

Chapitre 8 : Résolution numérique d'équations algébriques

I. Introduction

- De nombreux problèmes en physique, en SI, en mathématiques passent par la résolution d'équations algébriques. Nous allons voir dans ce chapitre comment évaluer numériquement une solution d'une équation de la forme $f(x) = 0$ avec f une application de \mathbb{R} dans \mathbb{R} , dans des cadres où la théorie nous assure qu'une telle solution existe, mais sans en fournir d'expression analytique.
- Les méthodes qui seront exposées sont des méthodes itératives ; le principe est de partir d'une **estimation grossière** de la solution puis d'en améliorer la précision par une application itérée d'un algorithme.
- L'existence de solutions à l'équation $f(x) = 0$ repose sur le théorème des valeurs intermédiaires.

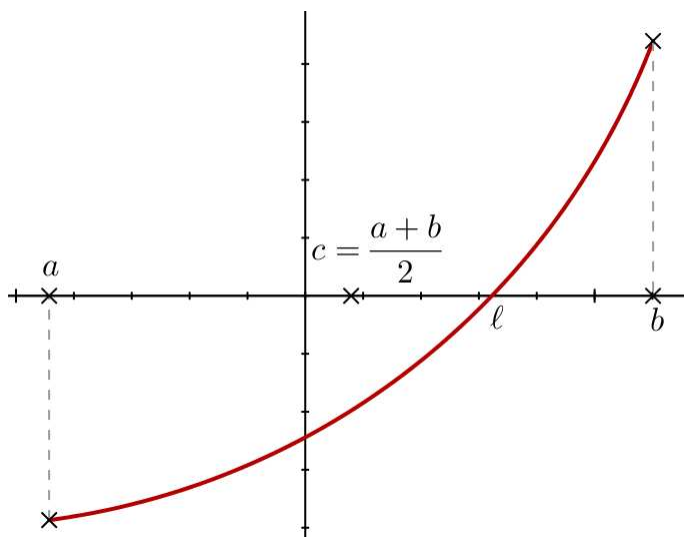
Théorème des valeurs intermédiaires

Soit f une fonction continue sur $[a; b]$ telle que $f(a)f(b) < 0$ alors il existe au moins un réel $\ell \in]a; b[$ qui vérifie $f(\ell) = 0$.
Si de plus f est strictement monotone sur $[a; b]$ alors ℓ est unique.

II. Méthode dichotomique

1. Principe théorique

La méthode consiste à approcher ℓ par le milieu du segment $[a; b]$ c'est-à-dire $c = \frac{a+b}{2}$.



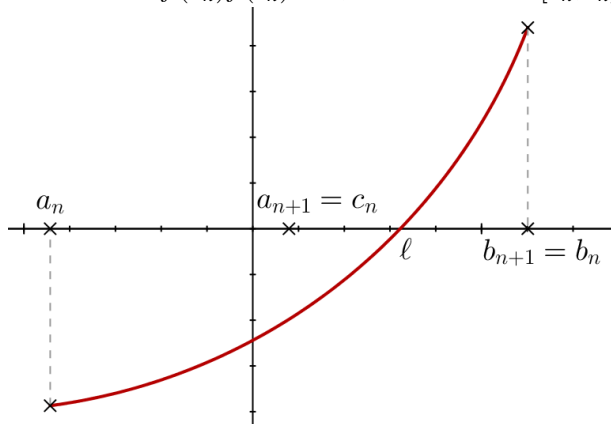
On a donc l'encadrement suivant : $|c - \ell| \leq b - a$: l'erreur commise est donc de l'ordre de $b - a$.

2. Construction de l'algorithme

On itère le procédé précédent en construisant les suites (a_n) , (b_n) et (c_n) définies par :

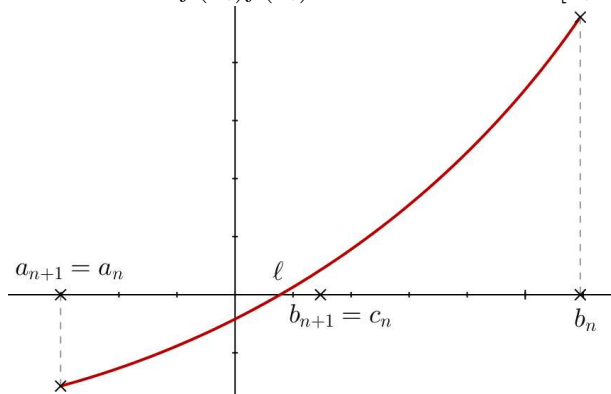
- $a_0 = a, b_0 = b$
- Pour tout $n \in \mathbb{N}$, $c_n = \frac{a_n + b_n}{2}$

- 1^{er} cas : Si $f(a_n)f(c_n) < 0$, c'est-à-dire $\ell \in [a_n; c_n]$ alors



$$\text{on pose } \begin{cases} a_{n+1} = a_n \\ b_{n+1} = c_n \end{cases}$$

- 2^{ème} cas : Si $f(a_n)f(c_n) > 0$, c'est-à-dire $\ell \in [c_n; b_n]$ alors



$$\text{on pose } \begin{cases} a_{n+1} = c_n \\ b_{n+1} = b_n \end{cases}$$

Théorème

Les suites (a_n) et (b_n) sont adjacentes et convergent vers ℓ .

Conséquences

On a les inégalités suivantes

$$|a_n - \ell| \leq b_n - a_n = \frac{b-a}{2^n}$$

$$|b_n - \ell| \leq b_n - a_n = \frac{b-a}{2^n}$$

En approximant ℓ par a_n ou b_n , on fait une erreur de l'ordre de $\frac{b-a}{2^n}$.

Remarque 1

Pour tout $n \in \mathbb{N}, c_n = \frac{a_n + b_n}{2}$ donc (c_n) converge vers $\frac{\ell + \ell}{2} = \ell$.

Remarque 2

Il ne faut pas confondre valeurs approchées par excès et par défaut à 10^{-p} près et valeur approchée avec p chiffres exacts après la virgule.

Pour obtenir une valeur approchée par excès ou par défaut à 10^{-p} près, il suffit que $\frac{b-a}{2^n} \leq 10^{-p}$: Les valeurs approchées différeront entre elles seulement d'une unité à partir de la p -ième décimale.

Pour obtenir p chiffres exacts après la virgule, il suffit que $\frac{b-a}{2^n} \leq 10^{-(p+1)}$

$$\iff 2^n \geq (b-a)10^{p+1} \iff n \geq \log_2((b-a)10^{p+1}).$$

On a donc p chiffres exacts après la virgule à partir de

$$n = \lfloor \log_2((b-a)10^{p+1}) \rfloor + 1.$$

3. Codage en python

```
def dichotomie(f, a, b, p):
    assert f(a) * f(b) <= 0
    c, d = a, b
    while abs(d - c) > 10**(-p):
        m = (c + d) / 2.
        if f(c) * f(m) <= 0:
            d = m
        else:
            c = m
    return (c + d) / 2
```

4. Complexité de l'algorithme

- Dans le cas de la méthode par dichotomie on peut obtenir explicitement le rang à partir duquel on atteint la précision souhaitée ε , il suffit de résoudre l'inéquation portant sur n :

$$\frac{b-a}{2^n} \leq \varepsilon \iff 2^n \geq \frac{b-a}{\varepsilon} \iff n \geq \log_2 \left(\frac{b-a}{\varepsilon} \right).$$

La complexité de cet algorithme est de type logarithmique en

$$O \left(\ln \left(\frac{b-a}{\varepsilon} \right) \right)$$

- En binaire, pour obtenir une valeur approchée du résultat avec p bits significatifs, on peut modifier la condition d'arrêt en :

```
while d - c > 2**(-p):
```

Le nombre d'itérations est alors de l'ordre de p .

5. Exemples

- Résolution de l'équation $x^2 - 2 = 0$ sur \mathbb{R}^+ .

```
def f(x) :
    return x**2-2

In [1]: dichotomie(f, 1, 2, 10)
Out[1]: (1.4142135623730951, 1.4142141342163086)
```

Ex1 Évaluer le nombre d'itérations ayant été réalisées au cours de l'appel de la fonction.

- Résolution de $\sin(x) = 0$ sur $[3, 4]$.

On peut utiliser la fonction `sin` de la bibliothèque `math` qu'on aura préalablement importée.

```
In [2]: math.pi, dichotomie(math.sin, 3, 4, 10)
Out[2]: (3.141592653589793, 3.141592653642874)
```

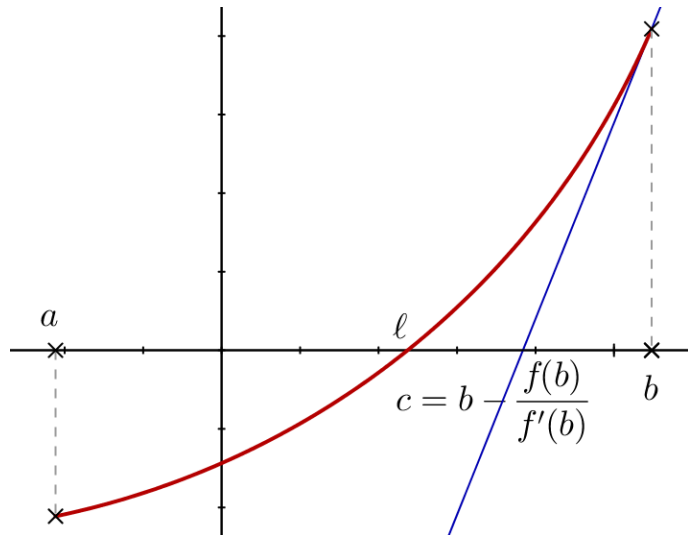
Ex2 Évaluer le nombre d'itérations ayant été réalisées au cours de l'appel de la fonction.

III. Méthode de Newton

1. Principe théorique

Soit $f : [a;b] \rightarrow \mathbb{R}$ de classe \mathcal{C}^2 , f' ne s'annule pas sur $[a;b]$ et il existe $\ell \in [a;b]$ tel que $f(\ell) = 0$.

La méthode de Newton est aussi appelée **méthode de la tangente**. Elle consiste à approcher ℓ par l'abscisse c du point d'intersection de la tangente à \mathcal{C}_f en $B(b, f(b))$ avec l'axe des abscisses.



La tangente a pour équation : $y - f(b) = f'(b)(x - b)$ d'où

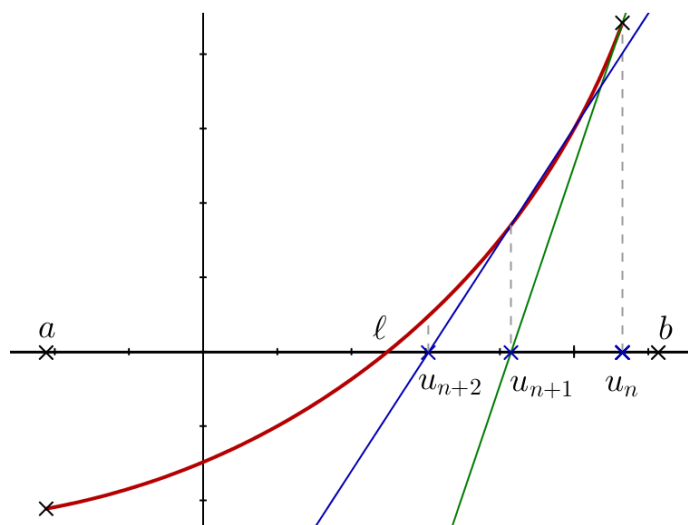
$$c = b - \frac{f(b)}{f'(b)}$$

2. Algorithme général

On itère le procédé précédent en construisant la suite (u_n) définie par :

On se donne $u_0 \in [a, b]$, on construit ensuite le point d'intersection de la tangente à \mathcal{C}_f en $(u_0, f(u_0))$ avec l'axe des abscisses dont on notera l'abscisse u_1 .

Supposons connus $u_0 \dots u_n$, on construit ensuite le point d'intersection de la tangente à \mathcal{C}_f en $(u_n, f(u_n))$ avec l'axe des abscisses dont on notera l'abscisse u_{n+1} .



Théorème

La suite (u_n) est alors définie par la relation de récurrence :

$$\forall n \in \mathbb{N}, u_{n+1} = u_n - \frac{f(u_n)}{f'(u_n)}$$

De plus, si $f(u_0)f''(u_0) > 0$ alors (u_n) converge vers ℓ .

3. Codage en python

```
def newton(f, fp, x0, p):
    u = u0
    v = u - f(u)/fp(u)
    while abs(v-u) > 10**(-p): # Le test se limite à estimer la différence entre deux termes consécutifs (faute de mieux)
        u, v = v, v - f(v)/fp(v)
    return v
```

4. Complexité de l'algorithme

On va utiliser un résultat permettant d'estimer la vitesse de convergence de la suite et la complexité de l'algorithme.

Dans la méthode, on est amené à itérer la fonction $g(x) = x - \frac{f(x)}{f'(x)}$.

Avec les hypothèses de travail citées en introduction, la fonction g est de classe \mathcal{C}^1 au voisinage de ℓ . La solution ℓ de $f(x) = 0$ est un point superattractif de g .

Théorème

On suppose que f est de classe \mathcal{C}^2 au voisinage de ℓ et que $f' \neq 0$ sur I . Alors il existe $M > 0$ tel que pour tout x au voisinage de ℓ ,

$$|g(x) - \ell| \leq M|x - \ell|^2$$

Conséquence (Majoration de l'erreur)

Sous les conditions du théorème précédent et pour tout point initial u_0 au voisinage de ℓ , on a

$$\forall n \in \mathbb{N}, |u_n - \ell| \leq \frac{1}{M}(M|u_0 - \ell|)^{2^n}$$

Remarque 1

Si on choisit $|u_0 - \ell| \leq \frac{1}{10M}$, alors on obtient $|u_n - \ell| \leq \frac{1}{M} \frac{1}{10^{2^n}}$. On a donc une **convergence quadratique** de la suite vers ℓ .

Remarque 2

Le nombre de décimales exactes double environ à chaque itération ; 10 itérations suffiraient ainsi théoriquement pour obtenir plus de 1000 décimales exactes !

Complexité

Pour ce qui est de la complexité, on constate qu'elle s'exprime comme un $O(\ln(\ln(\frac{1}{\epsilon})))$. Pour une précision fixée, l'algorithme de Newton converge plus vite vers ℓ que l'algorithme de dichotomie.

5. Exemple : algorithme de Babylone

- Revenons sur la résolution de l'équation $x^2 - 2 = 0$ sur \mathbb{R}^+ .

Proposition

La suite récurrente (u_n) est alors définie par :

$$u_0 > 0 \quad \text{et} \quad \forall n \in \mathbb{N}, u_{n+1} = \frac{1}{2} \left(u_n + \frac{2}{u_n} \right)$$

Alors (u_n) converge vers $\sqrt{2}$.

Cette suite va permettre de construire l'algorithme dit de « Babylone ».

- Appel de la procédure :

```
In [4]: newton(f, fp, 1.5, 10)
Out[4]: 1.4142156862745099
```

Ex 3 Évaluer le nombre d'itérations ayant été réalisées au cours de l'appel de la fonction. Conclure.

IV. Comparaison des algorithmes**1. Méthode dichotomique**

- La méthode de dichotomie permet d'approcher une solution avec p bits significatifs en p étapes, ce qui est déjà très efficace.
- Ses conditions d'utilisation sont assez basiques (on demande seulement à la fonction d'être continue et de changer de signe), ce qui en fait une méthode « robuste ».

2. Méthode de Newton

- La méthode de Newton a une vitesse de convergence assez diabolique : à chaque étape supplémentaire, le nombre de décimales correctes est multiplié par deux ! On atteint par exemple une précision de 50 bits en moins de dix étapes.
- Le prix à payer est un ensemble de conditions d'application parfois un peu délicates à vérifier. Sans avoir fait l'étude de la fonction f , le problème de démontrer la **validité** et la **terminaison** de l'algorithme, c'est-à-dire ici d'une part le fait qu'il n'y aura pas de division par zéro, et d'autre part que le résultat renvoyé v à la dernière itération sera tel qu'il existe bien un zéro de f , noté ℓ , tel que $|v - \ell| \leq 10^{-p}$.

Mais dans la pratique la méthode de Newton est un cauchemar pour l'informaticien : On peut rencontrer des divisions par zéro. La terminaison n'est pas assurée. Même si un résultat est renvoyé, il peut être éloigné d'un zéro de f .

En pratique, les programmes réalisant la méthode de Newton sont donc un peu plus compliqués.

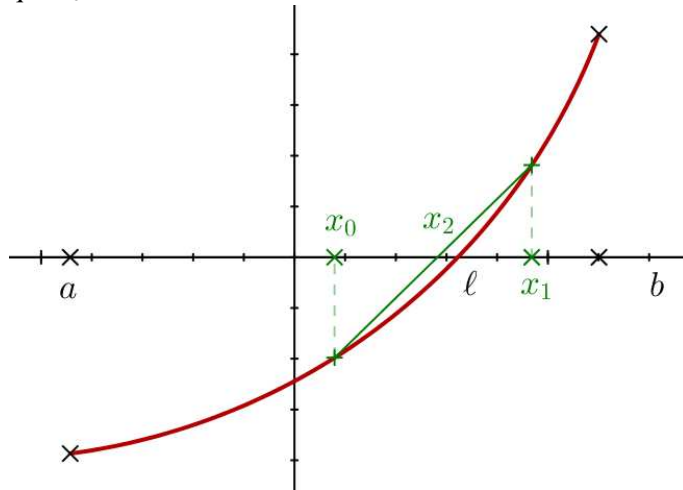
V. Méthode de la sécante

1. Principe théorique

- Dans certaines situations, la dérivée f' est très compliquée ou même impossible à expliciter (C'est le cas par exemple si la fonction f est le résultat d'un algorithme complexe). On ne peut alors utiliser la méthode de Newton.

L'idée est alors de remplacer f' par le taux d'accroissement de f sur un petit intervalle.

- Supposons que l'on dispose de deux valeurs approchées x_0 et x_1 de la solution ℓ recherchée. Supposons que $x_0 < \ell < x_1$.



- Le taux d'accroissement de f sur l'intervalle $[x_0; x_1]$ est

$$\tau_1 = \frac{f(x_1) - f(x_0)}{x_1 - x_0}$$

et l'équation de la sécante coupant la courbe représentative de f aux points d'abscisses x_0 et x_1 est

$$y = \tau_1(x - x_1) + f(x_1)$$

On obtient alors une nouvelle approximation x_2 de ℓ en calculant l'abscisse de l'intersection de la sécante avec l'axe Ox

$$x_2 = x_1 - \frac{f(x_1)}{\tau_1}$$

2. Construction de l'algorithme

- On itère le procédé précédent en construisant la suite (u_n) définie par :
On se donne u_0 et u_1 tels que $u_0 < \ell < u_1$. On construit ensuite le point d'intersection de la droite coupant le graphe de la fonction f aux points d'abscisse u_0 et u_1 avec l'axe des abscisses. Ce point a pour abscisse u_2 .
Supposons connus $u_0 \dots u_n$, on construit ensuite le point d'intersection de la droite coupant le graphe de la fonction f aux points d'abscisse u_{n-1} et u_n . Ce point a pour abscisse u_{n+1} .
- La suite (u_n) est alors définie par la relation de récurrence :

$$\forall n \in \mathbb{N}, u_{n+1} = u_n - \frac{f(u_n)}{\tau_n} \quad \text{avec} \quad \tau_n = \frac{f(u_n) - f(u_{n-1})}{u_n - u_{n-1}}$$

3. Codage en python

```
def secante(f, u0, u1, p):
    u,v = u0,u1
    t = (f(v)-f(u))/(v-u)
    while abs(v-u) > 10**(-p):
        u,v = v, v - f(v)/t
        t = (f(v)-f(u))/(v-u)
    return v
```

4. Complexité

- Nous allons citer un théorème assurant la convergence de la suite et permettant également d'obtenir une majoration de l'erreur.

Théorème

On suppose f de classe \mathcal{C}^2 telle que $f' \neq 0$ sur un voisinage V de ℓ .

On pose pour $i \in \{1, 2\}$, $M_i = \max_V |f^{(i)}|$, $m_i = \min_V |f^{(i)}|$.

Alors quel que soit le choix des points initiaux u_0 et u_1 éléments distincts de V , il existe un réel K tel que

$$\forall n \in \mathbb{N}, |u_n - \ell| \leq \frac{1}{K} (K \max(|u_0 - \ell|, |u_1 - \ell|))^{F_n}$$

avec F_n le n -ème nombre de Fibonacci.

Sachant que $F_n \sim_{n \rightarrow +\infty} \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2}\right)^{n+1}$ et $\frac{1+\sqrt{5}}{2} \simeq 1,618$, on a $F_n < 2^{n+1}$: la convergence de la suite est un peu moins rapide que dans le cas de l'algorithme de Newton.

- Cet algorithme a aussi une complexité en $O(\ln(\ln(\frac{1}{\epsilon})))$.

5. Exemples

- Résolution de l'équation $x^2 - 2 = 0$ sur \mathbb{R}^+ .

```
In [1]: math.sqrt(2), secante(f, 1, 2, 10)
Out[1]: (1.4142135623730951, 1.4142141342163086)
```

- Résolution de $\sin(x) = 0$ sur $[3, 4]$.

```
In [2]: math.pi, secante(math.sin, 3, 4, 10)
Out[2]: (3.141592653589793, 3.141592653642874)
```

6. Avantages et inconvénients de cette méthode

- Cette méthode permet donc d'obtenir une solution approchée de l'équation $f(x) = 0$ sans connaître l'expression analytique de f' , et ce avec une rapidité de convergence comparable à celle de la méthode de Newton.
- Par contre, les nombreuses conditions sur la fonction f et ses dérivées, restreignent son champ d'application. En outre, l'algorithme itératif ne peut démarrer que si on dispose déjà de deux valeurs approchées x_0 et x_1 de ℓ .