Lecture 2.1

Status update

We have now covered

- Part 0: **The Fundamentals** *revision of PH-113*
- Part 1:: **Fundamentals, functions, and finding numerical solutions**

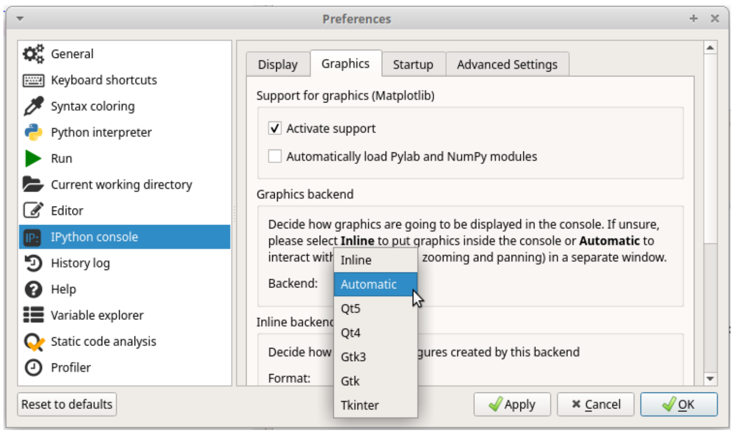
i.e. you have everything you need for Assignment 1. Doing it is hopefully a useful exercise. You have to combine components in novel ways. I am happy to answer questions, either in Office Hours, in or after Lectures, or by Email.
Next, we meet

- Part 2: **Numerical integrals, interpolation, and fitting to data**

Next two workshops are **unassessed** practice of this material, and (if time) assistance with Assignment 1.

Interactive/non-interactive plotting

Plots in Spyder are, by default, shown in the Terminal. This is often annoying. Switch to it making a new window, in which you can zoom etc, by changing, in Spyder Preferences, IPython Console/Graphics/Backend to Automatic.



Support for graphics (Matplotlib)

☑ Activate support

☐ Automatically load Pylab and NumPy modules

Graphics backend

Decide how graphics are going to be displayed in the console. If unsure, please select **Inline** to put graphics inside the console or **Automatic** to interact with zooming and panning) in a separate window.

Backend: Automatic

Inline backend

Decide how gures created by this backend

Format:

`plt.plot` needs explicit values for `x` and `y`.
Save a figure *from the code* using something like `plt.savefig("plot.pdf")`. Follow with `plt.close('all')` or subsequent plots may contain previous plots.

Data structures

You can assign two or more things at once:

```
a,b,c = 1,2,3
stuff = 1,2,3
a,b,c = stuff
```

`stuff` is a tuple. A tuple can be indexed like a list, but it cannot be changed. You can add/remove/sort elements of a list. A tuple is **immutable**.
By **convention**, a list contains lots of the same thing (e.g. integers), while a tuple contains several different things (e.g. names and ages).
There are *data structures*. Other useful data structures are dictionaries and sets.
A dictionary has several *keys* (e.g. people's names) each of which has an associated *value* (e.g. their ages):

```
ages = {'graham': 48, 'eric': 75, 'terry': 77, 'john': 78, 'michael': 75}
```

Dictionaries can be changed; they are **mutable**. Elements are referred to by key, e.g.

```
ages['terry'] = 76
```

Python set implements the notion of a mathematical set, including union, intersection, difference operations. A set contains only unique values. It is mutable. e.g. a start at implementing the Sieve of Eratosthenes:

```
numbers = set(range(2,10))
numbers -= set(range(4,10,2)) # remove all even numbers greater than 2
numbers -= set(range(6,10,3)) # remove all numbers divisible by 3 and greater than 3
numbers
```

```
{2, 3, 5, 7}
```

Procedural vs Functional

The factorial of a positive integer N is defined as the product of all integers from 1 to N.
Mathematically, we can write $f(N) = N \times (N-1) \times (N-2) \times \ldots 2 \times 1$, where I've chosen to write $f(N)$ rather than $N!$ because this is a function, like any other.
In Python, we can implement this like so

```
def factorial(N):
    answer = 1
    for n in range(1,N+1):
        answer *= n
    return answer
```

```
factorial(10)
```

```
3628800
```

An alternative way to write this definition is $f(N) = N \times f(N-1)$ with the special case $f(0) = 1$.
In Python, we can implement this like so:

```
def factorial(N):
    if N == 0: return 1
    return N*factorial(N-1)
```

```
factorial(10)
```

```
3628800
```

The former is very *procedural*. The latter is very *functional*. These are different styles of programming. Often, one is far simpler than the other for a given task.
Here's an example where naive functional approach is a terrible idea: Fibonacci numbers, where each term is the sum of the previous two, and the first two terms are 0 and 1.

```
def fib(N):
    if N == 0: return 0
    if N == 1: return 1
    return fib(N-1) + fib(N-2)
```

```
fib(10) # no problem for small N
```

```
def fib(N):
    nums = [0,1]
    for n in range(2,N+1):
        nums.append(nums[-1] + nums[-2])
    return nums[-1]
```

```
fib(100) # try this with the functional definition if you like...
```

```
354224848179261915075
```

Why is the functional implementation of Fibonacci so bad?

Numerical differentiation

We met numerical differentials

```
def fprime(x):
    h = 1e-6
    return (f(x+h)-f(x))/h
```

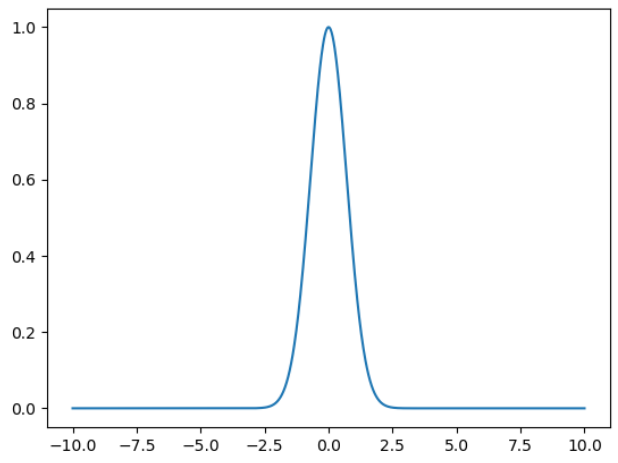
This is an *approximation* which, instead of taking $h \to 0$, uses a small, finite h.

Numeric integration: trapezium

The *integral* is the area under a curve. i.e. $F(x) = \int_0^x f(x')dx'$. We can *approximate* this in various ways. The simplest is the rectangle method. One better is the trapezium method. There's a whole family. Trapezium method is hopefully familiar.
To illustrate it's use, let's define a function (and plot it so we know what we're dealing with):

```
import numpy as np
import matplotlib.pyplot as plt
def f(x):
    return np.exp(-x**2)

x = np.linspace(-10,10,1001)
y = f(x)
plt.plot(x,y)
plt.savefig("figs/gaussian.png")
plt.close('all')
```



The trapezium rule is available from `scipy.integrate`.

```
from scipy.integrate import trapz
Integral = trapz(y,dx=x[1]-x[0]) # it needs to know dx, the width of the trapeziums
Integral
```

```
1.77245385091
```

Theoretical result: $\int_0^\infty e^{-x^2} dx = \sqrt{\pi} \approx 1.77245$, so "not bad". Quantify how close.

Numeric integration: trapezium, accuracy

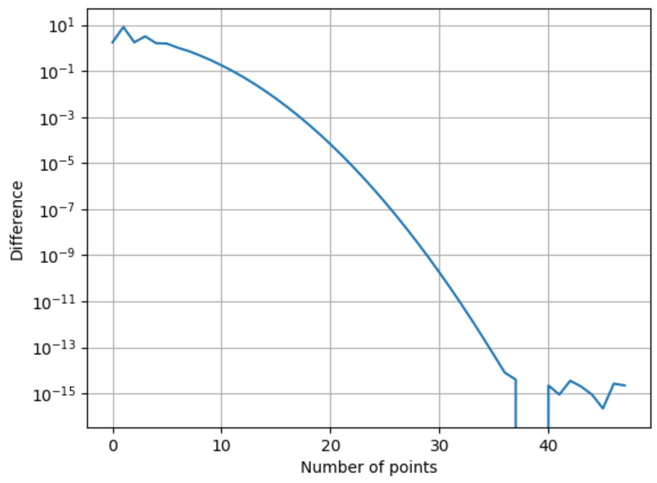
What if we did fewer points? How would the accuracy of our approximation worsen?

Let's do the integration with an increasingly large number of points.

```
analytical = np.sqrt(np.pi)

difference = list()
for N in range(2,50):
    x = np.linspace(-10,10,N)
    y = f(x)
    approximation = trapz(y,dx=x[1]-x[0])
    difference.append(abs(approximation-analytical))

plt.plot(difference)
plt.yscale('log')
plt.grid(True)
plt.xlabel('Number of points')
plt.ylabel('Difference')
plt.savefig('figs/numeric_integrate_diff.png')
plt.close('all')
```



This is best viewed on a log scale. Very quickly, we approach $\sim 10^{-15}$, and then no better because we're limited by the resolution of real numbers in Python.

Numeric integration: quad

trapz is fast and simple, but we need to tell it where to sample the function.

An alternative method is quad. We must pass quad the **function** to integrate, and the upper and lower limits, not the list of points (x, y) at which we've sampled the function.

```
from scipy.integrate import quad

def f(x):
    return np.exp(-x**2)

Integral = quad(f, -10, 10)
Integral
```

```
(1.772453850905516, 3.695852112137264e-13)
```

quad returns two things (as a tuple): the approximated integral, and an estimate of the uncertainty. Refer to these pieces either beyindex

```
Integral[0]
```

```
1.772453850905516
```

or by *putting more than one thing on the left hand side*

```
Integral, error = quad(f, -10, 10)
Integral
```

```
1.772453850905516
```

Note that quad can do **infinite** integrals:

```
Integral, error = quad(f, -np.inf, np.inf)
Integral, error
```

```
(1.7724538509055159, 1.4202636780944923e-08)
```

Notice that the error is **worse** here than when we did the finite integral.

In general, the uncertainty depends strongly on the method. You must keep an eye on it and always remember that numeric computing is approximate.