Outils numériques

Le cours

1 Tableaux NumPy 2

2 Outils graphiques 5

3 Tirages d'une variable aléatoire 8

4 Régression linéaire 11

5 Méthode dichotomique 12

6 Calcul numérique d'une dérivée 14

7 Calcul numérique d'une intégrale 16

8 Équation différentielle d'ordre 1 18

9 Équation différentielle d'ordre supérieur à 1 21

Cours

1 Tableaux NumPy

A Qu'est-ce qu'un tableau NumPy ?

Un tableau NumPy est un ensemble d'éléments, ceux-ci étant du même type (entier, flottant, booléen, ...). En physique, nous manipulerons exclusivement des nombres entiers ou flottants puisque l'objectif est d'effectuer des calculs numériques efficacement en manipulant des valeurs prises par des grandeurs physiques.

Il faudra tout d'abord **importer la bibliothèque NumPy** avec la commande **`import numpy as np`**.

Nous n'utiliserons que des **tableaux unidimensionnels** en physique. Pour créer un tel tableau à partir d'une liste de valeurs, il suffit d'utiliser la commande **`np.array([`***élément 1, élément 2, ...***`])`** :

```
import numpy as np

tableau = np.array([1., 2., 3., 4.])

# Affichage du tableau
print(tableau)

# Affichage de l'élément d'indice 1
print("Element d'incide 1 :", tableau[1])
```

⇛
```
[1. 2. 3. 4.]
Element d'incide 1 : 2.0
```

ATTENTION : le 1er élément du tableau est toujours d'indice 0 !

B Comment générer des tableaux rapidement

Cela sera nécessaire par exemple pour créer des valeurs à répartir sur l'axe des abscisses afin représenter graphiquement une fonction, ou bien pour générer une liste de valeurs du paramètre d'une courbe paramétrée (voir chapitre ON2).

B.1 `numpy.arange(`*valeur de début, valeur de fin, pas*`)`

Utile pour générer un tableau de valeurs croissantes avec un pas de temps fixé.

```
import numpy as np

a = np.arange(0,10,2)

print(a)
```

⇛
```
[0 2 4 6 8]
```

ATTENTION : la valeur de fin *(10 ici)* est exclue !

B.2 `numpy.linspace(`*valeur de début, valeur de fin, nombre de valeurs*`)`

```
import numpy as np

b = np.linspace(0,10,11)

print(b)
```

⇛
```
[ 0. 1. 2. 3. 4. 5. 6. 7. 8. 9. 10.]
```

Le pas (écart entre deux valeurs successives) est constant.

(R) *On aurait pu produire le même tableau avec* **`np.arange(0,11,1)`** .

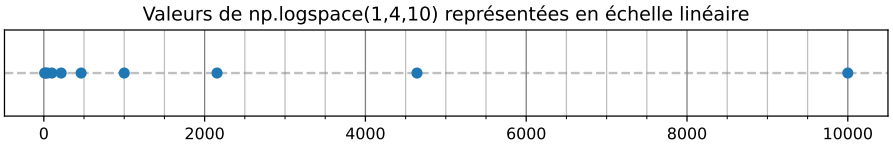
B.3 `numpy.logspace(`*exposant de début, exposant de fin, nombre de valeurs*`)`

Désormais, la répartition des valeurs suit une **échelle logarithmique**. Dans l'exemple ci-desous, les 10 valeurs seront comprises entre 10^1 et 10^4 :

```
1 import numpy as np
2
3 c = np.logspace(1,4,10)
4
5 print(c)
```

⇛
```
[ 10.  21.5443469  46.41588834  100.
 215.443469  464.15888336  1000.  2154.43469003
 4641.58883361  10000. ]
```

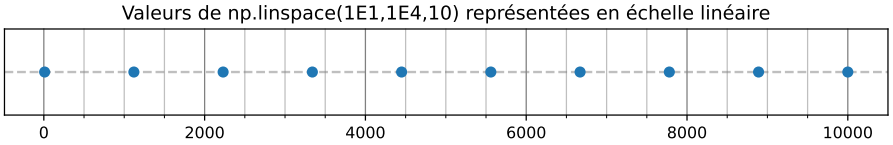
R *Voici la représentation graphique des valeurs de l'exemple ci-dessous suivant un axe horizontal exploitant une échelle linéaire :*



Avec **`d = np.linspace(1E1,1E4,10)`** *qui contient les valeurs ci-dessous (mêmes valeurs de début et de fin) :*

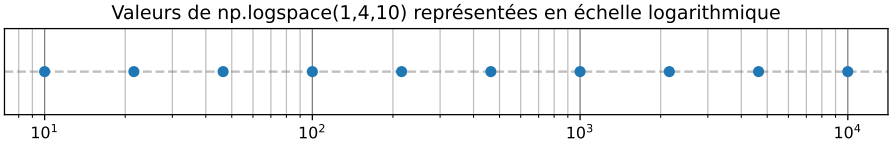
⇛
```
[ 10.  1120.  2230.  3340.  4450.  5560.  6670.  7780.  8890.  10000.]
```

on obtient plutôt cette répartition (le pas entre chaque valeur est constant) :



R ***Quel est l'intérêt de `np.logspace` ?***

Si on visualise les valeurs fournies par **`c = np.logspace(1,4,10)`** *à l'aide d'une* ***échelle logarithmique*** *(et non plus d'une échelle linéaire),les valeurs sont équitablement réparties :*



Avec l'échelle linéaire et `np.linspace`*, il n'y a qu'une seule valeur entre 10 et 1000 : c'est la valeur 10 ! Les 9 autres valeurs sont comprises entre 1000 et 10 000 ! ! Pas très pratique... Alors qu'****avec l'échelle logarithmique et `np.logspace`****, il y a autant de valeurs dans chaque décade* $[10^n, 10^{n+1}]$ *(3 valeurs dans* $[10; 100[$ *par exemple) ce qui permet une meilleure répartition des valeurs étudiées lorsqu'on est amené à manipuler des ordres de grandeurs très différents d'une même variable physique (typiquement, la fréquence des ondes sonores par exemple, comprise entre* 20 Hz *et* 20 kHz*).*

C Application d'une fonction à un tableau

On peut appliquer directement des fonctions à un tableau : la fonction s'appliquera à chaque élément du tableau et renverra ainsi un nouveau tableau NumPy regroupant les valeurs obtenues.

Voici un exemple ci-contre.

```
1 import numpy as np
2
3 def carre(x):
4     return x**2
5
6 a = np.linspace(0,10,11)
7
8 print(a)
9 print(carre(a))
```

⇛
```
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9.  10.]
[ 0.  1.  4.  9.  16.  25.  36.  49.  64.  81.  100.]
```

On peut aussi appliquer directement des **fonctions de la bibliothèque NumPy** : **`np.cos`**, **`np.sin`**, **`np.arcsin`**, **`np.exp`**, **`np.log10`**, **`np.sqrt`** *(fonction racine carrée)*, etc.

```
import numpy as np

b = np.array([1,4,9,16,25,36,49,64,81,100])

print(np.sqrt(b))
```

⇛
```
[ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
```

D Opérations sur un tableau et entre des tableaux

D.1 Addition, soustraction, multiplication, division

À un tableau donné, on peut très simplement additionner ou soustraire une même valeur, ou même multiplier ou diviser par une même valeur chaque élément du tableau de manière très intuitive.

Il est aussi possible d'effectuer, très intuitivement aussi, ces mêmes opérations entre deux tableaux. Ainsi, ladite opération sera effectuée entre chaque élément de même rang :

```
import numpy as np

a = np.array([4,1])
b = np.array([2,3])

print('a =',a,' b =',b)
print('2a = ',2*a)
print('b-1 = ',b-1)
print('a/2 = ',a/2)
print('a + b =', a+b)
print('a - b =', a-b)
print('a x b =', a*b)
print('a / b =', a/b)
```

⇛
```
a = [4 1]   b = [2 3]          a + b = [6 4]
2a = [8 2]                     a - b = [ 2 -2]
b-1 = [1 2]                    a x b = [8 3]
a/2 = [2. 0.5]                 a / b = [2. 0.33333333]
```

D.2 Puissance

On peut aussi porter un tableau à une certaine puissance, ou même se servir du tableau lui-même comme d'un exposant !

```
import numpy as np

a = np.array([2,9])
print('a^2 =',a**2)

b = np.array([3,0.5])
print('a^b =',a**b)
```

⇛
```
a^2 = [4 81]
a^b = [8. 3.]
```

R *D'ailleurs, on pourrait remarquer que* **`np.logspace(1,3,7)`** *et* **`10**np.linspace(1,3,7)`** *fournissent exactement le même tableau ...*

2 Outils graphiques

Capacités exigibles

- Utiliser les fonctions de base de la bibliothèque **matplotlib** pour représenter un nuage de points
- Utiliser les fonctions de base de la bibliothèque **matplotlib** pour tracer la courbe représentative d'une fonction
- Utiliser les fonctions de base de la bibliothèque **matplotlib** pour tracer une courbe plane paramétrée

A Représentation graphique d'un nuage de points

Comment faire ?

Utiliser l'instruction **`plt.scatter(X,Y,marker='+')`** où les **listes `X`** et **`Y`** regroupent respectivement les valeurs d'abscisses et les valeurs d'ordonnées correspondantes classées dans le même ordre.

Il est bien évident que **ces listes doivent avoir la même taille** (même nombre d'éléments).

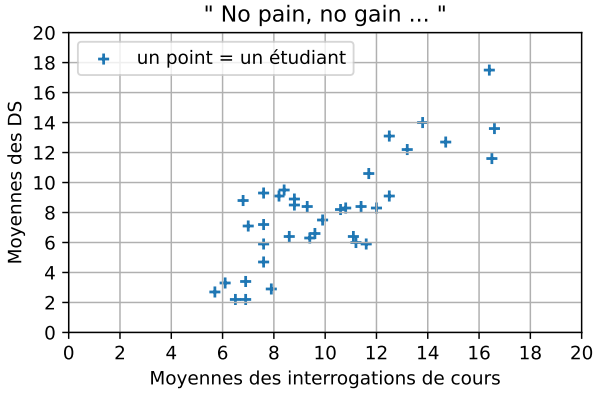
R *Plutôt que des listes, on peut aussi utiliser des tableaux NumPy à une dimension ...*

Exemple : corrélation entre interrogations de cours et réussite aux DS

À l'issue du 1er semestre, il peut être intéressant de représenter les moyennes individuelles des devoirs surveillés en fonction des moyennes individuelles obtenues en interrogation de cours :

```
# ===== IMPORTATION DES BIBLIOTHEQUES ==================
import matplotlib.pyplot as plt

# ===== SAISIE DES DONNEES ==================

# Moyennes des interrogations de cours
X = [5.7,7.6,7.6,9.3,13.2,11.7,6.9,9.4,11.1,11.2,6.8,9.6,11.4,
               8.8,13.8,12.0,8.8,7.0,6.1,8.2,14.7,6.5,7.6,16.6,6.9,12.5,
               8.6,12.5,8.4,11.6,16.5,7.6,16.4,7.9,9.9,10.6,10.8]

# Moyennes des devoirs surveillés
Y = [2.7,9.3,7.2,8.4,12.2,10.6,3.4,6.3,6.4,6.0,8.8,6.6,8.4,8.9,14.0,
            8.3,8.5,7.1,3.3,9.1,12.7,2.2,4.7,13.6,2.2,9.1,6.4,13.1,9.5,
            5.9,11.6,5.9,17.5,2.9,7.5,8.2,8.3]

# ===== TRACE DU NUAGE DE POINTS ==================

plt.scatter(X,Y,marker='+',label='un point = un étudiant')
plt.ylabel('Moyennes des DS')
plt.xlabel('Moyennes des interrogations de cours')
plt.legend() # pour afficher la légende
plt.xlim(0,20) # pour déterminer les valeurs extrêmes d'abscisses
plt.ylim(0,20)# pour déterminer les valeurs extrêmes d'ordonnées
plt.grid() # pour faire apparaître un quadrillage
plt.title('" No pain, no gain ... "')  # choix d'un titre
plt.show() # pour faire apparaître le graphique
```

On obtient la représentation graphique ci-contre.



Conclusion ? Si on apprend régulièrement son cours, on a plutôt tendance à mieux réussir en devoir surveillé.

*(Rappel : il n'est toutefois pas suffisant de bien apprendre son cours, **il faut aussi retravailler efficacement les exercices déjà effectués** ...)*

B Tracé graphique d'une fonction

Comment faire ?

1. Définir la fonction **f** dont on souhaite donner une représentation graphique :

```
def f(x,...):
    return    # préciser la valeur à renvoyer, dépendant de celle de x
```

R *Les pointillés ... représentent d'autres arguments éventuels de la fonction.*

2. Définir un tableau numpy **X** contenant des valeurs d'abscisses avec l'instruction :
X = np.linspace(*valeur de début, valeur de fin, nombre de valeurs***)**

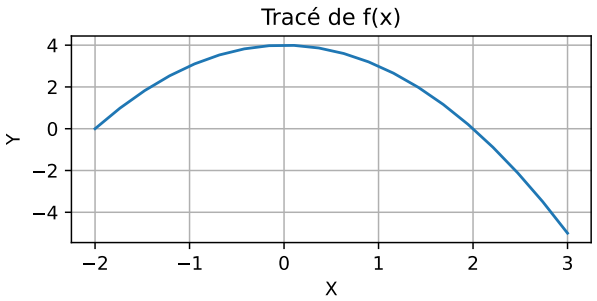
3. Appliquer la fonction **f** à **X** avec l'instruction **Y = f(X,...)**.
On obtient ainsi un tableau numpy **Y** regroupant les valeurs d'ordonnnées correspondant aux différentes abscisses et classées dans le même ordre.

4. Utiliser l'instruction **plt.plot(X,Y)** pour obtenir le tracé.

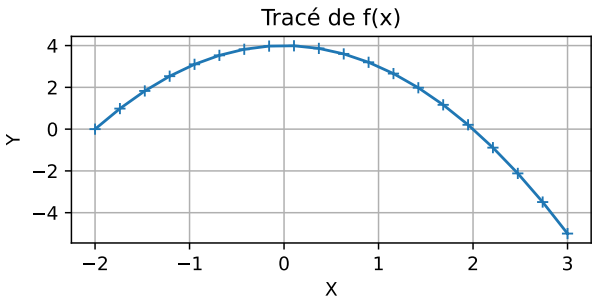
Exemple : tracé de $f(x) = 4 - x^2$

```
1  import numpy as np
2
3  # Définir la fonction à tracer
4  def f(x):
5      return 4-x**2
6
7  X = np.linspace(-2,3,100)  # 100 valeurs en abscisse entre -2 et 3
8  Y = f(X)  # les 100 valeurs en ordonnées correspondantes
9
10 # Tracé
11 plt.plot(X,Y)
12 plt.grid()
13 plt.xlabel("X")
14 plt.ylabel("Y")
15 plt.title("Tracé de f(x)")
16 plt.show()
```

On obtient la représentation graphique ci-contre.



Si on veut faire apparaître les points, on peut remplacer la ligne 11 par celle-ci : **plt.plot(X,Y,marker='+')**.



R *Au lieu d'utiliser* ***plt.plot()****, on utilisera* ***plt.semilogx()*** *pour une obtenir échelle logarithmique en abscisse, ou bien* ***plt.semilogy()*** *pour obtenir une échelle logarithmique en ordonnée ...*

C Tracé d'une courbe plane paramétrée

On parle de **courbe paramétrée** lorsque les abscisses et les ordonnées du tracé à effectuer dépendent de la valeur d'un *paramètre* t. En physique, très souvent, ce paramètre sera la variable de temps. Pour le tracé, on précision deux fonctions $x(t)$ et $y(t)$ représentant respectivement l'abscisse et l'ordonnée de la courbe :

$$\begin{cases} x(t) &= \dots \\ y(t) &= \dots \end{cases}$$

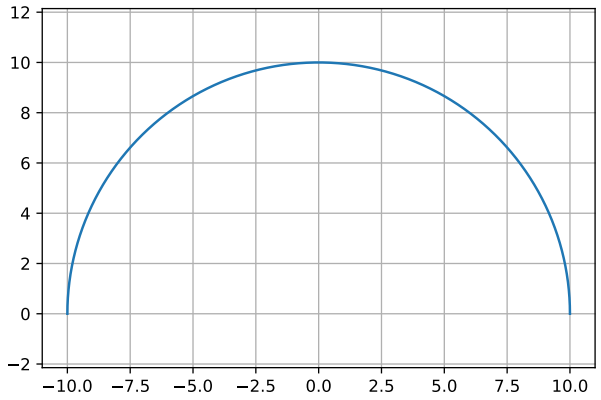
Comment faire ?

1. Définir un tableau numpy `t` contenant les valeurs prises successivement par le paramètre t.
   On se servira de l'instruction : **`t = np.linspace(`***valeur de début, valeur de fin, nombre de valeurs***`)`**
2. Calculer les valeurs de $x(t)$ et $y(t)$. Si besoin, définir des fonctions comme cela a été vu à en **A**.
3. Tracer y en fonction de x comme cela a été vu en **A**.

Exemple : tracé d'un demi-cercle

Ici, plutôt que le temps t, le paramètre est θ, position angulaire par rapport à l'axe Ox *(penser au cercle trigonométrique...)*. Pour tracer un demi-cercle, on peut faire varier θ de 0 à π par exemple :

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  # Rayon du cercle
5  R = 10 # par exemple ...
6
7  # Valeurs prises par le paramètre
8  theta = np.linspace(0, np.pi, 100)
9
10 # Calcul des abscisses et ordonnées
11 x = R*np.cos(theta)
12 y = R*np.sin(theta)
13
14 # Tracé
15 plt.plot(x,y)
16 plt.grid()
17 plt.axis("equal")
18 plt.show()
```



R *L'instruction* **`plt.axis("equal")`** *permet d'imposer la même échelle sur les deux axes. Autrement, le tracé aurait ressemblé à une demi-ellipse plutôt.*

Cours

3 Tirages d'une variable aléatoire

Capacités exigibles

- 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

A Histogramme, moyenne et écart-type

A.1 Tracé d'un histogramme

Pour représenter graphiquement un ensemble de valeurs sous forme d'un histogramme, on aura chargé préalablement le module **pyplot** de la bibliothèque **matplotlib** avec l'instruction :

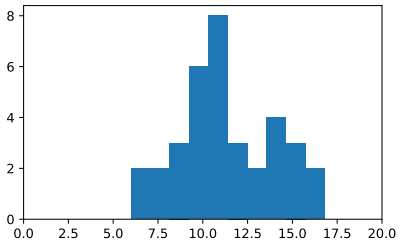
`import matplotlib.pyplot as plt`

Puis, on emploiera l'instruction :

`plt.hist(`*ensemble des valeurs étudiées***`)`**

Voici un exemple avec les moyennes individuelles d'une ancienne promotion sur une année complète :

```
1  import matplotlib.pyplot as plt
2
3  # Moyennes individuelles des étudiants
4  X = [7.4,10.8,6.0,10.6,11.4,15.1,14.1,13.8,15.3,
5       13.8,8.8,10.7,10.2,8.5,13.4,16.2,11.1,9.1,9.8,12.8,
6       9.7,9.6,10.6,14.6,16.8,11.7,11.1,9.6,10.4,9.3,15.2,
7       10.5,7.1,6.2,12.0]
8
9  # Tracé de l'histogramme
10 plt.hist(X)
11 plt.xlim(0,20)
12 plt.show()
```



A.2 Calcul d'une moyenne et d'un écart-type

Il faudra importer la bibliothèque **numpy** avec **`import numpy as np`**, puis :

- pour le calcul de la **moyenne** : **`np.average(`***liste des valeurs étudiées***`)`**
- pour le calcul de l'**écart-type de la population** σ : **`np.std(`***liste des valeurs étudiées***`)`**
- pour le calcul de l'**écart-type de l'échantillon** s_x (ou s_{n-1}) : **`np.std(`***liste des valeurs étudiées***`,ddof=1)`**

Poursuivons l'exemple précédent pour illustrer :

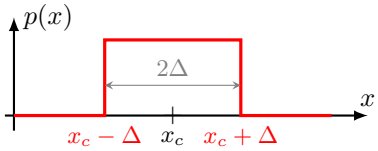
```
13 import numpy as np
14
15 moyenne = np.average(X)
16 sigma = np.std(X)
17 sx = np.std(X,ddof=1)
18
19 print(moyenne,sigma,sx)
```

⇛ `11.23... 2.74... 2.78...`

B Loi uniforme (ou rectangulaire)

Qu'est-ce que c'est ?

Une variable aléatoire X suit une loi rectangulaire de largeur 2Δ centré sur une valeur x_c lorsque la loi de probabilité $p(x)$ prend la forme ci-contre. Il y a donc équiprobabilité d'obtenir un tirage aléatoire dans l'intervalle $[x_c - \Delta, x_c + \Delta]$. En dehors de cet intervalle, la probabilité est nulle.



Tirage aléatoire avec Python

On aura chargé préalablement le module **random** de la bibliothèque **NumPy** avec l'instruction :

```
import numpy.random as rd
```

Si on connaît les valeurs $x_{min} = x_c - \Delta$ et $x_{max} = x_c + \Delta$, on utilisera l'instruction :

```
rd.uniform(xmin, xmax , N)
```

qui génère un tableau NumPy unidimensionnel de N valeurs tirées au sort de manière équiprobable dans l'intervalle $[x_{min}, x_{max}]$.

Si on connaît les valeurs x_c et Δ, on utilisera plutôt l'instruction :

```
rd.uniform(xc - Delta , xc + Delta, N)
```

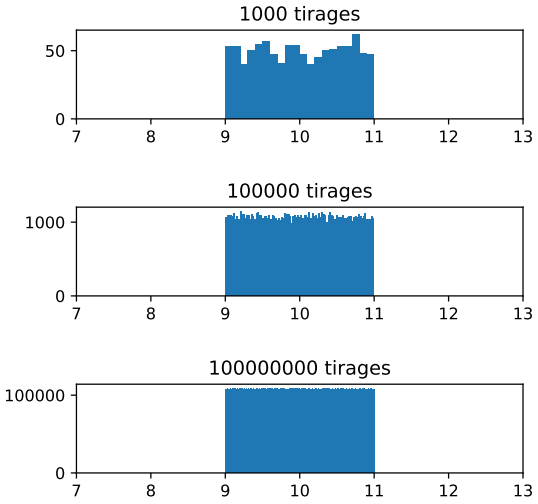
ou de manière équivalente :

```
xc + rd.uniform(-Delta , Delta, N)
```

En effet, `rd.uniform(-Delta , Delta, N)` génèrera un tableau NumPy de N valeurs dans $[-\Delta, \Delta]$. Donc, en additionnant x_c à ce tableau, on obtient bien des valeurs dans $[x_c - \Delta, x_c + \Delta]$.

Exemple

```
1  import matplotlib.pyplot as plt
2  import numpy.random as rd
3
4  N = 1000   # nombre de tirages
5  xmin = 9.
6  xmax = 11.
7
8  a = rd.uniform(xmin,xmax,N)
9
10 # ou bien, de manière équivalente :
11 xc = 10.
12 Delta = 1.
13 a = xc + rd.uniform(-Delta,Delta,N)
14
15
16 # tracé de l'histogramme
17 plt.hist(a, bins='rice')
18 plt.title("%i tirages" %N)
19 plt.show()
```



R *L'option* **`bins='rice'`** *permet d'optimiser le choix du nombre de barres à afficher.*

Poursuivons avec le calcul de la moyenne et de s_x :

```
20 import numpy as np
21
22 moyenne = np.average(a)
23 sx = np.std(a,ddof=1)
24
25 print("Pour",N,"tirages :")
26 print("moyenne =",moyenne)
27 print("sx =",sx)
```

⇛

```
Pour 1000 tirages :
moyenne = 9.982311156469073
sx = 0.5793309511215071
```

R *On retrouve logiquement une moyenne et un écart-type de l'échantillon proches respectivement de la valeur centrale x_c et de l'écart-type $\frac{\Delta}{\sqrt{3}} = 0,58$ attendu pour ce type de distribution.*

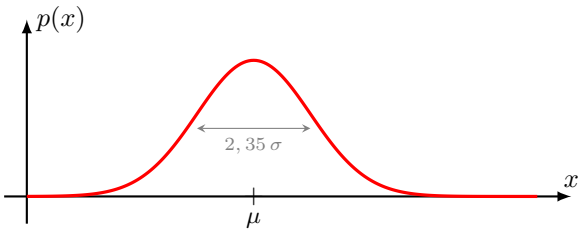
Cours

C Loi normale (ou gaussienne)

Qu'est-ce que c'est ?

Une variable aléatoire X suit une loi normale (ou gaussienne) d'espérance μ et d'écart-type σ lorsque la loi de probabilité s'écrit sous la forme :

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}}\, e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$



R *La largeur à mi-hauteur vaut environ* $2,35\,\sigma$.

Tirage aléatoire avec Python

On aura chargé préalablement le module **random** de la bibliothèque **NumPy** avec l'instruction :

```
import numpy.random as rd
```

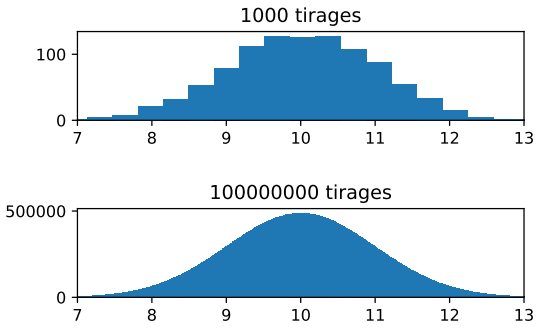
Puis, connaissant l'écart-type σ et l'espérance μ, on utilisera l'instruction :

```
rd.normal(mu, sigma, N)
```

permettant de générer un tableau NumPy unidimensionnel contenant N valeurs tirées au sort suivant la loi décrite précédemment.

Exemple

```
1  import matplotlib.pyplot as plt
2  import numpy.random as rd
3
4  N = 1000 # nombre de tirages
5  mu = 10.
6  sigma = 1.
7
8  b = rd.normal(mu,sigma,N)
9
10 # tracé de l'histogramme
11 plt.hist(b,bins='rice')
12 plt.title("%i tirages" %N)
13 plt.show()
```



R *L'option* ***bins='rice'*** *permet d'optimiser le choix du nombre de barres à afficher.*

Poursuivons avec le calcul de la moyenne et de s_x :

```
14 import numpy as np
15
16 moyenne = np.average(b)
17 sx = np.std(b,ddof=1)
18
19 print("Pour",N,"tirages :")
20 print("moyenne =",moyenne)
21 print("sx =",sx)
```

⇛

```
Pour 1000 tirages :
moyenne = 9.945380010087232
sx = 0.9693775291004446
```

R *On retrouve logiquement une moyenne et un écart-type de l'échantillon proches respectivement de la valeur de l'espérance* μ *et de l'écart-type* σ *choisie.*

Cours

4 Régression linéaire

Capacités exigibles

- Utiliser la fonction **polyfit** de la bibliothèque **numpy** (sa spécification étant fournie) pour exploiter des données

Charger la bibliothèque **numpy** avec `import numpy as np`, puis utiliser l'instruction :

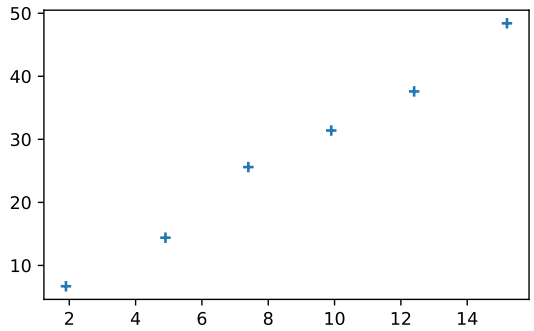
`p = np.polyfit(`*liste des valeurs d'abscisses*`,` *liste des valeurs d'ordonnées*`, 1)`

de sorte que :

- `p[0]` renvoie la valeur du **coefficient directeur** ;
- `p[1]` renvoie la valeur de l'**ordonnée à l'origine**.

Voici un **exemple** :

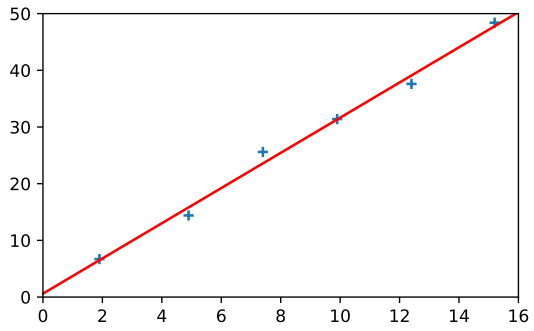
```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  # valeurs d'abscisses
5  X = np.array([1.9,  4.9,  7.4,  9.9, 12.4, 15.2])
6
7  # valeurs d'ordonnées
8  Y = np.array([6.7, 14.4, 25.6, 31.4, 37.6, 48.4])
9
10 # tracés des points expérimentaux
11 plt.scatter(X,Y,marker='+')
12 plt.show()
```



Visiblement, les points semblent s'aligner. On serait donc tenté d'effectuer une modélisation affine. On peut donc compléter le code précédent ainsi :

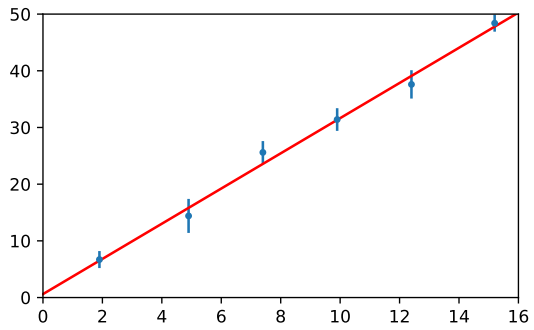
```
13 p = np.polyfit(X,Y,1)
14
15 print("coefficient directeur = ", p[0])
16 print("ordonnée à l'origine = ", p[1])
17
18 # abscisses pour la droite de modélisation
19 X_mod = np.linspace(0,20,10)
20
21 # ordonnées pour la droite de modélisation
22 Y_mod = p[0]*X_mod + p[1]
23
24 # tracé des points expérimentaux
25 plt.scatter(X,Y,marker='+')
26
27 # tracé de la droite de modélisation
28 plt.plot(X_mod,Y_mod,color='red')
29 plt.show()
```

⇛
```
coefficient directeur = 3.103501566306996
ordonnée à l'origine = 0.6081615036547237
```



Si on connaît les incertitudes, on les fait apparaître à l'aide de `plt.errorbar` au lieu de `plt.scatter` :

```
30 # valeurs des incertitudes en ordonnées
31 u_Y = np.array([1.5, 3, 2, 2, 2.5, 1.5 ])
32
33 # points expérimentaux avec barres d'incertitudes
34 plt.errorbar(X,Y, yerr= u_Y, marker='.', linestyle='')
35
36 # tracé de la droite de modélisation
37 plt.plot(X_mod,Y_mod,color='red')
38 plt.show()
```



L'affichage des barres d'incertitudes permet de valider d'autant mieux le modèle affine : la droite de modélisation «traverse» toutes les barres d'incertitudes, les points expérimentaux semblent donc en accord avec le modèle.

N. Gaudouen

Cours

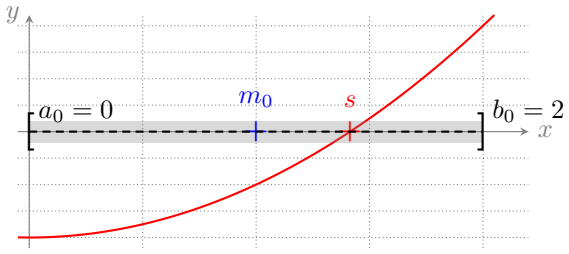
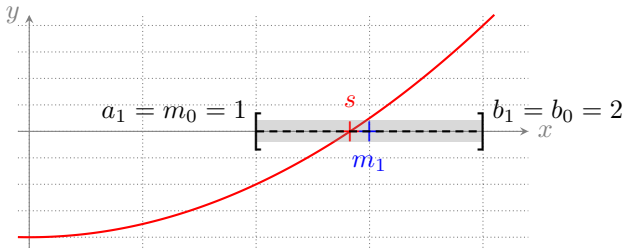
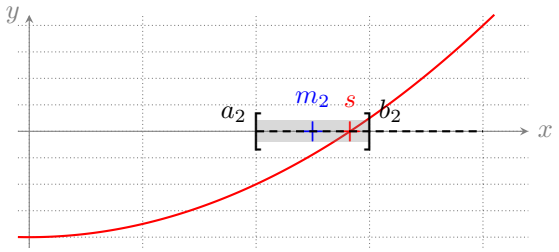
5 Méthode dichotomique

Capacités exigibles

- Déterminer, en s'appuyant sur une représentation graphique, un intervalle adapté à la recherche numérique d'une racine par une méthode dichotomique
- Mettre en œuvre une méthode dichotomique afin de résoudre une équation avec une précision donnée
- Utiliser la fonction **bisect** de la bibliothèque **scipy.optimize** (sa spécification étant fournie)

A Principe de la méthode dichotomique

OBJECTIF : résoudre une équation de la forme $f(x) = 0$.

PRINCIPE	Exemple : résoudre $f(x) = 0$, où $f(x) = x^2 - 2$, de solution $s = \sqrt{2} = 1,4142...$
Déterminer un 1er intervalle $[a_0, b_0]$ dans lequel on est sûr que la solution se situe. **Le milieu $m_0 = \dfrac{a_0 + b_0}{2}$ de $[a_0, b_0]$ est un 1er estimateur de la solution s.**	
Recommencer l'opération précédente avec un intervalle $[a_1, b_1]$ plus petit que $[a_0, b_0]$, de sorte qu'on soit sûr que la solution s recherchée s'y trouve. Comment en être sûr ? La fonction f change nécessairement de signe dans l'intervalle $[a_1, b_1]$. Donc, on choisit : ➤ $[a_1, b_1] = [a_0, m_0]$ si $f(a_0)$ et $f(m_0)$ sont de signes opposés, ➤ $[a_1, b_1] = [m_0, b_0]$ sinon. **Le milieu $m_1 = \dfrac{a_1 + b_1}{2}$ de $[a_1, b_1]$ est un 2ème estimateur de la solution s.**	
On réitère à nouveau. Ci-contre, le rang $n = 2$. **On arrête l'itération lorsque la précision ϵ souhaitée a été atteinte.** Concrètement : il faut : $\lvert b_n - a_n \rvert \leqslant \epsilon$ Ainsi, l'estimateur m_n est écarté de la valeur s d'au plus $\dfrac{\epsilon}{2}$.	

R ***Lien entre précision et nombre d'itérations ?***

La précision obtenue au rang n est $\epsilon_n = \dfrac{b_0 - a_0}{2^n}$, puisqu'à chaque itération, on ne garde que la moitié de l'intervalle de l'itération précédente.

Ainsi, pour une précision finale ϵ souhaitée, nécessitant N d'itérations, on veut $\epsilon_N = \epsilon \Leftrightarrow N = \dfrac{\ln\left(\frac{b_0 - a_0}{\epsilon}\right)}{\ln 2}$.

On perçoit clairement que plus ϵ est faible, plus N est grand. Cela paraît plutôt logique ...

Cours

B Implémentation sous Python

Voici la structure de l'algorithme :

- Indiquer des valeurs initiales de a et b telles qu'on soit sûr que la solution recherchée se trouve dans $[a,b]$
- Préciser une valeur de ϵ, taille de l'intervalle final souhaité, pour obtenir une solution suffisamment précise.
- Tant que $b-a>\epsilon$:
  - évaluer $m=\dfrac{a+b}{2}$
  - si $f(a)$ et $f(m)$ sont de signes opposés, alors la solution est dans $[a,m]$. Les bornes du nouvel intervalle sont donc $a=a$ et $b=m$
  - si $f(a)$ et $f(m)$ sont de même signe, alors la solution est dans $[m,b]$. Les bornes du nouvel intervalle sont donc $a=m$ et $b=b$
- Dès que $b-a\leqslant\epsilon$: l'estimation suffisamment précise de la solution est la dernière valeur de m obtenue.

On peut alors implémenter l'algorithme ainsi sous Python :

```
def solution_dichotomie(f,a,b,epsilon):
    while b-a > epsilon :
        m = (a+b)/2
        if f(a)*f(m)<0 : # càd si f(a) et f(m) sont de signes opposés
            b = m
        else: # sinon, càd si f(a) et f(b) sont de même signe
            a = m
    return m
```

Exemple

Reprenons l'exemple étudié dans l'encadré précédent en définissant préalablement la fonction f :

```
def f(x):
    return x**2-2

print('La racine carré de 2 vaut', solution_dichotomie(f,0,2,1E-10))
```

⇛ `La racine carré de 2 vaut 1.4142135623260401`

À comparer à la valeur «plus exacte» fournie par une calculatrice : `1.4142135623730950488016`.
L'écart entre les deux valeurs est $4,7.10^{-11}$ *qui est bien inférieur à* $\epsilon/2=5.10^{-11}$.

C Fonction bisect

La fonction **`bisect`** de la bibliothèque **`scipy.optimize`** exploite tout simplement la méthode dichotomique. Il faut indiquer les mêmes arguments que notre fonction **`solution_dichotomie`** mais sans nécessairement indiquer de précision ϵ particulière, qui vaut **`xtol = 2E-12`** par défaut.

Exemple

Choisissons le même exemple et la même précision que précédemment. Le résultat affiché est alors exactement le même que précédemment.

```
import scipy.optimize as sp
print(sp.bisect(f,0,2,xtol=1E-10))
```

⇛ `1.4142135623260401`

R *À l'avenir, il sera déraisonnable de coder nous-même une méthode de dichotomie quand un code existe déjà et a probablement été plus testé et optimisé que le nôtre. Par exemple, la fonction* **`bisect`** *peut renvoyer un message d'erreur spécifique lorsque l'intervalle $[a,b]$ initial a été mal choisi (lorsque la solution ne s'y trouve pas notamment).*

6 Calcul numérique d'une dérivée

Capacités exigibles

- Utiliser un schéma numérique pour déterminer une valeur approchée du nombre dérivé d'une fonction en un point

A Principe du calcul numérique d'une dérivée

Soient une fonction f dérivable et h un pas de calcul. Pour calculer numériquement $f'(x)$ en une valeur de x donnée, on peut utiliser différentes méthodes :

Différence finie en avant	Différence finie en arrière	Différence finie centrée
$f'(x) \simeq \dfrac{f(x+h)-f(x)}{h}$	$f'(x) \simeq \dfrac{f(x)-f(x-h)}{h}$	$f'(x) \simeq \dfrac{f(x+h)-f(x-h)}{2h}$
`def deriv_avant(f,x,h)` `  return (f(x+h)-f(x))/h`	`def deriv_arriere(f,x,h)` `  return (f(x)-f(x-h))/h`	`def deriv_centree(f,x,h)` `  return (f(x+h)-f(x-h))/2h`

Il faut choisir une valeur de h suffisamment petite afin de pouvoir retranscrire fidèlement les variations locales de la fonction f.

Toutefois, h devra bien entendu rester supérieur au plus petit écart possible entre deux nombre flottants différents, càd $h > 2.10^{-16}$ en 64 bits.

B Dérivée d'un tableau de valeurs

Soient un tableau de valeurs du temps t et un tableau de valeurs d'une grandeur $x(t)$.

La dérivée de $x(t)$ peut être calculée sous forme d'un tableau de la manière suivante (en suivant la méthode des différences finies en avant) :

```
import numpy as np

t = ...      # tableau des valeurs des instants
x = ...      # tableau des valeurs de x(t)

# initialisation du tableau des valeurs des dérivées :
v = np.zeros(len(x)-1)

""" le tableau v contiendra nécessairement un élément de moins que x
à cause de la méthode de dérivée numérique employée """


# Calcul des dérivées :

for i in range(0,len(v)):
    v[i] = ( x[i+1] - x[i] ) / ( t[i+1] - t[i] )

""" v[i] représente ainsi la dérivée de x(t) à l'instant t[i] """
```

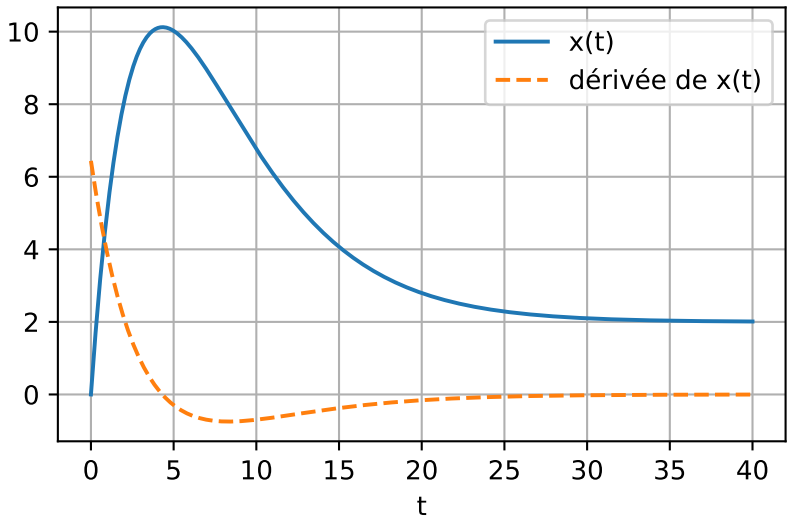
Cours

Exemple

On souhaite tracer la fonction $x(t) = (6t-2)e^{-t/4} + 2$, ainsi que sa dérivée.

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

t = np.linspace(0,40.,1000) # tableau des valeurs des instants
x = (6*t-2)*np.exp(-t/4) + 2    # tableau des valeurs de x(t)


# initialisation du tableau des valeurs des dérivées :
v = np.zeros(len(x)-1)

# Calcul des dérivées :
for i in range(0,len(v)):
    v[i] = ( x[i+1] - x[i] )/(t[i+1]-t[i])


# tracés

plt.plot(t,x,label="x(t)")

# avec np.delete(t,-1), on supprime le dernier instant pour le tracé de v(t)
plt.plot(np.delete(t,-1),v,'--',label="dérivée de x(t)")

plt.legend()
plt.xlabel("t")
plt.grid()
plt.show()
```

On obtient le tracé ci-dessous :



R *Le tracé est cohérent : au début, la fonction $x(t)$ est croissante puis devient décroissante, ainsi la dérivée est initialement positive puis s'annule et devient négative. À la fin du tracé, la fonction $x(t)$ devient constante, donc la dérivée s'annule.*

Cours

7 Calcul numérique d'une intégrale

Capacités exigibles

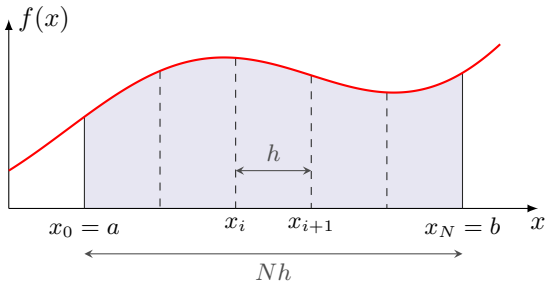
- Mettre en œuvre la méthode des rectangles pour calculer une valeur approchée d'une intégrale sur un segment

A Méthode des rectangles

On souhaite calculer l'intégrale $\int_a^b f(x)\mathrm{d}x$.

Pour cela, on «découpe» tout d'abord l'intervalle $[a,b]$ en N segments de longueur $\boldsymbol{h = \dfrac{b-a}{N}}$ comme dans l'exemple ci-contre.

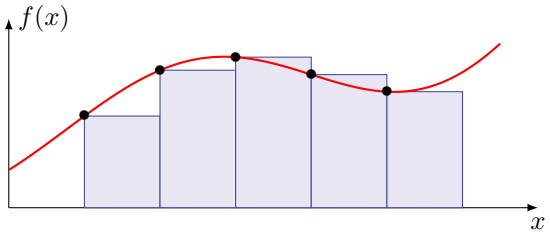
Dans la suite, notons $x_i = a + ih$.



A.1 Méthode des rectangles à gauche

Pour chaque valeur de $i \in [\![0, N-1]\!]$, on calcule l'aire d'un rectangle compris entre les abscisses x_i et x_{i+1} de hauteur $f(x_i)$ *(valeur de f à gauche à l'extrémité gauche de l'intervalle $[x_i, x_{i+1}]$)*.

L'aire du rectangle compris entre x_i et x_{i+1} est alors égale à $hf(x_i) = hf(a+ih)$.

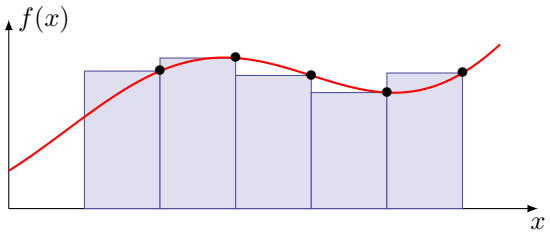


Puis, en sommant toutes ces aires, on obtient une valeur approchée de l'aire sous la courbe, et donc une valeur approchée de l'intégrale $\int_a^b f(x)\mathrm{d}x$:

Méthode des rectangles à gauche : $$\int_a^b f(x)\mathrm{d}x \simeq h\sum_{i=0}^{N-1} f(a+ih)$$

A.2 Méthode des rectangles à droite

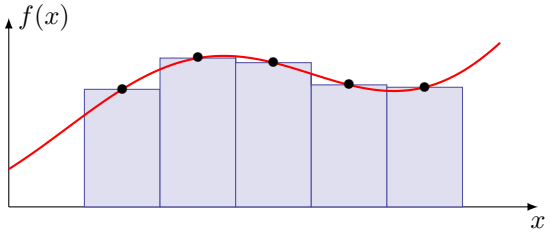
La méthode est la même que précédemment mais désormais la hauteur est choisie à $f(x_{i+1})$ *(valeur de f à gauche à l'extrémité droite de l'intervalle $[x_i, x_{i+1}]$)*. L'aire du rectangle compris entre x_i et x_{i+1} est alors égale à $hf(x_{i+1}) = hf(a+ih+h)$.



Méthode des rectangles à droite : $$\int_a^b f(x)\mathrm{d}x \simeq h\sum_{i=0}^{N-1} f(a+ih+h)$$

A.3 Méthode des rectangles au milieu

La méthode reste encore la même que précédemment mais désormais la hauteur est choisie à $f\left(x_i + \dfrac{h}{2}\right)$ *(valeur de f au milieu de l'intervalle $[x_i, x_{i+1}]$)*. L'aire du rectangle compris entre x_i et x_{i+1} est alors égale à $hf(x_{i+1}) = hf(a+ih+h)$.

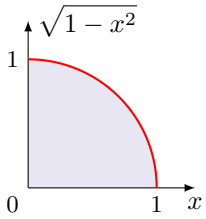


Méthode des rectangles au milieu : $$\int_a^b f(x)\mathrm{d}x \simeq h\sum_{i=0}^{N-1} f\left(a+ih+\frac{h}{2}\right)$$

N. Gaudouen

B Implémentation sous Python et exemple

On souhaite calculer la valeur de π. Pour cela, on peut calculer quatre fois l'aire d'un quart de cercle de rayon 1, décrit par la fonction $f(x) = \sqrt{1-x^2}$.



R *En effet, c'est l'expression obtenue lorsqu'on isole l'ordonnée y dans l'équation cartésienne de ce cercle : $x^2+y^2=1$, d'aire totale π ...*

Il suffit donc d'intégrer $f(x)$ sur $[0,1]$ et de multiplier par 4.

Voici une manière d'implémenter les méthodes vues précédemment sous Python :

```python
import numpy as np

#--------------------------------
# Méthode des rectangles à gauche
def int_rect_gauche(f,a,b,N):
    h = (b-a)/N
    S = 0
    for i in range(N):
        S = S + f(a+i*h)
    return S*h

# Méthode des rectangles à droite
def int_rect_droite(f,a,b,N):
    h = (b-a)/N
    S = 0
    for i in range(N):
        S = S + f(a+i*h+h)
    return S*h

# Méthode des rectangles au milieu
def int_rect_milieu(f,a,b,N):
    h = (b-a)/N
    S = 0
    for i in range(N):
        S = S + f(a+i*h+h/2)
    return S*h

#--------------------------------
# Fonction décrivant le quart de cercle
def f_quart_cercle(x):
    return np.sqrt(1-x**2)

#--------------------------------
# Nombre de rectangles
N = 1000

#--------------------------------
# Evaluation de pi
eval_pi =  4*int_rect_gauche(f_quart_cercle,0,1,N)
print("Méthode des rectangles à gauche :",eval_pi, "écart : ", eval_pi-np.pi)

eval_pi =  4*int_rect_droite(f_quart_cercle,0,1,N)
print("Méthode des rectangles à droite :",eval_pi, "écart : ", eval_pi-np.pi)

eval_pi =  4*int_rect_milieu(f_quart_cercle,0,1,N)
print("Méthode des rectangles au milieu :",eval_pi, "écart : ", eval_pi-np.pi)
```

⇛
```
Méthode des rectangles à gauche : 3.143555466911023   écart : 0.0019628133212297 17
Méthode des rectangles à droite : 3.139555466911023   écart : -0.0020371866787702864
Méthode des rectangles au milieu : 3.1416035449129063   écart : 1.0891323113160212e-05
```

On constate que **la méthode des rectangles au milieu converge plus rapidement** !

Et bien entendu, **la précision du résultat sera d'autant meilleure que le nombre N de rectangles est grand** *(subdivisions plus «fines» de l'intervalle $[a,b]$ et donc approximation de la fonction f par une fonction en escalier d'autant meilleure)*. Voici le résultat pour $N = 100\,000$ dans l'exemple précédent :

⇛
```
Méthode des rectangles à gauche : 3.1416126164020075 écart : 1.9962812214391334e-05
Méthode des rectangles à droite : 3.1415726164020077 écart : -2.0037187785426624e-05
Méthode des rectangles au milieu : 3.1415926644818297 écart : 1.0892036605980593e-08
```

8 Équation différentielle d'ordre 1

Capacités exigibles

- Mettre en œuvre la méthode d'Euler explicite afin de résoudre une équation différentielle d'ordre 1
- Utiliser la fonction **odeint** de la bibliothèque **scipy.integrate** (sa spécification étant fournie)

A Méthode d'Euler explicite

A.1 Principe

Soit $y(t)$ une fonction inconnue et vérifiant une équation du 1er ordre pouvant s'écrire sous la forme :

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f(y,t)$$

où $f(y,t)$ est une expression faisant intervenir directement y et éventuellement la variable de temps t

Exemple

La tension $u(t)$ aux bornes du condensateur du circuit RC série soumis à un échelon de tension E vérifie :

$$\tau\dot{u} + u = E$$

Cette équation peut s'écrire sous la forme :

$$\frac{\mathrm{d}u}{\mathrm{d}t} = f(u,t) \qquad \text{où} \quad f(u,t) = \frac{E-u}{\tau}$$

On cherche à tracer la fonction $y(t)$ entre deux instants t_i et t_f, connaissant la condition initiale $y(t_i) = y_i$.
La méthode d'Euler explicite consiste dans un premier temps à **subdiviser l'axe temporel** en N petits intervalles de temps de longueur $h = \dfrac{t_f - t_i}{N}$ (le pas de temps). On dit que l'on discrétise l'axe des temps. Notons $t_k = t_i + kh$ les instants successifs étudiés. Ainsi, $t_i = t_0$ et $t_f = t_N$.

Puis, **on intègre l'équation différentielle** $\dfrac{\mathrm{d}y}{\mathrm{d}t} = f(y,t)$ sur l'intervalle $[t_k, t_{k+1}]$:

$$y(t_{k+1}) - y(t_k) = \int_{t_k}^{t_{k+1}} f(y,t)\,\mathrm{d}t$$

pour faire apparaître une **relation de récurrence**, en notant $y_k = y(t_k)$:

$$y_{k+1} = y_k + \int_{t_k}^{t_{k+1}} f(y,t)\,\mathrm{d}t$$

Ainsi, connaissant $y_0 = y_i$, **on peut en déduire pas à pas toutes les valeurs y_k** grâce à la relation ci-dessus.

Mais encore faut-il pouvoir préciser la valeur de l'intégrale $\displaystyle\int_{t_k}^{t_{k+1}} f(y,t)\,\mathrm{d}t$. Pour cela, la méthode d'Euler explicite consiste à considérer que la fonction $f(y,t)$ prend la valeur constante $f(y_k,t_k)$ sur l'intervalle $[t_k, t_{k+1}]$:

$$\forall t \in [t_k, t_{k+1}]\,,\ f(y,t) = f(y_k,t_k)$$

R *Approximation similaire à la méthode des rectangles à gauche, voir partie 7.*

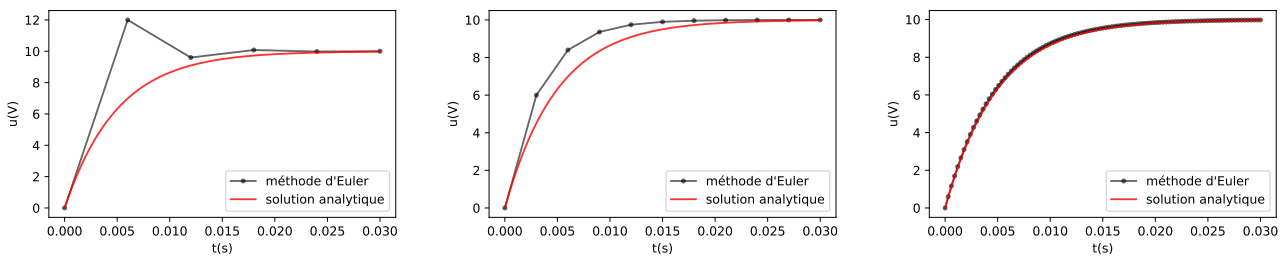
D'où $\displaystyle\int_{t_k}^{t_{k+1}} f(y,t)\,\mathrm{d}t \simeq f(y_k,t_k)\,h$. On obtient alors la relation de récurrence suivante :

$$\boxed{y_{k+1} = y_k + f(y_k,t_k)\,h}$$

A.2 Implémentation sous Python et exemple

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

# --------------------------------
# ALGORITHME D'EULER

def euler(f,yi,ti,tf,N):
    # pas de temps
    h = (tf-ti)/N

    # conditions initiales :
    t = ti
    y = yi

    # on initialise les listes qui accueilleront les valeurs successives
    # de t et y obtenues grâce à la boucle for
    liste_t = [t]
    liste_y = [y]

    for k in range(1,N+1):
        # application des relations de récurrence
        y = y + f(y,t)*h
        t = t + h

        # on remplit les listes à chaque itération
        liste_t.append(t)
        liste_y.append(y)

    return liste_t, liste_y

# --------------------------------
# CIRCUIT RC SERIE SOUMIS A UN ECHELON DE TENSION

E = 10 #V
R = 500 #Ohms
C = 10E-6 # F
tau = R*C

def f_RC_echelon(u,t):
    return (E-u)/tau

ti = 0
ui  = 0 # condensateur initialement déchargé
tf = 6*tau # supérieur à 5*tau, donc régime transitoire quasi-terminé
N = 100

t_euler,u_euler = euler(f_RC_echelon,ui,ti,tf,N)

plt.plot(t_euler,u_euler, marker='.',label = "méthode d'Euler")

# solution analytique pour comparer
ta = np.linspace(ti,tf,1000)
ua = E*(1-np.exp(-ta/tau))
plt.plot(ta,ua,label="solution analytique")

plt.legend(loc='lower right')
plt.xlabel('t(s)')
plt.ylabel('u(V)')
plt.show()
```

$N = 5$ $N = 10$ $N = 100$



La méthode d'Euler est d'autant plus satisfaisante que le pas de temps est faible (càd N élevé).

Cours

B Utilisation de la fonction odeint

La résolution d'une équation différentielle d'ordre 1 de la forme :

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f(y,t)$$

peut aussi s'effectuer avec la fonction **odeint** de la bibliothèque **scipy.integrate**.

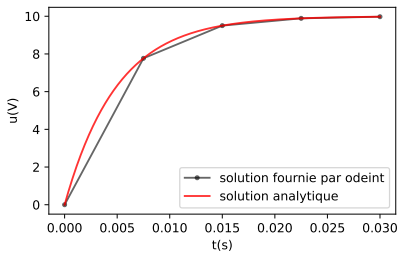
Il faut préalablement définir une échelle de temps, par exemple avec **`np.linspace`** et utiliser l'instruction suivante (en ayant préalablement chargé la bibliothèque scipy.integrate avec l'alias sp par exemple) :

`sp.odeint(`*fonction f, valeur initiale de $y(t)$, array des N instants t_k***`)`**

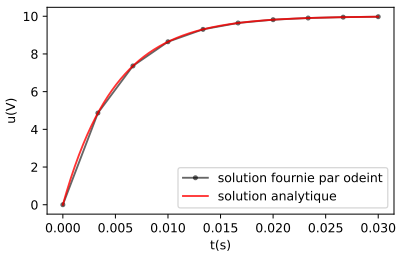
Reprenons l'exemple précédent pour illustrer. On rajoute les lignes suivantes pour cela :

```
60 import scipy.integrate as sp
61
62 # solution avec odeint
63
64 t_odeint = np.linspace(ti,tf,N)
65 u_odeint = sp.odeint(f_RC_echelon,ui,t_odeint)
66
67 plt.plot(t_odeint,u_odeint,marker='.',label="solution fournie par odeint")
68 plt.legend(loc='lower right')
69 plt.show()
```

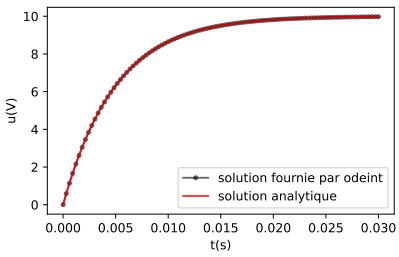
$N = 5$



$N = 10$



$N = 100$



La solution fournie par odeint est très satisfaisante même si le pas de temps est grand (càd N faible).

R *L'algorithme mis en œuvre par odeint est bien plus élaboré et robuste que le simple algorithme d'Euler, mais conformément au programme, nous ne rentrerons pas dans les détails...*

Bien entendu, il vaut tout de même mieux choisir un grand nombre N de points afin de lisser le tracé.

Cours

9 Équation différentielle d'ordre supérieur à 1

Capacités exigibles

- 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)

A Exemple du système masse-ressort

A.1 Se ramener à un système d'équations différentielles d'ordre 1

L'équation différentielle vérifiée par la position $x(t)$ de la masse est :

$$\ddot{x} + \frac{\omega_0}{Q}\,\dot{x} + \omega_0^2\,x = \omega_0^2\,x_{eq} \quad \text{où} \quad \omega_0 = \sqrt{\frac{k}{m}} \quad \text{et} \quad Q = \frac{1}{\lambda}\sqrt{km}$$

Pour résoudre numériquement cette équation différentielle d'ordre 2, l'idée est de **se ramener à un système de deux équations différentielles d'ordre 1**. Pour cela, on peut introduire une autre fonction inconnue qui serait simplement la vitesse $v_x(t) = \dot{x}(t)$:

$$\left\{\begin{array}{rcl} \dot{x}(t) & = & v_x(t) \\ \dot{v}_x(t) & = & -\omega_0^2\,(x(t) - x_{eq}) - \dfrac{\omega_0}{Q}\,v_x(t) \end{array}\right.$$

Si on définit le vecteur $y(t) = \begin{pmatrix} x(t) \\ v_x(t) \end{pmatrix}$, alors $\dfrac{dy}{dt} = \begin{pmatrix} \dot{x}(t) \\ \dot{v}_x(t) \end{pmatrix} \Leftrightarrow \dfrac{dy}{dt} = \begin{pmatrix} v_x \\ -\omega_0^2\,(x - x_{eq}) - \dfrac{\omega_0}{Q}\,v_x \end{pmatrix}$.

En notant $y = \begin{pmatrix} y_0 \\ y_1 \end{pmatrix}$ où $y_0 = x$ et $y_1 = v_x$, on obtient l'équation différentielle du 1er ordre suivante :

$$\boldsymbol{\frac{dy}{dt} = f(y,t)} \quad \text{où :} \quad f(y,t) = \begin{pmatrix} y_1 \\ -\omega_0^2\,(y_0 - x_{eq}) - \dfrac{\omega_0}{Q}\,y_1 \end{pmatrix}$$

qui peut être résolue numériquement avec **odeint** ! (voir partie 8).

A.2 Résolution avec odeint

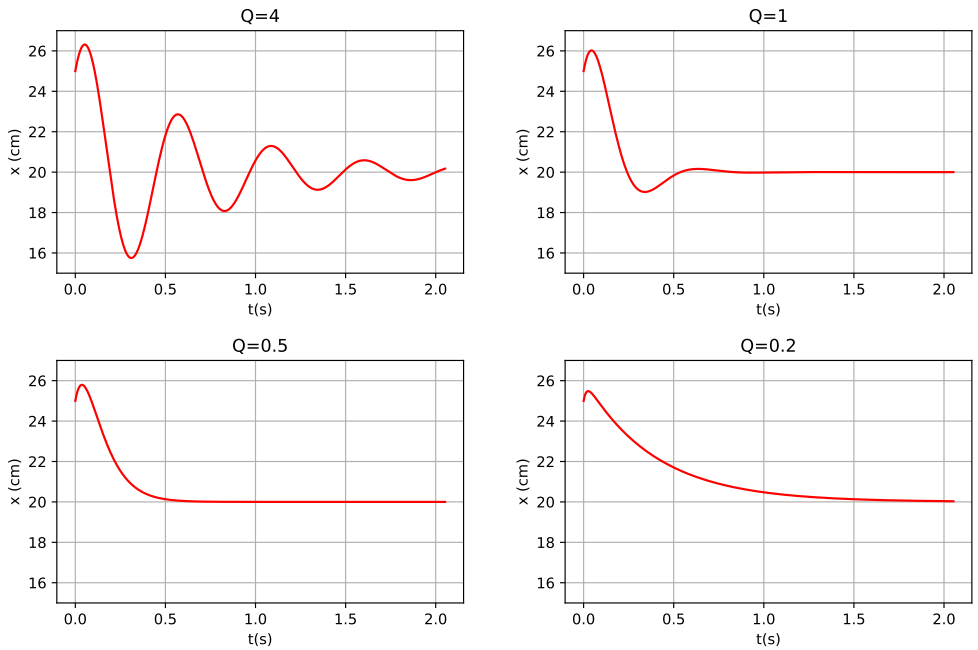
L'instruction `Y = odeint(`*fonction f , vecteur initial y_i , array des N instants t_k*`)` génère un tableau `Y` à N lignes (le nombre d'instants) et 2 colonnes (le nombre de fonctions inconnues, $y_0 = x$ et $y_1 = \dot{x}$) :

$y_0(t_0)$	$y_1(t_0))$
$y_0(t_1)$	$y_1(t_1)$
...	...
...	...
$y_0(t_{N-1})$	$y_1(t_{N-1}))$

L'instruction `Y[:,0]` permettra de ne récupérer que la 1ère colonne du tableau, c'est-à-dire les valeurs de $y_0(t) = x(t)$.
De même, `Y[:,1]` permettrait de récupérer la 2ème colonne du tableau, c'est-à-dire les valeurs de $y_1(t) = \dot{x}(t)$.

```python
# Importations des bibliothèques utiles
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate as sp

# ------------------------------
# Paramètres physiques
k = 30 # N.m-1
m = 200E-3 # kg

xeq = 20 # cm

w0 = np.sqrt(k/m)
T0 = 2*np.pi/w0

Q = 4

# ------------------------------
# Définition du second membre
def f_masse_ressort(y,t):
    f0 = y[1]
    f1 = -w0**2*(y[0]-xeq) - w0/Q * y[1]
    return [f0,f1]

# ------------------------------
# Résolution numérique avec odeint

# axe temporel
temps = np.linspace(0,4*T0,1000)

# condition initiales :
xi = xeq + 5 # cm
vi = 50 # cm.s-1
yi = [xi , vi]


Y = sp.odeint(f_masse_ressort,yi,temps)

# récupération des valeurs d'abscisses :
x = Y[:,0]

# ------------------------------
# Tracé de la solution
plt.plot(temps,x,color='red')
plt.grid()
plt.xlabel('t(s)')
plt.ylabel('x (cm)')
plt.title('Q=%r'%Q)
plt.show()
```

Pour différentes valeurs du facteur de qualité Q, on obtient des tracés tout à fait conformes :



Cours

B Généralisation

Soient n fonction inconnues $y_0(t)$, $y_1(t)$, ... vérifiant :

$$\frac{dy}{dt} = f(y,t) \quad \text{, où} \quad y(t) = \begin{pmatrix} y_0(t) \\ y_1(t) \\ y_2(t) \\ \dots \end{pmatrix} \text{ est un vecteur à } n \text{ dimensions}$$

alors l'instruction **`Y = odeint(`***fonction f* **`,`** *vecteur initial y_i* **`,`** *array des N instants t_k*`)` génère un tableau **`Y`** à N lignes (le nombre d'instants) et n colonnes (le nombre de fonctions inconnues). Par exemple pour $n = 4$, **`print(Y)`** affichera les valeurs :

$y_0(t_0)$	$y_1(t_0)$	$y_2(t_0)$	$y_3(t_0)$
$y_0(t_1)$	$y_1(t_1)$	$y_2(t_1)$	$y_3(t_1)$
...	...	...	...
...	...	...	...
$y_0(t_{N-1})$	$y_1(t_{N-1})$	$y_2(t_{N-1})$	$y_3(t_{N-1})$

L'instruction **`Y[:,0]`** permettra de ne récupérer que la 1ère colonne du tableau, c'est-à-dire les valeurs de $y_0(t)$. De même, **`Y[:,1]`** permettrait de récupérer la 2ème colonne du tableau, c'est-à-dire les valeurs de $y_1(t)$. Etc.