Cours 12 : Intégration numérique II
Primitivation, Méthode de Romberg, `scipy.integrate`

MPSI - Lycée Thiers

2014/2015

Introduction

• Il est nécessaire de connaître diverses méthodes théoriques de calcul approché d'intégrales, de résolution d'équations différentielles, ne serait-ce que pour :

1. avoir un aperçu des idées sous-jacentes,
2. pouvoir les implémenter dans un langage de programmation ne disposant pas de modules spécifiques de calcul numérique (comme c'est le cas de la plupart des langages),
3. les adapter à un problème spécifique.

• Mais il est aussi nécessaire au scientifique et à l'ingénieur de savoir utiliser les outils offerts par les logiciels dédiés au calcul numérique : dans 9 cas sur 10, il n'est pas nécessaire, voire même plutôt totalement improductif de "réinventer la roue".

• Des logiciels puissants tels :

1. `scilab`, `matlab`, ou
2. le module `scipy` de `python`

implémentent ces méthodes au moins aussi bien qu'on ne pourrait le faire (sauf cas spécifiques). Nous allons en voir certaines.

Primitivation numérique

• Pour chercher la primitive d'une application continue f on peut appliquer les méthodes des rectangles, et des trapèzes.

1. Il suffit d'avoir les valeurs de l'application f aux points d'une subdivision régulière. Par exemple, si l'application f est donnée :

```
import numpy as np
f = lambda x: x**2    # Application continue à primitiver
n = 100             # Nombre de segments
X = np.linspace(0,1,n+1)    # On primitivera entre 0 et 1
Y = f(X)             # Valeurs de f aux points de X
```

2. On construit un tableau de même taille. On choisit la valeur initiale (choix de la primitive) :

```
Pr = np.empty(n+1)
Pr[0] = 0     # Choix de la primitive
h = 1/n    # pas dX = (b-a)/n
```

3. Par la **méthode des rectangles** :

$$Pr(x + dx) = Pr(x) + dx \cdot f(x)$$

Connaissant `Pr[k]`, `h`, `Y`, on obtient le point suivant `Pr[k+1]` :

`Pr[k+1] = Pr[k] + h * Y[k]`

```
for k in range(1,n+1):
    Pr[k] = Pr[k-1] + h * Y[k-1]
```

Primitivation numérique

• Pour chercher la primitive d'une application continue f on peut appliquer les méthodes des rectangles, et des trapèzes.

1. Il suffit d'avoir les valeurs de l'application f aux points d'une subdivision régulière. Par exemple, si l'application f est donnée :

```
import numpy as np
f = lambda x: x**2    # Application continue à primitiver
n = 100             # Nombre de segments
X = np.linspace(0,1,n+1)    # On primitivera entre 0 et 1
Y = f(X)             # Valeurs de f aux points de X
```

2. On construit un tableau de même taille. On choisit la valeur initiale (choix de la primitive) :

```
Pt = np.empty(n+1)
Pt[0] = 0     # Choix de la primitive
```

3. Par la **méthode des trapezes :**

$$Pt(x + dx) = Pt(x) + dx \cdot (f(x) + f(x + dx))/2$$

Connaissant `Pt[k]`, `h`, `Y`, on obtient le point suivant `Pt[k+1]` :

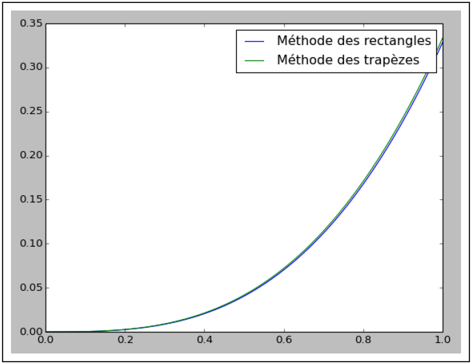
`Pt[k+1] = Pt[k] + h * (Y[k]+Y[k+1])/2`

```
for k in range(1,n+1):
    Pt[k] = Pt[k-1] + h * (Y[k-1] + Y[k])/2
```

Primitivation numérique

- On peut tracer la solution trouvée :

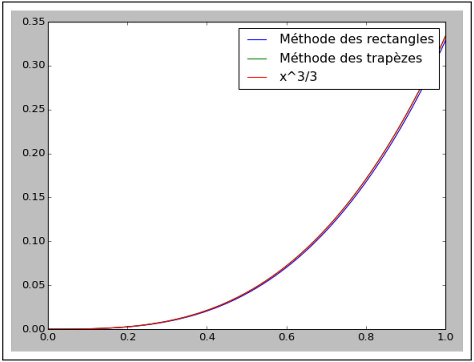
```
import matplotlib.pyplot as plt
plt.figure("Primitivation de x -> x2̂")
plt.plot(X,Pr,X,Pt)
plt.legend(('Méthode des rectangles','Méthode des trapèzes'))
plt.show()
```



Primitivation numérique

- On peut comparer avec la solution exacte (valant 0 en 0) $x \mapsto x^3/3$:

```
plt.figure(2)
plt.plot(X,Pr,X,Pt,X,X**3/3)
plt.legend(('Méthode des rectangles','Méthode des trapèzes','x3̂/3'))
plt.show()
```



Primitivation numérique

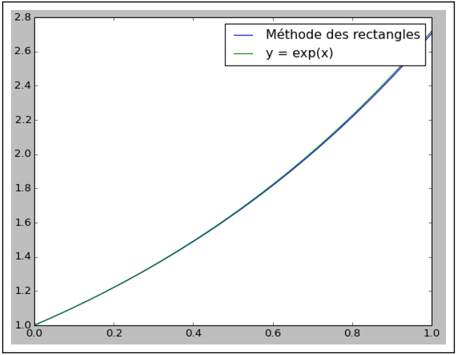
• Application : calcul approché de la fonction *exp* par la méthode des rectangles :

Elle vérifie $exp' = exp$ et $exp(0) = 1$.

```
# Application : le calcul de approché de exp(x) entre 0 et A
A = 1
n = 100    # Nombre de segments
X = np.linspace(0,A,n+1)
Exp = np.empty(n+1)
Exp[0] = 1
h = A/n
for k in range(1,n+1):
    Exp[k] = Exp[k-1] + h * Exp[k-1]
# Tracé
plt.figure(3)
plt.plot(X,Exp, X, np.exp(X))
plt.legend(('Méthode des rectangles','y = exp(x)'))
titre = 'Avec ' + str(n) + ' segments'
plt.title(titre)
plt.show()
```

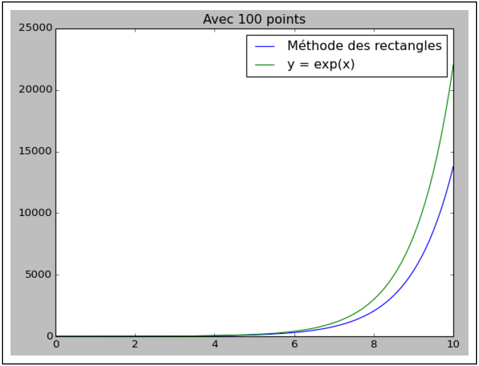
Primitivation numérique

- Sur $[0, 1]$, avec 100 segments (`A = 1 ; n = 100`) :



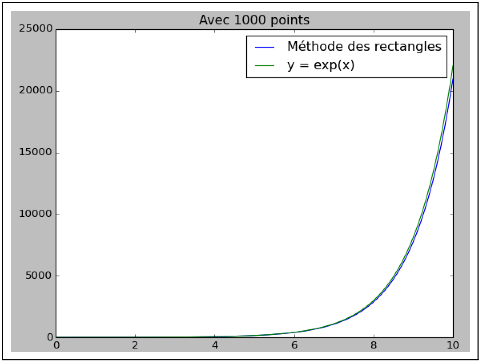
Primitivation numérique

- Sur $[0, 10]$, avec 100 segments (`A = 10 ; n = 100`) :



Primitivation numérique

- Sur $[0, 10]$, avec 1000 segments (`A = 10 ; n = 1000`) :



Primitivation numérique sous scipy

• Tout d'abord nous aurons besoin d'importer `numpy` pour bénéficier de la fonction `linspace()` :

```
>>> import numpy as np
```

• Ensuite, pour la primitivation, l'intégration (et la résolution d'équations différentielles), nous ferons usage du seul sous-module `integrate` de `scipy` :

```
>>> from scipy import integrate
```

• `scipy` bénéficie de nombreux autres modules pour d'autres domaines du calcul numérique, voir par exemple : `http://scipy-lectures.github.io/intro/scipy.html`, ou encore : `http://scipy.org`.

• Pour la primitivation nous utiliserons dans le sous-module `integrate` la méthode :

`cumtrapz()`

qui implémente la méthode des trapèzes, sur un échantillonnage (= tableau de valeurs) de la fonction à primitiver,

à l'aide de l'instruction :

```
>>> integrate.cumtrapz(paramètres)
```

Primitivation numérique sous scipy : cumtrapz()

• La méthode cumtrapz() de scipy.integrate implémente la méthode des trapèzes.

• Elle s'applique à un échantillonnage de la fonction, c'est à dire à ses valeurs prises sur tous les points d'une subdivision régulière.

Exemple : primitivation de $x \mapsto x^2$ sur l'intervalle $[0, 1]$:

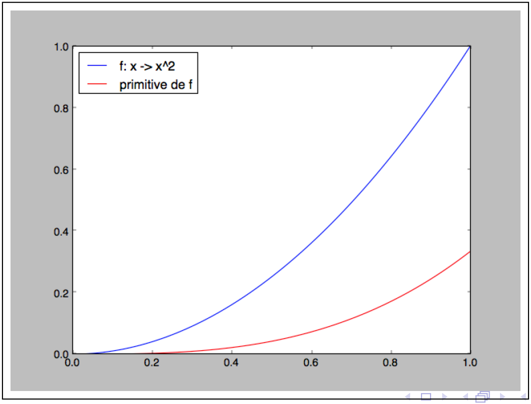
```
>>> f = lambda x: x**2          # fonction à primitiver
>>> x = np.linspace(0,1,100)    # subdivision régulière de [0,1]
>>> y = f(x)                    # échantillonage de f
>>> y1 = integrate.cumtrapz(y, x, initial=0)     # intégration
>>> # le résultat retourné est l'échantillonnage d'une primitive
>>> # sans l'option : initial = 0 la première valeur serait
>>> # manquante, il n'y aurait que 99 valeurs, ce qui poserait
>>> # problème pour le tracé. Avec initial = 0 on pose y1[0]=0
```

Le résultat est le même qu'avec l'algorithme (primitivation d'un échantillonnage par la méthode des trapèzes)

Primitivation numérique sous scipy : cumtrapz()

- Maintenant son tracé à l'aide de pyplot :

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(x,y,'b',x,y1,'r')
>>> plt.legend(('f:x -> x^2','primitive de f'), 'upper left')
>>> plt.show()
```



Primitivation numérique sous scipy : cumtrapz()

Attention : La primitive retournée est celle valant 0 au premier point de l'échantillonnage.

(Rappelons que toute fonction continue sur un intervalle admet une infinité de primitives, toutes différant d'une constante additive).

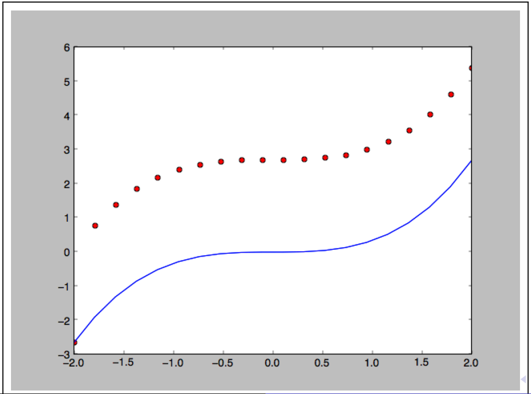
```
>>> x = np.linspace(-2,2,20) ; y = f(x)
>>> yf = integrate.cumtrapz(y,x,initial = 0)
>>> plt.plot(x,yf,'ro',x,x**3/3,'b-') ; plt.show()
```

Primitivation numérique sous scipy : cumtrapz()

Attention : La primitive retournée est celle valant 0 au premier point de l'échantillonnage.

(Rappelons que toute fonction continue sur un intervalle admet une infinité de primitives, toutes différant d'une constante additive).

```
>>> x = np.linspace(-2,2,20) ; y = f(x)
>>> yf = integrate.cumtrapz(y,x,initial = -2**3/3)     # en changeant 'initial'
>>> plt.plot(x,yf,'ro',x,x**3/3,'b-') ; plt.show()
```

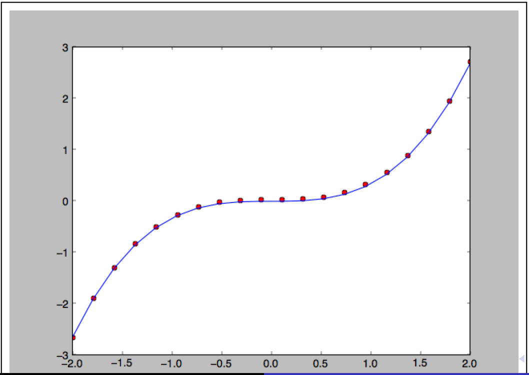


Primitivation numérique sous scipy : cumtrapz()

Attention : La primitive retournée est celle valant 0 au premier point de l'échantillonnage.

(Rappelons que toute fonction continue sur un intervalle admet une infinité de primitives, toutes différant d'une constante additive).

```
>>> x = np.linspace(-2,2,20) ; y = f(x)
>>> yf = integrate.cumtrapz(y,x,initial = 0)
>>> plt.plot(x,yf - 2**3/3,'ro',x,x**3/3,'b-') ; plt.show() # changement de primitive
```

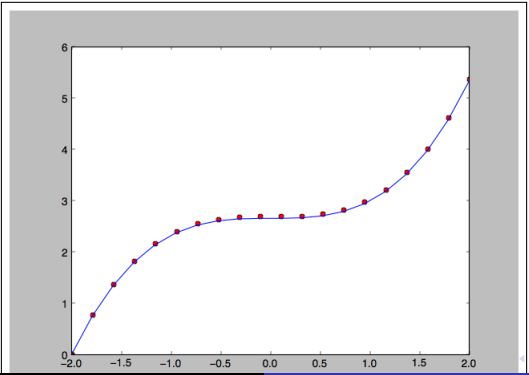


Primitivation numérique sous scipy : cumtrapz()

Attention : La primitive retournée est celle valant 0 au premier point de l'échantillonnage.

(Rappelons que toute fonction continue sur un intervalle admet une infinité de primitives, toutes différant d'une constante additive).

```
>>> x = np.linspace(-2,2,20) ; y = f(x)
>>> yf = integrate.cumtrapz(y,x,initial = 0)
>>> plt.plot(x,yf,'ro',x,x**3/3 + 2**3/3,'b-') ; plt.show() # en gardant la
primitive
```



Méthode de Romberg

• La méthode des trapèzes est simple à expliquer, à implémenter, et relativement assez efficace : avec une subdivision régulière en n segments de l'intervalle d'intégration, l'erreur produite est un $O(1/n^2)$.

• On peut lui appliquer des procédés d'*accélération de convergence*, comme le *procédé de Richardson* pour la promouvoir en une méthode extrêmement efficace. C'est la *méthode de Romberg*, un procédé récursif basé sur la méthode des trapèzes et le procédé d'accélération de Richardson.

• Admettons le résultat théorique suivant concernant la **Méthode des trapèzes** :

Théorème. *Si f est de classe C^{2k} sur l'intervalle $[a,b]$ et si $T(h)$ désigne l'approximation de $I = \int_a^b f(t)dt$ obtenue par la méthode des trapèzes pour un pas valant $h = \frac{b-a}{n}$ (c'est à dire pour une subdivision régulière en n segments (c.à d. $n+1$ points)), alors on a le développement limité suivant de $T(h)$ en 0 :*

$$T(h) = I + a_1 h^2 + a_2 h^4 + \cdots + a_{k-1} h^{2(k-1)} + O(h^{2k})$$

où $a_1, a_2, \ldots, a_k$ sont des constantes ne dépendant que de f, de a et de b.

• Alors, tout d'abord, on en déduit que la méthode des trapèze produit une erreur en $O(h^2)$ c'est à dire en $O(1/n^2)$ (ou encore en $o(h)$, soit en $o(1/n)$).

Méthode de Romberg

• Mais aussi nous allons en déduire un procédé d'accélération de convergence (de Richardson) :

Si :

$$T(h) = I + a_1 h^2 + a_2 h^4 + \cdots + a_{k-1} h^{2(k-1)} + O(h^{2k})$$

Alors :

$$T\left(\frac{h}{2}\right) = I + \frac{a_1}{4} h^2 + \frac{a_2}{16} h^4 + \cdots + \frac{a_{k-1}}{4^{k-1}} h^{2(k-1)} + O(h^{2k})$$

Mais on en déduit alors, en formant une combinaison linéaire pour supprimer le terme en h^2 :

$$\begin{aligned} 4T\left(\frac{h}{2}\right) - T(h) &= 3I - \frac{3a_2}{4} h^4 + O(h^4) \\ &= 3I + O(h^4) \end{aligned}$$

Ainsi :

$$\boxed{\frac{4T\left(\frac{h}{2}\right) - T(h)}{3} = I + O(h^4)}$$

est une approximation de l'intégrale $I = \int_a^b f(t)dt$ en $O(h^4)$... C'est à dire dont la convergence vers I lorsque h vers 0, (c'est à dire lorsque n, le nombre de subdivisions, augmente) est beaucoup plus rapide que celle obtenue par la méthode des trapèzes.

Méthode de Romberg

En poursuivant ce procédé, par récurrence, on obtient la méthode de Romberg :

On construit par récurrence un tableau $n \times n$: L'élément ligne i colonne j est noté $r(i,j)$ (prenons $(i,j) \in [[0, n-1]]^2$ pour plus de lisibilité). Alors :

$r(0,j)$ est le résultat d'intégration obtenu par la méthode des trapèzes avec un pas $h = \dfrac{b-a}{2^j}$.

Chaque élément d'une ligne s'obtient à partir des 2 éléments en haut et en haut à gauche de la ligne précédente (un peu 'à la triangle de Pascal'), grâce à la relation de récurrence :

$$\boxed{r(i+1,j) = \frac{4^{i+1} r(i,j) - r(i,j-1)}{4^{i+1} - 1}} \qquad \text{pour } i \in [[0, n-1]] \text{ et } j \in [[i+1, n]]$$

Subdivision en 2^0 segment	Subdivision en 2^1 segments	Subdivision en 2^2 segments	Subdivision en 2^3 segments
$r(0,0)$	$r(0,1) = T(\frac{b-a}{2})$	$r(0,2) = T(\frac{b-a}{4})$	$r(0,3) = T(\frac{b-a}{8})$
	$r(1,1) = \dfrac{4.r(0,0) - r(0,1)}{3}$	$r(1,2) = \dfrac{4.r(0,1) - r(0,2)}{3}$	$r(1,3) = \dfrac{4.r(0,2) - r(0,3)}{3}$
		$r(2,2) = \dfrac{4^2.r(1,1) - r(1,2)}{15}$	$r(2,3) = \dfrac{4^2.r(1,2) - r(1,3)}{15}$
			$r(3,3) = \dfrac{4^3.r(2,2) - r(2,3)}{63}$

En bas à droite : le résultat obtenu.

Méthode de Romberg : implémentation en python

- Implémentation en python :

On utilise la fonction trapeze() (attention : avec subdivision en n segments) :

```
import numpy as np

def trapeze(f,a,b,n):        # Intégration approchée par la méthode des trapèzes
    if a > b:
        return -trapeze(f,b,a)
    X = np.linspace(a,b,n+1)
    Y = f(X)
    return (b-a)/n * (sum(Y) - f(a)/2 - f(b)/2)
```

```
def romberg(f,a,b,n):        # Intégration approchée par la méthode de Romberg
    r = [[0 for j in range(n)] for i in range(n)]     # matrice (n,n) de zéros
    for j in range(n):    # Remplissage de la première ligne
        r[0][j] = trapeze(f,a,b,2**j)
    for i in range(1,n):    # Remplissage des lignes suivantes
        for j in range(i,n): # Remplissage de la i-ème ligne
            r[i][j] = (4**i * r[i-1][j] - r[i-1][j-1]) / (4**i-1)
    return r[n-1][n-1]    # Le résultat retourné est celui bas à droite du tableau
```

Méthode de Romberg : implémentation en scipy

• Sous scipy l'intégration approchée d'une fonction f sur un intervalle $[a, b]$ peut s'effectuer à l'aide de la méthode de Romberg, en utilisant la méthode `romberg(f,a,b)` du sous-module `integrate` de scipy :

```
>>> from scipy import integrate
>>> f = lambda x: x**2
>>> integrate.romberg(f,0,1)
0.33333333333333331
```

• On peut utiliser l'option `show = True` pour montrer les résultats intermédiaires (le tableau) de la méthode de Romberg (par défaut `show = False`) :

```
>>> integrate.romberg(f,0,1,show = True)
Romberg integration of <function vfunc at 0x1033d2050> from [0, 1]

Steps StepSize Results
    1 1.000000 0.500000
    2 0.500000 0.375000 0.333333
    4 0.250000 0.343750 0.333333 0.333333

The final result is 0.333333333333 after 5 function evaluations.
0.33333333333333331
```

(attention le 'tableau' est transposé : intervertir lignes/colonnes).

Intégration numérique sous scipy

- scipy dispose d'autres méthodes d'intégration de fonctions :

1. La méthode quad(f,a,b) pour intégrer f sur l'intervalle $[a, b]$:

```
>>> f = lambda x: x**2
>>> integrate.quad(f,0,1)
(0.33333333333333337, 3.700743415417189e-15)
```

Elle retourne un couple constitué de la valeur approchée de l'intégrale (1er élément) et d'une estimation de l'erreur commise. Pour accéder indépendamment à ces 2 valeurs :

```
>>> res = integrate.quad(f,0,1)
>>> res[0]     # Résultat obtenu
0.33333333333333337
>>> res[1]     # Estimation de l'erreur
3.700743415417189e-15
```

2. La méthode quadrature(f,a,b) pour intégrer f sur l'intervalle $[a, b]$ (par la méthode de quadrature gaussienne).

```
>>> integrate.quadrature(f,0,1)
(0.33333333333333315, 1.1102230246251565e-16)
```

La deuxième valeur retournée est la différence obtenue entre les résultats de deux dernières itérations (la méthode est récursive et s'arrête dès que cette valeur est inférieure à une borne donnée tol=1.48e-08 que l'on peut modifier en paramètre optionnel).

3. D'autres méthodes, par exemple pour le calcul d'intégrales doubles ou multiples sont disponibles dans scipy :

voir : http://docs.scipy.org/doc/scipy/reference/integrate.html.

Intégration à partir d'un échantillonnage

• Les méthodes suivantes ne s'appliquent pas à une fonction et à un intervalle, mais seulement à un échantillonnage (régulier) de la fonction sur l'intervalle d'intégration :

1. `trapz()` applique la méthode des trapèzes à un échantillonnage :

```
>>> import numpy as np
>>> x = np.linspace(0,1,100)
>>> y = x ** 2
>>> integrate.trapz(y, dx=0.01)
0.33001683501683515
```

   Attention à bien passer en argument le pas `dx` ($= h = \frac{b-a}{n}$) car par défaut `dx = 1` ce qui aboutirait à un résultat erroné.

2. `simps()` applique la méthode de Simpson (cf. TD 7) à un échantillonnage :

```
>>> integrate.simps(y, dx=0.01)
0.33000017005067522
```

3. Si l'échantillonnage est régulièrement espacé et comporte $2^k + 1$ points pour un entier k, alors la méthode `romb()` applique la méthode de Romberg à un échantillonnage :

```
>>> x = np.linspace(0,1,257)    # 257 = 2^8 +1
>>> y=x**2
>>> integrate.romb(y, dx=1/256)
0.33333333333333331
```

   Attention à bien spécifier l'option : $dx = 2^k$ (=valeur du pas h), qui vaudrait sinon 1 par défaut et aboutirait à un résultat erroné :

```
>>> integrate.romb(y)    85.333333333333329
```