Part II Computational Physics

Austen Lamacraft

1/17/23

Table of contents

Preface

These are the materials for the Part II Physics course Computational Physics, taught in Lent Term 2023 at the University of Cambridge.

Schedule

The course of eight Lectures will take place at 10.00 on Mondays and Fridays in the Pippard Lecture Theatre. After the lectures there will be four computing exercises to be completed in the last four weeks of full Lent term; one per week. Remember that the exercises count for 0.2 units or further work, or roughly 2% of your final mark for the year. Thus each exercise should only take you a few hours.

The schedule is as follows

- First lecture: Monday 23th January
- Last lecture: Friday 17th February
- First exercise: Friday 17th February – Friday 24th February
- Second exercise: Friday 24th February – Friday 3rd March
- Third exercise: Friday 3rd March – Friday 10th March
- Fourth exercise: Friday 10th March – Friday 17th March (last day of full Lent term)

Prerequisites

This course assumes a basic knowledge of the Python language, including variables, control flow, and writing and using functions, at the level of last year's IB course.

Learning outcomes

In this course you will learn

1. About the Python scientific stack (based on the NumPy library)
2. Its use in implementing some common algorithms in computational physics.
3. Basic ideas of computational complexity used in the analysis of algorithms

Outline

Here's a list of topics that I'd like to cover. We make not have time for all of them.

1. Setup. Running Python. Notebooks. Language overview
2. NumPy and friends
3. Floating point and all that
4. Soving differential equations with SciPy
5. Monte Carlo methods
6. Linear algebra with NumPy
7. Introduction to algorithms and complexity
8. The fast Fourier transform
9. Automatic differentiation

These notes. . .

. . . were prepared using Quarto. Each chapter should be thought of as a Jupyter notebook (actually, they *are* Jupyter notebooks), so you'll only see `import numpy as np` once in each chapter, for example.

In several places I've used examples from an earlier version of the course by David Buscher.

1 Introduction

> Science is what we understand well enough to explain to a computer. Art is everything else we do.
>
> Donald Knuth

Not a course on numerical analysis but computational physics

Computation is important for experimental physicists for analysing data. For theoretical physics, computation is used to deal with the awkward fact that physical theories are *generally not tractable*. You can't solve Maxwell's equations, the Navier–Stokes equation, or Schrödinger's equation in any but the simplest situations.

To be blunt, this means that your knowledge of physics, while very nice, is of *no use whatsoever* unless you can write a program to solve more complicated problems. Sorry.

It's important to understand that this need to apply our mathematical descriptions of nature in more general settings was the principal driving force behind the invention of the computer.

More than this,

Church Turing hypothesis

Turing's cathedral

Books

Part IB notes are very good

Garth Wells

https://github.com/CambridgeEngineering/PartIA-Computing-Michaelmas/

Rougier

https://www.labri.fr/perso/nrougier/from-python-to-numpy/ https://github.com/rougier/scientific-visualization-book

2 Getting Going

2.1 Finding your way

Everyone finds their own workflow for coding, depending on their preferred language, editor, how they run their code, and so on. The aim of the sections below is to give a roundup of some popular tools in the Python ecosystem.

2.2 Your coding environment

To run Python code on your computer you will need to have installed the Python language. I recommend the Anaconda distribution as it comes with all the parts of the toolkit we'll need such as Jupyter notebooks and the major libraries NumPy and SciPy.

Try running `python` at the command line. You should get something like

```
Python 3.9.12 (main, Apr  5 2022, 01:53:17)
[Clang 12.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

You should confirm that you are using Python 3 (the command `python3` will also work and guarantee this if you happen to have Python 2 as the default). The prompt `>>>` indicates that you have started the Python interactive shell or REPL and are good to go:

```
print("Hello world!")
1 + 2
```

```
Hello world!
```

```
3
```

To leave and return to the command line, you can run `quit()` or `exit()`.

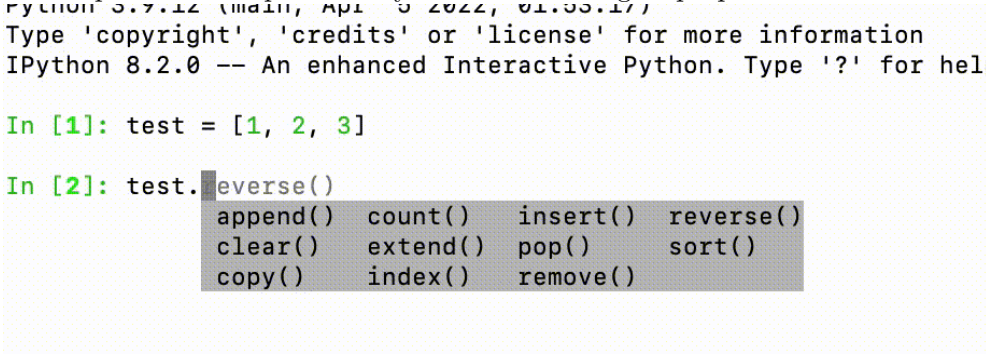
2.3 IPython

If you ran the above command from within `python` you may have noticed that the nice colour scheme that you see above was absent. This is called syntax highlighting and provides a visual guide to the syntax of the language.

IPython is an interactive shell that provides syntax highlighting and much more. If you have installed IPython (it comes with Anaconda) you can start it from the command line with `ipython`.

Among the most helpful features of IPython are:

1. Tab completion: hit `tab` to autocomplete. This is particularly useful for viewing all properties or methods of an object:



2. Typing `?word` or `word?` prints detailed information about an object (`??` provides additional detail).
3. Certain *magic commands* prefixed by `%` that provide certain additional functionality. For example, `%timeit` finds the executation time of a single line statement, which is useful when profiling the performance of code:

```
%timeit L = [n ** 2 for n in range(1000)]
```

```
214 µs ± 3.28 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

`%timeit` automatically runs several times to give some statistics on the execution time. For multiple lines you can use the `%%timeit` magic.

You can find much more exploring the documentation.

2.4 Running a Python program

Python code in a file with a `.py` extension can be run from the command line with `python hello_world.py` or `python -m hello_world`. In the latter case the `-m` option tells the interpreter to look for a *module* called `hello_world`. More on modules below.

From the IPython shell you can instead use `run hello_world.py` or just `run hello_world`.

TODO: These magics are normally documented with a `%`. When is it necessary?

2.5 Importing code

A Python module is just a file containing definition and statements. Breaking long code into modules is good practice for writing clear and reusable software. Users may not want to delve into the details of some function you have written in order to be able to us it, and separating the corresponding code into a separate file is a hygienic way to handle this.

Thus if I make the file `hello_world.py` containing the function:

```
def hello():
    print("Hello world!")
```

I can run this function by first importing the module:

```
import hello_world
hello_world.hello()
```

```
Hello world!
```

Notice that the function `hello` is accessed from the `hello_world` *namespace*. This is to avoid any confusion that may arise if more that one imported module has a function of the same name. If you are confident that's not an issue and want more concise code you can do this:

```
from hello_world import hello
hello()
```

```
Hello world!
```

or even:

```
from hello_world import *
hello()
```

```
Hello world!
```

The issue with the latter is that it may introduce a whole bunch of names that may interfere with things you already defined.

A collection of modules in a folder is called a *package*. You can import a package in the same way and access all the modules using the same `.` notation i.e. `package.module1`, `package.module2`, etc..

Since explicit namespaces are preferred to avoid ambiguity it's common to introduce shorthand names for the package or module you are importing, hence the ubiquitous:

```
import numpy as np
np.arange(10)
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

(You can call it what you like, of course!)

For details about where the interpreter looks to find modules you try to import are in the documentation.

2.6 Installing libraries

99% of the code [1] you run will have been written by somebody else in the form of a library (a collection of modules or packages). Package installation is handled by the command line utilities `pip` or `conda`, the latter being the package manager for the Anaconda distribution. If you have NumPy and SciPy installed you won't need to worry about this too much in this course.

[1] Sticking with integers

2.7 Editors

Modern editors come with a huge number of tools that make writing code much easier, and you would be crazy not to take advantage of them. These range from the visual cues provided by syntax highlighting – which we've already met – to code completion, parameter information and documentation popups as you type. These go under the general heading IntelliSense. The latest hotness is GitHub Copilot, which uses AI to make code suggestions. In my view, these are all part of a continuum of productivity enhancements that enable people to write better code faster. Use them (wisely).

I use Visual Studio Code.

2.8 Notebooks

While software developers write `.py` files, modules and packages, scientists and others doing more exploratory work tend to favour a Notebook format that mixes code, text, and plots. The dominant option is the Jupyter notebook, which comes with the Anaconda distribution and can be started from the command line with `jupyter notebook` (or from the Anaconda Navigator application). This will open the notebook as a web page in your browser, where it can be edited and saved. The default extension is `.ipynb`.

Jupyter notebooks can actually run code in different languages (the processes running a particular language is called a kernel), but the default process is IPython with all the benefits described above.

The text cells can be formatted using Markdown and also support LaTeX equations, which is pretty handy for us.

Google has their own cloud version of the Jupyter notebook called Colab. You can try it out for free, though you have to pay for significant compute. The "next generation" of the Jupyter notebook is called JupyterLab and can be started with `jupyter lab`. Notebook files can be opened in either Jupyter Lab or Jupyter Notebook

2.9 Codespaces

TODO

New from Github...

3 The Python Language

Extensive intro in Part IB

Objects

two language problem

Gotchas

Mutable and immutable

Python pass by reference

https://docs.python-guide.org/writing/gotchas/

4 NumPy and friends

The NumPy package is *the* key building block of the Python scientific ecosystem.

In this chapter we introduce a few of the key concepts. You should refer to the documentation for details. As with any mature software ecosystem, you should first **assume that what you want to achieve *can* be achieved in a highly optimised way within the existing framework**, and only resort to creating your own solution if and when you satisfy yourself that this is not the case.

There are a huge number of resources for learning NumPy online. This is one particular nice and compact tutorial.

4.1 Preamble: objects in Python

Everything in Python is an *object*. For example `[1,2,3]` is a `list`:

```
my_list = [1, 2, 3]
type(my_list)
```

```
list
```

You can think of an object as a container for *properties* and *methods*, the latter being functions associated with the object. Properties and methods are accessed with the `.` syntax. For example, lists have the `append` method, which adds an element to the end of the list:

```
my_list.append("boop")
my_list
```

```
[1, 2, 3, 'boop']
```

With IPython you can see all the available methods by hitting tab:

```
Python 3.9.12 (main, Apr  5 2022, 01:53:17)
Type 'copyright', 'credits' or 'license' for more information
IPython 8.2.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: test = [1, 2, 3]

In [2]: test.reverse()
         append()  count()   insert()  reverse()
         clear()   extend()  pop()     sort()
         copy()    index()   remove()
```

Dunder methods

You can list all of an objects properties and methods using `dir`:

```
dir(my_list)
```

```
['__add__',
 '__class__',
 '__class_getitem__',
 '__contains__',
 '__delattr__',
 '__delitem__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__gt__',
 '__hash__',
 '__iadd__',
 '__imul__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
```

```
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__reversed__',
 '__rmul__',
 '__setattr__',
 '__setitem__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'append',
 'clear',
 'copy',
 'count',
 'extend',
 'index',
 'insert',
 'pop',
 'remove',
 'reverse',
 'sort']
```

Notice that lots of these are methods have a name sandwiched between double underscores and for this reason are called *dunder methods* (or *magic methods*, or just *special methods*). This is to indicate that they are not to be used by you, but by the Python interpreter to implement certain standard functions that apply to many different classes of objects. For instance, when you write `len(my_list)` to find the length of `my_list` Python is actually calling the dunder method `my_list.__len__` which does the job of actually finding the length.

```
my_list.__len__()
```

```
4
```

In this way the same function (`len` in this case) can operate on many different objects, an example of what is called polymorphism in object oriented programming.

4.2 Arrays

The fundamental object in NumPy is the *Array*, which you can think of as a multidimensional version of a list. Let's start with two dimensions to demonstrate:

```
import numpy as np
my_array = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
```

```
type(my_array)
```

```
numpy.ndarray
```

Arrays can be indexed, similar to lists

```
print(my_array[0], my_array[1], my_array[3][1])
```

```
[1 2 3] [4 5 6] 11
```

but – different from a ordinary list of lists – the last one can be much more pleasantly achieved with the syntax

```
my_array[3,1]
```

```
11
```

We also have a generalization of the slice syntax

```
my_array[1:, 1:]
```

```
array([[ 5,  6],
       [ 8,  9],
       [11, 12]])
```

Slicing can be mixed with integer indexing

```
my_array[1:, 1]
```

```
array([ 5,  8, 11])
```

NumPy offers all sorts of fancy indexing options for slicing and dicing your data: see the documentation for details.

A fundamental property of an array is its `shape`:

```
# [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
my_array.shape
```

```
(4, 3)
```

The way to read off the shape of an array is as follows. To begin with you encounter a number of `[` corresponding to the rank of the array (two in the above example). You then scan over a number of entries that give the rightmost (innermost) dimension in the shape tuple before closing `]` (3 here). After a number of 1D arrays `[...]` equal to the next innermost dimension (4 here), we have another closing `]`, and so on.

It's definitely something that will take a bit of time getting used to!

Notice that slicing does not change the rank of the array

```
my_array[1:, 1:].shape
```

```
(3, 2)
```

but integer indexing does

```
my_array[1:, 1].shape
```

```
(3,)
```

NumPy has lots of methods to create arrays with a given shape and populated in different ways:

```
a = np.zeros((2,2))
print(a)

b = np.ones((2,2))
```

```
print(b)

c = np.full((2,2), 5)
print(c)

d = np.random.random((2,2)) # random numbers uniformly in [0.0, 1.0)
print(d)
```

```
[[0. 0.]
 [0. 0.]]
[[1. 1.]
 [1. 1.]]
[[5 5]
 [5 5]]
[[0.06254687 0.06180034]
 [0.55375253 0.67684538]]
```

There are also lots of methods to change the shape of arrays, for example

- numpy.reshape to change the shape of an array.
- numpy.expand_dims to insert new axes of length one.
- numpy.squeeze (the opposite) to remove new axes of length one.

A NumPy array has a `dtype` property that gives the datatype. If the array was created from data, this will be inferred

```
my_array.dtype
```

```
dtype('int64')
```

Functions that construct arrays also have an optional argument to specify the datatype

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

```
dtype('float64')
```

4.3 Mathematical operations with arrays

Now here comes the payoff. On lists, multiplication by an integer concatentates multiple copies

```
2 * [1, 2, 3]
```

```
[1, 2, 3, 1, 2, 3]
```

which is sometimes useful. But in numerical applications what we really want is this

```
2 * np.array([1, 2, 3])
```

```
array([2, 4, 6])
```

This illustrates a general feature of NumPy that **all mathematical operations are performed elementwise on arrays!**

```
print(np.array([1, 2, 3]) + np.array([4, 5, 6]))
print(np.array([1, 2, 3])**2)
print(np.sqrt(np.array([1, 2, 3])))
```

```
[5 7 9]
[1 4 9]
[1.         1.41421356 1.73205081]
```

This avoids the need to write nested loops to perform some operation on each element of some multidimensional data. Of course, the loops are still there, it's just that NumPy handles them in highly optimized C rather than Python. Code which operates in this way – rather than with explicit loops – is often described as *vectorized*, and in NumPy-speak vectorized functions are called *ufuncs*, short for *universal functions* (you can write your own if you need to). As a basic principle you should *never* use a Python loop to access your data in NumPy code. Loops may appear at a high level in stepping through time steps in a simulation, for example.

4.3.1 Broadcasting

Vectorization is even more versatile than the above examples might suggest. *Broadcasting* is a powerful protocol that allows us to combine arrays of different shapes. Thus we can add a number to an array

```
np.array([1, 2, 3]) + 2.3
```

```
array([3.3, 4.3, 5.3])
```

More generally, elementwise operations can be performed on two arrays of the same rank if in each dimension the sizes either match or one array has size 1.

```
# These have shape (2, 3) and (1, 3)
np.array([[1, 2, 3], [4, 5, 6]]) + np.array([[4, 3, 2]])
```

```
array([[5, 5, 5],
       [8, 8, 8]])
```

In fact, we can simplify this last example

```
# These have shape (2, 3) and (3,)
np.array([[1, 2, 3], [4, 5, 6]]) + np.array([4, 3, 2])
```

```
array([[5, 5, 5],
       [8, 8, 8]])
```

Broadcasting two arrays follows these rules:

1. If the arrays do not have the same rank, prepend the shape of the lower rank array with 1s until both shapes have the same length.
2. The two arrays are said to be compatible in a dimension if they have the same size in the dimension, or if one of the arrays has size 1 in that dimension.
3. The arrays can be broadcast together if they are compatible in all dimensions. After broadcasting, each array behaves as if it had shape equal to the elementwise maximum of shapes of the two input arrays.
4. In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension.

The documentation has more detail.

4.3.2 Example: playing with images

Nice example of a 2D array?

TODO

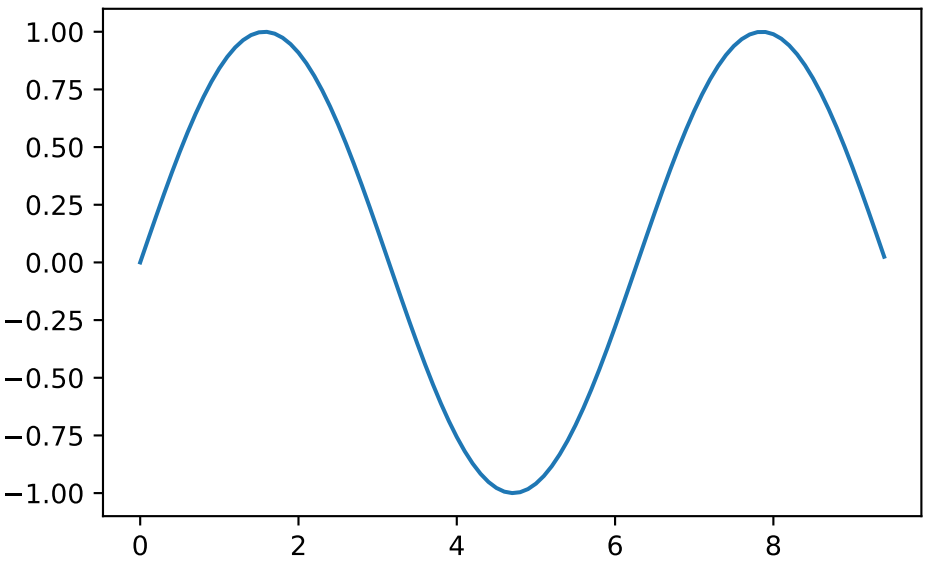
4.4 Plotting with Matplotlib

There are various specialized Python plotting libraries but the entry-level option is the catchily named Matplotlib. The `pyplot` module provides a plotting system that is similar to MATLAB (I'm told)

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

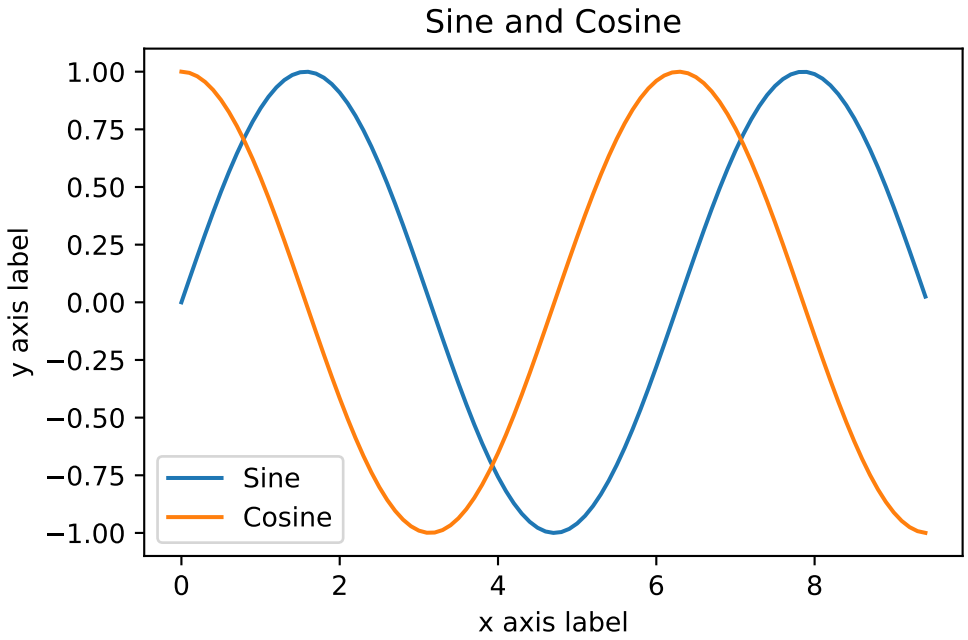
Here's a simple example of the `plot` function, used to plot 2D data

```
# Compute the x and y coordinates for points on a sine curve
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)

# Plot the points using matplotlib
plt.plot(x, y)
plt.show()
```



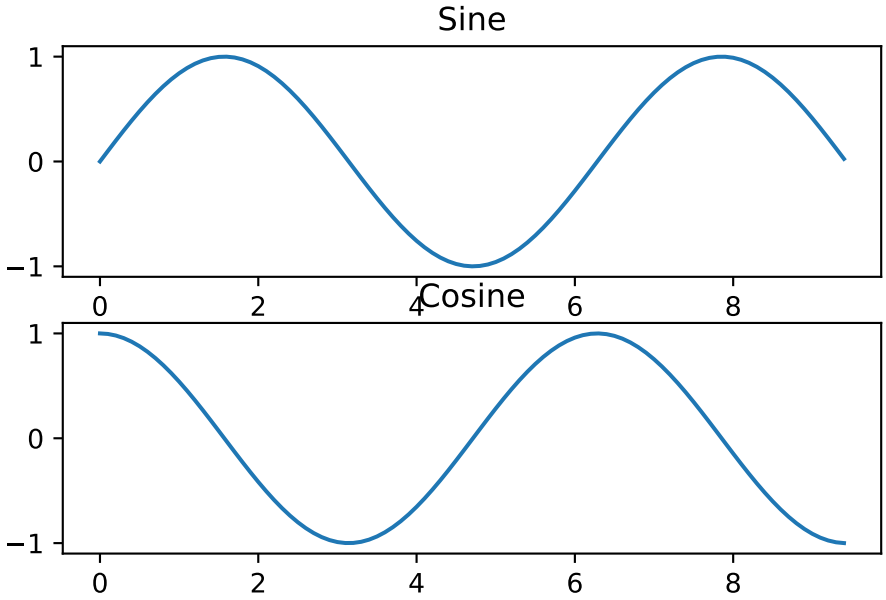
Note: you must call plt.show() to make graphics appear. Here's a fancier example with some labelling

```
# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Plot the points using matplotlib
plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
plt.show()
```



Often you'll want to make several related plots and present them together, which can be achieved using the `subplot` function

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

# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Set up a subplot grid that has height 2 and width 1,
# and set the first such subplot as active.
plt.subplot(2, 1, 1)

# Make the first plot
plt.plot(x, y_sin)
plt.title('Sine')

# Set the second subplot as active, and make the second plot.
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')

# Show the figure.
plt.show()
```



4.5 Saving and loading data

In the course of your work you are likely to produce, as well as consume lots of data. While it's good practice to keep notebooks capable of reproducing any of your analyses, this could be time consuming and resource heavy for larger computations. Thus at some point you'll probably want to save and load data. For example, after saving the data of a large scale simulation you'd like to load it and perform some analysis.

NumPy comes with its own save and load functions and associated binary format `.npy`. The benefit of using these is that after loading you get back a NumPy array ready to be used.

A related function savez allows several arrays to be saved and then loaded as a dictionary-like object.

5 Floating point and all that

Since physics is all about numbers we had better develop some understanding of how computers represent them, and the limitations of this representation. Hopefully this example is sufficiently motivating:

```
0.1  + 0.2 == 0.3
```

```
False
```

Ah...

5.1 Integers

Let's begin with something simpler

```
1 + 1 == 2
```

```
True
```

which is a bit more reassuring. Integers can be represented in binary

```
3 == 0b11
```

```
True
```

or octal or hexadecimal (with a prefix `0o` or `0h`). You can get the binary string representing an integer using the `bin` function

```
bin(-2)
```

```
'-0b10'
```

Python allows for arbitrarily large integers, so there is no possibility of overflow or rounding error

```
2**100
```

```
1267650600228229401496703205376
```

The only limitation is the memory required to store it.

Numpy integers are a different story

```
import numpy as np
np.int64(2**100)
```

```
OverflowError: Python int too large to convert to C long
```

Since NumPy is using C the types have to play nicely. The range of integers that can be represented with 32 bit `numpy.int32`s is $\approx \pm 2^{31} \approx \pm 2.1Ö10^9$ (one bit is for the sign) and 64 bit `numpy.int64`s is $\approx \pm 2^{63} \approx \pm 9.2Ö10^{18}$. Apart from the risk of overflow when working NumPy's integers there are not other gotchas to worry about.

5.2 Floating point numbers

The reason why $0.1 + 0.2 \neq 0.3$ in Python is that specifying a real number exactly would involve an infinite number of bits, so that any finite representation is necessarily approximate.

The representation computers use for the reals is called floating point arithmetic. It is essentially a form of scientific notation, in which a significand (it contains the significant figures) is multiplied by an *exponent*. The name *floating point* reflects the fact that the number of digits after the decimal point is not fixed (I'm using the base ten terms for convenience)

This representation requires the choice of a base, and Python's floating point numbers use binary. Numbers with finite binary representations therefore behave nicely

```
0.125 + 0.25 == 0.375
```

```
True
```

For decimal numbers to be represented exactly we'd have to use base ten. This can be achieved with the `decimal` module. Our $0.1 + 0.2$ example then works as expected

```
from decimal import *
Decimal('0.1') + Decimal('0.2')
```

```
Decimal('0.3')
```

Since there is nothing to single out the decimal representation in physics (as opposed to, say, finance) we won't have any need for it.

A specification for floating point numbers must give

1. A base (or *radix*) b
2. A precision p, the number of digits in the significand c. Thus $0 \leq c \leq b^p - 1$.
3. A range of exponents q specifed by emin and emax with $\text{emin} \leq q + p - 1 \leq \text{emax}$.

Including one bit s for the overall sign, a number then has the form $(-1)^s \times c \times b^q$. The smallest positive nonzero number that can be represented is therefore $b^{1+\text{emin}-p}$ (corresponding to the smallest value of the exponent) and the largest is $b^{1+\text{emax}} - 1$.

The above representation isn't unique: for some numbers you could make the significand smaller and the exponent bigger. A unique representation is fixed by choosing the exponent to be as small as possible.

Representing numbers smaller than b^{emin} involves a loss of precision, as the number of digits in the significand falls below p and the exponent has taken its minimum value . These are called subnormal numbers. For binary floats, if we stick with the normal numbers and a p-bit significand the leading bit will be 1 and so can be dropped from the representation, which then only requires $p - 1$ bits.

The specification for the floating point numbers used by Python (and many other languages) is contained in the IEEE Standard for Floating Point Arithmetic IEEE 754. The default Python `float` uses the 64 bit *binary64* representation (often called *double precision*). Here's how those 64 bits are used

- $p = 53$ for the significand, encoded in 52 bits
- 11 bits for the exponent
- 1 bit for the sign

Another common representation is the 32 bit *binary32* (*single precision*) with

- $p = 24$ for the significand, encoded in 23 bits
- 8 bits for the exponent
- 1 bit for the sign

5.2.1 Floating point numbers in NumPy

If this all a bit theoretical you can just get NumPy's finfo function to tell all about the machine precision

```
np.finfo(np.float64)
```

```
finfo(resolution=1e-15, min=-1.7976931348623157e+308, max=1.7976931348623157e+308, dtype=flo
```

Note that $2^{-52} = 2.22 \times 10^{-16}$ which accounts for the value 10^{-15} of the resolution. This can be checked by finding when a number is close enough to treated as 1.0.

```
x=1.0
while 1.0 + x != 1.0:
    x /= 1.01
print(x)
```

```
1.099427563084686e-16
```

For binary32 we have a resolution of 10^{-6}.

```
np.finfo(np.float32)
```

```
finfo(resolution=1e-06, min=-3.4028235e+38, max=3.4028235e+38, dtype=float32)
```

One lesson from this is that taking small differences between numbers is a potential source of rounding error, as in this somewhat mean exam question

> A3 The Apollo 11 spacecraft took 76 hours to travel from the Earth to the Moon, a distance of 384,400 km. Estimate the difference between the Earth-Moon distance as measured in the rest frame of the spacecraft during the transit and the Earth-Moon distance as measured in the Earth's rest frame. You may assume the spacecraft moves with constant velocity and you may ignore the Moon's motion relative to the Earth.

Solution

Solution: $x - x' = x(1 - \gamma^{-1}) \sim x\beta^2/2 \sim 4.2\text{mm}$.

```
import numpy as np
from scipy.constants import c
beta = 384400e3 / (76 * 3600) / c
gamma = 1/np.sqrt(1 - beta**2)
print(1 - np.float32(1/gamma), 1 - np.float64(1/gamma))
```

```
0.0 1.0981660025777273e-11
```

5.2.2 The dreaded NaN

As well as a floating point system, IEEE 754 defines Infinity and NaN (Not a Number)

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

```
/var/folders/xs/y8sn45v943s2_62flnxw0p940000gn/T/ipykernel_33246/2604490398.py:1: RuntimeWarn

divide by zero encountered in true_divide

/var/folders/xs/y8sn45v943s2_62flnxw0p940000gn/T/ipykernel_33246/2604490398.py:1: RuntimeWarn

invalid value encountered in true_divide

array([ inf, -inf,  nan])
```

They behave as you might guess

```
2 * np.inf, 0 * np.inf, np.inf > np.nan
```

```
(inf, nan, False)
```

NaNs propagate through subsequent operations

```
2 * np.nan
```

```
nan
```

which means that if you get a NaN somewhere in your calculation, you'll probably end up seeing it somewhere in the output (which is the idea).

6 Solving differential equations with SciPy

> Newton's fundamental discovery, the one which he considered necessary to keep secret and published only in the form of an anagram, consists of the following: *Data aequatione quotcunque fluentes quantitates involvente, fluxiones invenire; et vice versa*. In contemporary mathematical language, this means: "It is useful to solve differential equations".
>
> Vladimir Arnold, *Geometrical Methods in the Theory of Ordinary Differential Equations*

While Arnold (and Newton) are of course right the problem is that solving differential equations is *not possible in general*. Even the simplest example of a first order ordinary differential equation (ODE) in a single variable

$$\frac{dx}{dt} = f(x,t) \tag{6.1}$$

cannot be solved for general $f(x,t)$ [1]. Of course, formulating a physical (or whatever) system in terms of differential equations represents a nontrivial step on the road to understanding it, but a lot remains to be done.

Numerical analysis of differential equations is a colossal topic in applied maths and we are barely going to scratch the surface. The important thing is to be able to access existing solvers (and implement your own if necessary) and crucially to *understand their limitations*.

6.1 Euler's method

The basic idea behind all ODE solvers is to introduce a discretization of the equation and its solution $x_j \equiv x(t_j)$ at time points $t_j = hj$ for some step size h and $j = 0, 1, \ldots$. The very simplest approach is called Euler's method [2] and approximates the derivative on the right hand side of Equation **??** as

[1] Sticking with integers

[2] I've borrowed this example from Garth Wells' course

$$\left.\frac{dx}{dt}\right|_{t=t_j} \approx \frac{x_{j+1} - x_j}{h}. \tag{6.2}$$

Rearranging the ODE then gives the update rule

$$x_{j+1} = x_j + hf(x_j, t_j). \tag{6.3}$$

Once an *initial condition* x_0 is specified, subsequent values can be obtained by iteration.

Notice that Equation **??** involved a forward finite difference: the derivative at time t_j was approximated in terms of x_j and x_{j+1} (i.e. one step *forward* in time). Why do this? So that the update rule Equation **??** is an *explicit* formula for x_{j+1} in terms of x_j. This is called an *explicit method*. If we had used the backward derivative we would end up with backward Euler method

$$x_{j+1} = x_j + hf(x_{j+1}, t_{j+1}) \tag{6.4}$$

which is *implicit*. This means that the update requires an additional step to numerically solve for x_{j+1}. Although this is more costly, there are benefits to the backward method associated with stability.

6.1.1 Truncation error

In making the approximation Equation **??** we make an $O(h^2)$ *local truncation error*. To integrate for a fixed time the number of steps required is proportional to h^{-1}, which means that the worst case error at fixed time (the *global truncation error*) is $O(h)$. For this reason Euler's method is called *first order*. More sophisticated methods are typically higher order: the SciPy function scipy.integrate.solve_ivp uses a fifth order method by default.

TODO discuss midpoint method?

6.1.2 Rounding error

If you had unlimited computer time you might think you could make the step size h ever smaller in order to make the updates more accurate. This ignores the machine precision ϵ, discussed in Section **??**. The rounding error is roughly ϵx_j, and if the $N \propto h^{-1}$ errors in successive steps can be treated as independent random variables, the relative total rounding error will be $\propto \sqrt{N}\epsilon = \frac{\epsilon}{\sqrt{h}}$ and will dominate for h small.

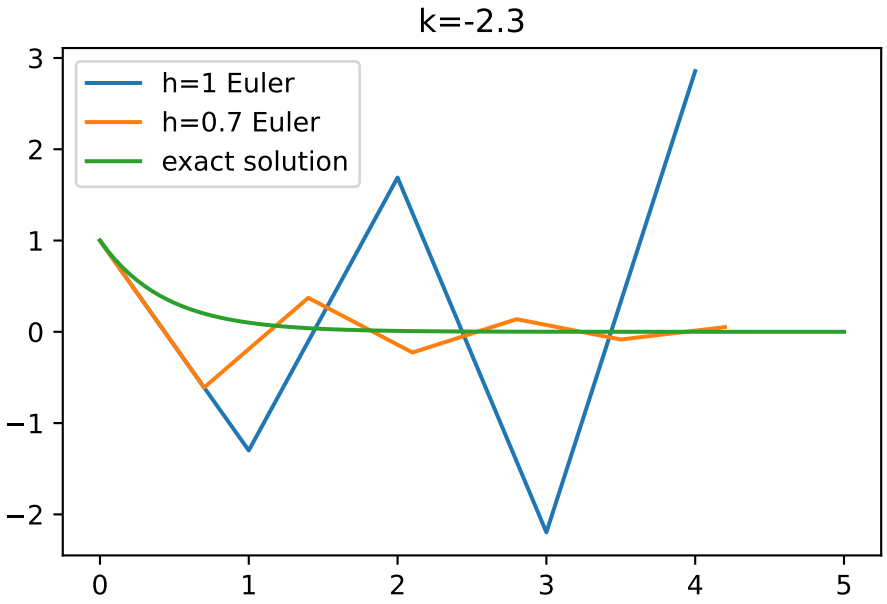
6.1.3 Stability

Apart from the relatively low accuracy that comes from using a first order method, the Euler method may additionally be unstable, depending on the equation. This can be demonstrated for the linear equation

$$\frac{dx}{dt} = kx$$

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

def euler(h, t_max, k=1):
    """
    Solve the equation x' = k x, with x(0) = 1 using
    the Euler method.

    Integrate from t=0 to t=t_max using stepsize h for
    num_steps = t_max / h.

    Returns two arrays of length num_steps: t, the time coordinate, and x_0, the position.
    """
    num_steps = int(t_max / h)
    # Allocate return arrays
    x = np.zeros(num_steps, dtype=np.float32)
    t = np.zeros(num_steps, dtype=np.float32)
    x[0] = 1.0  # Initial condition
    for i in range(num_steps - 1):
        x[i+1] = x[i] + k * x[i] * h
        t[i+1] = t[i] + h  # Time step
    return t, x

k = -2.3
t_max = 5
t, x = euler(1, t_max, k)
plt.plot(t, x, label="h=1 Euler")
t, x = euler(0.7, t_max, k)
plt.plot(t, x, label="h=0.7 Euler")
t = np.linspace(0, t_max, 100)
plt.plot(t, np.exp(k * t), label="exact solution")
plt.title("k=-2.3")
plt.legend()
```

```
plt.show()
```



For a linear equation the Euler update Equation **??** is a simple rescaling

$$x_{j+1} = x_j(1 + hk)$$

so the region of stability is $|1 + hk| \leq 1$. You can check that the backward Euler method Equation **??** eliminates the instability for $k < 0$.

6.2 Using SciPy

Coming up with integration schemes is best left to the professionals. Your first port of call for solving ODEs in Python should probably be the integrate module of the SciPy scientific computing library. The function scipy.integrate.solve_ivp provides a versatile API.

One important thing to understand is that all these integration schemes apply to systems of *first order* differential equations. Higher order equations can always be presented as a first order system, at the expense of introducing more equations. For example, in physics we are often concerned with Newton's equation

$$m\frac{d^2\mathbf{x}}{dt^2} = \mathbf{f}(\mathbf{x}, t),$$

which is three second order equations. We turn this into a first order system by introducing the velocity $\mathbf{v} = \dot{\mathbf{x}}$, giving the six equations

$$\begin{aligned}\frac{d\mathbf{x}}{dt} &= \mathbf{v} \\ m\frac{d\mathbf{v}}{dt} &= \mathbf{f}(\mathbf{x}, t).\end{aligned}$$

As a simple example, let's consider the pendulum equation

$$\ddot{\theta} = -\sin\theta$$

which can be cast as

$$\begin{aligned}\dot{\theta} &= l \\ \dot{l} &= -\sin\theta\end{aligned}$$

Solving the equation using SciPy just requires us to define a function giving the right hand side of these equations

```
def pendulum(t, y): return [y[1], -np.sin(y[0])]
# The pendulum equation: y[0] is theta and y[1] is l
```

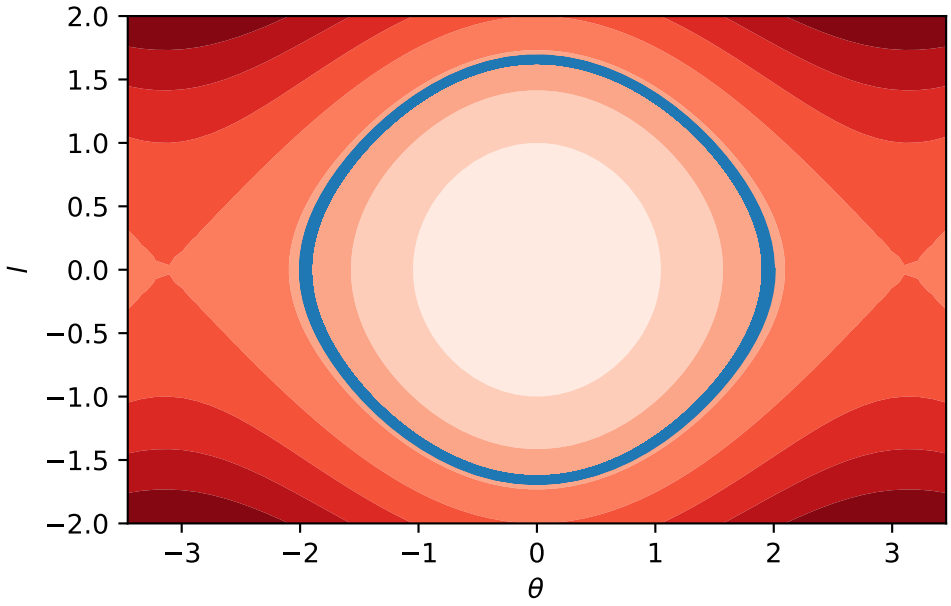
and then calling `solve_ivp`

```
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

t_max = 1000
pendulum_motion = solve_ivp(pendulum, [0, t_max], [2, 0], dense_output=True)
```

The option `dense_output=True` is used to specify that a continuous solution should be found. What this means in practice is that the returned object `pendulum_motion` has a `sol` property that is an instance of OdeSolution. `sol(t)` returns the computed solution at t (this involves interpolation). We can use this to plot the pendulum's trajectory in the $\theta - l$ phase plane, along with the contours of the conserved energy function

$$E(\theta, l) = \frac{1}{2}l^2 - \cos\theta$$

```
fig, ax = plt.subplots()

theta = np.linspace(-1.1 * np.pi, 1.1 * np.pi, 60)
l = np.linspace(-2, 2, 60)
E = -np.cos(theta[np.newaxis,:]) + (l[:,np.newaxis])**2 / 2
# Note the use of broadcasting to obtain the energy as a function of the phase space coord

xx, yy = np.meshgrid(theta, l)

ax.contourf(xx, yy, E, cmap='Reds')
t = np.linspace(0, t_max, 10000)
ax.plot(*pendulum_motion.sol(t))
plt.xlabel(r'$\theta$')
plt.ylabel(r'$l$')
plt.show()
```



The thickness of the blue line is due to the variation of the energy over the $t = 1000$ trajectory (measured in units where the frequency of linear oscillation is 2π). Notice that we did not have to specify a time step: this is determined *adaptively* by the solver to keep the estimate of the local error below `atol + rtol * abs(y)`, where `atol` and `rtol` are optional arguments that correspond to the absolute and relative tolerances, with default values of 10^{-6} and 10^{-3} respectively. The global error is of course much larger. In general, monitoring conserved quantities is a good experimental method for assessing the accuracy of integration.

The alternative to `dense_output=True` is to track "events", which are user-defined points of interest on the trajectory. We supply `solve_ivp` with functions `event(t, x)` whose zeros define the events. We can use events to take a "cross section" of higher dimensional motion. As an example let's consider the Hénon–Heiles system, a model chaotic system with origins in stellar dynamics

$$\begin{aligned}\dot{x} &= p_x \\ \dot{p}_x &= -x - 2\lambda xy \\ \dot{y} &= p_y \\ \dot{p}_y &= -y - \lambda(x^2 - y^2).\end{aligned}$$

These coupled first order systems for the N coordinates and N momenta of a mechanical system with N degrees of freedom are an example of Hamilton's equations. The phase space is now four dimensional and impossible to visualize.

The conserved energy is

$$E = \frac{1}{2}\left(p_x^2 + p_y^2 + x^2 + y^2\right) + \lambda\left(x^2 y - \frac{1}{3}y^3\right)$$

If we take a Poincaré section with $x = 0$ a system with energy E must lie within the curve defined by

$$E = \frac{1}{2}\left(p_y^2 + y^2\right) - \frac{\lambda}{3}y^3.$$

Starting from $x = 0$ we can generate a section of given E by solving for p_x

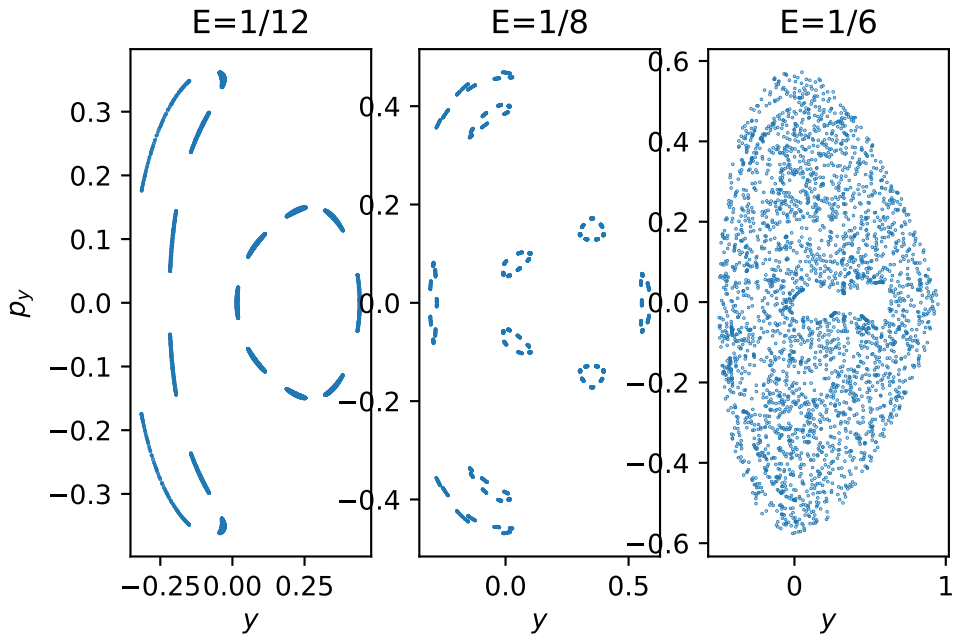
$$p_x = \sqrt{2E - y^2 - p_y^2 + \frac{2\lambda}{3}y^3}$$

```
def henon_heiles(t, z, ):
    x, px, y, py = z
    return [px, -x - 2 *  * x * y, py, -y -  * (x**2 - y**2)]

def px(E, y, py, ):
    return np.sqrt(2 * E - y**2 - py**2 + 2 *  * y**3 / 3)

def section(t, y, ): return y[0] # The section with x=0

t_max = 10000
 = 1
```

```
hh_motion = []
for E in [1/12, 1/8, 1/6]:
    hh_motion.append(solve_ivp(henon_heiles, [0, t_max], [0, px(E, 0.1, 0.1, ), 0.1, 0.1],
```

We can then plot a section of the phase space with increasing energy, showing the transition from regular to chaotic dynamics.

```
fig, ax = plt.subplots(1, 3)
energies = ["1/12", "1/8", "1/6"]
for idx, data in enumerate(hh_motion):
        ax[idx].scatter(*data.y_events[0][:, 2:].T, s=0.1)
        ax[idx].title.set_text(f"E={energies[idx]}")
        ax[idx].set_xlabel(r'$y$')

ax[0].set_ylabel(r'$p_y$')
plt.show()
```



TODO Leapfrog?

Nice demo on Poincaré sections

Symplectic integrator see e.g.

Look at leapfrog?

https://github.com/scipy/scipy/issues/12690

Problem is that it's hard to do in scipy

https://stackoverflow.com/questions/60338471/lyapunov-spectrum-for-known-odes-python-3

7 Monte Carlo methods

Many physical phenomena, notably those falling within the domains of statistical mechanics and quantum theory, depend in an essential way on *randomness*. The simulation of these phenomena therefore requires algorithms that incorporate random (or pseudo-random) elements in the most efficient way.

7.1 Sampling from a distribution

Let's suppose that we have a source of samples of a real valued random variable X that follows a particular probability density function p_X [1] (more about where they might come from in Section **??**). This means that the probability of drawing a sample in the region $[x, x + dx]$ is $p_X(x)dx$. If we now map the samples using a function f, what is the probability density p_Y of $y = f(x)$? The new probability density is defined in just the same way: the probability of y lying in the region $[y, y + dy]$ is $p_Y(y)dy$. Since x is being mapped deterministically to y these two probabilities are therefore the same

$$p_X(x)dx = p_Y(y)dy$$

or

$$p_Y(y) = p_X(x)\left|\frac{dx}{dy}\right| = \frac{p_X(x)}{|f'(x)|}, \qquad x = f^{-1}(y)$$

This formula shows that we can create samples from an arbitrary probability distribution by choosing an invertible map f appropriately. If p_X is a standard uniform distribution on $[0, 1]$ then $f(x)$ is the inverse of the cummulative probability distribution of Y i.e.

$$f^{-1}(y) = \int_{-\infty}^{y} p_Y(y')dy'$$

The same approach works in higher dimensions: $\left|\frac{dx}{dy}\right|$ is replaced by the inverse of the Jacobian determinant.

[1] Sticking with integers

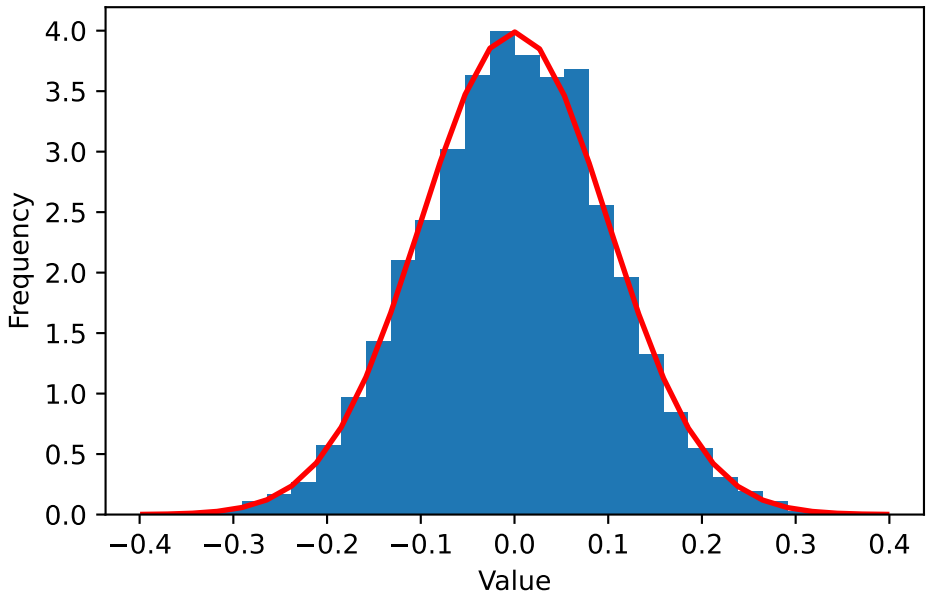
The Box–Muller transform is one example of this idea. Take two independent samples from a standard uniform distribution $u_{1,2}$ and form

$$x = \sqrt{-2 \log u_1} \cos(2\pi u_2)$$
$$y = \sqrt{-2 \log u_1} \sin(2\pi u_2).$$

x and y are independent samples from a standard normal distribution.

Various functions are available in the `numpy.random` module to generate random arrays drawn from a variety of distributions. Box–Muller has now been retired in favour of the Ziggurat algorithm.

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

mu, sigma = 0, 0.1 # mean and standard deviation
s = random.normal(mu, sigma, size=10000)
count, bins, ignored = plt.hist(s, 30, density=True)
plt.plot(bins, 1/(sigma * np.sqrt(2 * np.pi)) *
               np.exp( - (bins - mu)**2 / (2 * sigma**2) ),
         linewidth=2, color='r')
plt.xlabel("Value")
plt.ylabel("Frequency")
plt.show()
```



For complex multivariate (i.e. high dimensional) distributions there is no general recipe to construct an appropriate f. One very recent application of these ideas is in machine learning models called normalizing flows that use a mapping f parameterized by a neural network. The workhorse for sampling from complicated distributions is Markov chain Monte Carlo, as we discuss in Section **??**.

7.2 The Monte Carlo method

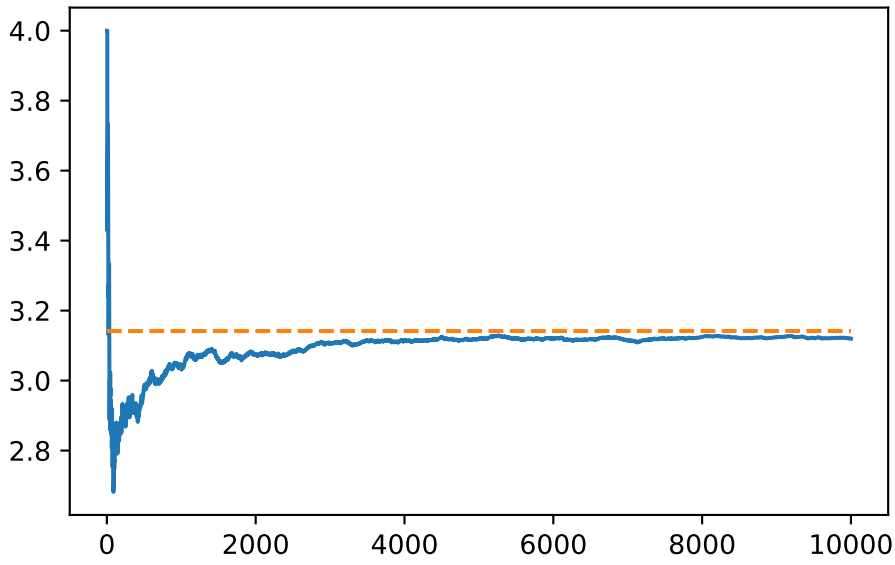
Monte Carlo is the general prefix applied to variety of numerical methods that use randomness in some way. Two of the main classes of problem encountered in physics that come under this heading are:

1. Interpret a numerical evaluation as an expectation value of some random variable and use sampling to estimate it. Monte Carlo integration is an example of this idea.
2. Sampling from a complex probability distribution (which may include taking expectation values). Example: Markov chain Monte Carlo.

7.2.1 Monte Carlo integration

The technique is exemplified by the following fairly dumb way of estimating π

```
max_samples = 10000
inside = 0
areas = []
for sample in range(1, max_samples + 1):
    x = random.uniform(-1, 1)
    y = random.uniform(-1, 1)

    if x ** 2 + y ** 2 <= 1:
        inside += 1
    areas.append(4 * inside / sample)

plt.plot(np.arange(1, max_samples + 1), areas)
plt.plot(np.arange(1, max_samples + 1), np.pi * np.ones(max_samples), linestyle='dashed')
plt.show()
```



In terms of integration, you can think of this as a way to compute the integral of a function which is one inside the unit disc, and zero outside it.

Although it's a silly method, this does illustrate one important feature of Monte Carlo methods in general: that the relative error with N samples is typically $\propto N^{-1/2}$ (thus at the 1% level for 10^4 samples) because the variance of a sum of N iid variables is $\propto N^{1/2}$.

TODO General setting. Importance sampling

Monte Carlo integration comes into its own for high dimensional problems. For low dimensional integrals the quadrature methods in `scipy.integrate` are preferable.

7.2.2 Markov chain Monte Carlo

Suppose you want to generate configurations at random (i.e. with a uniform distribution) from a "gas" of hard disks [2].

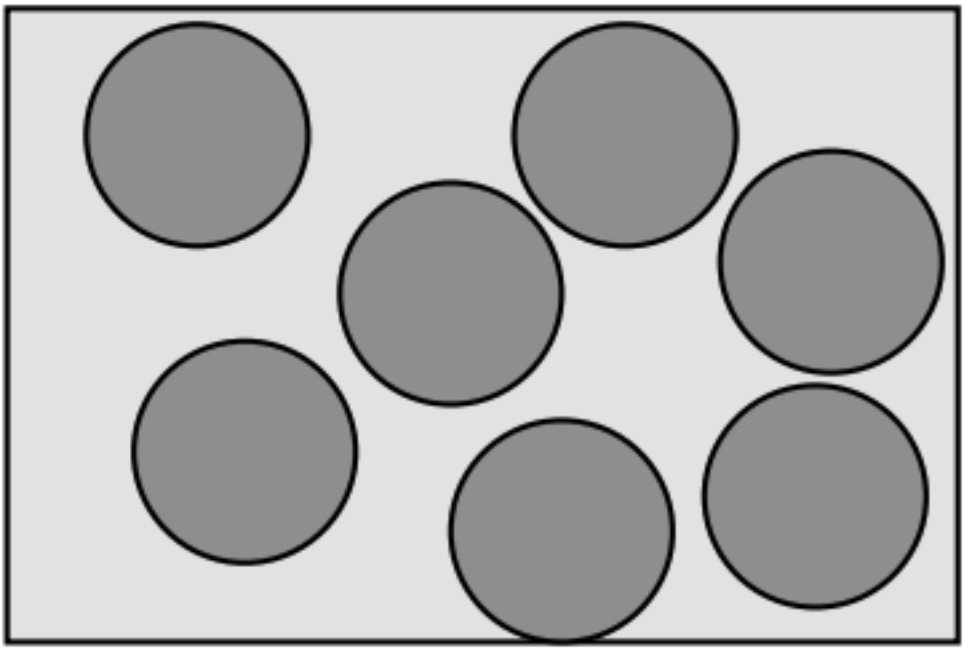


Figure 7.1: Coins in a shoe box (gas of hard disks). From Krauth (1998)

It's harder than it looks! The first guess you might have is to start adding coins at random, and if you get an overlap, try again until you don't. Obviously this will become inefficient as the box fills up, and most attempts fail. *Worse, it doesn't in fact yield a uniform distribution!*

TODO Why not? See Widom (1966) for an explanation

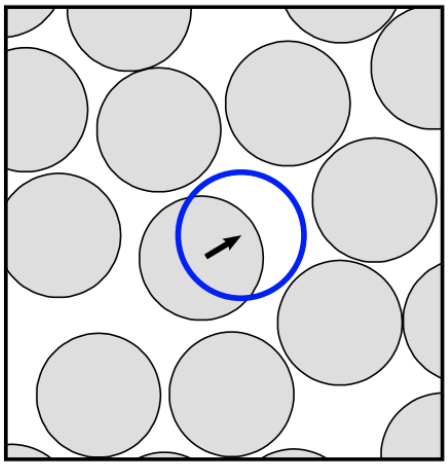
Here's an approach that works:

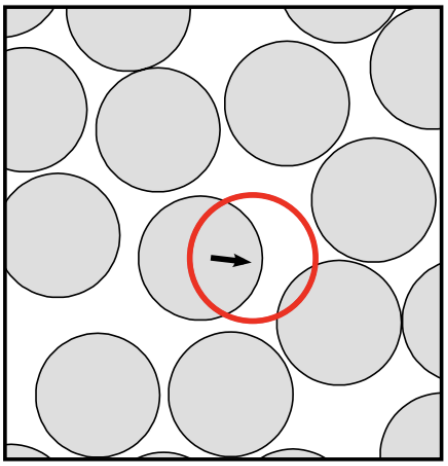
Example 7.1 (Metropolis algorithm for hard disks)**.**

1. Fix the number of disks and an initial configuration (some regular lattice configuration, say).
2. Pick a disk at random and attempt (or *propose*) to move it by a small random amount (i.e. random direction; random small magnitude).

[2] I've borrowed this example from Garth Wells' course

3. If this results in the moved disk intersecting another, *reject* the move, leaving the disk where it is. Otherwise, *accept* the move.
4. Repeat 2. and 3. many times.





accepted move rejected move

Metropolis algorithm

TODO js demo

This is the simplest example of the Metropolis–Hastings algorithm, the first Markov chain Monte Carlo (MCMC) algorithm.

More generally, the goal of MCMC is to come up with a sequential random process (a **Markov chain**) that generates (usually after many steps) a sample from a particular distribution.

You've all heard of a random walk, perhaps as a model for diffusion. At each step you make a move in a random direction, independently of your earlier moves. After many steps these random moves gives rise to a distribution of possible locations. A random walk is the simplest example of a Markov chain.

More generally, a Markov chain is a sequence of random variables X_n with each having a distribution that is is conditional on the value of the previous one, and so is defined in terms of **transition probabilities** $p(X_n = x_n | X_{n-1} = x_{n-1})$ (hence they form a "chain"). I'm going to immediately drop this cumbersome notation in favour of $p(x_n|x_{n-1})$, a function of x_n and x_{n-1}, but in general the function giving the transition probabilities can be different at each step (the random variables could all be different).

The probability of a particular sequence $X_1 = x_1 \dots X_n = x_n$ is therefore

$$p(x_n|x_{n-1})p(x_{n-1}|x_{n-2}) \cdots p(x_2|x_1)p^{(1)}(x_1)$$

X_1 has no "parent" so is not conditional on any other value.

Suppose we don't care about the earlier values and just want to know the **marginal distribution** $p^{(n)}(x_n)$ of the final variable. For a random walk this is easy, as x_n typically represents a displacement that is a sum of iid increments. In general this is not the case, however, as the marginal distribution is

$$p^{(n)}(x_n) = \sum_{x_{n-1},\dots x_1} p(x_n|x_{n-1})p(x_{n-1}|x_{n-2}) \cdots p(x_2|x_1)p^{(1)}(x_1)$$

(I'm writing all these expressions for discrete random variables, but the continuous version involving probability density functions is straightforward)

The sums are over all possible values that the random variables might take in the **state space** of the problem. These could be finite or infinite in number.

Things are not as bad as they appear, however, as the marginal distribution can be interpreted as the result of acting $n-1$ times on the vector of values of $p^{(1)}_j \equiv p^{(1)}(j)$ with the **transition matrix** with elements $\mathsf{P}_{jk} = p(j|k)$

$$\mathbf{p}^{(n)} = \mathsf{P}^{n-1}\mathbf{p}^{(1)}.$$

In a single step the marginal probabilities are updated as

$$\mathbf{p}^{(n)} = \mathsf{P}^{n}\mathbf{p}^{(n-1)}.$$

P has some structure. The matrix elements are positive, as they represent probabilities, and each row sums to one

$$\sum_j \mathsf{P}_{jk} = 1.$$

Such matrices are called stochastic.

Although $p^{(n)}$ — the probability distribution at the nth step — changes from step to step, you might expect that after many steps it tends to converge to a **stationary distribution** $p^{(n)} \to \boldsymbol{\pi}$. If it exists, this distribution must satisfy

$$\boldsymbol{\pi} = \mathsf{P}\boldsymbol{\pi}. \tag{7.1}$$

In other words, it is an eigenvector of P with eigenvalue one. This property is guaranteed by the Perron–Frobenius theorem [3].

Thus P determines $\boldsymbol{\pi}$. MCMC turns this idea on its head and asks: if there is some $\boldsymbol{\pi}$ that I would like to generate samples from, can I find a P that has it as a stationary distribution?

There is a trivial answer to this question. Sure, take $\mathsf{P}_{jk} = \boldsymbol{\pi}_j$. That is, jump straight to the stationary distribution no matter what the starting state. But we are interested in highly complicated distributions over large state spaces (think the Boltzmann distribution for a statistical mechanical system comprised of billions of particles). Thus what we really want is to be able to approach such a complicated distribution by making many transitions with *simple* distributions.

One more idea is useful before returning to concrete algorithms. The quantity

$$\mathsf{P}_{jk}\pi_k = p(j|k)\pi_k = p(j,k)$$

is the joint distribution of seeing state k followed by state j in the stationary distribution. A *reversible* Markov chain is one where $p(j,k) = p(k,j)$. Roughly, you can't tell the direction of time because any transition is equally likely to happen forward in time as backward. Random physical processes that respect time reversal symmetry are often modeled as reversible Markov processes.

Combining reversibility with the definition of the stationary state yields the condition of detailed balance

$$\mathsf{P}_{jk}\pi_k = \pi_j \mathsf{P}_{kj}. \tag{7.2}$$

This condition is stronger than the condition Equation **??** for a stationary state. This makes it easier to check: you don't have to do a sum over a state space index. The Metropolis algorithm Example **??** for the hard disk problem satisfies detailed balance for a stationary distribution that is constant when disks don't intersect and zero when they do.

When the stationary distribution $\boldsymbol{\pi}$ has more structure, designing an appropriate transition matrix is harder. The idea is to generalize the hard disk approach by separating the transition into a *proposal* distribution $p_{\text{prop}}(j|k)$ and an *acceptance* distribution $p_{\text{acc}}(a = 0, 1|j \leftarrow k)$ that gives the probability of a move from k to j being accepted ($a = 1$) or rejected ($a = 0$). The probability of moving from k to j is then

[3] Grover's algorithm for search has $O(\sqrt{n})$ complexity, but there's a catch: you need a *quantum computer*, and even then a $\sqrt{n}$ speedup is not going to get you a billion dollars in this economy.

$$p(j|k) = p_{\text{acc}}(a = 1|j \leftarrow k)p_{\text{prop}}(j|k).$$

Substituting this into the detailed balance condition Equation **??** gives

$$\frac{p_{\text{acc}}(a = 1|j \leftarrow k)}{p_{\text{acc}}(a = 1|k \leftarrow j)} = \frac{\pi_j}{\pi_k} \frac{p_{\text{prop}}(k|j)}{p_{\text{prop}}(j|k)}.$$

Any p_{acc} that satisfies this relation for all j and k will do the job. The Metropolis choice is

$$p_{\text{acc}}(a = 1|j \leftarrow k) = \min\left(1, \frac{\pi_j}{\pi_k} \frac{p_{\text{prop}}(k|j)}{p_{\text{prop}}(j|k)}\right). \tag{7.3}$$

This gives an extremely general algorithm, one of the top ten in applied mathematics, according to one list:

Example 7.2 (Metropolis algorithm)**.**

1. Starting from state k sample a next state j from the proposal distribution $p_{\text{prop}}(j|k)$.
2. Accept the proposal with probability $p_{\text{acc}}(a = 1|j \leftarrow k)$ and move to state j. Otherwise reject the proposal and stay in state k.
3. Repeat 1. and 2. many times.

MCMC has the benefit of being embarrassingly parallel. If you want to average something over $\boldsymbol{\pi}$, just run the algorithm many times independently and average the results. This is perfect for parallel computing.

The Metropolis algorithm has an Achilles' heel, however. To perform a move one has to sample from $p_{\text{prop}}(j|k)$ and from $p_{\text{acc}}(a|j \leftarrow k)$. The proposal therefore has to be tractable, like the small shift in position for the hard disk case. This may however, mean that that many of the js suggested correspond to very small π_j, and therefore a very low acceptance probability (c.f. Equation **??**). For example, in the hard disk case at high density many proposed moves will give rise to overlap of disks and be rejected. This means that many steps are required to have one successful update of the simulation. This kind of slowdown is a common feature of MCMC methods applied to complex distributions.

We'll see some more examples of MCMC algorithms for statistical mechanical problems in Section **??**, and ways in which this problem can be avoided.

7.2.3 Relaxation to equilibrium

Eigenvalues

Master equation

Transition matrix

7.3 Statistical mechanics

Statistical mechanics is a natural source of such complex distributions in physics. Remember the fundamental principle that the probability of finding a statistical mechanical system in a microstate $\mathbf{x}$ [4] with energy $\mathcal{E}(\mathbf{x})$ is

$$p(\mathbf{x}) = \frac{\exp\left[-\beta\mathcal{E}(\mathbf{x})\right]}{Z}, \tag{7.4}$$

where Z is a normalizing constant called the partition function and $\beta = 1/k_{\mathrm{B}}T$, where T is the temperature and k_{B} is Boltzmann's constant.

The *central problem* of statistical mechanics is computing ensemble averages of physical quantities, and the *principle difficulty* is the intractability of those averages for large systems. For example, if we are dealing with a classical gas, the configuration space point $\mathbf{x}$ corresponds to the positions of each of the gas molecules $\mathbf{x} = (\mathbf{x}_1, \ldots \mathbf{x}_N)$ and an average is a $3N$-dimensional integral. The only situation in which this integral is tractable is when the gas is noninteracting (ideal), in which case the energy function takes the form

$$\mathcal{E}(\mathbf{x}) = \sum_{n=1}^{N} \mathcal{E}_1(\mathbf{x}_n)$$

where $\mathcal{E}_1(\mathbf{x})$ is the single particle energy. In this case the integral factorizes. As soon as we introduce interactions between particles of the form

$$\mathcal{E}(\mathbf{x}) = \sum_{n<m}^{N} \mathcal{E}_2(\mathbf{x}_n, \mathbf{x}_m)$$

things get a lot harder. The same issue arises in models involving discrete random variables. The canonical example is the Ising model, in which a configuration corresponds to fixing the values of N "spins" $\sigma_n = \pm 1$ with an energy function of the form

[4] For a classical gas of point particles this would correspond to specifying all the positions and velocities, for example.

$$\mathcal{E}(\sigma) = \sum_n h_n \sigma_n + \sum_{m<n} J_{mn} \sigma_m \sigma_n.$$

The two terms correspond to a (magnetic) field that acts on each spin and a coupling between spins. As in the gas, it's the latter that causes problems / interest.

The Ising model comes in a great many flavours according to how the fields and couplings are chosen. They may reflect a lattice structure: $J_{mn} \neq 0$ for nearest neighbours, say, or longer range. They may be fixed or random, defining an ensemble of models.

The most pessimistic assessment is that to calculate an average we are going to have sum over 2^N configurations. Computing the partition function Z that normalizes the average (or which gives the free energy via $F = -k_\mathrm{B} T \log Z$) is another such sum.

Monte Carlo simulation is a much more attractive alternative. MCMC can be used to generate samples from $p(\sigma)$ which are then used to estimate the averages of interest (e.g. average energy $\langle \mathcal{E}(\sigma) \rangle$, average magnetization $\langle \sum_n \sigma_n \rangle$, etc.).

7.3.1 MCMC updates for the Ising model

How does MCMC work in practice for the Ising model? To apply the Metropolis alogorithm Example **??** we can use a simple proposal: pick each spin in turn in some order and try to flip it.

The form of $p(\sigma)$ means that, although we cannot compute the probabilities explicitly, we can calculate *ratios*, which is all we need for Metropolis. For two configurations that differ only by $\sigma_n = \pm 1$ we have

$$\begin{aligned} \frac{p(\sigma_n = 1 | \sigma_{m \neq n})}{p(\sigma_n = -1 | \sigma_{m \neq n})} &= \exp\left[-2\beta \left(h_n + \sum_{m \neq n} J_{mn} \sigma_m \right) \right] \\ &\equiv \exp\left[-\beta \Delta \mathcal{E} \right], \end{aligned}$$

where $\Delta \mathcal{E}$ is the energy difference between two configurations.

One alternative to Metropolis is the **Heat bath algorithm** (or Glauber dynamics or Gibbs sampling) [5]. The idea behind the name is that, since we can calculate the influence of the spin's environment (the "bath"), we can just choose the spin's orientation with the corresponding probabilities. Since there are only two probabilities the ratio is all we need and we get

$$p(\sigma_n = \pm 1 | \sigma_{m \neq n}) = \frac{1}{1 + e^{\pm \beta \Delta \mathcal{E}}}. \tag{7.5}$$

[5] Multiple names are sign that a technique was re-discovered by different communities who don't talk to each other.

The algorithm is then:

Example 7.3 (Heat bath algorithm)**.**

1. Pick a spin n. [6]
2. Compute ΔE, the energy difference between $\sigma_n = \pm 1$.
3. Set $\sigma_n = \pm 1$ with probabilities given by Equation **??**.
4. Repeat 1-3 many times

What happens if we try and come up with more complicated proposals, flipping many spins at once? For Metropolis, the problem is that without a cleverly designed proposal we will be suggesting moves that are likely to be rejected. For the heat bath algorithm, the more spins we flip, the more complicated the evaluation of the corresponding probabilities (2^n outcomes if we flip n spins).

The good news is that we *can* do better — much better — than the above algorithms. The Wolff algorithm is one example. This proposes a cluster of spins of the same orientation to be flipped by adding adjacent spins to an initially random chosen spin with probability p_{add}. It turns out that for the nearest neighbour Ising model with Ferromagnetic coupling $J < 0$ the "magic" value $p_{\text{add}} = 1 - e^{2\beta J}$ is *rejection free*: the probability to flip the whole cluster is always one. This makes for an extremely fast algorithm that is not subject to the usual *critical slowing down* at phase transitions.

```
class IsingModel:
    def __init__(self, L):
        self.L = L
        self.spins = np.random.choice(a=[1, -1], size=(L, L))
        stagger = np.empty(self.L, dtype = bool)
        stagger[::2] = True
        stagger[1::2] = False
        self.mask = np.logical_xor(stagger[:, np.newaxis], stagger[np.newaxis, :])

    def gibbs_update(self, beta, sublattice):
        fields = np.roll(self.spins, 1, 0) + np.roll(self.spins, -1, 0) + np.roll(self.spi
        delta_E = 2 * fields
        spin_up_probabilities = 1 / (1 + np.exp(- beta * delta_E))
        new_spins = 2 * (np.random.rand(self.L, self.L) < spin_up_probabilities) - 1
        self.spins = np.choose(np.logical_xor(sublattice, self.mask), [self.spins, new_spi
```

[6] This can be done deterministically (e.g. sequentially or in alternating blocks when the model is defined on a bipartite graph) — which is what is normally called Gibbs sampling — or at random, which corresponds to Glauber dynamics.

```
    def glauber_update(self, beta):
        x, y = np.random.randint(self.L, size=2)
        fields = 0
        for neighbour in [((x + 1) % self.L, y), ((x - 1) % self.L, y), (x, (y + 1) % self
            fields += self.spins[neighbour]
        delta_E = 2 * fields
        spin_up_probability = 1 / (1 + np.exp(- beta * delta_E))
        if np.random.rand() < spin_up_probability:
            self.spins[x, y] = 1
        else:
            self.spins[x, y] = -1

    def wolff_update(self, beta):
        initial_x, initial_y = np.random.randint(self.L, size=2)
        initial_spin = self.spins[initial_x, initial_y]
        cluster = deque([(initial_x, initial_y)])
        add_prob = 1 - np.exp(-2 * beta)

        while len(cluster) != 0:
            x, y = cluster.popleft()
            if self.spins[x, y] == initial_spin:
                self.spins[x, y] *= -1
                for neighbour in (((x + 1) % self.L, y), ((x - 1) % self.L, y), (x, (y + 1
                    if self.spins[neighbour] == initial_spin:
                        if np.random.rand() < add_prob:
                            cluster.append(neighbour)
```

Figure 7.2: Glauber dynamics, Block Gibbs sampling and Wolff updates compared. Change the temperature using the slider. The centre of the slider corresponds to the critical temperature $k_{\mathrm{B}}T = 2|J|/\log(1+\sqrt{2}) \sim 2.269|J|$.

7.4 The universe of Monte Carlo methods

Monte Carlo simulation is a vast field with practitioners and specialists across the natural sciences, engineering, machine learning, and statistics. In this section I'll mention a few important topics to give a taste of what's out there. For much more detail take a look at Krauth (2006) and / or MacKay (2003).

Probably the biggest single issue is: how do you kow when your MCMC simulation has reached the stationary distribution $\boldsymbol{\pi}$? The pragmatic approach is to monitor the averages of interest

(magnetization, say, in the case of the Ising model) over different simulations or over a time interval and see when they stop changing.

We've touched on the issue of the mixing time in a Markov chain.

1. Finite size effects
2. Approach to equilibrium
3. Critical slowing down / loss of ergodicity
4. Bias of estimators. Importance sampling

Exact sampling

Hamiltonian Monte Carlo.

Multispin encoding: 32 or 64 simulations Jacobs and Rebbi (1981)

https://en.wikipedia.org/wiki/Gibbs_sampling

Other updates

A huge topic, see Krauth (2006) for much more

Also Chapter 29 of MacKay (2003)

Comment at the end about typicality

MCMC in Bayesian inference

Relation to Ising models. Community detection. Why not?

https://arxiv.org/pdf/cond-mat/0005264.pdf

Bayesian inference

7.5 Random number generators

Computers are deterministic

This is covered in some detail in the Nature of Computation

This is a subject dealt with already

RNGs in Trebst?

Further reading: refer to Krauth notes or book

Other suggestions from Twitter

https://roomno308.github.io/blog/MCMC.html https://maximilianrohde.com/posts/code-breaking-with-metropolis/

8 Algorithms and computational complexity

8.1 First example: multiplication

How hard is it to multiply numbers? The bigger they are, the harder it is, as you well know. You also know that computers are very good at multiplying, so once you've switched from multiplying numbers yourself to multiplying them on a computer, you may well be tempted to forget about how hard it is. Nevertheless, computers find big numbers harder than small numbers. How much harder?

If you remember how you learnt to multiply numbers at school, it probably went something like this: [1]

Table 8.1: Multiplying two 3 digit numbers

		1	2	3
	×	3	2	1
_	_	1	2	3
_	2	4	6	
3	6	9		
3	9	4	8	3

For n digits we have to perform n^2 single digit multiplications. We then have to add together the n resulting n-digit numbers. This is another n^2 operations. Thus the overall number of operations is proportional to n^2: doubling the number of digits will make the problem four times harder.

Exactly how long this takes to perform in your head or on a computer will depend on many things, such as how long it takes you to multiply two digits, or get the previous values out of memory (or read them of the page), but you can't get away from the basic quadratic scaling law of this algorithm.

[1] Sticking with integers

8.2 Defining complexity

When computer scientists talk about the complexity of a problem, this question of *scaling of the number of steps involved* is what they have in mind. The complexity of any particular task (or calculation) may vary considerably — evaluating 100×100 is considerably easier than the general case, for example — so instead we ask about how a particular *general* algorithm performs on a *class* of tasks. Such a class is what computer scientists mean when they talk about a problem: multiplication of n digit numbers is a *problem*, and any particular pair of n digit numbers represents an *instance* of that problem. What we discussed above is a particular algorithm for multiplication that has *quadratic complexity*, or "$O(n^2)$ complexity" (say "order n squared").

This description only keeps track of how the difficulty scales with the size of the problem. There are various reasons why this level of detail is important:

1. It allows us to gloss over what exactly we mean by a *step*. Are we working in base ten or binary? Looking the digit multiplications up in a table or doing them from scratch?
2. Likewise we don't have to worry about how the algorithm is implemented exactly in software or hardware, what language we use, and so on.
3. Inevitably, we always want to look at harder and harder problems with bigger and bigger n (whatever n means for the problem at hand). If a simulation finishes for a system of length 10 we immediately want to run it again for length 20, and so on. It then becomes important to know whether our code is going to run for twice as long, four times as long, or 2^{10} times as long (exponential scaling).

8.2.1 Best / worst / average

Even when we focus on a problem in the above sense we still have to be careful in defining the complexity of an algorithm. In general we can characterize three complexities: best case, worse case, and average case. To see the difference between these three consider *search*, the very simple problem of finding an item in an (unordered) list of length n. How hard is this? You have to check every item until you find the one you are looking for, so this suggests the complexity is $O(n)$. You could be lucky and get it first try, however, or within the first ten tries. This means the *best case complexity* of search is $O(1)$: it doesn't increase with the size of the problem. The worst thing that could happen is that the sought item is last: the *worst case complexity* is $O(n)$. On average, you'll find your item in the middle of the list on attempt $\sim n/2$, so the *average case complexity* is $O(n/2)$. But this is the same as $O(n)$ (constants don't matter)

Thus for *linear search* we have:

	Complexity
Best case	$O(1)$
Worst case	$O(n)$
Average case	$O(n)$

We can check the average case performance experimentally by using randomly chosen lists: [2]

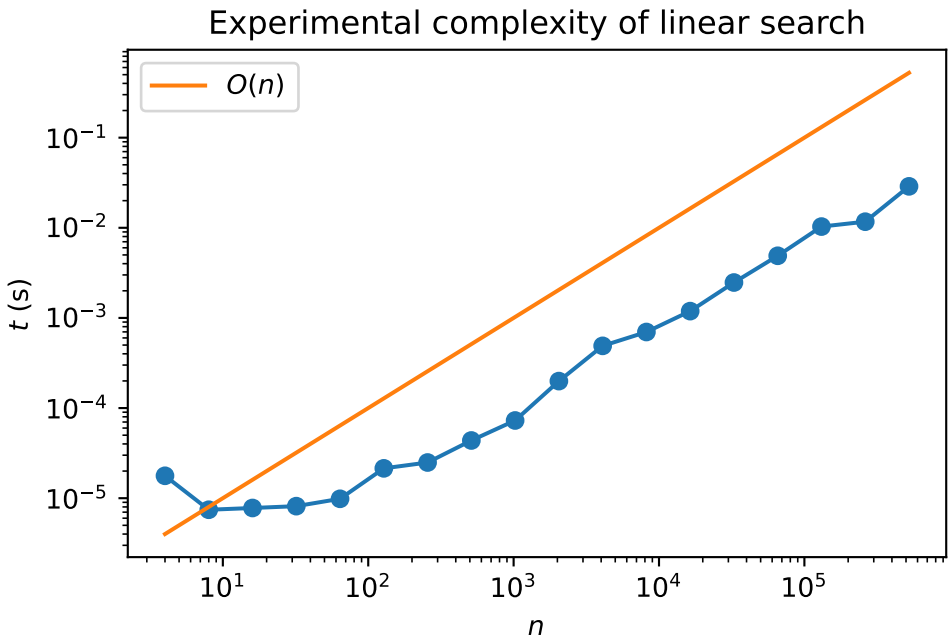
```
def linear_search(x, val):
    "Return True if val is in x, otherwise return False"
    for item in x:
        if item == val:
            return True
    return False
```

```
import numpy as np
# Create array of problem sizes n we want to test (powers of 2)
N = 2**np.arange(2, 20)

# Generate the array of integers for the largest problem to use in plotting times
x = np.arange(N[-1])

# Initialise an empty array to stores times for plotting
times = []

# Time the search for each problem size
for n in N:

    # Time search function (repeating 3 times) to find a random integer in x[:n]
    t = %timeit -q -n4 -r1 -o linear_search(x[:n], np.random.randint(0, n))

    # Store best case time (best on a randomly chosen problem)
    times.append(t.best)
```

```
import matplotlib.pyplot as plt
# Plot and label the time taken for linear search
plt.loglog(N, times, marker='o')
plt.xlabel('$n$')
plt.ylabel('$t$ (s)')
```

[2] I've borrowed this example from Garth Wells' course

```
# Show a reference line of O(n)
plt.loglog(N, 1e-6*N, label='$O(n)$')

# Add legend
plt.legend(loc=0)
plt.title("Experimental complexity of linear search")

plt.show()
```



The "experimental noise" in these plots arises because we don't have full control over exactly what our computer is doing at any moment: there are lots of other processes running. Also, it takes a while to reach the linear regime: there is an overhead associated with starting the program that represents a smaller fraction of the overall run time as n increases.

8.2.2 Better than linear?

It seems obvious that for search you can't do better than linear: you have to look at roughly half the items before you should expect to find the one you're looking for [3]. What if the list is *ordered*? Any order will do: numerical for numbers, or lexicographic for strings. This extra

[3] Grover's algorithm for search has $O(\sqrt{n})$ complexity, but there's a catch: you need a *quantum computer*, and even then a $\sqrt{n}$ speedup is not going to get you a billion dollars in this economy.

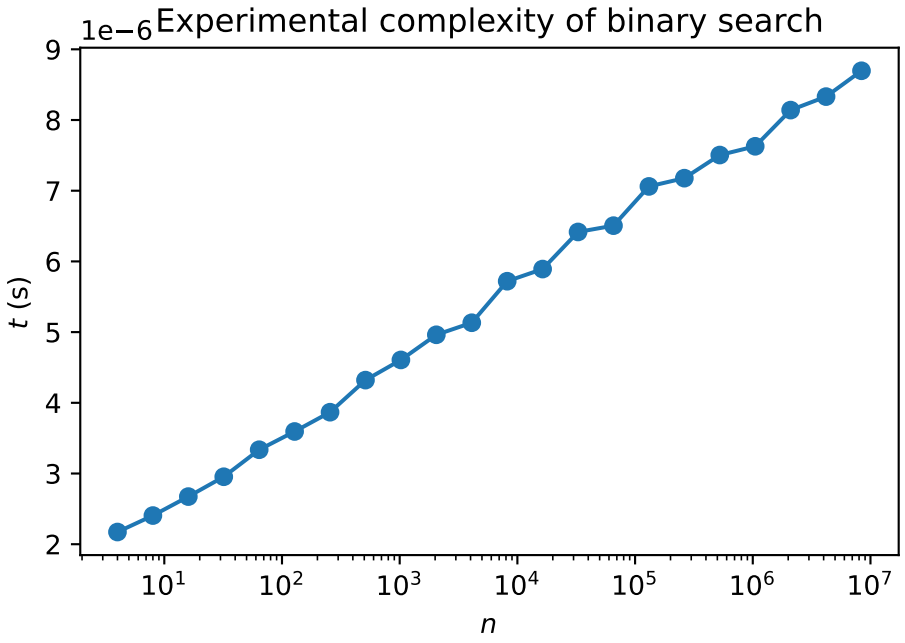
structure allows us to use an algorithm called binary search that you may have seen before. The idea is pretty intuitive: look in the middle of the list and see if the item you seek should be in the top half or bottom half. Take the relevant half and divide it in half again to determine which quarter of the list your item is in, and so on. Here's how it looks in code:

```
def binary_search(x, val):
    """Peform binary search on x to find val. If found returns position, otherwise returns

    # Intialise end point indices
    lower, upper = 0, len(x) - 1

    # If values is outside of interval, return None
    if val < x[lower] or val > x[upper]:
        return None

    # Perform binary search
    while True:

        # Compute midpoint index (integer division)
        midpoint = (upper + lower)//2

        # Check which side of x[midpoint] val lies, and update midpoint accordingly
        if val < x[midpoint]:
            upper = midpoint - 1
        elif val > x[midpoint]:
            lower = midpoint + 1
        elif val == x[midpoint]:  # found, so return
            return midpoint

        # In this case val is not in list (return None)
        if upper < lower:
            return None
```

And here's the performance

```
# Create array of problem sizes we want to test (powers of 2)
N = 2**np.arange(2, 24)

# Creat array and sort
x = np.arange(N[-1])
x = np.sort(x)
```

```
# Initlise an empty array to capture time taken
times = []

# Time search for different problem sizes
for n in N:
    # Time search function for finding '2'
    t = %timeit -q -n5 -r2 -o binary_search(x[:n], 2)

    # Store average
    times.append(t.best)

# Plot and label the time taken for binary search
plt.semilogx(N, times, marker='o')
plt.xlabel('$n$')
plt.ylabel('$t$ (s)')

# Change format on y-axis to scientific notation
plt.ticklabel_format(style='sci', axis='y', scilimits=(0,0))
plt.title("Experimental complexity of binary search")
plt.show()
```



Note the axes: the plot is linear-log, so the straight line indicates *logarithmic* growth of

complexity. This makes sense: if the length is a power of 2 i.e. $n = 2^p$, we are going to need p bisections to locate our value. The complexity is $O(\log n)$ (we don't need to specify the base as overall constants don't matter).

This strategy of getting better than linear scaling

Intro to recursion

8.3 From quadratic to linear

Of course, it is still worth

Example of checking pairs

A major goal in algorithm design

8.4 Examples of different complexities

Summary of asymptotic notation

XKCD plots

8.5 Space vs. time complexity

8.6 Back to multiplication: divide and conquer

Karatsuba algorithm nice story

Karatsuba's account

Karatsuba (1995) contains a nice account of the discovery of his algorithm (online version)

Karatsuba algo

Big O

Visualization of sorting algorithms

Simple example from Leetcode

Analaysis of algorithms

Example of finding a unique item in list

Nice examples from Garth Wells

https://github.com/CambridgeEngineering/PartIA-Computing-Michaelmas/blob/main/11%20Complexity.ipynb

Examples of multiplication

Breadth first and depth first

Importance of choosing a data structure to match algorithm

Examples: queue in Wolff. Was there a Numpy-ish way to do this faster? Priority queue in waiting time algo

FFT uses

https://en.wikipedia.org/wiki/Orthogonal_frequency-division_multiplexing

Needleman-Wunsch

Examples

1. Multiplication Karatsuba
2. Binary search
3. Linear algebra
4. Sorting
5. FFT
6. Taking powers (SICP). What is n in this case?
7. Euclidean algorithm (GCD) (SICP)

References. Nature of computation, grokking algos

Insertion in a list etc.

9 Linear algebra

Krylov subspaces

SVD and quantum mechanics. Quantum entanglement.

Image compression using SVD

http://timbaumann.info/svd-image-compression-demo/

Trebst has nice Einstein example

PCA for big data

10 Divide and Conquer

FFT. Use split step as illustration Matrix multiplication

11 Dynamic Programming

12 Automatic Differentiation

Karpathy's micrograd Michael Nielsen's book

13 Summary

In summary, this book has no content whatsoever.

References

Jacobs, Laurence, and Claudio Rebbi. 1981. "Multi-Spin Coding: A Very Efficient Technique for Monte Carlo Simulations of Spin Systems." *Journal of Computational Physics* 41 (1): 203–10.

Kapfer, Sebastian C, and Werner Krauth. 2013. "Sampling from a Polytope and Hard-Disk Monte Carlo." In *Journal of Physics: Conference Series*, 454:012031. 1. IOP Publishing.

Karatsuba, Anatolii Alexeevich. 1995. "The Complexity of Computations." *Proceedings of the Steklov Institute of Mathematics-Interperiodica Translation* 211: 169–83.

Krauth, Werner. 1998. "Introduction to Monte Carlo Algorithms." In *Advances in Computer Simulation*, 1–35. Springer.

———. 2006. *Statistical Mechanics: Algorithms and Computations*. Vol. 13. OUP Oxford.

MacKay, David JC. 2003. *Information Theory, Inference and Learning Algorithms*. Cambridge university press.

Widom, B. 1966. "Random Sequential Addition of Hard Spheres to a Volume." *The Journal of Chemical Physics* 44 (10): 3888–94.