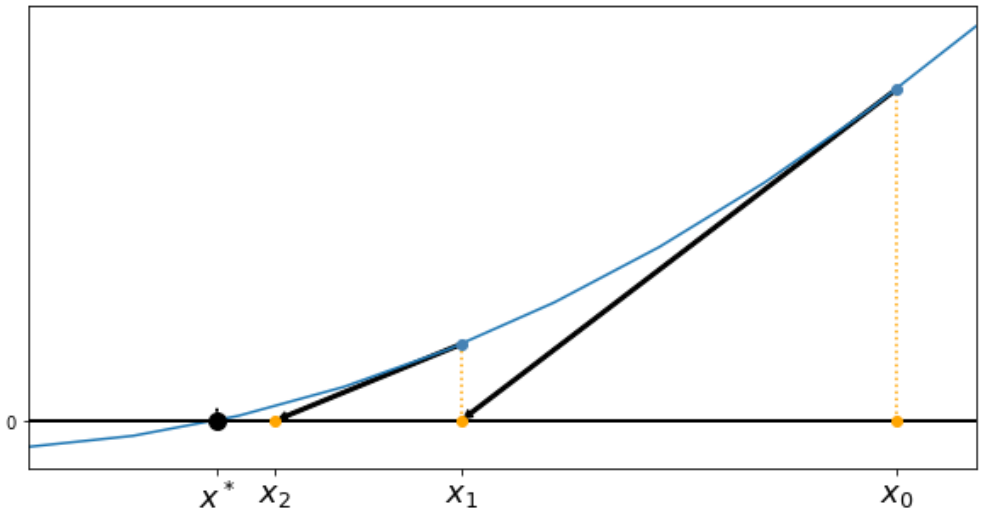
Bachelor of Ecole Polytechnique
APM_1S006_EP: Introduction to Numerical Analysis
Maxime Breden

Based on Notebooks created by Aline Lefebvre-Lepo

Root-finding for a function of one variable



In this chapter, we consider the problem of finding **roots of an equation in one variable**: find x such that $f(x) = 0$. We discuss numerical methods to approximate solutions of this kind of problems to an arbitrarily high accuracy. First, we formalize the notion of convergence and order of convergence for **iterative methods**. Then, we focus on

> three iterative algorithms for approximating roots of functions: **the bisection method**, **fixed point iterations** and **the Newton-Raphson method**. These methods are described, analyzed and used to solve 3 problems coming from physics, finance and population dynamics.

> The symbol [★] indicates supplementary material that is optional to understand. This material is provided for the sake of completeness and/or for interested readers.

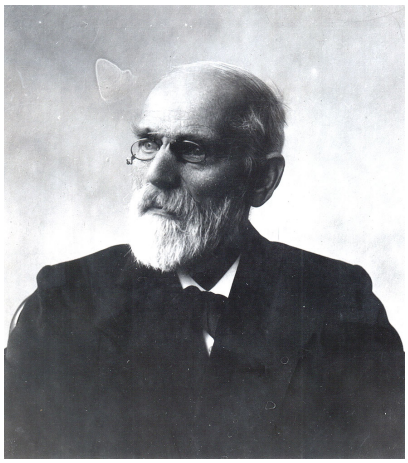
Table of contents


```
## loading python libraries

# necessary to display plots inline:
%matplotlib inline

# load the libraries
import matplotlib.pyplot as plt # 2D plotting library
import numpy as np              # package for scientific computing

```

Introduction

The zeros (or roots) of a function f are the x such that $f(x) = 0$. Such problems can be encountered in many situations, as it is common to reformulate a mathematical problem so that its solutions correspond to the zeros of a function. In many of these situations, the solution cannot be computed exactly and one has to design numerical algorithms to approximate the solution instead. We give below a few examples of such situations.

Case study 1: State equation of a gas [★]



Johannes Diderik van der Waals (1837-1923). He is a Dutch theoretical physicist. He was primarily known for his thesis work (1873) in which he proposed a state equation for gases to take into account their non-ideality and the existence of intermolecular interactions. His new equation of state revolutionized the study of the behavior of gases. This work was followed by several other researches on molecules that have been fundamental for the development of molecular physics.

The state equation of a gas relating the pressure p, the volume V and the temperature T proposed by van der Waals can be written

$$\left[p + a\left(\frac{N}{V}\right)^2\right](V - Nb) = kNT,$$

where N is the number of molecules of the gas, k is the Boltzmann-constant and a and b are coefficients depending on the gas. To determine the volume occupied by a gas at pressure p and temperature T, we need find the V which solves this equation. This problem is equivalent to finding a zero of the function f defined as

$$f(V) = \left[p + a\left(\frac{N}{V}\right)^2\right](V - Nb) - kNT.$$

Suppose one wants to find the volume occupied by 1000 molecules of CO_2 at temperature $T = 300\ K$ and pressure $p = 3.5 \cdot 10^7\ Pa$. Then, the previous equation has to be solved for V, with the following values of parameters a and b corresponding to carbon dioxide: $a = 0.401\ Pa\, m^6$ and $b = 42.7 \cdot 10^{-6}\ m^3$. The Boltzmann constant is $k = 1.3806503 \cdot 10^{-23}\ J\ K^{-1}$.

Case study 2: Investment fund [★]

Suppose someone wants to have a saving account valued at $S = 30\,000$ euros upon retirement in 10 years. The saving account is empty at first, but the person makes a deposit of $d = 30$ euros at the end of each month on this account. If we call i the monthly interest rate and S_n the capital after n months, we have:

$$S_n = \sum_{k=0}^{n-1} d(1+i)^k = d\frac{(1+i)^n - 1}{i}.$$

In order to know the minimal interest rate needed to reach at least $30\,000$ euros in 10 years, we must solve the following equation for i:

$$S = d\frac{(1+i)^{n_{end}} - 1}{i} \qquad \text{where} \qquad n_{end} = 120,$$

which can also be written as a root-finding problem.

Case study 3: A first population model [★]

Thomas Robert Malthus (1766-1834). He is a British economist. He is mainly known for his works about the links between the size of a population and its productions. He published anonymously in 1798 an *Essay on the principle of populations*. It is based on the idea that the growth of a population is essentially geometric while the growth of the production is arithmetic. This leads to the so-called Malthusianism doctrine suggesting that the population size has to be controlled to avoid a catastrophe.

Population dynamics is a branch of mathematical biology that gave rise to a great amount of research and is still very active nowadays. The objective is to study the evolution of the size and composition of populations and how the environment drives them. The first model that can be derived is a natural exponential growth model. It depends on two parameters: β and δ, the average numbers of births and deaths per individual and unit of time. If we suppose that these parameters are the same for all individuals and do not depend on the size of the population, we can denote the growth rate of the population by $\lambda = \beta - \delta$ and write:

$$\frac{dN}{dt} = \lambda N$$

where N is the population size. This model leads to exponentially increasing ($\lambda > 0$) or decreasing populations ($\lambda < 0$). Of course, this model can be made more realistic by including more effects, which leads for instance to the logistic population growth model. When the population is not isolated, one also has to take into account immigration or emigration. If we denote by r the average number of individuals joining the community per unit of time, a new model can be written as

$$\frac{dN}{dt} = \lambda N + r.$$

You will see in MAA105 how to solve such an equation. Assuming $\lambda \neq 0$, the number of individuals at time t is given by

$$N(t) = N(0)\exp(\lambda t) + \frac{r}{\lambda}(\exp(\lambda t) - 1).$$

If one wants to estimate the natural growth rate λ in France, one can use the following data:

Population 01/01/2016	Population 01/01/2017	migratory balance in 2016
66 695 000	66 954 000	67 000

and solve the corresponding equation for λ (unit of time = year)

$$N(2017) = N(2016)\exp(\lambda) + \frac{r}{\lambda}(\exp(\lambda) - 1).$$

Iterative methods: errors and convergence

Convergence / order of convergence

Some of the previous problems provide us examples where the exact solution cannot be computed through an explicit formula and has to be approximated through numerical methods.

Let us write these problems under the following generic root-finding form:

$$\text{given } \; f : [a,b] \to \mathbb{R}, \qquad \text{find } x^* \in [a,b] \text{ such that } f(x^*) = 0.$$

Methods for approximating a root x^* of f are often iterative: algorithms generate sequences $(x_k)_{k\in\mathbb{N}}$ that are supposed to converge to x^*. Given such a sequence, two questions one has to answer are:

- Does the sequence indeed converge to x^* ?
- if it converges, how fast does it converge to x^* ?

Before going further, we formalize below the notions of convergence and convergence speed.

Definition.

Convergence. Suppose that a sequence $(x_k)_k$ is generated to approximate x^*. The error at step k is defined as

$$e_k = | x_k - x^* |,$$

where $| \cdot |$ denotes the absolute value. The sequence $(x_k)_k$ is said to *converge to* x^* if

$$e_k \longrightarrow 0 \quad \text{when} \quad k \to \infty.$$

Most of the time, different sequences converging to x^* can be generated. One has to choose which one to use by comparing their properties such as the computational time or the speed of convergence.

Example. Let us consider the four following sequences converging to $x^* = 0$:

$$x_k = \frac{1}{k+1}, \quad \bar{x}_k = \left(\frac{1}{2}\right)^k, \quad \hat{x}_k = \left(\frac{1}{7}\right)^k, \quad \text{and} \quad \tilde{x}_k = \left(\frac{1}{2}\right)^{2^k}.$$

The values obtained for the first terms of these sequences are

k	0	1	2	3	4	5
x_k	1	0.5	0.33...	0.25	0.2	0.166..
$\bar{x}_k$	1	0.5	0.25	0.125	0.0625	0.03125
$\hat{x}_k$	1	0.14285	0.02041	0.00291	4.164 e -4	5.94 e -5
$\tilde{x}_k$	0.5	0.25	0.0625	0.00390..	1.52 e -5	2.328 e -10

The four sequences converge to zero but its seems that $\tilde{x}_k$ converges faster than $\hat{x}_k$, which converges faster than $\bar{x}_k$, which itself converges faster than x_k.

A way to quantify the speed at which a sequence converges is to estimate its *order of convergence*:

Definition.

Order of convergence for iterative algorithms. Suppose that the sequence $(x_k)_k$ converges to x^*. We say that its *order of convergence* is

$\alpha \geq 1$ if

$$\exists C > 0, \quad e_{k+1} \underset{k\to\infty}{\sim} C e_k^\alpha,$$

or equivalently, if

$$\exists C > 0, \quad \frac{e_{k+1}}{e_k^\alpha} \underset{k\to\infty}{\to} C.$$

The convergence is said to be

- *sublinear* if $\alpha = 1$ and $C = 1$,
- *linear* if $\alpha = 1$ and $C < 1$,
- *quadratic* if $\alpha = 2$.

The constant C is sometimes called the *rate* of convergence.

Remark.

- If we have an estimate of the form
$$\exists C > 0, \quad e_{k+1} \leq C e_k^\alpha \qquad \text{for all } k \text{ large enough},$$
but we do not know whether α is the optimal exponent (i.e., if maybe the same estimate would be true for a larger α, possibly with a different C), then we say that the order of convergence is *at least* α.
- The bigger the α, the faster the convergence when e_k gets close to 0. Roughly speaking, the number of correct digits in x_k is multiplied by α at each step. α being given, the smaller the C, the faster the convergence.

Do it yourself. Consider again the four following sequences converging to $x^* = 0$:

$$x_k = \frac{1}{k+1}, \quad \bar{x}_k = \left(\frac{1}{2}\right)^k, \quad \hat{x}_k = \left(\frac{1}{7}\right)^k, \quad \text{and} \quad \tilde{x}_k = \left(\frac{1}{2}\right)^{2^k}.$$

Explain the results observed in the previous example by studying the

order of convergence in each case. Justify your answers.

Answer.

Graphical study of convergence

Study of e_k versus k

We want to observe the convergence graphically. Let us first plot e_k versus k.

Do it yourself. Run the following cell to plot e_k versus k for the four sequences, and comment on the obtained picture.

```
K = np.arange(0,15,1)
err1 = 1./(K+1)
err2 = (1./2) ** K
err3 = (1./7) ** K
err4 = (1./2) ** (2**K)

fig = plt.figure(figsize=(12, 8))
plt.plot(K, err1, marker="o", label='error for $x_k$')
plt.plot(K, err2, marker="o", label=r'error for $\bar{x}_k$')
## the r in the label before the '' allows to display latex symbols
plt.plot(K, err3, marker="o", label=r'error for $\hat{x}_k$')
plt.plot(K, err4, marker="o", label=r'error for $\tilde{x}_k$')
plt.legend(loc='upper right', fontsize=18)
plt.xlabel('k', fontsize=18)
plt.ylabel('$e_{k}$', fontsize=18)
plt.title('Convergence', fontsize=18)
```

Answer.

Study of $\log(e_k)$ versus k

The $\log$ function is of great help to better understand the behavior of the error.

For example, since $x \to \log(x)$ is an increasing function with derivative going to infinity when x goes to zero, it allows to "zoom" on the smallest values of the error, and plotting $\log(e_k)$ versus k can allows us to check

that the error is still decreasing and not stagnating for big values of k (which can not be affirmed using the previous plot).

> Do it yourself.
> Modify the following cell to plot the error versus k in log-scale. More precisely, use the plt.yscale command to modify the scale for the y-axis. This will allow you to plot $\log(e_k)$ versus k, while keeping the values of e_k itself on the y axis. *In order to learn how to use this command, you can either type* `help(plt.yscale)` *or* `plt.yscale?` *in a code cell, or look for "matplotlib.pyplot.yscale" on the internet.*
>
> Comment on the obtained picture. In particular, what information can you deduce from the fact that some of the curves are lines? *You may want to comment out some of the errors, and to use larger values of k, in order to better determine if some curves are line or not.*

```
K = np.arange(0,7,1)
err1 = 1./(K+1)
err2 = (1./2) ** K
err3 = (1./7) ** K
err4 = (1./2) ** (2**K)

fig = plt.figure(figsize=(12, 8))
plt.plot(K, err1, marker="o", label='error for $x_k$')
plt.plot(K, err2, marker="o", label=r'error for $\bar{x}_k$')
# plt.plot(K, err3, marker="o", label=r'error for $\hat{x}_k$')
# plt.plot(K, err4, marker="o", label=r'error for $\tilde{x}_k$')
plt.legend(fontsize=18)
plt.yscale('log')
plt.xlabel('k', fontsize=18)
plt.ylabel('$e_{k}$', fontsize=18)
plt.title('Convergence', fontsize=18)
```

> Answer.

> Do it yourself.
> For the curves which seem to be lines, use the polyfit function from numpy in order to approximate the slope. *The function polyfit tries to best approximate (in a sense that will be discussed in the next chapter) a given set of points by a polynomial. If you ask it to find a polynomial of degree 1, you will therefore be able to recover the slope. Again, do not hesitate to use the help function or to look out for "numpy polyfit" on the internet in order to better understand how to use this function.*

Using the slope computed using polyfit, try to recover the rate of convergence for $\bar{x}_k$ and $\hat{x}_k$ from the data.

Answer.

Study of $\log(e_{k+1})$ *versus* $\log(e_k)$

In the above plot of $\log(e_k)$ versus k, we could recover the order of converge for some of the sequence, but only because this order was exactly equal to 1. In general, one has to use yet another scale in order to graphically find the order of convergence. Indeed, assume that the error is such that

$$e_{k+1} \approx C e_k^{\alpha},$$

for some unknown C and α. Then, we get

$$\log e_{k+1} \approx \log\left(C e_k^{\alpha}\right) = \alpha \log e_k + \log C.$$

As a consequence, the order of convergence can be graphically observed by plotting $\log e_{k+1}$ versus $\log e_k$ and finding the slope.

Do it yourself. Run the following cell and explain the resulting plot. You can add some plots to confirm the slopes of the lines.

```
K = np.arange(0,6,1)
x1 = 1./(K+1)
x2 = (1./2) ** K
x3 = (1./7) ** K
x4 = (1./2) ** (2**K)

fig = plt.figure(figsize=(12, 8))

plt.loglog(x1[:-1:], x1[1:], marker="o", label='curve for $x_k$') #
plt.loglog(x2[:-1:], x2[1:], marker="o", label=r'curve for $\bar x_
plt.loglog(x3[:-1:], x3[1:], marker="o", label=r'curve for $\hat x_
plt.loglog(x4[:-1:], x4[1:], marker="o", label=r'curve for $\tilde
plt.loglog(x3[:-1:],x3[:-1:],'--k',label='slope 1')
plt.loglog(x4[:-1:],x4[:-1:]**2,':k',label='slope 2')

## Instead of using plt.loglog everytime, we can also keep using pl
## scale of the axes at the end with plt.xscale('log') and plt.ysca

# plt.plot(x1[:-1:], x1[1:], marker="o", label='curve for $x_k$')
# plt.plot(x2[:-1:], x2[1:], marker="o", label=r'curve for $\bar x_
# plt.plot(x3[:-1:], x3[1:], marker="o", label=r'curve for $\hat x_
# plt.plot(x4[:-1:], x4[1:], marker="o", label=r'curve for $\tilde
# plt.plot(x3[:-1:],x3[:-1:],'--k',label='slope 1')
# plt.plot(x4[:-1:],x4[:-1:]**2,':k',label='slope 2')
# ## We now use log-scale in both directions
# plt.xscale('log')
# plt.yscale('log')

plt.legend(loc='lower right', fontsize=18)
plt.xlabel('$e_k$', fontsize=18)
plt.ylabel('$e_{k+1}$', fontsize=18)
plt.title('Order of convergence', fontsize=18)

```

Answer.

Remark.

We obtained the order of convergence visually, by comparing the error curves with some lines of known slope (here 1 and 2), but we could also have computed the slope as in the previous example, using polyfit, to find the order of convergence. This would work well except maybe for x_k, for which we do not get a line in log-log scale. This is because the convergence is sublinear for x_k, in which case one should use yet another scale to better analyze the convergence. In the remainder of this chapter we mostly deal with linear and quadratic convergence, but we will come back to sublinear convergence later on in the course.

Error bound

To finish, let us emphasize that, in real problems x^* is usually not known, and therefore we cannot compute the value of the true error at step k. Instead we try to find a (computable) bound for the error, which gives us a "worst-case" error.

> Definition.
> **Error bound**. Suppose that a sequence $(x_k)_k$ is generated to approximate x^*. The sequence $(\beta_k)_k$ is an error bound if
> $$e_k \leq \beta_k \text{ for all } k.$$

If the error bound $\beta_k \to 0$ when $k \to \infty$, we obtain that

- the sequence x_k converges to x^*
- the error goes to zero at least as fast as the sequence β_k.

One has to take care that an estimator only provides an upper bound on the error. As a consequence, the error can go to zero faster than the estimator.

> Remark (to go further).
> In practice, it is highly desirable (but not always possible) to obtain an error bound that is *computable*. For instance, while knowing that there exists a constant $c > 0$ such that $e_k \leq \frac{c}{k^2}$ does provide some measure of information as to how x_k converges to x^*, if we want to explicitly bound the error e_k for a given value of k such an error bound is essentially useless. On the other hand, if we know that $e_k \leq \frac{10}{k^2}$ for instance, then we do know that, for $k = 100$, the error between x_k and x^* is *at most* of 10^{-3}.

The bisection method

The first method to approximate a solution to $f(x) = 0$ is based on the Intermediate Value Theorem (see Appendix). Suppose f is a continuous function on the interval $[a, b]$ and that $f(a)$ and $f(b)$ have opposite signs:

$f(a)\, f(b) \leq 0$. Then, there exists x^* in (a, b) such that $f(x^*) = 0$.

Starting from an interval $I_0 = [a_0, b_0]$ such that $f(a_0)\, f(b_0) \leq 0$, such an x^* is approximated as follows. Let x_0 be the midpoint of I_0:

$$x_0 = \frac{a_0 + b_0}{2}.$$

Then, the bisection method iterates by choosing $I_1 = [a_1, b_1]$ and x_1 as follows:

- if $f(a_0)\, f(x_0) \leq 0$ then there exists a zero of f in $[a_0, x_0]$, so we set

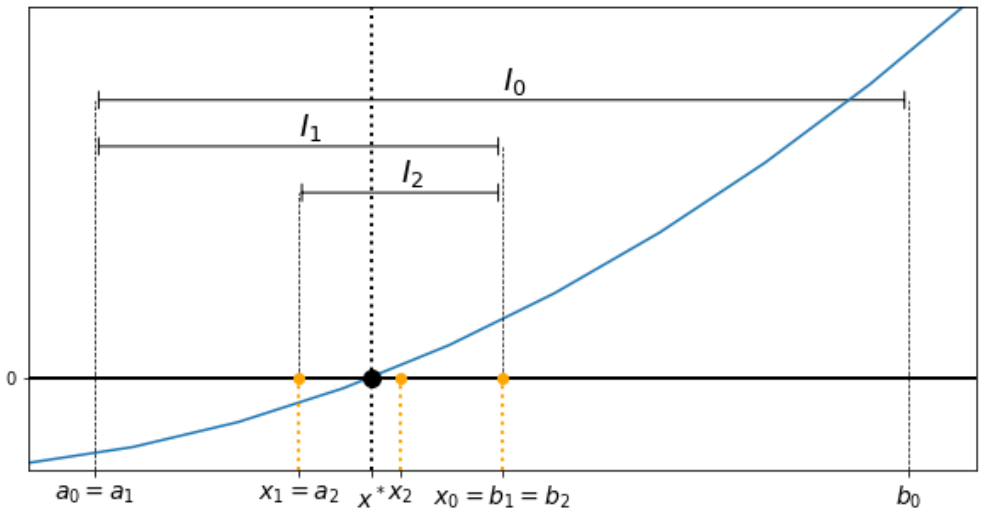
$$a_1 = a_0, \quad b_1 = x_0 \quad \text{and} \quad x_1 = \frac{a_1 + b_1}{2},$$

- else, we must have $f(x_0)\, f(b_0) \leq 0$ and there exists a zero of f in $[x_0, b_0]$, so we set

$$a_1 = x_0, \quad b_1 = b_0 \quad \text{and} \quad x_1 = \frac{a_1 + b_1}{2}.$$

As such, the bisection method generates a sequence x_k, and we will prove that this sequence does converge to a zero of f. However, in practice an algorithm can only do finitely many operations, which means we must add a so-called *stopping criterion* to tell the algorithm when to stop. This will be discussed in more details below.

An example of the first two iterations is illustrated on an example in the figure below.



The bisection method leads to the following algorithm:

Algorithm.

Bisection method. Computes an approximate solution x to $f(x) = 0$.

$$
\begin{aligned}
INPUT: &\quad f, a, b \\
DO: &\quad x = (a+b)/2 \\
&\quad \text{While stopping criterion is not met do} \\
&\qquad \text{If} \quad f(a)\,f(x) \leq 0, \quad b = x \quad \text{else} \quad a = x \\
&\qquad x = (a+b)/2 \\
&\quad \text{end while} \\
RETURN: &\quad x
\end{aligned}
$$

Remark.

The above cell is meant to provide an informal description of the bisection method, and to convey the overall structure of the algorithm, but there may be slight differences in the actual implementation below. In particular, in order to better study the properties of the bissection method, we are going to implement a version that returns all the values of x computed at each step, and not only the last one.

Stopping criterion

For any kind of iterative method, we would like to stop the algorithm only when the sequence x_k is *close enough to having converged*. Of course, without knowing what the limit x^* is, it is impossible to know for sure how close to the limit we are, so we have to make some kind of guess. A usual choice is to fix a tolerance ε, and to stop the algorithm once

$$|x_k - x_{k-1}| \leq \varepsilon \qquad \text{or} \qquad \frac{|x_k - x_{k-1}|}{|x_k|} \leq \varepsilon.$$

Since we are trying to find a zero of f, another natural possibility is to stop the algorithm once

$$|f(x_k)| \leq \varepsilon.$$

Notice that, if x_k does converge to a zero x^* of f, and if $f'(x^*) \neq 0$, then

$$|f(x_k)| \sim |f'(x^*)||x_k - x^*|,$$

and the stopping criterion is related to a criterion on $e_k = |x_k - x^*|$. However, for this to hold we must already know that x_k is close enough to x^*, so it is hard to make use of this information in practice.

In the absence of an error estimate (see below), one usually has to make do with such criteria, but it is important to keep in mind that, even for ε very small, neither of those them guarantees that x_k is close to x^*.

> Remark.
> In practice, whatever the chosen stopping criterion, it is important to add an extra one: the algorithm must stop if the number k of iteration reaches some prescribed threshold k_{max}. This is a safety net to ensure that the algorithm does not end up running forever, even if the initial stopping criterion never ends up being satisfied.

Example

We now implement the bisection method and test it to approximate x^*, the unique solution in $\mathbb{R}$ to $f(x) = x^3 - 2 = 0$.

> Do it yourself. Complete the following function encoding f.

```
## Function f: x -> x^3 -2

def ftest(x):
```

> Do it yourself. Complete the following function, so that it computes the sequence generated using the bisection algorithm for a given function f and initial interval $[a_0, b_0]$.
>
> The algorithm terminates when the stopping criterion of your choice is satisfied, or when a given maximal number k_{max} of iterations have been achieved. The output is a vector x containing the x_k which have been computed.

```python
def Bisection(f, a0, b0, k_max, eps):
    """
    Bisection algorithm with stopping criterion based on |f(x_k)|
    ----------------------
    Inputs:
    f : name of the function
    a0, b0 : bounds of the initial interval I_0=[a_0,b_0] with f(a_
    k_max : maximal number of iterations
    eps : tolerance for the stopping criterion

    Output:
    x = the sequence x_k of mipoints of I_k
    """
    ## We first check that the initial interval is guaranteed to co
    if f(a0)*f(b0) > 0:
        print("The inputs do not satisfy the assumptions of the bis
        return

    x = ...                 # create vector x of zeros with size k_ma
    k = 0                   # initialize k
    a = a0                  # initialize a
    b = b0                  # initialize b
    x[0] = ...              # initialize x_0
    while  ...  and  ... :  #stopping criterion and "safety conditi
        if ... :
            ...  # do something
        else:
            ...  # do something esle
        k = k+1
        x[k] = ...  # compute and store the new iterate
```

Do it yourself. Test the bisection method to compute $x^* = 2^{1/3}$ solution to $f(x) = 0$. Initialize with $[a_0, b_0] = [1, 2]$ and select values for the maximal number of iterations k_{max} and the tolerance ε. Plot the error e_k versus k. Use a log scale for the error (y-axis). Do not forget to add a title to the figure and labels to the axes (see the graphical study in the previous section for an example).

```python
# Test for f(x)=x^3-2 on I=[1,2]

xstar = 2**(1.0/3)

# parameters
a0 = ...
b0 = ...
k_max = ...
eps = ...

# compute the iterations of the bisection method for I0=[1,2]
x = ...

#print x^* and the iterations stored in x
print('xstar =', xstar)
print('x =', x)

# compute the error
# err is a vector, err[k] = |x[k]-x^*|
err = ...

# create the vector tabk : tabk[k] = k for each iteration made
tabk = ...

# plot the error versus k
fig = plt.figure(figsize=(12, 8))
plt.plot(..., ..., marker="o")
# set log scale for the error (y-axis)
...
# set title of the figure and labels of the axis
...

```

> Do it yourself. Comment on the previous plot.

> Answer.

Error bound and stopping criterion

The above experiment is encouraging, but we are only confident about the result because we knew x^* in advance. In order to use a more precise stopping criterion, related to the true error, in situations where x^* is not know, we need more information about the way the sequence converges to x^*. To do so, error bounds are very useful. For the bisection method, we have the following result.

Proposition.

Convergence of the bisection method. Let f be a continuous function on $[a, b]$ with $f(a)\, f(b) \leq 0$. Suppose $(x_k)_k$ is the sequence generated by the bisection method.

Then, the sequence $(x_k)_k$ converges to a zero x^* of f, and the following estimation holds:

$$\forall\, k \geq 0, \quad |x_k - x^*| \leq \frac{b-a}{2^k}.$$

Proof. By definition of the bisection method, the sequence $(a_k)_k$ is non-decreasing and bounded above (by b_0) while the sequence $(b_k)_k$ is non-increasing and bounded below (by a_0). Therefore $(a_k)_k$ converges, say to a^*, and $(b_k)_k$ converges, say to b^*. Besides, since the interval I_k is divided by 2 at each step of the method, we have

$$\forall\, k \geq 0 \quad |b_k - a_k| = \frac{b_0 - a_0}{2^k}.$$

In particular, taking the limit $k \to \infty$, this implies that $a^* = b^*$, and we define $x^* = a^* = b^*$. From the intermediate value theorem (see Appendix), we know that for each k there exists a zero x_k^* of f in $I_k = [a_k, b_k]$, that is, $a_k \leq x_k^* \leq b_k$. Taking once more the limit $k \to \infty$ we get that $(x_k^*)_k$ converges to x^*, and since $f(x_k^*) = 0$ for all k, we get $f(x^*) = 0$ by continuity of f. Therefore x^* is indeed a zero of f.

By construction x^* belongs to $I_k = [a_k, b_k]$ for each k, but so does x_k, which means that

$$\forall\, k \geq 0 \quad |x_k - x^*| \leq |b_k - a_k| \leq \frac{b-a}{2^k}.$$

This proves the convergence of x_k to x^* and provides the requested estimation.

Remark. The bisection method is said to be *globally convergent*. Indeed, the initialization of a and b doesn't need to be close to x^*. Whatever the choice for these parameters is, the generated sequence will converge to a zero x^* of f, provided that $f(a)\,f(b) \leq 0$.

This proposition provides a new stopping criterion. Indeed, we have just found an error estimator:

$$\forall\, k \geq 0,\ e_k \leq \beta_k \qquad \text{where} \qquad \beta_k = \frac{b-a}{2^k},$$

which is computable. This means that we can **guarantee** that the error is below some prescribed tolerance ε as soon as

$$\frac{b-a}{2^k} \leq \varepsilon.$$

Do it yourself. Rewrite the bisection algorithm so that it terminates when the stopping criterion $\frac{b-a}{2^k} \leq \varepsilon$, or when a maximal number k_{max} of iterations have been made.

```
def Bisection2(f, a0, b0, k_max, eps):
    """
    Bisection algorithm with stopping criterion based on the error
    ----------------------
    Inputs:
    f : name of the function
    a0, b0 : bounds of the initial interval I_0=[a_0,b_0] with f(a_
    k_max : maximal number of iterations
    eps : tolerance for the stopping criterion

    Output:
    x = the sequence x_k of mipoints of I_k
    """
    ...
```

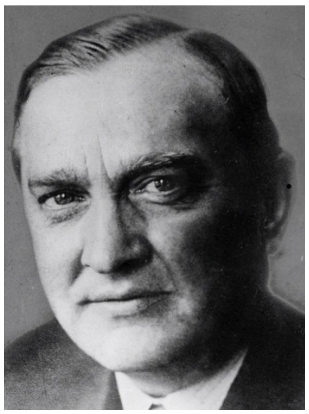
Do it yourself. Test this new algorithm on the same example as before. Plot on the same figure the error versus k and the corresponding error bound (in an appropriate scale). Do not forget the title, the labels of the axes and the legend.

```python
# Test for f(x)=x^3-2 on I=[1,2]

xstar = 2**(1.0/3)

# parameters
a0 = ...
b0 = ...
k_max = ...
eps = ...

# compute the iterations of the bisection method for I0=[1,2], with
x = ...

#print x^* and x
print('xstar =', xstar)
print('x =', x)

# create the vector tabk : tabk[k] = k for each iteration made
tabk = ...

# compute the error and the error bound
err = ...
errEstim = ...

# plot the error versus k, and the error bound versus k with the y-
fig = plt.figure(figsize=(12, 8))
...
```

Do it yourself. Comment the previous plot. What is the order of convergence of the error bound?

Answer.

Fixed point iterations

Luitzen Egbertus Jan Brouwer (1881 – 1966) and Stefan Banach (1892-1945). Brouwer is a Dutch mathematician and philosopher. He proved a lot of results in topology. One of his main theorem is his fixed point theorem (1909). One of its simpler form says that a continuous function from an interval to itself has a fixed point. The proof of the theorem does not provide a method to compute the corresponding fixed point. Among lots of other fixed point results, Brouwer's theorem became very famous because of its use in various fields of mathematics or in economics. In 1922, a polish mathematician, Stefan Banach, stated a contraction mapping theorem, proving in some case the existence of a unique fixed point and providing a constructive iterative method to approximate this fixed point. Banach is one of the founders of modern analysis and is often considered as one of the most important mathematicians of the 20-th century.

A fixed point for a function g is an x such that $g(x) = x$. In this section we consider the problem of finding solutions of fixed point problems. This kind of problem is equivalent to a zero-finding problem in the following sense:

- If x^* is a solution to $f(x) = 0$, we can find a function g such that x^* is a fixed point of g. For example, one can choose $g(x) = f(x) + x$.
- If x^* is a solution to $g(x) = x$, then, x^* is also a solution to $f(x) = 0$ where $f(x) = g(x) - x$.

While a fixed point problem and a zero-finding problem can be equivalent in the sense that they have the same solutions, each of them can be

analyzed or approximated with different techniques. In the previous section we introduced the bisection method which can be used for a zero-finding problem. In this section, we focus on fixed point problems, and then use them to solve zero-finding problems. In the following, functions f will be used for root-finding problems and g for corresponding fixed point problems.

First, note that, given a function f, the choice of g is not unique. For example, for any function g of the form $g(x) = G(f(x)) + x$, where $G(0) = 0$ and $G(x) \neq 0$ for $x \neq 0$, the fixed points of g are in one-to-one correspondence with the zeros of f.

Example.
Let us consider again the problem of computing an approximation of $x^* = 2^{1/3}$ as the root of $f(x) = x^3 - 2$. You can check that, for each of five following functions, a fixed point corresponds to a zero of f.

- $g_1(x) = x^3 - 2 + x$
- $g_2(x) = \sqrt{\dfrac{x^5 + x^3 - 2}{2}}$
- $g_3(x) = -\dfrac{1}{3}(x^3 - 2) + x$
- $g_4(x) = -\dfrac{1}{20}(x^3 - 2) + x$
- $g_5(x) = \dfrac{2}{3}x + \dfrac{2}{3x^2}$

From a numerical point a view, solutions to fixed point problems can be approximated by choosing an initial guess x_0 for x^* and generating a sequence by iterating the function g:

$$x_{k+1} = g(x_k), \quad \text{for} \quad k \geq 0.$$

Indeed, suppose that g is continuous and that the sequence $(x_k)_k$ converges to x_∞, then, passing to the limit in the previous equation gives

$$x_\infty = g(x_\infty),$$

so the limit x_∞ must be a fixed point of g. This leads to the following algorithm:

Algorithm.

Fixed point iterations method. Iterates a map g to try to approximate a fixed point x^* of g such that $g(x^*) = x^*$.

$$
\begin{aligned}
INPUT: \quad & g, x0 \\
DO: \quad & x = x0 \\
& \text{While stopping criterion is not achieved do} \\
& \qquad x = g(x) \\
& \text{end while} \\
RETURN: \quad & x
\end{aligned}
$$

Now, for a given function g, one has to answer the following questions:

- Does g have a fixed point ?
- Does the sequence generated using the fixed point iterations converge ?
- If the sequence converges, how fast does it converge ?

Graphical investigation

In order to better understand the behavior of fixed point iterations, one can try to visualize them on a graph.

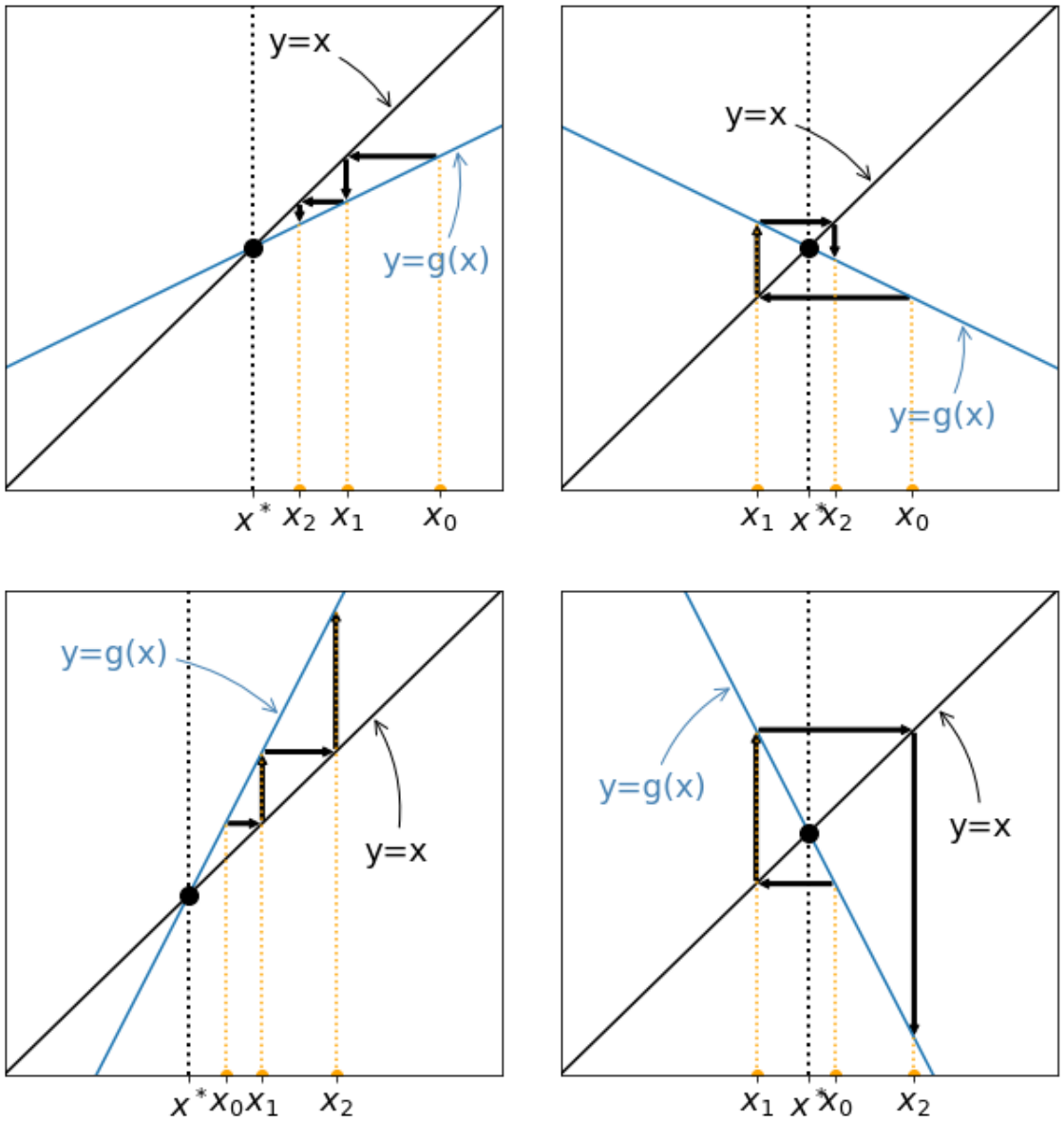
First, the fixed point of a function g can be found graphically, by searching for the intersection between the graph of g and the graph of the function $\phi(x) = x$.

Then, suppose x_0 is given and place it on the x-axis. To place $x_1 = g(x_1)$ on the same axis, proceed as follows:

- from $(x_0, 0)$, go up to find the point $(x_0, g(x_0)) = (x_0, x_1)$, when crossing the graph of g,
- from (x_0, x_1) move horizontally to find the point (x_1, x_1), when crossing the graph of ϕ,

- finally, go back down towards the x-axis to place the point $(x_1, 0)$.

Iterate the procedure get the successive points x_k. Four examples are given below:



Cases with increasing functions g are given on the left and leads to monotonous sequences. On the contrary, oscillating sequences are generated for non increasing functions g (right). For the two examples given at the top x_k seems to converge, but for the two examples given at the bottom it seems to diverge. Let's try to understand why.

Existence, uniqueness and convergence analysis

Theorem.

Existence of a fixed point. Let $g : [a, b] \to [a, b]$ be a continuous function. Then, g has a fixed point in $[a, b]$:

$$\exists x^* \in [a, b], \quad g(x^*) = x^*.$$

Do it yourself. Complete the proof of the previous theorem.

Proof.

Theorem.

Existence of a unique fixed point, and convergence of the iterates. Let $g : [a, b] \to [a, b]$ be a continuous function. Assume

- g is a contraction on $[a, b]$, that is,

$$\exists K < 1 \quad \text{such that} \quad \forall x, y \in [a, b], \quad |g(x) - g(y)| \leq K|x - y|.$$

Then, g has a unique fixed point in $[a, b]$:

$$\exists! x^* \in [a, b], \quad g(x^*) = x^*.$$

Besides, the sequence defined by $x_{k+1} = g(x_k)$ converges to x^* for any choice of $x_0 \in [a, b]$. Moreover we have

$$\forall\, k \geq 0, \quad |x_{k+1} - x^*| \leq K|x_k - x^*|,$$

so that the convergence is at least linear.

Proof. The existence of a fixed point x^* is given by the previous theorem. The fact that g is a contraction ensures the uniqueness of the fixed point. Indeed, if y^* is a fixed point of g, then

$$|g(x^*) - g(y^*)| \leq K|x^* - y^*|,$$

but x^* and y^* are fixed points of g so we get

$$|x^* - y^*| \leq K|x^* - y^*|,$$

and we must have $y^* = x^*$ because $K < 1$. The fixed point is unique.

In order to get the convergence estimate for (x_k) we repeat the same argument with x_k in place of y^*:

$$|g(x_k) - g(x^*)| \leq K|x_k - x^*|,$$

and since $x_{k+1} = g(x_k)$ this gives the announced estimates, which implies the convergence of $(x_k)_k$ to x^* because $K < 1$.

Remark.

- Similarly to what we had for the bisection method, the above theorem gives a *global* convergence result: for any initial condition x_0 in $[a, b]$, the sequence is guaranteed to converge to a fixed point of g. However, this comes at the cost of a rather strong assumption, namely that g is a contraction on $[a, b]$.
- Notice that, if g is differentiable on $[a, b]$, then the contraction hypothesis is equivalent to assuming that $|g'(x)| \leq K$ for all x in $[a, b]$ (one implication is obtained by letting y go to x, and the other comes from Taylor-Lagrange's formula).
- This assumption can be relaxed, but then the convergence result becomes local: if x_k is sufficiently close to a fixed point x^*, then the

behavior of x_{k+1} depends only on whether $|g'(x^*)|$ is smaller or larger than 1. This is made precise in the following theorem.

Theorem.

Local convergence/divergence for fixed point iterations. Let $g : (a, b) \to \mathbb{R}$ be a continuous function, having a fixed point x^* and such that g is differentiable at x^*. Consider the sequence $x_{k+1} = g(x_k)$ for $k \geq 0$, x_0 being given.

- If $|g'(x^*)| < 1$, there exists $\eta > 0$ such that, if $x_0 \in I_\eta = [x^* - \eta, x^* + \eta]$, then $(x_k)_k$ converges to x^*, and the convergence is at least linear. If $g'(x^*) \neq 0$, the convergence is exactly linear , and the rate of convergence is $|g'(x^*)|$.
- If $|g'(x^*)| > 1$, there exists $\eta > 0$ such that, if $x_0 \in I_\eta = [x^* - \eta, x^* + \eta] \setminus \{x^*\}$, then eventually x_k gets out of I_η.

Proof. We first consider the case $|g'(x^*)| < 1$. The Taylor-Young expansion of order 1 at x^* writes

$$g(x_k) = g(x^*) + g'(x^*)(x_k - x^*) + o(x_k - x^*),$$

or equivalently

$$x_{k+1} - x^* = g'(x^*)(x_k - x^*) + (x_k - x^*)\epsilon(x_k - x^*),$$

where the function ϵ goes to 0 at 0. Since $|g'(x^*)| < 1$, there exists $\alpha > 0$ such that $|g'(x^*)| + \alpha < 1$. Next, we consider $\eta > 0$ such that, if $x_k \in I_\eta = [x^* - \eta, x^* + \eta]$, $|\epsilon(x_k - x^*)| \leq \alpha$. This implies that, if $x_k \in I_\eta$,

$$\begin{aligned} |x_{k+1} - x^*| &= |g'(x^*) + \epsilon(x_k - x^*)||x_k - x^*| \\ &\leq \left(|g'(x^*)| + \alpha\right)|x_k - x^*|. \end{aligned}$$

Since $|g'(x^*)| + \alpha \leq 1$, $x_{k+1} \in I_\eta$. By induction, all subsequent iterates will stay in I_η, and because $|g'(x^*)| + \alpha < 1$, $(x_k)_k$ converges to x^*.

Finally, going back to $x_{k+1} - x^* = g'(x^*)(x_k - x^*) + o(x_k - x^*)$, we see that, if $g'(x^*) \neq 0$,

$$|x_{k+1} - x^*| \sim |g'(x^*)||x_k - x^*|,$$

and the convergence is indeed linear with rate $|g'(x^*)|$.

We now consider the case $|g'(x^*)| > 1$. This time, we can take $\alpha > 0$ such that $|g'(x^*)| - \alpha > 1$, and $\eta > 0$ such that, if $x_k \in I_\eta = [x^* - \eta, x^* + \eta]$, $|\epsilon(x_k - x^*)| \leq \alpha$. This implies that, if $x_k \in I_\eta$,

$$\begin{aligned} |x_{k+1} - x^*| &= |g'(x^*) + \epsilon(x_k - x^*)||x_k - x^*| \\ &\geq \left(|g'(x^*)| - \alpha\right)|x_k - x^*|. \end{aligned}$$

As long as the iterates belong to I_η, this estimate holds, and at each iteration $|x_k - x^*|$ is amplified by at least $|g'(x^*)| - \alpha > 1$. Therefore, as soon as $x_0 \neq x^*$, after a finite number of iteration $|x_k - x^*|$ will become greater than η.

Remark.

- Notice that the above theorem does not tell us anything about the case $|g'(x^*)| = 1$. Indeed, we will see later that both behavior (convergence or divergence) are possible in that case.
- We have just shown that, when it comes to the convergence, the smaller $|g'(x^*)|$, the better, at least while $|g'(x^*)| > 0$. In the next theorem, we show that this is actually true up to $|g'(x^*)| = 0$, because in that case the convergence is at least quadratic (order 2 or more).

Theorem.

"Better than linear" speed of convergence of fixed point iterations. Let $g : (a, b) \to \mathbb{R}$ be a continuous function, having a fixed point x^* and such that g is $p + 1$-times differentiable on a neighborhood of x^*, for some integer $p \geq 1$. Consider the sequence $x_{k+1} = g(x_k)$ for $k \geq 0$, x_0 being given. Suppose

- $g^{(q)}(x^*) = 0$ for $q = 1, \dots, p$.

Then, there exists $\eta > 0$ such that, if $x_0 \in I_\eta = [x^* - \eta, x^* + \eta]$, then $(x_k)_k$ converges to x^*, and the convergence is at least of order $p + 1$. If $g^{(p+1)}(x^*) \neq 0$, the convergence is exactly of order $p + 1$.

Proof. [★] The proof is very similar to the one done just above in the case $|g'(x^*)| < 1$. The only difference is that we do a Taylor-Young expansion of g around x^* at a higher order, namely $p + 1$, since all the lower order terms vanish by assumption, which yields

$$x_{k+1} - x^* = \frac{g^{(p+1)}(x^*)}{(p+1)!}(x_k - x^*)^{p+1} + o\left((x_k - x^*)^{p+1}\right).$$

The remainder of the proof follows as above, and we omit the details.

Stopping criterion

For fixed points problems, a natural stopping criterion is to ask that $|x_{k+1} - x_k|$ becomes less than some tolerance ε. Indeed, in that case $x_{k+1} = g(x_k)$ and therefore we are asking for $g(x_k) - x_k$ to be close to 0, i.e. x_k is an approximate fixed point. Of course, as mentioned previously, without some error bound we cannot certify that such a stopping criterion yields a solution which is actually close to a fixed point. However, let us mention that, if x_k does converge to a fixed point x^*, then using once more Taylor-Young's formula we see that

$$\begin{aligned} x_{k+1} - x_k &= g(x_k) - g(x^*) + x^* - x_k \\ &= (1 - g'(x^*))(x^* - x_k) + o(x^* - x_k), \end{aligned}$$

and therefore $|x_{k+1} - x_k|$ gives a good control on the error $|x_k - x^*|$, as long as $g'(x^*)$ is not too close to 1.

Numerical tests

> Do it yourself. Complete the following function. It shall compute the sequence generated using the fixed point algorithm for a given function g. The algorithm terminates when $|g(x_k) - x_k|$ is below some tolerance ε, or when a given number k_{max} of iterations have been made.

```python
def FixedPoint(g, x0, k_max, eps):
    """
    Fixed point algorithm x_{k+1} = g(x_k)
    ----------------------
    Inputs:
    g : name of the function
    x0 : initial point
    k_max : maximal number of iterations
    eps : tolerance for the stopping criterion

    Output:
    x = the sequence x_k
    """
    x = ...   # create a vector x of zeros with size k_max+1
    k = 0
    x[0] = x0
    while  ...  and  ... : #stopping criterion and "safety conditic
        ...
        ...
```

Do it yourself.

We consider again the 5 functions g_i, $i = 1, \dots, 5$, proposed at the beginning of the section to compute $x^* = 2^{1/3}$. Run the following cells to observe the behavior of the algorithm for these 5 functions. Try also to change the initial point x_0. Comment on the observed results in light of the previous theorems. *You may want to start with g_3 and g_4, before turning your attention to the other cases.*

```python
```

- $g_1(x) = x^3 - 2 + x$

```python
def g1(x):
    return x**3 - 2 + x

x0 = xstar + 0.001
#x0 = xstar - 0.001

k_max = 20
eps = 1e-5

x1 = FixedPoint(g1, x0, k_max, eps)
print('xstar =', xstar)
```

Answer.

$\sqrt{x^5 + x^3 - 2}$

```python
def g2(x):
    return np.sqrt( (x**5 + x**3 - 2) / 2 )

x0 = xstar - 0.001
#x0 = xstar + 0.001

k_max = 20
eps = 1e-5

x2 = FixedPoint(g2, x0, k_max, eps)
print('xstar =', xstar)
```

Answer.

- $g_3(x) = -\frac{1}{3}(x^3 - 2) + x$

```python
def g3(x):
    return - (x**3-2)/3 + x

x0 = xstar + 1
# x0 = xstar + 2

k_max = 20
eps = 1e-5

x3 = FixedPoint(g3, x0, k_max, eps)
print('xstar =', xstar)
print('x =', x3)

err3 = abs(x3-xstar)
```

Answer.

- $g_4(x) = -\frac{1}{20}(x^3 - 2) + x$

```python
def g4(x):
    return - (x**3-2)/20 + x

x0 = xstar + 1
# x0 = xstar + 5

k_max = 20
eps = 1e-5

x4 = FixedPoint(g4, x0, k_max, eps)
print('xstar =', xstar)
print('x =', x4)

err4 = abs(x4-xstar)
```

Answer.

- $g_5(x) = \frac{2}{3}x + \frac{2}{3x^2}$

```python
def g5(x):
    return 2*x/3 + 2/(3*x**2)

x0 = xstar + 1

k_max = 20
eps = 1e-5

x5 = FixedPoint(g5, x0, k_max, eps)
print('xstar =', xstar)
print('x =', x5)

err5 = abs(x5-xstar)
```

Answer.

Do it yourself. Compare graphically the convergence of the sequences obtained with g_3, g_4, and g_5:

- On a single figure, plot the three errors versus k with log-scale for the error.
- On a second figure, plot the e_{k+1} versus e_k in log-log scale for the three methods.

Do not forget titles, labels and legends.

```
# initialization
x0 = xstar + 1

# parameters for the algorithms
k_max = 100
eps = 1e-8

# computation of the iterates
x3 = FixedPoint(g3, x0, k_max, eps)
x4 = FixedPoint(g4, x0, k_max, eps)
x5 = FixedPoint(g5, x0, k_max, eps)

# computation of the errors
err3 = abs(x3-xstar)
err4 = abs(x4-xstar)
err5 = abs(x5-xstar)

# the index of each iterate stopping at the appropriate value in ea
tabk3 = ...
tabk4 = ...
tabk5 = np.arange(0, err5.size, dtype='float')

fig = plt.figure(figsize=(20, 10))

plt.subplot(121) # plot of e_k versus k for the three cases
plt.plot(..., ...)
plt.plot(..., ...)
plt.plot(..., ...)
...

plt.subplot(122) # plot of log e_{k+1} versus log e_{k} for the thr
plt.plot(..., ...)
plt.plot(..., ...)
plt.plot(..., ...)
...

```

Do it yourself. Comment on the previous figures.

Answer.

Do it yourself. Run the two following cells to test the fixed point algorithm for the functions:

- $g_6(x) = x - x^3$
- $g_7(x) = x + x^3$

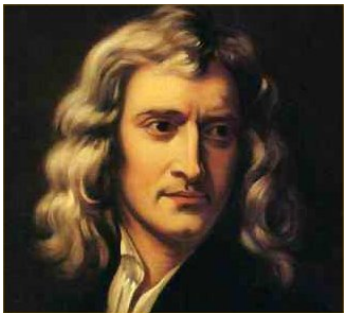
What can you conclude for the case $|g'(x^*)| = 1$?

```python
def g6(x):
    return x - x**3

xstar = 0

k_max = 20
eps = 1e-10
x0 = 0.1

x6 = FixedPoint(g6, x0, k_max, eps)
print('xstar =', xstar)
```

```python
def g7(x):
    return x + x**3

xstar = 0

k_max = 20
eps = 1e-10
x0 = 0.1

x7 = FixedPoint(g7, x0, k_max, eps)
print('xstar =', xstar)
```

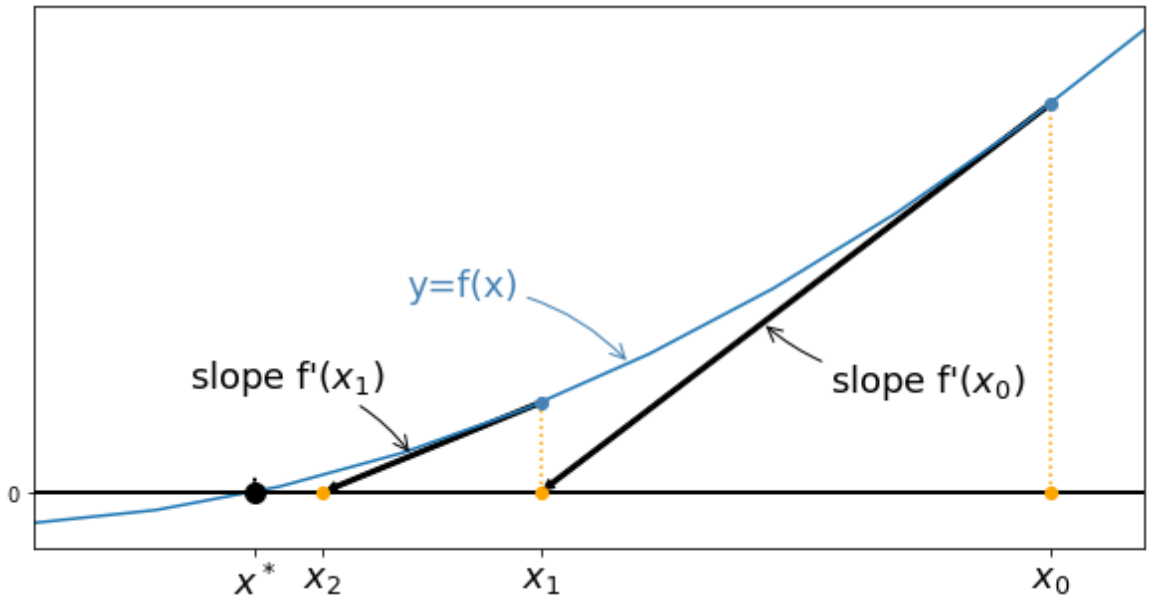
Answer.

The Newton-Raphson method



Isaac Newton (1643 – 1727). English mathematician, astronomer, theologian, author and physicist, Isaac Newton is known as one of the most important scientists. He made breaking contributions to classical mechanics, optic and also contributed to infinitesimal calculus. In particular, he described in 1671 a method to find zeros of polynomials, which was only published in 1736, and was the basis for what is now known as the Newton-Raphson method. Indeed, it was first published (with a reference to Newton) by another English mathematician, Joseph Raphson in 1690. Both of them focused only on zeros of polynomial functions, but the basis of the general method was already present in their works.

The Newton-Raphson (or simply Newton's) method is one of the most powerful and well-known method to solve $f(x) = 0$ problems. The simplest way to describe it is to see it as a graphical procedure: x_{k+1} is computed as the intersection with the x-axis of the tangent line to the graph of f at point $(x_k, f(x_k))$.



For a given x_k, the equation of the tangent line at $(x_k, f(x_k))$ is

$$y = f(x_k) + f'(x_k)(x - x_k),$$

therefore x_{k+1} is defined by

$$0 = f(x_k) + f'(x_k)(x_{k+1} - x_k),$$

and as soon as $f'(x_k)$ is non zero, this gives

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)},$$

which is the definition of Newton's method.

Algorithm.

Newton-Raphson method. Tries to compute an approximate solution x to $f(x) = 0$.

$INPUT$: $f, x0$

DO : $x = x0$

While stopping criterion is not met do

$$x = x - \frac{f(x)}{f'(x)}$$

end while

$RETURN$: x

Remark.

If we introduce the function

$$g(x) = x - \frac{f(x)}{f'(x)},$$

Newton's method is nothing but the fixed point iteration method for g, i.e. $x_{k+1} = g(x_k)$. Therefore, we can use the theorems we obtained on fixed point problems to study the convergence of Newton's method.

Do it yourself.

Can you make a link between Newton's algorithm and the function g_5 studied in the previous section?

Answer.

Theorem.
Local convergence of Newton's method. Let $f : (a, b) \to \mathbb{R}$ be a C^2 function having a zero x^*. Consider the sequence $(x_k)_k$ generated by Newton's method for $k \geq 0$, x_0 being given. Assume

- $f'(x^*) \neq 0$ (x^* is a simple root of f).

Then, there exists a neighborhood I of x^* such that, for any $x_0 \in I$, Newton's iterations converge to x^* and the convergence is at least of order 2.

Proof. We consider the function

$$g(x) = x - \frac{f(x)}{f'(x)}.$$

By assumption, f' is continuous in a neighborhood of x^*. Since $f'(x^*) \neq 0$, $f'(x)$ does not vanish in a neighborhood of x^*, therefore g is well defined at least in a neighborhood of x^*. Assuming for simplicity that f is thrice differentiable (see the second remark after the proof otherwise) g becomes twice differentiable (since f is thrice differentiable). Furthermore,

$$g'(x) = 1 - \frac{(f'(x))^2 - f(x)f''(x)}{(f'(x))^2} = \frac{f(x)f''(x)}{(f'(x))^2},$$

and since x^* is a zero of f, we have $g'(x^*) = 0$. We can therefore apply the theorem on "Better than linear" speed of convergence of fixed point iterations, with $p = 1$, which yields that, if x_0 is close enough to x^*, $(x_k)_k$ converges at least quadratically to x^*.

Remark.
Advantages and drawbacks of Newton's methods

Newton's method had two great advantages, which explain why it is so often used in practice:

- The order of convergence is **quadratic**.
- It is **straightforward to generalized in higher dimension** (looking for zeros of a function $f : \mathbb{R}^d \to \mathbb{R}^d$) which is not the case of the bisection method for instance.

However, it also suffers from several drawbacks:

- The convergence is only **local**, i.e. if x_0 is close enough to x^*, which means that we first need to have a rough guess of where the zero is.
- Dealing with the derivative can be challenging and/or **expensive**, especially in higher dimensions. If that is the case, one can use approximations of the derivative instead. This leads to the secant method in dimension one, where $f'(x_k)$ is replaced by
$$\frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}},$$
or more generally to so-called *Quasi-Newton methods* in higher dimensions.
- The quadratic convergence only holds if $f'(x^*) \neq 0$, which in particular implies that the zero x^* must be locally unique. If that condition is not satisfied, the algorithm might still converge, but the convergence will then be at most linear.

Remark (to go further).
We did not prove the theorem about the convergence of Newton's method with minimal smoothness assumptions on f. Indeed, it is enough to assume that f is C^2, but in that case one cannot simply apply the theorem on "Better than linear" speed of convergence of fixed point iterations to get the proof, because g is not necessarilly smooth enough. Instead, one must directly use some Taylor expansions for f.

Regarding the stopping criterion, notice that two of the usual candidates for a zero-finding problem, namely

$$|x_{k+1} - x_k| \leq \varepsilon \quad \text{and} \quad |f(x_k)| \leq \varepsilon,$$

are closely related in the case of Newton's method, at least as long as $|f'(x_k)|$ stays reasonably far away from 0 and $+\infty$, since we have

$$x_{k+1} - x_k = -\frac{f(x_k)}{f'(x_k)}.$$

Examples

We are now going to use Newton's method on our easy test problem for this lecture, namely to find a zero of $f(x) = x^3 - 2$. Slightly more sophisticated examples will be treated later and in the case studies.

> Do it yourself. Implement Newton's method and test it to approximate $x^* = 2^{1/3}$, the unique solution in $\mathbb{R}$ to $f(x) = x^3 - 2 = 0$. Check that you indeed have quadratic convergence by first printing the error at each step, and then by using an appropriate plot.

```
1  def Newton(f, df, x0, k_max, eps):
2      """
3      Newton's algorithm to find a zero of a scalar function f, x_{k+
4      -----------------
5      Inputs:
6      f: the function
7      df: the function's derivative
8      x0 : initial point
9      k_max : maximal number of iterations
10     eps : tolerance for the stopping criterion
11
12     Outputs:
13     x = the sequence x_k
14     """
15     x = ...    # create aa vector x of zeros with size k_max+1
16     x[0] = x0
17     k = 0
18     while  ...  and  ... : # stopping criterion and "safety conditi
19         ...
20         ...
```

```python
def ftest(x):
    return x**3 - 2

def dftest(x):
    return 3*x**2

k_max = ...
eps = ...
x0 = 1
xstar = 2**(1/3)

x = ...  # Iterates computes using Newton's method
print('xstar =', xstar)
print('x =', x)

err = ... # Error for each iterate
```

```python
# log-log plot of the error
...
```

Do it yourself. Consider now the function $F(x) = x^5 - x + 1$. First plot this function on $[-1.5, 1.5]$, and then try to use Newton's method to approximate it's real zero. Try several initial conditions , like $x_0 = -1.5$ and $x_0 = -0.5$, and comment on the results.

```python
def F(x):
    return ...

def dF(x):
    return ...

pts = np.linspace(-1.5, 1.5, 500)
fig = plt.figure(figsize=(12, 8))
plt.plot(pts, F(pts))
plt.plot(pts, 0*pts, '--k')
plt.xlabel('$x$', fontsize=18)
plt.title('$F(x)=x^5-x+1$', fontsize=18)

```

```python
k_max = ...
eps = ...
x0 = ...

x = Newton(F, dF, x0, k_max, eps)
print('x =', x)
```

Answer.

Do it yourself. Finally, consider the function $F_2(x) = x^2(x^2 + 2)$, and try to use Newton's method to approximate it's real zero. Study the convergence rate graphically, and comment on the results.

Answer.

Back to the case studies [★]

We come back here to the case studies described in the introduction and try to solve them using the methods presented above.

Case study 1: State equation of a gas, a solution using bisection

We use the bisection method to solve case study 1 and compute the volume of 1000 molecules of CO_2 at temperature $T = 300\,K$ and pressure $p = 3.5 \cdot 10^7\ Pa$.

To do so, we have to solve the following equation for V:

$$f(V) = \left[p + a\left(\frac{N}{V}\right)^2\right](V - Nb) - kNT = 0$$

with $N = 1000$, $k = 1.3806503 \cdot 10^{-23}\ J\,K^{-1}$, $a = 0.401\ Pa\,m^6$ and $b = 42.7 \cdot 10^{-6}\ m^3$.

Do it yourself. Solve the problem using the bisection method.

Case study 2: Investment fund, solutions using bisection or Newton's method

We recall that we have to find i solution to

$$f(i) = d\frac{(1+i)^{n_{end}} - 1}{i} - S = 0 \quad \text{where} \quad S = 30\,000, \quad d = 30, \quad \text{a}$$

> Do it yourself. Solve the problem using the bisection method and/or Newton's method.

Case study 3: A first population model, a solution using Newton's method

We want to find an approximation for the natural growth rate λ in France. To do so, we have to solve the following non-linear equation for λ (we know that $\lambda \neq 0$ since the population increases more than the migratory balance):

$$f(\lambda) = N(2017) - N(2016)\exp(\lambda) - \frac{r}{\lambda}(\exp(\lambda) - 1),$$

where $N(2016) = 66\,695\,000$, $N(2017) = 66\,954\,000$ and $r = 67\,000$.

> Do it yourself. Solve the problem using Newton's method.

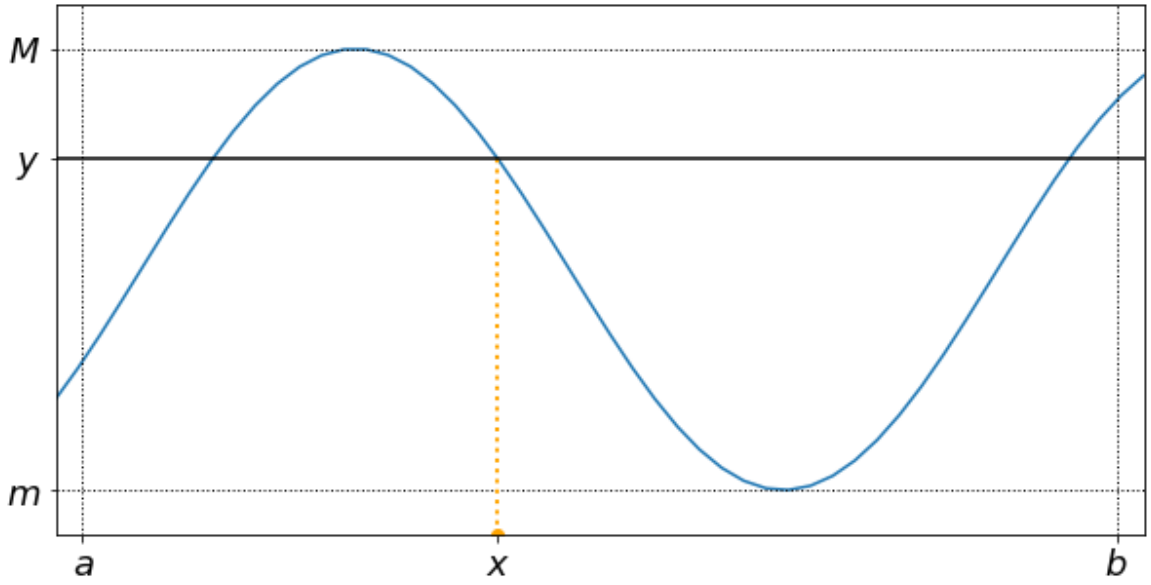
Appendix

Intermediate value theorem

> Theorem.
> Intermediate value Theorem
>
> Suppose $f : [a, b] \mapsto \mathbb{R}$ is continuous on $[a, b]$. Define $m = \min\{f(a), f(b)\}$ and $M = \max\{f(a), f(b)\}$. Then,
>
> $$\forall y \in]m, M[, \quad \exists x \in]a, b[\quad \text{such that} \quad f(x) = y.$$

> As a consequence, if a continuous function takes values of opposite signs in an interval, it has a root in this interval.

The following figure provides an example of x guaranteed by this theorem. In this case, the zero is not unique.




```
# execute this part to modify the css style
from IPython.core.display import HTML
def css_styling():
    styles = open("./style/custom3.css").read()
    return HTML(styles)
```