TD Info 1 : Calcul numérique - Comment fonctionne notre calculatrice ?

Contents

1 Calculs d'intégrale **2**
- 1.1 Partie "Modélisation" : Méthode des rectangles . . . 2
- 1.2 Partie "Informatique" . . . 6
- 1.3 Méthode des trapèzes (facultative) . . . 8

2 Interpoler des données numériques : les polynômes de Lagrange **11**
- 2.1 Pourquoi interpoler ? . . . 11
- 2.2 Un peu de théorie . . . 13

3 Calculs de racine carrées : méthode de Héron (facultative) **20**
- 3.1 De Héron à aujourd'hui : un peu d'histoire et de la géométrie… . . . 20
- 3.2 Réalisation de la fonction python . . . 21

[1]:
```
import numpy as np
import math
import matplotlib.pyplot as plt
```

Notre calculatrice est capable de nombreuses opérations complexes, de façon quasi-instantanée. Pour cela, elle utilise son microprocesseur, capable de faire "directement" des opérations logiques et mathématiques très simples comme l'addition, la soustraction et la multiplication d'entiers (en passant par l'écriture binaire). Vous avez peut-être déjà vu comment l'addition d'entier s'effectue en binaires : par exemple, on a $1011 + 0110 = 10001$. Pour des calculs plus complexes, par exemple la division, la racine carré, le modulo, et les calculs avec des nombres flottants, le microprocesseur est doté de FPU, unités de calcul en virgule flottante. Comment peuvent-être implémentés ces calculs ?

Durant ce travail, nous allons aborder des méthodes de calculs numériques importantes. Le but est de ré-utiliser les connaissances préalablement acquises (fonction, manipulation des listes, boucles, …), de voir des algorithmes utiles, de parler de leur convergence, et d'entrevoir la récursivité.

Question préliminaire : On dispose d'une liste de nombre `L`. Écrire le code permettant de faire la somme puis le produit des nombres contenus dans L.

Correction :

[2]:
```
from random import randint

L = [randint(1, 9) for i in range(20)]

somme = 0
produit = 1
for elem in L :
    somme += elem
    produit *= elem

print(L,)
print("Somme : ", somme)
print("Produit : ", produit)
```

```
[6, 4, 4, 6, 9, 8, 4, 7, 8, 1, 1, 1, 6, 3, 7, 2, 7, 3, 7, 9]
Somme :  103
Produit :  3097158156288
```

1 Calculs d'intégrale

On considère une fonction f d'une seule variable x, dont on veut calculer l'intégrale entre les bornes a et b (ces deux bornes sont incluses dans l'intervalle de définition de la fonction, il ne s'agit pas d'une intégrale impropre !) :

$$I = \int_a^b f(x)\,\mathrm{d}x$$

Cette valeur est évidemment intéressante pour des intégrales que nous ne sommes pas capables de calculer théoriquement, mais aussi par exemple pour calculer des valeurs moyennes de fonction :

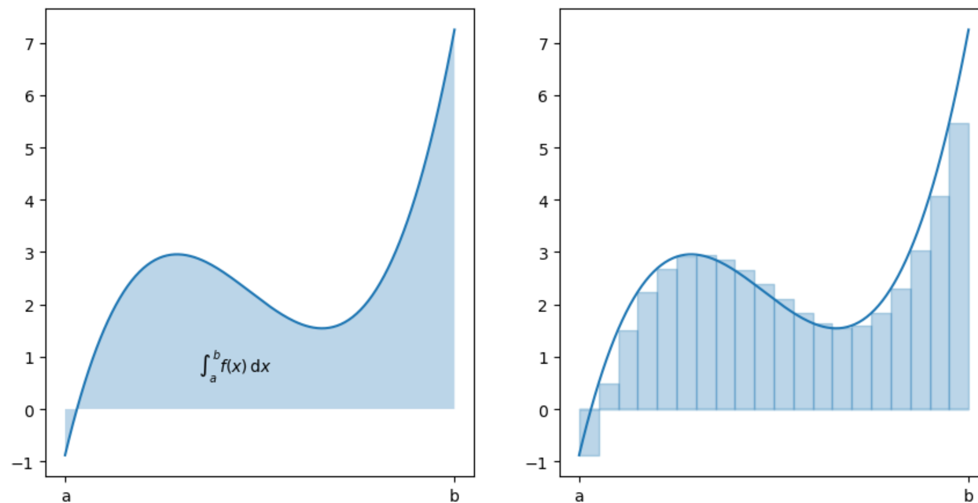
$$< f >= \frac{1}{b-a}\int_a^b f(x)\,\mathrm{d}x$$

1.1 Partie "Modélisation" : Méthode des rectangles

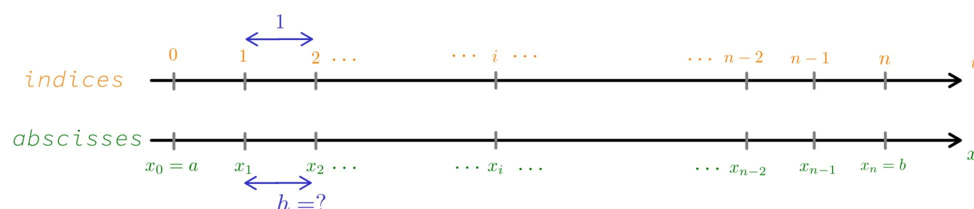
Il s'agit de la méthode la plus simple pour déterminer la valeur numérique d'une intégrale. Elle repose sur le fait que l'intégrale est aussi l'aire située sous la courbe représentant cette fonction.

[3]:
```
# Fonction à intégrer :
def f(x):
    return (x-2.5)**3 -1.5*x +6
# Bornes :
a = 0.7
b = 4.5

plt.figure(figsize=(7*width,7*width*0.5))

plt.subplot(121)
x = np.linspace(a, b, 200)
```

```
y = f(x)
plt.plot(x,y) # graphe de la fonction
plt.fill_between(x,y, alpha=0.3) # graphe de la fonction
plt.xticks([a,b],['a','b'])
plt.text(2, 0.7, r'$\int_a^b f(x)\, \mathrm{d} x$')

plt.subplot(122)
plt.plot(x,y) # graphe de la fonction
# Graphe des rectangles :
n = 20
x_r = np.linspace(a, b,n+1)
y_r = f(x_r)
for i in range(n): # graphe des rectangles à droite
    x_rect = [x_r[i], x_r[i], x_r[i+1], x_r[i+1], x_r[i]] # abscisses des sommets
    y_rect = [0   , y_r[i], y_r[i]  , 0     , 0   ] # ordonnees des sommets
    plt.fill_between(x_rect, y_rect,color = 'tab:blue', alpha = 0.3)
plt.xticks([a,b],['a','b'])

plt.show()
```



On va alors approximer cette aire sous la courbe par la somme des aires des rectangles représentée ci-dessus : il s'agit de la première étape de la méthode des rectangles, une étape de **discrétisation**. Pour cela, on divise l'intervalle continu $[a, b]$ en n sous-intervalles.



Chacun de ces sous-intervalles a pour longueur h, nommé **pas** de discrétisation.

Question 1 : Donner l'expression de h en fonction de a, b, et n.

Correction 1 : $h = \frac{b-a}{n}$.

Les valeurs de x sont discrétisées, avec les $n+1$ valeurs suivantes : $x_0 = a$, x_1, x_2, ...,x_{n-1},$x_n = b$. L'entier i est l'index des positions. Pour mieux comprendre cette discrétisation, on peut prendre l'exemple suivant : dans un ascenseur, pour déterminer la position finale qu'on veut atteindre, on utilise le numéro de l'étage, il s'agit de l'index (ou indice en bon français). On n'indique pas, par exemple, l'altitude exacte de l'étage ! L'altitude z, a priori variable *continue*, ne pourra en réalité ne prendre que certaines valeurs, par exemple uniquement l'altitude du sol de chaque étage. On peut alors définir la suite Z_n, suite des valeurs d'altitude pour chaque étage n.

Question 2 : Donner l'expression de x_i pour $0 \leqslant i \leqslant n$, en fonction de a, b, i et n.

Correction 2 : $x_i = a + ih = a + i\frac{b-a}{n}$. Ainsi on a bien $x_0 = a + h * 0 = a$ et $x_n = a + (b - a) = b$.

Dans la suite, nous allons remplacer la fonction f par des valeurs prises par cette même fonction, en certains points seulement : nous remplaçons bien une fonction continue par des valeurs discrètes (cela revient à passer d'une fonction à une suite). Le terme "discret" s'oppose ici à continue : la variable x ne varie plus continuement, mais prend des valeurs discrètes par pas de h; et de même pour la fonction $f(x)$.

L'intégrale s'écrit, en utilisant les sous-intervalles :

$$I = \int_a^b f(x)\,\mathrm{d}x = \int_{x_0}^{x_1} f(x)\,\mathrm{d}x + \int_{x_1}^{x_2} f(x)\,\mathrm{d}x + \ \dots\ + \int_{x_{n-1}}^{x_n} f(x)\,\mathrm{d}x$$

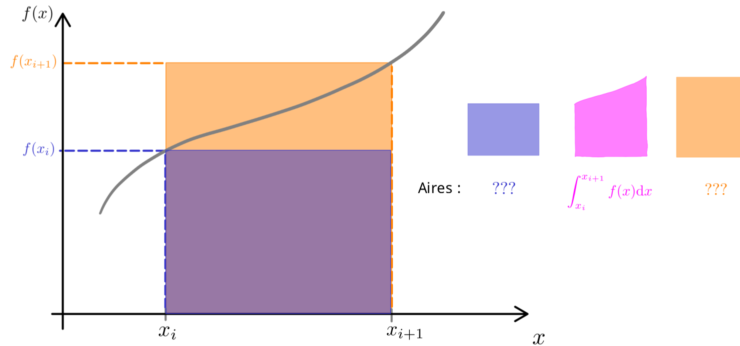
Il faut alors estimer la valeur de l'intégrale $\int_{x_i}^{x_{i+1}} f(x)\,\mathrm{d}x$ pour $i \in [\![0, n-1]\!]$: nous allons donner comme valeur approximative l'aire du ième rectangle. Mais comment choisir ce rectangle ?

On remplace, sur chaque sous-intervalle, la fonction f par une constante : ainsi on forme bien un rectangle, la valeur constante choisie étant sa hauteur. Pour choisir cette valeur constante, on utilise une valeur prise par la fonction sur l'intervalle considérée. Deux choix seront ici étudiés et proposés pour déterminer cette valeur, sur l'intervalle $[x_i, x_{i+1}]$. On pourra se contenter d'un seul des deux !

Méthode des rectangles à gauche : On peut prendre la valeur de la fonction f au début de cet intervalle : $f(x_i)$. L'intégrale sur ce sous-intervalle s'écrit alors $\int_{x_i}^{x_{i+1}} f(x)\,\mathrm{d}x \simeq \int_{x_i}^{x_{i+1}} f(x_i)\,\mathrm{d}x = (x_{i+1} - x_i)f(x_i) = hf(x_i)$. Cette valeur d'intégrale peut être vue comme l'aire d'un rectangle (voir figure plus loin), de largeur h et de hauteur $f(x_i)$.

Méthode des rectangles à droite : On peut prendre la valeur de la fonction f à la fin de cet intervalle : $f(x_{i+1})$. L'intégrale sur ce sous-intervalle s'écrit alors $\int_{x_i}^{x_{i+1}} f(x)\,\mathrm{d}x \simeq \int_{x_i}^{x_{i+1}} f(x_{i+1})\,\mathrm{d}x = (x_{i+1} - x_i)f(x_{i+1}) = hf(x_{i+1})$. Cette valeur d'intégrale peut être vue comme l'aire d'un rectangle (voir figure plus loin), de largeur h et de hauteur $f(x_{i+1})$.

Pour le ième rectangle :



Question 3 : Donner les aires des deux rectangles formés.

Correction 3 : Aire du ième rectangle "à gauche" : $h \times f(x_i) = hf(a+ih)$. Aire du ième rectangle "à droite" : $h \times f(x_{i+1}) = hf(a+(i+1)h)$.

Pour obtenir la valeur numérique de l'intégrale sur tout l'intervalle I, il reste à faire la somme des aires des différents rectangles.

Question 4 : Écrire l'intégrale I comme somme des aires des rectangles, en précisant quelle méthode des rectangles est employée (gauche ou droite), en fonction de f, a, b, i et n.

Correction 4 : Intégrale I, écrite ici pour la méthode des rectangles à gauche :

$$I = \int_a^b f(x)\,\mathrm{d}x = \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} f(x)\,\mathrm{d}x \approx \sum_{i=0}^{n-1} hf(x_i)$$

Ou plus précisément, en écrivant que $x_i = x_0 + ih = a + ih$:

$$I = \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} f(x)\,\mathrm{d}x \approx h \sum_{i=0}^{n-1} f\left(a+ih\right) = \frac{b-a}{n} \sum_{i=0}^{n-1} f\left(a + i\frac{b-a}{n}\right)$$

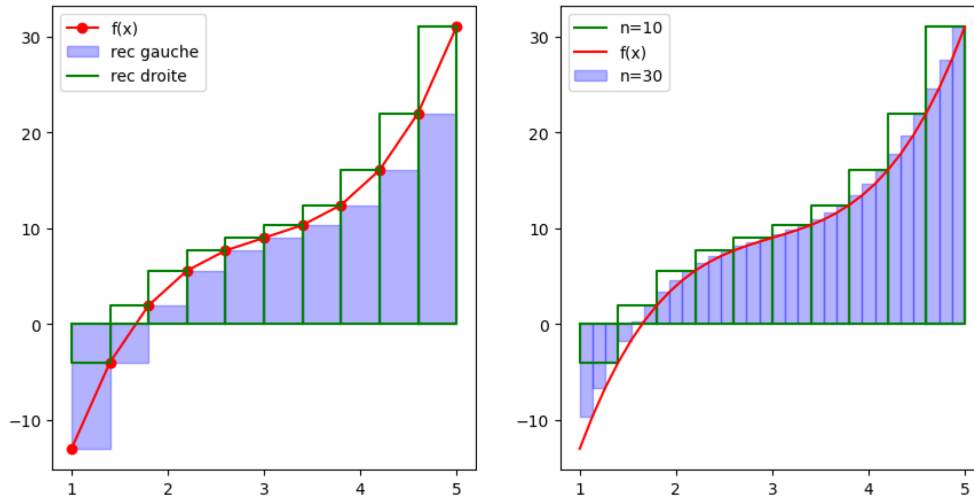
Pour la méthode des rectangles à droite :

$$I = \int_a^b f(x)\,\mathrm{d}x \approx \sum_{i=0}^{n-1} hf(x_{i+1}) = \sum_{i=1}^{n} hf(x_i)$$

Graphiquement, voici les deux premières méthodes pour une fonction, ainsi que la même méthode (à droite) en variant le nombre de rectangles :

[4]:
```
a = 1
b = 5
n = 10
x = np.linspace(a, b, n+1)
y = 2*(x-3)**3 +3*x

plt.figure(figsize=(7*width,7*width*0.5))

plt.subplot(121)
```

```
plt.plot(x,y,"ro-", label = 'f(x)') # graphe de la fonction
for i in range(n-1): # graphe des sommets rectangles à gauche
    x_rect = [x[i], x[i+1]] # abscisses des sommets
    y_rect = [ y[i], y[i]] # ordonnees des sommets
    plt.fill_between(x_rect, y_rect, color='blue', alpha=0.3)
x_rect = [x[n-1], x[n]] # abscisses des sommets
y_rect = [ y[n-1], y[n-1]] # ordonnees des sommets
plt.fill_between(x_rect, y_rect, color='blue', alpha=0.3,label='rec gauche')
for i in range(n): # graphe des rectangles à droite
    x_rect = [x[i], x[i], x[i+1], x[i+1], x[i]] # abscisses des sommets
    y_rect = [0   , y[i+1], y[i+1]  , 0     , 0   ] # ordonnees des sommets
    plt.plot(x_rect, y_rect,"g")
plt.plot(x_rect, y_rect,"g", label='rec droite')
plt.legend()

plt.subplot(122)
for i in range(n): # graphe des rectangles à droite
    x_rect = [x[i], x[i], x[i+1], x[i+1], x[i]] # abscisses des sommets
    y_rect = [0   , y[i+1], y[i+1]  , 0     , 0   ] # ordonnees des sommets
    plt.plot(x_rect, y_rect,"g")
plt.plot(x_rect, y_rect,"g", label='n=10')

n = 31
x = np.linspace(a, b, n)
y = 2*(x-3)**3 +3*x
plt.plot(x,y,'r',label = 'f(x)') # graphe de la fonction

for i in range(n-2): # graphe des sommets rectangles à droite
    x_rect = [x[i], x[i+1]] # abscisses des sommets
    y_rect = [ y[i+1], y[i+1]] # ordonnees des sommets
    plt.fill_between(x_rect, y_rect, color='blue', alpha=0.3)
x_rect = [x[n-2], x[n-1]] # abscisses des sommets
y_rect = [ y[n-1], y[n-1]] # ordonnees des sommets
plt.fill_between(x_rect, y_rect, color='blue', alpha=0.3,label='n=30')
plt.legend()
plt.show()
```



Graphiquement, on comprend aisément que plus n est grand, plus h est petit, et la méthode est précise. Elle devient exacte dans la limite $n \to \infty$, soit $h \to 0$:

$$I = \int_a^b f(x)\,\mathrm{d}x = \lim_{n\to\infty} \frac{b-a}{n} \sum_{i=0}^{n-1} f\left(a + i\frac{b-a}{n}\right)$$

On retrouve alors que l'aire sous la courbe (comptée algébriquement), soit la somme des aires des rectangles, est aussi la valeur de l'intégrale.

1.2 Partie "Informatique"

Question 5 : Écrire la fonction `int_rec(f:callable, a:float, b:float, n:int) -> float` permettant de donner la valeur numérique de l'intégrale de la fonction $f(x)$ entre les bornes a et b, en utilisant la méthode des rectangles de votre choix, avec n rectangles.

Correction 5 :

```
[5]: def int_rec_g(f:callable, a:float, b:float, n:int)->float :

         """ Retourne la valeur numérique de l'intégrale de la fonction f, entre les␣
      ↪bornes a et b, avec n rectangles à gauche """

         h = (b-a)/n # pas
         Int = 0 # valeur initiale de l'intégrale avant le premier rectangle

         for i in range(n): # pour chaque rectangle
             airei = h*f(a+i*h) # aire du ième rectangle
             Int += airei # on ajoute son aire à la somme des aires déjà calculée

         return Int

     def int_rec_d(f:callable, a:float, b:float, n:int)->float :
```

```
    """ Retourne la valeur numérique de l'intégrale de la fonction f, entre les␣
↪bornes a et b, avec n rectangles au milieu """

    h = (b-a)/n # pas
    Int = 0 # valeur initiale de l'intégrale avant le premier rectangle

    for i in range(n): # on ajoute l'aire des rectangles, un à un
        Int += h*f(a+(i+1)*h)

    return Int
```

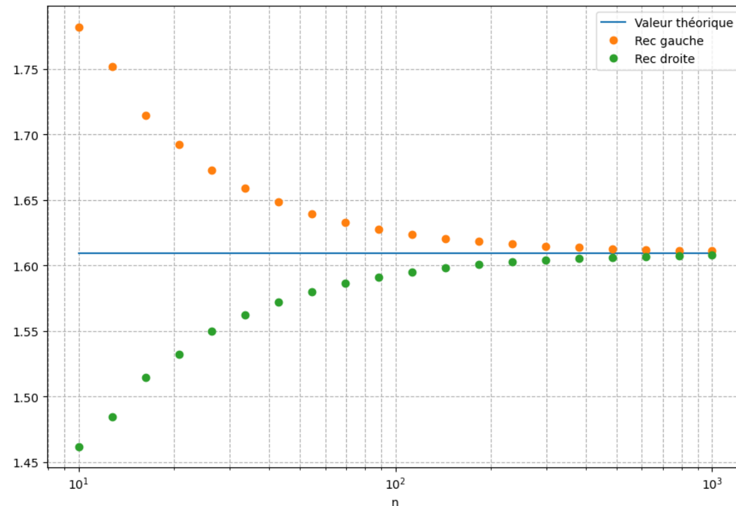
Question 6 : Grâce au code fournit dans le fichier *I1_Num.py*, prendre un exemple de fonction dont vous êtes capables de donner l'intégrale, et tester votre code. On ne prendra pas une fonction trop simple …. (la méthode des rectangles peut d'ailleurs donner un résultat exacte pour certaines fonctions… lesquelles ?). Il sera nécessaire de rentrer cette fonction dans python sous forme d'une …. fonction ! Faire varier la valeur du nombre de rectangles : qu'observe-t-on ?

Question 7 : Compléter le code permettant de représenter, sur un graphe, la convergence du résultat de votre méthode des rectangles : on représentera la valeur retournée en fonction du nombre de rectangles n.

Correction 6/7 :

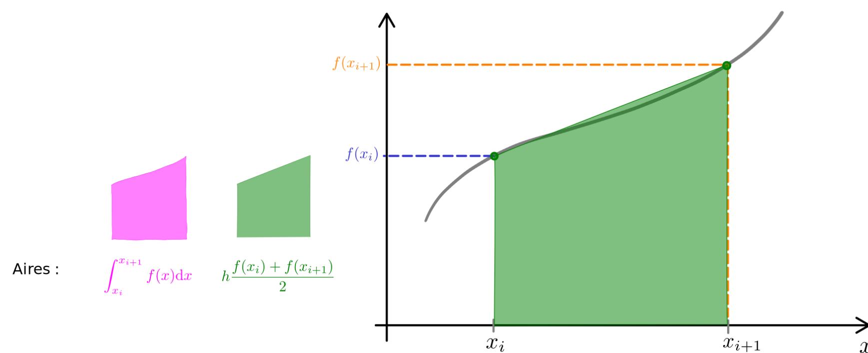
[6]:
```
# Fonction que nous savons intégrer :
def fonction_test (x):
    return 1/x
# Bornes d'intégration :
a = 1
b = 5
# Résultat théorique
R_th = math.log(5)

# Liste des valeurs de n, allant de 10 à 1000 en échelle log
N = np.logspace(1,3,20)

plt.figure(figsize=(7*width,7*width*2/3))

# valeur théorique
plt.plot([10,1000],[R_th,R_th],label='Valeur théorique')

# Valeurs calculées par méthodes des rectangles gauche et droite :
Int_num_g, Int_num_d = [], []
for n in N :
    Int_num_g.append(int_rec_g(fonction_test,a,b,int(n)))
    Int_num_d.append(int_rec_d(fonction_test,a,b,int(n)))
# Graphes :
plt.plot(N,Int_num_g,'o', label='Rec gauche')
plt.plot(N,Int_num_d,'o', label='Rec droite')
```

```
plt.xscale('log')
plt.xlabel('n')
plt.grid(True,which="both", linestyle='--')
plt.legend()

plt.show()
```



1.3 Méthode des trapèzes (facultative)

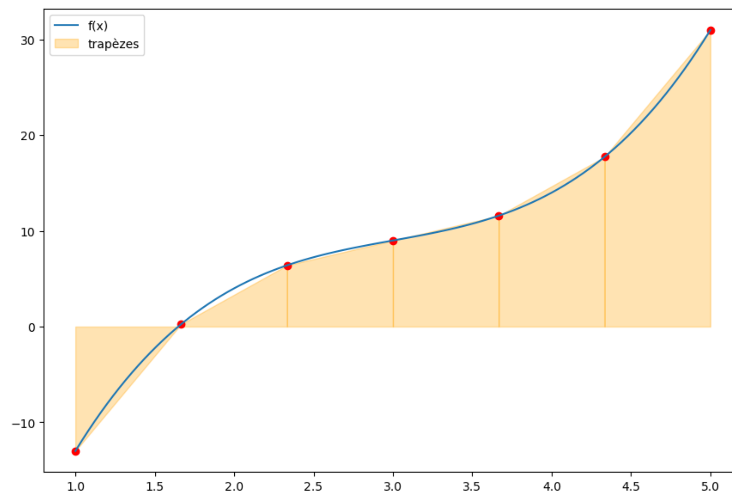
La méthode des trapèzes est similaire à celle des rectangles, mais dans cette méthode, on approxime la fonction f sur l'intervalle de x_i à x_{i+1} par une fonction affine (au lieu d'une fonction constante). La fonction affine étant la *meilleure* approximation de la fonction f est celle passant par les points $(x_i, f(x_i))$ et $(x_{i+1}, f(x_{i+1}))$:



Pour l'intégrale totale :

[7]:
```
a = 1
b = 5
n = 6
x = np.linspace(a, b, n+1)
y = 2*(x-3)**3 +3*x
```

```python
plt.figure(figsize=(7*width,7*width*2/3))

plt.plot(x,y,"ro") # graphe de la fonction aux points discrétisés
for i in range(n-1): # graphe des sommets des trapèzes
    x_rect = [x[i], x[i+1]] # abscisses des sommets
    y_rect = [ y[i], y[i+1]] # ordonnees des sommets
    plt.fill_between(x_rect, y_rect, color='orange', alpha=0.3)
x_rect = [x[n-1], x[n]] # abscisses des sommets
y_rect = [ y[n-1], y[n]] # ordonnees des sommets

n = 100
x = np.linspace(a, b, n+1)
y = 2*(x-3)**3 +3*x
plt.plot(x,y,label = 'f(x)') # graphe de la fonction aux point

plt.fill_between(x_rect, y_rect, color='orange', alpha=0.3,label='trapèzes')
plt.legend()
plt.show()
```



Remarque : La méthode des trapèzes peut aussi être vu comme la méthode des rectangles, mais avec un rectangle de hauteur la moyenne des hauteurs gauche et droite : $\frac{f(x_i)+f(x_{i+1})}{2}$. A ne pas confondre avec la méthode des rectangles milieu, pour laquelle la hauteur des rectangles est la valeur prise par la fonction à la moyenne des abscisses : $f\left(\frac{x_i+x_{i+1}}{2}\right)$. En général, les résultats sont en réalité peu différents...avec un avantage pour la méthode des trapèzes !

Question 8 : Écrire la fonction `int_trap(f,a,b,n)` permettant de donner la valeur numérique de l'intégrale de la fonction $f(x)$ entre les bornes a et b, en utilisant la méthode des trapèzes, avec n trapèzes.

Correction 8 :

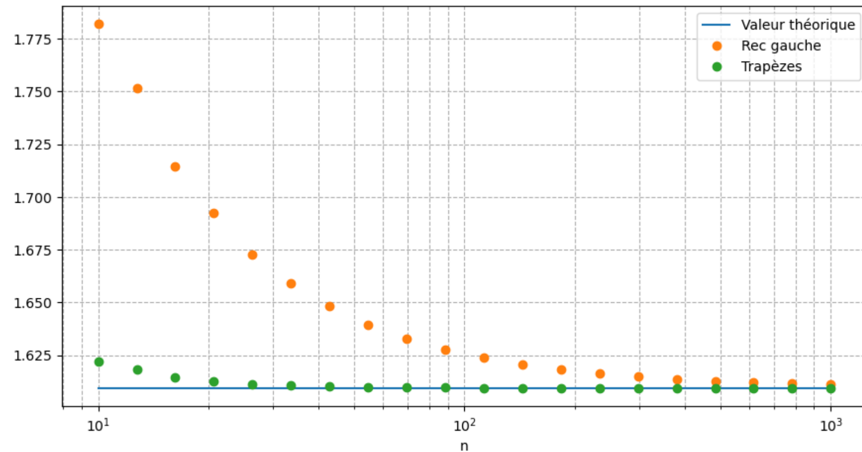
[8]:
```python
def int_trap(f,a,b,n) :
```

```
    """ Retourne la valeur numérique de l'intégrale de la fonction f, entre les␣
↪bornes a et b, avec n trapèzes """

    h = (b-a)/n # pas
    Int = 0 # valeur initiale de l'intégrale avant le premier trapèze

    for i in range(n): # on ajoute l'aire des rectangles, un à un
        Int += h*(f(a+i*h)+f(a+(i+1)*h))/2

    return Int
```

Question 9 : Comparer la convergence de la méthode des trapèzes et de la méthode des rectangles en complétant le code.

Correction 9 :

[9]:
```
plt.figure(figsize=(7*width,7*width*0.5))

# Liste des valeurs de n, allant de 10 à 1000 en échelle log
N = np.logspace(1,3,20)
# valeur théorique
plt.plot([10,1000],[R_th,R_th],label='Valeur théorique')
# Méthodes des rectangles
Int_num_rg, Int_num_t = [], []
for n in N :
    Int_num_rg.append(int_rec_g(fonction_test,a,b,int(n)))
    Int_num_t.append(int_trap(fonction_test,a,b,int(n)))
plt.plot(N,Int_num_rg,'o', label='Rec gauche')
plt.plot(N,Int_num_t,'o', label='Trapèzes')
plt.xscale('log')
plt.xlabel('n')
plt.grid(True,which="both", linestyle='--')
plt.legend()

plt.show()
```

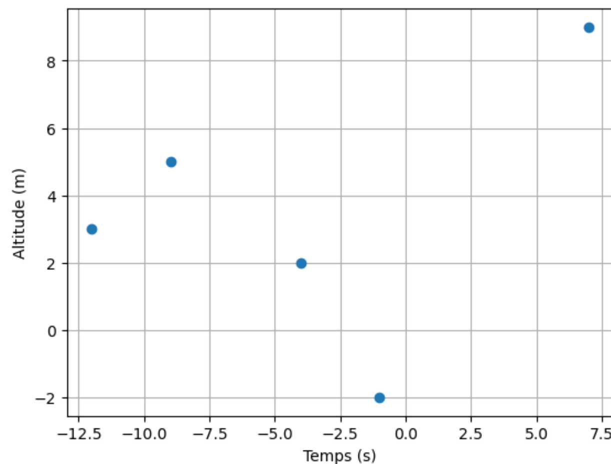


2 Interpoler des données numériques : les polynômes de Lagrange

2.1 Pourquoi interpoler ?

Considérons les points expérimentaux suivants, acquis lors d'un T.P. :

[10]:
```
T = [-12, -9, -4, -1, 7]
Z = [3, 5, 2, -2, 9]

plt.figure()
plt.plot(T,Z,'o')
plt.xlabel("Temps (s)")
plt.ylabel("Altitude (m)")
plt.grid()
plt.show()
```

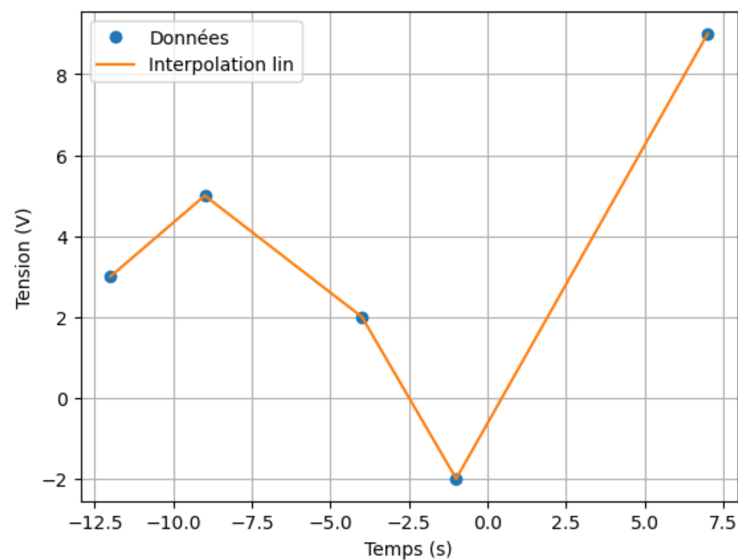


On cherche alors à obtenir la meilleure estimation possible de l'altitude en d'autres points, sans faire de nouvelles mesures : il s'agit ici d'**interpoler** les points expérimentaux obtenus (en nombre finis) par une fonction. On aura alors accès à la valeur de l'altitude interpolée en un nombre infini de points. Évidemment ces valeurs n'auront pas la valeur des *vrais* points expérimentaux, ils seront

seulement une interpolation à partir de données incomplètes.

Un des moyens les plus simples est d'interpoler ces points en traçant entre chaque un segment : c'est une interpolation linéaire (plus précisément : affine par morceaux).

[11]:
```
plt.figure()
plt.plot(T,Z,'o',label='Données')
plt.plot(T,Z,label='Interpolation lin')
plt.xlabel("Temps (s)")
plt.ylabel("Tension (V)")
plt.grid()
plt.legend()
plt.show()
```



Efficace, mais essayons de faire mieux !

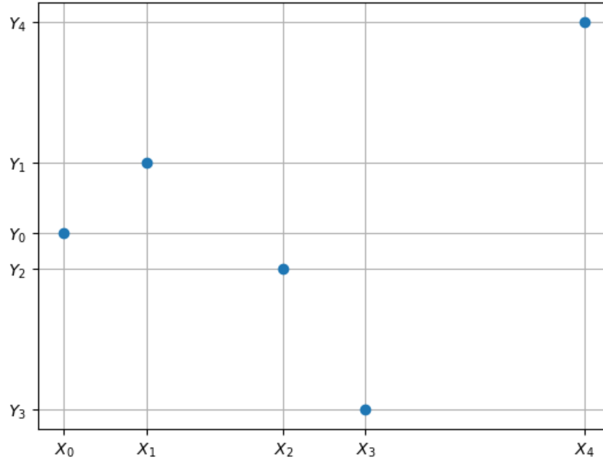
2.2 Un peu de théorie

Voici les notations générales que nous allons employer :

- les données sont les n points (X_i, Y_i) pour $i \in [\![0, n-1]\!]$
- on cherche la "meilleure" fonction d'interpolation f qui passe par tous les points (X_i, Y_i) : $f(X_i) = Y_i, \ \forall i \in [\![0, n-1]\!]$

Voyons ces notations sur l'exemple précédent :

[12]:
```
plt.figure()
plt.plot(T,Z,'o')
plt.xticks(T,[r'$X_0$', r'$X_1$',  r'$X_2$', r'$X_3$',r'$X_4$'])
plt.yticks(Z,[r'$Y_0$', r'$Y_1$',  r'$Y_2$', r'$Y_3$',r'$Y_4$'])
plt.grid()
plt.show()
```



On cherche une fonction f qui permettrait une "meilleure" interpolation qu'une "simple" fonction affine par morceaux, mais tout en restant assez simple. Une fonction affine étant une fonction polynomiale de degré 1, on peut penser à un polynôme de degré supérieur. Quel degré considérer ?

On peut montrer que le polynôme le plus simple (c'est-à-dire ici de plus bas degré) passant exactement par n points est de degré $d = n - 1$, et il est unique (tiens, comment peut-on montrer cette propriété ... ?) . Pour le construire, on va considérer les polynômes de Lagrange.

Question 10 : On construit le polynôme de Lagrange suivant :

$$P_0(x) = \frac{(x - X_1)(x - X_2)...(x - X_{n-2})(x - X_{n-1})}{(X_0 - X_1)(X_0 - X_2)...(X_0 - X_{n-2})(X_0 - X_{n-1})} = \frac{\prod_{j \in [\![1,n-1]\!]} x - X_j}{\prod_{j \in [\![1,n-1]\!]} X_0 - X_j} = \prod_{j \in [\![1,n-1]\!]} \frac{x - X_j}{X_0 - X_j}$$

Quel est son degré ? Quelles valeurs prend-il en X_j pour $j \in [\![0, n-1]\!]$? Compléter alors la phrase suivante : P_0 est le polynôme de degré ...... qui s'annule pour tous les X_j, sauf en ..... où il vaut ..... .

Correction 10 : P_0 est de degré $d = n - 1$, et on a : $P_0(X_0) = 1$, $P_0(X_1) = 0, \dots, P_0(X_{n-1}) = 0$. Soit :

$$P_0(X_j) = \begin{cases} 1 & \text{pour } j = 0 \\ 0 & \text{pour les autres valeurs possibles de } j \end{cases}$$

P_0 est le polynôme de degré $n - 1$ qui s'annule pour tous les X_j, sauf en X_0 où il vaut 1.

Question 11 : On introduit le polynome P_1, sur le modèle de P_0. Compléter alors la phrase : P_1 est le polynôme de degré ...... qui s'annule pour tous les X_j, sauf en ..... où il vaut ..... . Exprimer alors le polynôme P_1, puis P_i pour $i \in [\![0, n-1]\!]$.

Correction 11 : P_1 est le polynôme de degré $n - 1$ qui s'annule pour tous les X_j, sauf en X_1 où il vaut 1. On cherche donc à avoir $P_1(X_0) = 0$, $P_1(X_1) = 1$, $P_1(X_2) = 0, \dots, P_0(X_{n-1}) = 0$. Soit :

$$P_1(X_j) = \begin{cases} 1 & \text{pour } j = 1 \\ 0 & \text{pour les autres valeurs possibles de } j \end{cases}$$

Le polynôme à considérer, d'après la question précédente, est donc :

$$P_1(x) = \frac{(x - X_0)(x - X_2)...(x - X_{n-2})(x - X_{n-1})}{(X_1 - X_0)(X_1 - X_2)...(X_1 - X_{n-2})(X_1 - X_{n-1})} = \frac{\prod_{j \in [\![0,n-1]\!], j \neq 1} x - X_j}{\prod_{j \in [\![0,n-1]\!], j \neq 1} X_1 - X_j} = \prod_{j \in [\![0,n-1]\!], j \neq 1} \frac{x - X_j}{X_1 - X_j}$$

On cherche à avoir $P_i(X_0) = 0, \ldots, P_i(X_{i-1}) = 0$, $P_i(X_i) = 1$, $P_1(X_{i+1}) = 0, \ldots, P_0(X_{n-1}) = 0$. Soit :

$$P_i(X_j) = \begin{cases} 1 & \text{pour } j = i \\ 0 & \text{pour les autres valeurs possibles de } j \end{cases}$$

Le polynôme à considérer est donc :

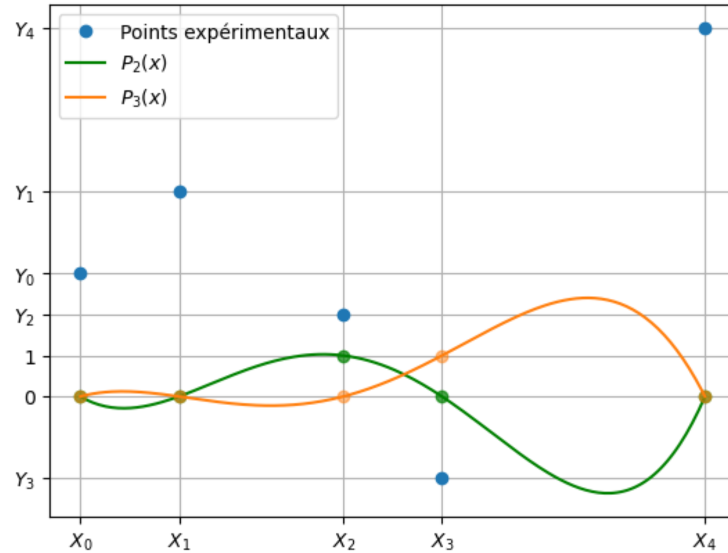
$$P_i(x) = \frac{(x - X_0)(x - X_1)...(x - X_{i-1})(x - X_{i+1})...(x - X_{n-2})(x - X_{n-1})}{(X_i - X_0)(X_i - X_1)...(X_i - X_{i-1})(X_i - X_{i+1})...(X_i - X_{n-2})(X_i - X_{n-1})}, \ \forall i \in [\![0, n-1]\!]$$

Ou bien :

$$P_i(x) = \frac{\prod_{j\in[\![0,n-1]\!],j\neq i} x - X_j}{\prod_{j\in[\![0,n-1]\!],j\neq i} X_i - X_j}, \ \forall i \in [\![0, n-1]\!] \prod_{j\in[\![0,n-1]\!],j\neq i} \frac{x - X_j}{X_i - X_j}$$

On dispose dorénavant de n polynômes de degré $d = n - 1$, qui prennent respectivement ("chacun à leur tour") la valeur 1 aux abscisses X_i (et la valeur 0 ailleurs). Voyons avec notre exemple initial deux exemples de ces polynômes, P_2 et P_3 :

[13]:
```
T_abs = np.linspace(-12, 7, 200)
n = len(T)
P2, P3 = np.ones(200), np.ones(200)
for j in range(n) :
    if j != 2 :
        P2 = P2 * (T_abs - T[j]) / (T[2] - T[j])
    if j != 3 :
        P3 *= (T_abs - T[j]) / (T[3] - T[j])

plt.figure()
plt.plot(T,Z,'o', label = "Points expérimentaux")
plt.plot(T_abs, P2,color='g', label=r'$P_2 (x)$')
plt.plot(T, [0,0,1,0,0],'o',color='g', alpha=0.7)
plt.plot(T_abs, P3,color='tab:orange', label=r'$P_3 (x)$')
plt.plot(T, [0,0,0,1,0],'o',color='tab:orange', alpha=0.5)
plt.xticks(T,[r'$X_0$', r'$X_1$',  r'$X_2$', r'$X_3$',r'$X_4$'])
plt.yticks(Z+[0,1],[r'$Y_0$', r'$Y_1$',  r'$Y_2$', r'$Y_3$',r'$Y_4$','0','1'])
plt.legend()
plt.grid()
plt.show()
```

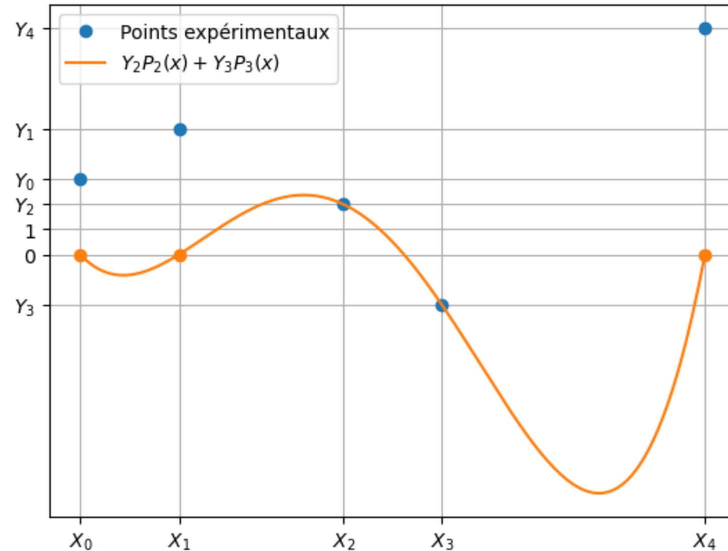


P_2 s'annule bien en tout point des données expérimentales, sauf en X_2 où il prend la valeur 1. De même, P_3 s'annule bien en tout point des données expérimentales, sauf en $X_3 = 3$ où il prend la valeur 1.

Question 12 : Quel polynôme, créé à partir de P_2 et P_3, va prendre les "bonnes" valeurs aux points 2 et 3 (d'abscisses respectives X_2 et X_3), c'est-à-dire les valeurs respectives Y_2 et Y_3; et 0 aux autres points de données ?

Correction 12 : Le polynôme $Y_2 \times P_2 + Y_3 \times P_3$. correspond (voir graphe ci-dessous).

[14]:
```
plt.figure()
plt.plot(T,Z,'o', label = "Points expérimentaux")
plt.plot(T_abs, Z[2]*P2+Z[3]*P3,color='tab:orange', label=r'$Y_2P_2 (x)+Y_3P_3␣
↪(x)$')
plt.plot([-12, -9, 7], [0,0,0],'o',color='tab:orange')
plt.xticks(T,[r'$X_0$', r'$X_1$',  r'$X_2$', r'$X_3$',r'$X_4$'])
plt.yticks(Z+[ 0, 1],[r'$Y_0$', r'$Y_1$',  r'$Y_2$', r'$Y_3$',r'$Y_4$', '0',␣
↪'1'])
plt.legend()
plt.grid()
plt.show()
```



Question 13 : En déduire l'expression de la fonction f, fonction polynomiale de degré $d = n-1$, qui passe par les n points (X_i, Y_i). Cette fonction utilisera les polynômes de Lagrange P_i définis précédemment.

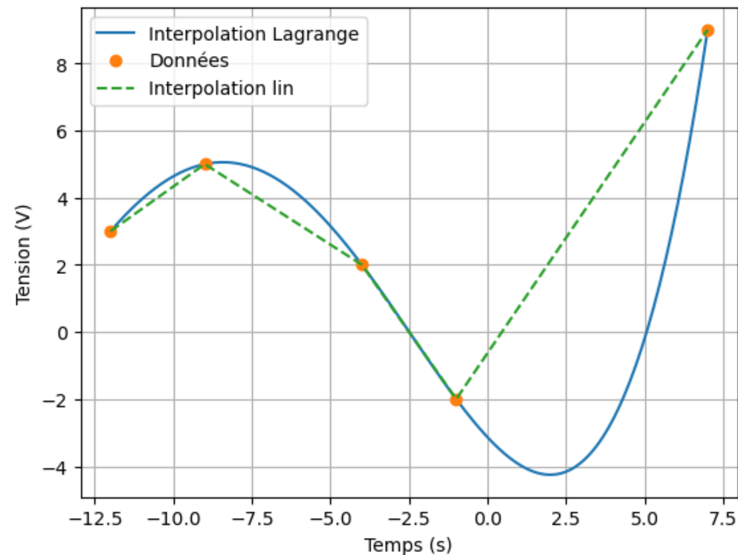
Correction 13 : $f(x) = Y_0 P_0(x) + Y_1 P_1(x) + ... + Y_{n-2} P_{n-2}(x) + Y_{n-1} P_{n-1}(x) = \sum_{i=0}^{n-1} Y_i P_i(x)$

Question 14 : Compléter le code de la fonction `interpolation_L(X:list, Y:list, t:float)->float`, qui permet de calculer une valeur de la fonction $f(x)$ adaptée à notre problème initial. On complétera aussi le code permettant de réaliser le graphe. Commenter.

[15]:
```
def interpolation_L(X, Y, x):
    """ Retourne la valeur à l'abscisse x du l'interpolation via polynômes de␣
↪Lagrange
    Nécessite les listes X des abscisses et Y des ordonnées"""

    # Nombre de points de données
    n = len(X)

    # Polynômes de Lagrange (évaluation en x)
    def P_Lagrange(x, i):
        """ Valeur en x du ième polynôme de Lagrange"""
        prod = 1
        for j in range(n) :
            if j != i :
                prod = prod * (x - X[j]) / (X[i] - X[j])
        return prod

    # Valeur de la fonction d'interpolation en t
    f = 0
    for i in range(n) :
        f = f + Y[i] * P_Lagrange(x, i)
```

```
    return f
```

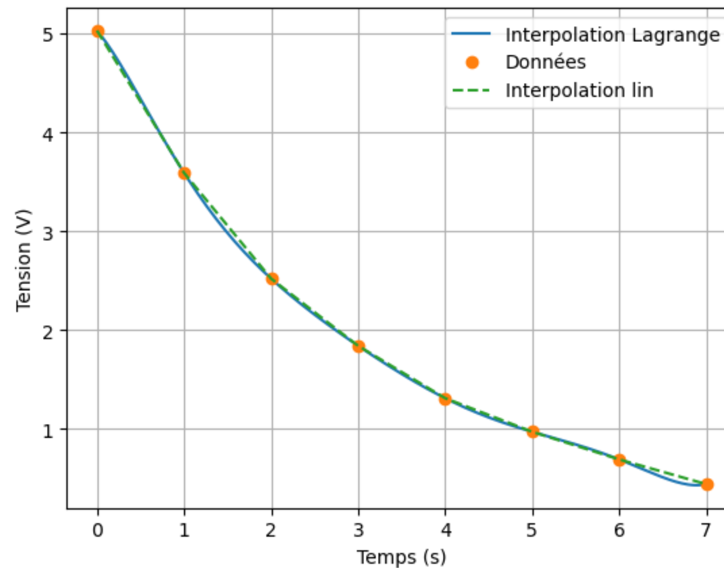
[16]:
```
T_abs = np.linspace(-12,7,200)
F = [interpolation_L(T, Z, t) for t in T_abs]

plt.figure()
plt.plot(T_abs, F, label="Interpolation Lagrange")
plt.plot(T,Z,'o',label='Données')
plt.plot(T,Z,'--', label='Interpolation lin')
plt.xlabel("Temps (s)")
plt.ylabel("Tension (V)")
plt.grid()
plt.legend()
plt.show()
```



Satisfaisant ! Mais prenons un autre exemple : l'acquisition de la tension aux bornes d'un condensateur dans un circuit RC en régime libre.

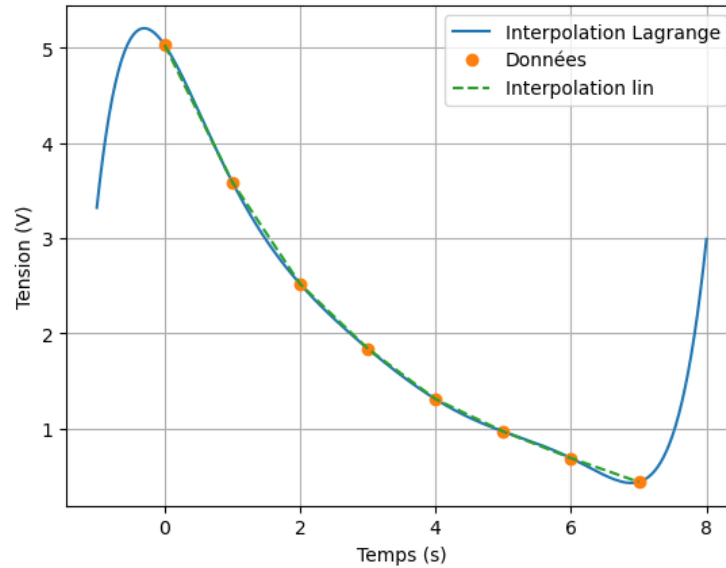
[17]:
```
T = [0, 1, 2, 3, 4, 5, 6, 7]
Uc = [5.03, 3.59, 2.52, 1.84, 1.31, 0.97, 0.69, 0.44]

T_abs = np.linspace(0,7,200)
F = [interpolation_L(T, Uc, t) for t in T_abs]

plt.figure()
plt.plot(T_abs, F, label="Interpolation Lagrange")
plt.plot(T,Uc,'o',label='Données')
plt.plot(T,Uc,'--', label='Interpolation lin')
plt.xlabel("Temps (s)")
plt.ylabel("Tension (V)")
plt.grid()
```

```
plt.legend()
plt.show()
```



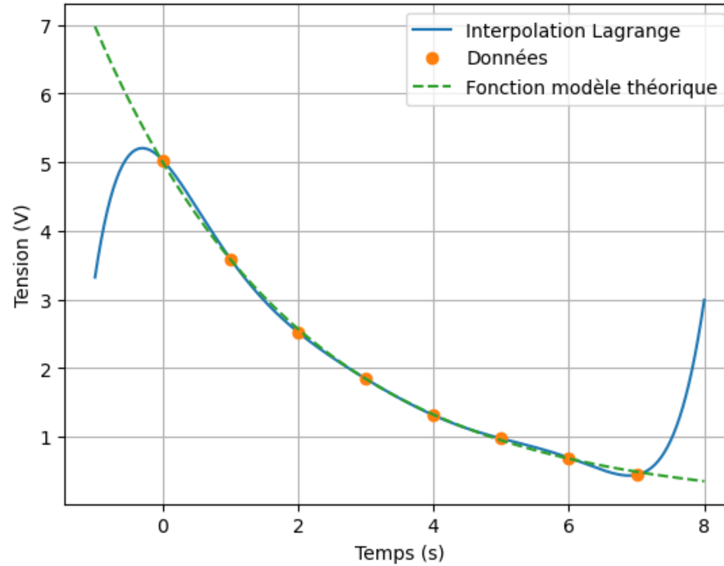
Pas très probant ici… Et c'est encore pire lorsqu'on veut interpoler en dehors de l'intervalle initial :

[18]:
```
T_abs = np.linspace(-1,8,200)
F = [interpolation_L(T, Uc, t) for t in T_abs]

plt.figure()
plt.plot(T_abs, F, label="Interpolation Lagrange")
plt.plot(T,Uc,'o',label='Données')
plt.plot(T,Uc,'--', label='Interpolation lin')
plt.xlabel("Temps (s)")
plt.ylabel("Tension (V)")
plt.grid()
plt.legend()
plt.show()
```



Dans le cas précis de l'exemple étudié ici (étude de la décharge d'un condensateur dans un circuit RC), on connaît théoriquement la fonction théorique selon laquelle les données doivent être disposées. Il est alors bien sûr préférable de modéliser les données à l'aide de la fonction théorique, et d'utiliser cette fonction "modèle" pour interpoler !

[19]:
```
Uc_th = [5 * math.exp(-t/3) for t in T_abs]

plt.figure()
plt.plot(T_abs, F, label="Interpolation Lagrange")
plt.plot(T,Uc,'o',label='Données')
plt.plot(T_abs,Uc_th,'--', label='Fonction modèle théorique')
plt.xlabel("Temps (s)")
plt.ylabel("Tension (V)")
plt.grid()
plt.legend()
plt.show()
```

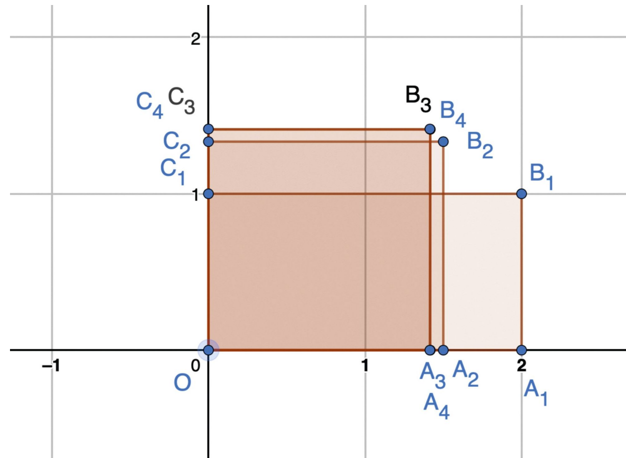


3 Calculs de racine carrées : méthode de Héron (facultative)

3.1 De Héron à aujourd'hui : un peu d'histoire et de la géométrie...

Le calcul numérique de la racine carré d'un nombre est aussi un calcul loin d'être évident : pour trouver $\sqrt{x}$, on ne peut se permettre de rechercher *bêtement* le nombre qui, multiplié par lui-même, donne x. Il existe de nombreuses méthodes pour obtenir la valeur approchée de $\sqrt{x}$ de façon approchée. Nous allons étudier la méthode de Héron, découverte par Héron d'Alexandrie, au 1er siècle après J.C. (voir https://fr.wikipedia.org/wiki/M%C3%A9thode_de_H%C3%A9ron).

Cette méthode repose sur le constat géométrique simple suivant : pour connaître la racine carré du nombre x, il faut déterminer le côté du carré d'aire x. Partons alors d'un rectangle de côtés 1 et x : ainsi l'aire de ce rectangle est bien x. On déforme alors progressivement ce rectangle pour le transformer en carré, tout en conservant sa surface ! Il faut donc diminuer sa longueur, et augmenter sa largeur. Héron propose l'algorithme suivant : à chaque étape, on diminue la longeur, en prenant pour nouvelle longeur la valeur moyenne de l'ancienne largeur et de l'ancienne longueur. On augmente alors la largeur, en prenant pour nouvelle valeur la *bonne* valeur permettant de conserver une aire de x. Ainsi la longueur et la largeur vont tendre vers la même valeur, le côté du carré d'aire x : il s'agit de $\sqrt{x}$.



Ci-dessus, les quatres premières itérations de l'algorithme, pour déterminer $\sqrt{2}$. On remarque que la convergence est rapide, le 4ème carré est déjà indiscernable du 3ème.

On peut montrer que cet algorithme est en réalité un cas particulier d'une méthode plus générale, la méthode de Newton. Il existe toujours aujourd'hui de la recherche afin d'optimiser les calculs numériques de ce type : on a, par exemple, recemment amélioré l'algorithme permettant à une machine de calculer le produit de 2 très grands nombres. De façon plus ludique, le jeu vidéo *Quake III*, sorti en 1999, est célèbre pour avoir, peut-être le premier, utilisé un nouvel algorithme pour le calcul de $\frac{1}{\sqrt{X}}$: l'algorithme de Racine Carré Inverse Rapide (voir https://fr.wikipedia.org/wiki/Racine_carr%C3%A9e_inverse_rapide). Cette fonction est en effet très utile, un moteur de rendu 3d calcul a besoin de déterminer fréquemment des vecteurs unitaires pour *pointer* une direction, et donc de diviser par une racine carrée.

3.2 Réalisation de la fonction python

Question 15 : Écrire le code permettant d'encadrer la valeur de $\sqrt{x}$, grâce à la méthode de Héron. On utilisera un nombre, la précision p, de telle sorte que l'algorithme s'arrêtera lorsque la longueur de l'intervalle est inférieure à cette précision. Le code retournera l'intervalle sous forme d'un tuple. Tester le code, et observer la convergence en modifiant la précision p.

Contruisons d'abord les deux suites théoriques adjacentes pour la largeur (U_n) et la longueur du rectangle (V_n). Mettons qu'on cherche à déterminer $\sqrt{x}$ avec $x > 1$, ainsi le côté du rectangle initialement de valeur x est sa longueur. Les suites se construisent alors de la façon suivante :

- Initialement, $U_0 = 1$ et $V_0 = x$
- Partons de la iième itération : à l'itération suivante $(i + 1)$, la longueur est diminuée pour prendre la valeur moyenne des précédentes valeurs de la longeur et de la largeur : $V_{i+1} = \frac{V_i+U_i}{2}$
- Concernant la nouvelle valeur de la largeur, on impose que l'aire du rectangle soit de x, ainsi $U_{i+1}V_{i+1} = x$ soit $U_{i+1} = \frac{x}{V_{i+1}}$

Pour résumer :

$$\begin{cases} U_0 & = 1 \\ V_0 & = x \\ V_{i+1} & = \frac{V_i+U_i}{2} \forall i \geq 1 \\ U_{i+1} & = \frac{x}{V_{i+1}} = \frac{2x}{V_i+U_i} \forall i \geq 1 \end{cases}$$

[20]:
```
def Heron(x,p):
    """ Retourne sqrt la racine carré du nombre x, sqrt_min sqrt_max un␣
 ↪encadrement, de longeur inférieure à la précision p """

    assert x > 0 # Ne pas essayer si x est négatif, cela n'a pas de sens␣
 ↪mathématique !

    u, v = 1, x

    while abs(u-v) > p :
        v = (u+v)/2
        u = x/v

    sqrt = (u+v)/2
    sqrt_min = min(u,v)
    sqrt_max = max (u,v)

    return sqrt, sqrt_min, sqrt_max
```

[21]:
```
print(Heron(4,0.1))
print(Heron(4,0.01))
print(Heron(4,0.001))
```

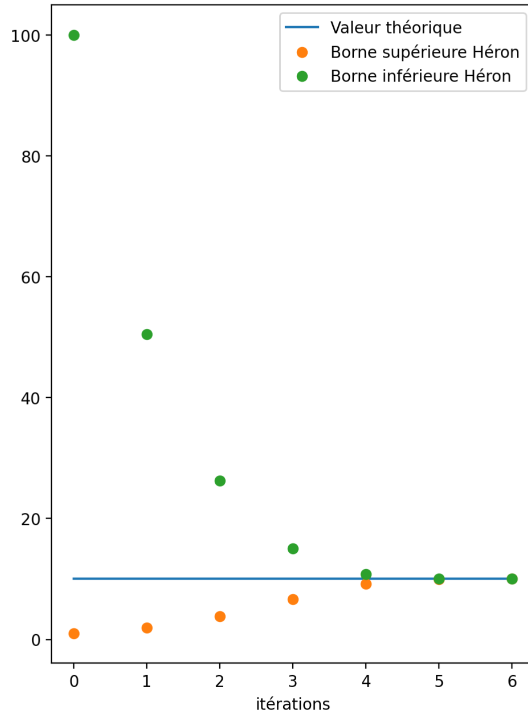
```
(2.000609756097561, 1.9512195121951221, 2.05)
(2.0000000929222947, 1.9993904297470284, 2.000609756097561)
(2.000000000000002, 1.9999999070777095, 2.0000000929222947)
```

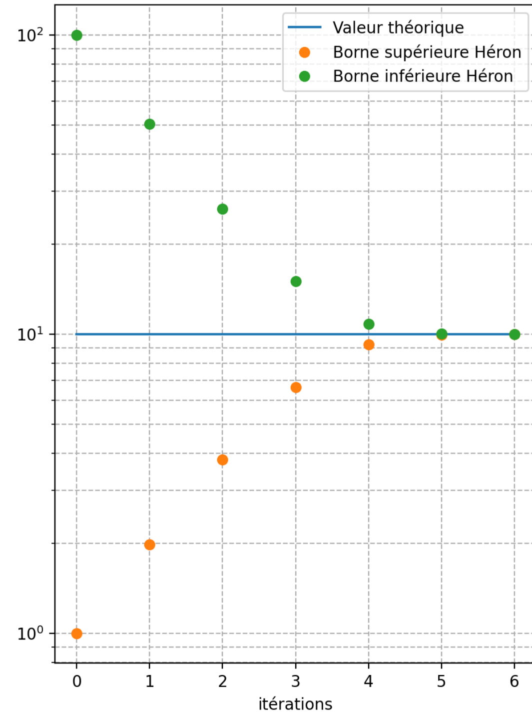
Regardons plus précisément la convergence de l'algorithme. Pour cela, modifions la fonction pour qu'elle retourne la liste des valeurs minimales et maximales :

[22]:
```
def Heron_conv(x,p):

    assert x > 0 # Ne pas essayer si x est négatif, cela n'a pas de sens␣
 ↪mathématique !

    u, v = min(1,x), max(1,x)
    i = 0

    SQRT_min = [u]
    SQRT_max = [v]
    cpt = 0
```

```python
    while v - u > p :
        v = (u+v)/2
        u = x/v
        cpt += 1
        SQRT_min.append(u)
        SQRT_max.append(v)

    return cpt, SQRT_min, SQRT_max
```

[23]:
```python
ratio =  1.5
plt.figure(figsize=(8*ratio,5*ratio),dpi = 200)

cpt, SQRT_min, SQRT_max = Heron_conv(100,0.01)
It = [i for i in range(cpt+1)]

plt.subplot(121)
# Valeur théorique
plt.plot([0,cpt],[10,10],label='Valeur théorique')
# Héron : encadrement
plt.plot(It,SQRT_min,'o', label='Borne supérieure Héron')
plt.plot(It,SQRT_max,'o', label='Borne inférieure Héron')
plt.legend()
plt.xlabel('itérations')

plt.subplot(122)
# Valeur théorique
plt.plot([0,cpt],[10,10],label='Valeur théorique')
# Héron : encadrement
plt.plot(It,SQRT_min,'o', label='Borne supérieure Héron')
plt.plot(It,SQRT_max,'o', label='Borne inférieure Héron')
plt.xlabel('itérations')
plt.yscale('log')
plt.grid(True,which="both", linestyle='--')
plt.legend()

plt.show()
```





C'est effectivement bien plus satisfaisant !