Recherche des zéros

Dans ce chapitre

Algorithmes

- Méthode de bisection
- Méthode de relaxation
- Méthode de Newton
- Généralisations de la méthode de Newton

Zéros des fonctions

Problème : Etant donné une fonction réelle dont on sait qu'elle a un zéro sur l'intervalle $I = [a, b]$, on cherche une valeur approximative de ce zéro.

Plus formellement :

Soit $I = [a, b]$ un intervalle et $f : I \to \mathbb{R}$ continue (ou même dérivable si necessaire) sur I, avec $f(a)f(b) \leq 0$. Alors d'après le théorème des valeurs intermédiaires il existe $x_0 \in I$ tel que $f(x_0) = 0$. Si de plus f est monotone alors x_0 est unique.
Problème : déterminer (un des) x_0 numériquement avec une précision minimale donnée.

Théorème des valeurs intermédiares :
Soit $f : [a, b] \to \mathbb{R}$ continue et soit y compris entre $f(a)$ et $f(b)$. Alors il existe $x \in [a, b]$ tel que $f(x) = y$.

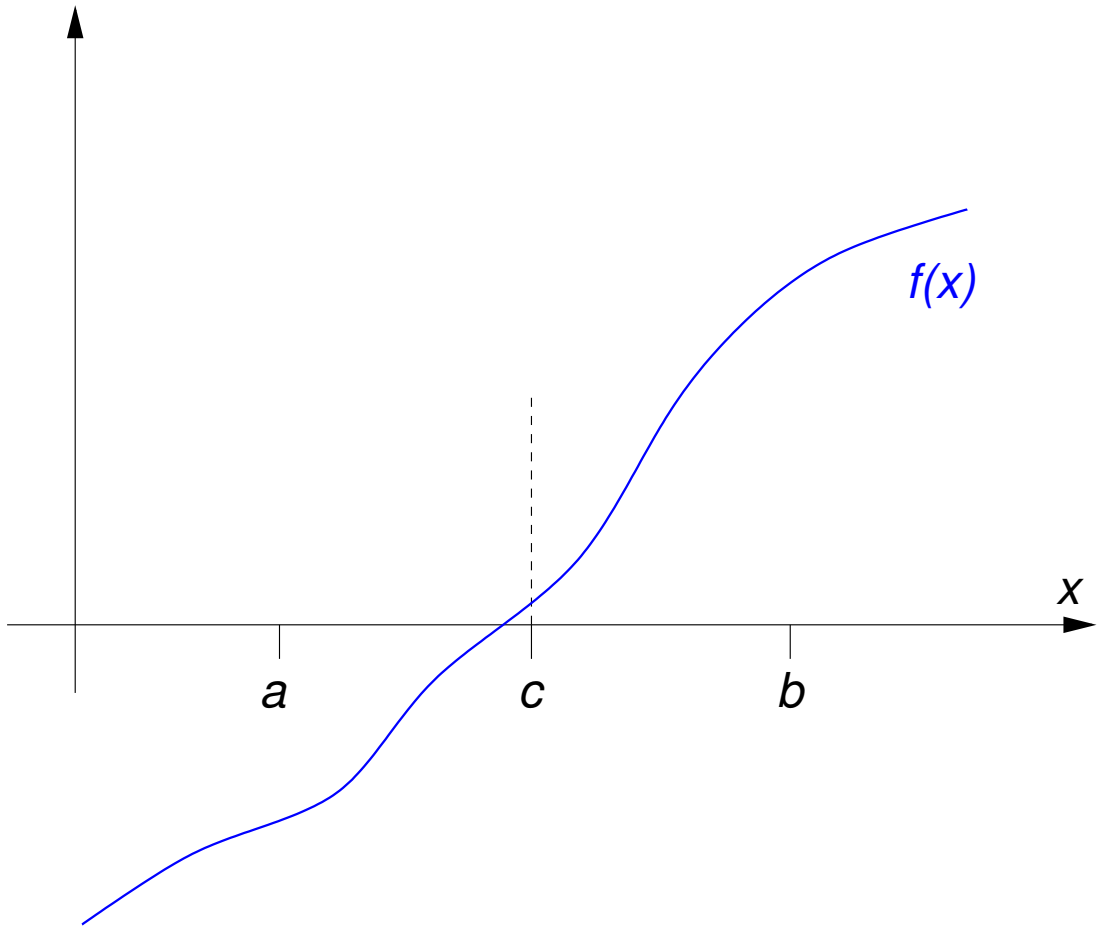
Méthode de bisection

Soit $\epsilon > 0$ la précision souhaitée.

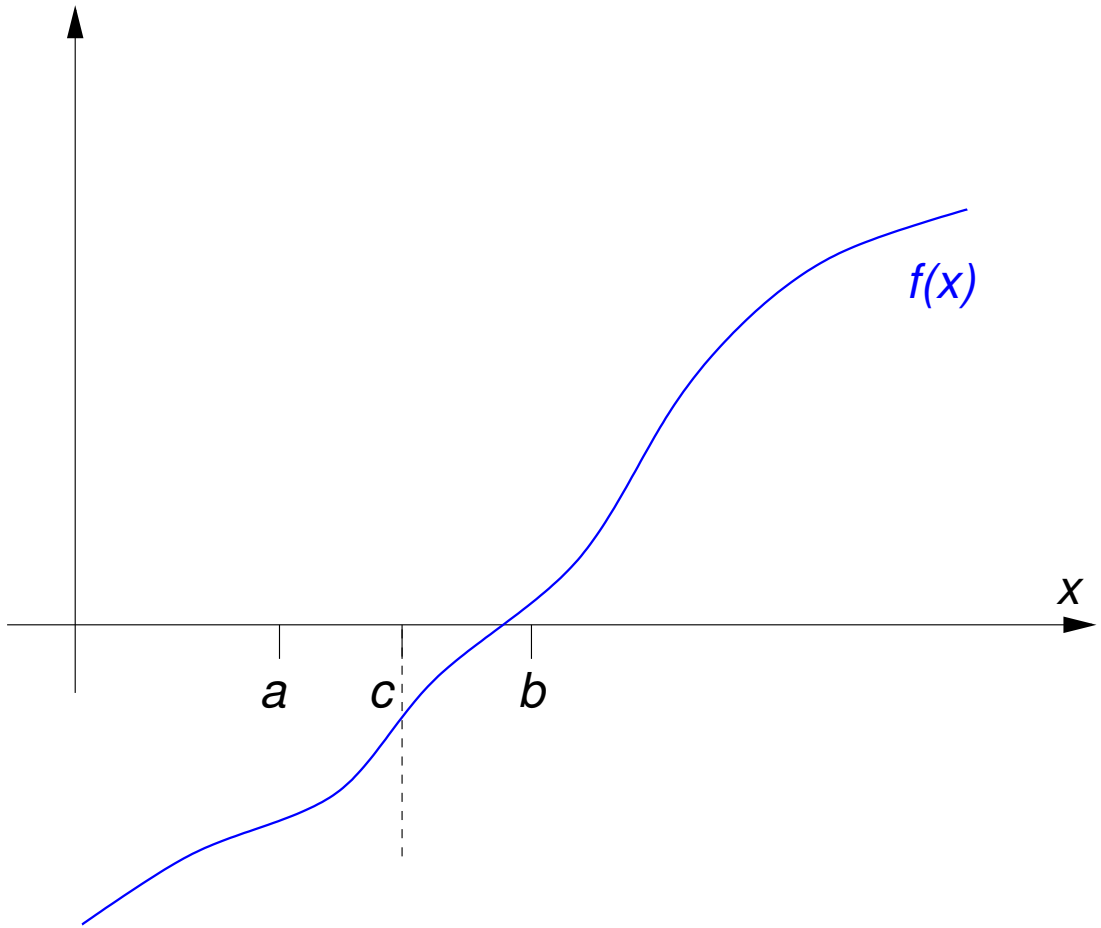
Algorithme :

1. Poser $c = (a + b)/2$, le milieu de l'intervalle I.
2. Si $b - a < 2\,\epsilon$: terminer et retourner $x_0 = c$.
3. Partager I en deux : $I_1 = [a,\, c]$ et $I_2 = [c,\, b]$.
4. Si $f(a)f(c) \leq 0$: il y a un zéro dans I_1, alors répéter avec $I = I_1$.
5. Sinon, répéter avec $I = I_2$.

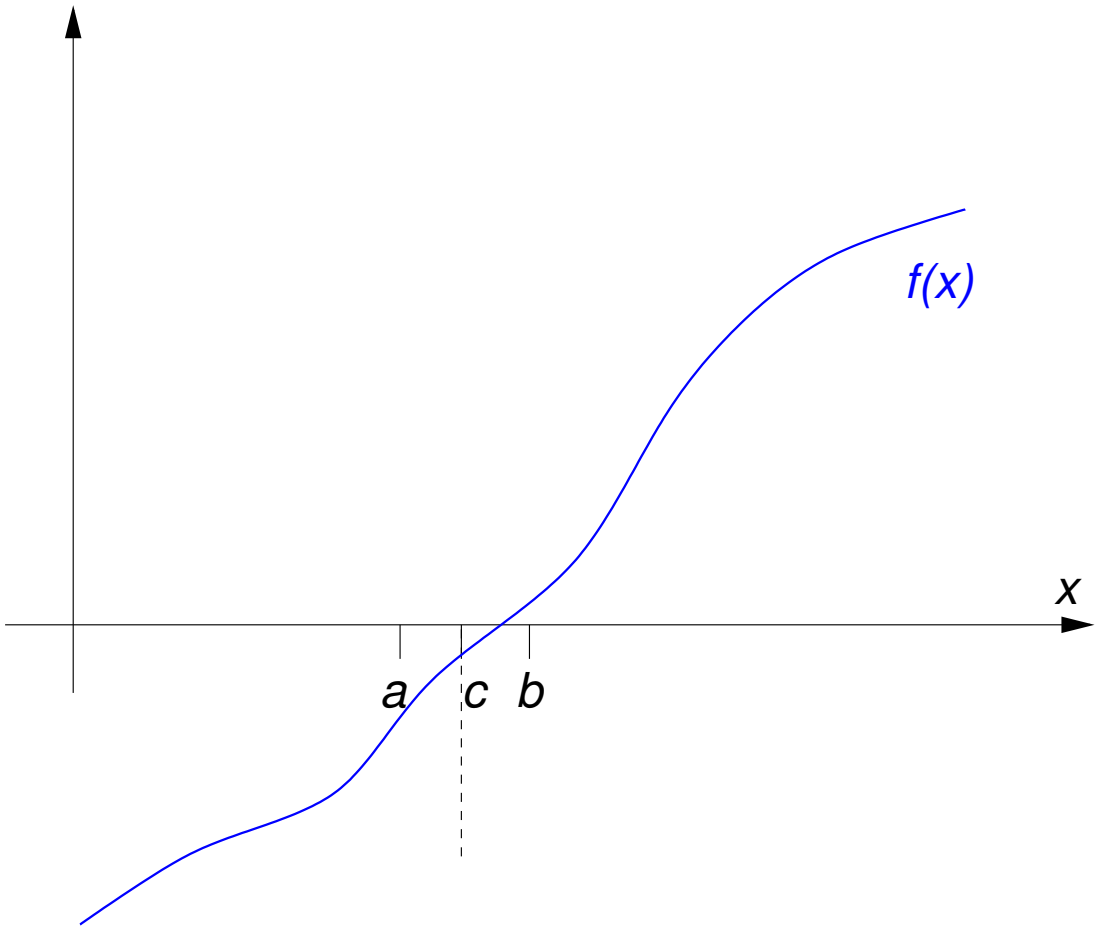
Méthode de bisection



Méthode de bisection



Méthode de bisection



Implémentation en Python

Voici le code Python pour $f(x) = x^5 - x - 1$, $I = [1, 2]$, $\epsilon = 10^{-5}$:

```
def f(x):          # la fonction dont on cherche un zéro
    return x**5 - x - 1

a, b = 1.0, 2.0     # f(a) = -1 et f(b) = 29
                    #  => il y a un zéro dans [a, b]
c = (a + b) / 2     # point du milieu
epsilon = 1.0E-5    # tolérance

while b - a >= 2 * epsilon: # on est assez proche? sinon:
    if f(a) * f(c) <= 0: # si zéro dans la moitié gauche:
        b = c               #  pt de droite <- pt du milieu
    else:                   # sinon:
        a = c               #  pt de gauche <- pt du milieu
    c = (a + b) / 2         # récalculer point du milieu

print("Le zéro est à x =", c)
```

Méthode de relaxation

Idée :

- Pour résoudre l'équation $f(x) = 0$, trouver une fonction $\phi(x)$ appropriée telle que

$$f(x) = 0 \;\Leftrightarrow\; \phi(x) = x$$

(il y a une infinité de choix pour ϕ — le succès de la méthode dépendra du choix).

- On cherche alors un point fixe x^* de la fonction ϕ.
- Essayer de trouver un point fixe par l'application repétée de la fonction ϕ sur un point de départ x_1 (qui est aussi au choix et doît être bien choisi pour que la méthode fonctionne)

$$\begin{aligned} x_2 &= \phi(x_1) \\ x_3 &= \phi(x_2) = \phi(\phi(x_1)) \\ x_4 &= \phi(x_3) = \phi(\phi(\phi(x_1))) \\ &\dots \\ \lim_{n \to \infty} x_n &\stackrel{?}{=} x^* \end{aligned}$$

Méthode de relaxation

Exemple :

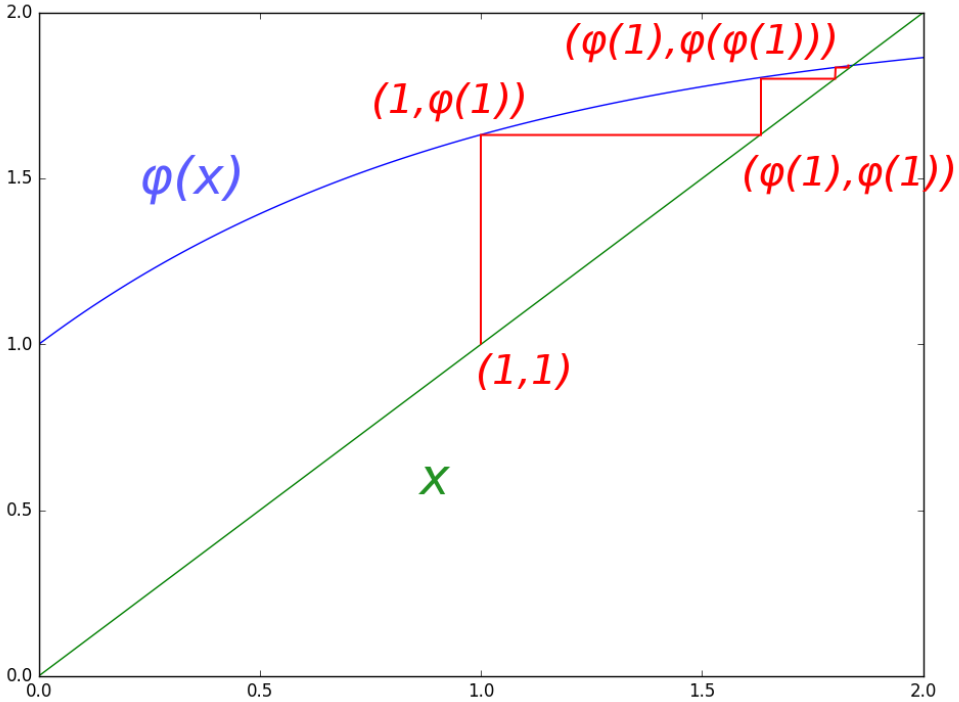
- On cherche un zéro x^* de la fonction $f(x) = 2e^x - xe^x - 1$.
- Équivalent : on cherche un point fixe x^* de la fonction $\phi(x) = 2 - e^{-x}$.
- Avec $x_1 = 1$ on trouve

$$
\begin{aligned}
x_2 &= \phi(x_1) = 1.63212 \\
x_3 &= \phi(x_2) = 1.80448 \\
&\dots \\
x_9 &= \phi(x_8) = 1.84141 \\
x_{10} &= \phi(x_9) = 1.84141 \\
x_{11} &= \phi(x_{10}) = 1.84141 \\
&\dots
\end{aligned}
$$

Conclusion : Pour $x^* \approx 1.84141$ on a

$$
\begin{aligned}
\phi(x^*) &= x^* \\
\Leftrightarrow \quad 2 - e^{-x^*} &= x^* \\
\Leftrightarrow \quad 2e^{x^*} - x^* e^{x^*} - 1 &= 0 \\
\Leftrightarrow \quad f(x^*) &= 0\,.
\end{aligned}
$$

Méthode de relaxation : Illustration



Condition suffisante pour la convergence : $|\phi'(x)| < 1$ partout
ou bien : ϕ est une contraction.

Détails mathématiques

Théorème du point fixe :

Soit $I = [a, b]$ un intervalle, $0 \leq k < 1$, et $\phi : I \to I$ continue tel que $|\phi(x) - \phi(y)| \leq k|x - y| \quad \forall\, x, y \in I$ (on dit que ϕ est une contraction).

Alors il existe un point fixe unique $x^* \in I$.

De plus, une suite (x_n) dans I vérifiant $x_{n+1} = \phi(x_n)$ convergera vers x^*.

Démonstration :

Soit $x_1 \in I$ quelconque et $x_{n>1}$ défini par récurrence : $x_{n+1} = \phi(x_n)$. On a $|x_{n+1} - x_n| \leq k^{n-1}|x_2 - x_1|$, donc

$$\begin{aligned} |x_{n+m} - x_n| \leq & |x_{n+1} - x_n| + |x_{n+2} - x_{n+1}| + \ldots + |x_{n+m} - x_{n+m-1}| \\ & \leq \left(k^{n-1} + k^n + \ldots + k^{n+m-1}\right) |x_2 - x_1| \\ & = k^{n-1} \frac{1 - k^m}{1 - k} |x_2 - x_1| \end{aligned}$$

Comme $k < 1$, l'expression dans la dernière ligne tend vers zero quand $n \to \infty$, alors les (x_n) forment une suite de Cauchy qui converge vers un x^*. On a $\phi(x^*) = x^*$ grace à la continuité de la fonction ϕ.

Méthode de Newton

Cas spécial : méthode de Newton

L'idée de la méthode de Newton est de linéariser f autour d'un point x_1, de trouver le zéro de la fonction tangente t_1 ainsi définie, et d'itérer :

- Poser
$$t_1(x) = f(x_1) + f'(x_1)(x - x_1)$$
(= développement limité de f en x_1 à l'ordre 2, alors $t_1(x) = f(x) + \mathcal{O}(|x - x_1|^2)$)
- Le zéro de $t_1(x)$ est à $x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$.
- Itérer cette procédure tant que la distance entre les deux valeurs consécutives x_n et x_{n+1} est $> \epsilon$. Quand $|x_{n+1} - x_n| < \epsilon$, terminer et renvoyer x_{n+1}.

D'après le théorème du point fixe, la suite des x_n convergera vers un point fixe x^* de la fonction auxiliaire $\phi : x \mapsto x - \frac{f(x)}{f'(x)}$ (si cette dernière est une contraction).

Or $\phi(x^*) = x^*$, alors $\frac{f(x^*)}{f'(x^*)} = 0$, alors $f(x^*) = 0$.

Méthode de Newton

Zéro de la fonction f = point fixe de la fonction auxiliaire

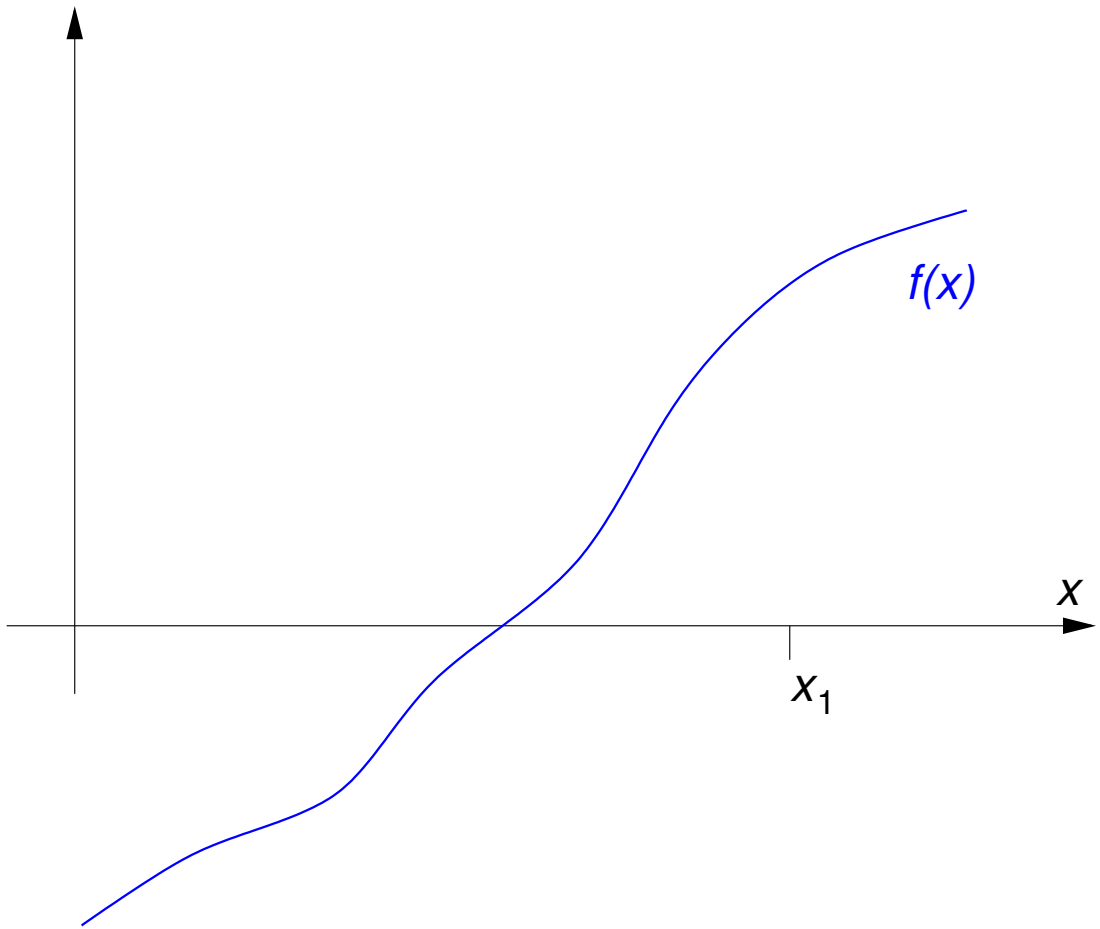
$$\phi : x \mapsto x - \frac{f(x)}{f'(x)}$$

Remarques :

- Si f est deux fois dérivable et $f'(x) \neq 0 \; \forall \, x \in I$, alors $\phi'(x) = \frac{f(x)f''(x)}{f'(x)^2}$.
- Dans ce cas : ϕ est une contraction $\Leftrightarrow \exists \, k < 1$ avec $|\phi'(x)| \leq k$, soit $\left| \frac{f(x)f''(x)}{f'(x)^2} \right| \leq k$
  ("$\Leftarrow$" vient du théorème des accroissements finis, "$\Rightarrow$" de la définition de la dérivée)
- Pour que l'algorithme converge : Il est suffisant mais pas nécessaire que ϕ soit une contraction.

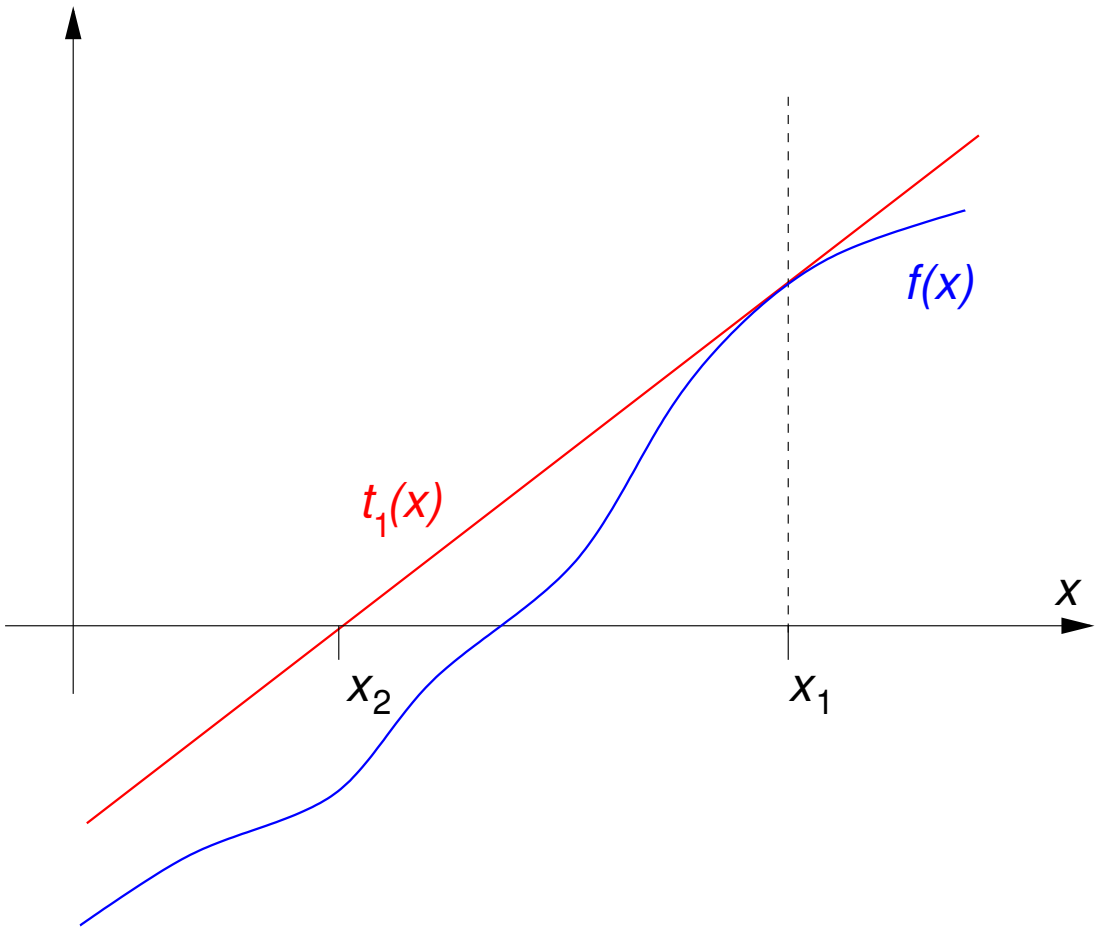
  Etablir un critère suffisant et nécessaire peut être très difficile voire impossible, voir exercices sur le cas complexe.
- Evidemment il faut bien choisir le point de départ (un extremum de f, par exemple, serait un mauvais choix — pourquoi ?)

Méthode de Newton



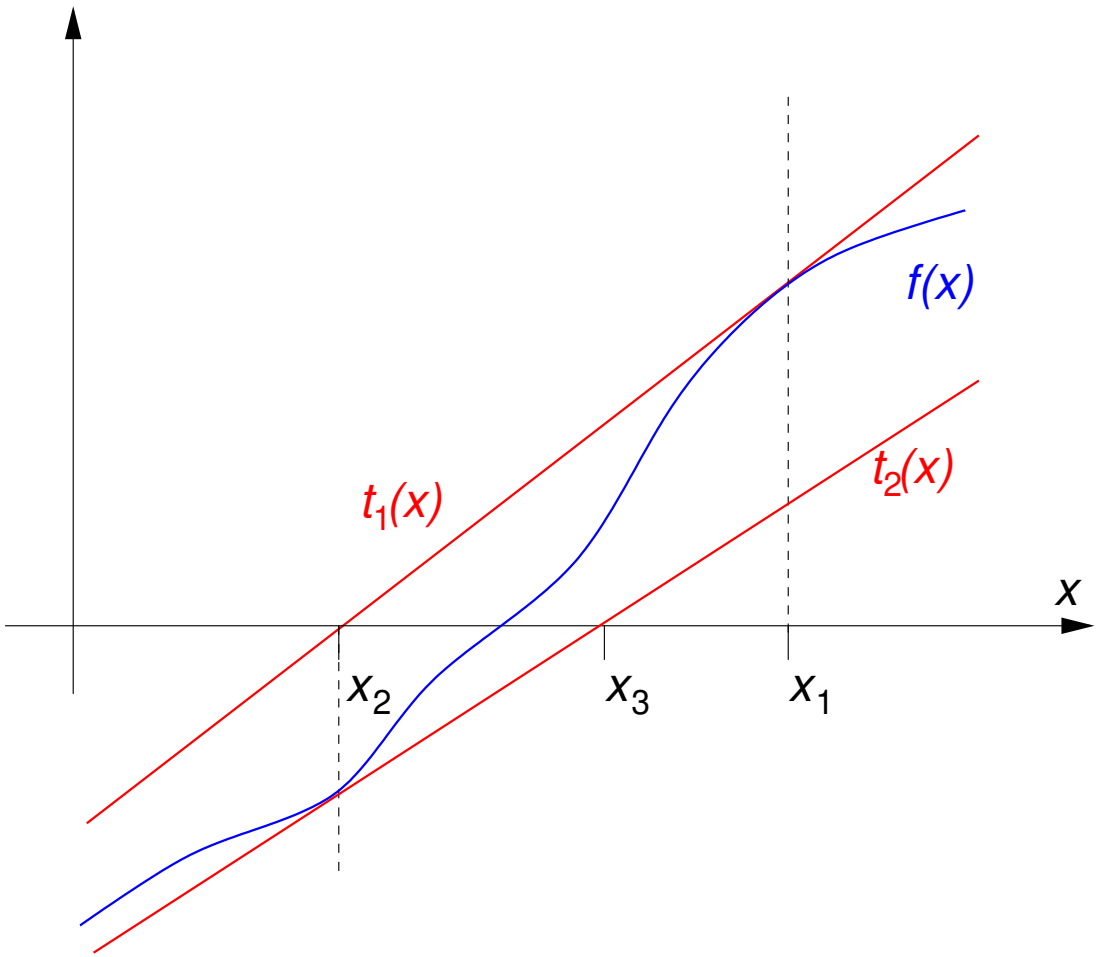
$$t_n(x) = f(x_n) + f'(x_n)(x - x_n)$$

Méthode de Newton



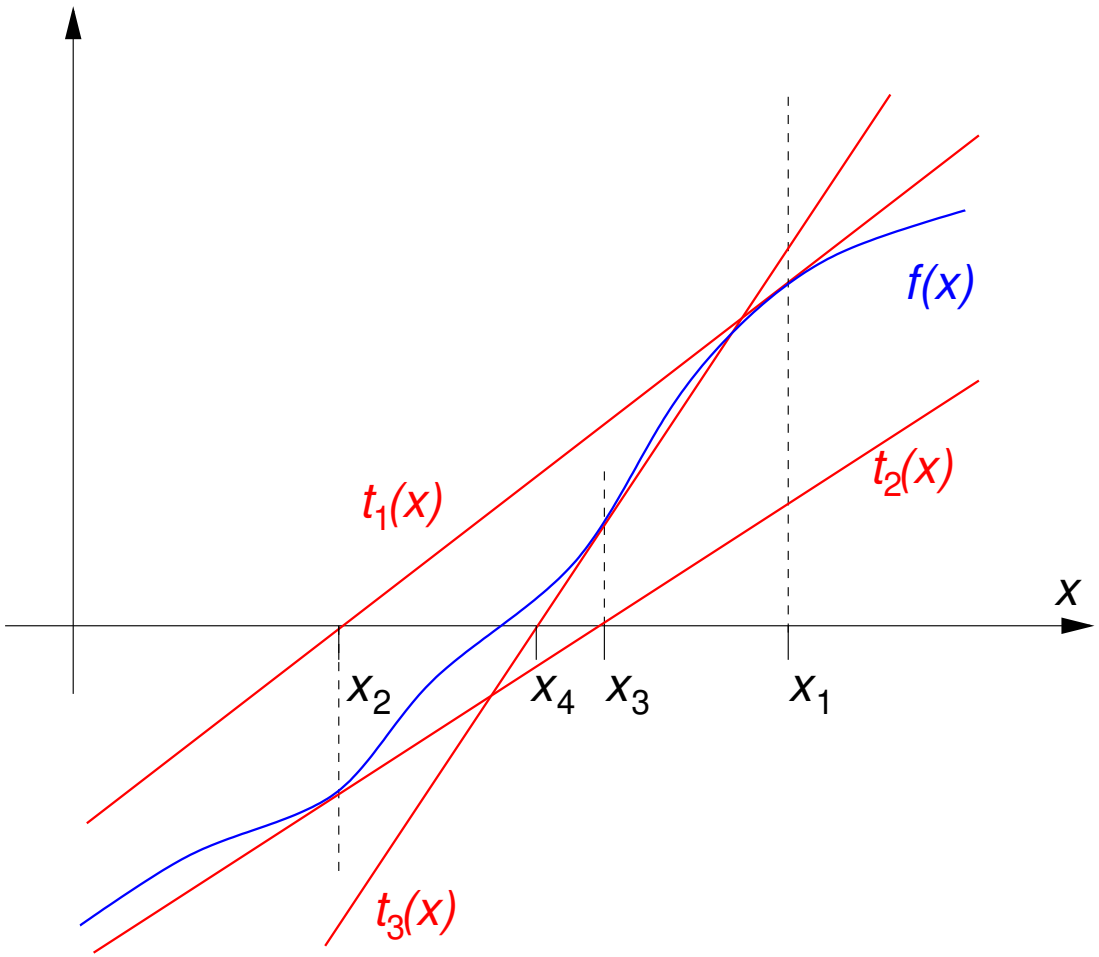
$$t_n(x) = f(x_n) + f'(x_n)(x - x_n)$$

Méthode de Newton



$$t_n(x) = f(x_n) + f'(x_n)(x - x_n)$$

Méthode de Newton



$$t_n(x) = f(x_n) + f'(x_n)(x - x_n)$$

Méthode de Newton

Algorithme :

1. Partir avec x au choix (mais intelligemment choisi : proche du zéro, pas un point critique...)
2. Remplacer $x_{\text{ancien}} \leftarrow x, \qquad x \leftarrow x - \frac{f(x)}{f'(x)}$.
3. Si $|x - x_{\text{ancien}}| < \epsilon$, on est suffisamment proche du zéro : terminer et renvoyer x. Sinon, répéter.

Pour tester la convergence, d'autres critères sont envisageables (par exemple, est-ce que $|f(x)| < \epsilon$?) en fonction du problème sous étude.

Méthode de Newton

Exemple :

On cherche un zéro de la fonction $f : x \mapsto x^5 - x - 1$ dont la dérivée est $f' : x \mapsto 5\,x^4 - 1$. La précision souhaitée est $\epsilon = 10^{-5}$ et le point de départ sera $x_0 = 1$.

```
def f(x):
    return x**5 - x - 1

def df(x):
    return 5 * x**4 - 1

epsilon = 1.E-5
x = 1.
x_ancien = x + 2 * epsilon  # valeur initiale pas importante

while abs(x - x_ancien) > epsilon:
    x_ancien = x
    x = x - f(x) / df(x)

print("Le zéro est à x =", x)
```

C'est souvent une bonne idée de limiter le nombre d'itérations pour éviter des boucles infinies (au cas où la méthode ne converge pas avec le point de départ choisi).

Méthode de la sécante

Pour appliquer la méthode de Newton, il faut connaître la dérivée de la fonction f, de préférence analytiquement.

Sinon : calculer la dérivée approximative numériquement avec le taux d'accroissement

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Par exemple, au point $x = x_n$ avec $h = x_n - x_{n-1}$:

$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$$

Insérer dans la formule de récurrence de la méthode de Newton, $x_{n+1} = x_n - f(x_n)/f'(x_n)$:

$$x_{n+1} = x_n - f(x_n) \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$$

Méthode de la sécante.

Algorithme :

- Commencer par deux points x_0 et x_1 ; poser $n = 1$.
- Calculer x_{n+1} avec la formule de récurrence ci-dessus.
- Itérer avec $n \leftarrow n + 1$; quand $|x_{n+1} - x_n|$ est suffisamment petit, terminer et renvoyer x_{n+1}.

Comparaison des méthodes

La table suivante montre l'erreur absolu après n itérations pour le zéro de $x^5 - x - 1$ (avec le choix $\phi(x) = (x+1)^{1/5}$ pour la méthode de relaxation) :

Itérations	Bisection	Relaxation	Newton
1	-8.27e-02	-7.84e-02	-4.66e-01
2	4.23e-02	-8.33e-03	-2.06e-01
3	-2.02e-02	-8.96e-04	-5.63e-02
4	1.11e-02	-9.65e-05	-5.43e-03
5	-4.57e-03	-1.04e-05	-5.59e-05
6	3.24e-03	-1.12e-06	-6.01e-09

On voit que la méthode de Newton a besoin de beaucoup moins d'itérations que les deux autres. Le gain du temps n'est pas signifiant pour cet exemple (implémentation pas optimisée, fonction f pas très compliquée). Voici le temps en secondes requis sur mon ordinateur pour atteindre une précision fixe :

Precision	Bisection	Relaxation	Newton
1.00e+00	5.51e-07	3.17e-07	4.48e-07
1.00e-02	2.47e-06	5.24e-07	1.40e-06
1.00e-04	4.71e-06	7.45e-07	1.64e-06
1.00e-06	6.57e-06	9.51e-07	1.87e-06

Généralisations de la méthode de Newton

- La méthode peut être appliquée sans modifications pour des fonctions complexes. Les domaines de convergence vers les zéros forment des structures gémoétriques très intéressantes : les "fractales de Newton" $\rightarrow$ exercices.
- Méthode de Halley : basée sur l'itération

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

  pour des fonctions au moins deux fois dérivables. Converge plus rapidement que la méthode de Newton, mais nécéssite l'évaluation de la deuxième dérivée.
- Méthodes de Householder : Soit f k-fois dérivable avec des dérivées continues. On itère

$$x_{n+1} = x_n + k\, \left. \frac{\frac{d^{k-1}}{dx^{k-1}} \left(\frac{1}{f(x)} \right)}{\frac{d^k}{dx^k} \left(\frac{1}{f(x)} \right)} \right|_{x=x_n}$$

  Pour $k = 1$ on retrouve la méthode de Newton, pour $k = 2$ celle de Halley.

La méthode de Newton n-dimensionnelle

Étant donnée une fonction $\vec{f}: \mathbb{R}^n \to \mathbb{R}^n$, un zéro de $\vec{f}$ peut être trouvé par la généralisation de la méthode de Newton : on itére

$$\vec{x}_{n+1} = \vec{x}_n - J^{-1}(\vec{x}_n)\, \vec{f}(\vec{x}_n)$$

où J est la matrice jacobienne des dérivées de $\vec{f}$,

$$J = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{pmatrix}$$

et J^{-1} est son inverse. Voir le chapître sur l'optimisation plus tard pour quelques applications.

Les bibliothèques NumPy et matplotlib

Dans ce chapitre

Python

- La bibliothèque `numpy`
- Manipulation des tableaux
- Calcul matriciel
- Importer et exporter des données
- Graphisme avec `matplotlib`

Les tableaux

On rappelle les caractéristiques d'une liste Python :

- contient plusieurs éléments qui ne sont pas forcement du même type
- taille variable, peut changer (p.ex. avec l'opérateur +=)
- 1-dimensionnel = 1 seul indice (sauf si les éléments sont eux-mêmes des listes)

La bibliothèque NumPy se base sur un objet similaire, le tableau (anglais : "array")

Caractéristiques d'un tableau NumPy :

- contient plusieurs éléments qui sont forcement du même type, toujours un type numérique (`int`, `float`, `complex` ...)
- taille fixe
- n-dimensionnel : vecteurs, matrices, tenseurs...
- optimisé pour le calcul numérique : plus rapide que les listes, beaucoup de fonctionnalité pour la manipulation efficace.

Créer un tableau

Exemples :

```
import numpy as np  # pour importer toute la bibliothèque

sigma3 = np.array([[1, 0], [0, -1]])  # matrice [[ 1  0]
                                      #          [ 0 -1]]

s = np.array([1, 0], dtype=complex) # vect. [ 1.+0.j 0.+0.j]

nul2x3 = np.zeros((2, 3))  # [[ 0. 0. 0.]
                           #  [ 0. 0. 0.]]

id3x3 = np.identity(3, dtype=int) # [[ 1 0 0]
                                  #  [ 0 1 0]
                                  #  [ 0 0 1]]

rng = np.arange(0.8, 2, 0.4)      # [ 0.8 1.2 1.6]
```

Créer un tableau

Un tableau peut se créer

- en spécifiant les éléments dans une liste (ou liste de listes...) avec la fonction `numpy.array()`
- en spécifiant les dimensions par un tuple `(x, y, z...)` des `int`
  - `numpy.zeros()` crée un tableau de zéros
  - `numpy.ones()` crée un tableau de uns
  - `numpy.empty()` crée un tableau sans initialiser les éléments
- cas spécial : la matrice d'identité $n \times n$, `numpy.identity(n)`
- tableaux 1-dimensionnels de nombres uniformement espacés :
  - `numpy.arange(debut, fin, pas)` : comme `range` mais avec des `float`
  - `numpy.linspace(debut, fin, N)` : `N` nombres entre `debut` et `fin` (inclus)

Si besoin, spécifier le type de données des éléments avec l'argument `dtype` lors de la construction

Opérations arithmétiques sur les tableaux

Les opérations arithmétiques `+ = * / // %` entre les tableaux `numpy` sont définies par élément :

```
import numpy as np
sigma3 = np.array([[1, 0], [0, -1]], dtype=float)
print(sigma3 * np.array([[2., 3.] [4., 5.]])) # [[ 2.  0.]
                                               #  [ 0. -5.]]
```

Si les dimensions ne s'accordent pas, une opération arithmétique impliquant deux tableaux produira une erreur.

En revanche, il est toujours possible de ajouter/soustraire/multiplier/diviser par un scalaire :

```
print(sigma3 - 1)   # [[  0. -1.]
                    #  [ -1. -2.]]
```

Indicer et couper un tableau

On peut indicer un tableau avec plusieurs indices selon ses dimensions :

```
sigma3 = np.array([[1, 0], [0, -1]], dtype=float)
print(sigma3[1, 1])   # "-1.0"
```

(avec la généralisation évidente pour des tableaux d-dimensionnels).

On peut aussi le couper comme une liste Python :

```
# tous les éléments de la deuxième colonne:
print(sigma3[:, 1])   # "[ 0. -1.]"
# tous les éléments de la première ligne
print(sigma3[0, :])   # "[ 1. 0.]"
```

Couper des tableaux (array slicing), méthodes avancées

Créer le vecteur $(0, 1, 2, \ldots, 11)$ et le réarranger dans une matrice 3×4 :

```
a = np.reshape(np.array(range(12)), (3, 4))
print(a)    # [[ 0  1  2  3]
            #  [ 4  5  6  7]
            #  [ 8  9 10 11]]
```

Avec l'opérateur `[i:j:k]` on accède aux éléments

- à partir de l'indice `i` (par défaut : début)
- jusqu'à l'indice `j` exclu (par défaut : fin)
- en sélectionnant un élément sur `k` (par défaut : 1)

Exemples :

```
print(a[1, ::2])   # [4 6] (2ème ligne, colonnes paires)
print(a[1, 1::2])  # [5 7] (2ème ligne, colonnes impaires)
print(a[0, 1:3])   # [1 2] (1ère ligne, colonnes 1 et 2)
print(a[1:,:2])    # [[4 5]  (derniers 2 éléments
                   #  [8 9]]  des premières 2 colonnes)
```

Multiplication matricielle avec les tableaux

Un tableau avec deux indices peut représenter une matrice. Un tableau avec un seul indice est un vecteur.
Les produits entre les matrices et vecteurs (produit scalaire entre deux vecteurs, action d'une matrice sur un vecteur, produit matriciel entre deux matrices) se calculent avec l'opérateur `@` (et non pas avec `*` qui est la multiplication élément par élément !)

Exemples :

```
M = np.array([[1., 2., 4.], [2., -1., 0.], [5., -2., 1.]])
v = np.array([0., 1., 2.])
w = np.array([1., -1., 1.])

print(v @ w) # produit scalaire v . w, résultat: 1.0

print(M @ v) # matrice agît sur vecteur, M . v
             #  résultat: [ 10. -1. 0.]

print(M @ M) # produit matriciel M . M
             #  résultat: [[ 25.  -8.   8.]
             #             [  0.   5.   8.]
             #             [  6.  10.  21.]]
```

Multiplication matricielle avec les tableaux

Exemple : Calcul des moyennes quantiques $\langle\psi|\sigma^{1,2,3}|\psi\rangle$ pour un système à deux niveaux

```
import numpy as np

sigma = np.array([[[ 0, 1], [1, 0]], # les matrices de Pauli
                  [[ 0, -1j], [1j, 0]],
                  [[ 1, 0], [0, -1]]], dtype = complex)

def norme(psi):  # la norme d'un vecteur complexe
    psic = np.conjugate(psi)
    return np.sqrt(psic @ psi)

psi1 = complex(input("Entrer psi1: "))  # composantes de psi
psi2 = complex(input("Entrer psi2: "))
psi = np.array([psi1, psi2], dtype=complex)
psi /= norme(psi)           # normaliser le vecteur psi
psic = np.conjugate(psi)  # le vecteur conjugué complexe
vm = [psic @ sigma[i] @ psi for i in range(3)]

print("Valeurs moyennes:\n <sigma1> = ", vm[0].real,
                        "\n <sigma2> = ", vm[1].real,
                        "\n <sigma3> = ", vm[2].real)
```

Méthodes utiles pour le calcul matriciel

La classe `numpy.ndarray` contient quelques autres champs et méthodes utiles pour manipuler des vecteurs et matrices : si `A` est un tableau, alors

- `numpy.transpose(A)` représente la transposée de `A` (raccourci : `A.T`)
- `numpy.trace(A)` calcule la trace $\sum_i A_{ii}$
- `numpy.conjugate(A)` calcule le tableau conjuguée complexe
- `numpy.amax(A)` calcule l'élément maximal
- `numpy.sum(A)` calcule la somme des éléments
- ...

Voir `https://docs.scipy.org/doc/numpy/reference/routines.html` pour documentation complète.

Fonctions sur les tableaux

Les fonctions élémentaires `sin`, `cos`, `exp`, `log`, `sqrt` etc. des bibliothèques `math` et `cmath` existent aussi dans la bibliothèque `numpy`. Si on donne un tableau comme argument, la valeur de retour sera également un tableau avec les valeurs de fonction des éléments : "array broadcasting".

```
import numpy as np

x = np.array([-1, 0, 1])  # un tableau
print(np.arccos(x)) # "[ 3.14159265  1.57079633  0.]"
```

À préférer par rapport au code équivalent (mais moins vite et moins nette)

```
import math

x = [-1, 0, 1] # une liste
acosx = [math.acos(t) for t in x]  # lent sur des grandes
                                   #  listes!
print(acosx)
```

Fonctions sur les tableaux

Pour convertir une fonction ordinaire en fonction qui peut s'appliquer sur un tableau numpy, on utilise la fonction `numpy.vectorize`.

Exemple :

```
import numpy as np

# Une fonction ordinaire:
def f(x, y):  # retourne x si x>y et y-x sinon
    if x > y:
        return x
    return y - x

# La fonction vectorisée:
vf = np.vectorize(f)

x = np.array([1., 3., 7.])
vf(x, 4)  # array([ 3., 1., 7.])
```

Copier un tableau

Une copie par référence se fait avec l'opérateur d'affectation $=$, une copie "superficielle" avec `numpy.copy()` :

```
import numpy as np

a = sigma3          # permet d'accéder à sigma3
                    #  avec la novelle référence a
b = np.copy(a)      # crée un nouveau tableau
                    #  qui est une copie de sigma3
```

Importer et exporter des données

La fonction `numpy.loadtxt` permet d'importer des données d'un fichier.

```
# Fichier de données "donnees.dat"

3.14159 2.71828 0.57721   # commentaires seront ignorés
1.      -2.     3.e5
```

```
# Fichier du programme

import numpy as np

a = np.loadtxt("donnees.dat")
print(a) # [[ 3.1415900e+00  2.7182800e+00 5.7721000e-01]
         #  [ 1.0000000e+00 -2.0000000e+00 3.0000000e+05]]
```

Dans le fichier de données :

- éléments doivent être séparés par un ou plusieurs espaces blancs
- lignes blanches et commentaires # sont ignorés

Importer et exporter des données

La fonction `numpy.savetxt` permet d'enregistrer des données dans un fichier.

```
import numpy as np

a = np.arange(0.0, 4.0, 1.0) # le tableau [ 0., 1., 2., 3.]
np.savetxt("mydata.dat", a, header="Commentaire facultatif")
```

Fichier `mydata.dat` résultant :

```
# Commentaire facultatif
0.00000000000000000e+00
1.00000000000000000e+00
2.00000000000000000e+00
3.00000000000000000e+00
```

Visualisation avec matplotlib

Python dispose d'une bibliothèque très puissante pour créer des graphiques : la bibliothèque `matplotlib.pyplot`.

Usage typique pour tracer le graphe d'une fonction :

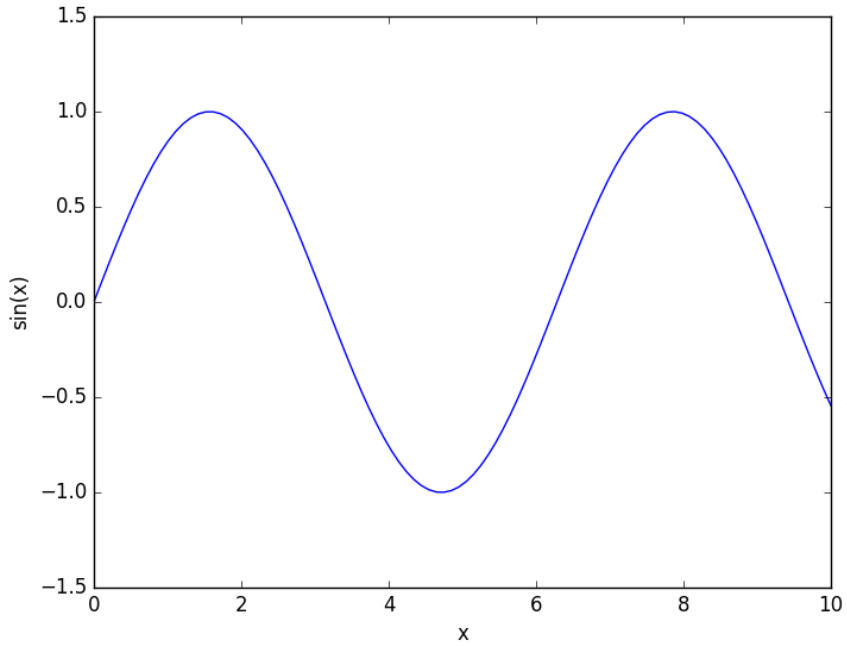
```
import numpy as np
import matplotlib.pyplot as plt

xpts = np.linspace(0., 10., 100) # 100 points entre 0 et 10
ypts = np.sin(xpts)              # Les sinus de ces points

plt.plot(xpts, ypts)         # tracer ypts sur xpts
plt.ylim([-1.5, 1.5])        # pour y entre -1.5 et 1.5
plt.xlabel("x")              # étiquette de l'axe des x
plt.ylabel("sin(x)")         #    ... et de l'axe des y
plt.show()                   # afficher graphique
```

Visualisation avec matplotlib

Résultat :



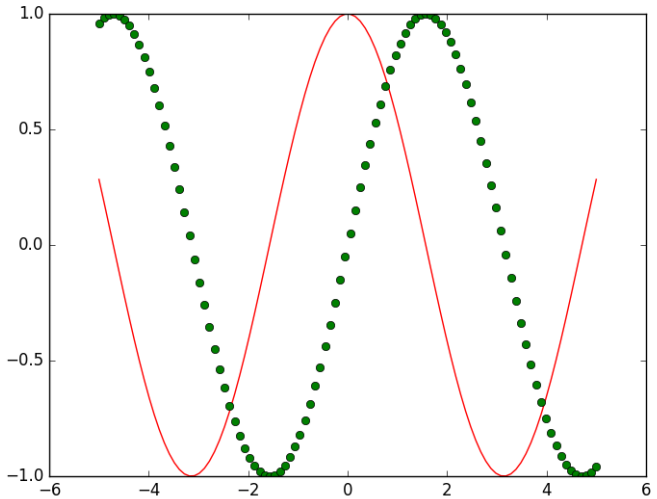
La fonction `matplotlib.pyplot.plot()`

Tracer des courbes ou des points : `matplotlib.pyplot.plot(x, y, m)`

- `x` = valeurs des x
- `y` = valeurs de fonction $y(x)$ à tracer
- optionnel : `m` = chaîne de caractères indiquant la couleur et le style du marqueur ou de la courbe, par exemple
  - `'r'`, `'g'`, `'b'` = rouge, vert, bleu (défaut)
  - `'-'`, `'--'`, `':'` = ligne solide (défaut), interrompue, pointillée
  - pour tracer des points individuels plutôt qu'une courbe :
    `'.'`, `','`, `'o'`, `'*'`, `'s'` = marqueur point, pixel, cercle, étoile, carré

Exemple :

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(-5, 5, 100)
cosx = np.cos(x)
sinx = np.sin(x)
plt.plot(x, cosx, 'r')
plt.plot(x, sinx, 'go')
plt.show()
```



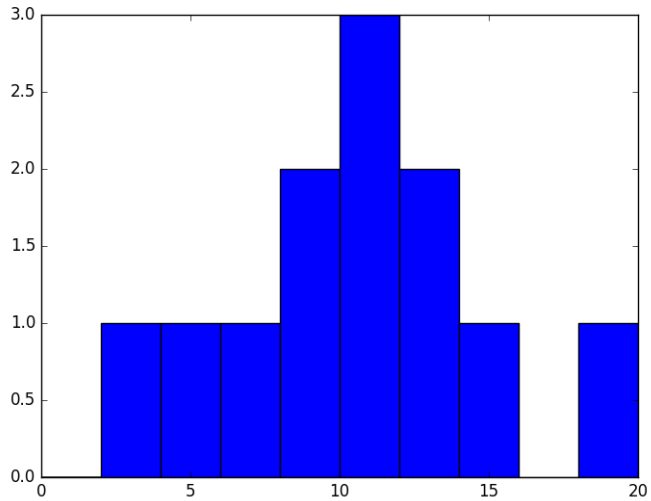
La fonction `matplotlib.pyplot.hist()`

Tracer des histogrammes : `matplotlib.pyplot.hist(x, bins, range)`

- `x` = valeurs à tracer
- optionnel : `bins` = nombre de barres
- optionnel : `range` = tuple `(x_minimal, x_maximal)`

Exemple :

```
import numpy as np
import matplotlib.pyplot as plt
notes = [13.5, 5.75, 10., 11.25,
         18., 7.5, 13., 8.75,
         10.5, 14., 9.25, 3.25]
plt.hist(notes, 10, (0, 20))
plt.show()
```



Tracer des fonctions de deux variables

Pour tracer une fonction $z(x, y)$ il faut d'abord créer un maillage (anglais : "meshgrid") pour représenter les binômes de coordonnées (x, y). Il convient de se servir de la fonction `numpy.meshgrid(x, y)`. Exemple :

```
import numpy as np

# 101 valeurs de x entre 0 et 10: [0.0 0.1 0.2 ... 10]
x = np.linspace(0, 10, 101)

# 10 valeurs de y,  3 <= y < 4: [3.0 3.1 3.2 ... 3.9]
y = np.arange(3.0, 4.0, 0.1)

X, Y = np.meshgrid(x, y)
```

Résultat : deux tableaux 10×101,

$$\mathtt{X} = \left.\begin{pmatrix} 0 & 0.1 & 0.2 & \dots & 10 \\ 0 & 0.1 & 0.2 & \dots & 10 \\ & & \dots & & \end{pmatrix}\right\} 10 \text{ lignes}, \quad \mathtt{Y} = \underbrace{\begin{pmatrix} 3 & 3 & \dots & 3 \\ 3.1 & 3.1 & \dots & 3.1 \\ \vdots & & & \vdots \\ 3.9 & 3.9 & \dots & 3.9 \end{pmatrix}}_{101 \text{ colonnes}}$$

La fonction `matplotlib.pyplot.imshow()`

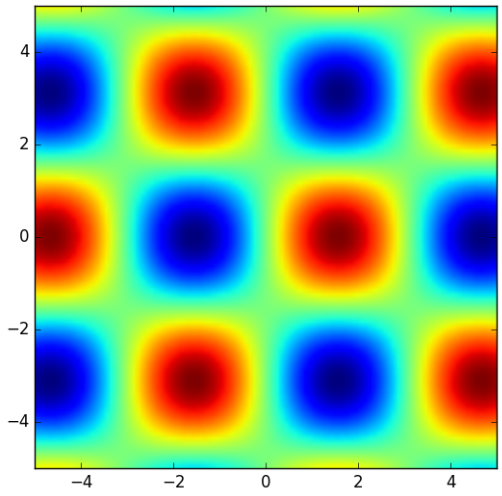
Tracer des cartes de chaleur (heat map) : `matplotlib.pyplot.imshow(z, extent)`

- `z` = tableau 2D avec les valeurs de fonction
- argument facultatif : `extent` = liste avec les `x` et `y` minimales et maximales

Exemple :

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

x = y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(x, y)
z = np.sin(X) * np.cos(Y)
plt.imshow(z, extent=[-5,5,-5,5])
plt.show()
```



La fonction `matplotlib.pyplot.contour()`

Tracer des courbes de niveau : `matplotlib.pyplot.contour(x, y, z)`

- `z` = tableau 2D avec les valeurs de fonction
- arguments facultatifs : `x`, `y` = tableaux avec les `x` et `y` correspondants

Exemple :

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

def f(x, y):
  if x == 0 and y == 0:
    return 0.0
  return x**2 * y / (x**4 + y**2)
vf = np.vectorize(f)
x = y = np.linspace(-2, 2, 200)
X, Y = np.meshgrid(x, y)
Z = vf(X, Y)
plt.contour(X, Y, Z)
plt.show()
```