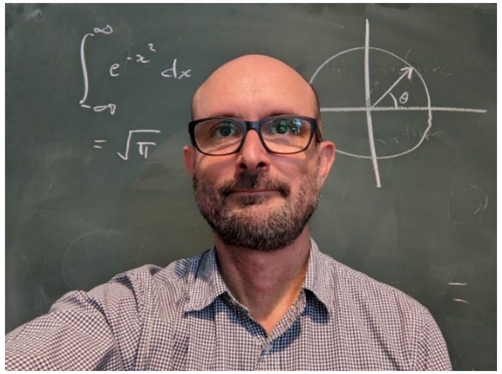
35006 Numerical Methods

Your Subject Coordinator/Lecturer this semester:



Chris Poulton (“Dr Chris, Dr Poulton”, “Chris”, etc etc, just please not “Christopher”)

Chris.Poulton@uts.edu.au

Emails answered by Monday, Wednesday mornings.

Classes and materials:

Workshop (Wrk1): Weekly workshop held online

Computer Labs (Cmp1): 3 hours per week

Assessment

The assessment has the following components:

Labs:	Due each week	10%
Assignments:	Due weeks 3,6,9,12	50%
Final Project:	Due end of Week 12	40%

Each week:

Week 1: Intro to Python; Error, precision and numerical differentiation

Workshop Class

- Blank slides.pdf
- Annotated Slides

Computer Lab this week

- Computer Lab 1.pdf
- Lab 1 solutions and scripts

Preparation for next week

- Preparation Tasks

Other Resources

- Link to Numerical Recipes Textbook
- Additional Reading

Assumed Knowledge and skills:

1. Thorough algebra

2. Calculus from Maths 2 / IMAM / MM2

3. Some previous coding experience
(if not, then please let me know)

What do we mean by Numerical Methods?

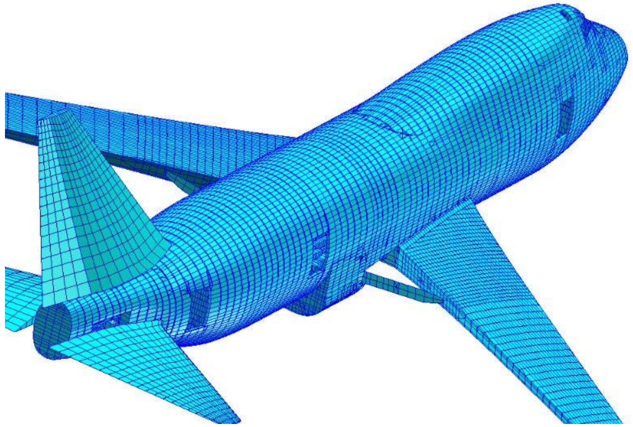
Procedure: what the computer does

Definition: A numerical method is a *procedure* run by a computer to give an approximate answer to some mathematical problem.

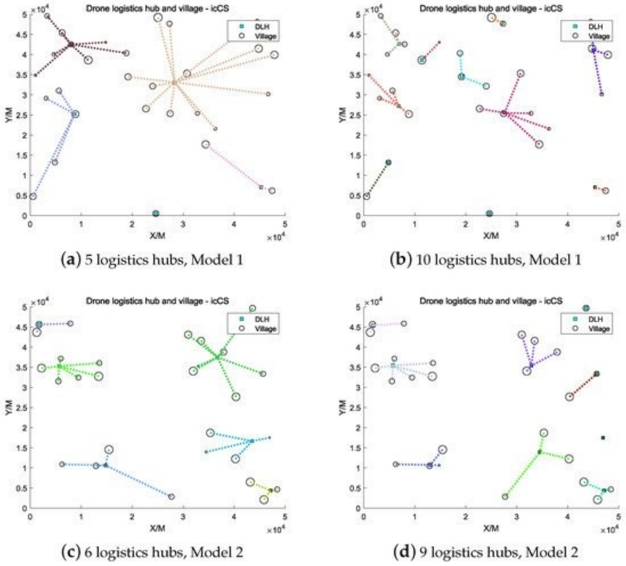
approximate: because while sometimes you can get an *exact* solution, almost always you can't.

By *mathematical* we mean any problem that can be stated mathematically, not limited to pure mathematics, but often arising in (say) physics and engineering.

Important applications:

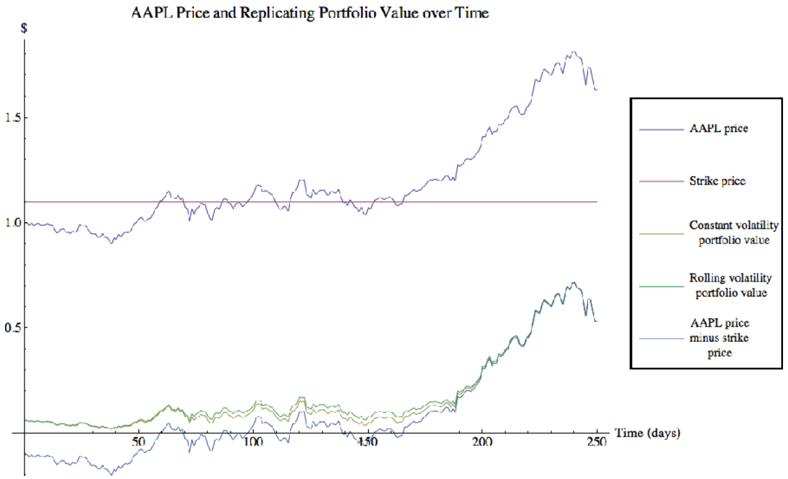


Structural engineering



Supply chain optimisation

Physics engines

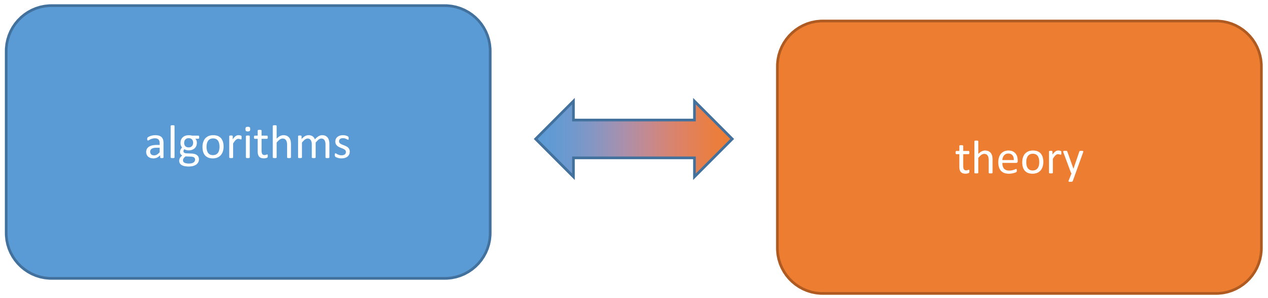


Mathematical finance

Is is sometimes useful to distinguish between *Numerical Methods* and *Numerical Analysis:*

Numerical Methods: what works

Numerical Analysis: why things work



The best route to understanding is

1) To use a mixture of *what* and *why*
2) To “get your hands dirty” fiddling around with things

This subject will mix the *what* and the *why*, but will concentrate mostly on *practical implementation*.

Why are numerical methods necessary?

Nowadays can't we just "throw massive amounts of computational power" at any problem we want to solve?

1. It's not that simple – someone has to programme the computers anyway, using… numerical methods

2. …

Main aims of this subject:

1. Introduce the main numerical methods currently in use in scientific and engineering computing

2. Get an appreciation for *how* these methods work, *and more importantly when they don't*

3. Give you practice in implementing each of these methods

Subject structure by week*:

1. Programming basics. Intro to Python: control structures, data types, style and technique. Numerical precision. Graphing functions. Numerical differentiation
2. Root finding and Numerical solution to nonlinear equations. Newton's method. Bisection, the secant method, etc.
3. Minimisation and maximisation in 1D
4. Numerical optimisation in higher dimensions – gradient descent, simplex methods
5. Interpolation and extrapolation – splines etc
6. Integration in 1D: quadrature etc.
7. Integration in multiple dimensions: Monte Carlo methods
8. Solution of linear systems
9. Methods for sparse linear systems
10. Solution to ODEs – Euler, Runge Kutta, Predictor-Corrector
11. Numerical solution of ODEs and PDEs by finite differences
12. The Fast Fourier Transform (time permitting)

*Note that this is the first time that this subject has been run, so this is (I hope) only a *good approximation* of what will be covered.

Programming Basics

Why Python?

Overall structure of non-parallel code

The important control structures

Stopping conditions

Data types

Style and Technique

Why are we using Python?

Disadvantages:

- it’s relatively slow
- it is not very “elegant” as a language
- the module libraries are a bit of a mess
- not great for memory-intensive tasks

Advantages

- it’s free
- it’s easy to learn
- it’s easy to read
- it is now the Industry Standard in scientific computing

Overall structure (of non-parallel code):

```python
# -*- coding: utf-8 -*-
"""
sum_integers.py

Created on Wed Jul 20 17:31:57 2022
@author: Chris Poulton

Script to add the first N integers together

"""

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

"""
Function definitions
--------------------
"""
def foo(n):
    foo = n*(n+1)/2
    return foo

"""
Main script starts
-------------------
"""
print("Please enter the maximum value:")
nmaxstr = input()
nmax = int(nmaxstr) #convert the input to an integer type

print("Summing all integers from 1 up to",nmax)

nsum = 0
for i in range(1,int(nmax)+1): #note that range(1,N) creates a list from 1 to n-1
    print(i,"+")
    nsum = nsum + i

print("=",nsum)

print("Analytic formula: sum =",foo(nmax))
```

A good “beginners cheat sheet” for python is given in canvas
(from https://nbisweden.github.io/workshop-python/img/cheat_sheet.pdf)

Python for Beginners – Cheat Sheet

Data types and Collections

integer	10
float	3.14
boolean	True/False
string	‘abcde’
list	[1, 2, 3, 4, 5]
tuple	(1, 2, ‘a’, ‘b’)
set	{‘a’, ‘b’, ‘c’}
dictionary	{‘a’:1, ‘b’:2}

Numerical Operators

+	addition
-	subtraction
*	multiplication
/	division
**	exponent
%	modulus
//	floor division

Comparison Operators

<	less
<=	less or equal
>	greater
>=	greater or equal
==	equal
!=	not equal

List Methods

l.append(x)	append x to end of list
l.insert(l, x)	insert x at position i
l.remove(x)	remove first occurrence of x
l.reverse()	reverse list in place

Operations

Index starts at 0

Strings:

s[i]	i:th item of s
s[-1]	last item of s

Lists:

l = []	define empty list
l[i:j]	slice in range i to j
l[i] = x	replace i with x
l[i:j:k]	slice range i to j, step k

Dictionaries:

d = {}	create empty dictionary
d[i]	retrieve item with key i
d[i] = x	store x to key l
i in d	is key i in dictionary

Logical Operators

and	logical AND
or	logical OR
not	logical NOT

Membership Operators

in	value in object
not in	value not in object

Conditional Statements

```
if condition:
    <code>

elif condition:
    <code>

else:
    <code>
```

Dictionary Methods

d.keys()	returns a list of keys
d.values()	returns a list of values
d.items()	returns a list of (key, value)

String Methods

s.strip()	remove trailing whitespace
s.split(x)	return list, delimiter x
s.join(l)	return string, delimiter s
s.startswith(x)	return True if s starts with x
s.endswith(x)	return True if s ends with x
s.upper()	return copy, uppercase only
s.lower()	return copy, lowercase only

Import from Module

from module import func	import func
from module import func as f	import func as f

Python for Beginners – Cheat Sheet

Built-in Functions

float(x)	convert x to float
int(x)	convert x to integer
str(x)	convert x to string
set(x)	convert x to set
type(x)	returns type of x
len(x)	returns length of x
max(x)	returns maximum of x
min(x)	returns minimum of x
sum(x)	returns sum of values in x
sorted(x)	returns sorted list
round(x, d)	returns x rounded to d
print(x)	print object x

Loops

```
while condition:
   <code>

for var in list:
   <code>
```

Control statements:

break	terminate loop
continue	jump to next iteration
pass	does nothing

String Formatting

```
"Put {}into a {}".format("values", "string")
'Put values into a string'

"Put whitespace after: {:<10}, or before:{:>10}".format("a","b")
'Put whitespace after: a         , or before:         b'

"Put whitespace around:{:^10}.".format("c")
'Put whitespace around:     c     .'
```

Regular Expressions

```
import re
```

p = re.compile(pattern)	compile search query
p.search(text)	search for all matches
p.sub(sub, text)	substitute match with sub

.	any one character
*	repeat previous 0 or more times
+	repeat previous 1 or more times
?	repeat previous 0 or 1 times
\d	any digit
\s	any whitespace
[abc]	any character in this set {a, b, c}
[^abc]	any character *not* in this set
[a-z]	any letter between a and z
a|b	a or b

Reading and Writing Files

```
fh = open(<path>,'r')
for line in fh:
   <code>
fh.close()

out = open(<path>,'w')
out.write(<str>)
out.close()
```

Functions

```
def Name(param1, param2 = val):
   <code>
   #param2 optional, default: val
   return <data>
```

sys.argv

import sys	import module
sys.argv[0]	name of script
sys.argv[1]	first cmd line arg

<u>The important control structures</u>

1. Sequential statements

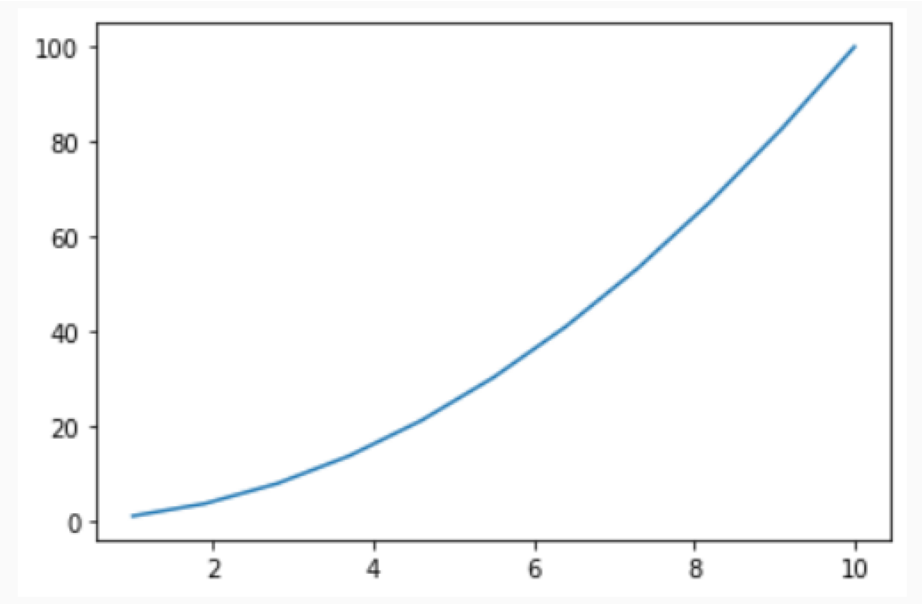
Tells computer to run commands in a fixed order

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

x = np.linspace(1,10,11)
b = 2

c = x**b

plt.plot(x,c)
```



See: https://www.educative.io/answers/what-are-control-flow-statements-in-python

2. Conditional statements

Creates two (or more) paths for the computer to follow

Instructions: if, else, elif

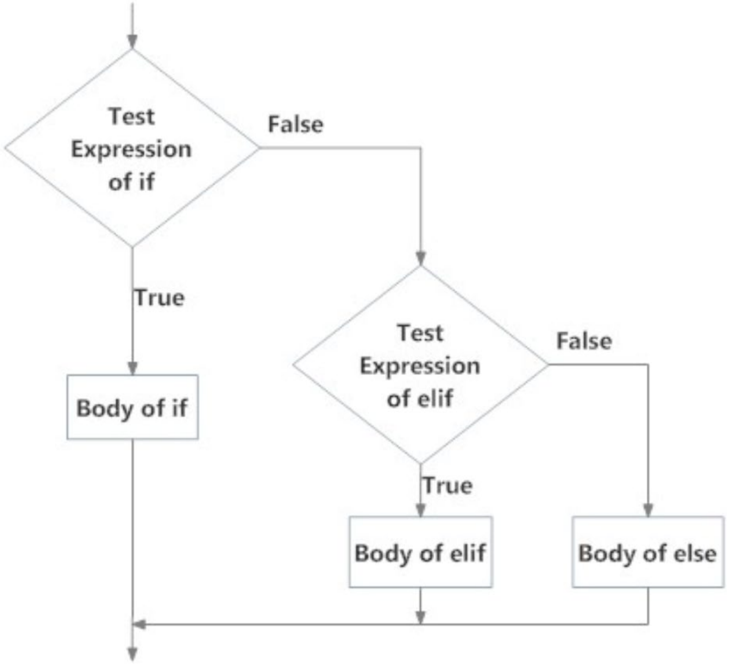


Fig: Operation of if...elif...else statement

```
a = 5
b = 10
c = 15
if a > b:
   if a > c:
      print("a value is big")
   else:
       print("c value is big")
elif b > c:
    print("b value is big")
else:
     print("c is big")
```

```
In [1]: run selection.py
c is big

In [2]:
```

3. Iteration (loops)

Performs a loop a number of times (**for**) or while a specified condition is true (**while)**

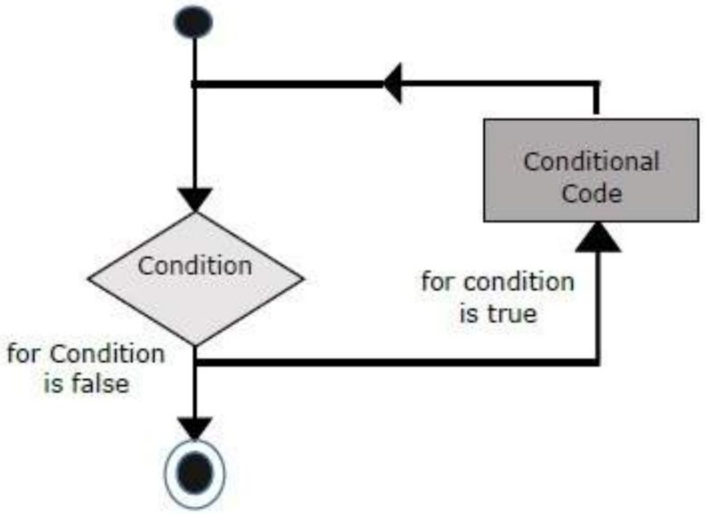
```python
import numpy as np
import matplotlib.pyplot as plt
import math as math


for j in range(0,10):
    print(j, end = " ")
```

```
In [5]: run iteration1
0 1 2 3 4 5 6 7 8 9

In [6]:
```

FOR loop:



Aside: The range() instruction in python

The instruction range(N) returns a list of integers, starting at zero and ending at N-1.
It is good for creating lists of integers.

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


for j in range(0,10):
    print(j, end = " ")
```

You can change the starting value from zero to anything you like:

```
for j in range(4,10):
    print(j, end = " ")
```

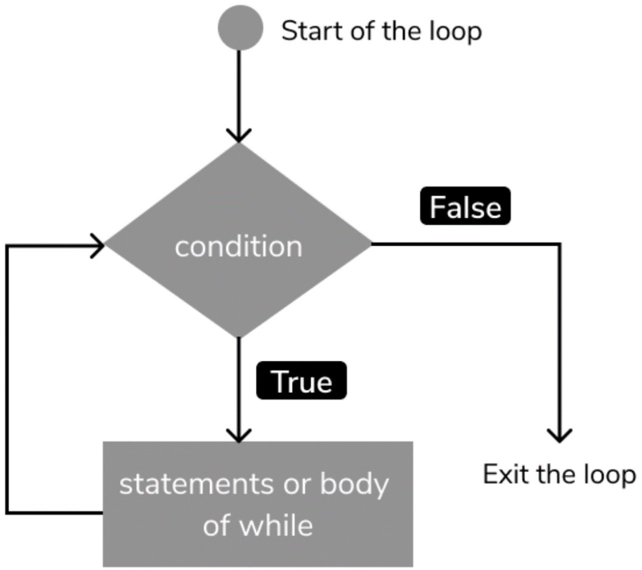
```python
m = 5
i = 0
while i < m:
     print(i, end = " ")
     i = i + 1
print("End")
```

```
In [5]: run iteration1
0 1 2 3 4 5 6 7 8 9

In [6]: run iteration2
0 1 2 3 4 End

In [7]:
```

WHILE loop:



<u>Dangers of a while loop:</u>

The code can run forever!

e.g. what happens here?

```python
m = -1
i = 0
while i > m:
     print(i, end = " ")
     i = i + 1
print("End")
```

Stopping conditions

It is very important to have a built-in “fail-safe” that tells your code where to stop (otherwise you have to stop it manually).

break: terminate the loop and go to the end

continue: jump over the remaining code inside the loop and go to the next iteration

pass: do nothing

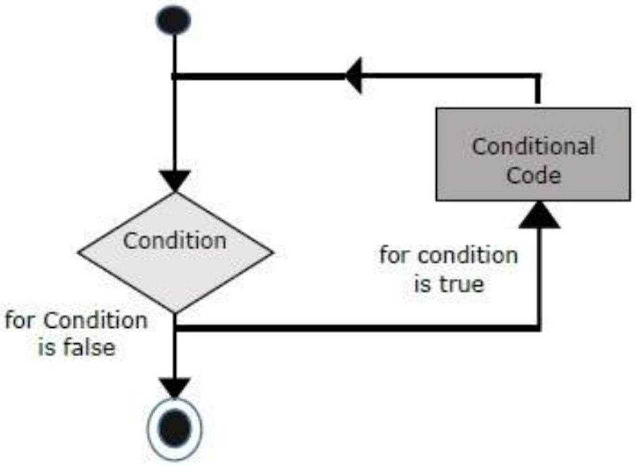
Loops

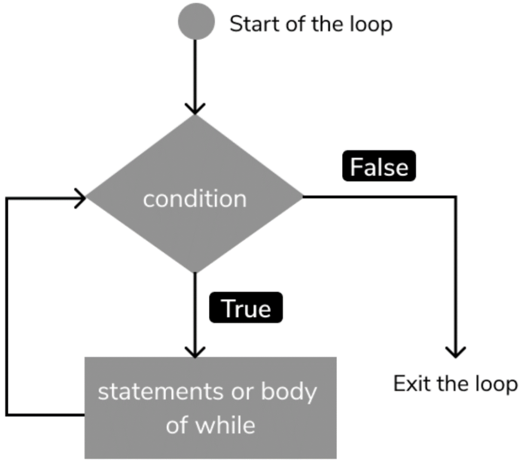
```
while condition:
  <code>

for var in list:
  <code>
```

Control statements:

break	terminate loop
continue	jump to next iteration
pass	does nothing





Data types in python

Data types and Collections	
integer	10
float	3.14
boolean	True/False
string	'abcde'
list	[1, 2, 3, 4, 5]
tuple	(1, 2, 'a', 'b')
set	{'a', 'b', 'c'}
dictionary	{'a':1, 'b':2}

Mostly we will be using *integers, floats, and Booleans.*

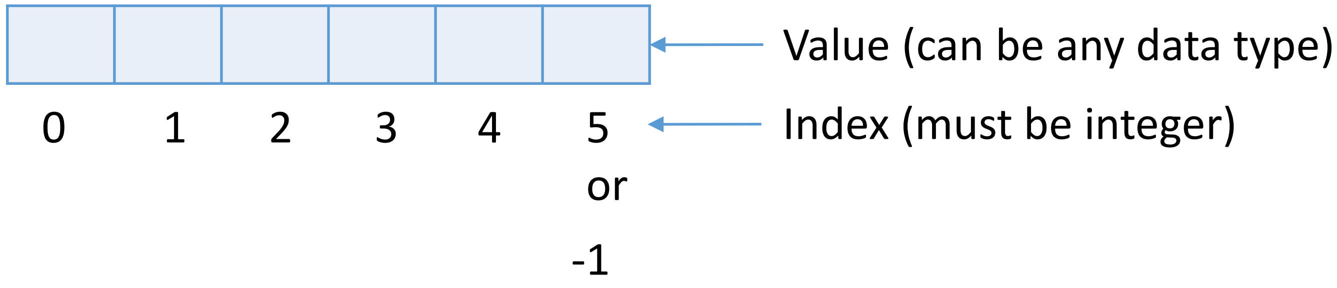
Lists

It will become important to gather numbers together in *Lists.*
Think of a list as a set of boxes into which we put numbers of a given type.

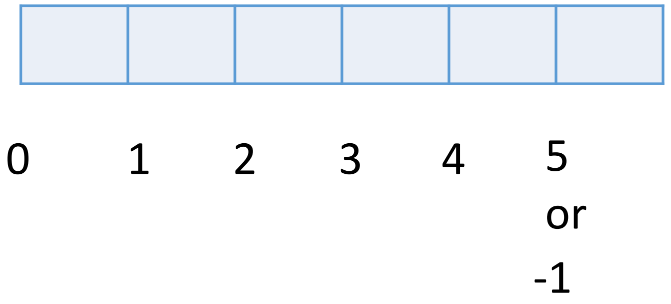
```
In [9]: mylist = [1.0, 1.2, 1.4, 1.6, 1.8, 2.0]

In [10]: mylist[1]
Out[10]: 1.2

In [11]:
```



It is sometimes useful to think of the index as labelling the *boundary* between two boxes



Lists can be created in the following ways

1. Manually:

```
In [9]: mylist = [1.0, 1.2, 1.4, 1.6, 1.8, 2.0]

In [10]: mylist[1]
Out[10]: 1.2

In [11]:
```

2. Using range():

```
In [1]: xrange = range(4,10)

In [2]: xrange[2]
Out[2]: 6

In [3]:
```

Or if you're careful, in a contracted form using range:

```
In [1]: pow2 = [x**2 for x in range(0,10)]

In [2]: print(*pow2)
0 1 4 9 16 25 36 49 64 81

In [3]:
```

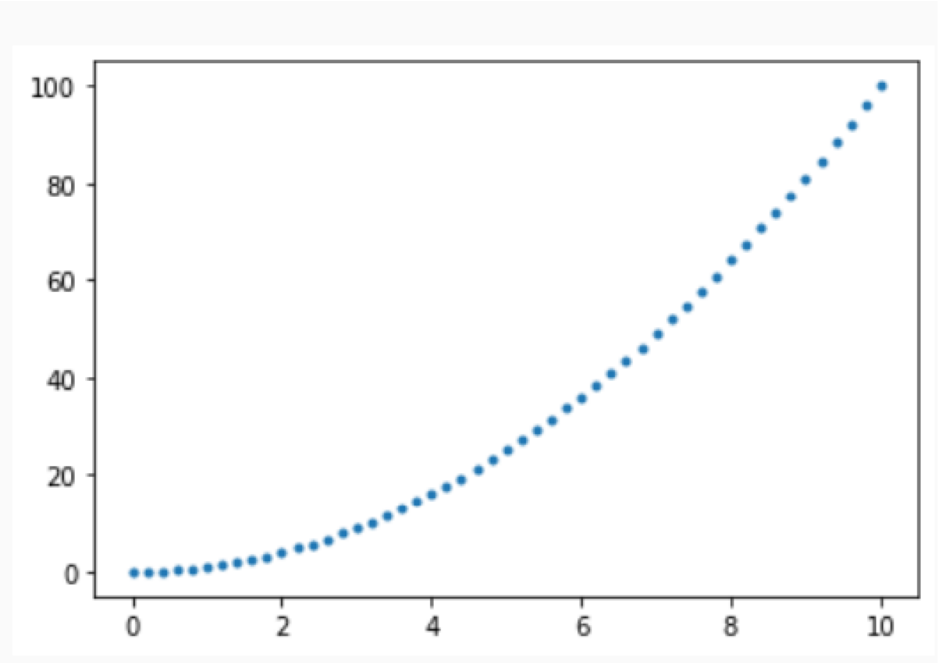
3. Using np.linspace (very important for this subject)

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


x = np.linspace(0,10,51)
y = x**2

plt.plot(x,y,'.')

```



Style

Developing good programming *style* is key to well-running code
(and getting good marks in this subject!)

Three general rules:

1. Always comment your code!

2. Be clear, not clever

3. Divide work into bite-size chunks (functions) and
only give each chunk what it needs to know (i.e. use information hiding)

1. Always comment your code!

(You will thanks yourself later)

```python

import numpy as np
def three(n):
    x3 = n**3
    return x3
x1 = 10
x2 = 0
for i in range(0,x1+1):
    x2 = x2 + three(i)
x2 = x2/(x1+1)
print(x2)
```

```python
# -*- coding: utf-8 -*-
"""
Created on Thu Jul 21 13:17:16 2022

@author: Chris

Code to compute the average of the first N cubes
starting from 0

"""

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

"""
Function definitions
--------------------
"""
def powfn(n):
    # takes n and raises it to the power of 3
    powfn = n**3
    return powfn

"""
Main script starts
------------------
"""

N = 10 # input maximum number

# sum over the first N
rsum = 0
for i in range(0,N+1):
    rsum = rsum + powfn(i)

# compute average by dividing by the number of terms:
av = rsum/len(range(0,N+1))

print("average:",av)
```

2. Be clear, not clever

```python
# confusing
x = float(input())
print("no") if x > 42 else print("yes") if x == 42 else print("maybe")

```

```python
#better
x = float(input())
if x>42:
    print('no')
elif x==42:
    print('yes')
else:
    print('maybe')

```

3. Divide work into bite-size chunks (functions) and

only give each chunk what it needs to know (i.e. use <u>information hiding</u>)

```
"""
Main script starts
-------------------
"""

pi = math.pi

xrange = np.linspace(0,2*pi,100)
dx = xrange[2]-xrange[1]

sin_int = 0
for x in xrange:
    # print and plot the values
    print(x, (math.sin(x))**2)
    plt.plot(x,(math.sin(x))**2,'.')

    sin_int = sin_int+(math.sin(x))**2*dx

av_sin = sin_int/(2*pi)

# show the plot with all the points and display result:
plt.show()
print("Average:",av_sin)


"""
Now do the same for cos**2(x):
"""

cos_int = 0
for x in xrange:
    # print and plot the values
    print(x, (math.cos(x))**2)
    plt.plot(x,(math.cos(x))**2,'.')

    cos_int = cos_int+(math.cos(x))**2*dx

av_cos = cos_int/(2*pi)

# show the plot with all the points and display result:
plt.show()
print("Average:",av_cos)
```

```python
"""
Function definitions
--------------------
"""
def mysquav(input_fun,xrange):
    # returns the average of the square of an input function
    # over the range given by xr, using Riemann integration.
    # input_fun: a single-variabled function
    # a range of evenly-spaced real numbers to integrate over

    dx = xrange[2]-xrange[1] # width of each interval
    L = xrange[-1]-xrange[0] # total length of interval

    fun_int = 0
    for x in xrange:
        # print and plot the values
        #print(x, (input_fun(x))**2)
        plt.plot(x,(input_fun(x))**2,'.')

        fun_int = fun_int+input_fun(x)**2*dx

    mysquav = fun_int/L

    return mysquav

"""
Main script starts
-------------------
"""

pi = math.pi
xrange = np.linspace(0,2*pi,100)

av_sin = mysquav(math.sin,xrange)
plt.show()
print("Average sin**2(x):",av_sin)

av_cos = mysquav(math.cos,xrange)
plt.show()
print("Average cos**2(x):",av_cos)
```

The golden rule: never write the same code segment twice!

Precision and numerical differentiation

How computers store real numbers

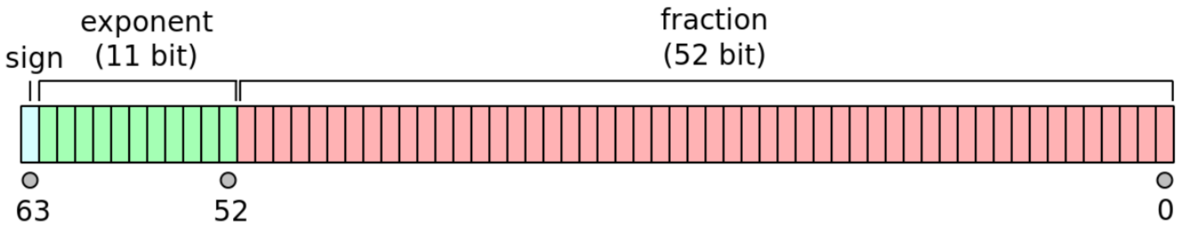
Machine precision

Types of errors

Finding numerical derivatives

<u>How python 3 stores real numbers</u>

Python stores real numbers in *double-precision, floating point format.* Each number is a string of ones and zeros in the following form:



If we want to convert a number from this form to a real value then the formula is:

$$\text{value} = (-1)^{\text{sign}} (1.b_{51}b_{50}...b_0)_{\text{base2}} \times 2^{e-1023}$$

This is a bit hard to grasp in binary, so let's imagine this would work in base 10 (the end result is the same):

2.5	1	0	9	9	2	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2.5×10^{-1}	1	0	9	8	2	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2.5×10^{-2}	1	0	9	7	2	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2.5×10^{30}	1	1	2	9	2	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

If two numbers get closer together, then eventually the computer *cannot distinguish them*.

e.g.

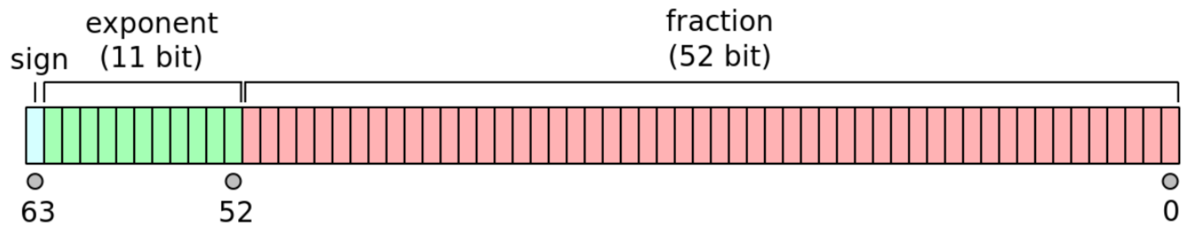
2.50000000000000052

1	0	9	9	2	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5

2.50000000000000057

1	0	9	9	2	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5

The threshold at which two numbers become indistinguishable is known as the *machine precision* ϵ_m



For double precision floats, the machine precision is $\epsilon_m = 2^{-52} \approx 2e - 16$

```
In [3]: np.finfo(float).eps
Out[3]: 2.220446049250313e-16

In [4]:
```

(Important: the machine precision is *not* the smallest number that can be stored on the machine.)

The prevision only depends on the length of the fraction, whereas the smallest number depends on the number of bits in the exponent

1	**0**	**0**	**0**	**2**	**5**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**5**

An error due to machine precision is a *fractional* or *relative* error – it tells us how much a number can be "off" *as a fraction of the number itself.*

To see this, consider the example from earlier: The machine cannot distinguish

2.50000000000000052

1	0	9	9	2	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5

2.50000000000000057

1	0	9	9	2	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5

But it has equal trouble distinguishing

$2.50000000000000052 \times 10^{30}$

1	1	2	9	2	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5

$2.50000000000000057 \times 10^{30}$

1	1	2	9	2	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5

Two types of errors:

Round-off error occurs when errors in the storing of numbers accumulate.

This error depends on the machine, and on how the machine stores numbers.

There is generally not much that can be done about this*.

Truncation error occurs when there is a difference between what you compute and the true answer. It occurs even when the round-off error is zero.

This error depends on your algorithm.

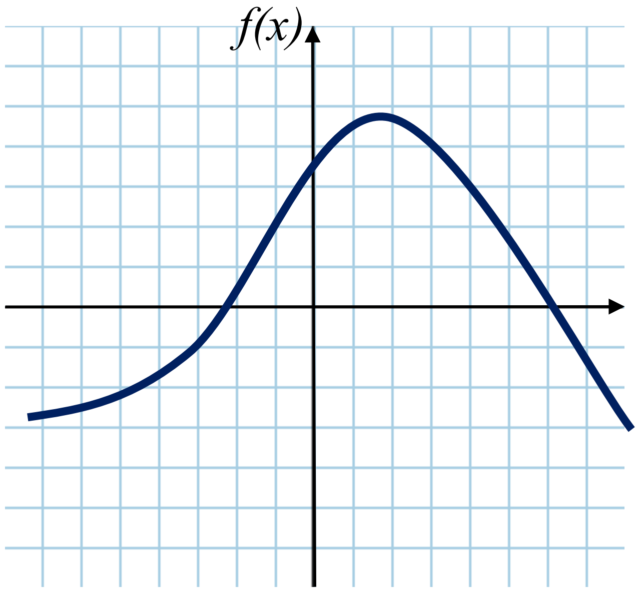
Reducing truncation error is practically the entire goal of all numerical methods.

*there are actually a few things that you can do to minimise round-off error but we'll get to them later

Topic 1: Numerical Differentiation

First rule of numerical differentiation: Avoid it if at all possible

Discussion question:
Imagine that we have a function $f(x)$ which we can evaluate only numerically.
How can we compute *the derivative* $f'(x)$?



The formula for computing the derivative

$$f'(x) \approx \frac{1}{h}\left(f(x+h) - f(x)\right)$$

is known as a *forward difference.*

There are other options: for example, we could also try the *balanced difference*

$$f'(x) \approx \frac{1}{2h}\left(f(x+h) - f(x-h)\right)$$

It turns out that the balanced difference method *performs far better* than the forward difference method.

The forward difference method has both *round-off error* and *truncation error*

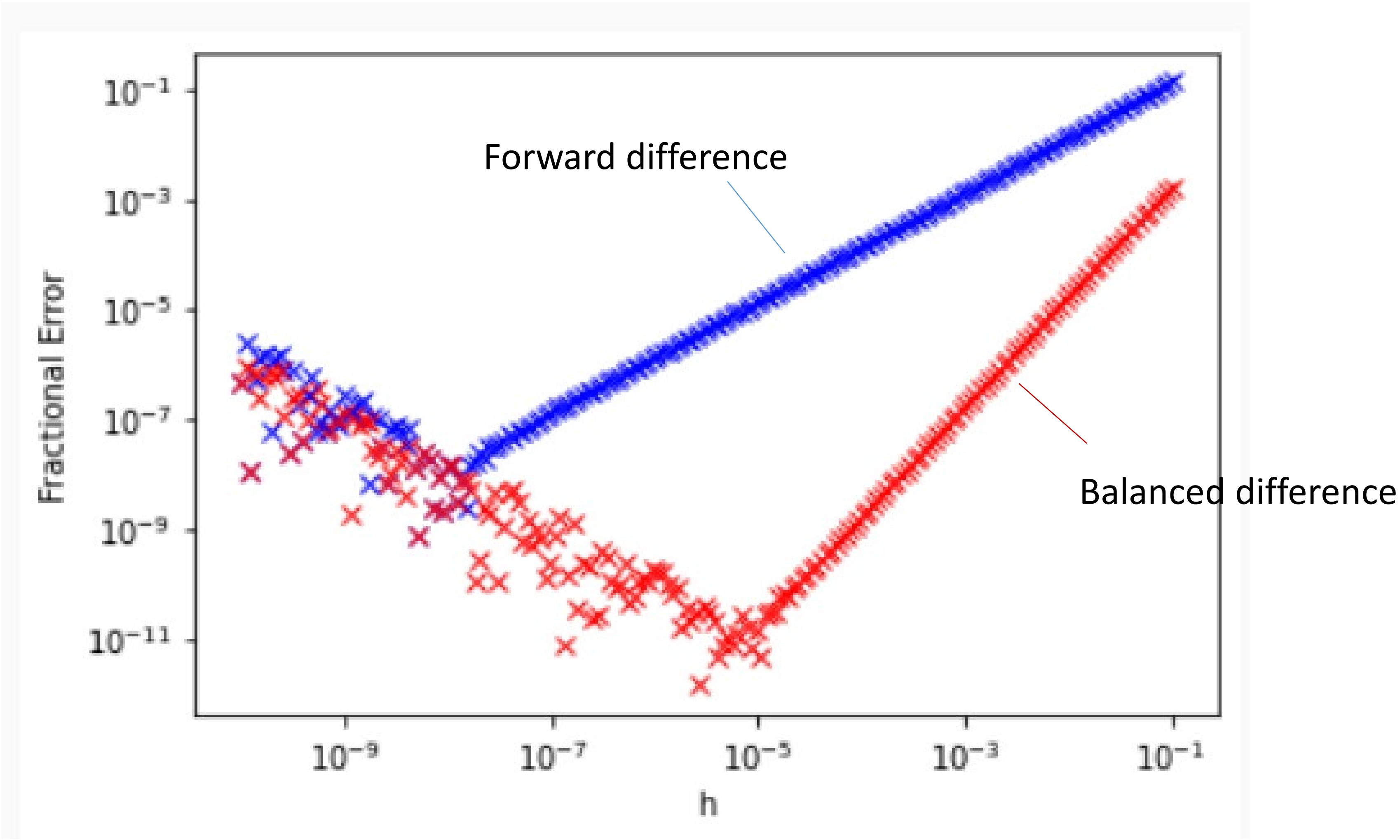
Round-off error calculation:

Imagine we can compute the function $f(x)$ to machine precision ϵ_m

That is:

Truncation error calculation:

From Taylor series, we know that



Forward difference
Balanced difference
Fractional Error
h
10^{-1}
10^{-3}
10^{-5}
10^{-7}
10^{-9}
10^{-11}