Objectifs : *calculer numériquement la dérivée d'une fonction, en effectuant sa dérivée "discrète" ; utiliser des instructions itératives (boucles for).*
(extrait du programme : par convention, le calcul de la dérivée discrète s'effectue par "différence avant").

NB ***Compte-rendu :** Exercice **D1**, script (fichier *.PY) à déposer sur Éclat :*
► *Espace des classes > Classe ATS > Dossiers partagés > TP info-physique >* ***séance N°2***

1. LES BOUCLES ITÉRATIVES (RAPPEL)

Au cours de ce TP, nous serons amenés à répéter plusieurs fois l'exécution d'une partie d'un programme. Nous devons donc au préalable rappeler la notion de *boucle*.

1.1. Boucles bornées et non bornées

a) boucle bornée
Quand on sait combien de fois doit avoir lieu la répétition, on utilise généralement une boucle `for`.

b) boucle non bornée
Si on ne connait pas à l'avance le nombre de répétitions, on choisit une boucle `while`.

1.2. Boucle `for`

a) syntaxe

```
for compteur in range(début, arrêt):
```

b) exemple d'utilisation

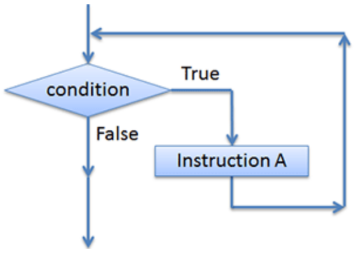
```
for i in range(0,4):
    print("i a pour valeur", i)
```

Après exécution, la console affiche :

```
In []:   runfile('chemin/sanstitre0.py', wdir='chemin')
i a pour valeur 0
i a pour valeur 1
i a pour valeur 2
i a pour valeur 3
```

L'instruction `for` est une instruction composée, c'est-à-dire une instruction dont l'en-tête se termine par deux-points `:`, suivie d'un bloc indenté qui constitue le corps de la boucle.
On dit que l'on réalise une *itération* de la boucle à chaque fois que le corps de la boucle est exécuté.
Dans l'en-tête de la boucle, on précise après le mot-clé `for` le nom d'une variable (`i` dans l'exemple ci-dessus) qui prendra successivement toutes les valeurs qui sont données après le mot-clé `in`. On dit souvent que cette variable (ici `i`) est un compteur car elle sert à numéroter les itérations de la boucle.

1.3. Boucle `while`



a) syntaxe :

```
while condition:
    Instruction A
```

b) exemple

```
x = 1
while x < 10:
    print("x a pour valeur", x)
    x = x * 2
print("Fin")
```

Après exécution, la console affiche :

```
In []:   runfile('chemin/sanstitre1.py', wdir='chemin')
x a pour valeur 1
```

```
x a pour valeur 2
x a pour valeur 4
x a pour valeur 8
Fin
```

Le mot-clé `while` signifie *tant que* en anglais. Le corps de la boucle (c'est-à-dire le bloc d'instructions indentées) sera répété tant que la condition est vraie.
Dans l'exemple ci-dessus, `x` sera multiplié par 2 tant que sa valeur reste inférieure à 10.

Remarque : Si la condition est fausse au départ, le corps de la boucle n'est jamais exécuté. Si la condition reste toujours vraie, alors le corps de la boucle est répété indéfiniment.

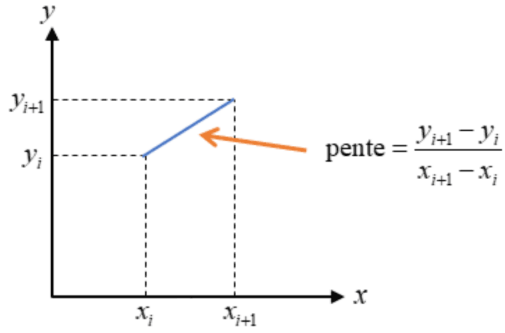
2. DÉRIVATION DISCRÈTE

2.1. Principe

On s'intéresse au calcul de la dérivée d'une fonction dont on connait les valeurs $f(x)$ pour des abscisses x_i.

Une *dérivation numérique* peut se faire simplement en calculant la pente de la courbe.
Nous allons considérer que nous disposons au préalable des valeurs de la fonction en un certain nombre nbx de points de coordonnées (x_i, y_i) où $y_i = f(x_i)$. Le nom de variable nbx est choisi pour signifier *nombre de x*.

Une approximation numérique de la dérivée est obtenue en calculant la pente entre deux points successifs de coordonnées (x_i, y_i) et (x_{i+1}, y_{i+1}).
La pente correspond au coefficient directeur de la droite qui passe par ces deux points (voir figure ci-contre).



En résumé : $dérivée = \frac{dy}{dx}$, $pente = \frac{\Delta y}{\Delta x} = \frac{y_{i+1}-y_i}{x_{i+1}-x_i}$ et quand on fait l'approximation $dérivée \approx pente$, on parle de calcul par *différence finie*.

La question qui se pose alors est de savoir à quel abscisse associer cette pente, autrement dit cette pente représente-t-elle la dérivée en x_i ou en x_{i+1} ? Dans le premier cas, on parle de calcul par *différence finie avant*, dans le deuxième cas on parle de calcul par *différence finie après*. Une troisième méthode, la plus précise mais aussi un peu plus contraignante, consiste à attribuer cette pente à l'abscisse moyenne $\frac{x_i + x_{i+1}}{2}$: on parle alors de calcul par *différence finie centrée*.

Nous nous intéresserons uniquement au calcul par *différence finie avant*, on écrira donc $f'(x_i) \approx \frac{y_{i+1}-y_i}{x_{i+1}-x_i}$

2.2. Exemple : calcul de la dérivée de la fonction cosinus.

a) script

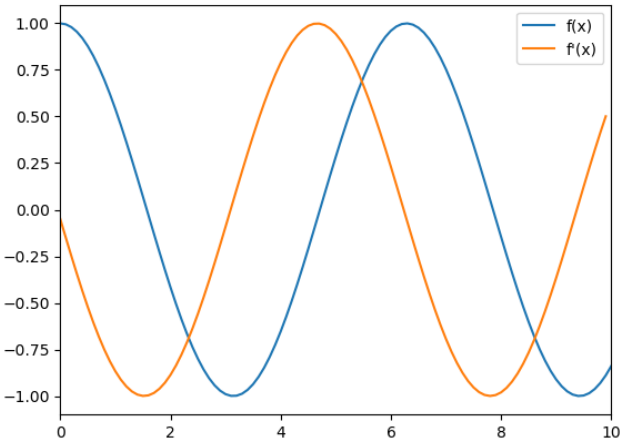
```
#chargement modules
import numpy as np
import matplotlib.pyplot as plt

nbx = 101
x = np.linspace(0, 10, nbx)
y = np.cos(x)

# préparation du tableau qui va recevoir les valeurs
yp = np.zeros(nbx-1)

# calcul des valeurs de la dérivée discrète
for i in range(nbx-1):
    yp[i] = (y[i+1] - y[i]) / (x[i+1] - x[i])
# tracé de la fonction et de sa dérivée discrète
plt.plot(x, y, label='f(x)')
plt.plot(x[0:nbx-1], yp, label='f\'(x)')
```

```
plt.xlim(0, 10)
plt.legend()
plt.show()
```

b) commentaires

- On prend `nbx = 101`, de manière à avoir 100 intervalles.
- La dérivée est notée `yp`, pour "y prime".
- la fonction `np.zeros(n)` renvoie une liste (tableau 1D) de n zéros. Ici, il s'agit de préparer le tableau `yp` à la bonne dimension, qui recevra les 100 valeurs de la dérivée.
- La boucle `for` remplit le tableau.
- `x[i]`, `y[i]` et `yp[i]` désignent les éléments d'indice i des tableaux `x`, `y` et `yp`.
- `x[0:nbx-1]` signifie qu'on prend les 100 premiers éléments du tableau `x`, qui en compte 101.
- Pour écrire $f'(x)$, nous avons le choix entre `"f'(x)"` et `'f\'(x)'` : l'antislash permet de distinguer l'apostrophe des guillemets simples. On aurait aussi pu écrire `'df/fx'`.
- `plt.xlim(0, 10)` est facultatif.

c) graphe obtenu



Cela paraît bien fonctionner, mais on peut tout de même voir comment se manifeste cette méthode de la *différence finie avant*.
Observer en particulier ce qui se passe en $x=0$ et en $x=10$.

d) test de validité de la méthode

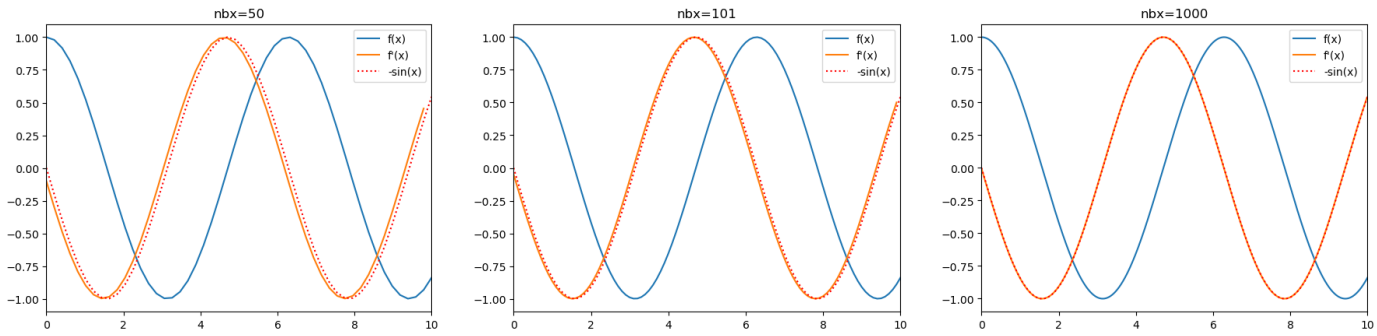
Pour évaluer cette méthode de calcul, et voir quel paramètre permet de l'améliorer, nous allons comparer le résultat avec la vraie dérivée. Nous allons donc superposer au graphe la fonction $-\sin(x)$ pour différentes valeurs de `nbx`.

Nous pouvons ajouter la ligne

```
plt.plot(x, -np.sin(x), label='- sin(x)', linestyle=':', color='red')
```

et tester le code pour différentes valeurs de `nbx` entre 50 et 1000 par exemple. Ligne de code possible pour le titre :

```
plt.title(f'nbx={nbx}')
```



3. APPLICATION À LA RECHERCHE DE L'EXTREMUM D'UNE FONCTION

Soit le problème suivant, déjà traité en TD de physique, concernant la position d'équilibre d'un ressort en extension :
Considérons un ressort vertical, de masse négligeable, maintenu par son extrémité supérieure à un point fixe O. Ses caractéristiques sont : longueur à vide $\ell_V = 0{,}1$ m, raideur $k = 20$ N·m^{-1}. On suspend à ce ressort une masse $m = 0{,}1$ kg.
On peut exprimer l'énergie potentielle de la masse en fonction de sa position x et la mettre sous la forme $Ax^2 + Bx + C$: avec un axe Ox vertical descendant, en prenant $g = 10$ m·s^{-2} et $C = 0{,}1$ J, on avait calculé $A = \frac{k}{2} = 10$ J·m^{-2} et $B = -mg - k\ell_V = -3$ J·m^{-1}.

Exercice D1 (à rendre)

1) Écrire un script permettant de calculer l'énergie potentielle à partir des valeurs `k, m, g` et `lv`.
2) Tracer la fonction `Ep(x)`, pour $x \in [0\ ;\ 0{,}5]$.
3) Calculer la dérivée de cette fonction pour estimer graphiquement la position d'équilibre (valeur nulle de la dérivée).
   NB : on pourra améliorer la lecture en limitant les abscisses à $[0\ ;\ 0{,}5]$ et les ordonnées à $[-1\ ;\ 1]$, et en affichant une grille. On peut également affiner la grille horizontale en ajoutant :
   `plt.xticks(np.arange(0,0.5,0.05))`.
4) Faire varier les différents paramètres, et observer les conséquences. Vérifier en particulier que la valeur de la constante `C` ne change pas la physique du problème.