Chapitre IV

Révisions 4 : Preuve et complexité - quelques algorithmes classiques

Sommaire

1 Preuve d'un algorithme

1.1 Définition

Définition - (1.1) - 1:

On appelle preuve d'un algorithme, la propriété qui assure à ce dernier :
- *de se terminer. On appelle cela la* TERMINAISON *de l'algorithme*
- *de réaliser ce qu'on attend de lui. On appelle cela la* CORRECTION *de l'algorithme.*

1.2 Comment «prouver» ?

a - Terminaison

Il est fréquent dans l'établissement d'un algorithme qu'un programmeur ait recours à une structure de boucle. Lorsque cette dernière est **conditionnelle** (while), et que l'algorithme exécute une première fois les instructions contenues dans la boucle, il est important de s'assurer que **l'algorithme sortira de la boucle et se terminera.** Cette propriété de l'algorithme s'appelle **la terminaison**

Ainsi :

le groupe d'instructions de la boucle doit permettre une modification de la condition de boucle.

Propriété - (1.2) - 1:

On assure la **terminaison** *d'un algorithme lorsque toutes les structures de boucles conditionnelles de celui-ci «terminent»*

b - Correction

En outre, la seconde préoccupation du programmeur sera d'assurer que son algorithme réalise bien le travail demandé. Cette propriété de l'algorithme s'appelle **la correction**. Elle est généralement plus délicate à établir.

Propriété - (1.2) - 2:

On assure la **correction** *d'un algorithme avec boucle en dégageant une propriété vérifiée avant l'entrée dans la boucle et qui le restera durant chaque itération i de boucle ; soit $\mathcal{P}_i$ cette propriété au rang i. Cette propriété doit permettre de renvoyer le résultat attendu au dernier rang de boucle. On l'appelle* **l'invariant de boucle**.

Définition - (1.2) - 2:

On appelle invariant de boucle une propriété vraie avant l'exécution de boucle et qui le restera à chaque itération.

1.3 Quelques exemples

a - Factorielle

On considère le script python suivant :

Listing IV.1 – Factorielle de n

```
n=input()
if type(n)==int and n>=0 :
    k=1
    f=1
    while k<=n :
        f=f*k
        k=k+1
    print f
else :
    print "Impossible"
```

- TERMINAISON :
  - ▶ Si n entré au clavier est négatif, le programme termine sur un message ("impossible").
  - ▶ Si n est nul la boucle n'est pas exécutée, 1 est renvoyé et le programme termine.
  - ▶ Si $n > 0$ k étant initialement à 1, la boucle est exécutée. A chaque itération, k est incrémenté de 1 et finit par être supérieur à n donc pour k=n+1, on sort de la boucle,le programme renvoie f, et termine.

  CONCLUSION : la terminaison est assurée.

- CORRECTION :
  Un invariant de boucle $\mathcal{P}_i$ est par exemple :

  «*après la* $i^{ième}$ *itération* k *contient* $i+1$ *et* f *contient* $i!$»

  Cette propriété est vraie au rang 0. Supposons la vraie au rang i, et montrons qu'elle est héréditaire :

  - ▶ au rang $i+1$, on a : k qui contient $i+1$ en début d'itération et $f = i! \times (i+1) = (i+1)!$
  - ▶ en fin d'itération k contient $i+2$

  Ceci est bien la propriété au rang $i+1$

  CONCLUSION : la correction est assurée.

b - Puissance de 2

On considère le code python calculant la puissance $n^{ième}$ de 2 :

Listing IV.2 – Puissance de 2

```
n=input()
if type(n)==int and n>=0 :
    p=1
    while n>0 :
        p=2*p
        n=n-1
    print p
else :
    print "Impossible"
```

- TERMINAISON :
  - ▶ Si n entré au clavier n'est pas un entier positif ou nul le programme termine sur un message ("impossible").

- Si n est nul la boucle n'est pas exécutée, 1 est renvoyé et le programme termine.
- Si $n > 0$, la boucle est exécutée. A chaque itération, n est décrémenté de 1 et finit par être nul, on sort de la boucle, le programme renvoie p, et termine.

Conclusion : la terminaison est assurée.

- Correction :
Un invariant de boucle est par exemple : «*après la* $i^{ième}$ *itération* p *contient* $2^{n_0-(n_0-i)} = 2^i$ *et* n *contient* $n_i = n_0 - i$»
Les conditions initiales assure qu'au rang 0 la propriété est vraie. Supposons la vraie au rang i, et montrons qu'elle est héréditaire :
  - au rang $i+1$, on a : p qui contient $2 \times 2^{n_0-(n_0-(i+1))} = 2^{i+1}$
  - en fin d'itération n contient $n_{i+1} = n_0 - (i+1)$

  Ceci est bien la propriété au rang $i+1$

  Conclusion : la correction est assurée.

2 Complexité des algorithmes

2.1 Outils et notations

Si le programmeur veille à la preuve de son algorithme, il est également soucieux d'optimiser les performances de ce dernier afin de limiter l'occupation du microprocesseur. Toute opération ordonnée par l'algorithme au microprocesseur représente un coût en terme de **temps d'occupation de ce dernier**.

Le coût total cumulé une fois l'algorithme terminé est appelé **complexité temporelle**.

Par ailleurs, le fonctionnement d'un algorithme en machine occupe également de la mémoire.

Le coût total cumulé en occupation mémoire est appelé **complexité spatiale**.

Définition - (2.1) - 3:

Soient (f_n) et (g_n) deux suites de réels positifs.

- *(f_n) est dite* ***minorée*** *par (g_n) si et seulement si :*

$$\exists N \in \mathbb{N} \quad \exists \lambda > 0 \quad \forall n \geqslant N \quad \lambda g_n \leqslant f_n$$

On note $f_n = \Omega(g_n)$ et on lit souvent : «f_n est un grand omega de g_n ».

- *(f_n) est dite* ***majorée*** *par (g_n) si et seulement si :*

$$\exists N \in \mathbb{N} \quad \exists \mu > 0 \quad \forall n \geqslant N \quad f_n \leqslant \mu g_n$$

On note $f_n = O(g_n)$ et on lit souvent : « f_n est un grand O de g_n ».

- *(f_n) et (g_n) sont dites* ***du même ordre*** *si et seulement si :*

$$f_n = \Omega(g_n) \text{ et } f_n = O(g_n)$$

soit :

$$\exists N \in \mathbb{N} \quad \exists \lambda > 0 \quad \exists \mu > 0 \quad \forall n \geqslant N \quad \lambda g_n \leqslant f_n \leqslant \mu g_n$$

On note $f_n = \Theta(g_n)$ et on lit souvent : « f_n est un grand theta de g_n ».

De manière très schématique, la notation O permet d'évaluer la **complexité dans le pire des cas** ; la notation Θ la **complexité «en gros»**.

Exemple :

Listing IV.3 – Boucles imbriquées

```
from numpy import *
n = 3
A = empty((n,n))
for i in range(n):
        for j in range(i):
                A[i,j] = i + j
```

Dans cet exemple, on exécute n itérations de la première boucle, et pour chaque itération de celle-ci au rang i, la seconde s'execute i fois pour remplir la matrice A.

Ainsi, le nombre total d'itérations de la structure imbriquée est :

$$f(n) = (1 + 2 + 3 + \ldots + n) = n \times \frac{1+n}{2}$$

Notons $g(n) = n^2$. Il est possible de trouver deux constantes positives λ et μ telles que :

$$\lambda g(n) \leqslant f(n) \leqslant \mu g(n)$$

Par exemple, $\lambda = 1/2$ et $\mu = 1$ conviennent. On établit ainsi :

$$f(n) = O(n^2) \qquad f(n) = \Omega(n^2) \qquad f(n) = \Theta(n^2)$$

La notation $O(n^2)$ peut s'interpréter en terme de **complexité asymptotique** : lorsque n devient grand, $f(n)$ est de l'ordre de n^2. L'algorithme précédent est $O(n^2)$; son temps d'exécution n'excède pas un certain μn^2, avec $\mu > 0$.

2.2 Classification

Ces outils de comparaison permettent de classer les algorithmes selon leur complexité [1] :

- **complexité constante** en $O(1)$;
- **complexité logarithmique** en $O(\log_2 n)$;
- **complexité linéaire** en $O(n)$;
- **complexité quasi-linéaire** en $O(n \log_2 n)$;
- **complexité polynomiale** en $O(n^k)$;
- **complexité exponentielle** en $O(2^n)$;

Le tableau suivant compare ces complexités pour des tailles n de données croissantes.

n	10^2	10^3	10^4
$\ln n$	$4,6$	$6,9$	$9,2$
$n \ln n$	461	$6,9 \times 10^3$	$9,2 \times 10^4$
n^2	10^4	10^6	10^8
2^n	$> 10^{30}$	$> 10^{300}$	$> 10^{3000}$

2.3 Exemples

a - Valeur moyenne

Le script python suivant défnit une fonction qui calcule la valeur moyenne des éléments de la liste *uneListe* donnée en argument :

Listing IV.4 – Valeur moyenne des éléments d'une liste

```
1  def moyenne(uneListe) :
2      """Calcul de la moyenne d'une liste de nombres passée en argument"""
3
4      # calcul de la somme des éléments de la liste
5      somme=0.     # initialisation
6      for elt in uneListe :     # boucle sur les éléments de la liste
7          somme=somme+elt     # ajout de l'élément courant
8
9      # division de la somme par le nombre de termes
10     return somme/len(uneListe)
```

En toute rigueur, le calcul de complexité doit également tenir compte des opérations d'affectation qui représentent un coût temporel pour le processeur. On recense donc :

- 1 affectation $somme = 0$, soit 1 opération
- 1 affectation et une addition pour chaque itération, soit au total $2n$ opérations
- 1 division pour le calcul final de la moyenne soit 1 opération

Le coût total en opération est donc : $f(n) = 2n + 2$

Posons la fonction $g(n) = n$. On peut trouver (λ, μ) tel que $\lambda g(n) < f(n) < \mu g(n)$; par exemple le couple $(\lambda = 2, \mu = 4)$

1. Le logarithme en base 2, noté $\log_2$, apparait régulièrement dans les calculs de complexité. C'est pourquoi les résultats sont exprimés en ses termes.

Ainsi : $\begin{cases} \text{la complexité est «en gros» est } \Theta(n) \\ \text{la complexité «au pire» est } O(n) \end{cases}$

b - Tri «bulle»

PRINCIPE : étant donnée une liste S d'éléments, on cherche à renvoyée la liste triée de ces élements en faisant "remonter" en surface les éléments les plus grands, d'où l'appellation de tri-bulle.

L'algorithme naturel est le suivant :

```
pour i de n à 2, faire:
   pour j de 1 à i-1, faire:
       si S[j]>S[j+1] faire:
           permutation S[j] et S[j+1] dans la liste S
```

ce qui donne en script python :

Listing IV.5 –

```
1 def bulle(L):
2     for i in range(len(L)-1,0,-1):
3         for j in range(0,i):
4             if L[j]>L[j+1]:
5                 L[j],L[j+1]=L[j+1],L[j]
6             print(L)
7 liste=[5,9,1,3,2,85,45,34]
8 bulle(liste)
```

NB : ce script inscrit l'état de la liste à chaque itération permettant de visualiser l'effet "bulle" : remontée du plus grand en fin de liste.

On recense :

- une première boucle procédant à $n-1$ itérations
- une seconde boucle procédant à $i-1$ itérations, i étant le rang de la première
- Une permutation correspondant à deux opérations élémentaires (intervention d'une troisième adresse mémoire intermédiaire, non visible ici)

Soit finalement en nombre d'opérations :

$$f(n) = (1+2+3+\ldots+(n-1)) \times \underbrace{C_{perm}}_{=2} = 2 \times \underbrace{(n-1)}_{nb\ termes} \times \frac{1+n-1}{2} = n(n-1)$$

On pose $g(n) = n^2$, ainsi :
Pour $n \geq 2$, on a : $\lambda g(n) < f(n) = \mu g(n)$ avec le couple $(\lambda = 0.5, \mu = 1)$

Ainsi : $\begin{cases} \text{la complexité est «en gros» est } \Theta(n^2) \\ \text{la complexité «au pire» est } O(n^2) \end{cases}$

3 Quelques algorithmes classiques

3.1 Recherche du zero d'une fonction

a - Méthode par dichotomie

Etant donnée une fonction $f(x)$ continue et strictement monotone sur un intervalle $[a,b]$ et telle que $f(a) \times f(b) < 0$, on souhaite dégager une valeur approchée à ϵ de la solution unique de (e) $f(x) = 0$ sur $[a,b]$.

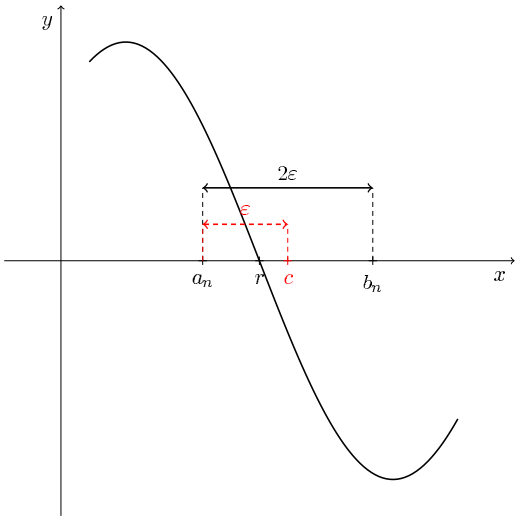


FIGURE IV.1 – Recherche d'un "zéro" de fonction par dichotomie

La méthode de recherche numérique de la solution de (e) par dichotomie est la suivante :

- On prend le milieu l'intervalle $[a,b]$: $c=\dfrac{a+b}{2}$
- On teste : si $f(a)\times f(c)>0 \implies$ la solution n'est pas dans $[a,c]$, elle est donc $[c,b]$ et on donne à a la valeur $c=\dfrac{a+b}{2}$.
- Sinon, la solution est dans l'intervalle $[a,c]$ et on donne à b valeur de $c=\dfrac{a+b}{2}$
- On itère cela jusqu'à ce que l'intervalle $[a,b]$ contenant la solution soit inférieur à la précision souhaitée err.

Le script python correspondant s'écrit :

Listing IV.6 –

```
def dichotomie(f,a,b,err):
    if a>b:
        b,a=a,b
    while (b-a)>err: #Verification de la condition d'arrêt
        c=(a+b)/2
        if f(a)*f(c)>0:
            a=c
        else:
            b=c
    return (a+b)/2

def fonc(x):
    return 2*x-1

#Programme principal
a=float(input(u"Entrez la valeur de a:"))
b=float(input(u"Entrez la valeur de b:"))
err=float(input(u"Entrez la valeur de précision: "))
print(dichotomie(fonc,a,b,err))
```

- TERMINAISON :
  A chaque itération et quelque soit l'ordre entre a,c et b, l'intervalle de recherche est divisé par 2 ; en effet :
  ▶ Si $f(a)\times f(c)>0$ l'intervalle après itération devient : $b-\dfrac{a+b}{2}=\dfrac{b-a}{2}$

► Si $f(a) \times f(c) < 0$ l'intervalle après itération devient $\frac{a+b}{2} - a = \frac{b-a}{2}$

Après i itérations, l'intervalle contenant la solution est donc $\frac{b-a}{2^i}$; ainsi, ce dernier ne fait que décroître et la condition $b - a > err$ finira par ne plus être vérifiée et l'algorithme terminera.

Conclusion : la terminaison est assurée.

- Correction :

On peut proposer comme invariant de boucle $\mathcal{P}_i$: *au rang i, la solution se trouve dans l'intervalle* $\Delta_i = \frac{b-a}{2^i}$
Cette propriété est vraie au rang nul puisque x_0 est contenu dans $b - a$ par hypothèse. Si elle est vraie au rang i, montrons qu'elle est héréditaire :
Au rang $i + 1$,

► Si $f(a) \times f(c) > 0$ l'intervalle après $i + 1$ itérations devient : $\Delta_{i+1} = \frac{1}{2}\frac{b-a}{2^i} = \frac{b-a}{2^{i+1}}$

► Si $f(a) \times f(c) < 0$ l'intervalle après $i + 1$ itérations devient $\Delta_{i+1} = \frac{1}{2}\frac{b-a}{2^i} = \frac{b-a}{2^{i+1}}$

Ceci est bien la propriété au rang $i + 1$

Conclusion : la correction est assurée.

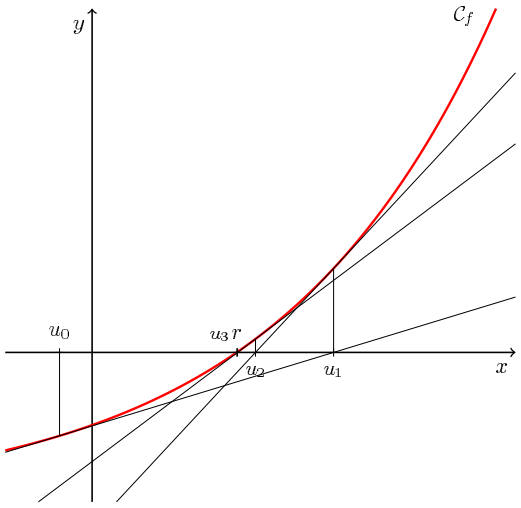
b - Méthode de Newton

Il s'agit toujours ici de résoudre l'équation $f(x) = 0$ sur l'intervalle $[a, b]$ connu pour contenir la solution. On suppose toujours que $f(a) \times f(b) \leq 0$, que f est strictement monotone sur $[a, b]$. Ajoutons comme hypothèse que la fonction est de classe $\mathcal{C}^1$ sur $[a, b]$.

On définit alors $(x_n)_{n\in\mathbb{N}}$ de la manière suivante. On prend $x_0 \in [a, b]$ le premier terme de l'itération. Pour $n \geq 0$, on calcule l'équation de la tangente au graphe de f en x_n. On définit alors x_{n+1} comme étant le point d'intersection de cette tangente avec l'axe des abscisses. La fonction f se comportant au voisinage de x_n comme sa tangente, x_{n+1} sera donc plus proche du zéro de f que x_n.

NB : L'équation de la tangente à f en x_n est : $h(x) = f'(x_n)|_{x_0} \times (x - x_n) - f(x_n)$

Cette fonction s'annule donc en : $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$



Le principe de l'algorithme est le suivant :

- On choisit $x_0 \in [a, b]$.
- On teste le critère de convergence : $|f(x_0)| < \varepsilon$.
- Si le critère n'est pas vérifié, on calcule le nouveau candidat $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$.
- On itère jusqu'à la vérification du critère d'arrêt. La solution approchée est alors x_k si l'on a produit k itérations.

On peut ainsi proposer le code suivant pour la fonction f sur $[a, b]$ avec une erreur de err :

Listing IV.7 – Méthode de Newton

```
# f est la fonction, g sa dérivée
def newton(f,g,a,err):
    x=float(a)
    fx=f(a)
    # on teste si |f(x)| est plus grand que epsilon.
    while abs(fx)>=err:
        x=x-(fx)/g(x)
        fx=f(x)
    return x
```

On remarquera que contrairement à la méthode par dichotomie, l'erreur ε n'est pas donnée sur les abscisses mais sur les ordonnées (on teste si $|f(x)| < \varepsilon$).

- Terminaison :
  - si le test d'arrêt est vérifié, alors l'algorithme renvoie $x = a$ et termine.
  - si le test d'arrêt n'est pas vérifié alors on entre dans la boucle. A la $i + 1^{\text{ième}}$ itération, x_{i+1} est plus proche de la solution que x_i (propriété de la tangente). Ainsi, $f(x_i)$ se rapproche de la solution et le critère d'arrêt $|f(x_i)| < err$ finit par être vérifié et l'algorithme termine.

  Conclusion : la terminaison est assurée.

- Correction :
  On peut proposer comme invariant de boucle $\mathcal{P}_i$: *au rang i, la solution approchée est x_i contenue dans l'intervalle $[a, b]$ et la solution vraie est dans l'intervalle $[a, b]$*
  Cette propriété est vraie au rang nul puisque la solution approchée à ce rang est $x_0 \in [a, b]$. Si elle est vraie au rang i, montrons qu'elle est héréditaire :
  Au rang $i+1$, la solution approchée est $x_{i+1} = x_i - \dfrac{f(x_i)}{f'(x_i)}$. Dans la mesure où l'interception de la tangente avec l'axe des abscisses se rapproche de la solution vraie alors la solution approchée est à fortiori dans l'intervalle $[a, b]$.
  Ceci est bien la propriété au rang $i + 1$

  Conclusion : la correction est assurée.

3.2 Calcul approché d'intégrales

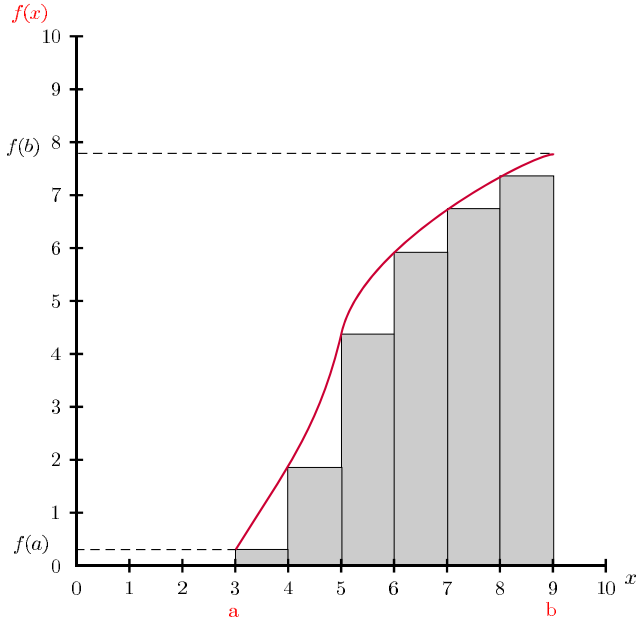
Si les méthodes, exposées plus haut, de recherche de solutions approchées d'équations sont indispensables dans les cas d'équations transcendantes (fréquentes en physique par exemple), il en va de même pour le calcul d'intégrales car très peu de primitives sont connues.

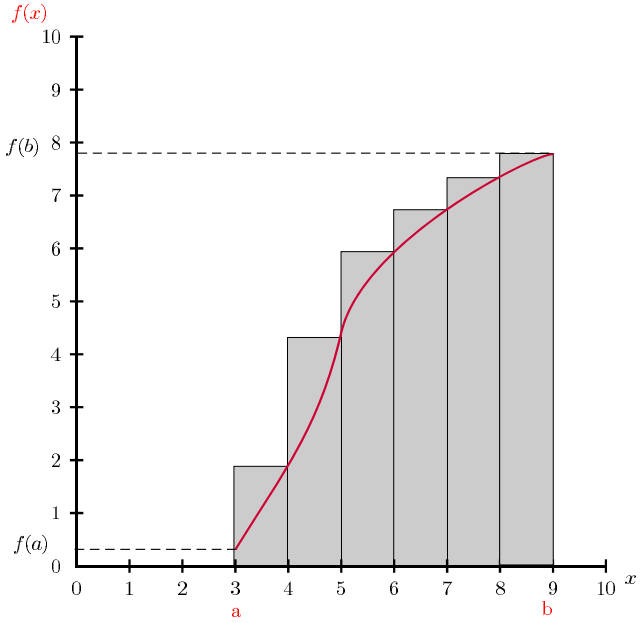
On propose dans cette partie de reprendre les principaux algorithmes de calcul approché d'intégrales vus en MPSI pour calculer l'intégrale :

$$I = \int_a^b f(x) \cdot dx$$

a - Méthode des rectangles

C'est la méthode la plus rudimentaire ; elle consiste à diviser l'intervalle $[a, b]$ en n sous-intervalles $[x_i, x_{i+1}], i \in [0, n-1]$ identiques et à remplacer $f(x)$ dans chaque sous-intervalle $[x_i, x_{i+1}]$ par $f(x_i)$ (bord à gauche) ou bien $f(x_{i+1})$ (bord à droite), puis réaliser la somme des aires des rectangles ainsi constitués :





Les aires approchées s'écrivent ainsi :

$$R_n^{(g)} = \sum_{i=0}^{n-1} (x_{i+1} - x_i) \cdot f(x_i) = \frac{b-a}{n} \sum_{i=0}^{n-1} f(x_i) \qquad \text{et} \qquad R_n^{(d)} = \frac{b-a}{n} \sum_{i=1}^{n} f(x_i)$$

Le code python pour le calcul de l'intégrale $R_n^{(g)}$ est :

Listing IV.8 – Méthode des rectangles avec bord à gauche

```
def rectanglesg(f,a,b,n):
    h=(b-a)/float(n)
    A=0
    for i in range(n) :
        A=A+f(a+i*h)
    return h*A
```

EXERCICE N°1: *Modifier le code précédent afin qu'il réalise le calcul de* $R_n^{(d)}$

EXERCICE N°2: *Proposer une méthode de calcul approché de* $ln(2)$ *par la méthode des rectangles en considérant la fonction* $f(x) = \dfrac{1}{1+x}$ *sur l'intervalle* $[0, 1]$.

ETUDE DE L'ERREUR :

A retenir :

PROPRIÉTÉ - (3.2) - 3:

Si f est de classe $\mathcal{C}_1$ sur $[a,b]$, alors en posant $M_1 = sup_{a,b}|f'|$, on a :

$$\left|\int_a^b f(x)\cdot dx - R_n^{(g)}\right| \leq \frac{M_1(b-a)^2}{2n}$$

Démonstration :

On a par l'inégalité des accroissements finis pour $\forall x \in [x_i, x_{i+1}]$:

$$|f(x) - f(x_i)| \leq M_1(x - x_i)$$

qui donne en intégrant entre x_i et x_{i+1} :

$$\left|\int_{x_i}^{x_{i+1}} (f(x) - f(x_i))\cdot dx\right| \leq \int_{x_i}^{x_{i+1}} M_1(x - x_i)\cdot dx = \frac{M_1}{2}(x_{i+1} - x_i)^2$$

or $(x_{i+1} - x_i) = \dfrac{b-a}{n}$ donc :

$$\left|\int_{x_i}^{x_{i+1}} (f(x) - f(x_i))\cdot dx\right| \leq \frac{M_1}{2}\frac{(b-a)^2}{n^2}$$

Par l'inégalité triangulaire il vient :

$$\left|\sum_{i=0}^{n-1}\int_{x_i}^{x_i+1} (f(x) - f(x_i))\cdot dx\right| \leq \sum_{i=0}^{n-1}\left|\int_{x_i}^{x_i+1} (f(x) - f(x_i))\cdot dx\right| \leq n \times \frac{M_1}{2}\frac{(b-a)^2}{n^2}$$

Ce qui permet de dégager un majorant de l'erreur ε (en négligeant toute erreur liée à la représentation des nombres en machine) :

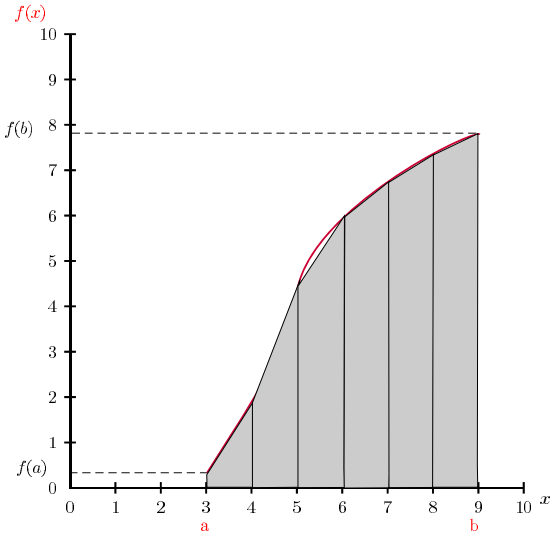
$$\varepsilon = \left|\sum_{i=0}^{n-1}\int_{x_i}^{x_i+1} f(x)\cdot dx - R_n^{(g)}\right| \leq \frac{M_1}{2}\frac{(b-a)^2}{n} \sim \frac{1}{n}$$

Conclusion : $R_n^{(g)}$ (ou bien $R_n^{(d)}$) converge bien vers I lorsque $n \longrightarrow \infty$.

COMPLEXITÉ "EN GROS" : $C(n) = \mathcal{O}(n)$

b - Méthodes des trapèzes

La méthode des trapèzes s'appuie sur le même principe que celui employé dans la méthode des rectangles, à ceci prêt que l'on remplace cette fois la courbe sur le segment de droite $[x_i, x_{i+1}]$ par le segment de droite reliant les deux points de la courbe d'abscisses x_i, et x_{i+1}, donc $(x_i, f(x_i))$ et $(x_{i+1}, f(x_{i+1}))$. Cela correspond au calcul d'une somme d'aires de trapèzes (cf schéma ci-contre).



L'aire approchée s'écrit pour un découpage en n sous intervalles :

$$T_n = \frac{b-a}{n} \sum_{i=0}^{n-1} \frac{f(x_i) + f(x_{i+1})}{2}$$

Le code python pour le calcul de l'intégrale T_n est :

Listing IV.9 – Méthode des trapèzes

```
def trapezes(f,a,b,n) :
    h=(b-a)/float(n)
    z=0.5*(f(a)+f(b))
    for i in range(1,n) :
        z=z+f(a+i*h)
    return h*z
```

Etude de l'erreur :

A retenir :

Propriété - (3.2) - 4:

Si f est de classe $\mathcal{C}^2$ sur $[a,b]$, alors en posant $M_2 = sup_{a,b}|f"|$, on peut majorer l'erreur numérique ε sur l'intégration avec :

$$\varepsilon = \left| \int_a^b f(x) \cdot dx - T_n \right| \leq \frac{M_2 (b-a)^2}{12 n^2}$$

Démonstration :

Partons de l'erreur commise sur un seul sous intervalle $[x_i, x_{i+1}]$ soit :

$$\varepsilon_i = \left| \int_{x_i}^{x_{i+1}} f(x) \cdot dx - \frac{f(x_i) + f(x_{i+1})}{2} (x_{i+1} - x_i) \right|$$

et posons la fonction $g(x)$:

$$g(x) = \int_{x_i}^{x} f(t)\cdot dt - \frac{f(x_i)+f(x)}{2}(x-x_i)$$

qui donne en dérivant une première fois

$$g'(x) = f(x) - \frac{f'(x)}{2}(x-x_i) - \frac{f(x)+f(x_i)}{2}$$

puis une seconde :

$$g"(x) = f'(x) - \frac{f"(x)}{2}(x-x_i) - \frac{f'(x)}{2} - \frac{f'(x)}{2} = -\frac{f"(x)}{2}(x-x_i)$$

donc en utlisant le majorant M_2 de $f"(x)$:

$$|g'(x)| = \int_{x_i}^{x} |g"(t)|\cdot dt \leq \frac{M_2}{2}\int_{x_i}^{x}(t-x_i)\cdot dt = \frac{M_2}{4}(x-x_i)^2$$

En intégrant de nouveau :

$$|g(x)| = \int_{x_i}^{x} |g'(t)|\cdot dt \leq \frac{M_2}{4}\int_{x_i}^{x}(t-x_i)^2\cdot dt = \frac{M_2}{12}(x-x_i)^3$$

Ainsi, l'erreur sur un sous-intervalle ε_i est majorée :

$$\varepsilon_i = |g(x_{i+1})| \leq \frac{M_2}{12}(x_{i+1}-x_i)^3 = \frac{M_2}{12n^3}(b-a)^3$$

et la majoration de l'erreur totale s'obtient évidemment par sommation sur les n intervalles et usage de l'inégalité triangulaire ; cela donne :

$$\boxed{\varepsilon = \left|\sum_{i=0}^{n-1}\int_{x_i}^{x_{i+1}} f(x)\cdot dx - T_n\right| \leq \frac{M_2}{12n^2}(b-a)^3 \sim \frac{1}{n^2}}$$

Conclusion : T_n converge bien vers I lorsque $n \longrightarrow \infty$.

Complexité "en gros" : $C(n) = \mathcal{O}(n)$

c - Méthodes natives de Python

Le module `scipy.integrate` de python possède deux commandes permettant un calcul numérique performant d'une intégrale, il s'agit de :

- `quad(f,a,b)` qui calcule une valeur approchée de $\int_a^b f(x)\cdot dx$ par une méthode optimisée en fonction des propriétés de la fonction f sur l'intervalle $[a,b]$ et
- `romberg(f,a,b)` qui fait de même en exploitant la méthode de Romberg, bien plus performante que les deux méthodes vues plus haut.

d - Performances "de terrain" : rectangles vs trapèzes vs méthodes natives

On peut par exemple comparer les différentes méthodes d'intégration en évaluant l'intégrale

$$\int_0^1 \frac{1}{1+x} \cdot dx = \ln(2) = 0.69314718056$$

n	Rectangles	Erreur ε_{rect}	Trapèzes	Erreur ε_{trap}	`quad`
1	1.0	0.30685281944	0.75	0.0568528194401	0.69314718056
10	0.718771403175	0.0256242226155	0.693771403175	0.000624222615483	– – –
100	0.695653430482	0.00250624992188	0.693153430482	$6.24992187881.10^{-6}$	– – –
1000	0.69339724306	0.000250062499992	0.69314724306	$6.24999920706.10^{-8}$	– – –
10000	0.693172181185	$2.50006249991.10^{-5}$	0.693147181185	$6.24999163534.10^{-10}$	– – –