Méthodes numériques pour la physique

Jean DERVIEUX – jdervieux42@gmail.com

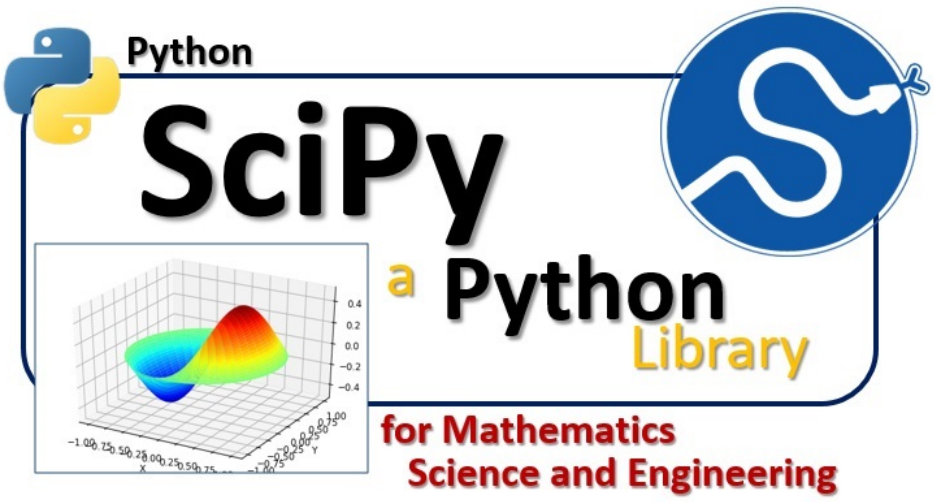


TABLE DES MATIÈRES

CHAPITRE 1

NUMPY ET SES TABLEAUX

1.1 Tableaux numériques (type 'ndarray')

Ces structures sont plus performantes que des listes pour travailler sur des grandes collections de nombres. Elles nécessitent le module NumPy, chargé dans la mémoire de travail par `import numpy as np`.

On gardera 'np' comme alias usuel.

Le type "`ndarray`" signifie "à n dimensions".

```
>>> l = [1, 3, 9, 27] ; type(l) # liste
<class 'list'>

>>> t = np.array(l) # conversion en tableau

>>> t ; type(t) # évaluation et type du tableau
array([ 1,  3,  9, 27])
<class 'numpy.ndarray'>

>>> print(l) ; print(t) # noter la différence !
[1, 3, 9, 27]
[ 1  3  9 27]
```

L'exemple ci-dessus montre un tableau à une dimension ou "**vecteur**"; on utilise fréquemment des tableaux à deux dimensions ou "**matrices**", présentés en lignes et colonnes.

On pratique indexation et slicing habituels.

On peut augmenter les niveaux d'imbrication, mais cela devient plus difficile à gérer.

```
>>> l1 = [1,2,3] ; l2 = [2,4,6] ; l3 = [3,6,9]

>>> t2 = np.array([l1,l2,l3]) ; print(t2)
[[1 2 3]
 [2 4 6]
 [3 6 9]]

>>> t2[0] ; t2[0:2] ; t2[2][2]
array([1, 2, 3])
array([[1, 2, 3],
       [2, 4, 6]])
9
```

Quelques tableaux 1D (vecteurs) utiles :

- La fonction `np.zeros(n)` génère un vecteur contenant n fois la valeur 0.
- La fonction `np.arange(start, stop, step)` génère des valeurs selon les règles de 'range' et renvoie le tableau 1D de ces valeurs.
- La fonction `np.linspace(start, stop, n)` génère n valeurs régulièrement espacées de 'start' à 'stop' et renvoie le tableau 1D de ces valeurs.

NB : dans 'arange', la valeur finale est exclue, alors qu'elle est incluse avec 'linspace' :
`np.arange(2,3,0.1)` ⇝ `array([2. , 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9])`
`np.linspace(2,3,11)` ⇝ `array([2. , 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9, 3.])`

Quelques opérations :

💣 **ATTENTION** : les tableaux numpy "ressemblent" à des listes mais n'en sont pas (par ex. leurs éléments doivent être homogènes) et disposent d'instructions spécifiques.

— La méthode `.append()` n'étant pas supportée, les tableaux doivent être bien dimensionnés à la création. Astuce : on peut créer un tableau de 0 et modifier les valeurs plus tard.

— La fonction **len**(t) donne la taille de la première dimension : nombre de lignes d'une matrice, ou nombre d'éléments d'un vecteur.

— La fonction np.**size**(t) donne le nombre d'éléments individuels (9 pour une matrice 3x3).

— Copie profonde : np.**copy**(t) crée une copie indépendante du tableau 't'.

1.2 Vectorisation

La force des tableaux numpy est de permettre d'agir en une seule commande sur tous les éléments, en particulier pour appliquer une fonction (*mapping*).
Dans la suite nous nous restreindrons au cas unidimensionnel ; un tableau 1D est un **vecteur**.

— L'opérateur + effectue une somme terme à terme entre deux vecteurs de même longueur : `np.array([1,2,3]) + np.array([4,5,6])` ⇝ `np.array([5, 7, 9])`.

— L'opérateur x * t (commutatif) renvoie le tableau obtenu en multipliant chaque élément de 't' par x : `1.5 * np.array([1,2,3])` ⇝ `np.array([1.5, 3., 4.5])`.

— Un opérateur booléen entre deux tableaux de même structure renvoie le tableau dont chaque élément est le résultat de l'opération booléenne entre les éléments correspondants des deux tableaux : `np.array([1,3,5]) < np.array([1,4,5])` ⇝ `array([False, True, False])`.

— La commande f(t) applique la fonction f à tous les éléments de 't' et renvoie le tableau correspondant.
💣 Attention, il faut prendre les fonctions de NumPy pour cela, pas celles du module 'math' qui ne savent pas traiter les tableaux :

```
>>> t = np.array([1, 2, 3, 4])

>>> np.sqrt(t)
array([1.        , 1.41421356, 1.73205081, 2.        ])

>>> math.sqrt(t)
TypeError: only size-1 arrays can be converted to Python scalars
```

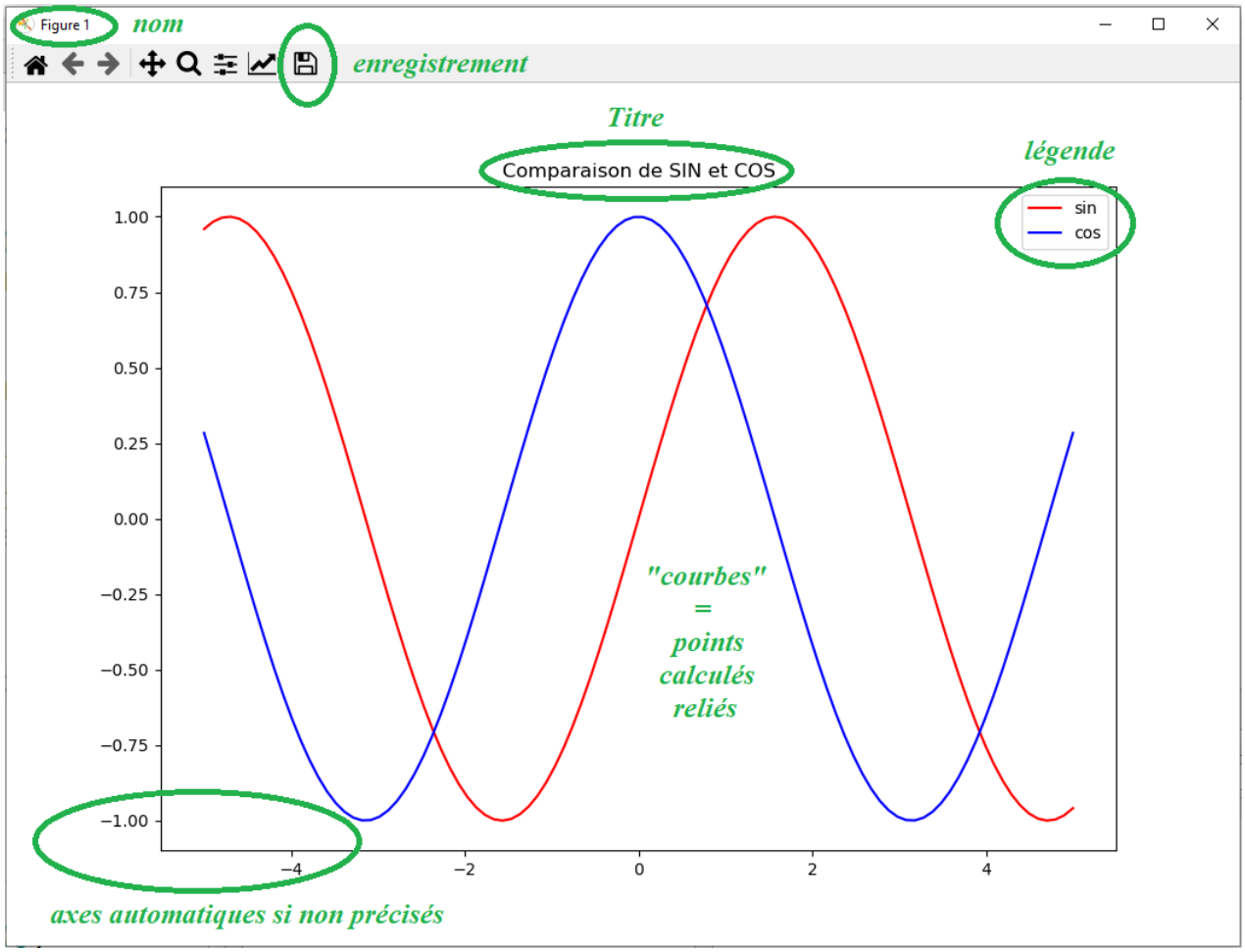
CHAPITRE 2

OUTILS GRAPHIQUES

2.1 La bibliothèque 'matplotlib'

La bibliothèque `matplotlib` est dédiée au tracé de graphes, particulièrement son module `pyplot`.
Le package `numpy` intervient également pour gérer des tableaux utilisés par `matplotlib`.
De façon habituelle : `import numpy as np` et `import matplotlib.pyplot as plt`.

Les graphiques s'affichent dans des fenêtres spéciales, en dehors de l'IDE :



NB :

— L'instruction `plt.figure(...)` sert à ouvrir une fenêtre graphique ; une chaîne de caractères peut être donnée en argument, sinon numérotation automatique.

— L'instruction `plt.close()` sert à fermer une fenêtre graphique (sinon elles s'empilent).

— Les graphiques sont calculés, *puis* affichés sur demande explicite par `plt.show()`.
Tant qu'une fenêtre est ouverte, plusieurs graphiques peuvent s'y superposer, chacun appelé par `plt.plot(...)`.

— L'instruction `np.linspace(début, fin, nb)` est souvent utilisée : elle génère un tableau de 'nb' valeurs régulièrement espacées, de 'début' à 'fin' (valeur incluse).
On peut aussi utiliser `np.arange(start, stop, step)`.

— Dans le cas où 'X' est un tableau de valeurs de type `ndarray`, une fonction f de `numpy` peut lui être appliquée pour donner le tableau des $f(x)$; voir exemple ci-dessous.

Exemple minimal typique :

```
import numpy as np
import matplotlib.pyplot as plt
X = np.linspace(-5,5,100)        # création d'un tableau de valeurs d'abscisses
Y = np.sin(X)                    # création d'un tableau de valeurs d'ordonnées
plt.figure()                     # ouverture de la figure
plt.plot(X,Y)                    # création de l'objet graphique (sans affichage)
plt.show()                       # affichage
```

Rqs :

— `plt.plot()` prend deux arguments : une liste ou un tableau 1D de valeurs d'abscisses et de même pour les ordonnées corerspondantes ;
ex. : `X = [0,1,1,0,0]` et `Y = [0,0,1,1,0]` pour un carré (dernier point = premier).

— par défaut, les points sont reliés par des segments ($\rightsquigarrow$ "courbe " si assez nombreux).

Quelques commandes utiles de pyplot :

```
plt.grid()                                  # afficher une grille
plt.axhline(), plt.axvline()                # axe horizontal, axe vertical
plt.axis("equal")                           # obtenir des axes orthonormés
plt.axis([x_min,x_max,y_min,y_max])         # choisir les valeurs limites
ou plt.xlim(x_min, x_max)
et plt.ylim(y_min, y_max)
plt.xlabel('...') et plt.ylabel('...')      # afficher des étiquettes d'axes
plt.xticks(val), plt.yticks(val)            # imposer des graduations d'axes
où val est un vecteur de flottants ordonnés.
plt.plot(x, y, label="...")                 # pour légender si plusieurs courbes
puis plt.legend()
plt.title('Titre de la figure')             # afficher un titre
plt.text(x,y,'texte',color=...)             # afficher un texte dans le graphique
plt.savefig('ma_courbe.png')                # enregistrer la figure (avec
ou plt.savefig('ma_courbe.pdf')               "Exécuter en tant que script")
```

Quelques options utiles de plt.plot :

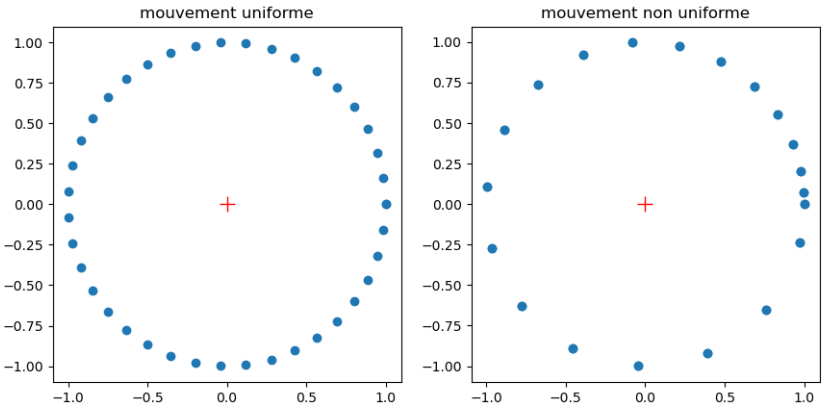
```
linestyle (ou ls) = '-', '--', ':', '-.' ('none' pour ne pas tracer de ligne)
linewidth (ou lw) = 1, 2... (un flottant en nbe de points typographiques)
marker = 'o', '8', 's', 'p', 'h', 'H', 'd', 'D', 'v', '^', '<', '>',
         '*', '+', 'x', '_', '|',  '1',  '2',  '3',  '4'
markersize = 1, 2... (un flottant en nbe de points typographiques)
color = 'k', 'w', 'c', 'm', 'y', 'b', 'g', 'r', RGB(0<R<1,0<G<1,0<B<1)
```

Des raccourcis sont possibles quand on connaît bien ces codes ; ainsi, `plt.plot(x,y,'b:o')` tracera en bleu (b) une courbe en pointillés (:) avec les points indiqués par des • (o).

Exemples et explications détaillées sur divers sites internet comme `http://matplotlib.org`.

2.2 Représentation d'un nuage de points

L'instruction `plt.scatter(X,Y)` permet d'obtenir un nuage de points non reliés. Cela peut être très intéressant en mécanique : en représentant une position à intervalle de temps fixé, on visualise l'évolution de la vitesse de déplacement par l'écartement plus ou moins grand des points :



2.3 Représentation graphique d'une fonction

Une "courbe" étant constituée de points reliés entre eux, il convient d'échantillonner la fonction pour obtenir une liste d'abscisses $[x_k]$ et une liste d'ordonnées $[f(x_k)]$.
Cela peut se faire avec des tableaux de numpy ou des listes :

```
import matplotlib.pyplot as plt
import numpy as np
X = np.linspace(-5,5,101) # tableau
Y = np.sin(X) # idem
plt.figure()
plt.plot(X,Y)
plt.show()
```

```
import matplotlib.pyplot as plt
import math as m
X = [5-k/10 for k in range(101)] # liste
Y = [m.sin(k) for k in X] # idem
plt.figure()
plt.plot(X,Y)
plt.show()
```

2.4 Courbes planes paramétrées

Une telle courbe est définie par $x = f_1(p)$ et $y = f_2(p)$, où la variable p est appelée *paramètre*.

Exemples : une parabole par $\begin{cases} x = 2t \\ y = 5t^2 \end{cases}$; un cercle par $\begin{cases} x = \cos\theta \\ y = \sin\theta \end{cases}$.

Le paramètre peut avoir une interprétation physique : un temps, un angle...

Comme précédemment, les valeurs du paramètre peuvent être définies par une liste ou un tableau, pour servir ensuite à calculer les listes ou tableaux d'abscisses et d'ordonnées.

Ainsi pour tracer un cercle :

```
import matplotlib.pyplot as plt
import numpy as np
theta = np.linspace(0,2*np.pi,100) # le paramètre est ici l'angle de Ox à OM
x = np.cos(theta) ; y = np.sin(theta)
plt.figure()
plt.axis('equal') # -> repère orthonormé
plt.plot(x,y)
plt.show()
```

Rq : sans `plt.axis('equal')`, le cercle apparaîtrait déformé en ellipse.

2.5 Graphiques multiples

Plusieurs graphiques peuvent être juxtaposés dans une fenêtre, par la commande `'subplot'`.

La syntaxe est plt.subplot(*nb_lignes*, *nb_colonnes*, *position*).

La position est un numéro dans la grille 'lignes x colonnes', en commençant par 1 en haut à gauche et en parcourant ligne après ligne.
Ainsi, pour deux graphes l'un au-dessus de l'autre : `plt.subplot(2,1,1)` et `plt.subplot(2,1,2)`.
Et pour deux graphes l'un à côté de l'autre (cf.2.2) : `plt.subplot(1,2,1)` et `plt.subplot(1,2,2)`.

Voir exemple : `http://www.python-simple.com/python-matplotlib/graphes-multiples.php`.

```
pyplot.figure(1)
pyplot.subplot(1, 2, 1)
pyplot.scatter(range(5), [x ** 2 for x in range(5)], color = 'blue')
pyplot.subplot(2, 2, 2)
pyplot.plot(range(5), color = 'red')
pyplot.subplot(2, 2, 4)
pyplot.bar(range(5), range(5), color = 'green')
```

CHAPITRE 3

ÉQUATIONS ALGÉBRIQUES

3.1 Position du problème

3.1.1 Équation algébrique $f(x) = 0$

On envisagera une fonction f de classe $\mathcal{C}^n, n \geq 1$, avec :

- un intervalle $[a, b]$ sur lequel elle s'annule au moins une fois : $f(\alpha) = 0$;
- une valeur initiale x_0 dans cet intervalle (par défaut $\frac{a+b}{2}$) ;
- une majoration de l'écart entre la valeur vraie de la solution et la valeur approchée retenue ; on notera $\boxed{\alpha = \text{valeur approchée retenue} \pm \epsilon}$, où ϵ est *l'incertitude absolue* sur le résultat.

3.1.2 Résolution exacte et valeur approchée

Envisageons un cas où l'analyse mathématique fournit l'expression d'une solution exacte, par exemple $x = \frac{-b \pm \sqrt{\Delta}}{2a}$ pour une équation du second degré. Pour calculer les valeurs numériques correspondantes, il faudra effectuer plusieurs calculs nécessairement approchés (en particulier pour calculer $\sqrt{\Delta}$), ce qui conduira à n'obtenir qu'une approximation de racines dont on connaît pourtant l'expression littérale mathématiquement exacte !

Ainsi pour le calcul du "nombre d'or" $\varphi = \frac{1+\sqrt{5}}{2}$; on ne peut que donner un arrondi de son écriture décimale infinie (1,618 033 988 749 894 848 204...) : en Python, on obtient $\phi \approx 1{,}618\,033\,988\,749\,895$.

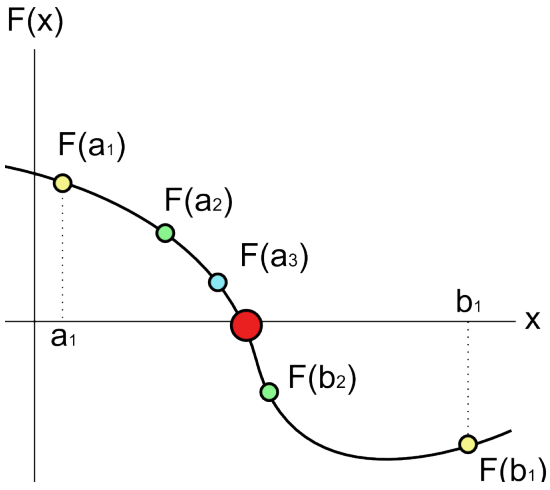
3.1.3 Résolution numérique

De façon plus générale, on ne dispose pas forcément d'expressions exactes des solutions, par exemple pour l'équation simple $\cos x = x$ qui admet à l'évidence une solution entre 0 et 1 !

Il faudra dans ce cas se reposer sur des méthodes qui ressortent de l'*analyse numérique*, c'est-à-dire des suites de calculs qui amènent progressivement à l'approximation souhaitée du résultat, sans qu'on dispose d'une expression théorique de celui-ci.

On s'attachera plus ici à la mise en pratique de ces méthodes qu'à la discussion des aspects mathématiques, qui n'en sont pas moins essentiels pour la validation des algorithmes, et pour la maîtrise des erreurs commises.

3.2 La méthode dichotomique (ou 'par bissection')



Étant donné un intervalle contenant un zéro de la fonction f, on le partage en deux parties égales, on sélectionne le sous-intervalle dans lequel existe un zéro, et on recommence jusqu'à obtenir un encadrement satisfaisant de la valeur notée α telle que $f(\alpha) = 0$.

Ainsi, dans l'exemple illustré ci-contre, l'intervalle contenant α est successivement $[a_1, b_1]$, puis $[a_1, b_2]$, puis $[a_2, b_2]$, puis $[a_3, b_2]$, etc.

Pour mettre en pratique la méthode, on utilise le fait que $f(x)$ change de signe au passage par 0 ; en effet, si f est continue sur l'intervalle $[a, b]$ et telle que $f(a)$ et $f(b)$ soient de signes opposés, alors d'après le théorème des valeurs intermédiaires f a au moins un zéro dans l'intervalle $[a, b]$.

Il y a *dichotomie* car *on divise l'intervalle en deux* en calculant $c = (a + b)/2$; deux cas sont alors possibles : soit $f(a)$ et $f(c)$ sont de signes contraires, soit $f(b)$ et $f(c)$ sont de signes contraires. Le zéro de f étant contenu dans l'intervalle pour lequel il y a changement de signe, on applique à nouveau la dichotomie à cet intervalle, et ainsi de suite.

Si l'indice 0 désigne l'état intial (i.e. $[a_0, b_0] = [a, b]$), après n itérations du processus l'intervalle contenant le zéro de f sera $[a_n, b_n]$, de largeur $(b - a)/2^n$. À chaque étape, on peut donner comme valeur approchée de α le milieu de l'intervalle le contenant, soit $c_n = (a_n + b_n)/2$; on a alors une majoration de l'erreur absolue par la demi-largeur de l'intervalle : $|\alpha - c_n| \leq (b_n - a_n)/2$, d'où :

$$|\alpha - c_n| \leq (b - a)/2^{n+1}.$$

On divise la largeur de l'intervalle par 2 à chaque étape, donc environ par 1 000 en 10 étapes.

Commentaires :

- la méthode est parfaitement sûre (en dehors des limites de calcul de la machine) ;
- mais elle est lente (10 étapes de plus pour gagner 3 chiffres significatifs) ; l'augmentation de précision est *proportionnelle au nombre d'étapes*, on parle de *convergence linéaire* ;
- pour atteindre une incertitude absolue ϵ donnée, on peut l'implémenter avec une boucle `for` si on calcule préalablement le nombre d'étapes nécessaires (à partir de la relation ci-dessus), ou, plus simplement, avec une boucle `while` testant la largeur de l'intervalle de travail et s'arrêtant lorsque celle-ci devient inférieure ou égale à 2ϵ ;
- quoique robuste, la méthode peut toutefois être mise en difficulté par des fonctions présentant des zones très "plates", i.e. $f(x) \approx C^{te}$ au voisinage de α ; la détermination du signe de $f(c)$ et du sous-intervalle à choisir peut être rendue aléatoire par les erreurs d'arrondi ; on peut alors choisir d'arrêter le calcul si $|f(c)|$ devient inférieure à une certaine valeur ;
- il est préférable d'utiliser la méthode sur un intervalle contenant un seul zéro de f ; sinon, au mieux on en trouvera un, mais on risque de n'en trouver aucun, selon la fonction, l'intervalle de départ et l'implémentation de l'algorithme...

3.3 Mise en œuvre

On peut typiquement écrire une fonction **dicho(f, a_0, b_0, ϵ)** comme celle-ci, qui contient par sécurité une vérification de l'adéquation de l'intervalle de départ donné :

```
1  def dicho(f,a,b,eps):
2      if f(a)*f(b) > 0:
3          print("Mauvais choix de l'intervalle [a,b].") ; return
4      else:
5          compt = 0                       # intialisation du compteur
6          an, bn = min(a,b), max(a,b),    # initialisation de l'intervalle courant
7          larg = bn - an                  # largeur courante
8          cn = ( bn + an ) / 2            # resultat courant a larg/2 pres
9          while larg > 2 * eps:           # resultat final au pire a eps pres
10             if f(an)*f(cn) > 0:
11                 an = cn
12             else:
13                 bn = cn
14             cn = ( bn + an ) / 2
15             larg = bn - an
16             compt += 1
17     return [cn, compt]
```

3.4 La fonction bisect

C'est une fonction qui fait partie de la bibliothèque **scipy.optimize** et qu'il faudra donc importer avant toute utilisation.
Rappel :

- soit on importe toute la bibliothèque avec un alias, par exemple **import** scipy.optimize **as** resol et on appellera la fonction par `resol.bisect(...)`,
- soit on se contente d'importer la fonction utile par **from** scipy.optimize **import** bisect ce qui permettre ensuite d'appeler directement `bisect(...)`.

Nous utiliserons cette fonction comme une "boîte noire", c'est-à-dire sans chercher à connaître son fonctionnement interne, mais son nom indique qu'il s'agit d'une méthode dichotomique.

Séquence d'appel minimale : bisect(f, a, b) pour un zéro de la fonction f entre a et b.
💣 Il faut avoir $f(a)\,f(b) \leq 0$ pour que la procédure démarre ; ainsi, on pourra résoudre $x^2 - 1 = 0$ entre 0 et 1 mais pas entre -2 et 2.
On note que l'utilisateur ne donne pas d'indications sur la précision du résultat et ne maîtrise pas le déroulement du travail.

```
>>> import scipy.optimize as so

>>> import numpy as np

>>> def f(x):
...     return np.cos(x) - x

>>> print(so.bisect(f, 0, 1))
0.7390851332147577
```

On trouvera une présentation détaillée sur le site `https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.bisect.html`, qui indique au passage l'existence d'*autres procédures* de résolution (rubrique 'See also'), qui ne figurent pas au programme de ce cours.

CHAPITRE 4

DÉRIVATION – INTÉGRATION

4.1 Calcul approché du nombre dérivé en un point

4.1.1 Définition

Pour une fonction définie sur $\mathbb{R}$ au voisinage d'un point x_0 ; son **taux d'accroissement** entre x_0 et x est :

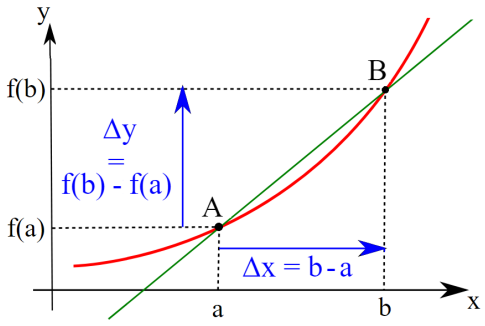
$$\boxed{\frac{\Delta f}{\Delta x} = \frac{f(x) - f(x_0)}{x - x_0}}$$

Lorsque ce taux d'accroissement admet une limite lorsque $x \longrightarrow x_0$, la fonction est dite **dérivable** et cette limite est le **nombre dérivé de f en x_0**, noté $f'(x_0)$ ou $\frac{df}{dx}(x_0)$.

$$\boxed{f'(x_0) = \frac{df}{dx}(x_0) = \lim_{\Delta x \to 0} \frac{\Delta f}{\Delta x}}$$

4.1.2 Signification géométrique

Sur la figure ci-contre, A et B étant deux points de la courbe représentative de f, la droite (AB) est une **sécante** de cette courbe (le segment $[AB]$ étant une 'corde').
Le taux d'accroissement de f quantifie la pente de la sécante dont l'équation est :



$$y = f(a) + \frac{f(b) - f(a)}{b - a}(x - a) \Leftrightarrow \frac{y - f(a)}{x - a} = \frac{f(b) - f(a)}{b - a}$$

Lorsque B se rapproche de A, à la limite $\Delta x \to 0$ la sécante devient la **tangente** en A à la courbe représentant f. Son équation est donc : $y = f(a) + f'(a)(x - a)$.

La courbe représentant f a donc pour tangente au point d'abscisse x_0 la droite d'équation :

$$\boxed{y = f(x) + f'(x_0)(x - x_0)}.$$

4.1.3 Calcul approché d'un nombre dérivé

La valeur de la limite figurant dans la définition peut être approchée par le calcul d'un taux d'accroissement, *avec un Δx "suffisamment" petit* :

$$\boxed{f'(x_0) \approx \frac{f(x_0+\Delta x)-f(x_0)}{\Delta x}}$$

Que signifie "suffisamment" petit ? On ne peut pas donner de valeur universelle, il faut se baser sur la courbure plus ou moins grande de la courbe : si courbure est faible, la courbe est proche d'une droite et Δx n'a pas besoin d'être très petit ; si la courbure est plus marquée, il faudra rapprocher davantage les points A et B pour faire apparaître la tangente.

Prenons l'exemple d'une fonction sinusoïdale : elle varie pratiquement linéairement au passage par 0, et sa courbure est maximale aux extrémums. En conséquence, l'approximation du nombre dérivé par le taux d'accroissement, avec un Δx pas trop petit, est bonne dans le premier cas et mauvaise dans le second.

fonction f	fonction dérivée f'	valeur de Δx	et taux d'acc. en 0	$f'(0)$
sin	cos	0,1	$\approx 0{,}9983$	1
cos	sin	0,1	$\approx -0{,}0500$	0

4.1.4 Calcul d'une liste de nombres dérivés

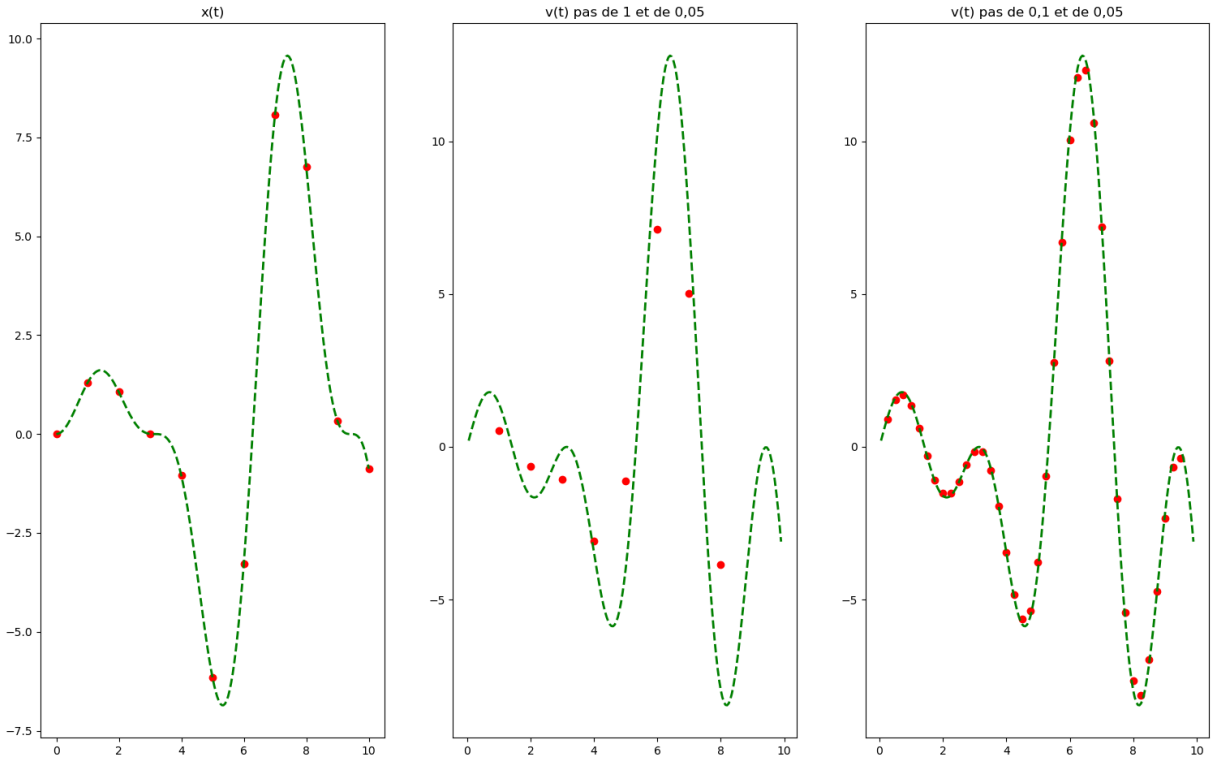
Considérons une liste de valeurs correspondant à un **échantillonnage** de f sur un intervalle :

$[x_0, x_1, x_2, ..., x_{n-1}] \rightsquigarrow [f(x_0), f(x_1), f(x_2), ..., f(x_{n-1})]$ (des positions, des valeurs de pH, etc).

Si l'échantillonnage est assez fin, on peut en déduire une liste de $n-1$ *nombres dérivés approchés* en construisant par itérations la liste des $\frac{f(x_{k+1})-f(x_{k-1})}{x_{k+1}-x_{k-1}}$ pour $k \in \{1, 2, ..., n-2\}$.

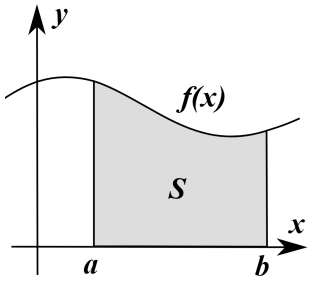
NB : souvent l'échantillonnage est *à pas constant* : $x_k = x_0 + k \times pas \Rightarrow \boxed{f'(x_k) \approx \frac{f(x_{k+1})-f(x_{k-1})}{2*pas}}$.

Commenter les approximations de $v(t_k)$ ci-dessous (avec $v = \dot{x}$) :



4.2 Calcul approché d'une intégrale sur un segment

En dehors des cas où on peut utiliser des primitives donnant des valeurs "exactes" (mais soumises aux erreurs d'arrondis comme vu dans la partie précédente), on part du fait que la valeur de l'intégrale cherchée mesure l'aire comprise entre la courbe représentative de $f(x)$ et l'axe des abscisses.
C'est cette aire dont on va calculer une valeur approchée.



Les méthodes proposées reposent sur l'idée suivante : découper l'intervalle d'intégration en sous-intervalles suffisamment petits pour qu'on puisse sur chacun approcher la courbe représentative par une courbe très simple, pour laquelle le calcul d'aire est facile ; il ne reste alors plus qu'à faire la somme des petites aires obtenues pour chaque tranche.

4.2.1 Méthode des rectangles

(ci-dessous, figures de gauche)

L'intervalle $[a,b]$ est divisé en n tranches $[x_k, x_{k+1}]$ de largeur $h = \frac{b-a}{n}$, avec $x_k = a + k \cdot h$ et $0 \leq k \leq n-1$. La simplification maximale est de considérer $f(x)$ constante sur chaque sous-intervalle. Ainsi, sur chaque tranche on remplace $f(x)$ par la valeur $f(x_k)$ (méthode des rectangles "à gauche") ou $f(x_{k+1})$ (méthode des rectangles "à droite"), et on approxime l'intégrale par la somme des aires des rectangles ainsi obtenus :

$$\int_a^b f(x)dx \approx R_n^g(f,a,b) = h \cdot \sum_{k=0}^{n-1} f(x_k) \qquad \text{ou} \qquad \int_a^b f(x)dx \approx R_n^d(f,a,b) = h \cdot \sum_{k=0}^{n-1} f(x_{k+1})$$

Illustration de la méthode des rectangles :

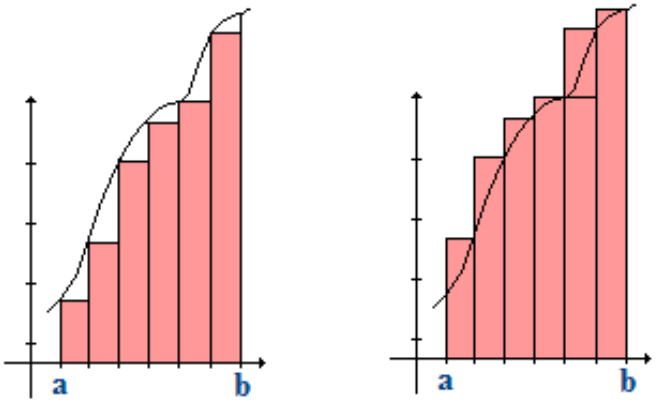
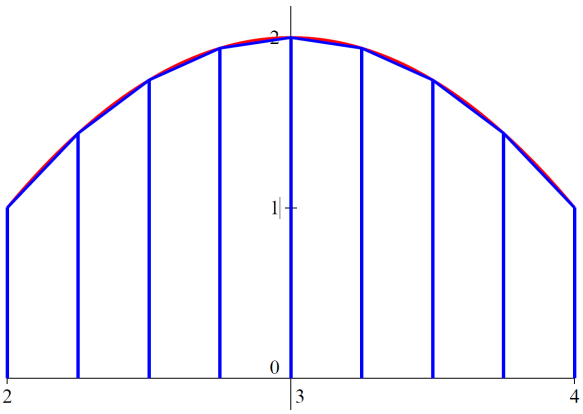


Illustration de la méthode des trapèzes :



Pour estimer la précision, considérons l'illustration ci-dessus à gauche : au premier ordre (linéarisation de f), l'erreur absolue commise sur chaque tranche vaut approximativement :

$$h \cdot |f(x_{k+1}) - f(x_k)|/2 \approx h \cdot |h \cdot f'(x_k)|/2 = |f'(x_k)| \cdot h^2/2$$

Si M_1 est un majorant de $|f'(x)|$ sur $[a,b]$, on majore l'erreur absolue sur la somme de n termes :

$$|R_n(f,a,b) - \int_a^b f(x)dx| \leq n \cdot M_1 \cdot h^2/2 = M_1 \frac{(b-a)^2}{2n}$$

L'algorithme est de complexité linéaire par rapport à n, mais sa convergence est très lente puisqu'il faut 10 fois plus d'étapes pour gagner un chiffre significatif ; il faudrait de l'ordre de 10^{10} calculs pour obtenir 10 chiffres significatifs ! Il n'est donc pas utilisé en pratique et n'est présenté que pour son intérêt pédagogique.

Commentaires :

— la méthode donne le résultat exact pour les fonctions constantes ;

— sur les tranches où f est croissante, les rectangles "à gauche" sous-estiment la valeur exacte, alors que les rectangles "à droite" la surestiment ; c'est le contraire si f décroît ;

— pour corriger cet effet, on peut prendre des rectangles de hauteur $f(\frac{x_k+x_{k+1}}{2})$, afin que sur chaque tranche une certaine compensation se produise ; cette méthode du *point médian* permet une convergence un peu plus rapide.

4.2.2 Méthode des trapèzes

Sur chaque segment $[x_k, x_{k+1}]$, on remplace la courbe par le segment de droite reliant les deux points extrêmes, ce qui conduit à calculer une somme d'aires de trapèzes :

$$\int_a^b f(x)dx \approx T_n(f,a,b) = h \cdot \sum_{k=0}^{n-1} \frac{f(x_k)+f(x_{k+1})}{2}$$

Pour estimer la précision de la méthode, il convient d'utiliser un développement d'ordre 2 de f (fonction de classe $\mathcal{C}^2$), on obtient alors :

$$|T_n(f,a,b) - \int_a^b f(x)dx| \leq M_2 \frac{(b-a)^3}{12n^2}$$

où M_2 est un majorant de $|f''|$ sur $[a,b]$.

Commentaires :

— la méthode donne le résultat exact pour les fonctions affines ;

— on remarque qu'à l'exception de $f(x_0) = f(a)$ et $f(x_n) = f(b)$, les autres $f(x_k)$ apparaissent deux fois dans la somme ; pour éviter de les calculer deux fois, on peut réécrire T_n ainsi :

$$T_n(f,a,b) = h \cdot (\frac{f(a)+f(b)}{2} + \sum_{k=1}^{n-1} f(x_k))$$

même complexité que pour la méthode des rectangles, linéaire par rapport à n ;

— en revanche, la convergence est bien meilleure puisque l'erreur décroît en $1/n^2$; ainsi, pour avoir 10 chiffres significatifs, il faudra n de l'ordre de 10^5, ce qui est accessible.

4.2.3 Méthodes existantes de Python

On les trouve dans le sous-module **scipy.integrate**. On retiendra **quad(f,a,b)** , la plus utilisée. Elle calcule une valeur approchée de $\int_a^b f(x)dx$ selon une méthode optimisée par Python, avec un pas variable selon la pente.

$$-= \mathcal{FIN} =-$$