Nous allons voir dans ce chapitre comment évaluer numériquement une solution d'équation de la forme $f(x) = 0$ avec f une fonction à valeurs réelles, dans des cadres où la théorie nous assure qu'une telle solution existe, mais sans en fournir d'expression analytique. Le but est donc de trouver une solution approchée en minimisant le temps d'exécution, mais en obtenant quand même une bonne approximation.

Deux méthodes classiques de recherche de zéro sont expliquées ici : la recherche par **dichotomie** et la méthode de **Newton.** On programmera et testera ces méthodes, puis on les comparera aux performances des fonctions fournies par la bibliothèque **numpy**.

1. Méthode dichotomique

On considère une fonction f **continue** sur un intervalle **[a, b]**, à valeurs réelles, telle que $f(a) \cdot f(b) < 0$. D'après le **théorème des valeurs intermédiaires**, la fonction f s'annule au moins une fois sur l'intervalle **[a, b]**.

L'idée est dans cette méthode de découper en deux l'intervalle de recherche. On note $m = \frac{a+b}{2}$. Et on cherche à savoir si le zéro est situé sur **[a, m]** ou sur **[m, b]** :

- si $f(a) \cdot f(m) < 0$, le zéro appartient à **[a, m]** ;
- sinon , le zéro appartient à **[m, b]**

En réitérant le procédé, la longueur de l'intervalle de recherche est chaque fois divisée en deux. On s'arrête lorsqu'on considère être arrivé à un intervalle suffisamment petit.

1.1. Principe de la méthode

Cette méthode consiste à construire trois suites a_n , b_n et c_n par récurrence selon le principe suivant :

- On pose $a_0 = a$ et $b_0 = b$.
- Puis on construit par récurrence une suite $\left(\left[a_n, b_n\right]\right)_n$ d'intervalles emboîtés en appliquant l'algorithme suivant : On partage l'intervalle $\left[a_n, b_n\right]$ en deux intervalles de même longueur à l'aide de $c_n = \frac{a_n + b_n}{2}$

- ► Si $f(a_n)$ et $f(c_n)$ sont de signe opposé, on pose $a_{n+1} = a_n$ et $b_{n+1} = c_n$.
- ► Sinon, on pose $b_{n+1} = b_n$ et $a_{n+1} = c_n$.

Remarques :

1. Les suites a_n et b_n sont **adjacentes** et convergent vers α qui est un zéro de la fonction ***f***
2. Dans la pratique on arrête le processus dès que $b_n - a_n = \frac{b-a}{2^n}$ est inférieur à la précision souhaitée.
3. Soit ***n*** le nombre d'étapes, pour obtenir une précision **ε**, il faudra que l'entier ***n*** vérifie l'inégalité $\frac{b-a}{2^n} \leq \varepsilon$, c'est-à-dire $n \geq \frac{\ln(b-a) - \ln(\varepsilon)}{\ln(2)}$

1.2. **Algorithme :**

Données : *f*, a, b, ε

Traitement

c ← a

d ← b

fc ← *f*(c)

fd ← *f*(d)

Tant que d - c > 2ε **faire**

m ← (c + d)/2

fm ← *f*(m)

Si fc × fm ≤ 0 **alors**

d ← m

fd ← fm

Sinon

c ← m

fc ←fm

Fin Si

Fin Tant que

Résultat : (c + d)/2

Remarque 1: Terminaison et validité de l'algorithme

- Il faut s'assurer que **la condition de sortie de boucle** est vérifiée en un **nombre fini d'étapes.** et que le résultat de sortie est bien le résultat voulu.
- Terminaison : Après n itérations de la boucle, on a $d-c=\frac{b-a}{2^n}$, comme $\lim_{n\to+\infty}\frac{b-a}{2^n}=0$, alors, on a, en un nombre fini d'étapes : $\frac{b-a}{2^n}\leq 2\varepsilon$ donc la boucle se termine (puisque ε > 0).
- L'invariant de boucle est : « ***A chaque itération on a*** $f(c).f(d)\leq 0$ » Cette propriété est toujours vraie (elle vraie avant l'entrée de la boucle. Ensuite on suppose qu'elle est vraie au début d'une itération et on démontre qu'elle reste vraie à la fin de celle-ci).
- Le réel **r** renvoyé est le milieu d'un intervalle de longueur majorée par 2ε (condition de sortie de boucle) et qui contient un zéro x_0 de f (invariant de boucle prouvé et le théorème des valeurs intermédiaires) on a alors bien $|x_0-r|\leq\varepsilon$

1.3. Mise en pratique de l'algorithme: Code Python

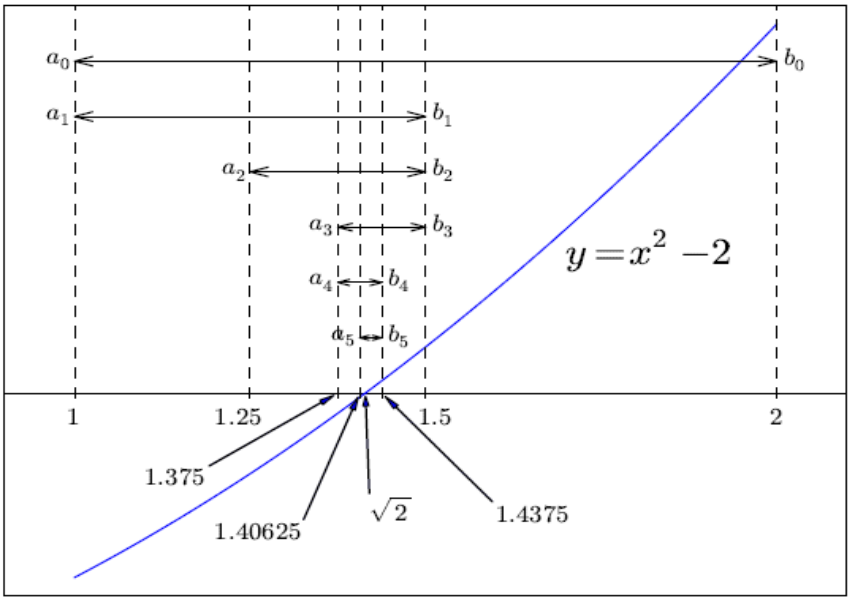
```
##Recherche approchée d'une solution d'une équation par dichotomie
def dichotomie(f, a, b, epsilon):
    assert f(a) * f(b) <= 0 and epsilon > 0
    c , d = a, b
    fc , fd = f(c) , f(d)
    while d - c > 2 * epsilon :
        m = (c + d) / 2.
        fm = f(m)
        if fc * fm <= 0 :
            d , fd = m , fm
        else :
            c , fc = m , fm
    return (c + d) / 2.
```

Exemple 1: Approximation de $\sqrt{2}$

Soit ***f*** une fonction définie par $f(x) = x^2 - 2$ continue sur l'intervalle **[1, 2]**, à valeurs réelles

- $f(1) = -1 < 0$ et $f(2) = 2 > 0$ donc l'équation **$f(x) = 0$** possède une solution dans **[1, 2]**

  La valeur médiane de cet intervalle est **1.5**.
- $f(1.5) = 0.25 > 0 > f(1)$, donc l'équation **$f(x) = 0$** possède une solution dans **[1, 1.5]**.
- $f(1.25) = -0.4675 < 0 < f(1.5)$, donc l'équation **$f(x) = 0$** possède une solution dans **[1.25, 1.5]**.
- **...**
- $f(1.41430664062) \approx 0.000263273715973 > 0 > f(1.4140625)$, donc l'équation **$f(x) = 0$** possède une solution dans **[1.4140625, 1.41430664062]**.




```
>>> dichotomie( lambda x : x ** 2 – 2 , 1 , 2 , 0.000001 )
1.4142141342163086
>>> math.sqrt(2)
1.4142135623730951
```

2. Méthode de Newton

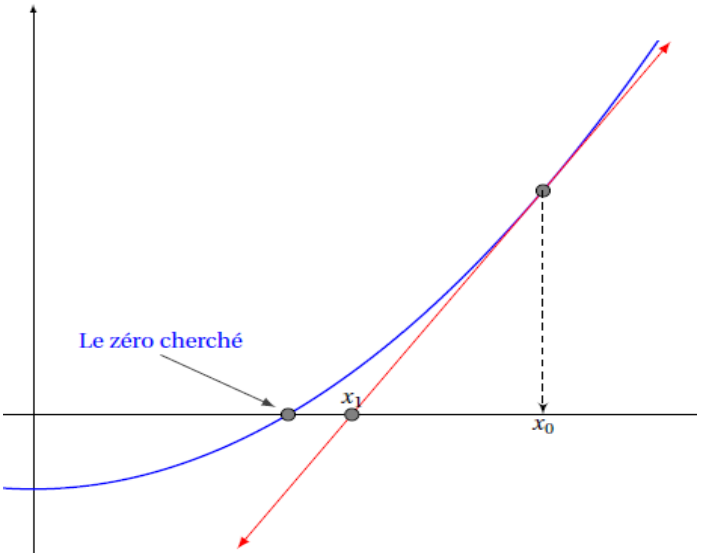
2.1. Principe de la méthode

Cette méthode consiste à approximer le graphe de la fonction au voisinage d'un point par la **tangente** en ce point. En un point d'abscisse x_0, la tangente a pour équation $y = f'(x_0).(x - x_0) + f(x_0)$, et coupe donc l'axe des abscisses en $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$ (a fonction f doit être de classe C^1, et on suppose ne pas tomber sur une dérivée nulle).

En itérant ce raisonnement, on peut alors construire une suite x_n définie par :

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

Elle est, sous certaines conditions qui ne seront pas étudiées ici, rapidement convergente, vers une solution de l'équation $f(x) = 0$.



2.2. Algorithme :

On part d'une première valeur et on suit la tangente ; on continue avec l'intersection de cette tangente avec l'axe des abscisses jusqu'à ce que la différence entre deux termes consécutifs soit assez petite. Il est nécessaire de ne jamais rencontrer de point en lequel la dérivée de f s'annule.

Données :

f : la fonction

fp : la dérivée de ***f***

x_0 : le point de départ

ε : la précision

nb_iter : compteur d'itération

max_iter : le nombre maximal d'itérations

Traitement

$x \leftarrow x_0$ # *On suppose que* x_0 *est convenablement choisi :* $fp(x_0) \neq 0$

$y \leftarrow x - f(x) / fp(x)$

$nb\_iter \leftarrow 1$

Tant que $|y - x| > \varepsilon$ **et** $nb\_iter \leq max\_iter$ **faire**

$x \leftarrow y$

$y \leftarrow y - f(y) / fp(y)$

$nb\_iter \leftarrow nb\_iter + 1$

Fin Tant que

Si $(nb\_iter > max\_iter)$

Afficher "On n'a pas encore la convergence vers la racine"

Sinon

Afficher y

Conditions d'arrêt :

- Si $|y - x| \leq \varepsilon$ alors y est le résultat de l'estimation de la racine
- Si $nb\_iter > max\_iter$ alors la méthode diverge ou n'a pas la convergence assez rapide pour fournir un résultat.
- <u>Attention</u> : Si $fp(y) = 0$, il y aura division par zéro.

2.3. Dérivation numérique :

La méthode de **Newton** nécessite la connaissance de la **dérivée** de la fonction en jeu. Pour calculer une valeur approchée de la dérivée d'une fonction numérique en un point (approximation des valeurs de ***f'*** au mieux), une première approche consiste à calculer un taux d'accroissement :

$$f(x_0 + h) = f(x_0) + h.f'(x_0) + O(h^2)$$

La dérivée peut être approximée par la valeur : $\dfrac{f(x_0 + h) - f(x_0)}{h}$

En théorie, il suffit de prendre *h* très petit pour obtenir une grande précision dans le calcul de la dérivée. Il ne faut toutefois pas perdre de vue la représentation des nombres réels en informatique.

Combiner $f(x_0+h)$ et $f(x_0-h)$:

$$f(x_0+h) = f(x_0)+h.f'(x_0)+\frac{h^2}{2}f''(x_0)+O(h^3)$$

$$f(x_0-h) = f(x_0)-h.f'(x_0)+\frac{h^2}{2}f''(x_0)+O(h^3)$$

permet d'obtenir une meilleure approximation de la dérivée par :

$$\frac{f(x_0+h)-f(x_0-h)}{2h}$$

Voici une implémentation possible du calcul d'une valeur approchée d'une dérivée qui utilise la deuxième formule d'approximation est proposée ci-après. En voici le code Python :

```
def derivee(f, x0, h):
    return ( f( x0 + h ) - f ( x0 - h ) ) / ( 2 * h )
```

2.4. Mise ne œuvre de la méthode de Newton :

Une fonction qui calcule une valeur approchée de la dérivée a donc été créée. Grâce à elle, il est possible d'écrire une fonction permettant la résolution numérique d'une équation de la forme ***f*(x)= 0** par la méthode Newton. Pour la version réalisée ici, on ne fait pas appel à cette fonction (***derivee***) . Voici l'implémentation de la méthode de Newton en Python (voir algorithme précédent):

```
def newton( f, fp , x0, epsilon , max_iter):
    x = x0
    y = x - f(x) / fp (x)
    nb_ iter = 1
    while nb_iter <= max_iter and abs ( x - y ) > epsilon :
        ( x , y ) = ( y , y - f( y ) / fp ( y ) )
```

```
        nb_iter += 1
    if nb_iter == max_iter + 1 :
        print("L'algorithme n'a pas converge")
    else :
        return y
```

Exemple 2 : Approximation de $\sqrt{2}$

```
>>> newton( lambda x : x ** 2 − 2 , lambda x : 2*x , 2 , 0.0001 , 10)
1.4142135623746899
>>> math.sqrt(2)
1.4142135623730951
```

3. Comparaison des méthodes

Pour pouvoir comparer la méthode dichotomique et la méthode de Newton, on effectuera uniquement **10 itérations** pour chercher une approximation de $\sqrt{2}$.

- Le milieu de l'intervalle obtenu par la méthode de **dichotomie** appliquée à $f(x) = x^2 - 2$ entre **0** et **100** est : **1.416015625**

- Le résultat de la méthode de **Newton** appliquée à $f(x) = x^2 - 2$ avec condition initiale **100** est : **1.4142135623730951**

- La valeur de $\sqrt{2}$ donnée par Python est : **1.4142135623730951**

La méthode de Newton offre une convergence nettement plus rapide que la méthode par dichotomie. Cependant, cette dernière offre une garantie de convergence que n'offre pas la méthode de Newton, qui ne converge que localement.

Si on connaît une bonne approximation d'un zéro et si on cherche la rapidité, alors on peut appliquer une méthode de la famille de Newton : la méthode de Newton proprement dite si on connaît f' ou si on décide de l'approcher numériquement, la méthode de la sécante si on ne peut/veut pas évaluer f'.

La convergence de cette méthode est trop compliquée pour qu'on la mette en oeuvre dans le cas général.

Si on cherche la simplicité et la robustesse, la méthode dichotomique s'impose : la continuité de la fonction et un premier intervalle [a, b] tel que $f(a).f(b) < 0$ sont suffisants.

4. Recherche des racines d'une fonction par le module scipy

Le module **scipy.optimize** regorge de procédures (cf **help(scipy.optimize)** pour le détail) permettant de trouver des minima, des **zéros** ou des points fixes. On va décrire ici deux algorithmes de recherche de racines:

- **sp.optimize.newton**(f,x0,fprime=None): Étant donné une fonction f et le voisinage x0 d'une de ses racines, cette méthode essaie de calculer la position du zéro, soit par la méthode de la **sécante** (cas où fprime n'est pas donné), soit par la méthode de Newton (cas où fprime, la dérivée de f, est donnée).
- **sp.optimize.bisect** (f,a,b): la méthode dichotomique qualifié de lente mais sûr Trouve une racine de f sur l'intervalle [a ; b] à condition que f(a).f(b) < 0 au départ
- **sp.optimize.brentq**(f,a,b): Trouve une racine de f sur l'intervalle [a ; b] à condition que f(a).f(b) < 0 au départ (Utilise l'algorithme de Brent (1973)).

Exemples :

```
>>> import scipy as sp
>>> import scipy.optimize
>>> f = lambda x: x**2 - 2
>>> fp= lambda x: 2*x
>>> sp.optimize.newton(f,1)                # Méthode des sécantes en partant de 1
1.4142135623730947
>>> sp.optimize.newton(f,1,fp)             # Méthode de Newton en partant de 1
1.4142135623730951
>>> sp.optimize.brentq(f,0,2)              # Méthode de Brent entre 0 et 2
1.4142135623731364
>>> sp.optimize.bisect(f,0,2)
1.4142135623724243
 >>> sp.sqrt(2)                           # La bonne réponse
```

```
1.4142135623730951
>>> sp.optimize.newton(f,100)          # Méthode des sécantes en partant de
100
1.4142135623730951
>>> sp.optimize.newton(f,100,fp)       # Méthode de Newton en partant de 100
1.4142135623730951
```