Chapitre 9 : Analyse numérique

L'analyse numérique est la discipline qui traite des méthodes permettant de donner des solutions approchées à des problèmes mathématiques.
Dans ce chapitre, nous détaillons le principe des algorithmes d'analyse numérique qui sont au programme ; en tant que tels, vous êtes susceptible de les rencontrer au concours, et devrez donc être à même de les implémenter (après que l'on vous ait rappelé leur principe).
D'autre part, on cherchera à travers l'étude de ces algorithmes à souligner l'importance de leur vitesse de convergence, ainsi que la nécessité du contrôle de la marge d'erreur associée.

1 Résolution de l'équation $f(x) = 0$

Dans toute cette partie, on considère deux réels $a < b$, et f une fonction de $\mathbb{R}$ dans $\mathbb{R}$ continue et strictement croissante sur $[a, b]$. On sait, par le théorème de la bijection, qu'on a le résultat suivant :

Proposition 1.1. Si $f(a) \leq 0$ et $f(b) \geq 0$, il existe un unique réel $x_0 \in [a, b]$ tel que

$$f(x_0) = 0.$$

Dans cette partie, on se posera le problème suivant si $\varepsilon > 0$, comment trouver une valeur approchée de x_0 à e près ? On va donner deux méthodes qui vont permettre d'obtenir une valeur $x \in [x_0 - \varepsilon, x_0 + \varepsilon]$.

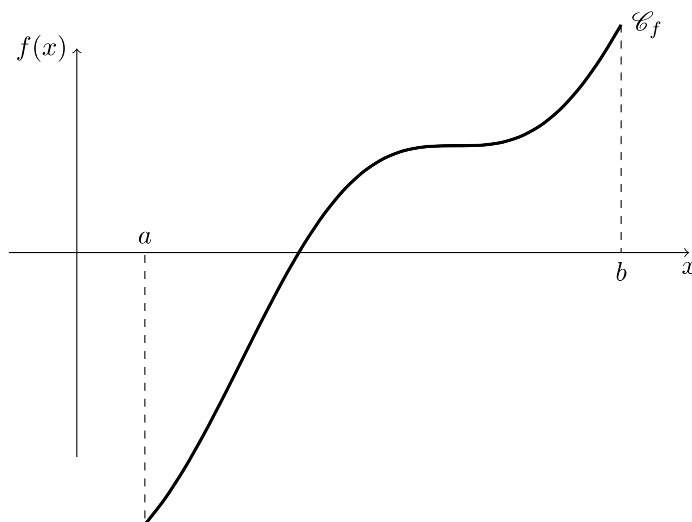
1 A) Méthode de dichotomie

1 A) a) Algorithmique

Cette méthode est fondée sur la démonstration faite en cours du théorème des valeurs intermédiaires qui faisait apparaître le résultat suivant :

Théorème 1.1. Il existe deux suites adjacentes (a_n) et (b_n) vérifiant pour tout $n \in \mathbb{N}$

$$a_n \leq x_0 \leq b_n \quad \text{et} \quad b_n - a_n = \frac{b-a}{2^n}.$$



Modes de définition des suites (a_n) et (b_n) :

Remarque 1.2.

1 A) b) Code Python

On va donner le code Python en deux étapes :

1. une fonction `suite_dicho(f,a,b,n)` qui renvoie une liste à deux élements : `[`a_n`,`b_n`]` ;
2. une fonction `dichotomie(f,a,b,e)` qui renvoie une liste à deux éléments `[`a_N, b_N`]` tel que $x_0 \in [a_N, b_N]$ et $b_N - a_N \leq e$.

Code Python des fonctions :

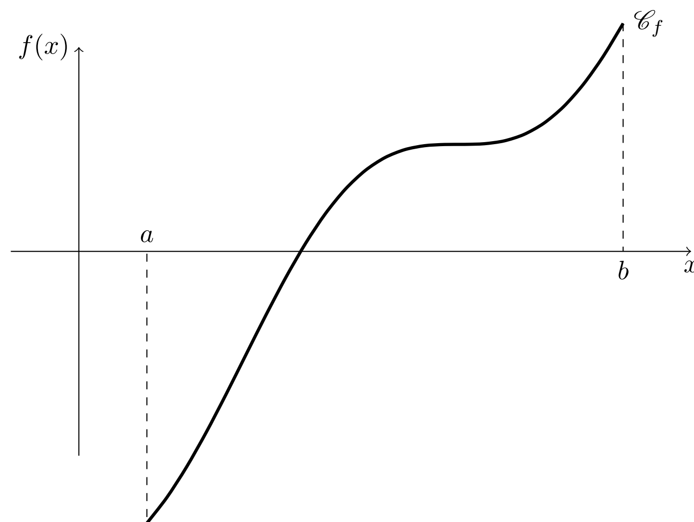
Remarque 1.3.

Exemple 1.4. Écrire une fonction `dichotomie(f,a,b,e)` qui renvoie une liste à deux éléments `[`a_N, b_N`]` tel que $x_0 \in [a_N, b_N]$ et $b_N - a_N \leq e$ mais où dans ce cas, f est seulement supposée strictement monotone.

1 B) Méthode de Newton

1 B) a) Algorithmique

L'idée de la méthode est de supposer que f est « régulière » et « proche » de sa tangente puis d'itérer cette méthode.



Principe de la méthode :

Remarque 1.5.

1 B) b) Convergence de la méthode

Dans toute la suite, on fait des hypothèses plus fortes sur f pour assurer la convergence de l'algorithme : on suppose que f est une fonction de classe $\mathscr{C}^2$ sur $[a, b]$ s'annulant en x_0 telle que f' ne s'annule pas sur $[a, b]$. Pour simplifier, f est strictement croissante. On donne le théorème suivant, qu'on démontrera dans le chapitre B5 :

Théorème 1.6 (Convergence de la méthode Newton)**.** Il existe une constante $M \in \mathbb{R}^{+,*}$ et un réel $h \in \mathbb{R}^{+,*}$ tel que pour tout $a_0 \in [x_0 - h, x_0 + h]$:

$$\forall n \in \mathbb{N}, \quad |a_n - x_0| \leq \frac{1}{M} |M(a_0 - x_0)|^{2^n}.$$

Remarque 1.7.

1 B) c) Code Python

On rappelle que g est la dérivée de f. On va donner le code Python en deux étapes :

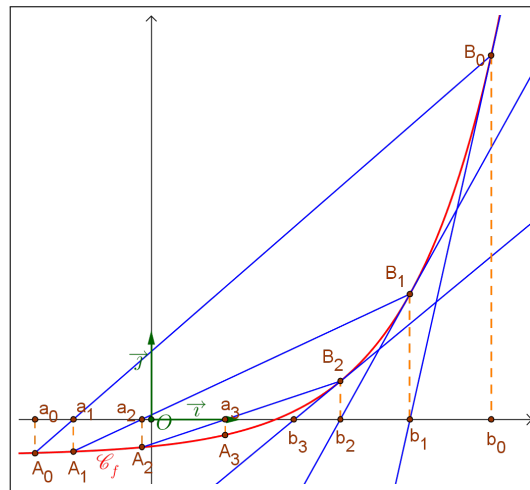
1. une fonction `suite_newton(f,g,a0,n)` qui renvoie a_n ;
2. une fonction `newton(f,g,a0,e)` qui renvoie a_n dès que $|f(a_n)| < e$.

Code Python des fonctions :

Remarque 1.8.

1 B) d) Encadrement cordes-tangentes

Une alternative simple aux conditions d'arrêts naïves (comme celle vue précédement sur le fait que $f(a_n) < \varepsilon$), est la méthode par encadrement corde-tangente de la solution c recherchée. Le principe de calcul de cet encadrement est le suivant. On part de l'encadrement initial : $a = a_0 < c < b_0 = b$. On calcule ensuite b_1 à partir de b_0 comme précédemment, et on calcule a_1 comme étant l'abscisse du point d'intersection de l'axe des abscisses et de la corde joignant les points de la courbe d'abscisses a_0 et b_0.
En itérant ce processus, on construit deux suites $(a_n)_{n\in\mathbb{N}}$ et $(b_n)_{n\in\mathbb{N}}$ vérifiant $\forall n \in \mathbb{N},\ a_n \leqslant c \leqslant b_n$ (sous l'hypothèse que f' et f'' sont strictement positives sur l'intervalle considéré). Il suffit donc de s'arrêter lorsque l'on a $b_n - a_n < \varepsilon$: a_n et b_n sont alors une valeur approchée de c à ε près.



Une équation de la corde qui joint les points $A_n(a_n, f(a_n))$ et $B_n(b_n, f(b_n))$ est :

$$y = \frac{f(b_n) - f(a_n)}{b_n - a_n}(x - a_n) + f(a_n).$$

Il suffit alors de prendre $y = 0$ dans cette équation pour déterminer a_{n+1}. On obtient alors la relation :

$$\forall n \in \mathbb{N},\ a_{n+1} = \frac{a_n f(b_n) - b_n f(a_n)}{f(b_n) - f(a_n)}.$$

Le code suivant implémente la méthode de Newton en utilisant l'encadrement corde-tangente. On applique alors l'algorithme à la fonction définie par $f(x) = x^2 - 2$ sur $[1, 3]$, ce qui donne finalement une valeur approchée de $\sqrt{2}$. On remarque sur cet exemple que l'algorithme converge en quatre étapes seulement, et donne une valeur approchée à 10^{-6} près de $\sqrt{2}$ (on observe par ailleurs que l'approximation obtenue avec la méthode naïve était correcte dans ce cas particulier !).

```
def Newton(f,fprime,a,b,epsilon):
    u=a
    v=b
    n=0
    while v-u>epsilon:
        u=(u*f(v)-v*f(u))/(f(v)-f(u))
        v=v-f(v)/fprime(v)
        n=n+1
    return(u,n)

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

```
def fprime(x):
    return(2*x)

>>> Newton(f,fprime,1,3,10**(-6))
(1.4142134584136423, 4)

>>> import numpy as np
>>> np.sqrt(2)
1.4142135623730951
```

Remarque 1.9.
Dans ce qui précède, nous avons supposé f' et f'' strictement positives. En fait, quitte à changer la fonction considérée, il suffit que f' et f'' ne s'annulent pas sur l'intervalle considéré pour pouvoir appliquer la méthode de Newton.

1 C) Conclusion

On a étudié dans cette partie deux méthodes d'approximation de solution de l'équation $f(x) = 0$:

1. la méthode dichotomie, la moins performante, mais ne nécessitant que peu de conditions sur f ;
2. la méthode de Newton, la plus performante, mais nécessitant de la régularité sur f ainsi qu'on choix judicieux de a_0.

En pratique, on utilise la méthode de dichotomie pour approcher grossièrement x_0 puis on affine avec la méthode de Newton.

2 Calcul approché d'une intégrale

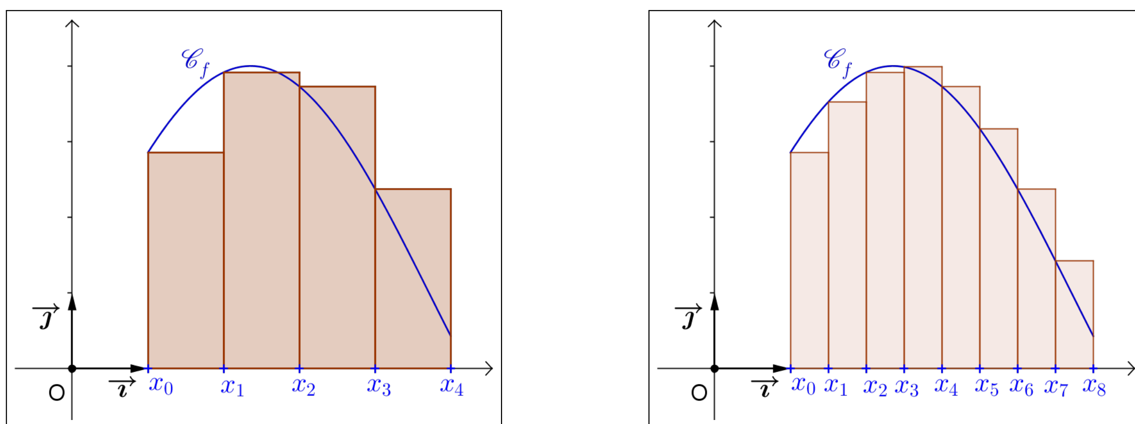
2 A) La méthode des rectangles

2 A) a) Algorithmique

Soit f une fonction continue définie sur un intervalle $[a, b]$ avec $a < b$. Pour calculer une valeur approchée de l'intégrale $\int_a^b f(t)\,\mathrm{d}t$, on peut procéder comme suit.

Soit $n \in \mathbb{N}^*$. Posons, pour tout $k \in [\![0, n-1]\!]$, $x_k = a + k\dfrac{b-a}{n}$. On a alors $a = x_0 < x_1 < \cdots < x_n = b$: on a subdivisé l'intervalle $[a, b]$ en n intervalles de longueurs égales $[x_0, x_1]$, $[x_1, x_2]$, . . . , $[x_{n-1}, x_n]$.

Géométriquement, l'intégrale $\int_a^b f(t)\,\mathrm{d}t$ correspond à l'aire algébrique délimitée par la courbe représentative de f, l'axe des abscisses et les droites verticales d'équations $x = a$ et $x = b$. On peut approximer cette aire en sommant, l'aire des rectangles dont la base est l'intervalle $[x_k, x_{k+1}]$ et la hauteur est $f(x_k)$, comme indiqué sur les dessins ci dessous (où on a pris $n = 4$ puis $n = 8$).



Il est clair que la somme des aires algébriques des n rectangles est donnée par :

$$\mathcal{A}_R(n) = \sum_{k=0}^{n-1}(x_{k+1} - x_k)f(x_k) = \frac{b-a}{n}\sum_{k=0}^{n-1} f(x_k).$$

De manière précise, on a le théorème suivant, qu'on démontrra dans le chapitre « Intégration » :

Théorème 2.1. On suppose que f est de classe $\mathscr{C}^1$ sur $[a, b]$. On a l'inégalité :

$$\forall n \in \mathbb{N}^*, \quad |I(f) - I_n(f)| \leq \frac{(b-a)^2}{n} \sup_{x\in[a,b]} |f'(x)|.$$

Ainsi, $I_n(f) \longrightarrow I(f)$.

Remarque 2.2.

2 B) Code Python

On donne la fonction Python donnant une approximation d'une intégrale de f sur $[a,b]$ avec $n+1$ point pour la méthode des rectangles sous le nom `rectangles(f,a,b,n)`

Code Python de la fonction

Remarque 2.3.

Le code ci-dessous est celui d'une fonction qui prend en entrée f, a, b, ε et M, puis renvoie une valeur approchée de $\displaystyle\int_a^b f(t)\,\mathrm{d}t$, ainsi que l'entier n utilisé pour ce faire. On peut ensuite l'appliquer à l'exemple de l'intégrale $\displaystyle\int_0^1 \frac{4\,\mathrm{d}x}{1+x^2} = \pi$. En effet, on a $\forall x \in [0,1]\,,\ f'(x) = \dfrac{-8x}{(1+x^2)^2}$, d'où $\forall x \in [0,1]\,,\ |f'(x)| \leqslant 8$ ce qui prouve $M = 8$ convient.

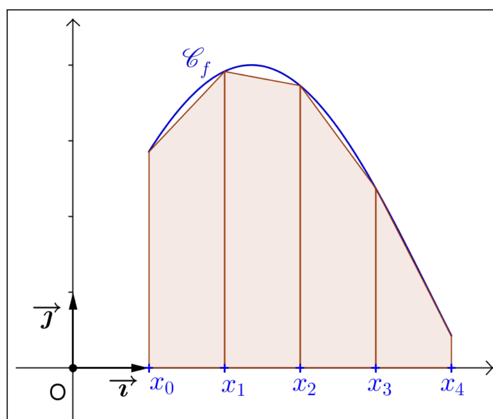
```
def rectangle2(f,a,b,epsilon,M):
    n=ma.floor(M*(b-a)**2/(2*epsilon))+1
    A=rectangle(f,a,b,n)
    return(A,n)
```

```
>>> rectangle2(g,0,1,10**(-2),8)
(3.144085382698032, 401)
>>> rectangle2(g,0,1,10**(-4),8)
(3.141617652860626, 40001)
```

Remarque 2.4.
L'emploi de cette méthode présuppose que l'on connaisse le réel M. Pour cela il faut notamment être capable de calculer la dérivée de la fonction f, ce qui est aisé lorsque l'expression $f(x)$ est donnée par une formule explicite, mais est plus délicat si la fonction f n'est connue qu'à travers un tableau donnant des valeurs $f(t_1), \ldots, f(t_n)$. Dans ce cas, on peut toujours donner des valeurs approchées de $f'(t_1), \ldots, f'(t_n)$ en prenant $f'(t_i) \simeq \dfrac{f(t_{i+1}) - f(t_i)}{t_{i+1} - t_i}$ (ce qui ne sera pertinent que si les écarts $|t_{i+1} - t_i|$ sont petits).

2 C) La méthode des trapèzes

Au lieu d'utiliser des rectangles, on peut construire des trapèzes, comme sur le dessin suivant.



Dans ce cas, si $\mathcal{A}_T(n)$ désigne la somme des aires algébriques des trapèzes, on a :

$$\mathcal{A}_T(n) = \sum_{k=0}^{n-1}(x_{k+1} - x_k) \times \frac{1}{2}\left(f(x_k) + f(x_{k+1})\right) = \frac{b-a}{2n}\sum_{k=0}^{n-1}\left(f(x_k) + f(x_{k+1})\right).$$

Si f est deux fois dérivable sur $]a, b[$ et si M_2 est une constante telle que $\forall x \in]a, b[, \ |f''(x)| \leqslant M_2$, alors on peut démontrer :

$$\left|\int_a^b f(t)\,\mathrm{d}t - \mathcal{A}_T(n)\right| \leqslant M_2\frac{(b-a)^3}{12n^2}.$$

Ainsi, si on souhaite donner une valeur approchée de $\displaystyle\int_a^b f(t)\,\mathrm{d}t$ à ε près, il suffit de déterminer un entier n tel que $M_2\dfrac{(b-a)^3}{12n^2} \leqslant \varepsilon$, c'est-à-dire $n \geqslant \left\lfloor\sqrt{\dfrac{M_2(b-a)^3}{12\varepsilon}}\right\rfloor + 1$, puis de calculer $\mathcal{A}_T(n)$.

On peut appliquer cette méthode à $f(x) = \dfrac{4}{1+x^2}$ sur $[0, 1]$, car on peut établir que $M_2 = 8$ convient.

```
def trapeze(f,a,b,n):
    A,x=0,a
    for k in range(n):
        A=A+f(x)+f(x+(b-a)/n)
        x=x+(b-a)/n
    return(((b-a)/(2*n))*A)

def trapeze2(f,a,b,epsilon,M2):
    n=ma.floor(sqrt(M2*(b-a)**3/(12*epsilon)))+1
    A=trapeze(f,a,b,n)
    return(A,n)
```

```
>>> trapeze2(g,0,1,10**(-2),8)
(3.139535044154281, 9)
>>> trapeze2(g,0,1,10**(-4),8)
(3.1415678667565685, 82)
```

Comme $\dfrac{1}{n^2}$ tend « plus vite » vers 0 que $\dfrac{1}{n}$, c'est-à-dire $\dfrac{1}{n^2} = o\left(\dfrac{1}{n}\right)$, il est clair que la méthode des trapèzes converge plus vite que la méthode des rectangles. On vient notamment de le vérifier dans le cadre de l'exemple précédent : il faut prendre $n = 40001$ pour avoir une valeur approchée de π à 10^{-4} près avec la méthode des rectangles, contre $n = 82$ avec la méthode des trapèzes.