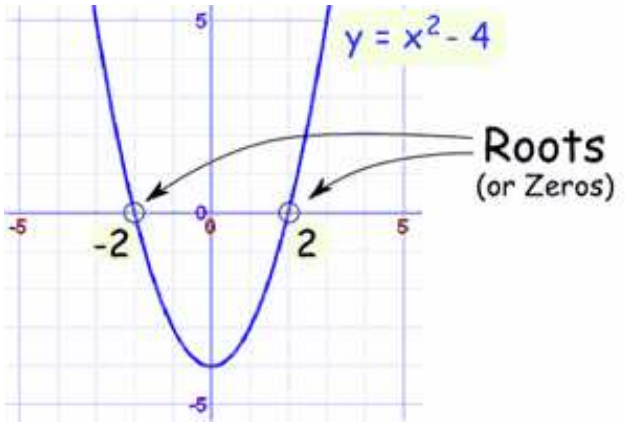
Recherche de zéro

Alessandro Torcini et Andreas Honecker

LPTM

Université de Cergy-Pontoise

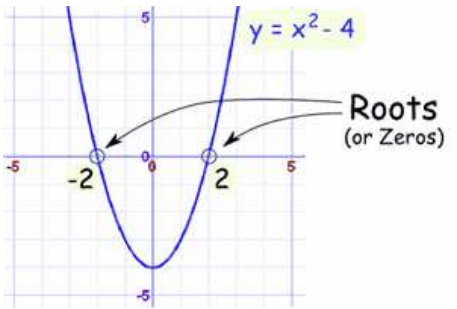


Recherche de zéro

Dans beaucoup des contextes il faut résoudre des équations non-linéaires et s'ils n'ont pas de solution analytique, on a encore une fois besoin de méthodes numeriques.
Supposons d'abord que nous avons une fonction $f : \mathbb{R} \to \mathbb{R}$ et nous cherchons un ou plusieurs zéros x_0 :

$$f(x_0) = 0 \tag{1}$$

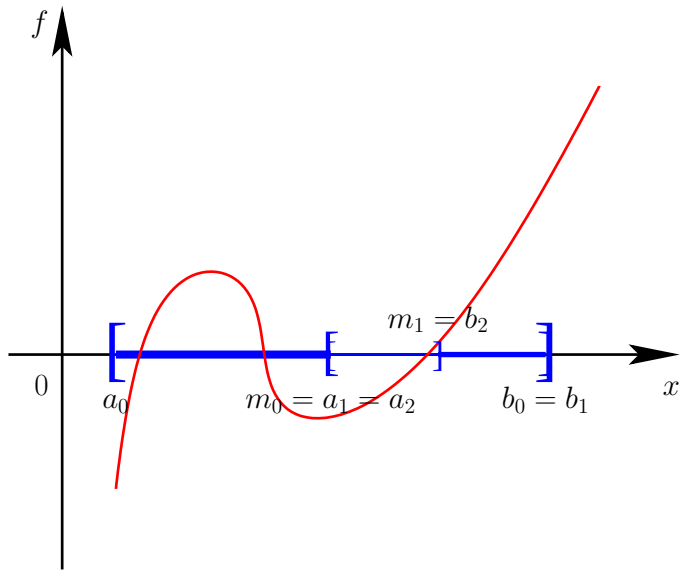
En premier lieu, il faut qu'on connait quelques propriétés de la fonction f.



1. L'équation (1) peut avoir pas du solution, une solution ou plusieurs solutions.
2. Si l'équation a plusieurs solutions, le(s)quelle(s) cherchons-nous ? Peut-être on peut répondre qu'on prend toutes, mais l'équation $\sin x = 0$ a des solutions $x_n = \pi\, n$ avec $n \in \mathbb{Z}$ et il est impossible de les trouver tous de manière numérique.
3. Même si l'équation (1) a seulement une solution, les conditions initiales peuvent jouer une rôle importante pour la recherche de zéro. Bref, si vous ne connaissez pas la fonction f très bien, il faudra mieux commencer avec une trace de f.

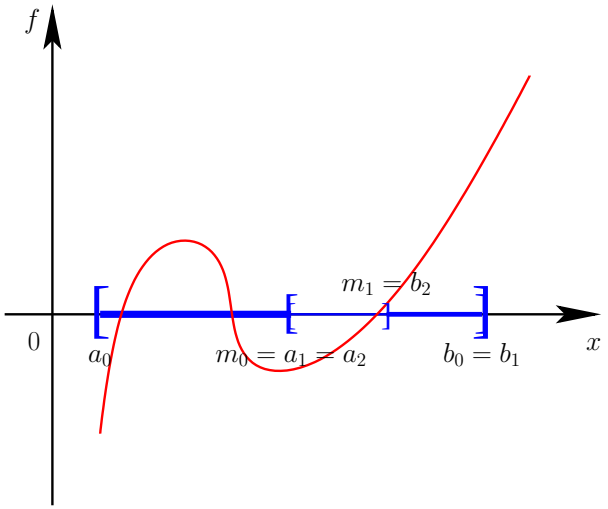
Recherche de zéro par dichotomie

Supposons que nous avons $a_0 < b_0 \in \mathbb{R}$ avec $f(a_0) < 0$ et $f(b_0) > 0$. Si la fonction f est continue sur l'intervalle $[a_0, b_0]$ nous savons que la function $f(x)$ au moins un zéro dans cet intervalle



La condition de continuité est nécessaire pour assurer cette conclusion. Prenons par exemple la fonction $f(x) = 1/x$. $f(-1) = -1 < 0$ et $f(1) = 1 > 0$, mais $f(x)$ n'a pas de zéro dans l'intervalle $[-1, 1]$, seulement une singularité à $x = 0$

Recherche de zéro par dichotomie



Nous voulons trouver le zéro de la fonction avec une certaine précision requise, par exemple :

$$\varepsilon = 10^{-9}$$

Maintenant on peut répéter les règles suivantes :

1. Calculer la moyenne
$$m_n = \frac{a_n + b_n}{2}.$$
2. Quand $f(m_n) > 0$, on prend $a_{n+1} = a_n$, $b_{n+1} = m_n$, soit on remplace b.
3. Quand $f(m_n) < 0$, on prend $a_{n+1} = m_n$, $b_{n+1} = b_n$, soit on remplace a.
4. Si $|f(m_n)| \leq \varepsilon$ on peut terminer, autrement le processus peut redémarrer à partir du point 1

Recherche de zéro par dichotomie

Ces règles définissent la méthode de dichotomie.
Pendant chaque pas, on prend la moitié d'intervalle, même si c'est pas très vite, on gagne d'information sur la position d'un zéro de f : après n pas, la longueur d'intervalle $[a_n, b_n]$ est

$$b_n - a_n = \frac{b_0 - a_0}{2^n} .$$

Donc la précision requise $\varepsilon = 10^{-K}$ est atteinte en m étapes,

$$m \simeq \frac{K}{\log_{10}(2)} = \frac{K}{0.301} = 3.32K$$

si $|b_0 - a_0| \sim \mathcal{O}(1)$.

Démonstration

Si

$$b_m - a_m = \frac{b_0 - a_0}{2^m} = \varepsilon = 10^{-K}$$

alors

$$-K = \log_{10}(b_0 - a_0) - m \log_{10}(2) \simeq -m \log_{10}(2)$$

Recherche de zéro par dichotomie

Exercice :

Prenez la fonction

$$f(x) := x^2 - 2\,.$$

Créez d'abord une trace de la fonction $f(x)$. Vérifiez que f a un zéro dans l'intervalle $[a_0, b_0] = [0, 2]$. Évaluez le nombre d'itérations m nécessaires pour atteindre la précision requise $\varepsilon = 10^{-K}$ avec $K = 3, 6, 9, 12$, enfin comparez votre résultat avec le résultat exact $\sqrt{2}$.

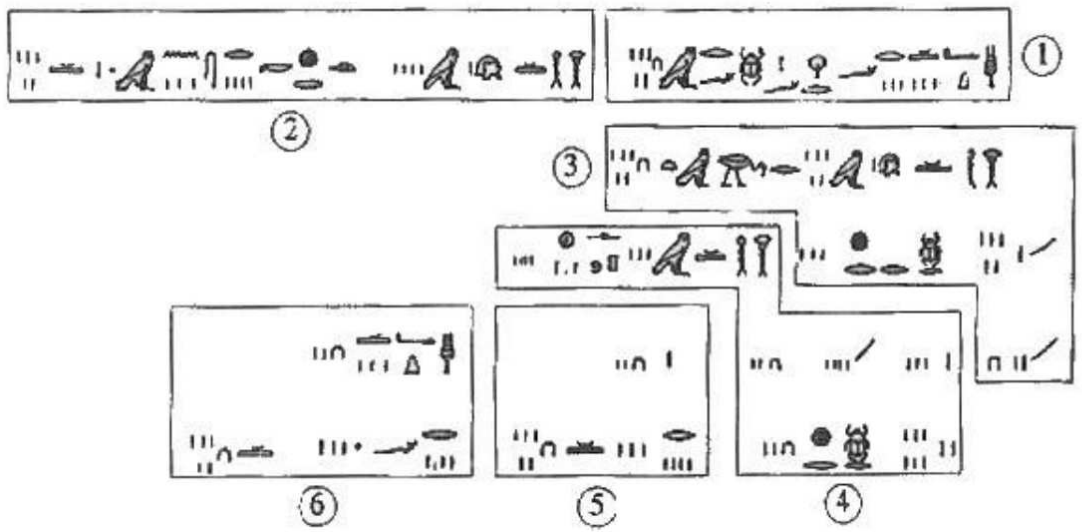
Recherche de zéro par dichotomie

```
import numpy as np                  # importer le module NumPy comme "np"
def f(x):
  return x*x-2                      # la fonction

prec=1.e-4                        #  précision  10^(-K)
a = 0                             # intervalle initiale
b = 2
if f(a) < 0 and f(b) > 0:  #f(0)=-2 et f(2)=2, nous avons un zero dans [0,2]
  print "l'intervalle est bon", "precision =", prec

for n in range(0,200):            # 200 iterations
  m = 0.5*(a+b)
  if f(m) < 0:                    # dichotomie :
     a = m                        # remplacer a si f(m) < 0
  else:
     b = m                        # autrement remplacer b
  dd=abs(b-a)
  if dd < prec:
    print "Nombre d'itérations pour atteindre la précision = ",n
    break
```

Méthode de la sécante

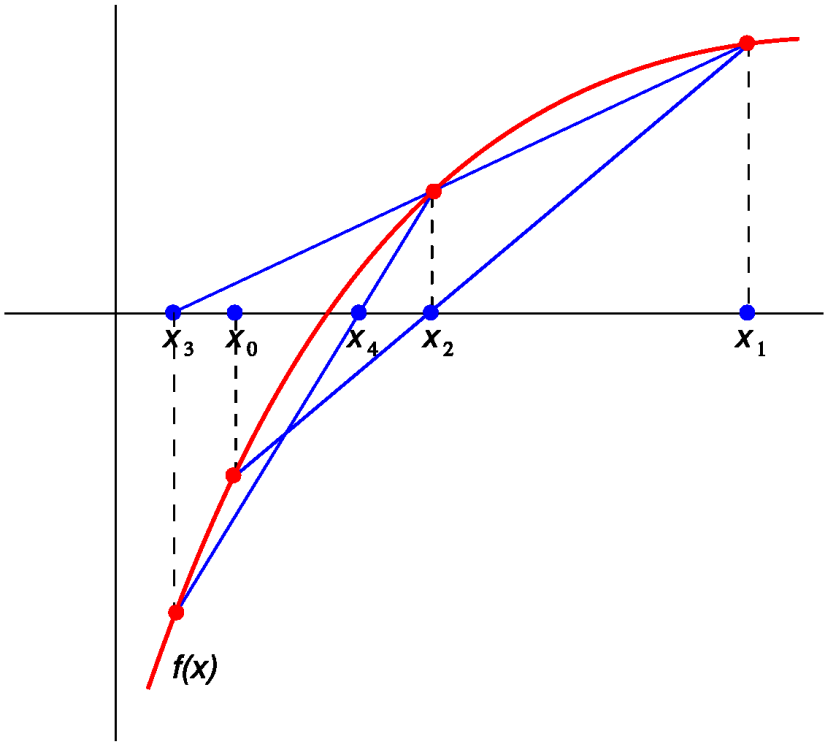
La méthode de dichotomie est lente. En plus, on peut seulement utiliser cette méthode quand on connait des points a_0 et b_0 où les signes de $f(a_0)$ et $f(b_0)$ sont différents.

La méthode de la sécante est très anciennne (env. 2000 av. J.C). . Cela vien du papyrus Rhind, qui est un célèbre papyrus de la deuxième période intermédiaire qui a été écrit par le scribe Ahmès. Depuis 1865 il est conservé au British Museum (à Londres). Il est une des sources les plus importantes concernant les mathématiques dans l'Égypte antique.



Papakonstantinou, J. M., & Tapia, R. A. (2013). Origin and evolution of the secant method in one dimension. American Mathematical Monthly, 120(6), 500-517.

Méthode de la sécante



La méthode de la sécante,suppose que $f(x)$ est presque linéaire dans l'intervalle $[x_{n-1}, x_n]$. Étant donnés x_{n-1} et $x_n]$ on construit la droite passant par $(x_{n-1}, f(x_{n-1}))$ et $(x_n, f(x_n))$ (la sécante de la courbe) pour trouver l'intersection avec l'axe zéro de la sécante : x_{n+1}.

x_{n+1} représente l'approximation du zéro de la fonction $f(x)$

$$(x_0, x_1) \to x_2 \qquad (x_1, x_2) \to x_3 \qquad (x_2, x_3) \to x_4$$

Méthode de la sécante

Pour être plus précis, la sécante est donnée par

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

et la solution de $s_f(x_{n+1}) = 0$ la relation de récurrence :

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

L'initialisation nécessite deux points x_0 et x_1, proches, si possible, de la solution recherchée. Il n'est pas nécessaire, contrairement à la méthode de dichotomie, que x_0 et x_1 encadrent une racine de f.

Exercice : Cherchez encore une fois la racine da la fonction

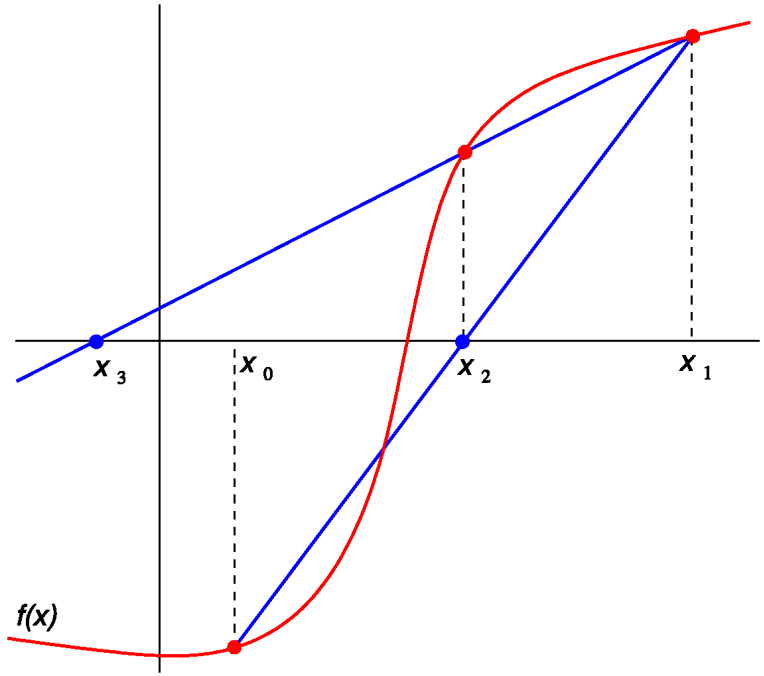
$$f(x) := x^2 - 2\,.$$

Essayez l'intervalle $[x_0, x_1] = [0, 2]$ et faites au plus 10 itérations de la méthode de la sécante et affichez x_n. Comparez x_{10} avec le résultat numérique de $\sqrt{2}$.

Méthode de la sécante

```
import numpy as np                  # importer le module NumPy comme "np"
def f(x):
  return float(x*x-2)               # la fonction (assurer flottantes !)

xvieux, xactuel = 0, 2

n = 0
while n<10 and xactuel != xvieux:     # 10 iterations maximum
  # ... et terminer quand xactuel=xvieux
  print "x(", n, ") =", xactuel
  fvieux = f(xvieux)                  # f(x(n-1)) - x(n-1) est dans xvieux
  factuel = f(xactuel)                # f(x(n)) - x(n) est dans xactuel
  # formule pour x(n+1):
  xnouveau = (factuel*xvieux-fvieux*xactuel)/(factuel-fvieux)
  xvieux = xactuel                    # stocker x(n) dans xvieux
  xactuel = xnouveau                  # et x(n+1) dans xactuel
  n += 1                              # prochaine iteration

print "x(", n, ") =", xactuel
r2 = np.sqrt(2)
```

Méthode de la sécante



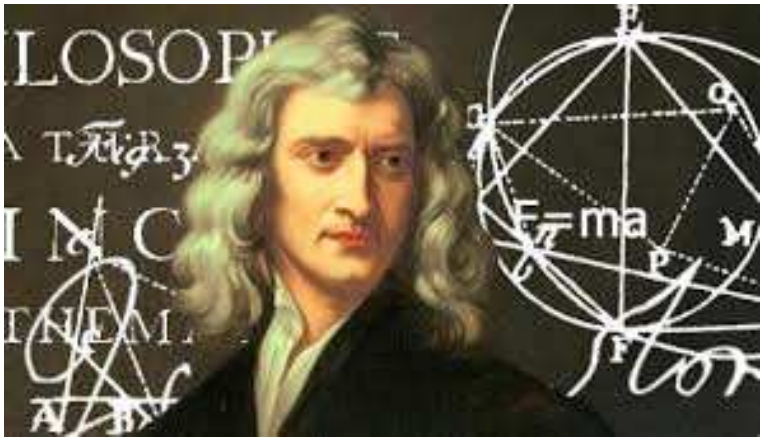
Évidemment, la méthode de la sécante est très vite quand elle converge, mais elle ne converge pas toujours, même s'il y'a un zéro dans l'intervalle $[x_0, x_1]$.

Exercice : Cherchez la racine de la fonction

$$f(x) := x\ \exp(-x^2)\,.$$

Essayez les intervalles $[x_0, x_1] = [0, 2]$, $[-1, 1]$ et $[-1, 3]$. Faites toujours au plus 10 itérations de la la méthode de la sécante et affichez x_n.

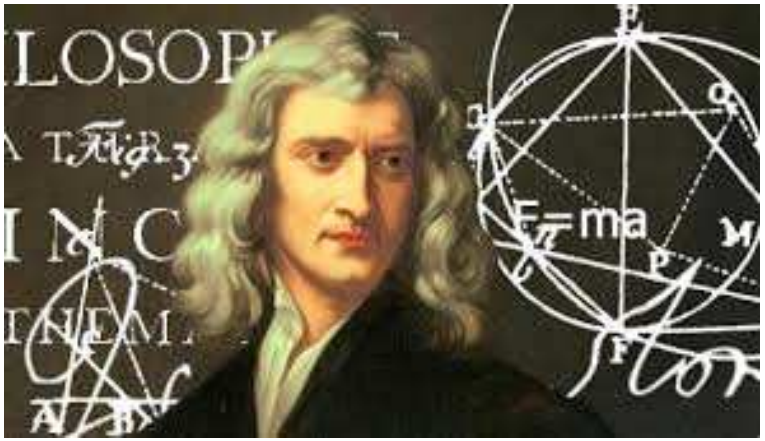
Méthode de Newton-Raphson



La méthode de résolution des équations numériques a été initiée par Isaac Newton vers 1669 sur des exemples numériques mais la formulation était assez compliqué.

1. En 1680, Joseph Raphson met en évidence une formule de récurrence.
2. Un siècle plus tard, Mouraille et Lagrange étudient la convergence des approximations successives en fonction des conditions initiales par une approche géométrique.
3. Cinquante ans plus tard, Fourier et Cauchy s'occupe de la rapidité de la convergence.

Méthode de Newton-Raphson



Si on prend la limite $x_{n-1} \to x_n$ de la méthode de la sécante, on arrive à la méthode de Newton-Raphson.

La sécante passant pour les deux points $(x_{n-1}, f(x_{n-1})$ et $(x_n, f(x_n))$ est donnée par

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

si on prend la limite $x_{n-1} \to x_n$ de le coefficient directeur de $s_f(x)$ on obtient le dérivé in x_n

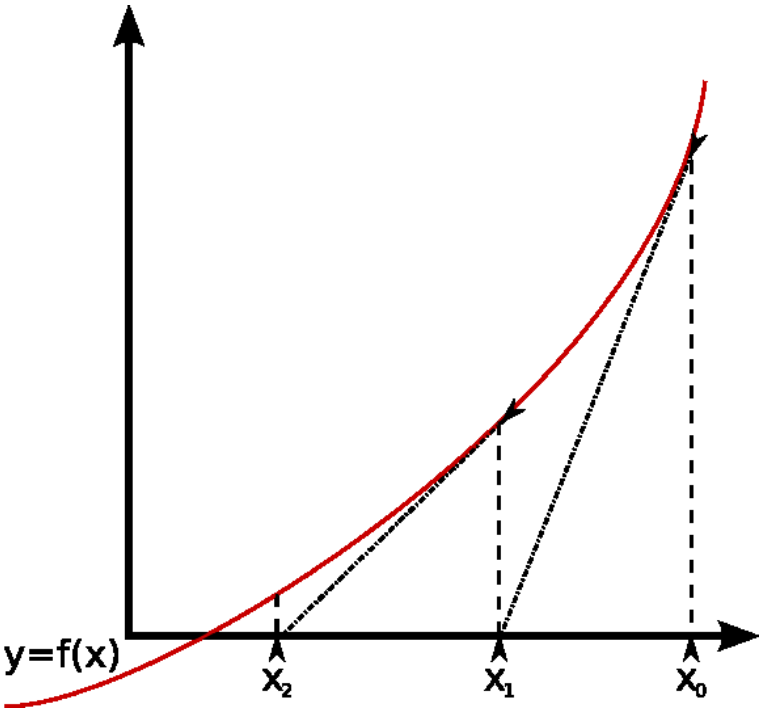
$$\lim_{x_{n-1} \to x_n} \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}} = \frac{df}{dx}(x_n) = f'(x_n)$$

et donc

$$\lim_{x_{n-1} \to x_n} s_f(x) \longrightarrow t_f(x) = f(x_n) + (x - x_n)\,f'(x_n)$$

au lieu de la sécante, on utilise la tangente au point x_n

Méthode de Newton-Raphson

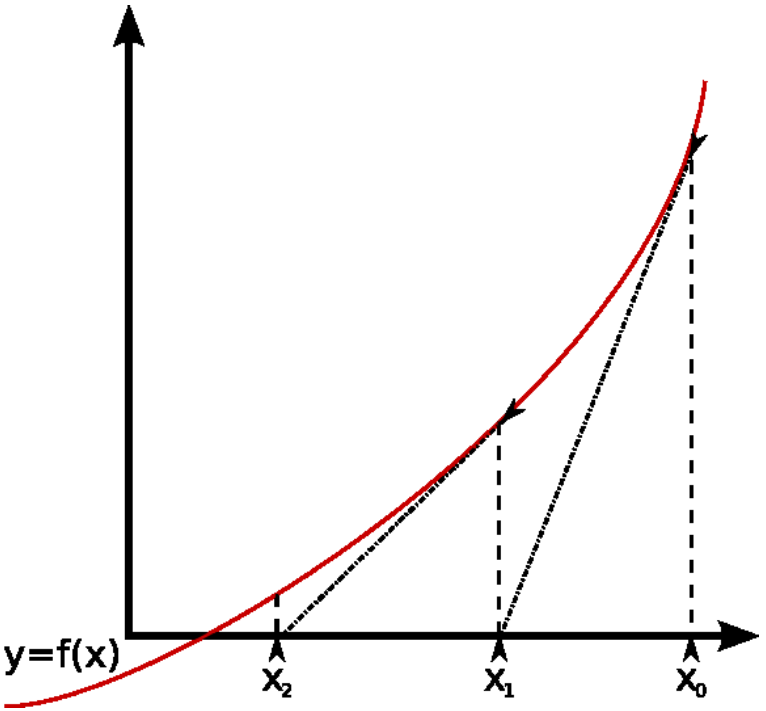


1. L'équation de la tangente en x_n est $t_f(x) = f(x_n) + (x - x_n)\, f'(x_n)$
2. x_{n+1} est l'abscisse du point d'intersection de la tangente t_f en x_n avec l'axe des abscisses.
3. Cette tangente coupe l'axe des abscisse quand $t_f(x_{n+1}) = 0$
4. $f(x_n) + (x_{n+1} - x_n)\, f'(x_n) = 0 \Longrightarrow$ $(x_{n+1} - x_n)\, f'(x_n) = -f(x_n)$
5. $(x_{n+1} - x_n) = -\frac{f(x_n)}{f'(x_n)}$

On a donc la relation de récurrence suivante :

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

Méthode de Newton-Raphson



La méthode consiste à introduire une suite $\{x_n\}$ d'approximation successives de l'équation $f(x) = 0$

1. On part d'un x_0 proche de la solution.
2. À partir de x_0 , on calcule un nouveau terme x_1 de la manière suivante : on trace la tangente à f en x_0. Cette tangente coupe l'axe des abscisses en x_1 comme indiqué sur le figure.
3. On réitère ce procédé en calculant x_2 en remplaçant x_0 par x_1
4. puis x_3 en remplaçant x_1 par x_2 et ainsi de suite . . .

Conditions d'application

Pour que la suite $\{x_n\}$ existe :

1. La fonction f doit être dérivable en chacun des points considérés. En pratique la fonction doit être dérivable dans un intervalle centré en α (la solution) et contenant x_0
2. La dérivée ne doit pas s'annuler sur cet intervalle.
3. En pratique, il faut prendre un x_0 assez proche de la valeur α qui annule la fonction.

Avantages et inconvénients de la méthode

1. Un avantage de cette méthode est qu'un seul point x_n suffit.
2. En revanche il nous faut maintenant une expression analytique de la dérivée de $f'(x)$ (en principe, on peut utiliser une dérivation numérique, mais si on fait ça, on revient à une version de la méthode de la sécante).

Algorithme

1. Lorsque la suite converge, elle converge de façon quadratique c'est à dire que le nombre de chiffres significatifs double à chaque itération
2. Si l'on s'en tient à une précision inférieure à 10^{-15} (double pécision - représentation des flottantes en Python) , la suite doit alors converger en moins de 10 itérations
3. On pourra mettre une condition d'arrêt de l'algorithme lorsque le nombre de boucle dépassera 10 car alors la suite ne converge pas. Il faudra alors prendre un x_0 plus proche de α
4. On prendra comme critère d'arrêt pour une précision de p :

$$|x_n - x_{n+1}| = \left| \frac{f(x_n)}{f'(x_n)} \right| < 10^{-p}$$

5. Pour utiliser cet algorithme, il faudra calculer la fonction dérivée f'

Exercice : Cherchez encore une fois la racine de la fonction $f(x) = x\mathrm{e}^{-x^2}$ mais maintenant avec la méthode de Newton-Raphson. Essayez $x_0 = 0.4, 0.5$ et 0.6. Faites toujours 10 itérations.

La méthode Newton-Raphson

```
import numpy as np                  # importer le module NumPy comme "np"

def f(x):                           # la fonction
  return x*np.exp(-x*x)

def df(x):                          # et sa derivee
  return (1-2*x*x)*np.exp(-x*x)

x = 0.4                             # le debut de la recherche
precision = 1.e-4                  # Précision requise
n, diff=0 , 33.

while n<10 and diff > precision: # 10 iterations maximum
  pas = f(x)/df(x)                  # L'incrément
  x -= pas # iteration
  diff=abs(pas)
  print "x(", n, ") =", x, "  L'incrément ",      diff # afficher x(n)
  n += 1

print "resultat final : ", x
```

Un exemple

Prenons l'exemple historique qu'avait pris Newton pour expliquer sa méthode :
Déterminer une approximation de la solution de :

$$x^3 - 2x - 5 = 0$$

1. On pose la fonction $f(x) = x^3 - 2x - 5$
2. La fonction f est dérivable (car polynôme) et : $f'(x) = 3x^2 - 2$
3. La dérivée est nulle in $f'(x) = 3x^2 - 2 = 0 \Longrightarrow x^2 = \frac{2}{3}$
   $\Longrightarrow x = \pm\sqrt{\frac{2}{3}} \simeq \pm 0.816$
4. On obtient le tableau de variation suivant :

x	$-\infty$		$-\sqrt{\frac{2}{3}}$		$\sqrt{\frac{2}{3}}$		$+\infty$
$f'(x)$		$+$	0	$-$	0	$+$	
$f(x)$	$-\infty$	↗	$\approx -3{,}911$	↘	$\approx -6{,}089$	↗	$+\infty$

Un exemple

x	$-\infty$		$-\sqrt{\frac{2}{3}}$		$\sqrt{\frac{2}{3}}$		$+\infty$
$f'(x)$		$+$	0	$-$	0	$+$	
$f(x)$	$-\infty$	↗	$\approx -3{,}911$	↘	$\approx -6{,}089$	↗	$+\infty$

1. Si $x \in]-\infty; +\sqrt{\frac{2}{3}}[$, alors $f(x) < 0$. La fonction ne peut s'annuler.
2. Si $x \in [+\sqrt{\frac{2}{3}}; +\infty[$ la fonction f est continue (car dérivable), monotone et $f(x) \in [-6.089; +\infty[$ et donc il existe un unique solution ou $f(\alpha) = 0$
3. On peut affiner l'intervalle de α : $f(2) = -1$ et $f(3) = 16$ donc $\alpha \in [2; 3]$

La fonction f ne s'annule qu'une seule fois et la solution $\alpha \in [2; 3]$

Un exemple

1. On peut utiliser pour x_0 soit 2 soit 3, mais f (2) est plus proche de 0, sa convergence est plus rapide.
2. Si l'on effectue l'algorithme à la main, on a le tableau suivant pour une précision de 10^{-3}
3. On s'aperçoit de la redoutable efficacité de cet algorithme car en deux termes il arrive à une précision de 10^{-3}

n	x_n	$f(x_n)$	$f'(x_n)$	$-\frac{f(x_n)}{f'(x_n)}$	x_{n+1}
0	2	-1	10	0,1	2,1
1	2,1	0,061	11,23	$-0{,}005\,43$	2,094 57
2	2,094 57	2×10^{-4}	11,16	-2×10^{-5}	2,094 55

On peut comparer cet algorithme avec l'algorithme de dichotomie à l'aide du nombre de boucles que le programme effectue pour une précision donné.

Précision	10^{-3}	10^{-6}	10^{-9}
Newton	2	3	3
Dichotomie	10	20	30

Nbre de boucles pour une précision donnée

Méthode de Newton-Raphson pour systèmes

Supposons que nous cherchons la solution simultanée de N équations non-linéaires dans N variables, donc la solution du système suivante

$$\begin{cases} F_1(x_1, x_2, x_3, \ldots, x_N) = 0 \\ F_2(x_1, x_2, x_3, \ldots, x_N) = 0 \\ \ldots \\ F_N(x_1, x_2, x_3, \ldots, x_N) = 0 \end{cases}$$

La méthode de Newton-Raphson se généralise facilement aN variables. Si on regroupe les équations F_i et les variables x_j, $i, j = 1, 2, \ldots N$ dans des vecteurs $\vec{F}$ et $\vec{x}$, nous pouvons réécrire le système comme :

$$\vec{F}(\vec{x}) = 0$$

On peux d'abord étendre la fonction $\vec{F}(\vec{x})$ au premier ordre avec la série de Taylor

$$\vec{T}_F(\vec{x}) = \vec{F}(\vec{x}_n) + J_F(\vec{x}_n)\,(\vec{x} - \vec{x}_n)$$

où la tangente a été remplacé par le vecteur $\vec{T}_F(\vec{x})$ et le dérivé par la matrice jacobienne J_F

Méthode de Newton-Raphson pour systèmes

La matrice jacobienne de $\vec{F}$ est

$$J_F = \begin{pmatrix} \frac{\partial F_1}{\partial x_1} & \cdots & \frac{\partial F_1}{\partial x_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_N}{\partial x_1} & \cdots & \frac{\partial F_N}{\partial x_N} \end{pmatrix} .$$

Après on peut résoudre l'équation $\vec{T}_F(\vec{x}_{n+1}) = 0$ et on arrive à la récurrence suivante

$$\vec{x}_{n+1} = \vec{x}_n - J_F^{-1}(\vec{x}_n)\, \vec{F}(\vec{x}_n) .$$

Cette équation est très semblable à celui pour la fonction scalaire, mais maintenant il s'agit de vecteurs et de matrices

Nous avons besoin de trouver l'inverse d'une matrice, nous avons besoin de l'algèbre linéaire en Python

Algèbre linéaire en Python

Je le fais avec un exemple d'un système linéaire de type $A\,\vec{x} = \vec{b}$:

$$A \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 5 \\ 6 \end{pmatrix} .$$

Nous cherchons les solutions du système

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = A^{-1} \begin{pmatrix} 5 \\ 6 \end{pmatrix} .$$

Nous utilisons encore une fois le module NumPy et traitons la matrice A et le vecteur $\vec{b}$ comme des `array`, C'est très facile avec Python

Algèbre linéaire en Python

```
import numpy as np
A = np.array([[1,2],[3,4]])  #   une matrice
b = np.array([5,6])          #    un vecteur
print "A*b =", A.dot(b)      #     A*b
Ainv = np.linalg.inv(A)      # calculer l'inverse d'A
print "A^{-1} =", Ainv
x = Ainv.dot(b)

print "solution de A*x=b :", x    # la solution de A*x=b
```

Les éléments du code ci-dessus sont :

1. A_1`.dot(`A_2`)` calcule le produit des deux `array` A_i ; ici c'est le produit d'une matrice et un vecteur.
2. La fonction `linalg.inv(`A`)` calcule l'inverse A^{-1} d'une matrice A.

D'abord, la notation des `array` est très similaire à des listes (ou des listes de listes). En particulier `b[0]` vous rend le premier élément b_1 de $\vec{b}$ et `A[1][0]` vous rend l'élément $A_{2,1}$ (rappel : en Python on commence à compter avec 0).

Un exemple bidimensionnel

Exercice : Cherchez les trois racines de deux équations suivantes :

$$\begin{cases} x^3 - 3\,x\,y^2 - 1 = 0 \\ 3\,x^2\,y - y^3 = 0 \end{cases}$$

Utilisez la méthode de Newton-Raphson et faites toujours 10 itérations. Essayez plusieurs points (x_0, y_0) pour commencer la recherche.

La méthode Newton-Raphson en 2 dimensions

```
import numpy as np                  # importer le module NumPy comme "np"

def F(xv):                          # la fonction
  x, y = xv[0], xv[1]               # les deux coordonnees
  return np.array([x**3-3*x*y**2-1,3*x**2*y-y**3])

def JF(xv):                         # et sa derivee, soit la matrice jacobienne
  x, y = xv[0], xv[1]               # les deux coordonnees
  return np.array([[3*x*x-3*y*y,-6*x*y],[6*x*y,3*x*x-3*y*y]])

precision=1.e-3
x = np.array([8.0,0])               # le point pour demarrer la recherche
```

La méthode Newton-Raphson en 2 dimensions

```
n, pas2 = 0, 4e4
# 10 iterations maximum et on peut s'arreter quand on ne fait plus des pas
while n<10 and pas2 > precision:
  print "x(", n, ") =", x, "pas2",pas2        # afficher x(n)
  # le pas (notez le produit de l'inverse de la matrice avec la "fonction")
  pas = np.linalg.inv(JF(x)).dot(F(x))
  x -= pas # iteration
  pas2 = np.sqrt(pas.dot(pas))
# pas2 est la racine carrée de||pas||^2,
# soit le module du produit scalaire de pas avec soi-meme
  n += 1

print "resultat final : ", x
```