INTRODUCTION TO COMPUTATIONAL PHYSICS

Referral/Deferral – Python Section

U24200 –Academic Session 2020–2021

Instructions

a) **You must undertake this assignment individually.**

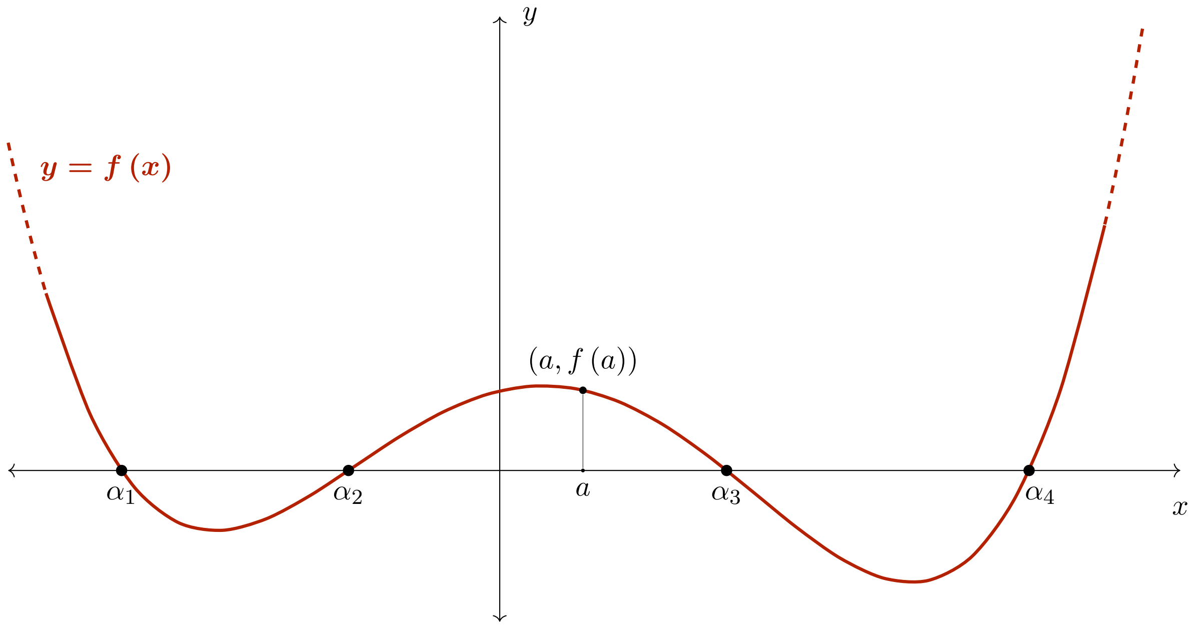
b) Submission method: **by Moodle through the available dropbox**.

1 Introduction

Let $f(x)$ be a continuous function of a single variable. It can be:

- a simple function, e.g. a polynomial $f(x) = a_n x^n + \cdots + a_1 x + a_0$ (for instance $f(x) = x - 1$, $f(x) = 3x^2 + 4x - 5$, etc)
- or a more complicated function such as $f(x) = \cos x$, $f(x) = e^x$, ...

Having one real input x and one real output $y = f(x)$ for every input, this function can be represented by a two-dimensional graph plotting *abscissae* x against *ordinates* y:



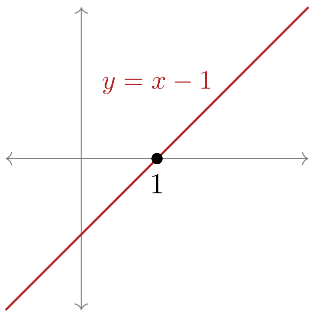
We have highlighted four values $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ in the above example. They are values of x such that $f(x) = 0$, i.e. the graph of f intersects the x-axis. We usually call such values **zeros** (or **roots**) of the function f, or **roots** (or simply *solutions*) of the equation $f(x) = 0$.

Two situations can arise, depending on the function $f(x)$ you are working with:

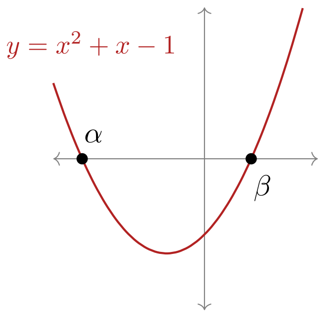
- sometimes finding or describing zeros of a function *symbolically* will be an easy task, e.g.
  - $f(x) = x - 1$ has only one root $\alpha = 1$,
  - $f(x) = x^2 + x - 1$ has two roots $\alpha = \frac{-\sqrt{5}-1}{2}$, $\beta = \frac{\sqrt{5}-1}{2}$ that are easy to find exactly using the *quadratic formula*,
  - $f(x) = \cos(x)$ has infinitely many roots but we know them: $\frac{(2k+1)\pi}{2}$ (k any integer),
  - and $f(x) = e^x$ has no zeros at all, because it is strictly positive on all values of x;

  by *symbolically* we mean roots we can represent exactly because we do not require decimal approximations to write them down (even if we may need approximations to *operate* with them in real-life scenarios, using $\sqrt{5} \simeq 2.23607\ldots$ and $\pi \simeq 3.14159\ldots$);

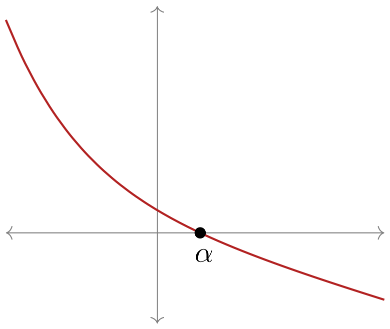
- most oftentimes, however, we may determine the *amount* of roots – but finding one, let alone all, symbolically will be either difficult or impossible. For instance, a thorough study of the roots of $e^{-x} = x$ (i.e. the zeros of function $f(x) = e^{-x} - x$) reveals that there is only one such zero α and it lies in the interval $(0.5, 0.6)$. But we cannot describe α in closed form, unlike the above examples; we cannot write it as an integer, a multiple of a known constant (e.g. π), a combination of square roots or simple functions of known values, etc. The only thing we can do is approximate α with a fixed amount of correct significant digits, e.g. $\alpha \simeq 0.5671432904097838$ with 16 correct decimal digits and $\alpha \simeq 0.56714329040978387299996866221035$ with 32 correct decimal digits.



$f(x) = 0$ equates to $x = 1$



Formula $\frac{-b\pm\sqrt{b^2-4ac}}{2a}$ applied to $ax^2 + bx + c$, $a = 1$, $b = 1$, $c = -1$



Function $f(x) = e^{-x} - x$ has only one zero α but we cannot find it exactly

Numerical solution of equations consists in approximating one or more roots of a given equation $f(x) = 0$ using certain *root-finding algorithms*. Some remarks are in order here:

- these algorithms will work regardless of whether or not we can find the roots symbolically, e.g. for $f(x) = x - 1$ we know the exact solution, $x = 1$, but we can also apply a root-finding algorithm if we want to; if implemented correctly, it will approximate

  $$x \simeq 1.\boxed{\text{large amount of 0's}}\boxed{\text{other digits}} \quad \text{or} \quad x \simeq 0.\boxed{\text{large amount of 9's}}\boxed{\text{other digits}}$$

  and the better the approximation, the more 0's or 9's after the decimal point.

- As said above, most functions will not afford us the luxury of a symbolic solution, and numerical root-finding algorithms to approximate it will be the only tools available.

- Before applying the root-finding algorithm, we need to know the *precision*, i.e. the number of correct digits that we desire, e.g. 16 in the first approximation of α for $e^{-x} = x$ above.

Most root-finding algorithms are **iterative** methods: they entail the creation of a sequence of numbers $\{x_k\}_{k\geq 0}$ such, that every new iteration (i.e. element of the sequence) x_k can be obtained as a fixed function of a fixed number of previous iterations:

$$\boxed{x_k = G\left(x_{k-1}, x_{k-2}, \ldots, x_{k-j}\right)} \qquad G \text{ and } j \text{ being fixed and depending on the method.} \tag{1}$$

This requires, as you will realise yourselves, a set of j initial conditions $x_0, \ldots, x_{j-1}$.

Assume you want to fix a precision; more specifically, assume you wish to approximate a root α with k correct decimal digits. You can do so by fixing a certain tolerance $\varepsilon = \frac{1}{2}10^{-k}$, which is what you will use as a criterion to stop your iteration process. The process (1) will stop at a step n where two consecutive iterations differ by less than ε:

$$|\boldsymbol{x_n} - \boldsymbol{x_{n-1}}| < \frac{1}{2}10^{-k}, \qquad \text{i.e. } \boldsymbol{x_n} \text{ and } \boldsymbol{x_{n-1}} \text{ share their first } k \text{ decimal digits.} \tag{2}$$

Obviously, the challenge here is finding a function G, depending on the original function f, such that the sequence generated above converges to the root we are looking for: $\lim_{n\to\infty} \boldsymbol{x_n} = \alpha$. Once G is established (and we will not explain the mathematical reasoning behind this now), **condition (2) ensures, in most cases, that the last iteration $\boldsymbol{x_n}$ in our method will share p correct decimal digits with α as well**.

There are many examples of iterative methods, i.e. choices of G:

- **secant**, **Newton** and **bisection**, all of which you saw during the first semester;
- other methods, such as Steffensen's method, fixed-point iteration, Halley's method, etc.

We will see a family of methods generalizing Newton.

2 Householder's methods

Denote $^{(p)}$ to be the p^{th} order derivative, e.g. $f^{(0)} = f, f^{(1)} = f', f^{(2)} = f'' = \frac{d^2 f}{dx^2}$, etc.

Householder methods are one-point algorithms, i.e. $j = 1$: every iteration only needs the previous one: $x_k = G\left(x_{k-1}\right)$, $k \geq 1$. Thus only one initial condition is needed. The general form of the p^{th} Householder method is:

$$x_{n+1} = x_n + p\frac{F^{(p-1)}\left(x_n\right)}{F^{(p)}\left(x_n\right)}, \qquad \text{where } F\left(x\right) := \frac{1}{f\left(x\right)}. \tag{3}$$

For example, if $p = 1$ Householder becomes **Newton's method** which you already know:

$$F^{(p-1)}\left(x\right) = F^{(0)}\left(x\right) = F\left(x\right) = \frac{1}{f\left(x\right)}, \qquad F^{(p)}\left(x\right) = F'\left(x\right) = \left(1/f\right)'\left(x\right) = -\frac{f'(x)}{f(x)^2},$$

using the quotient or the chain rule for derivation. Thus (3) becomes in this case

$$x_{n+1} = x_n + 1 \cdot \frac{\frac{1}{f(x_n)}}{-\frac{f'(x_n)}{f(x_n)^2}} = x_n - \frac{f\left(x_n\right)}{f'\left(x_n\right)},$$

which should be familiar to you from TB1 when you were programming Newton's method.

The **order of convergence** of the method, if it does converge, is $p+1$ (which is why Newton was generally considered to be quadratic, i.e. of order 2, hence the number of correct decimal places roughly doubled after every iteration).

Thus, for higher values of p, you obtain faster methods. The only drawback, of course, is that you need to make the additional effort to compute higher derivatives; you will realize this yourselves when you find the expression for Householder for values $p > 1$.

3 The Actual Coursework

Write a Python program to approximate the roots of any function. It must contain, at least, the following functions.

1. A routine `initial_plot` plotting the graph of any $f(x)$, then giving the user the option to zoom and create new plots, or stop plotting altogether. This is identical to the TB1 coursework.

2. A function `Householder` for approximating roots numerically using Householder methods. This function should have at least the following parameters passed to them as inputs:

- the value of p described above; **your program should be operational for at least one value $\boxed{>1}$ of p; you already know how to program Newton ($p = 1$) so that would entail 0 marks here**.
- the function $f(x)$ itself and its derivatives $f'(x), f''(x), \ldots$, upper bound depending on which values of p you choose; for instance, if you only choose $p = 3$ then your program should be able to use derivatives $f'(x), f''(x), f'''(x)$.
- the initial condition `x0` required by the given method;
- the tolerance `tol` determining the precision with which you wish to approximate the root.

Your function must write initial conditions and all iterations in external files (for different functions and different values of p), arrayed in two columns, so that:

- one of the columns corresponds to the index $k = 0, \ldots$ determining the iteration;
- and another one for the iteration x_k itself.

Make sure you write this in a way that makes it readable from another Python routine.

3. Function `read_from_file` reading an external file and returning the column corresponding to the iterations with which the method (`Householder`) filled the file originally. Identical to the same function described in the TB1 coursework.

4. A routine `final_plot` doing the exact same thing as `final_plot`, and in addition having slightly larger, properly labelled dots for some of the iterations, along with captions containing all the relevant information. Identical to the TB1 coursework.

Let `XXXXXX` be your student number. All that is needed for you to upload to Moodle is:

- One Python file `UPXXXXXX_MAIN.py` containing the main body of your program.
- One Python file `UPXXXXXX_METHODS.py` containing your methods and, perhaps, other routines.
- Optional `UPXXXXXX_FUNCTIONS.py` if you wish to define functions and derivatives in a separate file.
- A few sample `.txt` files containing iterations of some methods (i.e. values of p) applied to functions.
- A few initial plots, showing how you located and zoomed into roots of at least one function.
- A final plot obtained from `final_plot`.
- Optional: a Jupyter notebook if it makes any of the above easier.

Mark ranges: if your program

- does none of these: run without errors, produce a plot or a numerical output: **0-45 marks**
- fails to do *at least* one of the things described above: **30-60 marks**
- runs without errors and produces *one* plot, but lacks what is said below: **45-70 marks**
- same as item above and the process is correct for arbitrary $f(x)$: **60-100 marks**
- *two* Householder methods instead of one (e.g. $p = 2, 3$, $p = 2, 4$, etc): **10 bonus marks**

Mark ranges overlap because the global layout of your program and your ability to detect glitches will also count.