Révisions 4: Algorithmique: preuve, complexité, exemples

(Révisions MPSI)

MP3 Lycée Montaigne

Septembre-Octobre 2019

Plan

Définition

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.

Comment «prouver» ?

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 :

Comment «prouver» ?

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.

Comment «prouver» ?

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.

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

Comment «prouver» ?

Correction

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 $\mathscr{P}_i$ cette propriété au rang i. Cette propriété doit permettre de renvoyer le résultat attendu au dernier rang de boucle. C'est **l'invariant de boucle**.

Comment «prouver» ?

Correction

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 $\mathscr{P}_i$ cette propriété au rang i. Cette propriété doit permettre de renvoyer le résultat attendu au dernier rang de boucle. C'est **l'invariant de boucle**.

On appelle **invariant de boucle** une propriété $\mathscr{P}$ vraie avant l'exécution de boucle et qui le restera à chaque itération.

Quelques exemples

Factorielle

```
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 :

Quelques exemples

Factorielle

```
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").

Quelques exemples

Factorielle

```
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.

Quelques exemples

Factorielle

```
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"
```

<u>TERMINAISON</u> :

- 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 :

Quelques exemples

Factorielle

```
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.

Quelques exemples

Factorielle

```
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"
```

<u>CORRECTION</u> :

Un invariant de boucle $\mathscr{P}_i$ est par exemple :

Quelques exemples

Factorielle

```
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"
```

Correction :

Un invariant de boucle $\mathscr{P}_i$ est par exemple :

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

Quelques exemples

Factorielle

```
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"
```

<u>CORRECTION</u> :

Un invariant de boucle $\mathscr{P}_i$ est par exemple :

«*après la $i^{\text{ième}}$ 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 :

Quelques exemples

Factorielle

```
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"
```

<u>CORRECTION</u> :

Un invariant de boucle $\mathscr{P}_i$ est par exemple :

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

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 $f = i! \times (i+1) = (i+1)!$

Quelques exemples

Factorielle

```
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"
```

<u>CORRECTION</u> :

Un invariant de boucle $\mathscr{P}_i$ est par exemple :

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

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 $i+2$

Quelques exemples

Factorielle

```
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"
```

Correction :

Un invariant de boucle $\mathscr{P}_i$ est par exemple :

«*après la $i^{\text{ème}}$ 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 $i+1$

Quelques exemples

Factorielle

```
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"
```

<u>CORRECTION</u> :

Un invariant de boucle $\mathscr{P}_i$ est par exemple :

«*après la $i^{\text{ième}}$ 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 :

Quelques exemples

Factorielle

```
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"
```

Correction :

Un invariant de boucle $\mathscr{P}_i$ est par exemple :

«*après la $i^{\text{ième}}$ 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.

Quelques exemples

Puissance de 2

On considère le code python calculant la puissance $n^{\text{ième}}$ 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"
```

<u>Terminaison</u> :

Quelques exemples

Puissance de 2

On considère le code python calculant la puissance $n^{\text{ième}}$ 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"
```

<u>TERMINAISON</u> :

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

Quelques exemples

Puissance de 2

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

```python
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.

Quelques exemples

Puissance de 2

On considère le code python calculant la puissance $n^{\text{ième}}$ 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.

Quelques exemples

Puissance de 2

On considère le code python calculant la puissance $n^{\text{ième}}$ 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 :

Quelques exemples

Puissance de 2

On considère le code python calculant la puissance $n^{\text{ième}}$ 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.

Quelques exemples

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"
```

<u>Correction</u> :

Un invariant de boucle est par exemple :

Quelques exemples

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"
```

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 $n_i=n_0-i$»

Quelques exemples

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"
```

<u>CORRECTION</u> :

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 :

Quelques exemples

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"
```

<u>CORRECTION</u> :

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 $2\times 2^{n_0-(n_0-(i+1))}=2^{i+1}$

Quelques exemples

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"
```

CORRECTION :
Un invariant de boucle est par exemple :
«après la $i^{\text{ième}}$ 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 $n_{i+1}=n_0-(i+1)$

Quelques exemples

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"
```

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 $i+1$

Quelques exemples

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"
```

<u>CORRECTION</u> :

Un invariant de boucle est par exemple :

«après la $i^{\text{ième}}$ 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 :

Quelques exemples

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"
```

<u>CORRECTION</u> :

Un invariant de boucle est par exemple :

«*après la* $i^{\text{ième}}$ *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.

Outils et notations

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***.*

Outils et notations

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

Outils et notations

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 ».

Outils et notations

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 ».

Outils et notations

- (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 ».

$\Longrightarrow$

Outils et notations

- (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 ».

$\Longrightarrow \begin{cases} \text{la notation } O \text{ permet d'évaluer la } \textbf{complexité dans le pire des cas} \\ \text{la notation } \Theta \text{ la } \textbf{complexité «en gros»} \end{cases}$

Outils et notations

EXEMPLE :

```
import numpy as np
n = 3
A = np.zeros((n,n))
for i in range(n):
    for j in range(i+1):
        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.

$\Longrightarrow$

Outils et notations

<u>EXEMPLE :</u>

```python
import numpy as np
n = 3
A = np.zeros((n,n))
for i in range(n):
    for j in range(i+1):
        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.

$\Longrightarrow$ nombre total d'itérations : $f(n) = (1+2+3+\ldots+n) = n \times \dfrac{1+n}{2}$

Outils et notations

Notons $g(n) = n^2$. On peut avoir :

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

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 $\mu > 0$.

Classification

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

- **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	$> 10^{30}$	$> 10^{300}$	$> 10^{3000}$

Exemples

Valeur moyenne

```
def moyenne(uneListe) :
    """Calcul de la moyenne d'une liste de nombres passÃ©e en argument"""

    # calcul de la somme des Ã©lÃ©ments de la liste
    somme=0.     # initialisation
    for elt in uneListe :    # boucle sur les Ã©lÃ©ments de la liste
        somme=somme+elt     # ajout de l'Ã©lÃ©ment courant

    # division de la somme par le nombre de termes
    return somme/len(uneListe)
```

Exemples

Valeur moyenne

```python
def moyenne(uneListe) :
    """Calcul de la moyenne d'une liste de nombres passÃ©e en argument"""

    # calcul de la somme des Ã©lÃ©ments de la liste
    somme=0.    # initialisation
    for elt in uneListe :    # boucle sur les Ã©lÃ©ments de la liste
        somme=somme+elt    # ajout de l'Ã©lÃ©ment courant

    # division de la somme par le nombre de termes
    return somme/len(uneListe)
```

- 1 affectation $somme = 0$, soit 1 opération

Exemples

Valeur moyenne

```
def moyenne(uneListe) :
    """Calcul de la moyenne d'une liste de nombres passÃ©e en argument"""

    # calcul de la somme des Ã©lÃ©ments de la liste
    somme=0.    # initialisation
    for elt in uneListe :    # boucle sur les Ã©lÃ©ments de la liste
        somme=somme+elt    # ajout de l'Ã©lÃ©ment courant

    # division de la somme par le nombre de termes
    return somme/len(uneListe)
```

- 1 affectation $somme = 0$, soit 1 opération
- 1 affectation et une addition pour chaque itération, soit au total $2n$ opérations

Exemples

Valeur moyenne

```python
def moyenne(uneListe) :
    """Calcul de la moyenne d'une liste de nombres passÃ©e en argument"""

    # calcul de la somme des Ã©lÃ©ments de la liste
    somme=0.    # initialisation
    for elt in uneListe :    # boucle sur les Ã©lÃ©ments de la liste
        somme=somme+elt    # ajout de l'Ã©lÃ©ment courant

    # division de la somme par le nombre de termes
    return somme/len(uneListe)
```

- 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

Exemples

Valeur moyenne

```
def moyenne(uneListe) :
    """Calcul de la moyenne d'une liste de nombres passÃ©e en argument"""

    # calcul de la somme des Ã©lÃ©ments de la liste
    somme=0.    # initialisation
    for elt in uneListe :    # boucle sur les Ã©lÃ©ments de la liste
        somme=somme+elt    # ajout de l'Ã©lÃ©ment courant

    # division de la somme par le nombre de termes
    return somme/len(uneListe)
```

- 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

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

Exemples

Valeur moyenne

On pose $g(n) = n$
$\exists\ (\lambda, \mu)$ tel que $\lambda g(n) < f(n) < \mu g(n)$; par exemple le couple $(\lambda = 2, \mu = 4)$

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

Exemples

Tri «bulle»

Principe : étant donnée une liste S d'éléments, on cherche à renvoyer 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 :

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

Exemples

Tri «bulle»

On recense :

Exemples

Tri «bulle»

On recense :

- une première boucle procédant à $n-1$ itérations

Exemples

Tri «bulle»

On recense :

- une première boucle procédant à $n-1$ itérations
- une seconde boucle procédant à i itérations, i étant le rang de la première

Exemples

Tri «bulle»

On recense :

- une première boucle procédant à $n-1$ itérations
- une seconde boucle procédant à i 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)

Exemples

Tri «bulle»

On recense :

- une première boucle procédant à $n-1$ itérations
- une seconde boucle procédant à i 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)

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

On pose $g(n) = n^2$

Exemples

Tri «bulle»

On recense :

- une première boucle procédant à $n-1$ itérations
- une seconde boucle procédant à i 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)

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

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

Exemples

Tri «bulle»

On recense :

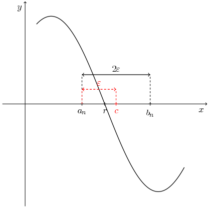
- une première boucle procédant à $n-1$ itérations
- une seconde boucle procédant à i itérations, i étant le rang de la première
- une permutation correspondant à deux opérations élémentaires (intervention d'une troisième adresse mémoire intermédiaire, non visible ici)

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

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

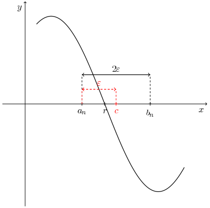
$$\begin{cases} \text{la complexité est «en gros» est } \Theta(n^2) \\ \text{la complexité «au pire» est } O(n^2) \end{cases}$$

Recherche du zero d'une fonction par dichotomie



La méthode de recherche numérique par dichotomie de la solution de $f(x) = 0$ contenue dans $[a, b]$ pour f strictement monotone sur [a,b] est la suivante :

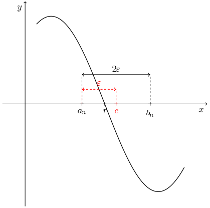
Recherche du zero d'une fonction par dichotomie



La méthode de recherche numérique par dichotomie de la solution de $f(x)=0$ contenue dans $[a,b]$ pour f strictement monotone sur [a,b] est la suivante :

- On prend le milieu l'intervalle $[a,b]$: $c = \dfrac{a+b}{2}$

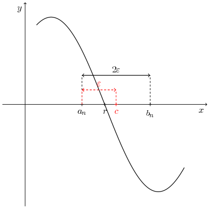
Recherche du zero d'une fonction par dichotomie



La méthode de recherche numérique par dichotomie de la solution de $f(x)=0$ contenue dans $[a,b]$ pour f strictement monotone sur [a,b] 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 dans l'intervalle $[c,b]$ et on donne à a la valeur $c=\dfrac{a+b}{2}$.

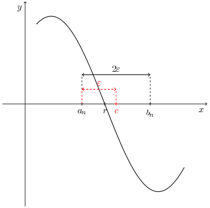
Recherche du zero d'une fonction par dichotomie



La méthode de recherche numérique par dichotomie de la solution de $f(x) = 0$ contenue dans $[a, b]$ pour f strictement monotone sur [a,b] 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 dans l'intervalle $[c, b]$ 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 $c = \dfrac{a+b}{2}$

Recherche du zero d'une fonction par dichotomie



La méthode de recherche numérique par dichotomie de la solution de $f(x) = 0$ contenue dans $[a, b]$ pour f strictement monotone sur [a,b] 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 dans l'intervalle $[c, b]$ 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*.

Recherche du zero d'une fonction par dichotomie

Le script python correspondant s'écrit :

```
def dichotomie(f,a,b,err):
    if a>b:
        b,a=a,b
    while (b-a)>err: #Verification de la condition d'arrÃªt
        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␣prÃ©cision:␣"))
print(dichotomie(fonc,a,b,err))
```

Recherche du zero d'une fonction par dichotomie

Terminaison :

A chaque itération l'intervalle de recherche est divisé par 2 ; en effet :

Recherche du zero d'une fonction par dichotomie

TERMINAISON :

A chaque itération l'intervalle de recherche est divisé par 2 ; en effet :

- Si $f(a) \times f(c) > 0$ l'intervalle après itération devient : $b - \dfrac{a+b}{2} = \dfrac{b-a}{2}$

Recherche du zero d'une fonction par dichotomie

TERMINAISON :

A chaque itération 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 $\dfrac{a+b}{2} - a = \dfrac{b-a}{2}$

Recherche du zero d'une fonction par dichotomie

TERMINAISON :
A chaque itération 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 $\dfrac{a+b}{2} - a = \dfrac{b-a}{2}$

$\Longrightarrow$

Recherche du zero d'une fonction par dichotomie

TERMINAISON :
A chaque itération 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 $\dfrac{a+b}{2}-a=\dfrac{b-a}{2}$

$\Longrightarrow$ après i itérations, l'intervalle contenant la solution est $\dfrac{b-a}{2^i}$ $\Longrightarrow$ intervalle décroissant $\Longrightarrow$ la condition $b-a>err$ finira par ne plus être vérifiée et l'algorithme terminera.

Recherche du zero d'une fonction par dichotomie

TERMINAISON :
A chaque itération 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 $\dfrac{a+b}{2} - a = \dfrac{b-a}{2}$

$\Longrightarrow$ après i itérations, l'intervalle contenant la solution est $\dfrac{b-a}{2^i}$ $\Longrightarrow$ intervalle décroissant $\Longrightarrow$ la condition $b - a > err$ finira par ne plus être vérifiée et l'algorithme terminera.

CONCLUSION :

Recherche du zero d'une fonction par dichotomie

TERMINAISON :

A chaque itération 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 $\dfrac{a+b}{2} - a = \dfrac{b-a}{2}$

$\Longrightarrow$ après i itérations, l'intervalle contenant la solution est $\dfrac{b-a}{2^i}$ $\Longrightarrow$ intervalle décroissant $\Longrightarrow$ la condition $b - a > err$ finira par ne plus être vérifiée et l'algorithme terminera.

CONCLUSION : la terminaison est assurée.

Recherche du zero d'une fonction par dichotomie

<u>CORRECTION :</u>
On peut proposer comme invariant de boucle $\mathscr{P}_i$:

$$\textit{au rang } i\textit{, la solution se trouve dans l'intervalle } \Delta_i = \frac{b-a}{2^i}$$

Recherche du zero d'une fonction par dichotomie

CORRECTION :

On peut proposer comme invariant de boucle $\mathscr{P}_i$:

$$\textit{au rang } i\textit{, la solution se trouve dans l'intervalle } \Delta_i = \frac{b-a}{2^i}$$

Cette propriété est vraie au rang nul puisque x_0 est contenu dans $b-a$ par hypothèse. Supposons la vérifiée au rang i et montrons qu'elle est héréditaire :

Recherche du zero d'une fonction par dichotomie

CORRECTION :

On peut proposer comme invariant de boucle $\mathscr{P}_i$:

$$\textit{au rang } i\textit{, la solution se trouve dans l'intervalle } \Delta_i = \frac{b-a}{2^i}$$

Cette propriété est vraie au rang nul puisque x_0 est contenu dans $b-a$ par hypothèse. Supposons la vérifiée au rang i et montrons qu'elle est héréditaire :
Au rang $i+1$:

Recherche du zero d'une fonction par dichotomie

CORRECTION :

On peut proposer comme invariant de boucle $\mathscr{P}_i$:

$$\textit{au rang } i\textit{, la solution se trouve dans l'intervalle } \Delta_i = \frac{b-a}{2^i}$$

Cette propriété est vraie au rang nul puisque x_0 est contenu dans $b-a$ par hypothèse. Supposons la vérifiée au rang i et montrons qu'elle est héréditaire :
Au rang $i+1$:

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

Recherche du zero d'une fonction par dichotomie

Correction :

On peut proposer comme invariant de boucle $\mathscr{P}_i$:

$$\textit{au rang } i\textit{, la solution se trouve dans l'intervalle } \Delta_i = \frac{b-a}{2^i}$$

Cette propriété est vraie au rang nul puisque x_0 est contenu dans $b-a$ par hypothèse. Supposons la vérifiée au rang i et 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
  $\Delta_{i+1} = \frac{1}{2}\frac{b-a}{2^i} = \frac{b-a}{2^{i+1}}$

Recherche du zero d'une fonction par dichotomie

Correction :

On peut proposer comme invariant de boucle $\mathscr{P}_i$:

$$\textit{au rang } i\textit{, la solution se trouve dans l'intervalle } \Delta_i = \frac{b-a}{2^i}$$

Cette propriété est vraie au rang nul puisque x_0 est contenu dans $b-a$ par hypothèse. Supposons la vérifiée au rang i et 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 $i+1$

Recherche du zero d'une fonction par dichotomie

CORRECTION :

On peut proposer comme invariant de boucle $\mathscr{P}_i$:

$$\textit{au rang } i\textit{, la solution se trouve dans l'intervalle } \Delta_i = \frac{b-a}{2^i}$$

Cette propriété est vraie au rang nul puisque x_0 est contenu dans $b-a$ par hypothèse. Supposons la vérifiée au rang i et 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 :

Recherche du zero d'une fonction par dichotomie

<u>CORRECTION :</u>
On peut proposer comme invariant de boucle $\mathscr{P}_i$:

$$\textit{au rang } i\textit{, la solution se trouve dans l'intervalle } \Delta_i = \frac{b-a}{2^i}$$

Cette propriété est vraie au rang nul puisque x_0 est contenu dans $b-a$ par hypothèse. Supposons la vérifiée au rang i et montrons qu'elle est héréditaire :
Au rang $i+1$:

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

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

CONCLUSION : **la correction est assurée.**

Recherche du zéro d'une fonction par méthode de Newton

HYPOTHÈSES :

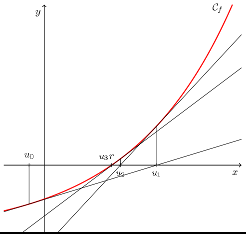
$\exists\ x / f(x) = 0$ sur l'intervalle $[a, b]$
$f(a) \times f(b) \leq 0$
f est strictement monotone sur $[a, b]$
f de classe $\mathscr{C}^1$ sur $[a, b]$

Recherche du zéro d'une fonction par méthode de Newton

$(x_n)_{n\in\mathbb{N}}$ défini 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 : $h(x) = f'(x)\big|_{x_n} \times (x - x_n) + f(x_n)$
Cette fonction s'annule donc en :

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



Recherche du zéro d'une fonction par méthode de Newton

Le principe de l'algorithme est le suivant :

Recherche du zéro d'une fonction par méthode de Newton

Le principe de l'algorithme est le suivant :

- On choisit $x_0 \in [a, b]$.

Recherche du zéro d'une fonction par méthode de Newton

Le principe de l'algorithme est le suivant :

- On choisit $x_0 \in [a, b]$.
- On teste le critère de convergence : $|f(x_0)| < \varepsilon$.

Recherche du zéro d'une fonction par méthode de Newton

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

Recherche du zéro d'une fonction par méthode de Newton

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.

Recherche du zéro d'une fonction par méthode de Newton

Proposition de script Python :

```
# f est la fonction , g sa dÃ©rivÃ©e
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 $|f(x)| < \varepsilon$).

Recherche du zéro d'une fonction par méthode de Newton

Terminaison :

Recherche du zéro d'une fonction par méthode de Newton

Terminaison :

- si le test d'arrêt est vérifié, alors l'algorithme renvoie $x = a$ et termine.

Recherche du zéro d'une fonction par méthode de Newton

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.

Recherche du zéro d'une fonction par méthode de Newton

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 :

Recherche du zéro d'une fonction par méthode de Newton

<u>TERMINAISON :</u>

- 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.

Recherche du zéro d'une fonction par méthode de Newton

<u>CORRECTION :</u>
On peut proposer comme invariant de boucle $\mathscr{P}_i$:

au rang i, la solution approchée est x_i contenue dans l'intervalle $[a, b]$ et la solution vraie est dans l'intervalle $[a, b]$

Recherche du zéro d'une fonction par méthode de Newton

CORRECTION :

On peut proposer comme invariant de boucle $\mathscr{P}_i$:

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 $x_0 \in [a, b]$

Recherche du zéro d'une fonction par méthode de Newton

CORRECTION :

On peut proposer comme invariant de boucle $\mathscr{P}_i$:

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 :

Recherche du zéro d'une fonction par méthode de Newton

CORRECTION :

On peut proposer comme invariant de boucle $\mathscr{P}_i$:

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 $[a,b]$.

Recherche du zéro d'une fonction par méthode de Newton

Correction :

On peut proposer comme invariant de boucle $\mathscr{P}_i$:

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 $i+1$

Recherche du zéro d'une fonction par méthode de Newton

CORRECTION :

On peut proposer comme invariant de boucle $\mathscr{P}_i$:

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 :

Recherche du zéro d'une fonction par méthode de Newton

Correction :

On peut proposer comme invariant de boucle $\mathscr{P}_i$:

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.

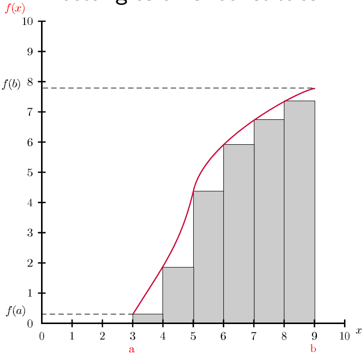
Calcul approché d'intégrales

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

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

Méthode des rectangles (bord à gauche)

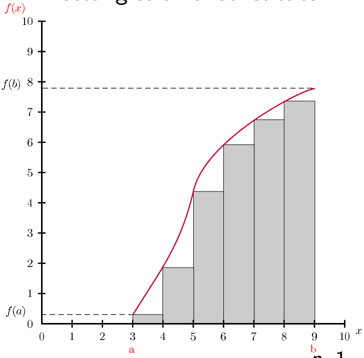
PRINCIPE : on divise l'intervalle $[a,b]$ en n sous-intervalles $[x_i, x_{i+1}], i \in [0, n-1]$ identiques et on remplace $f(x)$ dans chaque sous-intervalle par $f(x_i)$ (bord à gauche) ou bien $f(x_{i+1})$ (bord à droite), puis on réalise la somme des aires des rectangles ainsi constitués :



Aire approchée avec bord à gauche :

Méthode des rectangles (bord à gauche)

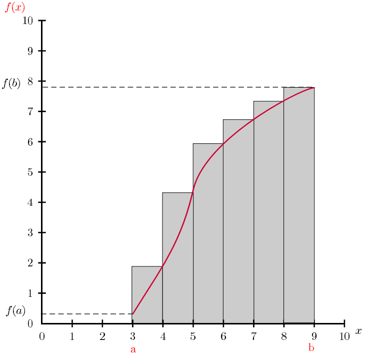
PRINCIPE : on divise l'intervalle $[a,b]$ en n sous-intervalles $[x_i, x_{i+1}], i \in [0, n-1]$ identiques et on remplace $f(x)$ dans chaque sous-intervalle par $f(x_i)$ (bord à gauche) ou bien $f(x_{i+1})$ (bord à droite), puis on réalise la somme des aires des rectangles ainsi constitués :



Aire approchée avec bord à gauche :

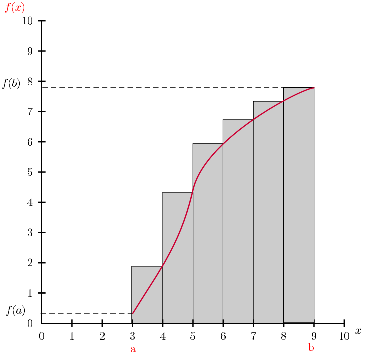
$$R_n^{(g)} = \sum_{i=0}^{n-1} (x_{i+1} - x_i) \cdot f(x_i) = \frac{b-a}{n} \sum_{i=0}^{n-1} f(x_i)$$

Méthode des rectangles (bord à droite)



Aire approchée avec bord à droite :

Méthode des rectangles (bord à droite)



Aire approchée avec bord à droite :

$$R_n^{(d)} = \frac{b-a}{n} \sum_{i=1}^{n} f(x_i)$$

Méthode des rectangles

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

```
def rectangles(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 $R_n^{(d)}$

Solution :

Méthode des rectangles

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

```
def rectangles(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°2:

Modifier le code précédent afin qu'il réalise le calcul de $R_n^{(d)}$

Solution :

```
def rectangles_droite(f,a,b,n):
    h=(b-a)/float(n)
    A=0
    for i in range(1,n+1) :
        A=A+f(a+i*h)
    return h*A
```

Méthode des rectangles

Exercice n°3:

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* $[0,1]$.

Méthode des rectangles

ETUDE DE L'ERREUR :

A retenir :

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

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

Méthode des rectangles

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

Méthode des rectangles

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} :

Méthode des rectangles

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} :

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

Méthode des rectangles

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 :

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

Méthode des rectangles

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) :

.

Méthode des rectangles

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) :

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

.

Méthode des rectangles

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) :

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

<u>CONCLUSION :</u>

.

Méthode des rectangles

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) :

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

<u>Conclusion :</u> $R_n^{(g)}$ $\left(\text{ou bien } R_n^{(d)}\right)$ converge bien vers I lorsque $n \longrightarrow \infty$ avec une erreur décroissant en $\dfrac{1}{n}$.

Méthode des rectangles

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) :

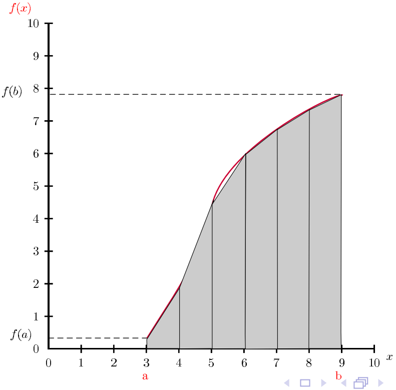
$$\boxed{\varepsilon = \left|\sum_{i=0}^{n-1}\int_{x_i}^{x_i+1} f(x)\cdot dx - R_n^{(g)}\right| \leq \frac{M_1}{2}\frac{(b-a)^2}{n} \sim \frac{1}{n}}$$

<u>Conclusion :</u> $R_n^{(g)}$ $\left(\text{ou bien } R_n^{(d)}\right)$ converge bien vers I lorsque $n \longrightarrow \infty$ avec une erreur décroissant en $\dfrac{1}{n}$.

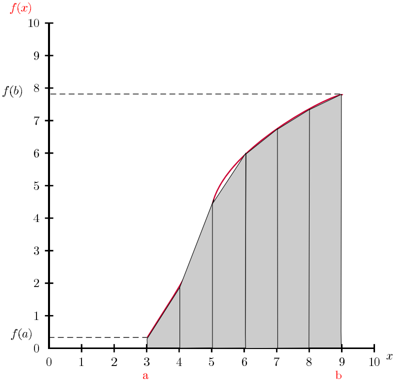
<u>Complexité "en gros" :</u> $C(n) = \mathcal{O}(n)$

Méthode 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 le segment de droite horizontal entre les abscisses x_i et 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 :



Méthode des trapèzes



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

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

Méthode des trapèzes

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

```
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
```

Méthode des trapèzes

Etude de l'erreur :

A retenir :

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

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

Méthode des trapèzes

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

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

Méthode des trapèzes

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

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

Posons la fonction $g(x)$:

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

Méthode des trapèzes

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

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

Posons la fonction $g(x)$:

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

<u>NB :</u> notons tout de suite que $\epsilon_{\mathbf{i}} = |\mathbf{g}(\mathbf{x}_{\mathbf{i}+1})|$

Méthode des trapèzes

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

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

Posons la fonction $g(x)$:

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

<u>NB :</u> notons tout de suite que $\boldsymbol{\epsilon_i = |g(x_{i+1})|}$
Par une première dérivation il vient :

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

Méthode des trapèzes

En dérivant une seconde fois, on obtient :

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

Méthode des trapèzes

En dérivant une seconde fois, on obtient :

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

donc en intégrant et en utlisant le majorant M_2 de $f''(x)$:

Méthode des trapèzes

En dérivant une seconde fois, on obtient :

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

donc en intégrant et 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 :

Méthode des trapèzes

En dérivant une seconde fois, on obtient :

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

donc en intégrant et 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 :

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

Méthode des trapèzes

En dérivant une seconde fois, on obtient :

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

donc en intégrant et 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 :

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

L'erreur sur un sous-intervalle ε_i est donc majorée :

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

Méthode des trapèzes

Par sommation sur les n intervalles et usage de l'inégalité triangulaire, on dégage la majoration attendue de l'erreur totale :

Méthode des trapèzes

Par sommation sur les n intervalles et usage de l'inégalité triangulaire, on dégage la majoration attendue de l'erreur totale :

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

Méthode des trapèzes

Par sommation sur les n intervalles et usage de l'inégalité triangulaire, on dégage la majoration attendue de l'erreur totale :

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

CONCLUSION :

Méthode des trapèzes

Par sommation sur les n intervalles et usage de l'inégalité triangulaire, on dégage la majoration attendue de l'erreur totale :

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

CONCLUSION : T_n converge bien vers I lorsque $n \longrightarrow \infty$ avec une erreur décroissant en $\dfrac{1}{n^2}$.

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

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 :

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 $[a, b]$

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

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.

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

On peut par exemple comparer les performances des différentes méthodes numériques précédentes 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	$6.24999163534.10^{-10}$	$---$