L2-S3 : UE Simulations numériques

SEANCE 3

Ajustement de données, Intégration d'une fonction d'une variable

22 septembre 2024

Ajustement de fonctions : méthode des moindres carrés

En science, on est souvent amené à comparer des données expérimentales à un modèle mathématique.

- On mesure par exemple dans une expérience deux grandeurs x et y.
- Les résultats des mesures notés x_i, y_i sont entachés d'incertitudes. Par ailleurs, il y a dans l'expérience des fluctuations (aléatoires ?) que l'on maîtrise mal.
- On voudrait savoir si un lien (une loi $f(x)$) existe entre les mesures de y et les mesures de x.

Ajustement de fonctions : méthode des moindres carrés

Supposons que l'on souhaite interpréter les données avec une fonction $f(x)$, par exemple de la forme :

$$f(x) = ax + b \tag{1}$$

On cherche les valeurs des paramètres a et b qui minimisent l'écart entre les données et la fonction.
Cela revient à minimiser la somme χ^2 des écarts quadratiques entre points expérimentaux y_i et valeurs données par le modèle $f(x_i)$, pondérés par les incertitudes expérimentales :

$$\chi^2(a,b) = \sum_i \frac{(y_i - f(x_i))^2}{\sigma_i^2} = \sum_i w_i r_i^2(x_i)$$

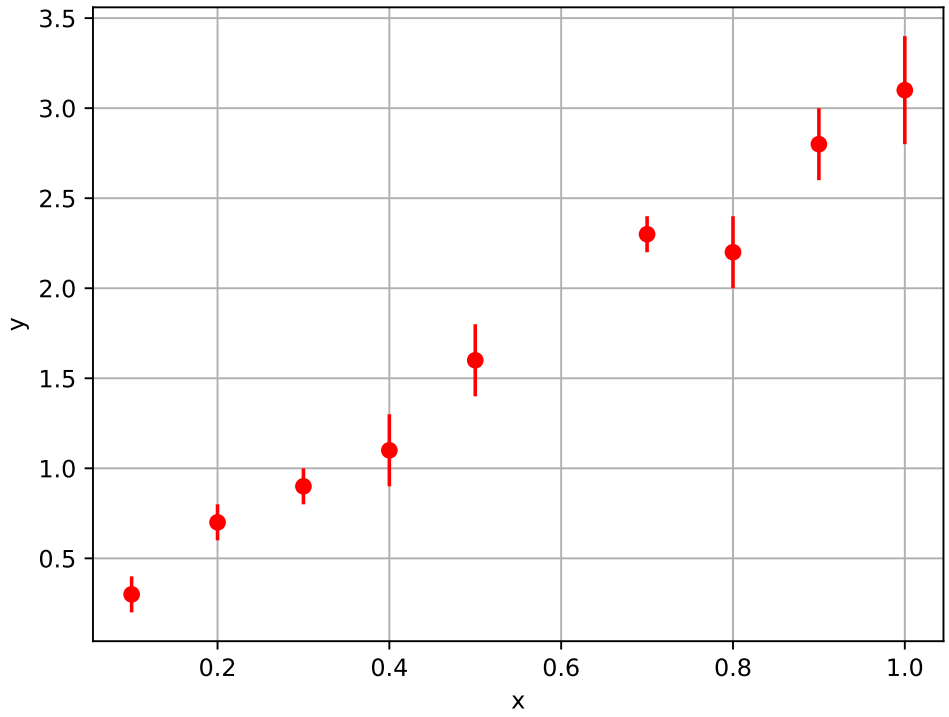
$r_i(x_i)$ est appelé résidu et $w_i = 1/\sigma_i^2$ est le poids qui pondère chaque mesure, d'autant plus faible que l'incertitude σ_i est grande. χ^2 est appelé "Chi deux" ou "Chi carré" : c'est la somme des résidus pondérés au carré, considérée comme une fonction des paramètres de f.

Ajustement de fonctions : méthode des moindres carrés

Exemple :

```
1  import numpy
2  import matplotlib.pyplot as plt
3
4  x=numpy.array([0.1, 0.2, 0.3, 0.4, 0.5, 0.7, 0.8, 0.9, 1])
5  y=numpy.array([0.3, 0.7, 0.9, 1.1, 1.6, 2.3, 2.2, 2.8, 3.1])
6  sigma_y=numpy.array([0.1, 0.1, 0.1, 0.2, 0.2, 0.1, 0.2, 0.2,
       0.3])
7
8  plt.figure()
9  plt.errorbar(x ,y, yerr=sigma_y, marker='o', color='r',
       linestyle='None')
10 plt.xlabel('x')
11 plt.ylabel('y')
12 plt.grid()
13 plt.show()
```

Ajustement de fonctions : méthode des moindres carrés



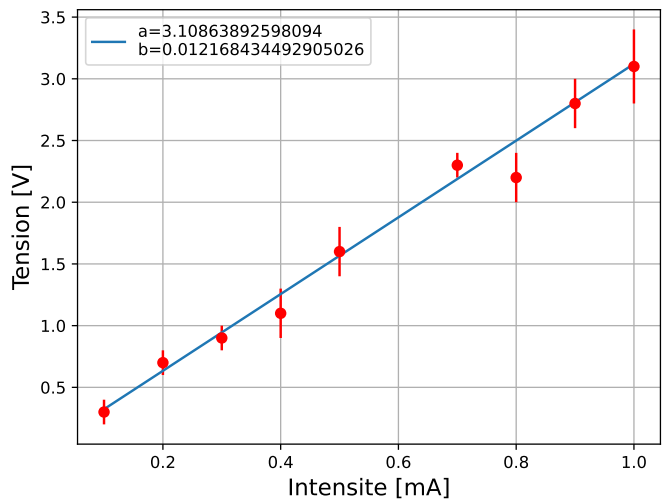
Comment ajuster un modèle à ces données avec leurs incertitudes et en extraire la valeur des paramètres et l'incertitude sur ces valeurs pour que le modèle décrive au mieux les données ?

Comment déterminer si le modèle décrit alors les données de manière satisfaisante ?

Fonction `curve_fit` du module `scipy.optimize`

Ajustement de fonctions : méthode des moindres carrés

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

def fit_func(x, a, b):
    return a*x + b

x = np.array([0.1, 0.2, 0.3, 0.4, 0.5, 0.7, 0.8, 0.9, 1 ])
y = np.array([0.3, 0.7, 0.9, 1.1, 1.6, 2.3, 2.2, 2.8, 3.1 ])
erreur_y = np.array([0.1, 0.1, 0.1, 0.2, 0.2, 0.1, 0.2, 0.2,
      0.3])

a0=1.0
b0=0.0
params_initiaux = [a0,b0] #initialisation de la regression

params,cov = curve_fit(fit_func, x, y, p0=params_initiaux,
      sigma=erreur_y)

[a, b] = params

plt.figure()
plt.errorbar(x, y, yerr=sigma_y, marker='o', color='r',
      linestyle='None')
plt.plot(x, fit_func(x,a,b), label="a="+str(a)+"\nb="+str(b))
plt.xlabel('Intensite [mA]', fontsize=14)
plt.ylabel('Tension [V]', fontsize=14)
plt.legend()
plt.grid()
plt.show()
```



Ajustement de fonctions : méthode des moindres carrés

Arguments de la fonction `curve_fit`

```
1 params,cov=curve_fit(fit_func,x,y,p0=None,sigma=None,...)
```

Paramètres d'entrée :

- `fit_func` est le nom de la fonction servant de modèle.
- `x,y` sont des tableaux `np.array` contenant les valeurs expérimentales x_i et y_i
- `p0` est un tableau contenant les valeurs initiales des paramètres de la fonction (dans l'exemple ci-dessus a et b) pour amorcer l'algorithme itératif de moindres carrés (optionnel mais parfois nécessaire pour des régressions compliquées).
- `sigma` est un tableau `np.array` contenant les valeurs des incertitudes σ_{y_i} sur les valeurs y_i (argument optionnel)

Ajustement de fonctions : méthode des moindres carrés

Résultats de la fonction `curve_fit`

```
1 params,cov=curve_fit(fit_func,x,y,p0=None,sigma=None,...)
```

Paramètres de sortie :

- `params` est un tableau 1D contenant les résultats de la minimisation des moindres carrés : ici $params[0] = a$ et $params[1] = b$.
- `cov` est un tableau 2D. C'est la matrice de covariance. **Ses termes diagonaux contiennent les incertitudes au carré sur les paramètres de la régression :** $cov[0,0] = \sigma_a^2$ et $cov[1,1] = \sigma_b^2$

Remarque : Il existe également la méthode `np.polyfit` qui ajuste des fonctions polynomiales par régression linéaire en prenant en compte les incertitudes.

Le modèle décrit-il suffisamment les données ?

Écarts modèle-points expérimentaux – Notion de χ^2 réduit
Après minimisation des moindres carrés, les résidus ne sont pas complètement nuls. Si la régression a convergé et que les incertitudes ont été évaluées "honnêtement", résidus et incertitudes doivent être du même ordre de grandeur, et la moyenne des résidus doit être nulle.

Pour évaluer si le modèle s'ajuste bien aux données, étant données les incertitudes de mesure, on calcule **le χ^2 réduit** :

$$\chi^2_\nu = \frac{1}{\nu} \sum_i \frac{(y_i - f(x_i))^2}{\sigma_i^2} = \frac{1}{\nu} \sum_i w_i r_i^2(x_i) \qquad (2)$$

où $\nu = N - k$, N le nombre de points expérimentaux et k le nombre de paramètres de la fonction modèle ($k = 2$ pour une droite) :

- si la valeur de χ^2_ν est beaucoup plus grande que 1, le modèle n'est pas pertinent ou on a sous-estimé les incertitudes
- si χ^2_ν <0.1, les incertitudes ont été surestimées.

Ajustement par une fonction quelconque

Tout ce que nous avons illustré avec l'exemple d'une droite est valable pour une fonction $y = f(x)$ quelconque.

- Notre modèle est de la forme : $y = f(x, \beta_1, \beta_2...)$, où $\beta_1, \beta_2...$ sont des paramètres (dans le cas linéaire il s'agissait de $\beta_1 = a$ et $\beta_2 = b$).
- On cherche les valeurs des paramètres $\beta_1, \beta_2...$ qui minimisent la somme des écarts entre points expérimentaux y_i et valeurs données par le modèle $f(x_i, \beta_1, \beta_2...)$.
- On pondère la contribution des points par un coefficient $w_i = \frac{1}{\sigma_i^2}$

On cherche donc les valeurs des paramètres $\beta_1, \beta_2...$ qui minimisent :

$$\chi^2 = \sum_i \frac{(y_i - f(x_i, \beta_1, \beta_2...))^2}{\sigma_i^2} \tag{3}$$

Ajustement par une fonction quelconque

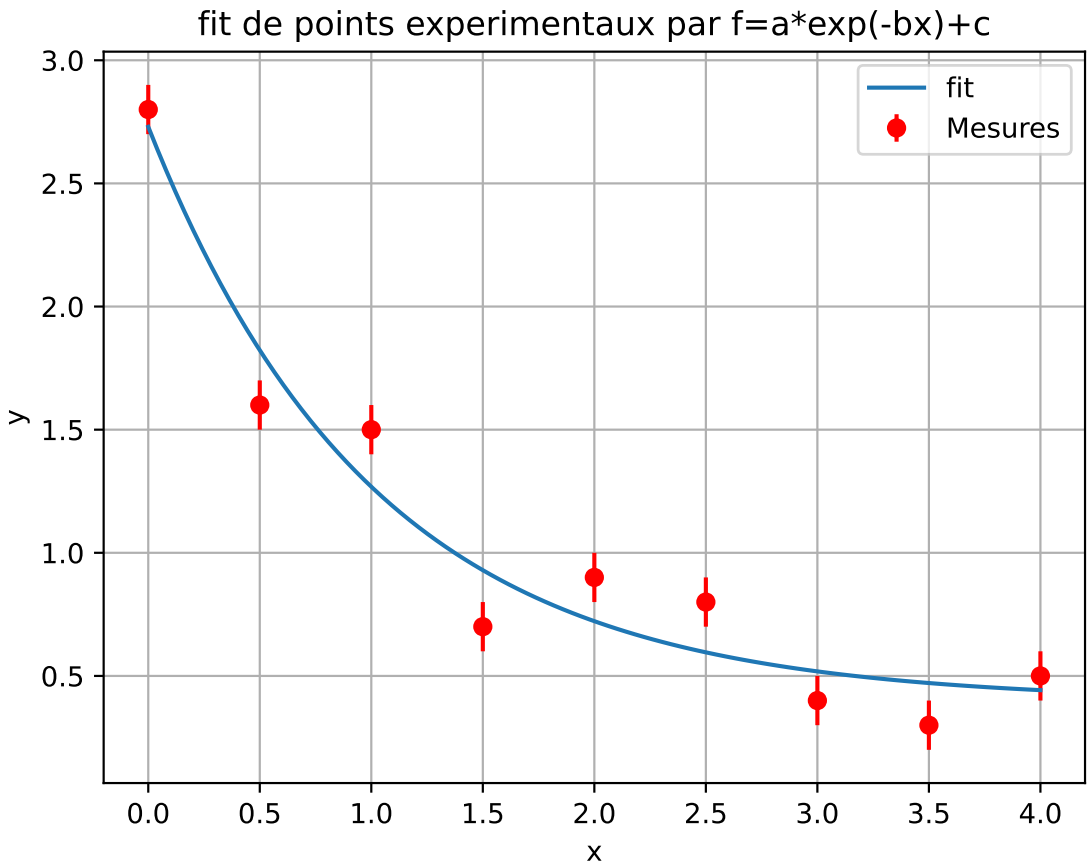
Exemple : ajustement de points expérimentaux par une fonction : $f(x) = ae^{-bx} + c$

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from scipy.optimize import curve_fit
4
5  def func(x, a, b, c):
6      return a * np.exp(-b * x) + c
7
8  xdata = np.array([ 0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0])
9  ydata = np.array([2.8, 1.6, 1.5, 0.7, 0.9, 0.8, 0.4, 0.3, 0.5])
10 erreur_y = np.array([0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1])
11 xdatafit = np.linspace(np.min(xdata), np.max(xdata), 100)
12
13 param0 = np.array([1.0, 1.0, 0.0]) #initial guess
14 popt,cov = curve_fit(func, xdata, ydata, p0=param0, sigma=erreur_y)
15 print('\tValeur de a :',round(popt[0],1),'+/-',round(np.sqrt(cov[0,0]),1),' unite')
16 print('\tValeur de b :',round(popt[1],1),'+/-',round(np.sqrt(cov[1,1]),1),' unite')
17 print('\tValeur de c :',round(popt[2],1),'+/-',round(np.sqrt(cov[2,2]),1),' unite')
18
19 plt.figure()
20 plt.errorbar(xdata,ydata,yerr=erreur_y,marker='o',color='r',label='Mesures',linestyle='None')
21 plt.plot(xdatafit,func(xdatafit,popt[0],popt[1],popt[2]),label='fit')
22 plt.xlabel('x')
23 plt.ylabel('y')
24 plt.title('fit de points experimentaux par f=a*exp(-bx)+c')
25 plt.legend()
26 plt.grid()
27 plt.show()
```

Ajustement par une fonction quelconque

Out[0]:

```
Valeur de a : 2.3 +/- 0.2  unite
Valeur de b : 1.0 +/- 0.3  unite
Valeur de c : 0.4 +/- 0.2  unite
```



Calculs d'intégrales

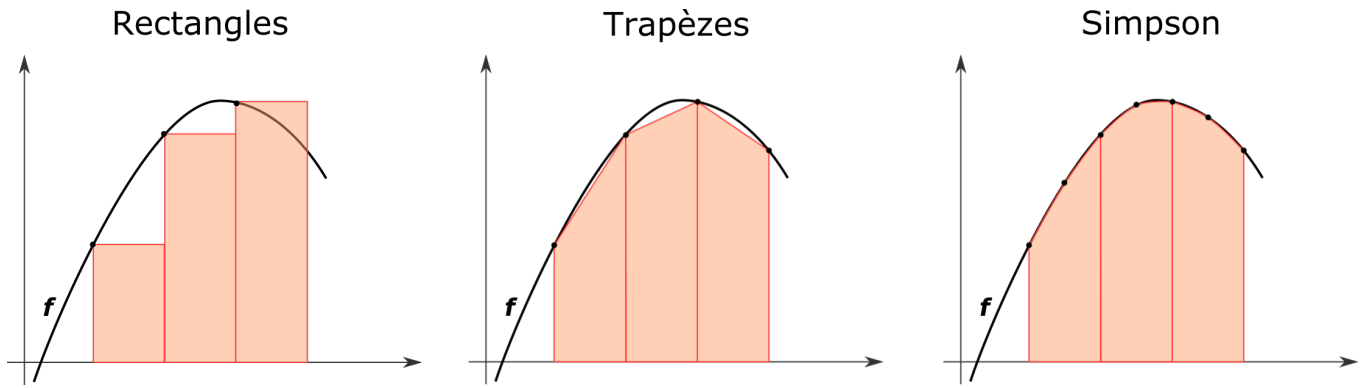
La fonction f à intégrer est échantillonnée en un nombre restreint de points $\rightarrow$ plusieurs outils sont à notre disposition, dont :

- interpolation linéaire par morceaux, puis intégration $\rightarrow$ **méthode des trapèzes** : fonction `trapz` du module `scipy.integrate` (*simple et robuste - standard pour une fonction suffisamment régulière*)
- "optimisation" des points où est évaluée la fonction $\rightarrow$ **quadrature Gaussienne** : fonction `leggauss` du module `numpy.polynomial.legendre`(*plus de liberté - plus difficile d'estimer l'erreur commise*)
- méthode de quadrature plus sophistiquée : fonction quad du module `scipy.integrate`. **La** méthode d'intégration à utiliser en général
- échantillonage aléatoire de la fonction $\rightarrow$ **calcul "Monte-Carlo"** (*spécifiquement pour des fonctions très irrégulières ou des fonctions à N variables avec N assez grand*)

Calculs d'intégrales

Vu en L1 : $f(x)$ est échantillonnée entre a et b. On souhaite estimer $\int_a^b f(x)\mathrm{d}x$.

- ordre 0 : somme de Riemann
- ordre 1 : méthode des trapèzes
- ordre 2 : méthode de Simpson



Syntaxe de la fonction trapz

It = trapz(y,x)
Attention : y d'abord et x ensuite

- x : ensemble de valeurs entre les bornes d'intégration a et b (incluses)
- y=f(x) : valeurs de la fonction à intégrer
- It : valeur calculée de l'intégrale

Dans le cas où les N valeurs de x sont équidistantes de $\delta_x = \frac{b-a}{N-1}$:
It = trapz(y,dx=δ_x*)* ou *It = trapz(y,dx=x[1]-x[0])* **Précision du calcul** :

lorsqu'on multiplie le nombre de trapèzes (c'est à dire le nombre de valeurs de x) par 10, l'erreur de calcul est divisée par 10^2 : la méthode des trapèzes a une convergence en $1/N^2$, où N représente le nombre de points.

Fonction cumtrapz

Permet de déterminer numériquement la fonction primitive F d'une fonction f :
F = cumtrapz(y,x)

- x : ensemble de valeurs entre les bornes d'intégration a et b incluses
- y = f(x) : valeurs de la fonction à intéger
- $F(x_i) = \int_a^{x_i} f(t)dt$: ensemble des valeurs de la primitive de f

Il est clairement beaucoup plus économique de calculer en une seule fois l'ensemble des valeurs de la primitive, plutôt que d'appeler la fonction `trapz` pour calculer chacune de ces valeurs
Dans le cas de valeurs de x équidistantes de δ_x, F = cumtrapz(y,dx=δ_x)

Méthode de Gauss-Legendre

peut être omis

L'idée est d'utiliser des points espacés de manière non régulière dans l'intervalle d'intégration. De manière générale l'intégrale est calculée comme :

$$\int_a^b f(x)\mathrm{d}x \approx \frac{b-a}{2}\sum_{i=1}^{n} w_i f(x_i')$$

Les **poids** w_i et les **points** x_i' sont choisis de manière à ce que la méthode donne la **valeur exacte** de l'intégrale pour tous les **polynômes** d'ordre $2n-1$.
Ceci est garanti dès que la méthode donne la valeur exacte de l'intégrale pour tous les monômes x^k, où $k = 0, 1, 2, 3, ..., 2n-1$

Méthode de Gauss-Legendre

Exemple quadrature de Gauss à deux points : $n = 2$
On cherche w_1, x_1', w_2 et x_2' tels que :

$$I = \int_{-1}^{1} f(x)\mathrm{d}x \approx w_1 f(x_1') + w_2 f(x_2') = \tilde{I}$$

Pour l'ensemble des monômes de degré $k = 0, 1, 2, 3 = 2n - 1$ avec :

$$\int_{-1}^{1} x^k \mathrm{d}x = w_1 f(x_1') + w_2 f(x_2')$$

- pour le degré 0, $f(x) = 1$ et $\int_{-1}^{1} \mathrm{d}x = 2 = w_1 + w_2$
- pour le degré 1, $f(x) = x$ et $\int_{-1}^{1} x\mathrm{d}x = 0 = w_1 x_1' + w_2 x_2'$
- pour le degré 2, $f(x) = x^2$ et $\int_{-1}^{1} x^2\mathrm{d}x = \frac{2}{3} = w_1 {x_1'}^2 + w_2 {x_2'}^2$
- pour le degré 3, $f(x) = x^3$ et $\int_{-1}^{1} x^3\mathrm{d}x = 0 = w_1 {x_1'}^3 + w_2 {x_2'}^3$

On a 4 équations avec 4 inconnues. La solution est :

$$w_1 = w_2 = 1, \quad x_1' = -\frac{\sqrt{3}}{3}, \quad x_2' = \frac{\sqrt{3}}{3}, \quad \tilde{I} = f\left(-\frac{\sqrt{3}}{3}\right) + f\left(\frac{\sqrt{3}}{3}\right) = I$$

Méthode de Gauss-Legendre

Les valeurs x_1' et x_2' trouvées correspondent aux zéros du polynôme de Legendre $P_2(x)$.
Rappel : les polynômes de Legendre $P_n(x)$ sont les solutions de l'équation différentielle

$$\frac{d}{dx}\left[(1-x^2)\frac{d}{dx}P_n(x)\right] + n(n+1)P_n(x) = 0$$

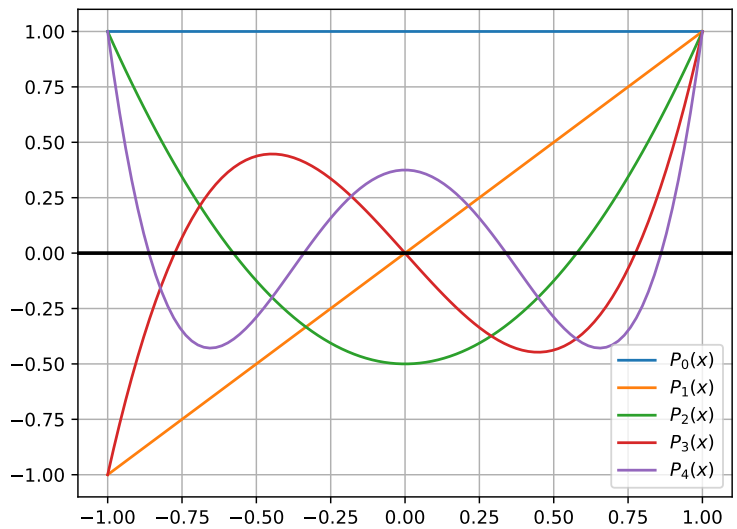
$$P_0(x) = 1, \quad P_1(x) = x, \quad P_2(x) = \frac{1}{2}(3x^2-1), \quad P_3(x) = \frac{1}{2}(5x^3-3x)$$

In [1]:

```
from scipy.special import legendre
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-1, 1, 100)
for n in range(5):
  Pn = legendre(n)
  plt.plot(x, Pn(x), label="$P_"+str(n)+"(x)$")
plt.grid(); plt.legend()
plt.axhline(0, color="k", lw=2)
plt.show()
```

Méthode de Gauss-Legendre

$$P_0(x) = 1, \quad P_1(x) = x, \quad P_2(x) = \frac{1}{2}(3x^2 - 1), \quad P_3(x) = \frac{1}{2}(5x^3 - 3x)$$



Méthode de Gauss-Legendre

Généralisation $n > 2$:

L'intégrale que l'on souhaite calculer est exprimée sous forme :

$$\int_{-1}^{1} f(x)\mathrm{d}x \approx \sum_{i=1}^{n} w_i f(x_i')$$

On peut montrer que cette intégrale est exacte pour tous les polynômes de degré $2n-1$ si :

- x_i' est la i^{eme} racine du polynôme de Legendre P_n ;
- les poids sont donnés par $w_i = \dfrac{2}{(1-x_i'^2)P_n'^2(x_i')}$.

On peut aussi montrer que :

$$\forall n > 1,\ \sum_{i=1}^{n} w_i = (b-a) = 2 \text{ dans le cas présent}$$

Méthode de Gauss-Legendre

Fonction leggauss

La fonction `leggauss` du module de `numpy.polynomial.legendre` permet d'obtenir les poids (array w) et les racines (array x) pour un n donné :

In [2]:
```
1 x, w = np.polynomial.legendre.leggauss(2)
2 print(x)
3 print(w)
```

Out[2]:
```
[-0.57735027  0.57735027]
[1. 1.]
```

Méthode de Gauss-Legendre

Généralisation $n > 2$:

Pour calculer une intégrale entre des bornes quelconques, la méthode est la même avec :

$$\int_a^b f(x)\mathrm{d}x \approx \frac{(b-a)}{2} \sum_{i=1}^{n} w_i f(x_i'')$$

où les poids sont les mêmes que précédemment mais par contre les racines sont :

$$x_i'' = \frac{(b-a)}{2} x_i' + \frac{(a+b)}{2}$$

Méthode de Gauss-Legendre

Généralisation $n > 2$:
La fonction ci-dessous renvoie l'intégrale de $\int_0^{\pi} \sin(x)\mathrm{d}x = 2$ par la méthode de Gauss-Legendre avec 10 points ainsi que l'écart relatif à la valeur exacte :

In [3]:

```
def f(x):
    return np.sin(x)
def trapz(f,a,b,N):
    x=np.linspace(a,b,N+1) # N: nombre d'intervalles
    y=f(x)
    return (x[1]-x[0]) * (np.sum(y) - (y[0]+y[-1])/2 )
def intg(f,a,b,n):
    x, w = np.polynomial.legendre.leggauss(n)
    y=f(x*(b-a)/2 + (a+b)/2) * w
    return np.sum(y) * (b-a)/2

a, b, n = 0, np.pi, 10    # bornes d'integration et nombre d'intervalles
print('Methode de G-L:     ', intg(f,a,b,n), 'erreur', abs((intg(f,a,b,n)-2)/2))
print('Meth des trapezes: ', trapz(f,a,b,n), '
```

Méthode de Gauss-Legendre

```
Out[3]:      Methode de G-L:      2.0000000000000004    erreur
                  2.220446049250313e-16
             Meth des trapezes:   1.983523537509454 6    erreur
                  0.008238231245272676
```

Avec un nombre de points équivalent, l'intégration par cette méthode est beaucoup plus précise qu'avec les méthodes de Newton-Cotes.

Fonction quad

Un outil général d'intégration des fonctions 1D existe dans le module `scipy.integrate` : quad (pour quadrature). Il prend comme argument une fonction et ses bornes puis renvoie l'intégrale et la précision absolue sur cette intégrale.
Syntaxe : *Int, Err = quad(f,a,b)*
Arguments :

- f : nom de la fonction à intégrer
- : a,b : bornes d'intégration

Résultats :

- Int : valeur de l'intégrale
- : Err : estimation de l'incertitude sur Int

On peut aussi écrire : *Int = quad(f,a,b)[0]*

In [4]:

```
from scipy.integrate import quad

a=0
b=np.pi
f=lambda x: np.sin(x)
print(quad(f,a,b))
```

Out [4]: (2.0, 2.220446049250313e-14)

Fonction quad

La fonction (d'une variable) à intégrer peut dépendre de paramètres. Par exemple :

In [5]:
```
def f(t, omega, phi):
    return np.sin(omega*t + phi)
```

Parmi les arguments de la fonction, placer la variable en premier
Dans l'appel à quad, signaler les arguments :
quad(fonction, borne inf, borne sup, args=(omega, phi))
Attention au cas d'un argument unique : quad(fonction, borne inf, borne sup, args=(arg1,))

In [6]:
```
a,b = 0, np.pi
omega = 1.
phi = np.pi
print(quad(f,a,b,args=(omega, phi)))
```

Out[6]:
```
(-1.9999999999999998, 2.2204460492503128e-14)
```

Intégration par la méthode de Monte-Carlo

Il s'agit d'évaluer numériquement la valeur de l'intégrale d'une fonction f définie dans un espace $\mathbb{R}^d$

$$I = \int_{\Omega} f(\mathbf{x}) d\mathbf{x}$$

La vitesse de convergence est un indicateur de l'efficacité de la méthode utilisée pour évaluer l'intégrale.

- Méthode des trapèzes : $n^{-2/d}$
- Méthode de Simpson : $n^{-4/d}$

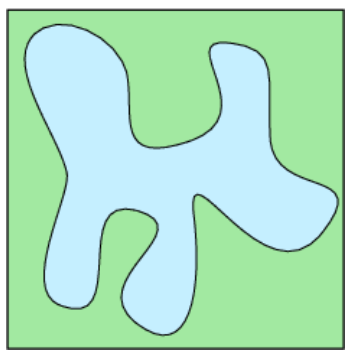
Pour ces méthodes la vitesse de convergence diminue quand d augmente. Ces algorithmes évaluent l'intégrande sur une grille régulière dans $\mathbb{R}^d$.

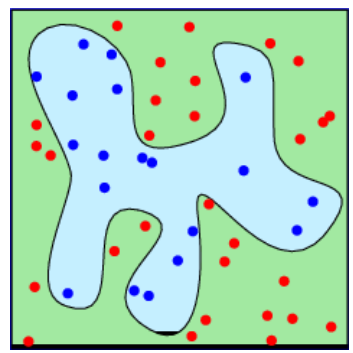
Intégration par la méthode de Monte-Carlo

Méthode vue en L1 : calcul de l'aire d'une surface

- Il s'agit de trouver *manu militari* la valeur de la surface S_L d'un lac à l'intérieur d'un terrain de surface connue S_T.
- On tire N boulets de canon sur le terrain d'une façon aléatoire et homogène et on compte le nombre de boulets M qui sont tombés dans le lac.
- Pour un nombre N très grand on peut estimer la surface du lac comme :

$$S_L = \frac{M}{N} S_T$$





Rappelez-vous, comment calculer π par cette méthode ?

Intégration par la méthode de Monte-Carlo

Méthode de la moyenne.

Dans la méthode Monte-Carlo, les N points $\mathbf{x}_i$ sont répartis aléatoirement dans le domaine d'intégration Ω :

$$x_i \textbf{ est une variable aléatoire et } \mathbf{x_1} \cdots \mathbf{x_N} \in \Omega$$

On définit une fonction densité de probabilité $p(\mathbf{x_i})$ qui donne la probabilité que la variable aléatoire $\mathbf{x_i}$ prenne une valeur comprise entre $\mathbf{x_i}$ et $\mathbf{x_i} + d\mathbf{x}$ de sorte que $\int_\Omega p(\mathbf{x}) d\mathbf{x} = 1$

Or la moyenne d'une fonction continue $f(\mathbf{x})$ où $\mathbf{x}$ est une variable aléatoire suivant la densité de probabilité $p(\mathbf{x})$ est

$$\langle f \rangle_{\mathbf{x}} = \int_\Omega f(\mathbf{x}) p(\mathbf{x}) d\mathbf{x} \approx \frac{\sum_{i=1}^{N} p(\mathbf{x_i}) f(\mathbf{x_i})}{\sum_{i=1}^{N} p(\mathbf{x_i})}$$

Intégration par la méthode de Monte-Carlo

Le volume du domaine d'intégration est : $V = \int_\Omega d\mathbf{x}$
Une densité de probabilité uniforme correspond à une fonction $p(\mathbf{x_i}) = 1/V$. On a alors :

$$\langle f \rangle = \int f(x) \frac{1}{V} dx \approx \frac{\sum_{i=1}^{N} f(\mathbf{x_i})/V}{\sum_{i=1}^{N} 1/V} = \frac{1}{N} \sum_{i=1}^{N} f(\mathbf{x_i})$$

Avec une densité de probabilité $p(\mathbf{x})$ uniforme sur le domaine Ω on peut alors écrire :

$$\frac{V}{N} \sum_{i=1}^{N} f(\mathbf{x_i}) \xrightarrow[N \to \infty]{} I = \int_\Omega f(\mathbf{x}) d\mathbf{x}$$

L'erreur commise sur I est donc proportionnelle à $1/\sqrt{N}$ d'après le théorème central limite, **quelle que soit la dimension d du problème**.
Rappel : https ://fr.wikipedia.org/wiki/Théorème_central_limite

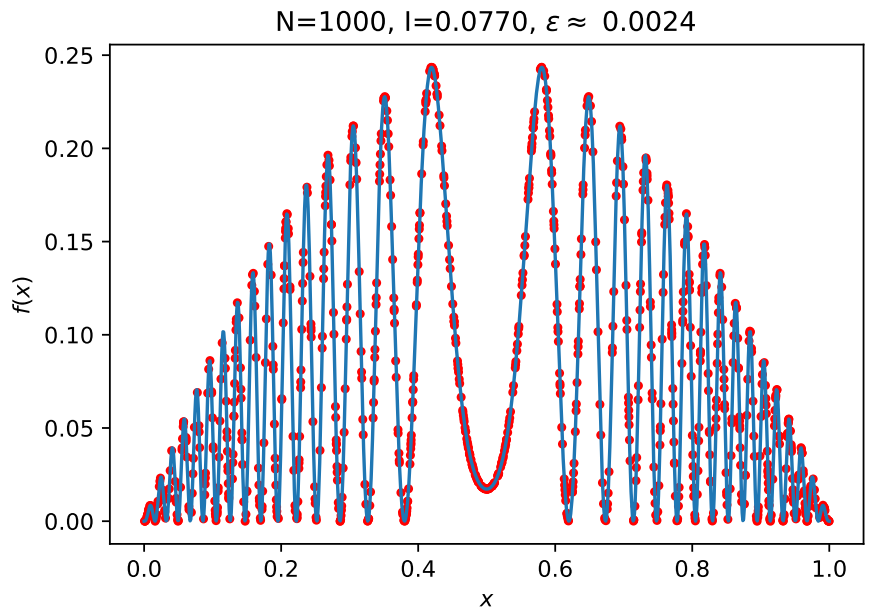
Intégration par la méthode de Monte-Carlo

Intégrale d'une fonction à une variable, par exemple :

$$f(x) = x(1-x)\sin^2\left(200x(1-x)\right), \qquad \int_0^1 f(x)dx \approx 0.080498$$

On applique directement la formule :

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3  from scipy.integrate import quad
4
5  # fonction a integrer
6  def f(x):
7      return x*(1-x) * ( np.sin(200*x*
         (1-x)) )**2
8
9  a,b = 0,1  # bornes d'integration
10 N = 1000    # nombre de points
11 points = np.random.uniform(0,1,size=N)
12 I = (b-a) * np.sum(f(points)) / N
13
14 print("N =",N,": I =",I)
15 print("Scipy:", quad(f,a,b))
16 xx  = np.linspace(a, b, 1000)
17 fig = plt.figure(figsize=(6,4))
18 plt.plot(xx, f(xx))
19 plt.scatter(points,f(points),marker=".
       ",c="red")
20 plt.xlabel('$x$')
21 plt.ylabel('$f(x)$')
22 plt.title(f'N={N}, I={I:.4f}, $\
       epsilon \\approx$ {I/np.sqrt(N)
       :.4f}')
23 plt.show()
```



Intégration par la méthode de Monte-Carlo

A retenir :

- La vitesse de convergence est proportionnelle à $n^{-1/2}$ indépendamment de la dimension du problème.
- La méthode Monte Carlo est ainsi particulièrement utile pour les intégrales à plusieurs dimensions.
- La densité de probabilité uniforme est souvent utilisée mais des lois de probabilité différentes, plus efficaces, peuvent être choisies (voir les notions de Importance Sampling).