculation. For example:

```
>>> i = 1
>>> j = 3
>>> a = float(i) / float(j)
```

4.4.2. Representing real numbers

To represent non-integer numbers, computers use a *floating-point* representation as this allows numbers of very different orders to be represented and manipulated as accurately as possible.

We convert our non-integer decimal number into a binary representation that is made up of three components, for a 32-bit binary representation:

- 1 bit to represent the sign (0 = +ve; 1 = -ve)
- 8 bits to represent the exponent
- 23 bits to represent the mantissa

This limits the number of decimal places that can be represented by a binary representation of a floating-point number. If we use 32-bits (usually known as *single-precision*) then we can accurately represent a maximum of about 7 decimal places.

Many scientific codes use *double-precision* floating point numbers with use 64-bits to represent the number. This leads to being able to accurately represent around 15-16 decimal places. In Python the 'float' primitive type denotes double-precison floating-point numbers.

Printing floating-point numbers

In any program you can set an abitrary level of precision to which to print the value of a floating point number. However, you must remember that not all the decimal digits you print may be significant. You can print a single-precision number to 15 decimal places (in the mantissa) but only the first 6 or 7 will actually be significant.

This can be a particular problem when debugging your programs as you can see a value change and think there is something wrong with the program when it is just noise from non-significant decimal digits in the mantissa.

Comparing floating-point numbers

The explanation above indicates why we cannot generally compare two floating point numbers using the '==' and '!=' operators as these operators imply bit-equivalence. Depending on how the floating-point number has been calculated two equivalent decimal numbers may have different binary representations.

If you really need to test for equivalence in your program you need to define a tolerance below which two numbers will be considered equal. For example:

```
# Set the comparison threshold
THRESH = 1.0e-7

# Get two 'equal' numbers
a = 2.1 + 1.0
b = 4.1 - 1.0

# Do the comparison
if (math.abs(a-b) < THRESH):
    print("The numbers are the same.")
```

Manipulating very large and very small numbers

Due to the way in which floating-point numbers are represented you need to take particular care when combining very large and very small numbers in floating-point arithmetic. For example, the mantissa of a very large number may not contain enough precision to perform valid arithmetic when combined with a very small number.

Consider:

```
pi = 3.1415926
bigFactor = 6.023e23
mySum = bigFactor*pi + pi
```

At the end of this calculation the variables `mySum` and `bigFactor*pi` represent the same number whereas strictly they should be different.

4.5. Errors and Debugging

One of the key programming skills that you should learn is the ability to diagnose any errors in your program and fix them.

A programming mistake that leads to an error at compile- or run-time is known as a *bug* and the process of finding and removing them is known as *debugging* the program. Python is an interpreted language without a compilation step, and hence only run-time errors exist.

There are three classes of errors: *syntax errors*, *semantic errors* and *logic errors*.

4.5.1. Syntax errors

Syntax errors occur when you write code in a manner not allowed by the language. For example, in Python, if you omit the colon at the end of a conditional expression line then you would introduce a syntax error.

4.5.2. Semantic errors

A semantic error does not have incorrect syntax but instead the *meaning* of the code is not what was intended. Sematic errors are not usually caught by the syntax check of the interpreter as the code is syntactically correct but instead cause your program to terminate abnormally or hang at *run-time*. This behaviour is also know as a *crash*.

4.5.3. Logic errors

A logic error is a special case of a semantic error that does not cause the code to crash but instead the code continues to run with an incorrect *internal state*.

This means that variables will have an incorrect values and/or the program will follow the incorrect branch in 'if' constructs or 'for' loops. This type of run-time error usually manifests itself as the program producing incorrect output. For example, in a astrophysical simulation gravity would act in a repulsive rather than attractive manner (the logic error in this case would probably be that the gravitational force has the wrong sign).

4.5.4. Run-time errors

Errors at run-time will prevent you from running your program. The interpreter will usually exit with an error that indicates why the program execution has stopped and the line number of your source code that it thinks is causing the problem.

For example, if I forget to include the colon at the end of a conditional `if` statement I may see something like:

```
user@cplab001 $ python3 newton.py
  File "newton.py", line 10
    if (m1==m2)
```

```
                ^
SyntaxError: invalid syntax
```

For inexperienced programmers the error messages produced by the interpreter can sometimes be a little cryptic. In addition, the line number associated with the error may not be where the problem actually is (particularly when the error is due to a syntax error such as unmatched parenthesis).

When debugging run-time errors you should examine the interpreter message to understand what has gone wrong and use the line number listed as a starting point when looking for the bug.

A set of the most common error messages from the Python intepreter and their 'usual' causes is given in `Appendix B: Error messages`.

4.5.5. Incorrect results

Incorrect results can result from a large number of different bugs but the method for tracking down the problem is usually the same:

1. Select a set of input values that you can compute the output by hand.
2. Insert `print()` statements in you code at strategic points to print the values of key variables.
3. Rerun the code
4. Compare the values your program prints out with those from your calculation by hand.
5. Use the point at which they differ as an indicator of where to start searching for the bug.

If you have a complex line in your code it is often worth splitting it up into smaller steps and printing the intermediate values to make sure you are calculatingwhat you think you are calculating. For example, if I am computing the gravitational energy between two particles separated by a distance r and with masses m_1 and m_2 I could do this in a single step with:

```
e = (m1 * m2) / (r * r)
```

If I find I am getting an error in the energy I could break down the calculation to enable to examine each step independently with:

```
m_prod = m1 * m2
rsq = r * r
e = m_prod / rsq
```

Although this example is trivial it demonstrates how you might break down a calculation into simpler steps.

A list of common Python bugs is listed in `Appendix B: Error messages` but this list is not exhaustive.

One of the reasons that this is a practical course is to give you experience of debugging reasonably complex programs.

Appendix

Appendix A.

General tips

A.1. Organise your work

You will find it much easier to keep track of the versions of various files and input and output from programs if you start out with a well-organised directory structure.

A good suggestion is to have a subdirectory for each of the checkpoints you are working on and to within each subdirectory have a further subdirectory where you keep a copy of your latest working code (so that you can always roll back if you make a change that breaks the code).

For example, if I am working on three checkpoints, the directory structure would look like:

```
checkpoint1/
checkpoint1/latest_working
checkpoint2/
checkpoint2/latest_working
checkpoint3/
checkpoint3/latest_working
```

Alternatively, you could use git when commiting changes, as this makes rolling changes back easy, and keeps a backup online. Which is a good reminder for the most basic lesson of them all: Backup your work!

A.2. Debug before asking for help

You should follow the basic debugging advice in the notes above before asking for help from a demonstrator as if they find you have not tried this then they will simply ask you to try yourself before helping out. You will have wasted the time waiting for a demonstrator if you do not try to debug before asking for help.

This is also related to the art of asking good questions, especially online. While lecturers and TA's are here to help you, they are also meant to help 50–odd more people per session. Help them help you. The sentence *"It doesn't work"* confers zero information. If it did work, after all, you wouldn't be asking a question! You could go through a set of steps:

1. What does not work, and what is the error message?

2. Can you see a mistake in the line of the error, or in the immediate ones?

3. Can you compare the expected vs. the actual outcome?

4. Have you tried fixing it? What steps have you tried?

5. If the problem feels very complicated, can you think of an easy test case to help you debug the issue?

The final point is particularly relevant in the case of the projects. Many students are tempted to code it all in one go, and only start debugging once a basic simulation is finished. And of course the *"basic"* test is either a full Solar System, or a Lennard–Jones solid with 128 particles. Not so fast!

Simple 3-body systems can be set up for both cases, in which you can compute initial forces simply, as well as the initial updates. If these simple but non-trivial cases work, the rest is likely to follow.

Appendix B.

Error Messages

B.1. Syntax errors

Some errors are caused by violations of the syntax of Python. They are usually announced by the message `SyntaxError: invalid syntax` if you attempt to run the code. Although they are easy to understand, there is no easy way to find the exact cause of such errors except by checking the code around the location of the error character by character looking for the syntax error.

Some of the most common syntax errors are explained below.

1. **Misspelling of keywords**

   Protected keywords such as `for`, `def`, and `else` must be spelled correctly. Python is case-sensitive, so be sure not to capitalize any of those keywords either.

```
>>> While (n>5):
  File "<stdin>", line 1
    While (n>5):
               ^
SyntaxError: invalid syntax
```

2. **Missing characters**

   The syntax of Python is very specific about the required punctuation. An error occurs, for example, if you forget a colon at the end of a conditional statement or do not balance parentheses:

```
>>> if (i > j)
  File "<stdin>", line 1
    if (i > j)
             ^
SyntaxError: invalid syntax
>>> if (i > j :
  File "<stdin>", line 1
    if (i > j :
              ^
SyntaxError: invalid syntax
```

Unfortunately, this syntax error is not necessarily caught precisely at the point of the mistake so you must carefully check the preceding characters in the line or even in a previous line in order to find the problem.

3. **Unclosed string expressions**

   String literals must be enclosed in quotation marks. This error occurs if you fail to terminate the literal with quotation marks. Fortunately, the syntax of Python requires that a string literal appear entirely on one line so the error message appears on the same line as the mistake. If you need a string literal that is longer than a single line, create two or more literals and concatenate them with +:

```
>>> my_string = "hello " + " world
  File "<stdin>", line 1
    my_string = "hello " + " world
                                 ^
SyntaxError: EOL while scanning string literal
```

4. **Incorrect indentation**

   Python is quite strict about indentation levels, i.e. the number of white spaces at the beginning of each line. That is because four white spaces denote a sub-section of code, inside conditionals, loops, methods, etc. – Python thus expects indents for all such sections, and nowhere else. The error is straightforward and usually easy to fix.

```
>>> for n in range(5):
... print(n**2)
  File "<stdin>", line 2
    print(n**2)
        ^
IndentationError: expected an indented block
>>> x = 0
>>>   y = 1
  File "<stdin>", line 1
    y = 1
    ^
IndentationError: unexpected indent
```

   While not often an issue with the older Python2, the Python3 interpreter will exit with an error if the code is indented with a mix of spaces and tabs. Although not visible on screen, error messages clearly state where such mix-ups occur.

```
user@cplab001 $ python3 newton.py
  File "newton.py", line 16
    n = n + 1
            ^
TabError: inconsistent use of tabs and spaces in indentation
```

There are two ways to avoid ending up with a very large clash of tabs and spaces:

a) Code everything by yourself, never be involved in a group project[1], and always use the same editor with the same settings

b) Make sure your editor is set up appropriately, ideally following the Python standard (one indentation = 4 spaces).

B.2. Run-time errors

This group of errors results from code that is syntactically correct but violates the semantics of Python.

1. **Uninitialized variables**

   An instance variable declared in a class has a default initial value. However, a local variable declared within a method does not, so you are required to initialize it before use, either in its declaration or in an assignment statement:

```
>>> x = y**2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'y' is not defined
```

   The variable must be initialized on the path from declaration to use. If that particular path is not taken during the specific execution of the code, it will run through. In the example below, setting `x=8` on the first line will run fine.

```
x = 0    # NameError, triggered in else: section
if (x == 8):
    y = x**2
else:
    x = y**2
Traceback (most recent call last):
  File "newton.py", line 5, in <module>
    x = y**2
NameError: name 'y' is not defined
```

2. **Misspelling of variables**

   A subsidiary of the previous error: Python is case-sensitive and also does not understand typos in variable names. Either will treat the respective variable as new, which is likely to be uninitialised:

```
>>> my_var = 1.0
>>> square = My_var**2
```

[1] This prescient line was originally written before the COVID-19 epidemic

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

A variant of this error happens when you mistype a variable... and the incorrect name refers to another, actually initialised variable. These issues can be very hard to detect. The best way to avoid them is to name your variables such that this is unlikely to happen.

3. **Wrong application of operators**

   Operators are only defined for certain types, although implicit type conversion is allowed between certain numeric and even logical types. If you attempt to use an operator on data types for which it is not defined, a `TypeError` will be shown:

```
>>> a = 5
>>> b = True
>>> a + b
6
>>> c = "True"
>>> a + c
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

   From the example above, it is possible in Python (somewhat counterintuitively) to "add" `int` and `bool` variables, but not `int` and `str` variables.

4. **Out-of-range for lists and strings**

   Attempts to access list elements or string characters that go beyond the length of either will result in an `IndexError`:

```
>>> a = [x for x in range(10) ]
>>> a[12]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> c = "hello"
>>> c[12]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

   The same error type will be shown if you attempt to access a Numpy arrary beyond its length.

5. **Non-existent dictionary keys**

   Dictionaries don't have indices, but entries are accessed by key. If you attempt to access a key that does not exist, a `KeyError` will be thrown:

```
>>> my_dict = { "key1":"Adam", "key2":"Boris" }
>>> my_dict["key3"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'key3'
```

6. **File access denied**

   If you attempt to open a file that does not exist or that you don't have access to, you will see an `IOError` (in Python 2.x) or a `FileNotFoundError` (in Python 3.x):

```
>>> my_file = open("nonexistent.dat", "r")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'nonexistent.dat'
```

7. **Import of external Python modules**

   If you attempt to import external modules be sure to avoid typos and capitalization errors, as these will result in an `ImportError`. These are usually straightforward to identify:

```
>>> import Numpy
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named Numpy
```

Appendix C.

Modern computer hardware

A basic understanding of what components constitute a modern computer is required to understand scientific programming an many of the issues behind algorithmic design, accuracy issues and performance optimisation.

Most computers have three major components:

- A *central processing unit (CPU)* which performs the calculations and other operations.
- *Main memory* which stores your program while it is running and supplies the information to the CPU.
- *Data storage* which houses your files.

The CPU is connected to main memory via a *bus* which allows fast information transfer (Figure C.1). Main memory is also connected to the data storage via a bus which is slower but allows more data through (wider).

C.1. Central Processing Unit (CPU)

CPU's generally consist of many different components and a full discussion of CPU architecture is beyond the scope of this course. We will briefly discuss the components that are used to perform the actual calculations: the *floating-point unit* (FPU) and *integer unit* (IU). Each of these components is responsible for performing arithmetic operations on floating-point (real) numbers and integer numbers respectively. The existence of these two distinct units is why floating-point and integer numbers are treated differently in programming languages. Generally, integer numbers are stored exactly and integer arithmetic is performed exactly while the floating-point equivalents are stored and performed to a certain precision.

Most modern CPUs include the ability to perform *vector* operations on floating-point numbers. Essentially, the same arithmetic operation (add, multiply, divide) can be performed on a contiguous set of operands simultaneously. This can be used to speed up computation significantly with the caveat that the operands must be stored sequentially in memory. This is also known as *Single Instruction, Multiple Data* (SIMD).

Another recent innovation in CPU development has been the advent of *multicore* designs. Each physical piece of silicon (a chip) usually contains more than one CPU, each

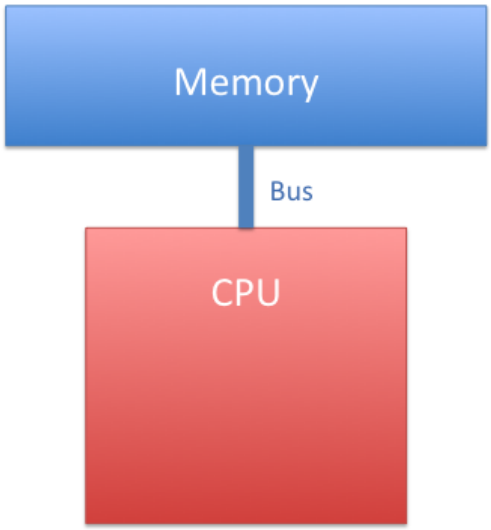

Figure C.1.: CPU and memory connected by a memory bus.

with its own IU and FPU. Much modern programming effort is being expended on producing simulation codes that can effectively use these multiple *cores* in parallel.

C.2. Memory

How a computer stores the values associated with a program is useful knowledge for understanding what happens when you are working with scientific programs.

C.2.1. Storing variables and memory references

Storing transient data such as that used only when a program is running is achieved by using main memory. In order to perform operations on this data the CPU must access it and to do this it must know where in main memory the data is stored. All data in main memory can be located by using its *reference* which is simply an address in memory. For example, if we set up a variable 'x' to hold a floating-point number then when we ask for the value of 'x' in our program it uses the reference associted with the variable 'x' to find the address in memory where bytes corresponding to this variable's value are stored.

C.2.2. Copying variables

This introduces a complication when copying variables - when we copy a variable, do we copy the value and create a new reference to this identical value or do we just copy the reference, so that the new variable and the old variable refer to the same address in memory? Confusingly, Python does both depending on the type of the variable you are copying.

If you copy an immutable type (int, float, bool, tuple) then you will create a new variable with its own reference to its own address in memory that contains the bytes representing the value. For example:

```
a = 2012
b = a
```

will result in two variables referring to different addresses in memory containing the same (immutable) values. If we then assign `a` to something else, that will not affect `b`, which is pointing to an immutable integer.

However, if we copy a mutable object in the same way, we only just create a new variable that refers to the same address in memory. For example, if we copy a list with:

```
a = [ x for x in range(5) ]
b = a
```

then the variables 'a' and 'b' will refer to the same address in memory. This means that altering the values of either 'a' or 'b' will seem to alter the values of both variables (although in reality we are only altering a single thing). Let us change the contents of `a[2]`:

```
>>> a = [n for n in range(5)]
>>> b = a
>>> a[2] = 'ohlala'
>>> a
[0, 1, 'ohlala', 3, 4]
>>> b
[0, 1, 'ohlala', 3, 4]
```

which is exactly what we saw when modifying the values inside `numpy` arrays.

C.2.3. Cache hierarchy

As access to main memory can be very slow with respect to actual calculations on the CPU most modern processor designs include a *cache hierarchy*. A cache is a small amount of memory space actually on the processor which is significantly faster to access than main memory. The processor tries to arrange the data in the cache so that it is reused during a program to speed up the performance and reduce the number of times it has to retrieve data from main memory.

C.3. Supercomputers

The importance of simulation has led to large amounts of money being used to purchase supercomputing resources for use by scientists. Writing code to exploit these resources is complex.

Supercomputers consist of a large number of processors connected together using a high-performance *interconnect* which is used to exchange data between tasks on individual processors (Figure C.2). Data is exchanged using parallel programming models, for example, by using shared-memory programming or message-passing programming.

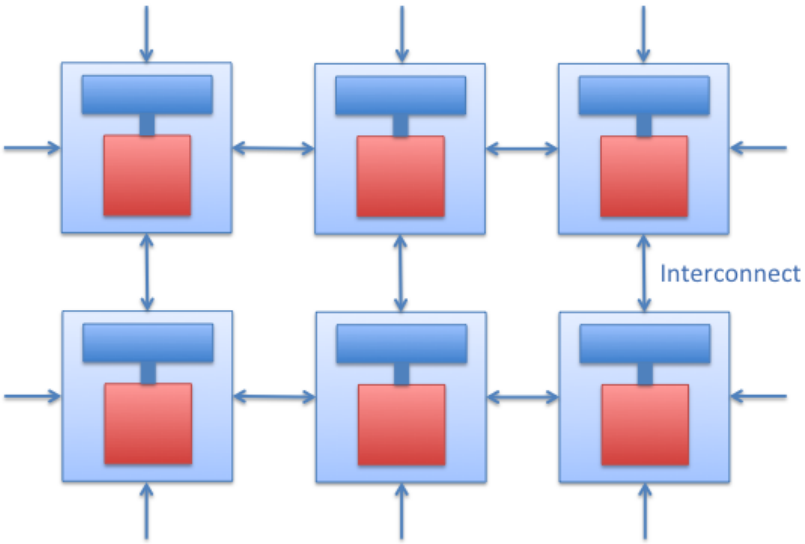

Figure C.2.: Diagram of a distributed-memory supercomputer with individual compute units connected by an interconnect.

C.4. Accelerators

Modern scientific computing resources may also now include *accelerators* which are additional specialised processors attached to the the main CPU. Most often, accelerators are *General-purpose Graphics Processing Units* (GPGPU's) which are large SIMD devices capable of performing hundreds of floating-point operations in parallel. Although these accelerators can yield large gains in performance for some classes of simulation problems, they are in general very difficult to program effectively.

Figure C.3.: The UK national supercomputering facility: ARCHER (`http://www.archer.ac.uk`). It has a total of 118,080 CPU cores and can perform about 60,000 operations per second for every person on Earth.

Appendix D.

Academic Misconduct

Academic Misconduct is a general term that describes any attempt by a student to gain credit for a piece of work by unfair means. This includes plagiarism of all forms — for example copying other people's work and presenting it as your own without proper attribution or cutting and pasting material from websites. This applies not only to text, but also to materials such as images, computer code or experimental data as well as to other people's ideas. Academic Misconduct also includes such behaviour as collusion (sharing of answers to exercises when a piece of work is individually assessed) and cheating, for example taking unauthorised materials into an exam hall.

During your degree, there will be many occasions where you are encouraged to collaborate. This of course does not count as Academic Misconduct, but doing so when not specifically allowed does. Remember that Computer Modelling **no longer group submissions**. We would not request, and in fact we discourage, that you work in a bubble. However, and perhaps unwittingly, students sometimes cross the line by copying another group's solution for a computing checkpoint. The easiest way to avoid this is simply to always write up your work for assessment in your own words. If you are unsure about how you are to conduct a piece of assessment, always ask a member of the course team for advice.

It is important to get this right because Plagiarism and other forms of Academic Misconduct are **very serious offences** and will lead to disciplinary action being taken. In the most severe cases, it could lead to exclusion from the University and termination of your degree. Further guidance on what constitutes plagiarism and tips for avoiding it can be found here: link1, link2.

We routinely check submissions for plagiarism, either with Turnitin's similarity score, or `moss` for semantic code similarity. Sadly, because copying code is so easy, we report a few cases to the School Academic Misconduct Officer every year.