**Université Claude Bernard - Lyon 1** Semestre d'Automne
L3 Mathématiques pour l'enseignement
Géométrie et Arithmétique avec des logiciels

---

**TP : Introduction à Python**

---

## Pour commencer - utilisation d'un Notebook

Python est un langage de programmation interprété haut niveau. De nombreux packages sont disponibles pour enrichir ses fonctionnalités. **Pour ces TP, il n'y a pas d'installation à effectuer (que ce soit sur les machines des salles de TP ou sur votre propre machine)** [1].

Nous allons utiliser la plateforme *Jupyter* de l'Université Lyon 1 et le *notebook* associé.

`https://jupyter.univ-lyon1.fr/` (serveur L3)

1. Rendez vous sur cette plateforme.
2. Choisissez "Serveur L3"
3. Connectez vous avec vos login et mot de passe de l'université.
4. Sélectionnez le type de serveur "Notebook classique" puis cliquez sur "Start".
5. Cliquez sur *Nouveau –> Python 3*
6. Renommer votre fichier en cliquant en haut sur "Untitled", par exemple « Prise en main ».

### Notebook

Utiliser cet outil permet de combiner texte et code. La mise en forme se fait à l'aide de *cellules*.
Chaque cellule s'exécute avec la commande ▶ `Exécuter`, ou en utilisant "Maj+Entrée".

7. Exécuter la commande `print('Hello world')` dans une cellule.

La sauvegarde se fait sous la forme d'un fichier avec l'extension `ipynb`, on peut aussi exporter sous forme de script Python (extension `.py`), ou de fichier pdf par exemple.

Pour utiliser l'aide (par exemple pour la fonction print), on peut écrire `help(print)` ou `? print` dans une cellule, puis l'exécuter.

8. Exécuter la commande `help(round)`, puis `? range`.

## Conseils généraux de programmation

Il est important de bien structurer son code et de suivre quelques bonnes pratiques afin qu'il soit :

(1) facilement compréhensible pour une tierce personne (ou pour soi-même si on le relit longtemps après)
(2) facile à "maintenir".

Avec ces objectifs en tête :

- On évitera autant possible la duplication de code (et les copiés / collés) : risque de duplication de bug, dur à maintenir dès qu'on fait un changement, code long. Si vous devez répéter des instructions, créez une fonction et appelez cette fonction.

1. Si vous souhaitez installer Python sur votre propre machine, une implémentation peut être téléchargée avec Miniconda3 par exemple `https://conda.io/miniconda.html`. Une fois Python installé, il y a plusieurs possibilités :
   — utilisation d'un *Notebook*,
   — utilisation d'un *IDE* (environnement de développement), par exemple *Pyzo* ou *Spyder*.

- Essayez de garder des fonctions aussi "élémentaires" que possible : une fonction = UNE fonctionnalité (préférez avoir plusieurs fonctions effectuant chacune une tâche élémentaire plutôt qu'une grosse fonction faisant tout en même temps).
- Donnez des noms (pour les variables, les fonctions...) explicites et faciles à comprendre.
- Commentez votre code en gardant en tête qu'une tierce personne est censée pouvoir comprendre votre code. Commentaire sur une ligne : utilisez le signe # avant votre commentaire. Commentaire multi-lignes : utilisez un triple guillemets " " " en début et en fin de commentaire.
- Gardez les commentaires à jour quand vous modifiez le code ! (pour éviter que les commentaires ne contredisent le code).

## Ressources et package

**Ressources :**

— Site OpenClassrooms pour des tutos Python (gratuits) : OpenClassrooms

— Règles et des bonnes pratiques qui aident à standardiser l'écriture en Python (en anglais) : PEP8

— Bonne pratique d'écriture en Python (en anglais) : PEP20

— Traductions des bonnes pratiques en français : The Hitchhiker's Guide to Python

**Modules et packages :**

Un module est un fichier contenant des fonctions et des définitions qui ne sont pas inclus par défaut. On pourra alors les utiliser dans notre code (par exemple la fonction exponentielle, la constante $\pi$ etc.).

Un package est une collection de modules. Pour importer un package nommé "nom_package", il faut écrire la commande

`import` *nom_package*,

par exemple `import` *math*. Pour utiliser une fonction ou un attribut (constante,...) d'un package, on écrira alors

*nom_package.nom_fonction* ou *nom_package.nom_constant*,

par exemple `math.exp` est la fonction exponentielle et `math.e` est la constante $e$.

On peut aussi utiliser la commande `from math import e` pour éviter d'écrire `math.e` (on écrira alors simplement `e`). Il est cependant parfois utilise de savoir de quel package viennent nos fonctions / constantes.

Notez que vous pouvez aussi importer vos propres fichiers (si par exemple vous avez défini une fonction dans un fichier externe `mon_fichier.py` et que vous voulez l'utiliser). Pour cela utiliser la même syntaxe

`import` *mon_fichier.py*

après vous être assuré que *autre_fichier.py* est dans le même répertoire que votre fichier actuel.

On peut renommer un package (ou même les fonctions ou constantes d'un package) de la manière suivante :

`import` *nom_package* `as` *nouveau_nom_package*.

Par exemple, `import` *numpy* `as` *np* ou `from` *math* `import` *e* `as` *constante_e*. Pour appeler les fonctions du package `numpy` il suffira alors d'écrire `np.`*nom_fonction* au lieu de `numpy.`*nom_fonction*. Attention, ne renommez pas vos packages si cela rend votre code moins lisible.

On sera amené à utiliser les package suivants :

— `math`.

— `numpy`.

— `matplotlib` (pour tracer des fonctions etc.). On l'utilise souvent avec l'interface `pyplot`. Tapez la commande `from matplotlib import pyplot`.

— Ne pas hésiter à consulter l'aide (ou l'aide en ligne).

— Bien respecter la syntaxe, la casse (majuscules/minuscules), et les espaces (en particulier pour les boucles et les fonctions).

## Premiers pas en Python

### Variables

— Une variable pour Python est un ensemble de trois choses : un nom, une valeur et un type (voir après).

— Pour déclarer ou changer la valeur d'une variable, il suffit d'écrire :

```
nom_variable = valeur_variable
```

Exemple : `a = 2` va créer une variable appelé `a` et lui attribuer la valeur 2 (de type *Entier*).

— Pour les noms de variable, vous ne pouvez pas commencer par un chiffre. Choisissez des noms explicites et évitez les abréviations.

— Les types de données les plus simples (ou primitifs) de Python sont :

1. les entiers (Integers en anglais). Exemple : -1,0,2,...
2. les virgules flottantes (Float en anglais). Exemple : 3.5, 10.0,...
3. les chaînes de caractères (Strings en anglais). Exemple : "poisson", 'chat', ... (remarquez qu'on peut utiliser des guillemets simples ou doubles...).
4. les booléens (True ou False).

On peut avoir des types de données plus complexes comme les listes, les dictionnaires etc.

### Listes

Création d'une liste : `ma_liste = ['pomme', 'banane', 'orange']`
Afficher ma liste : `print(ma_liste)`

Les listes sont indexées à 0. Par exemple l'instruction `ma_liste[0]` renvoie `pomme`.

- Ajouter un élément à la fin d'une liste : `ma_liste.append('element_ajout')`
- Supprimer un élément d'une liste : `ma_liste.remove('element_suppr')`
  *Remarque* : si ma liste comporte plusieurs fois l'élément `'element_suppr'`, la fonction `remove` supprimer uniquement le premier élément portant ce nom (en partant de la gauche).
- Longueur d'une liste : `ma_liste.len()`
- Trier ma liste (ordre alphabétique ou numérique suivant le type de données) : `ma_liste.sort()`

### Dictionnaires

Création d'un dictionnaire : `mon_dico = {'Brassens':'Georges','Brel':'Jacques','Reggiani':'Serge'}`
Afficher mon dictionnaire : `print(mon_dico)`
Afficher la valeur associé à la "clef" *Brassens* : `mon_dico['Brassens']`.

Un dictionnaire est une sorte de liste dont on peut choisir les "clefs" (dans l'exemple ci-dessus : le nom de famille). Sa gestion est aussi différente que celle des listes.

- Ajouter un élément de clef `key` dans mon dictionnaire : `mon_dico['key'] = 'nouvelle_valeur'` *Remarque* Si `key` existait déjà, on écrase l'ancienne valeur en la remplaçant par `nouvelle_valeur`.
- Supprimer l'élément de clef `'clef_a_suppr'` d'un dictionnaire : `del mon_dico['clef_a_suppr']`
- Vérifier que la clef `'clef_exist'` existe dans mon dictionnaire : `in 'clef_exist' mon_dico` (renvoie un booléen `True` ou `False`).

### Opérateurs de comparaison et structure du "if"

Pour comparer deux objets $a$ et $b$ (par exemple deux nombres réels), on a les instructions suivantes :

- égal à : `a==b`
- différent de : `a!=b`
- strictement inférieur (resp. supérieur) à : `a < b` (resp. `a > b`)
- inférieur (resp. supérieur) ou égal à : `a <= b` (resp. `a >= b`)

On utilisera souvent ces opérateurs dans les conditions de la structure `if` :

```
if (condition 1):
   bloc d'instructions 1
elif (condition 2):
      bloc d'instructions 2
⋮
else:
   bloc final d'instructions
```

*Traduction :*

- On commence par tester la condition `condition 1`.
- Si elle est satisfaite on exécute le bloc d'instructions `bloc d'instructions 1` et on saute tout le reste.
- Si elle n'est pas satisfaite, on ne fait rien et on teste `condition 2` du deuxième bloc (`elif` = "else if", qui signifie "sinon, si ...").
- Si `condition 2` est satisfaite, on effectue les instructions du 2ème bloc puis on saute tout le reste.
- On procède ainsi jusqu'à la fin. Le dernier bloc `else` (qui signifie "sinon") et à effectuer si aucune des précédentes conditions n'a été satisfaite.

*Remarque :*

- On peut utiliser les opérateurs `and`, `or` et `not` pour construire des conditions un peu plus complexes.
- Notez l'importance des alinéas (4 espaces pour Python). Ce sont ces espaces qui permettent à Python de délimiter le début et la fin d'un bloc. Ils sont **indispensables** et contribuent par ailleurs à la lisibilité du code.

### Boucles for et boucles while

Il est souvent nécessaire d'effectuer en boucle un certain nombre d'opérations (traitement de données,...). On dispose de deux structures pour cela.

**Boucle `for`**

```
liste = [2,3,4,5,6,7]
for j in liste:
   bloc d'instructions (qui peut dépendre de la valeur de j)
```

*Traduction :*

- On commence avec $j = 2$ (premier élément de `liste`). On exécute le bloc d'instructions.
- On passe à $j = 3$ (élément suivant de `liste`). On effectue de nouveau le bloc d'instructions.
- On continue ainsi de suite. La variable $j$ parcourt `liste`. On s'arrête lorsque $j$ atteint le dernier élément de `liste` (ici $j = 7$) et qu'on a exécuté le dernier bloc d'instructions.

Grâce à l'instruction `range(n)` on peut parcourir facilement les entiers compris entre 0 et $n-1$. Exemple :

```
for j in range(10):
   print(j)
```

Cela affiche 0, 1, ..., 9.

**Boucle `while`**

```
while (condition):
   bloc d'instructions
```

La boucle `while` (signifie "tant que") vérifie si `condition` est satisfaite ou non à chaque étape. Tant que la condition est satisfaite, on exécute le bloc d'instructions.

**Attention aux boucles infinies !** Assurez-vous que `condition` devient fausse en un nombre fini d'étapes pour éviter une boucle infinie.

**Exemple de boucle finie**

```
j=0
while j < 100:
   j = j+1
   print('Comme j'augmente j à chaque étape, au bout d'un moment j>=100')
```

**Exemple de boucle infinie**

```
j=0
while j < 100:
   print('j est toujours égal à 0 puisque que je ne l'augmente pas...')
```

### Définir ses propres fonctions

Ils existent un certain nombre de fonctions intégrées (ou provenant de packages). Il est aussi souvent utile de définir ses propres fonctions. Pour définir une fonction appelée `nom_de_ma_fonction` prenant en argument des paramètre $x$, $y$, ... et renvoyant $R = f(x, y, \dots)$, on procédera ainsi :

```
def nom_de_ma_fonction(x, y,...):
   blabla (blocs d'instructions)
   return R
```

**Pour tous les exercices, écrire les commandes dans des les parties adéquates du *Notebook* fourni et observer les réponses afin d'identifier l'utilisation de chacune des commandes.**

**Exercice 1 - Un peu de Python**

Effectuer des opérations :

```
2+5
2345/34578

2**3
5**(1/3)
```

Définition de variables :

```
a = 10
b = a + 5
b

a, b, c = 1, 2, 3
print(a, b, c)

a, b = b, a
print(a, b)

s = "Hello !"
print(s)
```

Les listes : il y aurait beaucoup à dire dessus, voici quelques opérations que l'on sera amené à utiliser plus tard.

```
c = [1, -2, 7, 0, 10]
d = [3, 4]
c + d
2 * c

10 * c[0]

print(c[0], c[1])
print(c[-1], c[-2])

# créer l'intervalle des entiers entre
# 0 et 9
e = range(10)
print(e)

# pour en créer la liste
e = list(e)
print(e)

e = range(5, 11)
print(list(e))

e = [i**2 for i in range(11)]
print(e)
```

Une fonction Python se déclare comme suit :

```
def nom_de_fonction(input1, input2, ...):
   instructions
   return output1, output2, ...
```

où les `input` sont les arguments d'entrée de la fonction et les `output` les argument de sortie. Ni les uns ni les autres ne sont obligatoires.

**Attention !** Les indentations sont très importantes en Python. C'est ce qui définit si les instructions font partie de la fonction ou du script. En effet, il n'y a rien qui indique la fin de la fonction à part les indentations !

Par exemple :

```
def f(x):
    x=x+1
    x=x*x
    return x
f(7)
```

Les fonctions mathématiques usuelles sont accessibles dans le package `math` :

```
sqrt(2)
import math
math.sqrt(2), math.pi, math.log(5)
```

Nous allons maintenant voir comment effectuer des tests et des boucles.

— Les tests avec le mot-clé `if`. La syntaxe est la suivante :

```
if une_condition:
  instructions
elif une_autre_condition:
  instructions
else:
  instructions
```

Remarques :

— Les symboles `:` après le `if`, `elif` et `else` sont obligatoires. Ils marquent le début des blocs.

— Les indentations sont importantes ! Elles définissent si les instructions font partie des blocs ou non. Il n'y a, en effet, pas de mot-clé pour marquer la fin des blocs en Python.

— Les mot-clés `elif` et `else` ne sont pas obligatoires.

Par exemple :

```
def f(n):
    if n <0:
        return(-n)
        print('Nombre strictement négatif')
    elif n == 0 or n == 1:
        n=1-n
        print('Zéro ou Un')
    else:
        n=math.factorial(n)
        print('Strictement plus grand que Un'
            )
    return(n)
```

Tester `f` sur différentes valeurs. Par exemple $-5$, 1, 3, 1.5.

À votre tour ! Coder la fonction

$$g : \mathbb{R} \to \mathbb{R}, \quad x \mapsto \begin{cases} x \sin x & \text{si } x < 0 \\ \ln x & \text{si } x > 0 \\ 0 & \text{si } x = 0 \end{cases}$$

— Les boucles `for`. La syntaxe est :

```
for variable_boucle in une_variable:
    instructions
```

Remarques :

— la variable `variable_boucle` va, une par une, prendre les valeurs dans la variable `une_variable`. Pour chacune de ces valeurs, les instructions dans le bloc `for` seront exécutées.

— `une_variable` peut être, par exemple, une liste Python ou un tableau (on le verra plus tard).

— Comme pour les `if`, les `:` et les tabulations sont nécessaires.

Par exemple :

```
for i in range(5):
    print(i*i)
```

— Les boucles `while`. La syntaxe est :

```
while condition:
    instructions
```

Les instructions dans le bloc `while` sont exécutées tant que `condition` est vraie. Exemple (suite de Fibonacci) :

```
a, b = 0, 1
while b < 1000:
    a, b = b, a + b
    print(round(b/a, 3), end=", ")
```

### Exercice 2 - Le calcul avec Numpy et les tableaux

Le package **Numpy** est **LE package de référence** pour le calcul numérique en Python. Il doit être importé avec la commande `import numpy`.

Premiers exemples de tableaux :

```
# Les listes Python
c = [1, -2, 7, 0, 10]
2 * c
```

```
# Avec un tableau Numpy
import numpy as np
c = np.array([1, -2, 7, 0, 10])
2 * c
print(2 * c)

c.size
np.size(c)

c[3]
c[0] + c[1] + c[2] + c[3] + c[4]
c[0] = 0
print(c)

d = np.array([[0, 0, 0], [0, 0, 0]])
print(d)

d = np.zeros((2, 3))
print(d)
d.shape
np.shape(d)

a = np.array([[0, 1], [0, 0]])
print(a)
a[1, 1] = 1
a[0, 0] = a[1, 1]
print(a)
```

D'autres opérations sur les tableaux numpy. Bien observer le résultat de chaque commande pour comprendre son fonctionnement.

```
print(np.arange(0,10))
print(np.arange(0,10,0.5))


print(np.linspace(0, 10, 10))
print(np.linspace(0, 10, 11))

print(np.linspace(0, 10, 21))

print(np.linspace(0,2*np.pi,7))

x = np.array([2,-1,np.pi,-3,-2,-np.e])
print(x)
print(np.max(x))
print(np.min(x))

x = np.arange(0,20)
print(x)
print(x.size)
print(x[2:17])
print(x[2])
print(x[17])

print(x[:-2])
print(x[3:])
print(x[9:-8])
```

Les dernières commandes montrent qu'il est possible d'extraire un sous-tableau d'un tableau plus grand. Pour cela, il suffit de taper : `tableau[début:fin+1]`.

**Attention à la copie de tableaux** :

```
y=x
y[0]=-1
print(x)

z=y.copy()
z[0]=0
print(y)
```

**Fonctions usuelles**

Ces fonctions existent à la fois dans le package math et dans le package Numpy. Comme on utilisera essentiellement des tableaux Numpy, il est préférable d'utiliser les versions Numpy.

```
np.abs(-2)
np.sqrt(3)/2
np.exp(2)
np.log(np.e)
np.sin(np.pi/6)
np.sin(np.pi/3)
```

Ces fonctions marchent aussi sur des tableaux en opérant élément par élément.

```
x= np.linspace(0,np.pi,7)
y = np.sin(x)
print(np.round(y,3))
```

### Exercice 3 - Approximations du maximum d'une fonction de classe $C^1$ sur un segment $[a, b]$

Définir une fonction `max_approx` prenant en argument une fonction réelle $f$, les extrémités $a$ et $b$ et un entier $n \geq 1$, et retournant la valeur

$$\max_{0 \leq k \leq n} f\left(a + k\frac{(b-a)}{n}\right).$$

```
def max_approx(f,a,b,n):
    # compléter - utiliser la commande
       linspace
    return # compléter
```

Test avec la fonction sinus sur $[0,2]$ :

```
print([max_approx(np.sin,0,2,10**i) for i in
    range(2,7)])
```

**1)** Pourquoi la fonction sinus admet-elle un maximum sur $[0,2]$ ?

**2)** Que vaut-il ? Où est-il atteint ?

Fixons un entier $n \geq 1$. Notons $p$ l'unique entier tel que $\pi/2 \in [2p/n, 2(p+1)/n[$.

**3)** Montrer que

$$\max_{0\leq k\leq n} \sin\left(\frac{2k}{n}\right) = \max\left\{\sin\left(\frac{2p}{n}\right), \sin\left(\frac{2(p+1)}{n}\right)\right\}.$$

**4)** Expliquer pourquoi $\pi/2$ ne peut être le milieu du segment $[2p/n, 2(p+1)/n]$. On notera $x_n$ l'extrémité la plus proche de $\pi/2$.

**5)** En utilisant le fait que : $\forall x \in \mathbb{R}$, $\sin\left(\frac{\pi}{2}+x\right) = \sin\left(\frac{\pi}{2}-x\right)$, montrer que $\max_{0\leq k\leq n} \sin\left(\frac{2k}{n}\right)$ est atteint en $x_n$.

**6)** Montrer enfin que

$$\left|1 - \max_{0\leq k\leq n} \sin\left(\frac{2k}{n}\right)\right| \leq \frac{1}{n}.$$

Observer ce que retourne la commande suivante :

```
print([(1-max_approx(np.sin,0,2,10**i))
    *(10**i) for i in range(2,7)])
```

**7)** Montrer que $\dfrac{\sin(x_n)-1}{x_n - \frac{\pi}{2}} \xrightarrow[n\to\infty]{} 0$.

**8)** En déduire que $n\left(1 - \max_{0\leq k\leq n} \sin\left(\frac{2k}{n}\right)\right) \xrightarrow[n\to\infty]{} 0$.

Coder la fonction $f : \mathbb{R} \to \mathbb{R},\ x \mapsto |x(x-1)|$.

**9)** Qu'observe-t-on en utilisant la fonction `max_approx` sur $f$ entre $[0,1]$ selon la parité de $n$ ?

**10)** Déterminer théoriquement le maximum de $f$ sur $[0,1]$.

Coder la fonction $g : \mathbb{R} \to \mathbb{R},\ x \mapsto |(x+1)x(x-1)|$.

**11)** Déterminer théoriquement le maximum de $g$ sur $[-1,1]$, puis vérifier votre résultat à l'aide de la fonction `max_approx`.

### Exercice 4 - Tracer des graphiques

Il existe beaucoup de packages en Python pour tracer des graphiques. Le plus utilisé est `matplotlib`.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2*np.pi, 100)
plt.plot(x, np.sin(x))
plt.show()

plt.plot(np.cos(x), np.sin(x))
plt.show()

plt.plot(x, np.cos(x), x, np.sin(x))
plt.show()

plt.plot(x, np.cos(x))
plt.plot(x, np.sin(x))
plt.show()
```

Les graphiques sont tracés uniquement lorsque la commande `show()` est lancée.

Changer dans le code ci-dessus la valeur 100 par 10. Qu'observe-t-on ?

**RÉPONSE :**

Pour afficher plusieurs fenêtres :

```
x = np.linspace(0, 2*np.pi, 100)
y = np.linspace(0., 1., 100)

plt.figure(1)
plt.plot(x, np.sin(x))

plt.figure(2)
plt.plot(y, y * np.sin(y))

plt.show()
```

Premières options de `plot.` :

```
x = np.linspace(0, 4 * np.pi, 30)
plt.plot(x, np.sin(x), 'r*--')
plt.show()
```

Le troisième argument est une option permettant de spécifier la couleur de la courbe (`r` red), la représentation des coordonnées (`*` étoile) et le type du trait (`--` discontinu). Voir l'aide `? plt.plot` pour avoir la liste des options possibles.

On peut également mettre une légende, donner des titres aux axes et à la figure.

```
plt.plot([0, 1], [1, 0], 'b')
plt.plot(0.1, 0.9, 'r+')
plt.xlabel('x')
plt.ylabel('y')
plt.title('joli graphe')
plt.legend(['segment', 'point'])
plt.show()
```

La commande `fill_between` :

```
x =np.linspace(0,7,100)
plt.fill_between(x,np.cos(x),0)
plt.show()

x =np.linspace(0,7,10)
plt.fill_between(x,np.cos(x),0)
plt.show()

plt.fill_between(x,np.cos(x),0,step='pre')
plt.show()
```

**Exercice 5 - Représentations graphiques de méthodes d'intégration numérique - FACULTATIF**

Tester, puis analyser le code suivant

```
import numpy as np
import matplotlib.pyplot as plt

def rep_int(f,a,b,n,methode):
    x = np.linspace(a,b, 100)
    plt.plot(x,f(x),color='red')
    x,h = np.linspace(a,b, n+1, retstep=True)
    if methode=='rectgauche':
        y=f(x)
        titre='à gauche'
    elif methode=='rectdroite':
        y=f(x+h)
        titre='à droite'
    plt.fill_between(x,y,0,step='post',
        facecolor='green',alpha=0.5)
    plt.vlines(x[:-1],0,y[:-1])
    plt.vlines(x[1:],0,y[:-1])
    plt.hlines(y[:-1],x[:-1],x[1:])
    plt.hlines(0,a-h/2,b+h/2)
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title('méthode des rectangles '+titre
        )
    plt.legend(['f','aire'])
    plt.show()

rep_int(np.cos,0,3*np.pi/2,5,'rectgauche')

rep_int(np.cos,0,3*np.pi/2,5,'rectdroite')
```

**On pourra lors de l'étude des méthodes en question (fiche suivante) :**

- compléter la fonction `rep_int` pour représenter la méthode des rectangles au milieu ;
- adapter cette fonction, pour représenter la méthode des trapèzes.