RESOLUTION APPROCHEE D'UNE EQUATION SUR LES REELS

Le but de ce chapitre est de déterminer des méthodes d'évaluation numériques d'une solution d'équation du type $f(x) = 0$ avec f une fonction de $\mathbb{R}$ vers $\mathbb{R}$, lorsque l'on sait qu'une solution existe.

I) Méthode de dichotomie

Principe de la dichotomie

On sait qu'une fonction continue f sur un intervalle [a, b], à valeurs réelles et pour laquelle $f(a)$ et $f(b)$ sont de signe opposé , s'annule entre a et b. Le principe de dichotomie a pour but de "découper en deux" l'intervalle [a, b] de façon à se retrouver dans les mêmes conditions que précédemment, c'est-à-dire, un intervalle $[a_1, b_1]$ inclus dans [a, b], de longueur moitié du précédent et sur lequel f s'annule. En réitérant le procédé, on se trouvera dans un intervalle de longueur aussi petite que l'on veut et dans lequel on est sûr d'avoir une solution de notre équation initiale.

Pour assurer la convergence du procédé vers une solution approchée, il est essentiel de vérifier à chaque étape que les valeurs prises par f aux bornes de l'intervalle choisi soient bien opposées.

Prenons un exemple. On considère la fonction définie sur [0, 3] par : $f(x) = x^2 - 4$.

Par des arguments de continuité et le fait que $f(0) = -4 < 0 < 5 = f(3)$, on sait que l'équation $f(x) = 0$ possède une solution l (et en fait une seule), dans l'intervalle [0, 3]. Déterminons une valeur approchée d'une telle solution (…).

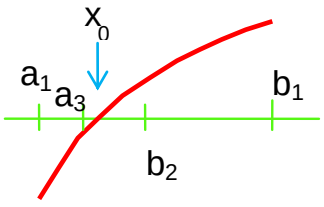
On évalue $f(1.5)$ car 1.5 est la valeur médiane dans [0, 3].

- On trouve : $f(1.5) = -1.75 < 0 < 5 = f(3)$ et donc l est dans [1.5 , 3].
- $f(2.25) = 1.0625 > 0 > f(1.5) = -1.75$ et donc l est dans [1.5 , 2.25].
- $f(1.875) = -0.484375 < 0 < f(2.25)$ et donc l est dans [1.875 , 2.25].
- $f(2.0625) = 0.25390625 > 0 > f(1.875)$ et donc l est dans [1.875 , 2.0625].
- $f(1.96875) = -0.1240234375 < 0 < f(2.0625)$ et donc l est dans [1.96875 , 2.0625].

On déduit de ces calculs que l vaut 2.015625 à 0.046875 près… On préférera écrire : $l = 2.01$ à 6.10^{-2} près

Principe théorique

Elle s'applique à un intervalle I = [a,b] où la fonction f a un zéro et un seul x_0 et est monotone. Cas usuel $f(a)\,f(b) < 0$ et f ' de signe constant sur]a,b[.



On suppose ici f ' >0 donc f croissante.

Posons $I_0 = [a,b]$. Soit d quelconque: $d \in]a,b[$, formons f(d).

Si f(d)=0 d est la racine cherchée.

Si $f(d) \neq 0$ Soit $f(d)<0$ et $x_0 \in]d,b[$ et on note $I_1 = [d,b]$

Soit $f(d)>0$ et $x_0 \in]a,d[$ et on note $I_1 = [a,d]$

On a ainsi déterminé I_1 tel que $I_1 \subset I_0$, $I_1 \neq I_0$ et $x_0 \in I_1$.

On réitère l'opération sur I_1, d'où I_2,... puis I_n tels que $I_n \subset I_{n-1} \subset ... \subset I_1 \subset I_0$ et $x_0 \in I_n$.

On a fabriqué deux suites de segments emboîtés d'extrémités a_n , b_n telles que $\forall n \in \mathbb{N}^*$, $a_n \leq x_0 \leq b_n$.

Donc $y_n = \frac{1}{2}(a_n + b_n)$ est une valeur approchée de x_0 avec une erreur inférieure ou égale à $\frac{1}{2}\delta_n$

où $\delta_n = b_n - a_n$ diamètre de I_n.

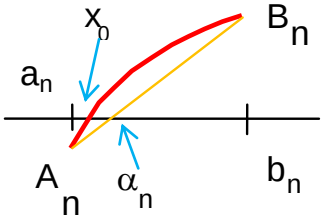
Cas particulier usuel: a et b entiers consécutifs (b-a=1).

On part de $I_0 = [a,b]$, puis de $\alpha_0 = \frac{a+b}{2}$, d'où $I_1 = [a_1, b_1]$, $\alpha_1 = \frac{a_1 + b_1}{2}$, d'où I_2 …

Au rang n, $\delta_n = \frac{1}{2^n}$. Donc

$\alpha_n = \frac{a_n + b_n}{2}$ est une valeur approchée de x_0 à $\frac{1}{2^{n+1}}$ près.

Amélioration possible de la méthode



En confondant le graphe Γ de f avec le segment $[A_n, B_n]$ au voisinage de la racine et en utilisant le théorème de Thalès, on peut considérer que x_0 divisera $[a_n, b_n]$ dans le rapport $\frac{|f(a_n)|}{|f(b_n)|}$ et on repartira avec une valeur α_n satisfaisant approximativement cette condition. On se limitera dans notre étude du cas général.

Algorithme

On arrête le processus lorsque le diamètre de l'intervalle est inférieur à la précision demandée (voire à son double si on retourne $\frac{a_n + b_n}{2}$). On utilisera l'algorithme suivant :

Données : f , a, b, ε — Rem : ε est la précision demandée
c, d ← a, b — c et d seront les bornes courantes des intervalles $[a_n, b_n]$
Fc, Fd ← $f(c)$, $f(d)$
tant que d – c > 2 ε **faire**
    m ← (c + d) / 2 — Il faut tester si la solution est dans [c ,m] ou dans [m, d]
    Fm ← $f(m)$
    si Fm × Fc ⩽ 0 **alors** — Dans ce cas on choisit [c, m]
        d ← m
        Fd ← Fm
    sinon — Sinon on choisit [m, d]
        c ← m
        Fc ← Fm
Résultat : (c + d) / 2

Terminaison, correction et complexité

Si on part bien de f continue, a < b avec $f(a) \times f(b) < 0$, et ε > 0 (et supérieure à la précision machine), le diamètre de [c, d] est divisé par 2 à chaque étape et finit donc par être inférieur à 2 ε : **la terminaison** de l'algorithme est donc assurée.

Pour ce qui est de la **correction**, on constate que la condition $f(c) \times f(d) \leqslant 0$ est vérifiée à chaque étape et qu'elle assure que la solution cherchée est toujours dans l'intervalle [c, d].

Pour ce qui est de la **complexité**, on a déjà vu qu'après n itérations le diamètre de l'intervalle [c, d] est $\delta_n = \frac{b-a}{2^n}$. Donc, au moment où l'algorithme se termine, le nombre d'itérations est : $E\left(\log_2\left(\frac{b-a}{\varepsilon}\right)\right)$.
Pour passer d'une précision de 10^{-p} à $10^{-(p+1)}$, il faut ajouter en moyenne 3.32 itérations supplémentaires.

Programme Python

```
def dichotomie (f, a, b, epsilon) :
    assert f(a) * f(b) < = 0  and  epsilon > 0
    c, d = a , b
    fc, fd = f(c) , f(d)
    while d – c > 2 * epsilon :
        m = (c + d) / 2
        fm = f(m)
        if fc * fm <= 0 :
            d, fd = m , fm
        else :
            c , fc = m , fm
    return ( (c + d) / 2 )
```

On teste si on est bien dans le cadre voulu. Si ce n'est pas le cas, l'appel à dichotomie donne un message d'erreur

II) Méthode de Newton

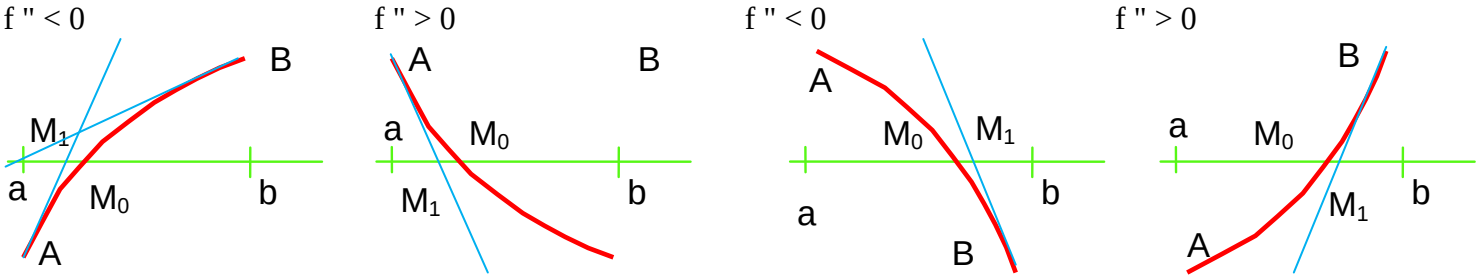
Principe et cadre de la méthode de Newton

1) Cette méthode s'applique sur l'intervalle [a, b] contenant un seul zéro de f et suffisamment petit pour que la fonction f et ses 2 premières dérivées vérifient:

Hypothèses	H1: f, f ' et f " définies sur [a, b] H2: f ' et f " sont de signe constant sur [a, b] H3: f(a) f(b) < 0
Conclusions	C1: f est monotone sur [a, b] C2: f est convexe ou concave. C3: x_0 est un zéro simple de f.

2) Principe de la méthode de Newton

f est convexe ou concave sur [a, b]. L'arc de courbe AB est d'un même côté de la corde AB et d'un même côté d'une tangente quelconque MT à l'arc AB.



On remplace l'arc AB par la tangente en M en un point quelconque et on prendra pour valeur approchée par défaut ou par excès de x_0 (abscisse de M_0), l'abscisse de l'intersection M_1 de cette tangente en M et de Ox. (On choisira pour M le point A ou le point B)

3) Calcul du terme correctif et évaluation de l'erreur dans la méthode de Newton

* Dans le cas où f(a) et f(b) sont, en valeur absolue, du même ordre: on applique la méthode de Newton à celui des deux points A et B pour lequel f et f " sont de même signe. "Règle de Fourier": choisir x tel que f(x) f " (x) ≥ 0: la valeur approchée obtenue x_1 est plus près de x_0 que le point x.

* Dans le cas où f(a) et f(b) sont nettement différents: on peut avoir intérêt à appliquer la méthode à celui des deux points le plus proche de x_0, i.e, le point où |f(x)| est minimum.

* Calcul en A: (|f(a)| très petit ou f(a) f "(a) ≥ 0)

La tangente en A a pour équation: Y – f(a) = (X – a) f '(a) donc on en déduit l'abscisse x_1 de M_1 :

$$x_1 = a - \frac{f(a)}{f'(a)} = a + h_1 \text{ où } h_1 = -\frac{f(a)}{f'(a)} \text{ terme correctif}$$

* Majorant de l'erreur de méthode: Posons $x_0 = a + k$, on a f(a+k)=0.

Or, d'après la formule de Taylor-Mac Laurin $f(a + k) = f(a) + h\ f'(a) + \frac{k^2}{2} f''(a + \theta k)$ avec $\theta \in]0, 1[$

k est donc solution de : $k = -\frac{f(a)}{f'(a)} - \frac{k^2}{2}\frac{f''(a + \theta k)}{f'(a)} = h_1 - \frac{k^2}{2}\frac{f''(a + \theta k)}{f'(a)}$

D'où en majorant : $|k - h_1| \leq \frac{k^2}{2}\frac{|f''(a + \theta k)|}{|f'(a)|}$

Soit M un majorant de $|f''|$ sur [a,b], D un majorant de k ((b – a) convient): il vient

$$\varepsilon_m = |k - h_1| \leq \frac{D^2}{2}\frac{M}{|f'(a)|} \leq \frac{(b-a)^2}{2}\frac{M}{|f'(a)|} = \varepsilon'_m$$

* Encadrement final:

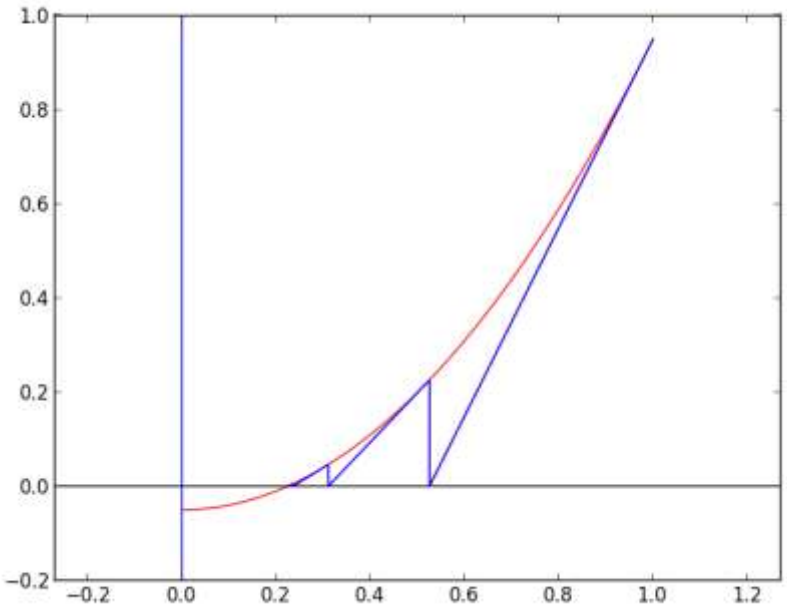
f(a) f "(a) > 0 $a + h_1 \leq x_0 \leq a + h_1 + \varepsilon'_m$

f(b) f "(b) > 0 $b + h_1 - \varepsilon'_m \leq x_0 \leq b + h_1$

On calcule les valeurs approchées de:

$a + h_1$ par défaut
$a + h_1 + \varepsilon'_m$ par excès
$b + h_1 - \varepsilon'_m$ par défaut
$b + h_1$ par excès

* En itérant le processus: On remarque que l'intervalle dans lequel on travaille n'est plus [a, b] mais [$a + h_1$, $a + h_1 + \varepsilon'_m$] dans le premier cas, ou [$b + h_1 - \varepsilon'_m$, $b + h_1$] dans le second cas… c'est-à-dire que l'erreur commise à l'itération suivante est majorée par $\frac{(\varepsilon'_m)^2}{2}\frac{M}{|f'(a)|}$: lorsque la convergence est assurée, elle est quadratique…
Sur la figure suivante, on a du mal à voir au delà des premières itérations



Reprenons l'exemple précédent. On considère la fonction définie sur [0, 3] par : $f(x) = x^2 - 4$.
Par des arguments de continuité et le fait que $f(0) = -4 < 0 < 5 = f(3)$, on sait que l'équation $f(x) = 0$ possède une solution l (et en fait une seule), dans l'intervalle [0, 3]. Déterminons une valeur approchée d'une telle solution (…).

L'équation de la tangente en 3 est : $Y = 6(X - 3) + 5$, donc le premier terme obtenu par la méthode de Newton, en partant de 3 est : $\frac{13}{6}$ soit 2.166$\underline{6}$

En réitérant le processus, on obtient respectivement : 2.0064102564…. puis 2.00001024… puis 2 à 10^{-10} près

Algorithme

On aimerait pouvoir arrêter le processus lorsque l'erreur de méthode est inférieure à la précision demandée. Malheureusement ce test n'est pas simple à réaliser. Une autre idée est de s'arrêter lorsque la distance entre le terme calculé et le précédent est inférieur à la précision demandée voire lorsque l'image par f du terme calculé est suffisamment proche de 0…

Version avec le test sur la différence de deux termes successifs

Données : f , g, u_0, ε Rem : g représente la dérivée de f

$u \leftarrow u_0$
$v \leftarrow u - f(u) / g(u)$
tant que $|v - u| > \varepsilon$ **faire**
    $u \leftarrow v$
    $v \leftarrow v - f(v) / g(v)$

Résultat : v

Version avec le test sur l'image par f de la valeur obtenue

Données : f , g, u_0, ε Rem : g représente la dérivée de f

$u \leftarrow u_0$

tant que $|f(u)| > \varepsilon$ **faire**

  $u \leftarrow u - f(u) / g(u)$

Résultat : u

Terminaison, correction et complexité

Disons-le tout de suite : il n'est pas garanti, dans le cas général, que cet algorithme se termine (par exemple il y a un risque de division par 0 dans l'expression $f(u) / g(u)$, ni, s'il se termine, qu'il propose une valeur approchée d'un zéro de f (on connait des suites divergentes pour lesquelles $u_{n+1} - u_n$ tende vers 0 par exemple ou, pour la seconde version, imaginons que f et f' s'annulent toutes les deux en x_0 , on a au voisinage de 0, $|f(x_0 + h)| = O(h^2)$: le fait que $|f(u)|$ soit inférieur à 10^{-8} ne signifie pas que u soit proche de x_0 à 10^{-8}…). Donc le contrôle de la terminaison et la correction doit être fait pour chaque fonction étudiée.

Ceci dit dans le cadre des fonctions localement concaves ou convexes, on a bien convergence de l'algorithme, et on a même une convergence quadratique : si au bout de n itérations on a une précision de 10^{-p} , l'itération suivante fournira une précision de 10^{-2p}

Reste un petit souci : l'algorithme suppose connues f et sa dérivée g. Or si on n'a pas la possibilité de calculer formellement cette dérivée, il faut pouvoir évaluer numériquement la valeur de cette dérivée.

Evaluation de la dérivée

En première idée pour calculer $f'(u)$ de manière approchée, est de constater que, pour h "petit", $f'(u)$ ne doit pas être très éloigné de $\frac{f(u+h) - f(u)}{h}$.

Plus précisément, si f est de classe $\mathscr{C}^2$ et $f''(u)$ est non nul, $\frac{f(u+h) - f(u)}{h} - f'(u) \sim h \frac{f''(u)}{2}$

On peut cependant faire mieux de façon simple. En effet si f est de classe $\mathscr{C}^3$ et $f^{(3)}(u)$ est non nul, $\frac{f(u+h) - f(u-h)}{2h} - f'(u) \sim h^2 \frac{f^{(3)}(u)}{12}$ et on gagne encore un ordre si $f^{(3)}(u)$ est nul.

Programme Python

Version avec la donnée de la dérivée

```
def newton (f, fp, x0, epsilon) :
    u = x0
    v = u – f(u) / fp(u)
    while abs(v – u) > epsilon :
        u , v = v , v – f(v) / fp(v)
    return ( v )
```

Version sans la donnée de la dérivée

```
def newton_bis (f, x0, epsilon) :
    u, h = x0, epsilon
    deriv = ( f(u + h) – f(u – h) ) /(2 * h)
    v = u – f(u) / deriv
    while abs(v – u) > epsilon :
        deriv = ( f(v + h) – f(v – h) ) /(2 * h)
        u , v = v , v – f(v) / deriv
    return ( v )
```

En fait cette méthode s'appelle alors la méthode de la sécante… qui, de fait, converge un peu moins vite que la méthode de Newton…

III) Utilisation de Numpy-Scipy

Les procédures précédentes ont évidemment été implémentées et améliorées dans les bibliothèques logicielles scientifiques Numpy et Scipy. Il est alors plus raisonnable de les utiliser. Toutefois, à titre d'illustration du fait que la procédure écrite correspond bien à l'algorithme vu, on pourra lire le code de la fonction newton du fichier ….\Python32\Lib\site-packages\scipy\optimize\zeros.py

La fonction **numpy.roots** détermine les racines d'un polynôme donné par la liste de ses coefficients (donnés dans l'ordre décroissant des puissances)

numpy.roots([2, 0, - 1]) donne la réponse : array([0.70710678, -0.70710678])

A noter que cette fonction permet également d'obtenir les racines complexes

numpy.roots([1, 0, 0, - 1]) donne : array([- 0.5 + 0.8660254j, - 0.5 - 0.8660254j, 1.0+0.j])

Mais des fois la réponse est curieuse et doit de toute façon être interprétée par l'utilisateur :

numpy.roots([1, 2, 4, 8]) donne : array([- 2.00000000e+00 + 0.j, - 6.10622664e-16 + 2.j, - 6.10622664e-16 - 2.j])

La méthode dichotomique est implémentée dans la fonction **scipy.optimize.bisect**

scipy.optimize.bisect(lambda x : x3 – 8, 0, 3)** donne la réponse : 2.0000000000002274

La méthode de Newton est implémentée dans la fonction **scipy.optimize.newton**

scipy.optimize.newton(lambda x : x3 – 8, 3, lambda x : 3*x**2)** donne la réponse : 2.0

Si on ne donne pas la dérivée, **scipy.optimize.newton** utilise la méthode de la sécante

scipy.optimize.newton(lambda x : x3 – 8, 3)** donne la réponse : 2.000000000000003

A contrario, si on donne la dérivée et la dérivée seconde, **scipy.optimize.newton** utilisera la méthode de Halley. On peut aussi utiliser **scipy.optimize.brentq** qui implémente la méthode de Brent qui mélange la dichotomie et la métode de la sécante, ou la fonction **scipy.optimize.fsolve** qui s'applique aussi à des équations à plusieurs variables….

IV) Exercices

1) **a)** Ecrire **l'algorithme de Babylone** qui consiste à déterminer une racine carrée de K par la méthode de Newton.

b) **E**n appliquant cet algorithme (et le programme associé) avec K = 1 + i (que l'on peut obtenir en écrivant K = 1 + 1j ou K = complex(1,1)) et plusieurs valeurs initiales u_0, montrer que l'algorithme de Babylone permet d'obtenir une racine carrée d'un nombre complexe … mais on n'obtient pas toujours la même.

c) **A**ppliquer la méthode précédente pour obtenir des racines carrées de plusieurs complexes…

2) Appliquer la méthode de Newton pour trouver des solutions de l'équation $x^3 + c\,x + 1 = 0$.

a) Pour c > 0, constater (puis montrer si possible) que la méthode de Newton converge pour tout choix de la valeur initiale.

b) Pour c = 0. Prédire le comportement de la méthode de Newton en partant de u0 = 0.8

c) On prend c = – 1 . La méthode de Newton converge-t-elle ?

3) Appliquer la méthode de Newton pour trouver des solutions de l'équation $x^3 - 1 = 0$. Selon les valeurs initiales, la méthode de Newton fournira la racine 1, j ou j^2 (et, dans certains cas, ne converge pas…). En colorant différemment les points selon le comportement de la méthode de Newton, dresser un graphe illustrant les bassins d'attractions de 1, de j et de j^2.