Outils numériques

1 Tableaux

Extrait du programme

1. Tableaux	
Tableaux à une ou deux dimensions.	Choisir une structure de données appropriée à la modélisation d'un problème physique. Réaliser des opérations algébriques simples sur des tableaux. Utiliser les fonctions de base de la bibliothèque **numpy** (leurs spécifications étant fournies) pour manipuler des tableaux.

Le module `NumPy` introduit un nouveau type l'`array` qui est semblable à une liste, mais où tous les éléments doivent être du même type (entier, flottant, booléen ou encore chaîne de caractères). Le tableau peut être unidimensionnel ou multidimensionnel.

Exemple :

`tab = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]] )` crée un tableau `numpy` de dimension 2.

Fonctionnalités

La bibliothèqe `numpy` contient de nombreuses fonctions optimisées pour le calcul sur des `array`. Ces fonctions font appel à du code compilé dont l'exécution est entre 100 et 1000 fois plus rapide que les mêmes fonctions s'exécutant sur des listes.[a]

Il est possible d'effectuer les opérations classiques $+, -, \times, \div$ entre deux tableaux (l'opération s'effectue alors terme à terme) ou entre un tableau et une constante (l'opération est appliquée à tous les termes avec la constante).

Numpy fournit un certain nombre de fonctions mathématiques dans le domaine de la trigonométrie, des exponentielles et des logarithmes (`np.cos()`, `np.log()`, `np.tanh()`, etc.). Ces fonctions prennent en argument un `array` et renvoient également un `array`.

Il existe également des fonctions pour renvoyer la moyenne (`np.mean()`), le minimum (`np.min()`, l'écart-type (`np.std`), la somme (`np.sum()`), le module (`np.abs()`), l'argument (`np.angle()`) des éléments d'un tableau.

a. Attention, la complexité des algorithmes est inchangée, ce sont les opérations élémentaires qui s'exécutent plus rapidement. Comme les tableaux gardent des tailles constantes et des éléments de même nature, il y a beaucoup moins de précautions à prendre sur l'occupation en mémoire.

Vous devez savoir qu'il y a beaucoup de fonctions très utiles dans cette bibliothèque, mais vous n'avez pas à les connaitre par cœur.

2 Outils graphiques

Extrait du programme

Domaines numériques	Capacités exigibles
1. Outils graphiques	
Représentation graphique d'un nuage de points.	Utiliser les fonctions de base de la bibliothèque **matplotlib** pour représenter un nuage de points.
Représentation graphique d'une fonction.	Utiliser les fonctions de base de la bibliothèque **matplotlib** pour tracer la courbe représentative d'une fonction.
Courbes planes paramétrées.	Utiliser les fonctions de base de la bibliothèque **matplotlib** pour tracer une courbe plane paramétrée.

2.1 Représentation d'un nuage de points

En TP ou en TIPE, vous êtes souvent amenés à tracer des courbes issues de résultats expérimentaux ou numériques. Le but de cette partie est de vous donner quelques lignes de codes permettant d'obtenir une courbe présentable pour un rapport ou une présentation. Beaucoup des fonctions présentées ici ne sont pas à connaitre, mais elles vous serons peut-être utiles un jour.
Pour commencer, on importe le module graphique `matplotlib.pyplot`.

```
import matplotlib.pyplot as plt
```

Les données peuvent être stockées sous la forme d'un tableau `numpy` ou d'une liste. Ici pour notre exemple, on prendra deux listes :

```
x = [0, 1, 2, 3, 4, 5]
y = [0.2, 1.3, 2.4, 3.3, 4.4, 5.5]
```

On veut afficher les valeurs contenues dans `y` en fonction de celles stockées dans `x`. On commence par créer une figure :

```
plt.figure(1, figsize=[6/2.54, 6/2.54]) #figure 1, avec sa taille en pouce
plt.clf() # on vide la figure
```

Remarque :

Il est important de préciser la taille de la figure, cela vous évitera de la réduire/agrandir au moment de l'inclure dans votre rapport. Ainsi, la figure que vous aurez dans votre dossier sera exactement la même que celle tracée sous python. Attention, pour matplotlib la taille doit être en pouce (2.54 cm = 1 pouce). Une fois la figure créée, on peut y inclure un système d'axe avec la fonction `axes`. Cette fonction est très utile si l'on souhaite tracer plusieurs courbes sur une même figure ou placer un encart dans une figure. On entre en argument, une liste avec : la position à gauche, la position en bas, la largeur et la hauteur des axes.

```
plt.axes([0.2,0.25,0.7,0.7]) #position et taille en ratio de la figure.
```

Arrive le moment de tracer la courbe. Pour cela, on utilise la fonction `plot`. Celle-ci accepte de nombreux arguments dont voici les principaux :

```
plt.plot(x, #1er argument : données en abscisse.
         y, # 2nd argument : données en ordonnées. Les arguments suivants peuvent être
   ajoutés dans n'importe quel ordre.
         marker = 's', #le type de marqueur que l'on souhaite, ici des carrés (square),
         markerfacecolor  = 'r', #la couleur de l'intérieur des marqueur (ici rouge)
         markeredgecolor = [0.1,0.1,0.1], #la couleur du bord du marqueur (ici on entre
   le code RGB de la couleur souhaité)
         markeredgewidth = 1, #la largeur du bord du marqueur
         markersize = 8, #la taille du marqueur
         linestyle = '', #type du segment qui relie les points (le mieux c'est encore de
    ne pas relier les points  ( linestyle = '')
```

```
        color = 'k',#couleur du trait
        linewidth = 1 #largeur du trait
        )
```

On évite de relier les points de mesure expérimentaux entre eux et on utilise des marqueurs de tailles et couleurs cohérentes. Si l'on trace plusieurs courbes sur le même système d'axes, il faut bien penser à mettre des marqueurs différents (attention parfois les rapports sont imprimés en noir et blanc). Il faut ensuite nommer les axes. Évitez autant que possible les titres trop longs, contentez-vous par exemple du nom des variables avec leur unité.

Remarque :

L'avantage de Python est qu'il permet d'insérer du texte en LaTeX et donc d'avoir la possibilité d'écrire des formules mathématiques. Celles-ci s'écrivent entre $:

- $\frac{a}{b}$ s'écrit `\frac{a}{b}` ,
- a^{b+1} s'écrit `a^{b+1}` ,
- $\sqrt{T_0}$ s'écrit `\sqrt{T_{0}}` .

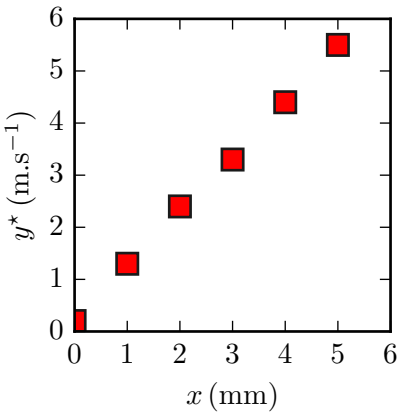
Les formules sont rendues en italiques alors que les unités en physique doivent être écrite de manière droite, on utilise alors la commande `\rm` , le code `\,` sert à insérer une espace entre la variable et son unité.

```
Taille_police = 11 # la police sera de taille Taille_police
plt.xlabel(r' $x \, (\rm mm)$',fontsize = Taille_police) #fontsize : taille de la police
plt.ylabel(r' $y^\star \, (\rm m.s^{-1})$',fontsize = Taille_police)
```

Il ne faut pas oublier la taille de la police. Dans le cas d'un rapport (figure au milieu d'un texte) le texte de la figure doit être de la même taille et de la même police que le corps du texte[1]. Par exemple, Times en police 11. En revanche, pour une figure destinée à une présentation le texte doit être visible de loin, il faut donc l'écrire avec une police gigantesque (ne pas hésiter à exagérer un peu). Il existe de nombreux autres arguments pour tout ce qui est texte (couleur de la police, orientation, italique, gras, etc. cherchez `help(plt.text)` pour tous les obtenir). De même, on peut jouer sur la taille, position, polices des graduations, pour cela, consultez `help(plt.tick_params)` , `help(plt.ticklabel_format)`, `help(plt.xticks)`.
Pour un affichage bien adapté aux données, on peut fixer l'échelle en abscisse et en ordonnée :

```
plt.xlim([0, 6]) # On choisi les limites pour les axes
plt.ylim([0, 6])
```

Une fois que votre figure vous semble correcte et présentable, c'est le moment de l'enregistrer. Il faut absolument l'enregistrer dans un format vectoriel (.eps ou .pdf) et **bannir le .PNG ou .JPG**. Voici la courbe que l'on vient de réaliser.



2.2 Représentation graphique d'une fonction

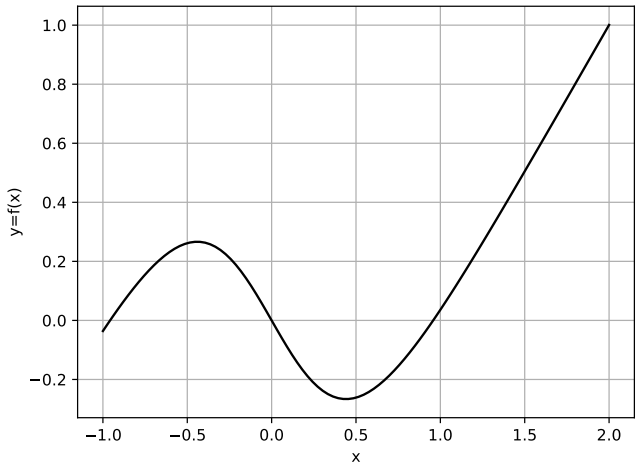
Pour tracer courbe représentative d'une fonction, on peut déjà définir cette fonction. Prenons la fonction f telle que $\forall x \in \mathbb{R}$, $f(x) = x - \tanh(2x)$

1. On comprend alors l'intérêt de préparer la figure à la bonne taille dès le départ

```
def f(x):
    ''' entrée x :un flottant
        sortie y = x - tanh(2x)'''
    y = x - np.tanh(2*x)
    return y
```

Ensuite, on crée un tableau `x` contenant les valeurs des abscisses pour lesquelles on veut évaluer la fonction. Le plus simple et le plus efficace est d'utiliser un tableau `numpy`, l'instruction `x = np.linspace(deb, fin, nb_points`, permets de créer un tableau 1D de `nb_points` points équirépartis entre `deb` et `fin` inclus.
On créé le tableau des ordonnées correspondantes : `y = f(x)`, et enfin, on trace `y` en fonction de `x`.

```
plt.plot(x, y, 'k')
plt.xlabel('x')
plt.ylabel('y=f(x)')
plt.grid('on')
plt.show()
```

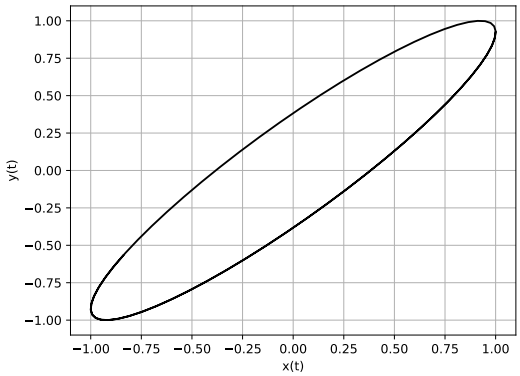


2.3 Courbes planes paramétrées

Le tracer d'une courbe paramétrée suit le même principe : on définit une tableau `t` puis deux autres $x(t)$ et $y(t)$, enfin on trace `y` en fonction de `x`.

Exemple :

```
deb = 0
fin = 10
nb_point = 1000

t = np.linspace(deb, fin, nb_points)
x = np.cos(t)
y = np.cos(t + np.pi/8)

plt.plot(x, y, 'k')
plt.xlabel('x(t)')
plt.ylabel('y(t)')
plt.grid('on')
plt.show()
```


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
y(t)
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
x(t)

3 Résolution d'équations algébriques

Extrait du programme

2. Équations algébriques	
Résolution d'une équation algébrique ou d'une équation transcendante : méthode dichotomique, méthode de Newton.	Déterminer, en s'appuyant sur une représentation graphique, un intervalle adapté à la recherche numérique d'une racine par la méthode dichotomique ou par la méthode de Newton. Mettre en œuvre la méthode dichotomique ou la méthode de Newton afin de résoudre une équation avec une précision donnée. Utiliser les fonctions **bisect** ou **newton** de la bibliothèque **scipy.optimize** (leurs spécifications étant fournies).
Systèmes linéaires de n équations indépendantes à n inconnues.	Définir les matrices A et B adaptées à la représentation matricielle AX = B du système à résoudre. Utiliser la fonction **solve** de la bibliothèque **numpy.linalg** (sa spécification étant fournie).

3.1 Dichotomie

Soit $f : [a,b] \to \mathbb{R}$ une applications réelle ($a > b$), continue sur $[a,b]$ et admettant au moins une valeur $c \in [a,b]$ telle que $f(c) = 0$. On cherche un algorithme donnant une valeur approchée de c.

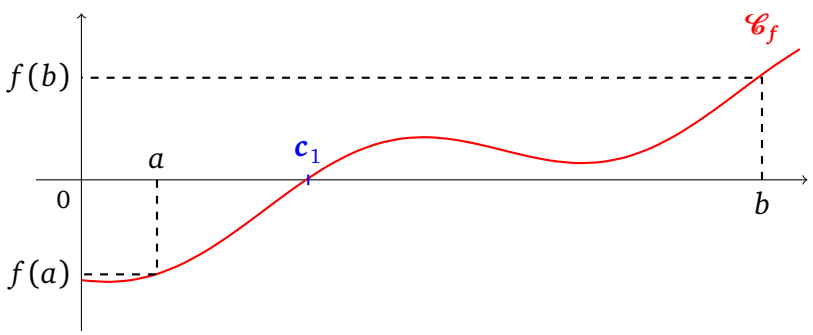
Méthode

Le principe de la dichotomie repose sur :

- le théorème des valeurs intermédiaires :
  Si $f : [a;b] \to \mathbb{R}$ est continue sur le segment $[a,b]$ et si $f(a)$ et $f(b)$ sont de signes contraires, alors l'équation $f(x) = 0$ admet au moins une solution dans l'intervalle $[a,b]$
- le théorème de la valeur intermédiaire :
  Si $f : [a;b] \to \mathbb{R}$ est continue et strictement monotone sur $[a,b]$ et si $f(a)$ et $f(b)$ sont de signes contraires, alors l'équation $f(x) = 0$ admet une et une seule solution dans l'intervalle $[a,b]$.

Pour obtenir des valeurs approchées d'une solution à l'équation $f(x) = 0$ dans l'intervalle $[a,b]$, on peut procéder par **dichotomie.** La fonction $f : [a;b] \to \mathbb{R}$ est supposée continue sur l'intervalle $[a,b]$ et $f(a)$ et $f(b)$ sont supposés de signe contraire.

- on part de l'intervalle $[a,b]$ donc $g_0 = a$ et $d_0 = b$ (avec $a < b$)
- à chaque étape, on évalue la fonction f au point milieu de l'intervalle $[g_n;d_n]$ i.e. en $m = \frac{g_n+d_n}{2}$.
  Si $f(g_n)$ et $f(m)$ sont de signe contraire, on se place sur l'intervalle $[g_n;m]$ i.e. $g_{n+1} = g_n$ et $d_{n+1} = m$, car il contient une solution de l'équation $f(x) = 0$.
  Sinon $f(d_n)$ et $f(m)$ sont de signe contraire, et on se place sur $[m;d_n]$ i.e. $g_{n+1} = m$ et $d_{n+1} = d_n$.
- on s'arrête quand la précision est satisfaisante.
  À chaque étape, la longueur de l'intervalle est divisée par 2. La borne gauche g_n est une valeur approchée par défaut de la solution recherchée c avec $|g_n - c| \leqslant |b-a| \times 2^{-n}$.
  La borne droite fournit une valeur approchée par excès.



Difficultés pratiques

Les suites $(g_n)_{n\in\mathbb{N}}$ et $(d_n)_{n\in\mathbb{N}}$ obtenues par la méthode de dichotomie convergent toujours vers un zéro de la fonction f (sous réserve que f soit continue sur $[a;b]$ et $f(a)\times f(b)<0$). Mais s'il y a plusieurs zéros sur l'intervalle considéré, on ne sait pas vers lequel les suites vont converger.
Le seuil d'arrêt peut être difficile à estimer, d'autant que les calculs des valeurs prises par f sont nécessairement arrondies comme tout calcul avec des flottants.

Exemple :

Un exemple de programmation itérative

```
def dicho(f, a, b, eps):
    ''' Donne une estimation à eps près d'un zéro de f entre a et b
        entrées : f, callable, fonction dont on cherche le zero
                  a, float, borne inférieure de l'intervalle de recherche
                  b, float, borne supérieure de l'intervalle
                  eps, float, précision sur la recherche du zéro
        sortie : m, float, estimation du zéro. '''
    while b-a > eps :
        m = (a+b)/2
        if f(m)*f(a) < 0 :  # le zéro est entre a et m
            b = m
        else :
            a = m
        m = (a+b)/2
    return m
```

Exemple :

Un exemple de programmation récursive

```
def dicho_rec(f, a, b, eps):
    ''' Donne une estimation à eps près d'un zéro de f entre a et b
        entrées : f, callable, fonction dont on cherche le zero
                  a, float, borne inférieure de l'intervalle de recherche
                  b, float, borne supérieure de l'intervalle
                  eps, float, précision sur la recherche du zéro
        sortie : m, float, estimation du zéro. '''
    m = (a+b)/2
    if b-a<eps :
        return m
    elif f(m)*f(a) < 0 :  # le zéro est entre a et m
        b = m
    else :
        a = m
    return dicho_rec(f, a, b, eps)
```

Les deux fonctionnent fournissent bien la même valeur à ϵ près (avec $f(x)=x-\tanh(2x)$) :

```
print(dicho(f, 1e-3, 1, 1e-6))
print(dicho_rec(f, 1e-3, 1, 1e-6))
```

affiche :

```
0.9575039030462502
0.9575037544071674
```

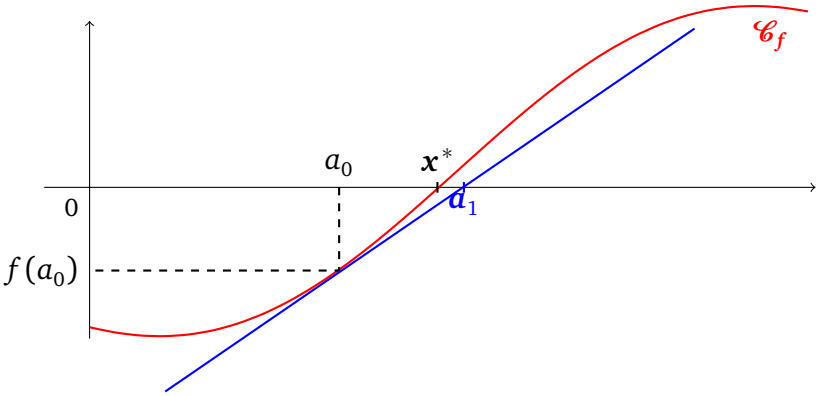
3.2 Méthode de Newton

Méthode

On cherche à résoudre numériquement une équation de la forme $f(x) = 0$ où f est une fonction d'une variable réelle, à valeurs réelles, assez régulière (i.e. au moins dérivable). On note x^* une solution de cette équation que l'on souhaite approcher.

On part de a, une valeur approchée assez grossière de x^*. Pour améliorer sa précision :

- on trace la tangente à la courbe de f au point a ;
- on repère le point d'intersection de cette tangente avec l'axe des abscisses ;
- l'abscisse a' de ce point est une nouvelle valeur approchée de x^* dont on espère qu'elle est plus précise que a !



L'équation de la tangente à la courbe au point d'abscisse a est : $y = f(a) + f'(a)(x-a)$
Donc le point d'intersection de cette droite avec l'axe des abscisses a pour coordonnées $(a', 0)$ avec : $0 = f(a) + f'(a)(a'-a)$ d'où :

$$a' = a - \frac{f(a)}{f'(a)}$$

On peut réitérer le procédé pour améliorer cette fois la précision de a' : on obtient une nouvelle valeur approchée a'', que l'on peut elle aussi affiner... et ainsi de suite aussi longtemps qu'on le souhaite. On construit ainsi une suite d'approximations $(a_n)_{n\in\mathbb{N}}$ de la solution x^* qui vérifient :

$$a_0 = a \qquad \text{et} \qquad \forall\, n \in \mathbb{N},\ a_{n+1} = a_n - \frac{f(a_n)}{f'(a_n)}$$

Exemple :

```
def Newton(f, fd, x0, epsilon) :
    ''' Recherche à espilon près du zéros de la fonction f,
      de dérivée fd, proche de x0
      entrées : f callable
                fd callable
                x0 float
                epsilon float
      sortie :  y float'''
    x = x0
    y = x0 - f(x0)/fd(x0)
    while abs(y - x) > epsilon :
        x = y
        y = x - f(x)/fd(x)
    return y
```

Remarque :

Comparons les performances de la méthode de Newton et de la dichotomie pour obtenir des valeurs approchées de $\sqrt{2} \simeq$ `1.4142135623730951` :

n	Méthode de Newton	(erreur)	Méthode par dichotomie	(erreur)
0	`2.0000000000000000`	$5{,}9.10^{-1}$	`1.0000000000000000`	$-4{,}1.10^{-1}$
1	`1.5000000000000000`	$8{,}6.10^{-2}$	`1.5000000000000000`	$8{,}6.10^{-2}$
2	`1.4166666666666667`	$2{,}5.10^{-3}$	`1.2500000000000000`	$-1{,}6.10^{-1}$
3	`1.4142156862745099`	$2{,}1.10^{-6}$	`1.3750000000000000`	$-3{,}9.10^{-2}$
4	`1.4142135623746899`	$1{,}6.10^{-12}$	`1.4375000000000000`	$2{,}3.10^{-2}$
5	`1.4142135623730951`	$< 10^{-16}$	`1.4062500000000000`	$-8{,}0.10^{-3}$
6	`1.4142135623730950`	$-2{,}2.10^{-16}$	`1.4218750000000000`	$7{,}7.10^{-3}$
7	`1.4142135623730951`	$< 10^{-16}$	`1.4140625000000000`	$-1{,}5.10^{-4}$

On constate qu'ici la méthode de Newton converge beaucoup plus rapidement que la méthode par dichotomie, vers $\sqrt{2}$: en 5 itérations seulement, on obtient la précision maximale autorisée par la représentation des flottants en double précision. Pendant ce temps, la dichotomie n'a produit qu'une seule décimale exacte.

3.3 Utilisation d'une fonction dédiée

La fonction `bisect` du module `scipy.optimize` donne une estimation du zéro d'une fonction. La syntaxe n'est pas à connaitre, la documentation doit vous être fournie, néanmoins son utilisation reste assez transparente :

```
from scipy.optimize import bisect

c = bisect(f, 1e-3, 1)

print(c)
```

affiche

```
0.9575040240768902
```

3.4 Systèmes linéaires de n équations à n inconnues

Un **système linéaire** de n équations à n inconnues s'écrit sous la forme suivante :

$$(\mathrm{S}) : \begin{cases} a_{1,1}\,x_1 + \cdots + a_{1,n}\,x_n = b_1 \\ \vdots \\ a_{n,1}\,x_1 + \cdots + a_{n,n}\,x_n = b_n \end{cases} \quad \text{avec} \begin{cases} a_{i,j} & : \text{ les coefficients du système (S),} \\ b_j & : \text{ les seconds membres du système (S),} \\ x_i & : \text{ les inconnues du système (S).} \end{cases}$$

Écriture matricielle du système (S) : $\quad (S) \Leftrightarrow AX = B$

$$\text{avec} \quad A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{pmatrix} \quad , \quad X = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \quad \text{et} \quad B = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

Plusieurs méthodes numériques existent pour résoudre un tel système, vous avez juste à connaitre l'existence de la fonction `solve` du module `numpy.linalg`. Les matrices A et B sont représentés par des tableaux `numpy` 2D.

Exemple :

Voici un exemple de code d'utilisation pour la résolution du système :

$$(\mathrm{S}) : \begin{cases} 1\,x_1 + 2\,x_2 = 1 \\ 3\,x_1 + 5\,x_2 = 2 \end{cases}$$

```
A = np.array([[1, 2], [3, 5]])
B = np.array([1, 2])
X = np.linalg.solve(A, B)
```

La variable `X` est un tableau 1D contenant les valeurs de x_1 et de x_2.

4 Intégration et dérivation

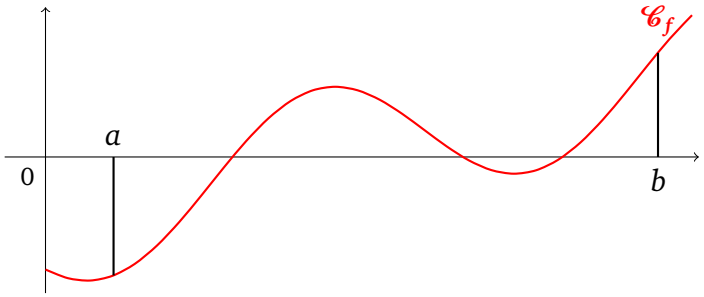
Extrait du programme

3. Intégration – Dérivation	
Calcul approché d'une intégrale sur un segment par la méthode des rectangles.	Mettre en œuvre la méthode des rectangles pour calculer une valeur approchée d'une intégrale sur un segment.
Calcul approché du nombre dérivé d'une fonction en un point.	Utiliser un schéma numérique pour déterminer une valeur approchée du nombre dérivé d'une fonction en un point.

En mathématiques, comme en physique, on est souvent amené à calculer des intégrales. Cependant, un calcul à l'aide de primitives peut s'avérer inutilisable en pratique :

- il n'est pas toujours possible de déterminer des primitives explicites ;
- on a parfois accès seulement à un échantillonnage de f i.e. des valeurs isolées $f(x_0), f(x_1), \dots, f(x_n)$ notamment lorsque f modélise des mesures concrètes effectuées lors d'une expérience.

Ainsi, on a besoin de **méthodes numériques de calcul d'intégrale** (aussi appelées méthodes de quadrature) qui fournissent des valeurs approchées de $\int_a^b f(x)\,\mathrm{d}x$ quand f est une fonction continue sur le segment $[a;b]$. Rappelons l'interprétation géométrique en terme d'aire d'une intégrale : si f est une fonction réelle continue sur $[a;b]$ alors $\int_a^b f(x)\,\mathrm{d}x$ représente **l'aire algébrique** de la région du plan délimitée par l'axe des abscisses et le graphe de f, délimitée à gauche et à droite par les droites verticales correspondant aux abscisses a et b.



L'idée sous-jacente de la méthode présentée (rectangles) est d'approcher cette aire en approchant la fonction f par des fonctions d'intégrale simple à calculer, typiquement des polynômes de bas degré.

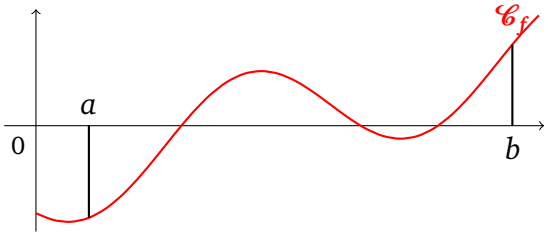
4.1 Méthode des rectangles

Dans la méthode des rectangles à gauche,

- On subdivise le segment $[a,b]$ en n sous-intervalles de même longueur $h = \frac{b-a}{n}$
- On approche la fonction f sur chaque sous-intervalle par une fonction constante, égale à la valeur de f à l'extrémité gauche du sous-intervalle i.e. sur $[\underbrace{a+kh}_{=a_k}, a+(k+1)h]$ par $f(a+kh)$

Ainsi on approche l'aire $\displaystyle\int_a^b f(t)\mathrm{d}t$ par

$$G_n = \sum_{k=0}^{n-1} \frac{b-a}{n} f\left(a + k\frac{b-a}{n}\right) = \frac{b-a}{n} \sum_{k=0}^{n-1} f\left(a + k\frac{b-a}{n}\right)$$



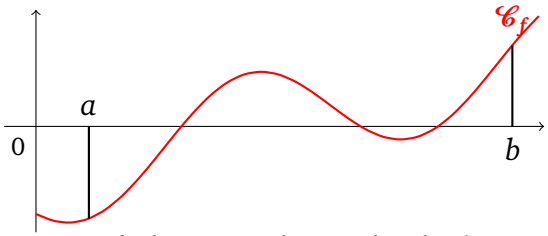
Visuellement, l'aire a été approchée via celles de rectangles s'appuyant à gauche sur le graphe de f.

Dans la méthode des rectangles à droite, la méthode est la même sauf que la valeur choisie sur un sous-intervalle est celle prise par f à droite du sous-intervalle.

Ainsi pour une méthode avec n sous-intervalles,

on approche l'aire $\int_a^b f(t)\mathrm{d}t$ par

$D_n = \sum_{k=1}^{n} \frac{b-a}{n} f\left(a + k\frac{b-a}{n}\right) = \frac{b-a}{n} \sum_{k=1}^{n} f\left(a + k\frac{b-a}{n}\right)$



Là encore, l'aire a été approchée via celles de rectangles s'appuyant à droite sur le graphe de f.

4.2 Calcul de la dérivée

Méthode

Considérerons que l'on dispose au préalable des valeurs de la fonction en n points de coordonnées (x_i, y_i) où $y_i = f(i)$ . .
Une approximation numérique de la dérivée est obtenue en calculant la pente entre deux points 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.
Comme la pente est calculée entre deux abscisses x_i et x_{i+1}, on associera cette dérivée à l'abscisse située au milieu $x_{new,i} = \frac{x_i + x_{i+1}}{2}$.

Attention !

Il est important de noter qu'il y aura seulement $n-1$ valeurs pour la dérivée.

Exemple :

On donne ici un premier code qui calcule les valeurs de la dérivée qui seront stockées dans un tableau `yp`.

```
deb = 0
fin = 10
n = 101

x = np.linspace(deb, fin, n)
y = np.cos(x)

# préparation des tableaux qui vont recevoir les valeurs
xnew = np.zeros(n-1)
yp = np.zeros(n-1)

# calcul des abscisses et des valeurs de la dérivée
for i in range(n-1):
    xnew[i] = (x[i] + x[i+1]) / 2
    yp[i] = (y[i+1] - y[i]) / (x[i+1] - x[i])

plt.plot(x, y, 'r')
plt.plot(xnew, yp, 'k')
plt.show()
```

Remarque :

La technique ci-dessus pour calculer la dérivée correspond à une méthode de différence finie centrée car elle associe la valeur de la dérivée à une abscisse située au centre entre x_i et $xi+1$.
L'avantage de cette méthode est qu'elle respecte une certaine symétrie entre les abscisses qui permettent le calcul et la position située au centre à laquelle on associe la valeur de la dérivée.
L'inconvénient est qu'elle nécessite de créer un tableau supplémentaire pour stocker les nouvelles abscisses.
Pour éviter cela, il est possible d'associer la valeur de la dérivée à une des deux abscisses déjà connues x_i ou x_{i+1}. Le code devient le suivant :

```
deb = 0
fin = 10
n = 101

x = np.linspace(deb, fin, n)
y = np.cos(x)

# préparation des tableaux qui vont recevoir les valeurs
xnew = np.zeros(n-1)
yp = np.zeros(n-1)

# calcul des abscisses et des valeurs de la dérivée
for i in range(n-1):
    yp[i] = (y[i+1] - y[i]) / (x[i+1] - x[i])

plt.plot(x, y, 'r')
plt.plot(x[0:n-1], yp, 'k')
plt.show()
```

5 Résolutions approchées d'équations différentielles

Extrait du programme

4. Équations différentielles	
Équations différentielles d'ordre 1.	Mettre en œuvre la méthode d'Euler explicite afin de résoudre une équation différentielle d'ordre 1.
Équations différentielles d'ordre supérieur ou égal à 2	Transformer une équation différentielle d'ordre n en un système différentiel de n équations d'ordre 1. Utiliser la fonction **odeint** de la bibliothèque **scipy.integrate** (sa spécification étant fournie).

La méthode d'Euler permet de résoudre de manière approchée des équations différentielles présentées sous la forme

$$\begin{cases} y'(t) = F\big(y(t), t\big), \quad t \in [t_0, t_0 + T] \\ y(t_0) = y_0, \end{cases}$$

où F est une fonction de deux variables (y, t), définie au voisinage de (y_0, t_0). Ce cadre permet d'étudier la plupart des équations différentielles du premier ordre sur un intervalle de temps fini $[t_0, t_0+T]$, munie d'une condition initiale à l'instant t_0.

On rappelle qu'une solution est : une fonction $y : t \in [t_0; t_0 + T] \mapsto y(t) \in \mathbb{R}^p$ vérifiant la condition initiale $y(t_0) = y_0$ et pour tout $t \in [t_0; t_0 + T]$, la relation $y'(t) = F(y(t), t)$. Dans les bons cas, F est définie sur une partie de $\mathbb{R}^p \times [t_0; t_0 + T]$. Les cas les plus usuels correspondent à $p \leqslant 3$ et même très souvent $p = 1$.

5.1 Cas des équations différentielles du premier ordre dans $\mathbb{R}$

Ici, on s'intéresse au cas $p = 1$ (qui nous servira de référence par la suite).

Exemple :

$$\begin{cases} y'(t) = -2\,y(t), \quad t \in [0,1] \\ y(0) = 1, \end{cases} \qquad F : (x,t) \in \mathbb{R} \times \mathbb{R} \mapsto -2x \in \mathbb{R}$$

$$\begin{cases} y'(t) + \cos(t)\,y(t) = \sin(t), \quad t \in [0, 2\pi] \\ y(0) = 0, \end{cases} \qquad F : (x,t) \in \mathbb{R} \times \mathbb{R} \mapsto \sin(t) - x\cos(s) \in \mathbb{R}$$

$$\begin{cases} y'(t) = 2y(t)(1-y(t)), \quad t \in [0, 10] \\ y(0) = 10^{-4}, \end{cases} \qquad F : (x,t) \in \mathbb{R} \times \mathbb{R} \mapsto 2x(1-x) \in \mathbb{R}$$

$$\begin{cases} y'(t) = \sqrt{1 - t(y(t))^2}, \quad t \in [0,1] \\ y(0) = 0, \end{cases} \qquad F : (x,t) \in \{(x,t) \in \mathbb{R} \times \mathbb{R} \mid t\,x^2 < 1\} \mapsto \sqrt{1 - tx^2} \in \mathbb{R}_+$$

5.1.1 Principe de la méthode d'Euler

Graphiquement, la méthode d'Euler consiste à approcher la courbe (le graphe) de la fonction solution y, inconnue, par une ligne brisée que l'on construit point après point (approximation affine par morceaux).
On peut aussi penser une situation physique où un mobile est à un instant t à une position $M(t)$, avec une vitesse $\vec{v}(t) = F(M(t), t)$. Comment approcher sa position à l'instant $t + h$?

$M(t + h) = M(t) + h\vec{v}(t)$.

On imagine que si l'on prend h assez petit, on pourra ainsi tracer la trajectoire comme une succession de lignes brisées approchant assez bien la réalité, la vitesse aux instants utiles étant facile à obtenir en utilisant l'approximation de trajectoire faite.

Méthode

On décompose l'intervalle d'étude en y répartissant $n+1$ instants régulièrement espacés (donc il y a n intervalles de temps) : $t_0 < t_1 < t_2 < \cdots < t_{n-1} < t_n = t_0 + T$.
Deux instants consécutifs sont donc séparés d'un même pas temporel $h = \frac{T}{n}$

- La valeur de la solution en l'instant t_0 est donnée par l'équation : c'est y_0.
  On partira donc du point $M_0\ (t_0, y_0)$
- Pour construire le point suivant, on ne connaît pas la courbe solution entre t_0 et t_1, mais sa tangente au point d'abscisse t_0 (vitesse à la date t_0) est connue grâce à l'équation différentielle. En effet, la pente de cette tangente (la vitesse) vaut

  $$y'(t_0) = F(y(t_0), t_0) = F(y_0, t_0) \text{ qui est calculable.}$$

  On décide d'assimiler la courbe de y sur l'intervalle $[t_0; t_1]$ à sa tangente en t_0 (on suppose la vitesse constante pendant l'intervalle). On en déduit une approximation de $y(t_1)$:

  $$y(t_1) = y(t_0 + h) \simeq y(t_0) + hF(y_0, t_0) = y_0 + hF(y_0, t_0)$$

  On pose donc $y_1 = y_0 + hF(y_0, t_0)$ **et on place le point** $M_1\ (t_1, y_1)$
- On passe de l'instant t_1 à l'instant t_2 de manière similaire. En l'instant t_2, la tangente à la courbe solution a pour pente $y'(t_1) = F(y(t_1), t_1) \simeq F(y_1, t_1)$
  En assimilant la courbe de y entre t_1 et t_2 à sa tangente en t_1, on obtient

  $$y(t_2) = y(t_1 + h) \simeq y(t_1) + hF(y(t_1), t_1) \simeq y_1 + hF(y_1, t_1)$$

  On pose donc $y_2 = y_1 + hF(y_1, t_1)$ **et on place le point** $M_2\ (t_2, y_2)$
- Et ainsi de suite jusqu'à arriver à t_n :

$$\begin{aligned} \forall k \in [\![0, n-1]\!], \quad y(t_{k+1}) = y(t_k + h) &\simeq y(t_k) + hy'(t_k) \\ &= y(t_k) + hF\big(y(t_k), t_k\big) \\ &\simeq y_k + hF(y_k, t_k) \end{aligned}$$

Bilan

Pour résoudre de manière approchée le problème de Cauchy

$$\begin{cases} y'(t) = F\big(y(t), t\big), \quad t \in [t_0, t_0 + T], \\ y(t_0) = y_0, \end{cases}$$

la méthode d'Euler consiste à introduire des instants régulièrement espacés

$$t_k = t_0 + kh \quad \text{où } h = \frac{T}{n} \text{ est le pas de la méthode ,}$$

puis d'approcher les valeurs de la solution aux instants t_k par les nombres y_k calculés de manière récurrente par :

$$\forall k \in [\![0, n-1]\!], \quad y_{k+1} = y_k + hF(y_k, t_k)$$

Les points $M_k(t_k, y_k)$ dessinent une ligne brisée qui approche la courbe solution sur l'intervalle $[t_0, t_0 + T]$.

Remarque :

Qualité de l'approximation
On a l'impression que plus le pas sera petit, mieux la ligne brisée sera ajustée à la courbe de la solution exacte (autrement dit, meilleure sera l'approximation de la solution). Pour étudier ce phénomène, on peut observer comment se comporte un exemple où l'on sait résoudre explicitement l'équation différentielle.

Prenons l'équation $y' = y$ sur $[0, 1]$ avec la condition initiale $y(0) = 1$. On sait que la solution exacte est $y : t \mapsto \exp(t)$ et qu'à l'instant final $y(1) = \exp(1)$.

La valeur y_n à l'instant final fournit donc une approximation de exp. Voici les résultats obtenus par la méthode d'Euler en doublant à chaque fois le nombre de pas n :

n	y_n	Erreur
1	2.000000000000000	-0.718282
2	2.250000000000000	-0.468282
4	2.441406250000000	-0.276876
8	2.565784513950348	-0.152497
16	2.637928497366600	-0.080353
32	2.676990129378183	-0.041292
64	2.697344952565100	-0.020937
128	2.707739019688019	-0.010543
256	2.712991624253433	-0.005290
512	2.715632000168990	-0.002650
1024	2.716955729466436	-0.001326

(numpy donne exp $\simeq$ `2.718281828459045`)

À partir du milieu du tableau, à chaque fois que le nombre de pas est multiplié par 2, l'erreur est approximativement divisée par 2.
L'erreur semble à peu près proportionnelle au pas de la méthode, donc à $\frac{1}{n}$.

Sur un intervalle d'étude $I = [t_0, t_0 + T]$ fixé, si F est de classe $\mathscr{C}^1$ et que la solution y est définie jusqu'à l'instant $t_0 + T$, quand on fait tendre le nombre de pas n vers l'infini, on a, en notant $y_{N,k}$ l'approximation au temps $t_0 + kh$ avec un pas $h = \frac{T}{N}$, l'approximation $y_{n,n}$ de la solution à la date $t_0 + T = t_0 + nh$ vérifie :

$$\left| y(t_0 + T) - y_{n,n} \right| = O\left(\frac{1}{n}\right)$$

$$\text{i.e.} \exists C \in \mathbb{R}_+, tq \forall n > 0, \left| y(t_0 + T) - y_{n,n} \right| \leqslant \frac{C}{n}$$

On dit que **la méthode d'Euler est une méthode d'ordre 1.**

5.2 Équations différentielles vectorielles du premier ordre

La méthode d'Euler fonctionne à l'identique pour les systèmes différentiels, comportant plusieurs fonctions à valeurs réelles, inconnues $y_1, y_2, \ldots, y_p$ reliées entre elles par un système d'équations différentielles.

On les regroupe dans un vecteur inconnu...

$$X(t) = \begin{pmatrix} y_1(t) \\ y_2(t) \\ \cdots \\ y_p(t) \end{pmatrix}$$

... ce qui permet de réécrire les équations différentielles sous la forme

$$X'(t) = F\big(X(t), t\big), \quad t \in [t_0, t_0 + T]$$

où F est une fonction de deux variables (Z, t), définie sur $\mathbb{R}^p \times [t_0, t_0 + T]$ et à valeurs dans $\mathbb{R}^p$.

Dans ce contexte, on a besoin de p conditions initiales (réelles) en l'instant t_0 :

$y_1(t_0) = a_1, \quad y_2(t_0) = a_2, \quad \ldots, \quad y_p(t_0) = a_p$
soit encore une condition initiale dans $\mathbb{R}^p$ à l'instant t_0 i.e. $X(t_0) = X_0 = (a_1; \ldots; a_p)$
On procède comme précédemment : l'intervalle $I = [t_0, t_0 + T]$ est subdivisé de la même manière et la solution $t \mapsto X(t)$ est approchée par les valeurs discrètes $X_0, X_1, \ldots, X_n$ vérifiant

$$X_0 \text{ donné (C.I.) et} \quad \forall k \in [\![0, n-1]\!]$$

$$X_{k+1} = X_k + hF(X_k, t_k)$$

5.3 Équations différentielles d'ordre 2

Les équations différentielles d'ordre 2 (ou plus) peuvent se traiter en se réécrivant sous forme d'un système différentiel vectoriel d'ordre 1. On pourra travailler dans l'espace des phases (comme en physique lorsque l'on trace un portrait de phase). Plus précisément, on prend comme inconnues non plus seulement la fonction mais la fonction et toutes ses dérivées successives jusqu'à l'ordre de l'équation différentielle moins 1. On admet que les solutions de l'équation différentielle initiale sont exactement la partie "non dérivée" des vecteurs solutions de l'équation du premier ordre.
Dans le cas d'une équation différentielle d'ordre 2, d'inconnue y à valeurs réelles il suffit d'introduire le vecteur $X(t) = \begin{pmatrix} y(t) \\ y'(t) \end{pmatrix}$ et d'exprimer X' en fonction de t et de X.
La condition initiale $X(0) = X_0$ revient à donner la valeur en t_0 de la fonction inconnue y ainsi que de sa dérivée y' à la même date.

Exemple :

Un pendule libre non amorti vérifie comme équation différentielle $\ddot{\theta} + \omega_0^2 \sin(\theta) = 0$. Il se traite en posant $X = \begin{pmatrix} \theta \\ \alpha \end{pmatrix}$, α représentant la vitesse angulaire $\dot{\theta}$ du pendule. On a alors

$$X' = \begin{pmatrix} \dot{\theta} \\ \dot{\alpha} \end{pmatrix} = \begin{pmatrix} \dot{\theta} \\ \ddot{\theta} \end{pmatrix} = \begin{pmatrix} \dot{\theta} \\ -\omega_0^2 \sin(\theta) \end{pmatrix} = \begin{pmatrix} \alpha \\ -\omega_0^2 \sin(\theta) \end{pmatrix} \quad \text{donc ici} \quad F(X,t) = \begin{pmatrix} \alpha \\ -\omega_0^2 \sin(\theta) \end{pmatrix}$$

La méthode d'Euler s'écrit donc dans ce cas :

$$\theta_0, \alpha_0 \text{ conditions initiales données et } \begin{cases} \theta_{k+1} = \theta_k + h\ \alpha_k \\ \alpha_{k+1} = \alpha_k - h\ \omega_0^2 \sin(\theta_k) \end{cases}$$

Ainsi, après application de la méthode d'Euler, on peut facilement tracer θ en fonction de la date t, mais aussi $\dot{\theta}$ en fonction de θ, i.e. la courbe dans l'espace des phases.

```
omega0 = 1

def F(X: np.array, t: float) -> np.array :
    return np.array([X[1], -omega0**2*sin(X[0]) ])


def Euler(X0: np.array, F: callable, T: float, n: int) -> tuple:
    h = T/n
    theta = [X0[0]]
    X = X0
    t = [0]
    for i in range(n):
        X = X + h*F(X, t[-1])
        t.append(t[-1] + h)
        theta.append(X[0])
    return t, theta


X0 = np.array([3, 0])
T = 20
n = 100000

t, theta = Euler(X0, F, T, n)
```

5.4 fonction odeint

Des méthodes plus efficaces que la methode d'Euler existent pour résoudre des problèmes de Cauchy. On peut citer, par exemple, la méthode de Runge et Kutta. Il s'agit d'un schéma d'intégration numérique qui

s'appuye sur un découpage plus astucieux du temps. Nous ne rentrerons pas dans les détails. Nous utilserons la fonction (odeint) du module `scipy.integrate` (odeint : Ordinary Differential Equation INTegration). Sa spécification doit vous être fournie. On donne ci-dessous un exemple d'utilisation de cette fonction pour la résolution de l'équation différentielle

$$\begin{cases} y'(t) = -y + \exp(-t**2), & t \in [0, 10] \\ y(t_0) = 0, \end{cases}$$

```
from scipy.integrate import odeint

T = 10
n = 100000
t = np.linspace(0, T, n)
X0 = np.array([0])

def F(x, t):
    return -x + np.exp(-t**2)

# Résolution numérique
sol = odeint(F, X0 , t)

plt.plot(t, sol)
plt.xlabel('$t$')
plt.ylabel('$y$')
plt.grid()
plt.show()
```

6 Statistiques et probabilités

Extrait du programme

5. Probabilité - statistiques	
Variable aléatoire.	Utiliser les fonctions de base des bibliothèques **random** et/ou **numpy** (leurs spécifications étant fournies) pour réaliser des tirages d'une variable aléatoire. Utiliser la fonction **hist** de la bibliothèque **matplotlib.pyplot** (sa spécification étant fournie)
	pour représenter les résultats d'un ensemble de tirages d'une variable aléatoire. Déterminer la moyenne et l'écart-type d'un ensemble de tirages d'une variable aléatoire.

Rappelons qu'en science, chaque expérience est un processus souvent complexe qui fait s'entremêler plusieurs processus. De cette complexité résulte une variabilité des résultats. Cette variabilité est naturelle et fait partie de la mesure. Il ne faut pas chercher à la supprimer car elle contient beaucoup d'informations sur les processus physiques.

Définitions

La quantification de la variabilité d'une mesure x d'une grandeur est appelée incertitude-type et notée $u(x)$. Par définition, l'incertitude-type correspond à l'écart-type de la distribution des données issues d'une répétition de la mesure.
Soit un ensemble de N mesures notées x_i avec i allant de 1 à N. On définit la moyenne $\overline{x}$ de l'ensemble, qui nous permet de définir l'écart-type, et donc l'incertitude-type, grâce aux relations :

$$\overline{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$$

et

$$u(x) = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2}$$

Remarque :

On fera à ce stade deux remarques :

- Pour estimer l'incertitude-type du résultat d'une unique mesure, il faut donc répéter un grand nombre de fois le processus de mesure. Cette répétition et les valeurs supplémentaires servent uniquement à estimer la variabilité du processus de mesure.
- L'incertitude-type est l'estimation d'une variabilité qui est unique à chaque processus de mesure. Il est donc naturel que deux personnes réalisant exactement la même expérience aient une variabilité, et donc une incertitude-type, différente.

Attention !

Certaines expériences n'ont pas de variabilité observée. C'est le cas lorsque l'on mesure naïvement la taille d'un objet avec la même règle graduée. La mesure peut être répétée plusieurs fois, on obtiendra toujours le même résulat. Reproduire la mesure n'apporte pas d'information. Cette absence de variabilité observée n'implique pas une absence de variabilité. Cela signifie juste qu'à l'échelle de cette expérience, avec l'appareil de mesure choisi, la variabilité est plus faible que la précision de la mesure. Ce phénomène n'est pas uniquement lié à l'appareil de mesure. En effet, selon les conditions expérimentales, il n'est parfois pas matériellement possible (ou souhaité) de reproduire le processus de mesure. Dans ce cas, une seule valeur est accessible et il faut tout de même estimer son incertitude-type. Il faut donc estimer théoriquement la variabilité de la mesure sans l'observer. Nécessairement, cela est possible sous certaines hypothèses qui ne seront pas forcément adaptées à toutes les expériences.

Méthode

Lors d'une mesure sans variabilité observée, on estime la plus petite plage dans laquelle l'expérimentateur est certain de trouver la valeur recherchée. On note $\overline{x}$ la valeur centrale de cette plage et Δ sa demi-largeur. Autrement dit, l'expérimentateur est **certain** de trouver la valeur recherchée dans l'intervalle $[\overline{x}-\Delta, \overline{x}+\Delta]$. Dans ce cas, le résultat de la mesure est $\overline{x}$ et $u(\overline{x}) = \frac{\Delta}{\sqrt{3}}$.

- L'intervalle Δ doit être pris le plus faible possible selon les critères personnels de l'expérimentateur et selon les conditions de l'expérience. **Il ne doit pas y avoir de règle générale, c'est le bon sens qui domine** .

6.1 Application en optique

On dispose d'une lentille convergente de distance focale f' parfaitement connue dont le centre optique est repéré par le point O. On place un objet réel en A, la lentille forme son image réelle en A' (si $D > 4f'$...). On rappelle que la relation de conjugaison de Descartes donne : $\overline{OA'} = \frac{\overline{OA}f'}{\overline{OA}+f'}$.

Si l'on souhaite comparer la valeur de $\overline{OA'}$ mesurée sur le banc optique à celle calculée via la relation de conjugaison, il faut être capable d'estimer les incertitudes types sur :

- la valeur de $\overline{OA'}$ mesurée expérimentalement : notée $\overline{OA'_{exp}}$.
- la valeur de $\overline{OA}$ mesurée expérimentalement et donc, la valeur de $\overline{OA'}$ calculée théoriquement : notée $\overline{OA'_{th}}$.

Bien qu'il existe des méthodes théoriques pour calculer l'incertitude-type sur $\overline{OA'_{th}}$, nous la mesurerons à l'aide de l'outil numérique.

Méthode

L'idée est :

- de générer un très grand nombre (N) de tirages aléatoires de $\overline{OA}$ répartis uniformément dans l'intervalle $\left[\overline{OA}-\Delta, \overline{OA}+\Delta\right]$
- en déduire N valeurs de $\overline{OA'_{th}}$
- de mesurer l'écart-type des résultats

On parle de méthode de **Monte-Carlo** dont voici un exemple de code :

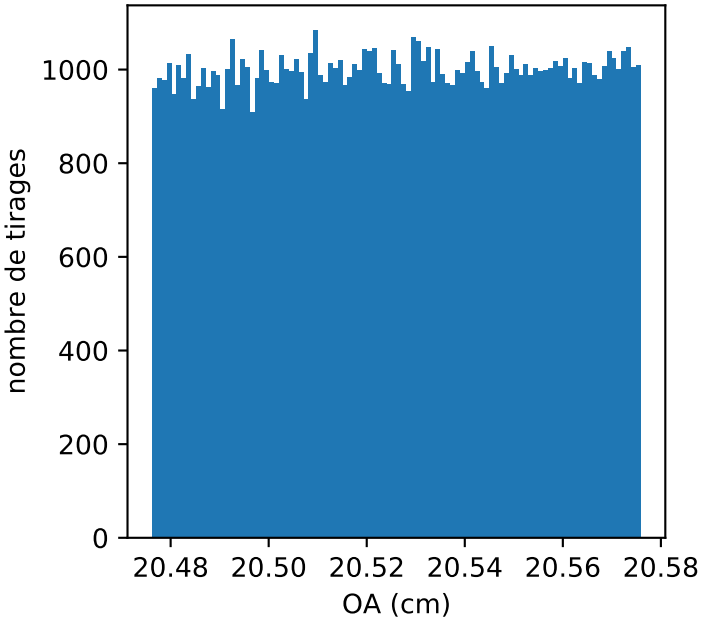
```
import random as rd

#distances en cm
Delta = 0.5
OA = 65
f = 30

def calcule_OA_prime(OA, f):
    return OA*f/(OA+f)

#tirage de N valeurs et calcul des positions de OA
N = 100000
L_OA_prime = []

for i in range(N):
    OA_tiree = rd.uniform(OA-Delta, OA+Delta)
    L_OA_prime.append(calcule_OA_prime(OA_tiree, f))

plt.figure('histogramme', figsize = [4, 4])
plt.axes([0.2, 0.15, 0.7, 0.7])
plt.hist(L_OA_prime, bins = 100)
plt.xlabel('OA (cm)')
plt.ylabel('nombre de tirages')
plt.show()

moyenne_OA_prime = np.mean(L_OA_prime)
ecart_type_OA_prime = np.std(L_OA_prime, ddof = 1)

print("moyenne OA'", moyenne_OA_prime, 'cm')
print("ecart-type OA'", ecart_type_OA_prime, 'cm')
```

On remarque, l'utilisation des fonctions

- `uniform` du module `random`, cette fonction permet de faire des tirages aléatoires uniformes dans un intervalle.
- `hist` du module `matplotlib.pyplot`, cette fonction permet de tracer un histogramme.
- `mean` du module `numpy`, cette fonction renvoie la valeur moyenne du tableau ou de la liste passé en argument.
- `std` du module `numpy`, cette fonction renvoie l'écart-type du tableau ou de la liste passé en argument. Attention, à ne pas oublier l'argument optionnel `ddof = 1` qui précise que c'est bien $\frac{1}{N-1}$ qui apparaît dans la définition de l'écart-type et non pas $\frac{1}{N}$.

L'exécution du code ci-dessus provoque les affichages suivants :

```
"moyenne OA'" 20.526134376201377 cm
"ecart-type OA'" 0.028790956170895127 cm
```



Pour pouvoir comparer les deux mesures entre elles, il faut un critère quantitatif pour indiquer si ces deux mesures sont considérées comme compatibles ou incompatibles.

Définitions

L'écart normalisé E_N entre deux processus de mesure donnant les valeurs m_1 et m_2 et d'incertitudes types $u(m_1)$ et $u(m_2)$ est défini par

$$E_N = \frac{|m_1 - m_2|}{\sqrt{u(m_1)^2 + u(m_2)^2}}$$

Par convention, on qualifie souvent deux résultats de compatibles si leur écart normalisé vérifie la propriété $E_N \leqslant 2$.

L'écart normalisé s'appelle aussi parfois « z-score ». Ce seuil à 2 est d'origine historique. On le retrouve dans de nombreux champs scientifiques, comme la médecine, la pharmacie, la biologie, la psychologie, l'économie, l'écologie, etc. Ce seuil peut différer selon le domaine : par exemple, pour démontrer l'existence d'une nouvelle particule en physique subatomique, il faut atteindre un seuil de 5.

7 Régression linéaire

Extrait du programme

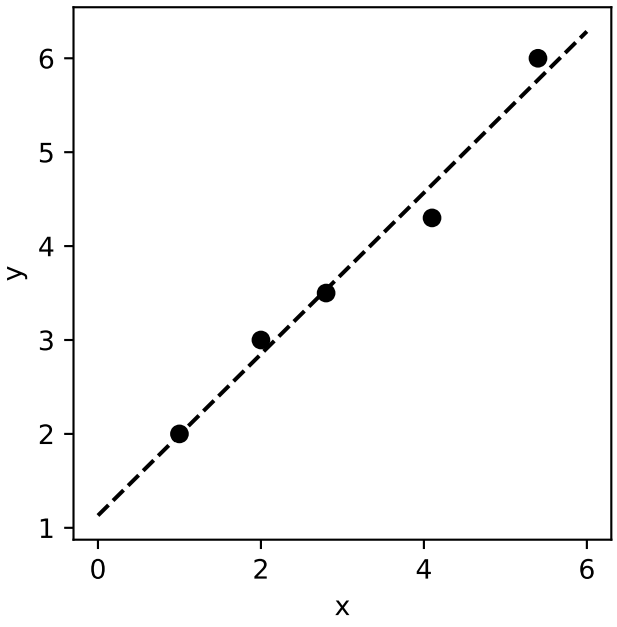
Régression linéaire.	Utiliser la fonction **polyfit** de la bibliothèque **numpy** (sa spécification étant fournie) pour exploiter des données. Utiliser la fonction **random.normal** de la bibliothèque **numpy** (sa spécification étant fournie) pour simuler un processus aléatoire.

La fonction `np.polyfit` permet de faire des regressions linéaires. Sa documentation doit vous être fournie. Voici un exemple d'utilisation.

```
x = [1, 2, 2.8, 4.1, 5.4]
y = [2, 3, 3.5, 4.3, 6]

a, b = np.polyfit(x, y, 1)

x_fit = np.linspace(0, 6, 10)
y_fit = a * x_fit + b

plt.figure('régression linéaire', figsize = [4, 4])
plt.axes([0.2, 0.15, 0.7, 0.7])
plt.plot(x, y, 'ko')
plt.plot(x_fit, y_fit, 'k--')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

Remarque :

la fonction `polyfit` prend comme arguments les données en abscisse et en ordonnées. Il ne faut pas oublier le troisième argument (`1`) qui précise que l'on fait une régression par un polynôme d'ordre 1.



7.1 Prise en compte des incertitudes

Prenons l'exemple de la mesure de la vitesse du son c dans l'air. On dispose d'un émetteur et d'un récepteur d'onde ultrasonore distants de d, on mesure le temps de vol t de l'onde entre l'émetteur et le récepteur. Les grandeurs c, d et t sont liés par la relation $d = d_0 + ct$. Les inconnues c et d_0 peuvent être déterminées à l'aide de la régression linéaire de d en fonction de t. Cependant, comme les mesures effectuées possèdent une variabilité, la vitesse c sera connue avec une certaine incertitude qu'il faut évaluer. Pour cela, on va utiliser une méthode dite de Monte-Carlo :

Méthode

- Pour chaque point de mesure (d_i, t_i), on tire aléatoirement un nouveau point $\left(d_i', t_i'\right)$ avec $d_i' \in [d_i - \Delta_d, d_i + \Delta_d]$ et $t_i' \in [t_i - \Delta_t, \Delta t_i + \Delta_t]$. Où Δ_d et Δ_t sont les demi-largeurs des plages de variabilité de d et t.
- On effectue une régression linéaire sur les points $\left(d_i', t_i'\right)$ et on stocke la pente c_i.
- On itère ce processus un très grand nombre de fois (typiquement, 10 000 ou 100 000 fois) pour calculer la moyenne, $\overline{c}$ des c_i et leur écart-type $u(\overline{c})$. On trouve alors la vitesse des ondes et l'incertitude-type associée à cette mesure.

Voici un exemple de code :

```
lst_d = [...]
lst_t = [...]

Delta_d = ...
Delta_t =  ...

N = 100000

## Fonctions

def tirage_un_point(t, d, Delta_t, Delta_d):
    ''' Donne de manière aléatoire les coordonées d'un point
    comprises entre t +/- Delta_t et d +/- Delta_d
    entrées : d un flottant
              t un flottant
              Delta_d un flottant
              Delta_t un flottant
    sorties : x un flottant
              y un flottant'''

    x = rd.uniform(t - Delta_t, t + Delta_t)
    y = rd.uniform(d - Delta_d, d + Delta_d)
    return x, y

def creation_points_aleatoires(lst_t, lst_d, Delta_t, Delta_d):
    '''Renvoie deux listes de même taille que lst_d et lst_t
    consituées de points tirée aléatoirement
    entrées : lst_d une liste de flottants
              lst_t une liste flottants
              Delta_d un flottant
              Delta_t un flottant
    sorties : lst_x une liste de flottants
              lst_y y une liste de flottants'''
    n = len(lst_d)
    lst_x = []
    lst_y = []
```

```
    for i in range(n):
        x, y = tirage_un_point(lst_t[i], lst_d[i], Delta_t, Delta_d)
        lst_x.append(x)
        lst_y.append(y)
    return lst_x, lst_y

def calcule_une_vitesse(lst_x, lst_y):
    ''' Calcule une estimation de la vitesse des ondes en faisant une
    régression linéaire
    entrées : lst_x une liste de flottants
              lst_y y une liste de flottants
    sorties : c un flottant '''

    c, d_0 = np.polyfit(lst_x, lst_y, 1)
    return c, d_0

def creation_liste_vitesses(lst_t, lst_d, Delta_t, Delta_d, N):
    '''Donne une liste de N estimations de la vitesse des ondes
    entrées : lst_d une liste de flottants
              lst_t une liste flottants
              Delta_d un flottant
              Delta_t un flottant
              N un entier
    sorties : lst_c une liste de flottants
              lst_d0 une liste de flottans
              '''
    lst_c = []
    lst_d0 = []
    for i in range(N):
        lst_x, lst_y = creation_points_aleatoires(lst_t, lst_d, Delta_t, Delta_d)
        c, d0 = calcule_une_vitesse(lst_x, lst_y)
        lst_c.append(c)
        lst_d0.append(d0)
    return lst_c, lst_d0

## Execution

lst_c, lst_d0 = creation_liste_vitesses(lst_t, lst_d, Delta_t, Delta_d, N)

c = np.mean(lst_c)
u_c = np.mean(lst_d0, ddof = 1)

print('vitesse moyenne : ', c, 'm/s')
print('Incertitude-type:', u_c, 'm/s')
```

8 Équations aux dérivées partielles

Extrait du programme

2. Équations différentielles et équations aux dérivées partielles	
Équation de diffusion à une dimension.	Mettre en œuvre une méthode des différences finies explicite pour résoudre l'équation de diffusion à une dimension en régime variable.

On étudie la diffusion thermique à une dimension le long de l'axe (Ox), en l'absence de sources internes. Le champ de température $T(t,x)$ dans le solide vérifie l'équation de la diffusion à une dimension :

$$\frac{\partial T}{\partial t} = D\frac{\partial^2 T}{\partial x^2}$$

où D est la diffusivité thermique.

Méthode

- Au lieu d'étudier le champ de température $T(t,x)$ à tout instant de la plage de temps $[0, t_{\max}]$, on va l'étudier à des instants discrets équirépartis $t_i = i\mathrm{d}t$ avec $i \in [\![0, M]\!]$. Ainsi, $\mathrm{d}t = \frac{t_{\max}}{M}$ est appelé le pas de temps temporel. $\mathrm{d}t$ (et donc M) est choisi *a priori* par l'utilisateur.
- Au lieu d'étudier le champ de température $T(t,x)$ en tout point du domaine $[0, L]$, on va l'étudier en des points abscisses discrètes équiréparties $x_i = i\mathrm{d}x$ avec $i \in [\![0, N]\!]$. Ainsi, $\mathrm{d}x = \frac{L}{N}$ est appelé le pas de temps spatial. $\mathrm{d}x$ (et donc N) est choisi *a priori* par l'utilisateur.
- Finalement, la fonction T est échantillonnées en t et en x, et la représente sous la forme d'un tableau `numpy` à deux dimensions noté `T`. On a donc `T[i, j]` $= T(i*dt, j*dx)$. Chaque ligne du tableau contient les valeurs de T à un instant donné. Chaque colonne contient les valeurs de T à une abscisse donnée.

Comme nous l'avons vu plus haut, à l'ordre 1, la dérivée $\frac{\partial T}{\partial t}$ peut être estimée par `(T[i+1,j]-T[i,j])/dt`.
De même, un développement de Taylor spatial de $T(t,x)$ en $x+\mathrm{d}x$ et en $x-\mathrm{d}x$ à l'ordre 2 donne[2] :

$$\frac{\partial^2 T}{\partial x^2} \simeq \frac{T(t,x+\mathrm{d}x)+T(t,x-\mathrm{d}x)-2T(t,x)}{\mathrm{d}x^2}$$

Ainsi, on estime $\frac{\partial^2 T}{\partial x^2}$ par `(T[i, j+1]+T[i, j-1] -2*T[i, j])/dx**2`.
En réinjectant ces deux relations dans l'équation de diffusion, on trouve la relation de récurrence :
`T[i+1, j] = T[i, j] + C*(T[i, j+1]-T[i, j-1]-2*T[i, j])` où `C = D*dt/dx**2`.
Connaissant le champ de température à t_i et les conditions aux limites, on peut donc trouver le champ de température à l'instant t_{i+1}.

Attention !

Ce schéma d'intégration numérique ne fonctionne que si $C < 0,5$.

Exemple :

```
# Paramètres physiques

D = 0.54e-6                     # Diffusivité thermique en m2/s
L = 1                           # Epaisseur de l'échantillon en m
t_tot = 400000                  # Durée totale en s

# Paramètres de la résolution
```

2. Ce développement est également à l'origine de la méthode d'intégration de Verlet, très utilisé pour les systèmes mécaniques.

```
N = 100                          # Nombre d'intervalles en abscisses
dx = L/N                         # Pas en abscisse
x = np.linspace(0, L, N+1)       # Plage d'abscisse
M = 5000                         # Nombre d'intervalles de temps
dt = t_tot/M                     # Pas de temps

# Conditions initiales et aux limites

T0 = 293                           # Conditions aux limites en x = 0
TL = 273                           # Condition aux limites en x = L
Tinitiale = 293                    # Température initiale


T = np.zeros((M+1,N+1))             # Initialisation du tableau des températures T(ti,xk)
T[0, :] = Tinitiale                # Première ligne (indice i = 0) : température initiale
    des points xk

C = D*dt/dx**2


def Resolution_eqdiff(T):
    for i in range (0, M):
        T[i+1, 0] = T0             # Condition aux limites en x = 0
        T[i+1, N] = TL           # Condition aux limites en x = L
        for k in range(1, N):
            T[i+1,k] = T[i,k] + C*(T[i,k+1] + T[i,k-1] - 2*T[i,k])
    return T
```

9 Transformée de Fourier discrète (PSI uniquement)

Extrait du programme

Outils numériques	Capacités exigibles
Transformée de Fourier discrète.	Calculer la transformée de Fourier discrète d'un signal à valeurs réelles en utilisant la fonction **rfft** de la bibliothèque **numpy.fft** (sa spécification étant donnée).

Un signal unidimensionnel, par exemple un signal sonore, peut être vu comme une fonction définie dans le domaine temporel :

$$x : t \to x(t)$$

Dans le cas du traitement numérique du signal, ce dernier n'est pas continu dans le temps, mais échantillonné. Le signal échantillonné est obtenu en effectuant le produit du signal $x(t)$ par un peigne de Dirac de période T_e :

$$x_e(t) = x(t) \sum_{k=-\infty}^{+\infty} \delta(t - kT_e)$$

L'idée de la transformée de Fourier discrète est de transformer une liste de valeurs discrètes d'une fonction temporelle en une liste de valeurs discrètes de son spectre en amplitude. Si le signal temporel est échantillonné aux instants $t_n = nT_e$ avec $n \in [\![1, (N_e - 1)]\!]$, alors le spectre en amplitude est échantillonné aux fréquences $f_k = k\frac{f_e}{N_e}$ avec $k \in [\![0, E(\frac{N_e}{2})]\!]$ En conséquence, une sinusoïde présente dans le signal ne donne une raie fine et de bonne hauteur uniquement si sa fréquence correspond à $k\frac{f_e}{N_e}$

Attention !

Si on crée les instants des échantillons temporels avec `np.linspace(0, Ne*Te, Ne)`, le pas d'échantillonnage n'est pas T_e mais $T_e\frac{N_e}{N_e-1}$, car les bornes 0 et N_eT_e sont incluses. Les fréquences du spectre seraient donc $k\frac{N_e-1}{N_e^2T_e}$, avec $k \in [\![0, E(\frac{N_e}{2})]\!]$. Pour que les instants des échantillons temporels soient bien $t_n = nT_e$ avec $n \in [\![1, (N_e - 1)$, il faut les utiliser l'instruction : `np.linspace(0, (Ne-1)*Te, Ne)`.

La fonction `rfft` du module `numpy.fft` permet de calculer efficacement la transformée de Fourier d'un signal échantillonné.

Attention !

Il faut penser à normaliser le tableau renvoyé par la fonction `rfft` en divisant chaque terme par `N_e//2+1`.

Exemple :

Voici ci-dessous un exemple de code pour le calcul de la transformé de Fourier de

$$x(t) = 15\sin(2\pi t) + 15/3\sin(6\pi t) + 15/5\sin(10\pi t)$$

```
from numpy.fft import rfft

A = 15
T = 1
Tmes = 5*T
Ne = 1000

t = np.linspace(0, Tmes, Num)

def x(t, T):
    return A*np.sin(2*np.pi*t/T)+A/3*np.sin(2*np.pi*3*t/T)+A/5*np.sin(2*np.pi*5*t/T)
```

```
fft = np.fft.rfft(x(t, T))/(Ne//2+1)

freq_k = np.array([i/Tmes for i in range(len(fft))])


plt.figure()
plt.plot(freq_k[:Ne//10+1],np.abs(fft)[:Ne//10+1],'-x')
plt.xlabel('fréquence')
plt.ylabel('amplitude')
plt.grid('on')
plt.show()
```

L'exécution provoque l'affichage suivant :