

NOM :

PRÉNOM :

Pour faire du calcul scientifique en Python, la bibliothèque `numpy` est incontournable. Elle permet de manipuler les flottants assez précisément.

```
import numpy as np
```

## 1 Représentation des flottants

La norme IEEE 754 fixe la représentation suivante pour les nombres à virgule flottante (les *flottants*) en 32 et 64 bits (les nombres entre parenthèses représentent des nombres de bits) :

$S(1)$	$E(8)$	$M(23)$
$S(1)$	$E(11)$	$M(52)$

Le bit de signe  $S$  vaut 0 si le flottant est positif, 1 sinon. La valeur absolue d'un flottant s'obtient par la formule (le (2) en indice signifie que la base de numération est 2) :

$$1.M_{(2)} \times 2^{E-D}$$

où la constante  $D$  vaut 127 dans le cas des 32 bits et 1023 dans le cas des 64 bits. Remarquer qu'un bit implicite se rajoute aux bits explicites de la mantisse  $M$ . On dispose donc en réalité de 24 bits pour la mantisse, en simple précision, et de 53 bits en double précision.

La norme IEEE 754 prévoit aussi des codages pour des nombres flottants spéciaux (dits *anormaux*). Les plus importants sont plus ou moins l'infini et `nan` (acronyme pour *not a number*). Les deux infinis apparaissent naturellement lorsqu'un flottant atteint une valeur qui dépasse, en valeur absolue, la plus grande valeur représentable. Le flottant `nan` apparaît lorsqu'on calcule la racine carrée d'un nombre négatif, par exemple. Les opérations arithmétiques sont également définies pour ces flottants anormaux.

**Question 1.** En faisant une recherche sur `numpy` et `dtype` dans la documentation officielle [numpy.org/doc/stable/reference](https://numpy.org/doc/stable/reference) indiquer comment on désigne dans `numpy` les types *flottant 64 bits* et *flottant 32 bits*. Grâce à cette information et en utilisant `numpy` et `info`, déterminer les valeurs maximales que peuvent prendre les flottants normaux dans ces deux types. Affecter à une variable  $a$  le flottant 32 bits 3. Vérifier en consultant l'attribut `dtype` de  $a$ . Produire les flottants `inf` et `nan` par calcul.

Réponse.

## 2 Les erreurs d'arrondi

### 2.1 La soustraction

L'exemple suivant montre que l'ordre des calculs peut influencer sur la précision du résultat et qu'il faut se méfier de la soustraction de deux flottants de valeurs très proches.

**Question 2.** Exécuter les instructions suivantes. Quel est le résultat le plus précis ? Expliquer.

```
a = np.float32(1e7)
b = np.float32(25.1234)
c = np.float32(10000025)
a + b - c
a - c + b
```

Réponse.

## 2.2 Le choix de la bonne formule

L'exemple suivant montre qu'avec des flottants, le choix de la bonne formule n'est pas toujours simple. Voici l'exemple de la résolution de l'équation polynomiale de degré deux.

On triche ! On part de deux flottants  $x_1$  et  $x_2$ . On en déduit les coefficients d'un polynôme de degré deux admettant  $x_1$  et  $x_2$  pour racines. Le but du jeu consiste à retrouver les racines.

$$P = (x - x_1)(x - x_2) = x^2 + \underbrace{(-x_1 - x_2)}_b x + \underbrace{x_1 x_2}_c.$$

```
x1 = np.float64(1.23456e8)
x2 = np.float64(1.3579177e7)
a = np.float64(1)
b = - x1 - x2
c = x1 * x2
```

**Question 3.** Résoudre l'équation  $ax^2 + bx + c$  en appliquant la méthode classique. Le résultat est-il précis ? Recommencer en remplaçant l'exposant 7 de  $x_2$  par  $-10$ . Qu'observe-t-on ? Il existe une formule alternative pour calculer  $x_2$  à partir de  $c$  et de  $x_1$ . Laquelle ? Fonctionne-t-elle mieux ? Expliquer ce qui se passe en consultant [https://en.wikipedia.org/wiki/Loss\\_of\\_significance#Instability\\_of\\_the\\_quadratic\\_equation](https://en.wikipedia.org/wiki/Loss_of_significance#Instability_of_the_quadratic_equation)

Réponse.

## 2.3 L'effet papillon

On peut montrer que la suite suivante

$$\begin{aligned} x_0 &= a + b, \\ x_1 &= 2a + \frac{b}{2}, \\ x_{n+2} &= \frac{5}{2}x_{n+1} - x_n, \end{aligned} \tag{1}$$

admet pour solution générale

$$x_n = a2^n + \frac{b}{2^n}. \tag{2}$$

Par conséquent, si on choisit  $a = 0$  et  $b$  quelconque, la suite  $(x_n)$  doit tendre vers zéro.

**Question 4.** Prenons  $(a, b) = (0, 1)$ . Écrire une suite d'instructions Python qui calcule les 80 premiers termes de la suite  $(x_n)$  en utilisant la formule (1). En consultant la formule (2), indiquer si la suite semble tendre vers sa limite théorique.

Pour les calculs, utiliser l'instruction suivante qui affecte à `x` un tableau de 81 flottants 64 bits indicé de 0 à 80 inclus ; affecter  $x_n$  à `x[n]` pour  $0 \leq n < 81$  ; afficher `x[80]`.

```
x = np.empty (81, dtype=np.float64)
```

**Question 5.** Même question pour  $(a, b) = (0, \frac{1}{3})$ . Expliquer ce qui se passe.

**Question 6.** Reprendre les questions précédentes en n'utilisant que deux variables `x0` et `x1` au lieu du tableau `x`. À la fin du calcul, la variable `x1` doit contenir  $x_{80}$ .

Réponse.

### 3 Conclusion

Le terme *flottant* est une abréviation pour *nombre à [ ] flottante*. La représentation des flottants est fixée par la norme [ ]. Elle est constituée d'un bit de signe, d'un exposant et d'une [ ], qui correspond (en gros) à la suite des "décimales" en base deux du flottant. Par exemple, sur un flottant 64 bits, elle comporte [ ] bits, plus un bit "gratuit". En plus des flottants normaux, il existe des flottants [ ]. Par exemple, le calcul  $1/0$  produit le flottant [ ] ; le calcul  $\sqrt{-1}$  produit le flottant [ ] (un sigle qui signifie [ ]). Le paquetage [ ] permet de préciser le type de flottant à utiliser. Par exemple, le type *flottant 64 bits* se note [ ]. Les calculs avec des flottants sont sujets aux erreurs [ ]. En particulier, il faut se méfier de la [ ] de deux flottants de valeurs très [ ], des formules mathématiquement [ ] ne le sont pas toujours avec des flottants et, dans des cas extrêmes, une toute petite erreur d'arrondi sur les données initiales peut avoir des conséquences arbitrairement grandes à la fin d'un long calcul. C'est ce qu'on appelle l'effet [ ].

## 1 Graphiques avec matplotlib

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

Le principe est le suivant : on construit un graphique initialement vide en ajoutant soit des points isolés (fonction `scatter` du sous-paquetage `pyplot` de `matplotlib`, nommée `plt` lors de l'import) soit des courbes (fonction `plot`). Le graphique lui-même n'est pas affiché tant que la fonction `show` n'est pas appelée. Après un appel à `show`, tous les objets de l'ancien graphique sont supprimés et on revient à un graphique vide.

```
# On affecte à abscisses et ordonnees les coordonnées de 4 points
abscisses = np.array([2,7,3,1])
ordonnees = np.array([7,45,12,2])
```

On "affiche" les 4 points en bleu avec une légende (paramètre `label`). Pour l'instant, rien n'est visualisé

```
plt.scatter (abscisses, ordonnees, color='blue', label='points expérimentaux')
```

On va maintenant superposer le graphe d'une fonction  $f$  pour  $x \in [0, 8]$ . Pour cela, on passe en argument à `plot` deux tableaux nommés `xplot` et `yplot`. Le tableau `xplot` contient 50 abscisses équidistantes dans l'intervalle  $[0, 8]$ . Le tableau `yplot` contient les valeurs de  $f(x)$  pour chaque  $x \in \text{xplot}$ . La fonction `plot` relie chaque point au point suivant par un segment de droite de couleur orange, donnant l'illusion d'une courbe lisse. On prévoit une légende. On peut rendre le graphique plus ou moins fin en augmentant ou en diminuant la valeur 50.

```
# Définition de f
def f(x) :
    return (.5*x+3.1)*x-1.05
```

```
# Les 50 points équidistants entre 0 et 8 s'obtiennent avec la fonction linspace de numpy
xplot = np.linspace (0, 8, 50)
yplot = np.array ([f(x) for x in xplot])
plt.plot (xplot, yplot, color='orange', label='graphe de f')
```

Il ne reste plus qu'à visualiser le graphique. L'appel à `legend` est nécessaire pour que la légende s'affiche. Après l'appel à `show`, l'ancien graphique est supprimé.

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

Des commandes supplémentaires peuvent parfois être utiles pour améliorer la lisibilité : `xlabel` et `ylabel` pour mettre une légende sur les axes, `title` pour un titre, `grid` pour mieux lire les coordonnées des points.

**Question 1.** Exécuter les commandes ci-dessus.

## 2 Évaluation d'un polynôme

La façon la plus simple de représenter un polynôme

$$A(x) = a_0 + a_1 x + \dots + a_n x^n \tag{1}$$

consiste à stocker ses coefficients dans un tableau  $A$  indicé de 0 à  $n$  (inclus).

**Question 2.** Affecter le polynôme  $A(x) = -40 + 82x - 35x^2 - 19x^3 + 12x^4$  à une variable  $A$ , sous la forme d'un tableau `numpy`.

**Question 3.** Tracer le graphe de  $A(x)$  dans l'intervalle  $[-2.5, 2]$ . Utiliser la fonction `polyval`<sup>1</sup> du paquetage `numpy.polynomial.polynomial` pour évaluer  $A$ . Utiliser la fonction `axhline` de `pyplot` pour ajouter l'axe horizontal. Combien ce polynôme admet-il de racines réelles dans l'intervalle considéré (répondre à partir du graphique) ?

## 2.1 Évaluation naïve

**Question 4.** Écrire une fonction Python `eval_naive`, paramétrée par un flottant  $x$ , un polynôme  $A$  et qui retourne  $A(x)$  en appliquant la formule (1). Utiliser une boucle `for` pour le calcul de la somme et l'expression `x**k` pour calculer  $x^k$ .

Vérifier votre fonction en l'utilisant pour tracer le graphique de  $A(x)$ .

En supposant<sup>2</sup> que l'expression `x**k` effectue  $k-1$  multiplications, déterminer en fonction du degré  $n$  de  $A$ , combien d'additions et de multiplications sont effectuées par la fonction.

## 2.2 Tester votre fonction

Dans cette section, on suppose que la fonction `eval_naive` est programmée dans un fichier `tp2.py`. Le code Python de la figure 1 (page 5) doit être placé dans un fichier `test_tp2.py`. Il permet d'exécuter trois fonctions qui sont autant de *tests unitaires* de la fonction `eval_naive`. À l'exécution, il donne :

```
Exécution de test_eval_naive_1 : succès
Exécution de test_eval_naive_2 : succès
Exécution de test_eval_naive_3 : succès
Nombre de tests   : 3
Nombre de succès  : 3
Nombre d'échecs   : 0
```

**Question 5.** Qu'est-ce que les trois tests unitaires testent ?

Réponse.

## 2.3 Le schéma de Horner

Il fournit un meilleur algorithme d'évaluation de polynôme. Il est d'ailleurs utilisé par `polyval`. Il revient à calculer :

```
y = a[n]
y = y*x + a[n-1]
y = y*x + a[n-2]
...
y = y*x + a[0]
```

**Question 6.** Programmer une fonction Python `eval_Horner`, ayant même paramétrage que `eval_naive` et qui retourne  $A(x)$ , évalué par le schéma de Horner. Vérifier votre fonction. Déterminer, en fonction du degré  $n$  de  $A$ , combien d'additions et de multiplications sont effectuées par la fonction. Conclusion ?

---

1. Attention : **ne pas utiliser** la fonction obsolète `numpy.polyval` qui applique une convention incompatible avec le TP.

2. En fait, c'est faux : l'expression `x**k` utilise un algorithme plus général que l'exponentiation entière.

Réponse.

**Question 7.** Compléter le fichier `test_tp2.py` en y ajoutant des tests unitaires pour `eval_Horner`.

**Évaluation de la dérivée.** Dans un souci de lisibilité, notons  $y_k(x)$  les valeurs successives prises par la variable `y` dans le schéma de Horner. On obtient

$$\begin{aligned} y_n(x) &= a_n, \\ y_{n-1}(x) &= y_n(x)x + a_{n-1}, \\ y_{n-2}(x) &= y_{n-1}(x)x + a_{n-2}, \\ &\vdots \\ y_0(x) &= y_1(x)x + a_0. \end{aligned} \tag{2}$$

Posons  $d_k(x) = y'_k(x)$ . En dérivant les expressions de la suite (2), on s'aperçoit que  $d_n(x), d_{n-1}(x), \dots, d_0(x)$  peuvent se calculer en même temps que les  $y_k(x)$ .

**Question 8.** Préciser la remarque précédente et programmer une fonction Python `eval_derivee`, ayant même paramétrage que `eval_naive` et qui retourne  $A'(x)$ . Compléter le fichier `test_tp2.py` pour qu'il teste également `eval_derivee`.

**Question 9.** Le raisonnement précédent permettrait-il de calculer la dérivée seconde ?

### 3 Polynôme donné par ses racines

Un sous-paquetage de `numpy` contient une fonction nommée `polyfromroots` qui permet de calculer les coefficients d'un polynôme dans la base des monômes, à partir de ses racines.

**Question 10.** Utiliser cette fonction pour vérifier que le polynôme  $A(x)$  admet  $1, -2, \frac{5}{4}, \frac{4}{3}$  pour racines.

**Le polynôme de Wilkinson.** Il est défini par

$$W(x) = (x-1)(x-2)(x-3)\cdots(x-20).$$

**Question 11.** Affecter à une variable `W` les coefficients de  $W(x)$  dans la base des monômes. Vérifier avec `polyval` qu'il admet bien les entiers de 1 à 20 pour racines. Qu'observe-t-on ? Expliquer le phénomène observé en consultant [https://en.wikipedia.org/wiki/Wilkinson%27s\\_polynomial](https://en.wikipedia.org/wiki/Wilkinson%27s_polynomial)

### 4 Conclusion

La bibliothèque `numpy` permet de manipuler des tableaux à une dimension (vecteurs, polynômes) ou à deux dimensions (matrices) en précisant le type de leurs éléments. Elle fournit des fonctionnalités pour manipuler des polynômes représentés par le tableau de leurs coefficients (dans la base des ). Le schéma de  est un important algorithme pour évaluer les polynômes. Il s'adapte facilement pour évaluer leur . Il est disponible dans un sous-paquetage `numpy` sous le nom de . En pratique, les polynômes ne sont pas toujours présentés par leurs coefficients, dans la base des monômes.

Ils peuvent par exemple être définis par leurs racines. Le réflexe naturel qui consiste à transformer la représentation initiale pour se ramener à la base des monômes (fonction `polyfromroots`) est , parce que les calculs en virgule flottante engendrent des erreurs d'arrondis et que les racines sont très  aux perturbations, même minimales, sur les coefficients (exemple du polynôme de ). Enfin, à chaque fois qu'on programme une fonction, il est très utile de programmer des tests  qui permettent de vérifier systématiquement la qualité du code écrit.

```

#!/bin/python3
# Fichier test_tp2.py

import numpy as np

# Cette commande importe la fonction à tester (à adapter !)
from tp2 import eval_naive

# On écrit une fonction Python par test unitaire
# Le contenu du test doit bien sûr être adapté au cas par cas

def test_eval_naive_1():
    A = np.array ([], dtype=np.float64)
    assert (eval_naive (17, A) == 0), "0 attendu"

def test_eval_naive_2():
    A = np.array ([-1, 0, 1], dtype=np.float64)
    assert (eval_naive (-1, A) == 0), "0 attendu"

def test_eval_naive_3():
    A = np.array ([-1, 0, 1], dtype=np.float64)
    assert (eval_naive (1, A) == 0), "0 attendu"

# La liste L des fonctions définies ci-dessus (à adapter !)
L = [test_eval_naive_1, test_eval_naive_2, test_eval_naive_3]

# A partir d'ici, il n'y a plus rien à changer.
# La boucle suivante exécute chaque test et compte les succès.
nb_succès = 0
nb_échecs = 0
for test in L :
    print ('Exécution de', test.__name__, end='')
    try:
        test ()
        nb_succès += 1
        print (' : succès')
    except:
        nb_échecs += 1
        print (' : échec')

# Synthèse des résultats
print ('Nombre de tests :', len(L))
print ('Nombre de succès :', nb_succès)
print ("Nombre d'échecs :", nb_échecs)

```

FIGURE 1 – Le fichier `test_tp2.py` contient des tests unitaires pour la fonction `eval_naive`, supposée appartenir au fichier `tp2.py`.



Par deux points distincts, il passe une unique droite. Cette observation se généralise.

**Théorème 1** *Étant donnés  $n + 1$  points*

$$\begin{pmatrix} x_0 \\ y_0 \end{pmatrix}, \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}, \dots, \begin{pmatrix} x_n \\ y_n \end{pmatrix},$$

*d'abscisses distinctes deux-à-deux, il existe un unique polynôme  $P(z)$  de degré au plus  $n$ , dont le graphe passe par chacun des points. Ce polynôme est appelé polynôme d'interpolation.*

Voici un exemple ( $n = 3$ ) :

$i$	0	1	2	3
$x_i$	1	2	3	5
$y_i$	1	4	2	5

Le polynôme d'interpolation est

$$P(z) = -\frac{25}{2} + \frac{247}{12}z - 8z^2 + \frac{11}{12}z^3.$$

## 1 La matrice de Vandermonde

La façon la plus simple de se convaincre de l'existence et de l'unicité du polynôme d'interpolation consiste à lui attribuer des coefficients inconnus  $a_0, a_1, \dots, a_n$

$$P(z) = a_0 + a_1 z + \dots + a_n z^n$$

et à écrire les conditions que doivent vérifier les  $a_i$  pour que le polynôme interpole les  $n + 1$  points :

$$\begin{aligned} y_0 &= a_0 + a_1 x_0 + \dots + a_n x_0^n, \\ y_1 &= a_0 + a_1 x_1 + \dots + a_n x_1^n, \\ &\vdots \\ y_n &= a_0 + a_1 x_n + \dots + a_n x_n^n. \end{aligned}$$

Sous forme matricielle, ce système s'écrit

$$\underbrace{\begin{pmatrix} 1 & x_0 & \dots & x_0^n \\ 1 & x_1 & \dots & x_1^n \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & \dots & x_n^n \end{pmatrix}}_A \underbrace{\begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix}}_v = \underbrace{\begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix}}_b.$$

La matrice  $A$  est connue sous le nom de *matrice de Vandermonde*. Elle est carrée et on peut montrer qu'elle est inversible si les  $x_i$  sont distincts deux-à-deux.

**Question 1.** Affecter à deux variables  $x$  et  $y$  des tableaux `numpy` contenant les abscisses et les ordonnées des points de l'exemple sous la forme de flottants 64 bits. Affecter à une variable  $A$  la matrice de Vandermonde de l'exemple, en utilisant la fonction `vander` de `numpy`, en réglant un paramètre optionnel pour que les colonnes apparaissent dans le même ordre que sur l'énoncé. Résoudre le système d'équations linéaires avec la fonction `solve` du sous-paquetage `linalg` de `numpy`. Vérifier le résultat.

**Question 2.** Écrire une fonction Python `Vandermonde` paramétrée par un tableau d'abscisses  $x$ , qui retourne la matrice de Vandermonde  $A$  définie par  $x$ . Utiliser la fonction `empty` de `numpy` pour définir  $A$ . Remplir  $A$  au moyen de deux boucles `for` imbriquées.

**Remarque.** L'approche par la matrice de Vandermonde cherche à calculer les coefficients du polynôme d'interpolation dans la base des monômes. C'est une mauvaise idée (le polynôme de Wilkinson est un polynôme d'interpolation). Les algorithmes qui suivent s'y prennent plus subtilement.

## 2 L'algorithme de Neville

Mathématiquement, l'algorithme de Neville<sup>1</sup> consiste à calculer un tableau  $P$  de polynômes. Pour tous  $0 \leq j \leq n$  et  $j \leq i \leq n$ , le polynôme  $P_{i,j}$  est défini comme le polynôme d'interpolation dont le graphe passe par les points d'indices  $i, i-1, \dots, i-j$ .

$i$	$x_i$	$j$			
		0	1	2	3
0	1	1			
1	2	4	$3z - 2$		
2	3	2	$-2z + 8$	$-\frac{5}{2}z^2 + \frac{21}{2}z - 7$	
3	5	5	$\frac{3}{2}z - \frac{5}{2}$	$\frac{7}{6}z^2 - \frac{47}{6}z + 15$	$\frac{11}{12}z^3 - 8z^2 + \frac{247}{12}z - \frac{25}{2}$

**Question 3.** Où se trouve le polynôme d'interpolation recherché ?

**Question 4.** Quels points  $P_{2,1}$  est-il censé interpoler ? Vérifier. Même question pour  $P_{2,2}$ .

**La formule.** Le tableau  $P$  peut se construire colonne par colonne grâce aux formules suivantes :

$$\begin{aligned} P_{i,0} &= y_i & (0 \leq i \leq n), \\ P_{i,j} &= \frac{(x_i - z) P_{i-1,j-1} + (z - x_{i-j}) P_{i,j-1}}{x_i - x_{i-j}} & (1 \leq j \leq n, \quad j \leq i \leq n). \end{aligned} \quad (1)$$

D'un point de vue informatique, il ne serait pas pratique du tout de construire un tableau de polynômes. L'idée consiste à voir les formules (1) comme un algorithme permettant d'évaluer le polynôme d'interpolation  $P_{n,n}$  pour un flottant  $z$  donné (mais sans expliciter les coefficients de  $P_{n,n}$  dans la base des monômes!).

**Question 5.** Écrire une fonction Python `eval_Neville`, paramétrée par deux tableaux  $x$  et  $y$  contenant les abscisses et les ordonnées des points à interpoler, un flottant  $z$  et qui retourne le flottant  $P_{n,n}(z)$ . Vérifier en produisant un graphique comportant les points et le graphe du polynôme d'interpolation.

**Question 6.** Écrire un fichier `test_tp3.py` similaire à celui du TP 2, contenant quelques tests unitaires pour `eval_Neville`.

### 2.1 Justification

On rappelle les points interpolés par quelques polynômes du tableau  $P$ . On suppose, pour simplifier la preuve, que chaque  $P_{i,j}$  est de degré  $j$ .

	indices des points interpolés					deg
$P_{i,j}$	$i$	$i-1$	$\dots$	$i-j+1$	$i-j$	$j$
$P_{i,j-1}$	$i$	$i-1$	$\dots$	$i-j+1$		$j-1$
$P_{i-1,j-1}$		$i-1$	$\dots$	$i-j+1$	$i-j$	$j-1$

1. Eric Harold Neville (1889-1961).

On remarque que  $P_{i,j} - P_{i,j-1}$  s'annule sur les abscisses des  $j$  points  $i, \dots, i-j+1$ . Il est donc de degré  $j$  et a la forme suivante. Le coefficient  $c$  est égal au coefficient dominant de  $P_{i,j}$  puisque  $\deg P_{i,j} > \deg P_{i,j-1}$ .

$$P_{i,j} - P_{i,j-1} = c(z - x_i) \cdots (z - x_{i-j+1}).$$

De même,  $P_{i,j} - P_{i-1,j-1}$  s'annule sur les abscisses des  $j$  points  $i-1, \dots, i-j$ . Il est donc de degré  $j$  et a la forme suivante (même coefficient  $c$ ) :

$$P_{i,j} - P_{i-1,j-1} = c(z - x_{i-1}) \cdots (z - x_{i-j}).$$

Par conséquent,

$$\frac{P_{i,j} - P_{i,j-1}}{P_{i,j} - P_{i-1,j-1}} = \frac{z - x_i}{z - x_{i-j}}.$$

La formule (1) s'obtient en tirant  $P_{i,j}$  de l'égalité ci-dessus.

## 2.2 Les différences divisées de Newton

Un raisonnement du même genre permet de calculer les coefficients dominants (notons-les  $c_{i,j}$ ) des polynômes  $P_{i,j}$  plutôt que les polynômes eux-mêmes. On aboutit alors à

$$\begin{aligned} c_{i,0} &= y_i & (0 \leq i \leq n), \\ c_{i,j} &= \frac{c_{i,j-1} - c_{i-1,j-1}}{x_i - x_{i-j}} & (1 \leq j \leq n, \quad j \leq i \leq n). \end{aligned} \quad (2)$$

Sur l'exemple précédent on trouve

$i$	$x_i$	$j$			
		0	1	2	3
0	1	1			
1	2	4	3		
2	3	2	-2	$-\frac{5}{2}$	
3	5	5	$\frac{3}{2}$	$\frac{7}{6}$	$\frac{11}{12}$

Il ne reste plus qu'à reconstruire le polynôme d'interpolation  $Q_n = P_{n,n}$  à partir du tableau des différences divisées. La formule recherchée est :

$$Q_n = c_{0,0} + c_{1,1}(z - x_0) + c_{2,2}(z - x_0)(z - x_1) + \cdots + c_{n,n}(z - x_0)(z - x_1) \cdots (z - x_{n-1}). \quad (3)$$

**Question 7.** Programmer en Python une fonction `eval_Newton_interp` paramétrée par le tableau des abscisses  $x$ , le tableau des différences divisées  $c$  et un flottant  $z$ , qui retourne la valeur de (3) en appliquant une variante du schéma de Horner.

**Remarque.** L'approche de Newton<sup>2</sup> présente un avantage vis-à-vis de celle de Neville : le tableau des différences divisées ne dépend pas de  $z$  et peut donc se calculer une seule fois.

**Question 8.** Proposer une implantation Python de l'algorithme de Newton, appelée `eval_Newton`, paramétrée par un flottant  $z$  et d'autres paramètres éventuels, qui retourne  $P_{n,n}(z)$  tout en tirant parti de la remarque précédente.

**Question 9.** Compléter le fichier `test_tp3.py`.

---

2. Isaac Newton (1643-1727).

## 2.3 Justification

On veut se convaincre que la formule (3) est correcte. La stratégie est la suivante :

1. on part de la formule (3) ; on suppose que les coefficients  $c_{i,i}$  sont indéterminés ( $0 \leq i \leq n$ ) ;
2. on montre qu'il existe des valeurs numériques pour les  $c_{i,i}$  telles que l'égalité (3) soit vraie, c'est-à-dire telles que le polynôme  $Q_n$  défini par la formule est bien le polynôme d'interpolation dont le graphe passe par tous les points ;
3. on montre enfin que les coefficients  $c_{i,i}$  ainsi définis sont égaux aux coefficients dominants des polynômes d'interpolation  $P_{i,i}$  de Neville ( $0 \leq i \leq n$ ).

Il n'y a rien à faire à l'étape 1. Passons à l'étape 2. Il s'agit de montrer qu'il existe des valeurs numériques pour les  $c_{i,i}$  telles que, si on évalue le membre droit de la formule (3) en  $x_i$  alors on obtient  $y_i$  ( $0 \leq i \leq n$ ). On remarque que des simplifications apparaissent dans le membre droit de la formule (3) lorsqu'on l'évalue en  $x_i$  : les termes qui dépendent de  $c_{i+1,i+1}, \dots, c_{n,n}$  disparaissent. Il s'agit donc de montrer que le système d'équations suivant admet une solution. Les inconnues sont les  $c_{i,i}$ . En raison de la nature triangulaire du système, l'existence d'une solution est évidente.

$$\begin{aligned} y_0 &= c_{0,0}, \\ y_1 &= c_{0,0} + c_{1,1}(x_1 - x_0), \\ &\vdots \\ y_n &= c_{0,0} + c_{1,1}(x_1 - x_0) + \dots + c_{n,n}(x_n - x_0)(x_n - x_1) \dots (x_n - x_{n-1}). \end{aligned} \tag{4}$$

Passons à l'étape 3. On suppose que les  $c_{i,i}$  vérifient le système (4). Les polynômes  $Q_i$  définis par la formule (3) sont donc les polynômes d'interpolation définis par les points d'indices 0 à  $i$ . Ils sont donc égaux aux polynômes  $P_{i,i}$  de Neville, en raison de l'unicité du polynôme d'interpolation. La structure du membre droit de la formule (3) montre que chaque polynôme  $Q_i$  n'a qu'un seul monôme de degré  $i$ , qui est égal à  $c_{i,i} z^i$ . Donc les  $c_{i,i}$  sont bien les coefficients dominants des polynômes  $P_{i,i}$  de Neville.

## 3 Conclusion

Étant donnés  $n + 1$  points d'abscisses  deux-à-deux, il existe un  polynôme de degré  dont le graphe passe par chacun des points. Ce polynôme est appelé polynôme . Les coefficients de ce polynôme dans la base des monômes peuvent s'obtenir en résolvant un système d'équations linéaires dont la matrice des coefficients a une forme très particulière. Il s'agit de la matrice de . Cette méthode est . Une bien meilleure méthode, dite des , a été inventée par  au début du dix-huitième siècle. Le principe de cette méthode s'explique bien si on part de l'algorithme de , qui a pourtant été inventé deux cents ans plus tard.

## 1 Un exemple concret

Le tableau ci-dessous [1, chapter 6, Figure 6.04, page 134] (reproduite en Figure 1) mesure l'évaporation de l'eau (en pouces) au Waite Institute, Adelaide, Australie. Chaque mesure est une moyenne, calculée à partir de relevés journaliers, sur une période de 23 ans. Les mesures montrent que l'évaporation décroît graduellement de janvier à juin-juillet, pour remonter d'août à décembre (les saisons sont inversées, dans l'hémisphère sud).

mois	$x_i$	1	2	3	4	5	6	7	8	9	10	11	12
évaporation (pouces)	$y_i$	8.6	7	6.4	4	2.8	1.8	1.8	2.3	3.2	4.7	6.2	7.9

**Question 1.** La donnée du mois de mars ( $x_i = 3$ ) peut sembler aberrante. Tracer<sup>1</sup> sur un même graphique les graphes des polynômes d'interpolation avec et sans la donnée du mois de mars, sur l'intervalle  $[.9, 12.1]$ . Le graphe est-il fort différent suivant que cette donnée est présente ou pas ? Les différences sont-elles plutôt localisées au voisinage du mois de mars ? Comparer avec la courbe de la Figure 1.

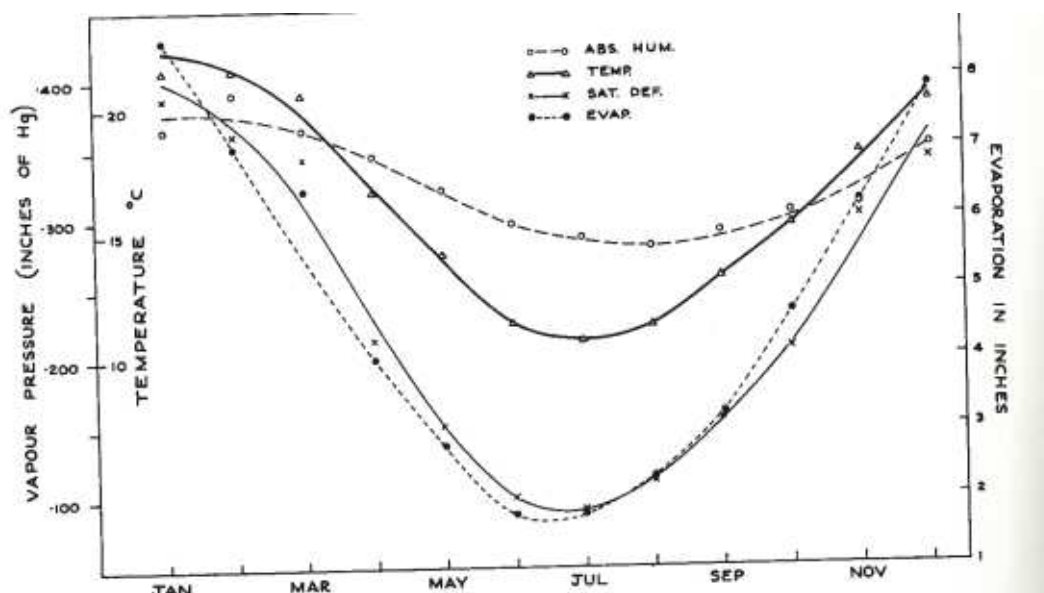


FIG. 6.04.—Showing the secular trends of atmospheric temperature and humidity at the Waite Institute, Adelaide. The curves are based on mean daily records for 23 years. Note that changes in absolute humidity are slight and gradual compared to the more abrupt changes in saturation deficiency and evaporation, which are closely related to temperature as well as to absolute humidity.

FIGURE 1 – Extrait de [1, chapter 6, Figure 6.04, page 134]. Les données qui nous intéressent sont les petits disques pleins, interpolés par la courbe en pointillés. Remarquer que la courbe ne passe pas par la donnée correspondant au mois de mars.

**Splines cubiques.** Dans beaucoup d'applications concrètes, on préfère utiliser des *splines cubiques* plutôt que de simples polynômes d'interpolation. Reprendre la question précédente en utilisant la fonction `CubicSpline` de la bibliothèque `scipy`. Conclusion ?

1. Utiliser de préférence une des fonctions écrites au TP précédent ; à défaut utiliser la fonction `krogh_interpolate` de la bibliothèque `scipy.interpolate`.

## 2 Splines cubiques

On suppose donnés  $n + 1$  points

$$\begin{pmatrix} x_0 \\ y_0 \end{pmatrix}, \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}, \dots, \begin{pmatrix} x_n \\ y_n \end{pmatrix}.$$

Les abscisses des points sont appelées des *nœuds* (*knots* en Anglais). On les suppose ordonnées par ordre croissant

$$x_0 < x_1 < \dots < x_n.$$

Les nœuds  $x_1, \dots, x_{n-1}$  sont appelés *nœuds intérieurs*. Les nœuds  $x_0$  et  $x_n$  sont appelés *extrémités*.

**Définition 1** Une spline cubique est une fonction  $s(x)$  deux fois dérivable sur l'intervalle  $[a, b]$ , qui se réduit à un polynôme de degré 3 sur chaque sous-intervalle  $[x_i, x_{i+1}]$  de  $[a, b]$ .

Concrètement, toute spline cubique est donc une fonction définie par morceaux de la forme suivante. Des conditions sur les coefficients font que les raccords entre les morceaux (au niveau des nœuds intérieurs) sont  $C^2$ .

$$s(z) = \begin{cases} s_0(z) = a_0 + b_0(z - x_0) + c_0(z - x_0)^2 + d_0(z - x_0)^3, & (z \leq x_1), \\ s_1(z) = a_1 + b_1(z - x_1) + c_1(z - x_1)^2 + d_1(z - x_1)^3, & (x_1 \leq z \leq x_2), \\ s_2(z) = a_2 + b_2(z - x_2) + c_2(z - x_2)^2 + d_2(z - x_2)^3, & (x_2 \leq z \leq x_3), \\ \vdots & \\ s_i(z) = a_i + b_i(z - x_i) + c_i(z - x_i)^2 + d_i(z - x_i)^3, & (x_i \leq z \leq x_{i+1}), \\ \vdots & \\ s_{n-1}(z) = a_{n-1} + b_{n-1}(z - x_{n-1}) + c_{n-1}(z - x_{n-1})^2 + d_{n-1}(z - x_{n-1})^3, & (x_{n-1} \leq z). \end{cases}$$

Le fait d'écrire chaque cubique  $s_i(x)$  sous la forme d'un développement limité en  $z = x_i$  simplifie considérablement les formules qui donnent les coefficients  $a_i, b_i, c_i$  et  $d_i$  à partir des points à interpoler. Notons

$$h_i = x_{i+1} - x_i, \quad (0 \leq i \leq n-1). \quad (1)$$

### 2.1 Splines interpolantes naturelles

Une spline est dite *interpolante* si son graphe passe par les  $n + 1$  points. Elle est dite *naturelle* si ses dérivées secondes sont nulles aux deux extrémités :  $c_0 = c_n = 0$ .

**Question 2.** Sachant que  $s_i(x_i)$  doit être égal à  $y_i$ , que vaut  $a_i$  ?

**Question 3.** Programmer une fonction Python  $h$ , paramétrée par un entier  $i$  et qui retourne  $h(i)$ . Le tableau  $x$  est une variable globale et n'a donc pas besoin d'être passé en paramètre.

**Calcul du vecteur  $c$ .** Le vecteur  $c$  s'obtient en résolvant un système d'équations linéaires pour les nœuds intérieurs (on obtiendra  $c_1, \dots, c_{n-1}$ ). Il ne restera plus qu'à rajouter deux coordonnées nulles (une au début et une à la fin du vecteur) pour exprimer les conditions de spline naturelle et obtenir le vecteur  $c$  complet.

On introduit la matrice  $A$  de dimension  $(n-1) \times (n-1)$  suivante (pour une justification, utiliser (4) plus l'hypothèse  $c_0 = c_n = 0$ ) :

$$A = \begin{pmatrix} 2\left(\frac{h_0}{3} + \frac{h_1}{3}\right) & \frac{h_1}{3} & 0 & \dots & 0 & 0 \\ \frac{h_1}{3} & 2\left(\frac{h_1}{3} + \frac{h_2}{3}\right) & \frac{h_2}{3} & \dots & 0 & 0 \\ 0 & \frac{h_2}{3} & 2\left(\frac{h_2}{3} + \frac{h_3}{3}\right) & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 2\left(\frac{h_{n-3}}{3} + \frac{h_{n-2}}{3}\right) & \frac{h_{n-2}}{3} \\ 0 & 0 & 0 & \dots & \frac{h_{n-2}}{3} & 2\left(\frac{h_{n-2}}{3} + \frac{h_{n-1}}{3}\right) \end{pmatrix}$$

ainsi que le vecteur  $v$  de dimension  $n - 1$  suivant :

$$v = \begin{pmatrix} \frac{1}{h_0} y_0 - \left(\frac{1}{h_0} + \frac{1}{h_1}\right) y_1 + \frac{1}{h_1} y_2 \\ \frac{1}{h_1} y_1 - \left(\frac{1}{h_1} + \frac{1}{h_2}\right) y_2 + \frac{1}{h_2} y_3 \\ \vdots \\ \frac{1}{h_{n-2}} y_{n-2} - \left(\frac{1}{h_{n-2}} + \frac{1}{h_{n-1}}\right) y_{n-1} + \frac{1}{h_{n-1}} y_n \end{pmatrix}.$$

**Question 4.** Écrire des instructions Python qui construisent la matrice  $A$ , le vecteur  $v$ , qui résolvent le système avec la fonction `solve` du paquetage `linalg` de `numpy` et qui affectent le résultat à  $c$ . Utiliser la fonction `zeros` de `numpy` pour initialiser  $A$ .

**Question 5.** Modifier le vecteur  $c$  pour lui ajouter deux coordonnées nulles (une au début et une à la fin) :

```
c = np.array([0, *c, 0], dtype=np.float64)
```

**Question 6.** Écrire des instructions Python qui calculent les vecteurs  $b$  et  $d$ , tous les deux de dimension  $n$ . Les formules sont les suivantes (pour  $0 \leq i < n$ ) :

$$d_i = \frac{c_{i+1} - c_i}{3 h_i}, \quad (2)$$

$$b_i = \frac{a_{i+1} - a_i}{h_i} - c_i h_i - d_i h_i^2. \quad (3)$$

**Question 7.** Écrire une fonction `eval_spline_naturelle`, paramétrée par un flottant  $z$ , qui retourne  $s(z)$ . Les tableaux  $x, y, a, b, c, d$  sont des variables globales et n'ont donc pas besoin d'être passés en paramètre.

La fonction doit commencer par déterminer l'indice  $i$  correspondant à  $z$  (utiliser éventuellement la fonction `searchsorted` de `numpy`).

Elle doit ensuite évaluer la cubique  $s_i(z)$  en appliquant un schéma de Horner.

Vérifier le résultat en traçant le graphe de  $s(x)$  entre  $x_0$  et  $x_n$  et en comparant le résultat avec le tracé de la fonction `CubicSpline` (attention à passer le paramètre optionnel `bc_type='natural'` à la fonction `CubicSpline`).

**Question 8.** Écrire un fichier `test_tp4.py` contenant des tests unitaires pour `eval_spline_naturelle`.

## 2.2 Justification

La continuité de la dérivée seconde de  $s$  aux nœuds intérieurs se traduit par le fait que  $s''_{i+1}(x_{i+1}) = 2 c_{i+1}$  doit être égal à  $s''_i(x_{i+1}) = 6 d_i h_i + 2 c_i$ . En tirant  $d_i$  on obtient (2).

La continuité de  $s$  aux nœuds intérieurs se traduit par le fait que  $s_{i+1}(x_{i+1}) = a_{i+1}$  doit être égal à  $s_i(x_{i+1}) = a_i + b_i h_i + c_i h_i^2 + d_i h_i^3$ . En tirant  $b_i$  on obtient (3).

La continuité de la dérivée première de  $s$  aux nœuds intérieurs se traduit par le fait que  $s'_i(x_i) = b_i$  doit être égal à  $s'_{i-1}(x_i) = b_{i-1} + 2 c_{i-1} h_{i-1} + 3 d_{i-1} h_{i-1}^2$ . En remplaçant  $b_{i-1}$ ,  $d_{i-1}$ ,  $b_i$  et  $d_i$  par leur valeur (2), (3), on obtient (4) ci-dessous, qui fournit le système d'équations linéaires à résoudre pour trouver  $c$ .

$$\begin{aligned} \frac{h_{i-1}}{3} c_{i-1} + 2 \left( \frac{h_{i-1}}{3} + \frac{h_i}{3} \right) c_i + \frac{h_i}{3} c_{i+1} = \\ \frac{1}{h_{i-1}} a_{i-1} - \left( \frac{1}{h_{i-1}} + \frac{1}{h_i} \right) a_i + \frac{1}{h_i} a_{i+1}, \quad (1 \leq i \leq n-1). \end{aligned} \quad (4)$$

## 2.3 Splines cubiques interpolantes périodiques

Une spline périodique est une spline dont le graphe est périodique.

Pour qu'elles soient bien définies, on impose que le premier et le dernier point aient même ordonnée ( $y_0 = y_n$ ). La longueur de la période est alors donnée par la différence des abscisses ( $x_n - x_0$ ). Pour obtenir une spline périodique sur l'exemple de l'évaporation de l'eau, il suffit de dupliquer la donnée du mois de janvier :

mois	$x_i$	1	2	3	4	5	6	7	8	9	10	11	12	<b>13</b>
évaporation (pouces)	$y_i$	8.6	7	6.4	4	2.8	1.8	1.8	2.3	3.2	4.7	6.2	7.9	<b>8.6</b>

La spline périodique de l'exemple peut s'obtenir avec la fonction `CubicSpline` de la bibliothèque `scipy`. Il suffit de lui passer le paramètre `bc_type='periodic'`.

**Question 9.** En utilisant `CubicSpline`, tracer le graphe de la spline périodique définie par l'exemple ci-dessus sur l'intervalle  $[1, 20]$ . Vérifier visuellement qu'elle est périodique et que le raccord entre les mois de décembre et de janvier est bien lisse.

### 2.3.1 Calcul des coefficients de la spline

Pour calculer les coefficients de la spline périodique, l'idée consiste à supprimer les deux équations  $c_0 = c_n = 0$  qui donnent une spline naturelle et à les remplacer par deux autres équations  $b_0 = b_n$  et  $c_0 = c_n$  qui imposent que les dérivées première et seconde soient identiques aux deux extrémités.

On introduit la matrice  $A$  de dimension  $n \times n$  suivante (pour une justification, utiliser (4) et le fait que  $h_n, c_n, c_{n+1} = h_0, c_0, c_1$ ) :

$$\begin{pmatrix} \frac{h_0}{3} & 2\left(\frac{h_0}{3} + \frac{h_1}{3}\right) & \frac{h_1}{3} & 0 & \dots & 0 & 0 \\ 0 & \frac{h_1}{3} & 2\left(\frac{h_1}{3} + \frac{h_2}{3}\right) & \frac{h_2}{3} & \dots & 0 & 0 \\ 0 & 0 & \frac{h_2}{3} & 2\left(\frac{h_2}{3} + \frac{h_3}{3}\right) & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 2\left(\frac{h_{n-3}}{3} + \frac{h_{n-2}}{3}\right) & \frac{h_{n-2}}{3} \\ \frac{h_{n-1}}{3} & 0 & 0 & 0 & \dots & \frac{h_{n-2}}{3} & 2\left(\frac{h_{n-2}}{3} + \frac{h_{n-1}}{3}\right) \\ 2\left(\frac{h_{n-1}}{3} + \frac{h_0}{3}\right) & \frac{h_0}{3} & 0 & 0 & \dots & 0 & \frac{h_{n-1}}{3} \end{pmatrix}$$

ainsi que le vecteur  $v$  de dimension  $n$  suivant (c'est le même vecteur qu'en section 2.1, prolongé d'une coordonnée en utilisant l'hypothèse  $y_n, y_{n+1} = y_0, y_1$ ) :

$$v = \begin{pmatrix} \frac{1}{h_0} y_0 - \left(\frac{1}{h_0} + \frac{1}{h_1}\right) y_1 + \frac{1}{h_1} y_2 \\ \frac{1}{h_1} y_1 - \left(\frac{1}{h_1} + \frac{1}{h_2}\right) y_2 + \frac{1}{h_2} y_3 \\ \vdots \\ \frac{1}{h_{n-1}} y_{n-1} - \left(\frac{1}{h_{n-1}} + \frac{1}{h_0}\right) y_0 + \frac{1}{h_0} y_1 \end{pmatrix}.$$

En résolvant le système d'équations linéaires  $Ac = v$ , on obtient les coordonnées  $c_0, \dots, c_{n-1}$  de  $c$  mais il manque encore  $c_n$ . Sachant que  $c_0 = c_n$ , il suffit de rajouter une copie de la première coordonnée de  $c$  à la fin du vecteur pour obtenir le vecteur  $c$  complet.

**Question 10.** Écrire des instructions Python qui construisent la matrice  $A$ , le vecteur  $v$ , qui résolvent le système  $Ac = v$  et qui rajoutent la bonne coordonnée à la fin du vecteur  $c$  (voir ci-dessus).

Note : pour simplifier l'écriture du code, vous pouvez utiliser des indices qui « tournent » dans l'intervalle  $[0, n-1]$  par un calcul de reste de division euclidienne :

```
ipu = (i + 1) % n    # i_plus_un = le reste de la division euclidienne de i+1 par n
ipd = (i + 2) % n    # i_plus_deux = le reste de la division euclidienne de i+2 par n
```

**Question 11.** Donner les instructions Python permettant de calculer les tableaux  $a$ ,  $b$ ,  $c$  et  $d$  (les formules sont les mêmes que pour les splines naturelles).



**Question 12.** Adapter en une fonction `eval_spline_periodique` la fonction d'évaluation écrite pour les splines naturelles. L'algorithme est le suivant : si le paramètre  $z \notin [x_0, x_n]$  alors il faut lui ajouter ou lui retrancher un multiple de la période  $x_n - x_0$  pour se ramener au cas  $z \in [x_0, x_n]$  ; il suffit ensuite de retourner  $s(z)$  comme dans le cas de la spline naturelle.

Vérifier votre résultat en comparant les graphes obtenus avec votre fonction et celui de `CubicSpline`.

**Question 13.** Une baraque à frites ouvre quatre jours par semaine. Le tableau ci-dessous donne les pommes de terre consommées en kilos pour chaque jour. Le propriétaire décide d'ouvrir tous les jours. Il souhaiterait une estimation du nombre de kilos de pommes de terre à acheter pour la semaine.

jour	$x_i$	Di	Lu	Ma	Me	Je	Ve	Sa
qté cons.	$y_i$	40			21	23		44

## 2.4 Splines cubiques lissantes naturelles

Un exemple suffit à expliquer pourquoi les splines lissantes sont très importantes en pratique. Voir Figure 2. Elles sont le sujet de [2, Chapter 5].

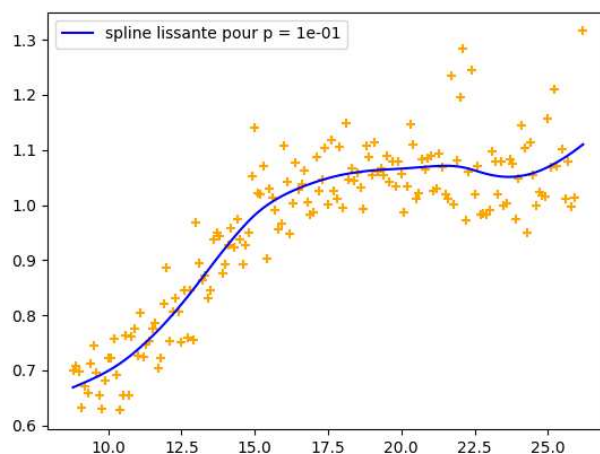


FIGURE 2 – Spline lissante obtenue sur les données du fichier `nuage_de_points.py` (il s'agit de mesures de densité minérale osseuse) en fixant le paramètre de lissage  $p$  à  $10^{-1}$ .

On suppose donnés  $n+1$  points. On suppose les nœuds triés par ordre croissant. Le texte fondateur est celui de Reinsch [3]. L'idée consiste à fixer un paramètre  $S > 0$  et à chercher, parmi toutes les fonctions deux fois dérivables  $g(x)$  qui vérifient la contrainte

$$\sum_{i=0}^n (g(x_i) - y_i)^2 \leq S, \quad (5)$$

la fonction — notons-la  $s(x)$  — qui minimise l'intégrale

$$\int_{x_0}^{x_n} (g''(x))^2 dx \quad (6)$$

On peut montrer que la fonction  $s(x)$  recherchée est une spline cubique. Elle est interpolante si  $S = 0$  mais elle ne l'est plus dès que  $S > 0$ . Le paramètre  $S$  contrôle l'importance du lissage mais il n'est pas très pratique à manipuler. Dans ce TP, on le remplace par un autre paramètre  $p > 0$  qui a le mérite de simplifier les calculs (chacun des deux paramètres peut se calculer à partir de l'autre).

On reprend la matrice  $A$  de dimension  $(n-1) \times (n-1)$  introduite en section 2.1 et on introduit la matrice  $Q$  de dimension  $(n+1) \times (n-1)$  suivante :

$$Q = \begin{pmatrix} \frac{1}{h_0} & 0 & \dots & 0 & 0 \\ -\left(\frac{1}{h_0} + \frac{1}{h_1}\right) & \frac{1}{h_1} & \dots & 0 & 0 \\ \frac{1}{h_1} & -\left(\frac{1}{h_1} + \frac{1}{h_2}\right) & \dots & 0 & 0 \\ 0 & \frac{1}{h_2} & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & -\left(\frac{1}{h_{n-3}} + \frac{1}{h_{n-2}}\right) & \frac{1}{h_{n-2}} \\ 0 & 0 & \dots & \frac{1}{h_{n-2}} & -\left(\frac{1}{h_{n-2}} + \frac{1}{h_{n-1}}\right) \\ 0 & 0 & \dots & 0 & \frac{1}{h_{n-1}} \end{pmatrix}.$$

**Calcul de  $c$  et de  $a$ .** Les étapes de l'algorithme sont :

1. calculer  $B = Q^T Q + p A$  ;
2. calculer  $v = p Q^T y$  ;
3. résoudre  $B c = v$  ;
4. prendre  $a = y - \frac{1}{p} Q c$  ;
5. rajouter deux 0 à  $c$  (comme en section 2.1) pour obtenir le vecteur  $c$  complet.

Les formules pour  $b$  et  $d$  sont inchangées. L'évaluation se fait avec `eval_spline_naturelle`.

Note : dans le cas où le paramètre  $p$  tend vers l'infini, la spline lissante tend vers la spline interpolante. Le vecteur  $v$  donné en section 2.1 peut s'obtenir à partir de  $Q$  par la formule  $v = Q^T y$ .

**Question 14.** Écrire les instructions Python qui calculent les vecteurs  $a$ ,  $b$ ,  $c$  et  $d$  à partir de  $p$ ,  $x$  et  $y$ . Tracer le graphe de la spline lissante. Les instructions suivantes peuvent vous être utiles :

```
np.transpose (Q)    # retourne la transposée de Q
np.dot (A, B)       # retourne le produit de la matrice A par la matrice B
np.dot (A, v)       # retourne le produit de la matrice A par le vecteur v
```

Le fichier de données de la figure 2 s'obtient par

```
from nuage_de_points import x,y
```

Remarque : des splines lissantes sont implantées en Python (chercher « *smoothing splines* » dans la documentation de `scipy`) mais je ne suis pas parvenu à les faire fonctionner correctement.

La justification de la théorie des splines lissantes est trop longue pour être incorporée au sujet de TP. Elle est détaillée dans le document `smoothing-splines.pdf` disponible sur le dépôt git du cours.

### 3 Conclusion

Dans les applications concrètes, il est rare qu'on utilise le polynôme d'interpolation défini par l'ensemble des points parce que son graphe peut présenter des caractéristiques qui le rendent , notamment aux  de l'intervalle d'interpolation. On préfère souvent interpoler les points avec d'autres méthodes telles que les , dont le graphe ressemble davantage au graphe qu'un être humain aurait tracé à la main. Les splines aussi mettent en œuvre des méthodes d'interpolation polynomiale mais de façon plus subtile : les équations qui définissent une spline  interpolante imposent non seulement que son graphe passe par les points à interpoler mais que les dérivées  et  des

cubiques soient égales au niveau des . Il existe plusieurs variétés de splines cubiques : les splines  ont leurs dérivées secondes nulles aux  ; les splines  sont bien adaptées à des données qui se répètent de façon hebdomadaire ou annuelle. Enfin, lorsque les données constituent des nuages de points, il devient nécessaire d'employer des splines .

## Références

- [1] L. C. Birch and H. G. Andrewartha. *The Distribution and Abundance of Animals*. The University of Chicago Press, 1954.
- [2] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning. Data Mining, Inference and Prediction*. Springer Series in Statistics. Springer, 2nd edition, 2009. Available at <https://hastie.su.domains/ElemStatLearn>.
- [3] Christian H. Reinsch. Smoothing by Spline Functions. *Numerische Mathematik*, 10 :177–183, 1967.

Soit  $f : \mathbb{R} \rightarrow \mathbb{R}$  une fonction d'une variable réelle et  $[a, b]$  un intervalle sur lequel  $f$  est définie. On cherche à calculer

$$\int_a^b f(x) dx. \quad (1)$$

Si on connaît une primitive  $F$  de  $f$  alors l'intégrale (1) est facile à calculer : elle vaut  $F(b) - F(a)$ . Malheureusement  $F$  n'est pas toujours connue<sup>1</sup>. Les méthodes étudiées dans ce TP, appelées *formules de quadrature*, contournent cette difficulté mais ne calculent qu'une approximation plus ou moins précise de l'intégrale recherchée.

## 1 Fabrication d'un exemple

On triche ! Pour vérifier les calculs, il est important de prendre une fonction un peu générale (éviter les polynômes de petit degré) et des bornes  $a$  et  $b$  un peu arbitraires. Une façon simple de construire des exemples consiste à partir d'une primitive  $F(x)$  et d'en déduire la fonction  $f(x) = F'(x)$  à intégrer. La solution attendue est alors  $F(b) - F(a)$ . Le paquetage `sympy` de Python nous permet de calculer les dérivées automatiquement. Partons par exemple de

$$F(x) = \frac{x^{-\frac{1}{2}}}{(1+x)^{\frac{25}{2}}} e^{-x} \cos x$$

```
from sympy import *
x = var('x')
F = x**(-Rational(1,2))/(1+x)**Rational(25,2)*exp(-x)*cos(x)
```

Sa dérivée est une expression compliquée à souhait !

```
f = Derivative (F).doit ()

-exp(-x)*sin(x)/(sqrt(x)*(x + 1)**(25/2))
- exp(-x)*cos(x)/(sqrt(x)*(x + 1)**(25/2))
- 25*exp(-x)*cos(x)/(2*sqrt(x)*(x + 1)**(27/2))
- exp(-x)*cos(x)/(2*x**(3/2)*(x + 1)**(25/2))
```

$$f(x) = -\frac{e^{-x} \sin(x)}{\sqrt{x}(x+1)^{\frac{25}{2}}} - \frac{e^{-x} \cos(x)}{\sqrt{x}(x+1)^{\frac{25}{2}}} - \frac{25e^{-x} \cos(x)}{2\sqrt{x}(x+1)^{\frac{27}{2}}} - \frac{e^{-x} \cos(x)}{2x^{\frac{3}{2}}(x+1)^{\frac{25}{2}}}$$

Il ne reste plus qu'à coder la primitive et sa dérivée sous la forme de fonctions Python. Pour vous aider, un petit module de génération de code, contenant une fonction nommée `Python`, est disponible dans le dépôt git du cours. Les codes Python écrits par cette fonction supposent chargé le paquetage `math`.

```
>>> from code_generation import Python
>>> Python (F, fname='F')
def F (x) :
    t1 = 1. / math.sqrt (x)
    t3 = 1 + x
    t2 = t3**(-25/2)
    t4 = math.cos (x)
    t6 = (-1) * x
    t5 = math.exp (t6)
    t0 = t1 * t2 * t4 * t5
    return t0
```

1. D'une part, pour la plupart des fonctions  $f$ , il n'existe aucune formule finie pour  $F$  ne faisant figurer que les fonctions bien connues (exponentielle, sinus, arc tangente, etc.) ; d'autre part, il est courant que la fonction  $f$  soit donnée sous la forme d'un programme d'ordinateur ou comme la sortie d'un appareil de mesure et dans ce cas-là, pas de primitive non plus.

```
>>> Python (f, fname='f')
def f (x) :
    t2 = 1. / math.sqrt (x)
    t4 = 1 + x
    t3 = t4**(-25/2)
    t5 = math.cos (x)
    t7 = (-1) * x
    t6 = math.exp (t7)
    t1 = (-1) * t2 * t3 * t5 * t6
    t9 = math.sin (x)
    t8 = (-1) * t2 * t3 * t6 * t9
    t11 = t4**(-27/2)
    t10 = (-25/2) * t2 * t11 * t5 * t6
    t13 = x**(-3/2)
    t12 = (-1/2) * t13 * t3 * t5 * t6
    t0 = t1 + t8 + t10 + t12
    return t0
```

Fixons deux bornes

```
a = .20345
b = .01
```

Le but du jeu consiste à retrouver la valeur suivante qui vaut approximativement 8.567158575462104 :

```
integrale_exacte = F(b) - F(a)
```

Pour vous aider, voici quelques valeurs attendues pour la dérivée :

```
f(.1), f(.2), f(.3)

(-15.105318555935721, -2.593771026134607, -0.6124760076003707)
```

## 2 La formule des trapèzes

On suppose donnés  $f$ ,  $a$  et  $b$  ainsi qu'un nombre de pas  $n$ . On note

$$h = \frac{b - a}{n}$$

la longueur d'un pas, et  $x_i = a + i h$  (pour  $0 \leq i \leq n$ ) les abscisses de  $n + 1$  points. On note  $y_i = f(x_i)$  les ordonnées de ces points. On a donc une approximation du graphe de  $f$  sous la forme d'une suite de points dont les abscisses sont équidistantes (Figure 1). La méthode consiste à relier les points par des segments de droite. Chaque segment définit un trapèze (Figure 2). La somme des surfaces des  $n$  trapèzes fournit une approximation  $I$  de l'intégrale recherchée. Cette somme se simplifie un peu parce qu'on a supposé les abscisses équidistantes. La formule de quadrature ainsi obtenue est appelée *formule des trapèzes composite* :

$$I = h \left( \frac{y_0}{2} + y_1 + y_2 + \cdots + y_{n-1} + \frac{y_n}{2} \right). \quad (2)$$

**Question 1.** Programmer une fonction `trapezes`, paramétrée par une fonction  $f$ , deux flottants  $a, b$ , un entier  $n$ , qui retourne une approximation de (1) par la formule (2).

**Question 2.** Vérifier votre fonction sur l'exemple de la section 1.

**Question 3.** Écrire un fichier `test_tp5.py` contenant quelques tests unitaires de la fonction `trapezes`.

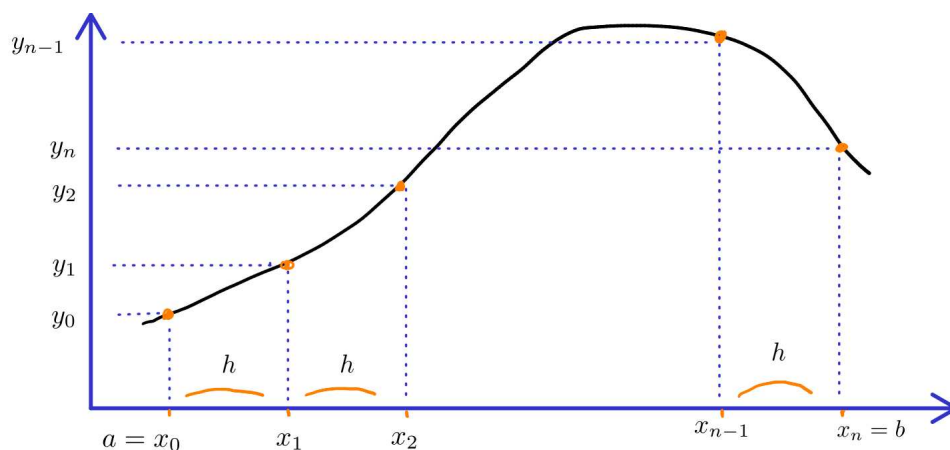


FIGURE 1 – Pour approximer l'intégrale (1), la méthode des trapèzes a uniquement besoin de connaître  $n + 1$  points appartenant au graphe de  $f$ .

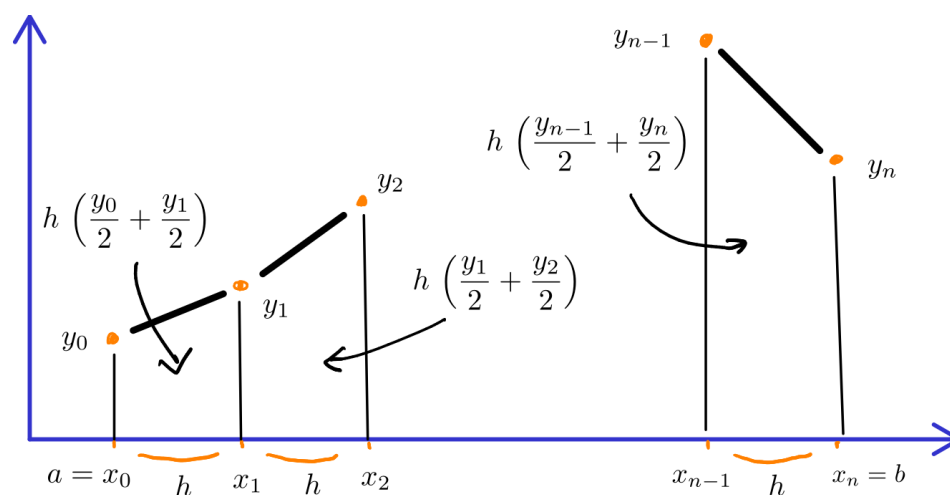


FIGURE 2 – La formule des trapèzes consiste à approximer l'intégrale (1) par la somme des aires des  $n$  trapèzes définis par les  $n + 1$  points appartenant au graphe de  $f$ .

**Question 4.** Pour estimer le nombre de décimales exactes obtenues par calcul, on peut utiliser la formule suivante :

$$\begin{aligned} \text{erreur} &= \frac{\text{valeur exacte} - \text{valeur calculée}}{\text{valeur exacte}}, \\ \text{nb décimales exactes} &\simeq -\log_{10} |\text{erreur}| \end{aligned}$$

En utilisant cette formule, combien de pas faut-il pour obtenir 2, 4, 6 décimales exactes. Qu'observe-t-on ?

## 2.1 Ordre de la méthode des trapèzes

**Définition de l'ordre.** Une formule de quadrature est dite d'ordre  $p$  si elle est exacte<sup>2</sup> pour tout polynôme  $f$  de degré strictement inférieur à  $p$ .

2. Une formule de quadrature est dite *exacte* si elle donne la valeur exacte de l'intégrale (1), en supposant qu'il n'y ait aucune erreur d'arrondi.

**Erreur en fonction de l'ordre.** Supposons qu'on applique une formule de quadrature d'ordre  $p$  pour approximer l'intégrale (1) d'une fonction  $f$  au moins  $p$  fois dérivable. Alors l'erreur vérifie l'inégalité

$$\text{erreur} \leq c h^p \quad (3)$$

où  $c$  est un nombre qui dépend de la largeur  $b - a$  de l'intervalle d'intégration et de la fonction  $f$ . Pour un énoncé plus précis, voir [1, Théorème 2.3].

**Diminution de l'erreur en fonction de l'ordre et d'une diminution de la longueur du pas.** Supposons qu'on applique une formule de quadrature d'ordre  $p$  à une fonction  $f$  sur un intervalle  $[a, b]$  pour un nombre de pas  $n_0$ . Pour simplifier, supposons que  $b - a = 1$  de telle sorte que le pas  $h_0 = 1/n_0$  et que, dans la formule (3), le symbole d'inégalité soit une égalité et que  $c = 1$ . On obtient un résultat avec une erreur

$$\text{erreur}_0 = h_0^p.$$

Supposons qu'on refasse exactement le même calcul mais avec un nombre de pas  $n_1 = 10 n_0$  ou, ce qui revient au même, une longueur de pas  $h_1 = h_0/10$ . On obtient un résultat avec une erreur

$$\text{erreur}_1 = h_1^p = \frac{h_0^p}{10^p} = \frac{\text{erreur}_0}{10^p}.$$

L'erreur a été divisée par  $10^p$  : on a donc gagné  $p$  décimales exactes.

**Question 5.** L'ordre de la méthode des trapèzes peut donc se vérifier expérimentalement : c'est la pente de la droite tracée par les instructions suivantes. Note : pour éviter les dépassements de capacité, on a utilisé la base 2 plutôt que la base 10 mais le raisonnement est le même. Que vaut cet ordre ?

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

xplot = np.array ([k for k in range (6,12)])
yplot = np.array ([-math.log2(abs((integrale_exacte - trapezes(f, a, b, 2**k))/integrale_exacte))
                  for k in xplot])
plt.plot (xplot, yplot)

droite_regression_lineaire = np.polynomial.polynomial.polyfit(xplot, yplot, 1)
pente = droite_regression_lineaire[1]
pente = np.around (pente, 2)

plt.xlabel ('logarithme en base 2 du nombre de pas')
plt.ylabel ('nombre de bits exacts')
plt.scatter (xplot, yplot,
            label='méthode des trapèzes: pente = ' + str(pente),
            color='blue')
plt.legend ()
plt.show ()
```

## 2.2 Lien entre méthode des trapèzes et interpolation

Il existe un lien entre la méthode des trapèzes et l'interpolation polynomiale. Intéressons-nous au premier trapèze de la figure 2 qui est défini par les points  $(x_0, y_0)$  et  $(x_0 + h, y_1)$ . Notons  $p$  le polynôme d'interpolation défini par ces points. Le graphe de  $p$  est la droite qui passe par ces points et on a

$$\text{aire du premier trapèze} = \int_{x_0}^{x_0+h} p(x) dx = h \left( \frac{y_0}{2} + \frac{y_1}{2} \right).$$

### 3 La méthode de Simpson

Plutôt que de prendre les points deux par deux et de les relier par des segments de droites, on prend trois par trois, on les relie par des arcs de paraboles et on intègre ces paraboles (Figure 3). Il faut bien sûr que  $n$  soit un nombre pair. Intéressons-nous à la première parabole de la figure 3 qui est définie

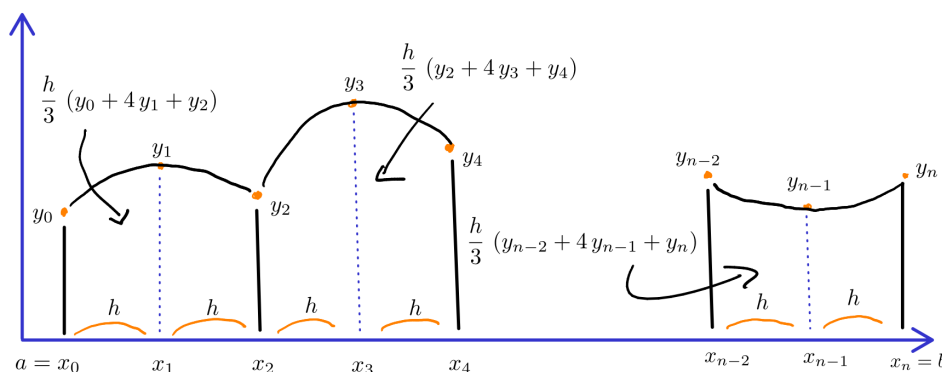


FIGURE 3 – Pour approximer l'intégrale (1), la méthode de Simpson relie les points par des arcs de paraboles et les intègre. Il est bien sûr nécessaire que  $n$  soit pair.

par les points  $(x_0, y_0)$ ,  $(x_0 + h, y_1)$  et  $(x_0 + 2h, y_2)$ . Notons  $p$  le polynôme d'interpolation défini par ces points. Le graphe de  $p$  est la parabole qui passe par ces points et on a (formule de Simpson<sup>3</sup>)

$$\text{intégrale de la première parabole} = \int_{x_0}^{x_0+2h} p(x) dx = \frac{h}{3} (y_0 + 4y_1 + y_2).$$

La somme des intégrales des paraboles fournit une approximation  $I$  de l'intégrale recherchée. Cette somme se simplifie un peu parce qu'on a supposé les abscisses équidistantes. La formule de quadrature obtenue est appelée *formule de Simpson composite* :

$$I = \frac{h}{3} (y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \dots + 2y_{n-2} + 4y_{n-1} + y_n). \quad (4)$$

**Question 6.** Programmer une fonction `Simpson`, paramétrée par une fonction  $f$ , deux flottants  $a, b$ , un entier  $n$  supposé pair, qui retourne une approximation de (1) par la formule (4). Vérifier votre fonction sur l'exemple de la section 1.

**Question 7.** Reprendre la question sur la vérification expérimentale de l'ordre de la méthode des trapèzes pour l'adapter à la méthode de Simpson. Superposer les deux graphiques. Quel semble être l'ordre de la méthode de Simpson ?

**Question 8.** Compléter `test_tp5.py`

**Note.** Que fait-on lorsque  $n$  est impair et qu'on souhaite utiliser la méthode de Simpson ? On applique la formule de Simpson composite sur tous les points jusqu'à ce qu'il ne reste plus que trois intervalles à prendre en compte. À ces trois intervalles, on applique la formule suivante, qui a le même ordre que la formule de Simpson. Notons  $p$  la cubique définie par les trois derniers intervalles. On a

$$\int_{x_0}^{x_0+3h} p(x) dx = \frac{3h}{8} (y_0 + 3y_1 + 3y_2 + y_3).$$

3. Thomas Simpson (1710-1761) qui affirme que la formule lui a en fait été communiquée par Newton.



### 3.1 Conclusion sur les formules des trapèzes et de Simpson

Pour définir un trapèze, il suffit de deux points. On dit que la formule des trapèzes est une formule à  $s = 2$  étages. Pour définir une parabole, il suffit de trois points. On dit que la formule de Simpson est une formule à  $s = 3$  étages.

Le raisonnement qui conduit à la formule de Simpson se généralise et permet de définir des formules d'un nombre quelconque d'étages : ce sont les formules dites de Newton-Cotes et on peut montrer que l'ordre  $p$  d'une formule de Newton-Cotes à  $s$  étages est égal au plus petit nombre pair supérieur ou égal à  $s$  [1, Théorème 1.4, page 4].

Les deux premières formules de Newton-Cotes (les trapèzes et Simpson) sont très populaires, surtout lorsqu'on ne dispose pas de la fonction  $f$  mais d'une suite de points  $(x_i, y_i)$  donnés par un appareil de mesure, une étude statistique et où les données peuvent être bruitées (imprécision des capteurs, erreurs de mesure, ...).

**Question 9.** Dans la documentation de `scipy.integrate`, quelle est l'expression utilisée pour désigner le cas où on ne dispose pas de la fonction  $f$ ? Quelles sont les formules de Newton-Cotes proposées? Sous quel nom?

Réponse.

## 4 Les formules de quadrature de Gauss

Dans les cas où on dispose de la fonction  $f$  (au minimum d'un algorithme pour l'évaluer) et où cette fonction est suffisamment dérivable, on a tout intérêt à utiliser des formules d'ordre élevé.

On suppose qu'on dispose de la fonction  $f$ . Gauss a montré que, si on s'autorise à placer les abscisses  $x_i$  d'une façon non équidistante et bien choisie, on peut obtenir des formules de quadrature à  $s$  étages d'ordre  $p = 2s$  [1, Section I.3]. Voici un exemple d'une formule de quadrature de Gauss à  $s = 3$  étages :

$$\begin{aligned} \int_a^b f(x) dx &\simeq (b-a) \sum_{i=0}^2 d_i f(a + c_i (b-a)) \quad \text{avec} \\ (d_0, d_1, d_2) &= \left( \frac{5}{18}, \frac{8}{18}, \frac{5}{18} \right), \\ (c_0, c_1, c_2) &= \left( \frac{1}{2} - \frac{\sqrt{15}}{10}, \frac{1}{2}, \frac{1}{2} + \frac{\sqrt{15}}{10} \right). \end{aligned} \tag{5}$$

**Question 10.** D'après ce qui précède, quel est l'ordre de cette formule?

**Question 11.** Programmer une fonction `Gauss6` paramétrée par  $f$ ,  $a$ ,  $b$  et un entier  $n$  et qui applique la méthode décrite Figure 4.

**Question 12.** Vérifier expérimentalement votre réponse sur l'exemple de la section 1.

Note : la formule (3) reste valable lorsque les abscisses  $x_i$  ne sont pas équidistantes. Dans ce cas,  $h$  désigne le maximum des longueurs des pas.

**Question 13.** Tester la fonction `Gauss6` sur l'exemple suivant. Vérifier le résultat en utilisant la constante `math.pi`.

$$\int_0^1 \frac{4}{x^2 + 1} dx = \pi.$$

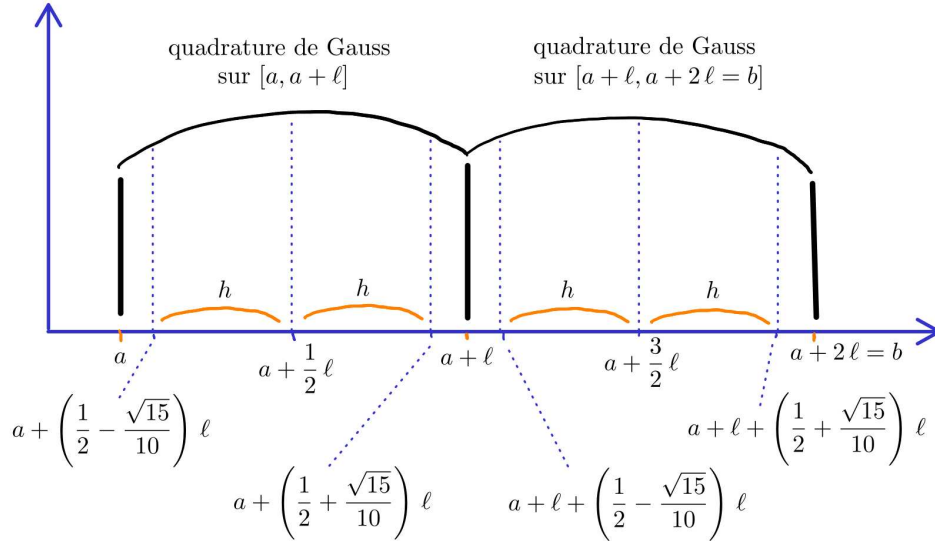


FIGURE 4 – Pour approximer l’intégrale (1), la fonction `Gauss6` subdivise l’intervalle  $[a, b]$  en  $n$  (ici  $n = 2$ ) sous-intervalles de longueur  $\ell$ . Elle applique la formule (5) sur chacun des  $n$  sous-intervalles et retourne la somme des  $n$  intégrales.

**Question 14.** Compléter `test_tp5.py`.

**Note.** Les formules de quadrature de Gauss sont implantées dans la fonction `fixed_quad` de la bibliothèque `scipy.integrate`. Le nombre d’étages (improprement appelé *order* dans la documentation Python) est spécifié par le paramètre  $n$ .

## 5 Intégrales généralisées

### 5.1 L’intégrale de Gauss

Une variable aléatoire  $X$  possède une densité de probabilité  $f$  si la probabilité que  $X \in [a, b]$  est égale à

$$\int_a^b f(x) dx.$$

Par conséquent, pour qu’une fonction  $f$  soit une densité de probabilité, il faut que

$$\int_{-\infty}^{+\infty} f(x) dx = 1. \quad (6)$$

Dans la théorie des probabilités, on est amené à étudier plusieurs densités de probabilité données par des courbes en cloche de la forme ( $c$  est un paramètre) :

$$f(x) = c e^{-x^2}.$$

Pour que ces fonctions vérifient la contrainte (6), il est nécessaire d’ajuster la valeur de  $c$ . Par exemple, comme

$$\int_{-\infty}^{+\infty} e^{-x^2} dx = \sqrt{\pi}, \quad (7)$$

il est nécessaire de prendre  $c = 1/\sqrt{\pi}$ . L’intégrale (7) est appelée *intégrale de Gauss*.

## 5.2 Calcul de l'intégrale de Gauss

On cherche à calculer

$$\int_0^{+\infty} e^{-x^2} dx = \frac{\sqrt{\pi}}{2}. \quad (8)$$

L'idée consiste à ramener l'intégrale (8) à une intégrale entre 0 et 1, par le changement de variable  $x = \frac{1}{t} - 1$ . Voir Figure 5.

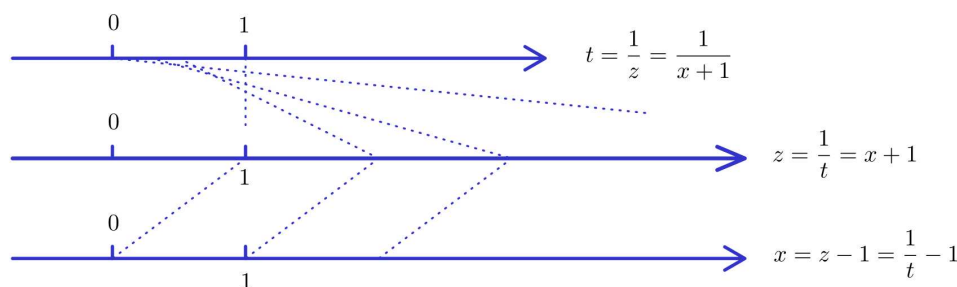


FIGURE 5 – Le changement de variable  $x = \frac{1}{t} - 1$  met en bijection  $x \in [0, +\infty[$  avec  $t \in [0, 1]$ .

**Question 15.** Reformuler l'intégrale (8) en appliquant le théorème bien connu :

**Théorème 1** Supposons  $x = \varphi(t)$  avec  $\varphi(\alpha) = a$  et  $\varphi(\beta) = b$ . Alors

$$\int_a^b f(x) dx = \int_\alpha^\beta f(\varphi(t)) \varphi'(t) dt.$$

**Question 16.** Est-il possible d'utiliser la formule des trapèzes ou la formule de Simpson pour approximer l'intégrale ci-dessus ?

**Question 17.** Est-il possible d'utiliser la quadrature de Gauss de la section précédente ?

**Question 18.** Estimer numériquement l'intégrale.

## 6 Conclusion

Pour intégrer une fonction  $f$  sur un intervalle  $[a, b]$ , les méthodes numériques ne calculent pas une  de  $f$ . Elles  la fonction sur les abscisses  $x_i \in [a, b]$  de plusieurs points et retournent une combinaison linéaire des valeurs ainsi obtenues. Même en supposant qu'il n'y ait aucune erreur d'arrondi, ces formules de  ne retournent qu'une  du résultat exact.

L'efficacité de ces formules est résumée par un nombre, leur , qui vaut  pour la formule des trapèzes et  pour celle de .

La formule des trapèzes et celle de  sont des cas particuliers des formules de . Elles sont bien adaptées au cas où on ne dispose pas de la fonction  $f$  mais uniquement d'une suite de .

Lorsqu'on dispose de la fonction  $f$ , on a tout intérêt à utiliser des formules  élevé telles que les formules de  de . Ces dernières permettent également de calculer des intégrales .

## Références

- [1] Ernst Hairer. Polycopié du cours “Analyse Numérique”. Accessible sur [www.unige.ch/~hairer/poly/poly.pdf](http://www.unige.ch/~hairer/poly/poly.pdf), octobre 2001.