Module C1

Calcul scientifique avec Python 3

Édition révisée

Timothée Duruisseau

Département de génie mécanique
Polytechnique Montréal

12 septembre 2023

Objectifs du module

Dans la pratique courante de l'ingénierie, le professionnel est souvent confronté à des problèmes mathématiques d'envergures. La résolution exacte de ces problèmes n'est généralement pas nécessaire. Seule une solution approximative est habituellement requise si la précision de calcul est suffisante. Ainsi, les méthodes numériques abordées dans ce module permettent de calculer des fonctions, des racines, des dérivées ou des intégrales.

À la fin du module C1, l'étudiant devra être en mesure de calculer et afficher des fonctions mathématiques, trouver les racines et intersections de fonctions, ainsi que faire des dérivations et des intégrations numériques. La maîtrise du traçage des courbes en 2D est essentielle à la poursuite des modules suivants.

Table des matières

Chapitre 1

Calculer une fonction

1.1 Introduction

Une fonction mathématique, telle que $y = f(x)$, est une relation explicite entre des variables (ici x et y). Dans le langage Python, une fonction logicielle est plutôt la définition d'une portion de code permettant de calculer des variables sorties à partir de variables d'entrées.

Dans un petit script Python, il est possible de calculer directement dans le script principal la fonction mathématique. Il faut alors éviter de répéter ce calcul à plusieurs endroits afin de faciliter les futures modifications de votre fonction mathématique. Dans un script plus grand, il est préférable créer une fonction Python qui calcul la fonction mathématique. Ainsi, les modifications de la fonction mathématique peuvent se faire à un seul endroit. De plus, il devient alors possible de transmettre le nom de cette fonction Python à d'autres fonctions afin de demander son utilisation.

Dans un souci de simplicité et d'optimiser le traitement des données, les vecteurs et matrices sont préférés, permettant de faire des calculs sur l'ensemble des données en une seule commande. Par exemple, la fonction mathématique $f(x) = 3x + 2$ peut être directement écrite en Python de la manière `f = 3*x+2`, avec la variable indépendante (ou d'entrée) `x` préalablement définie comme étant un scalaire, un vecteur ou une matrice. La variable dépendante (ou de sortie) `f` sera alors du même type et dimension que la variable indépendante `x`.

1.2 Calculer directement une fonction mathématique

Lorsque le script principal est petit, il suffit de programmer directement dans le script principal le calcul de la fonction mathématique à un seul endroit.

Le script 1 montre le calcul de la fonction mathématique $f(x) = 3x^{1.5}$ directement dans le script principal. La figure 1.1 montre le graphique résultant du script 1. Dans ce script : les lignes 2-3 importent le module `numpy` (calcul numérique) et `matplotlib` (traçage) ; la ligne 6 crée le vecteur `x` contenant 10 valeurs uniformément réparties de 0 à 9 (soit 0, 1, 2, ... , 8, 9) ; la ligne 9 crée le vecteur `f` de même dimension que `x` et contenant le calcul de la fonction $f(x) = 3x^{1.5}$ pour chacune des valeurs de `x` en une seule commande ; la ligne 12 crée l'objet `fig` de type Figure et l'objet `ax` de type Axes ; les lignes 13-16 demandent un tracé solide rouge avec des cercles localisés aux points et des titres ; finalement, la ligne 18 demande l'affichage de toutes les figures ouvertes (ici, il n'y en a qu'une seule).

SCRIPT 1
Calcul d'une fonction mathématique et affichage d'un graphique

```
# Importation des modules
import numpy as np
import matplotlib.pyplot as plt

# Calcul 10 valeurs de x uniformement reparties de 0 a 9
x = np.linspace(0,9,10)

# Calcul de f(x) = 3*x**1.5 (3 fois x puissance 1.5)
f = 3*x**1.5

# Creer un objet de type Figure et un objet de type Axes
fig, ax = plt.subplots()
ax.plot(x,f,'-or')
ax.set_xlabel('Axe x')
ax.set_ylabel('Axe y')
ax.set_title('Graphique de la fonction f(x) = 3*x**1.5')

plt.show() # Affiche la figure
```

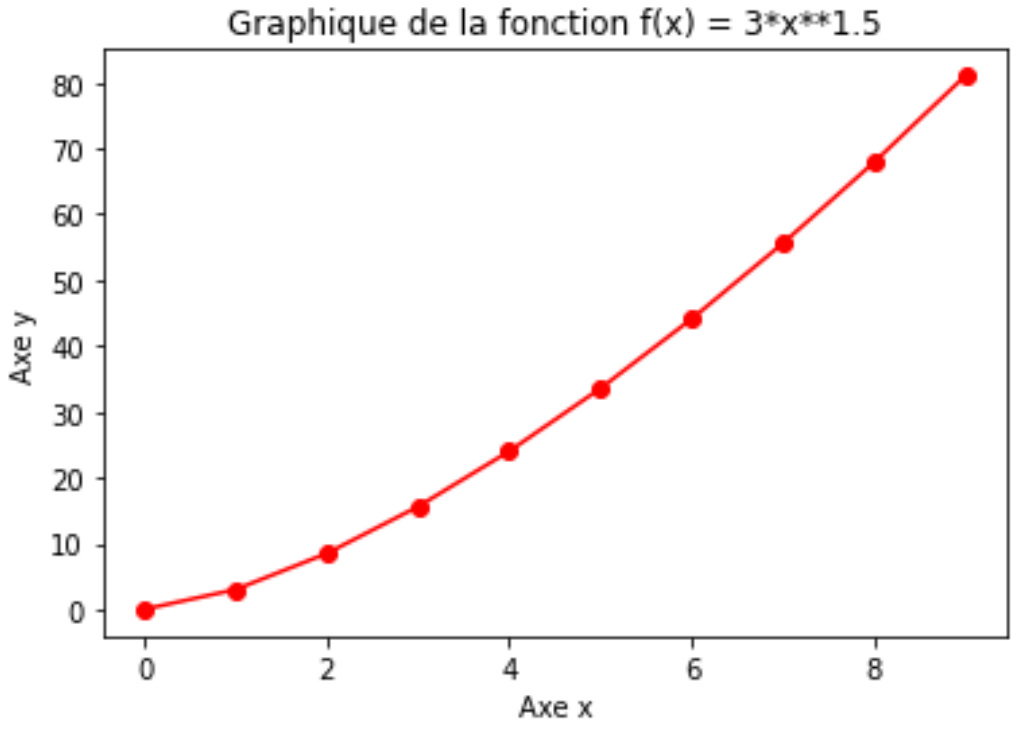

FIGURE 1.1 – Graphique de $f(x) = 3x^{1.5}$ résultant du script 1

1.3 Créer une fonction Python

Lorsque le script est plus grand, une bonne pratique de programmation est de créer une fonction Python pour calculer la fonction mathématique. La définition de cette fonction Python peut soit être programmée directement dans le script principal ou programmée dans un script secondaire, mais inclus dans le script principal par le mot-clé `import`. Afin de simplifier la gestion des scripts, il est préférable que le script principal et le script secondaire soient situés dans le même dossier.

La création d'une fonction Python se fait par : `def nom_fonction(entrees):`. Les lignes suivantes font partie de la fonction et doivent être indentées. La fin de la fonction se fait par `return sorties`. La partie `sorties` définit ce qui sera retourné par la fonction et qui peut être soit une variable calculée aux lignes précédentes ou être directement l'expression de calcul de la fonction mathématique. Dans ce dernier cas, il est fréquent que cette expression soit entièrement inscrite à la suite du `return`. Par exemple, `def f(x) : return 2*x/9`,où `x` est la variable d'entrée indépendante et la valeur dépendante de sortie est directement calculée par l'expression mathématique à droite du `return`. L'argument d'entrée doit préalablement être défini avant l'appel de la fonction. Le nom utilisé dans la définition est pour l'utilisation interne à la fonction seulement et donc n'a pas d'impact sur le reste du script.

Le script 2 montre la définition d'une fonction Python directement dans le script principal afin de calculer la fonction mathématique $y = f(x) = 3\frac{e^x}{\log(x+2)}$. La figure 1.2 montre le graphique résultant du script 2. Dans ce script : les lignes 1-2 importent le module `numpy` (calcul numérique) et `matplotlib` (traçage) ; les lignes 5-6 définissent la fonction Python nommée `fonc` avec `x` comme variable indépendante d'entrée et le `return` contient l'expression de la fonction mathématique ; la ligne 8 crée le vecteur `x` contenant 33 valeurs uniformément réparties de 0 à 8 (soit 0, 0.25, 0.5, 0.75, ... , 7.75, 8) ; la ligne 9 crée le vecteur `y` de même dimension que `x` et demande l'exécution de la fonction Python `fonc(x)` pour chacune des valeurs de `x` en une seule commande ; la ligne 11 ouvre un objet de type Figure ; les lignes 12-18 demandent un tracé tireté bleu avec des carrés localisés aux points, des titres, des grilles et une légende ; finalement, la ligne 20 demande l'affichage de la figure.

SCRIPT 2
Définition d'une fonction et son utilisation

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

# Definition d'une fonction
def fonc(x):
    return 3*np.exp(x)/np.log(x+2)

x = np.linspace(0,8,33)  # Calcul 33 valeurs de x de 0 a 8
y = fonc(x)              # Calcul la fonction aux valeurs de x

plt.figure(1)    # Creer une figure avec titres, legende et grilles
plt.plot(x,y,'--sb',label='y = 3*e**x/log(x+2)')
plt.xlabel('Titre de l\'axe x')
plt.ylabel('Titre de l\'axe y')
plt.title('Titre principal ligne 1\nTitre principal ligne 2')
plt.legend()
plt.grid(axis='x')
plt.grid(axis='y')

plt.show()       # Affiche la figure
```

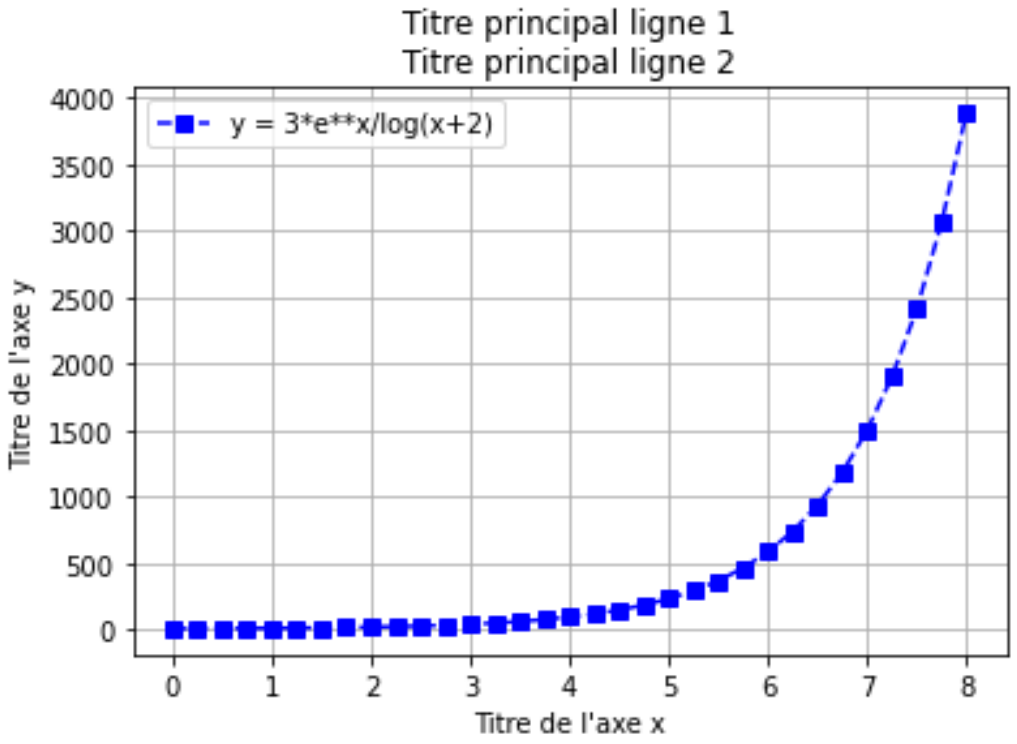

FIGURE 1.2 – Graphique de $f(x) = 3\frac{e^x}{\log(x+2)}$ résultant du script 2

Chapitre 2

Calculer une racine

2.1 Introduction

Plusieurs problèmes de résolution numérique d'une équation algébrique, telle que $f(x) = b$, peuvent se réduire au calcul des racines de la fonction modifiée $g(x) = f(x) - b = 0$, dont il suffit de calculer approximativement les racines. Par exemple, le calcul approximatif de $\sqrt{2}$ peut être formulé comme le calcul des racines de $f(x) = x^2 - 2$. Dans ce chapitre, nous abordons le calcul des racines de polynômes, puis de l'intersection de ceux-ci. Les racines des fonctions nonlinéaires sont calculées par la méthode de la bissection et par une fonction d'optimisation de la librairie scientifique `scipy`.

2.2 Fonctions polynomiales

2.2.1 Polynôme de degré 1

Soit le polynôme en x de degré 1 avec $a_1 \neq 0$:

$$p_1(x) = a_0 x + a_1$$

possède une seule racine à

$$r_1 = -a_1/a_0$$

2.2.2 Polynôme de degré 2

Soit le polynôme en x de degré 2 avec $a_2 \neq 0$:

$$p_2(x) = a_0 x^2 + a_1 x + a_2$$

possède deux racines à

$$r_{1,2} = \frac{-a_1 \pm \sqrt{a_1^2 - 4a_0 a_2}}{2a_0}$$

Le script 3 contient la fonction `RacinePoly2` et son utilisation. La figure 2.1 montre le graphique résultant du script 3. Dans ce script : les lignes 4-10 définissent la fonction `RacinePoly2` avec `a` comme variable d'entrée indépendante et le `return` contient les racines `r` ; la ligne 12 crée le vecteur `a` contenant les coefficients du polynôme, soit $[3, 2, -8]$; les lignes 13-14 appellent la fonction `RacinePoly2` et affichent la valeur des 2 racines ; la ligne 16 crée le vecteur `x` contenant 21 valeurs de -3 à +3 ; la ligne 17 crée le vecteur `y` de même dimension que `x` et le remplit de zéros ; les lignes 18-18 bouclent pour calculer chacune des valeurs de `y` ; et finalement, les dernières lignes créer un graphique du polynôme avec ses racines (cercles rouges).

Le script 4 effectue les mêmes calculs que le script 3 en utilisant les fonctions `roots` (calcul des racines d'un polynôme de coefficients a) et `polyval` (évalue le polynôme de coefficient a aux valeurs de x). Un polynôme est entièrement déterminé par ses coefficients. Ainsi, un polynôme de degré n nécessite $n+1$ coefficients.

SCRIPT 3
`RacinePoly2` : Racines d'un polynôme de degré 2

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

def RacinePoly2(a):  # Racines d'un polynome de degre 2
    if a[0]==0:
        raise ValueError("Ce n'est pas un polynome de degre 2!")
    d = np.sqrt(a[1]**2 - 4*a[2]*a[0])
    r1 = (-a[1]-d)/(2*a[0])
    r2 = (-a[1]+d)/(2*a[0])
    return r1, r2

a = np.array([3, 2, -8])  # Coefficients du polynome de degre 2
r = RacinePoly2(a)        # Calcul les 2 racines
print('Racines en x: %f et %f' % (r[0], r[1]))

x = np.linspace(-3,+3,21) # Calcul 21 valeurs de x de -3 a +3
y = np.zeros(len(x))      # Calcul y = a[0] x**2 + a[1] x + a[2]
for i in range(len(x)):
    y[i] = a[0]*x[i]**2 + a[1]*x[i] + a[2]

plt.figure(1)  # Creer une figure du polynome avec ses racines
plt...
```

Racines en x : -2.000000 et 1.333333

SCRIPT 4
`roots` : Racines d'un polynôme de degré $n = 2$

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

a = np.array([3, 2, -8])  # Coefficients du polynome de degre n=2
r = np.roots(a)           # Calcul les n=2 racines
print('Racines en x: %f et %f' % (r[0], r[1]))

x = np.linspace(-3,+3,21) # Calcul 21 valeurs de x de -3 a +3
y = np.polyval(a,x)       # Calcul y = a[0] x**2 + a[1] x + a[2]

plt.figure(1)  # Creer une figure du polynome avec ses racines
plt...
```

Racines en x : -2.000000 et 1.333333

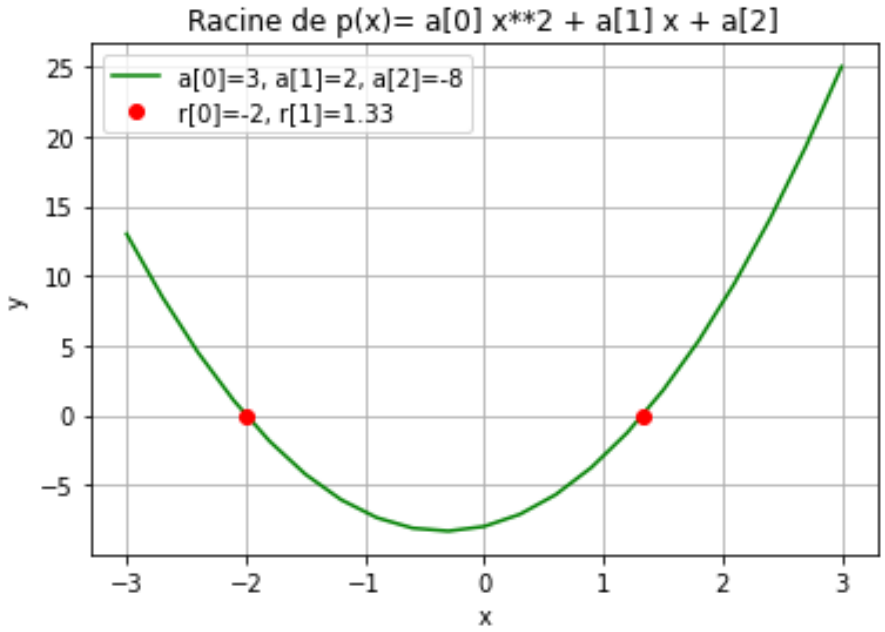

FIGURE 2.1 – Graphique du polynôme de degré 2 et ses racines résultants des scripts 3 ou 4

2.2.3 Intersection de polynômes

Pour certains problèmes d'intersection de polynômes, il est possible de reformuler analytiquement le problème en effectuant algébriquement la soustraction des deux polynômes. Supposons l'intersection d'un premier polynôme de degré 1 (une droite) et un deuxième polynôme de degré 2, c'est-à-dire :

$$p_1(x) = a_0 x + a_1 \qquad p_2(x) = b_0 x^2 + b_1 x + b_2$$

Ainsi, nous avons

$$p(x) = p_2(x) - p_1(x) = b_0 x^2 + (b_1 - a_0)x + (b_2 - a_1)$$

dont les racines sont les solutions des points d'intersections entre $p_1(x)$ et $p_2(x)$.

2.3 Fonctions nonlinéaires

Pour une fonction nonlinéaire, il est souvent impossible de trouver une solution explicite. Il faut alors procéder par itérations successives afin de faire converger une valeur approximative vers d'une racine en évaluant la fonction selon une stratégie d'optimisation. Nous allons présenter la méthode de la bissection et montrer l'utilisation d'un outil d'optimisation de la librairie `scipy`.

2.3.1 Méthode de la bissection

La méthode de la bissection est basée sur le théorème du point milieu qui prédit la présence d'une racine dans un intervalle s'il y a un changement de signe de la fonction évaluée aux bornes de cet intervalle, tel que montré à la figure 2.2.

Théorême 2.3.1 (Point milieu) *Si $f(x)$ est continue entre a et b, et que $sign(f(a)) \neq sign(f(b))$, alors il existe un c tel que $a < c < b$ avec $f(c) = 0$.*

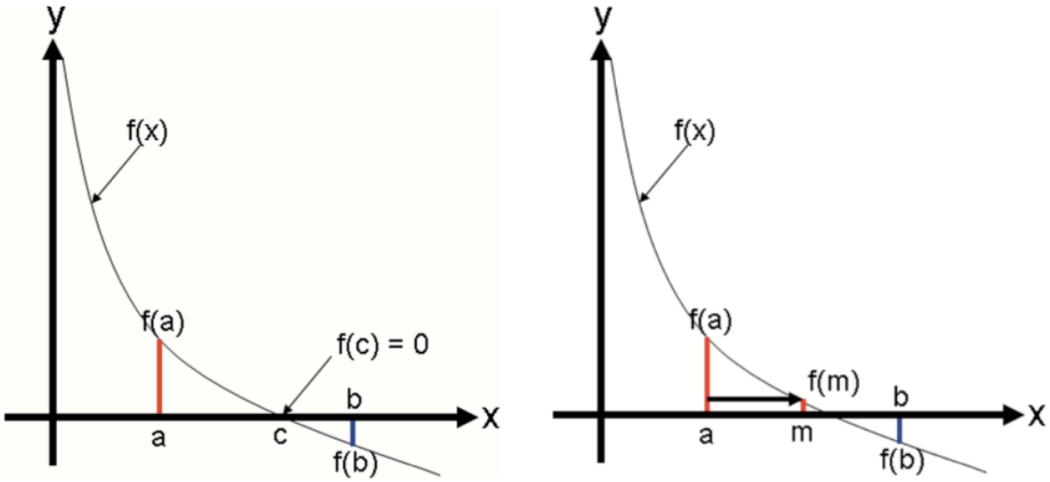

FIGURE 2.2 – Théorême du point milieu et méthode de la bissection

Le principe de base de la méthode de la bissection est d'évaluer la fonction $f(x)$ aux bornes a et b, ainsi qu'au point milieu de l'intervalle (a, b), soit à $m = (a + b)/2$. Si $f(m) = 0$ ou très près de zéro, alors m est la racine c recherchée. Si $f(a) * f(m) > 0$ alors $f(a)$ et $f(m)$ sont de même signe, m est une amélioration de la borne de gauche a et la racine c est située dans l'intervalle (m, b), comme montré à la figure 2.2 de droite. Si $f(a) * f(m) < 0$ alors $f(a)$ et $f(m)$ sont de signe opposés, m est une amélioration de la borne de droite b et la racine c est située dans l'intervalle (a, m). Il est facile de valider cette logique avec une fonction $f(x)$ ascendante où $f(a) < f(b)$.

Pour cet algorithme, il est pertinent d'utiliser une boucle itérative `while` et une structure de contrôle `if`. La boucle `while` permet de vérifier si une évaluation supplémentaire est nécessaire, alors que la structure de contrôle `if` permet d'évaluer dans quel sous-intervalle se trouve le changement de signe.

Le script 5 contient la fonction `bissection` et son utilisation. Dans ce script : les lignes 4-17 définissent la fonction `bissection`, les lignes 19-20 définissent la fonction `f`, alors que les lignes 22-23 demandent l'exécution de la fonction `bissection` et affichent son résultat. Les lignes 11 et 17 expriment la racine retournée. La ligne 4 définit les variables d'entrées indépendantes, soient : 1) le nom de la fonction `f` ; 2) la borne de gauche `a` ; 3) la borne de droite `b` ; 4) la précision requise pour l'arrêt des itérations (par défaut `tol=1e-6` ou 1×10^{-6}) ; et finalement, 5) le nombre maximum d'itérations (par défaut `maxiter=100`). Les lignes 5-6 permettent de quitter la fonction `bissection` s'il n'existe pas de racine dans l'interval `(a, b)`, c'est-à-dire lorsque `f(a)` et `f(b)` sont de même signe. La ligne 7 initialise le compteur d'itération `i=0`. La boucle `while` de la ligne 8 à 16 permet de répéter les itérations de calcul tant que le demi-interval `(b-a)/2` est plus grand que la précision désirée `tol` et que le nombre d'itération `i` est plus petit que le maximum `maxiter`. La ligne 9 calcule la valeur du point milieu `m`. La structure de contrôle `if` des lignes 10-15 permet d'exécuter 3 cas : si `f(m)==0` la ligne 11 retourne la racine `m` ; si `f(a)*f(m)>0` il n'y a pas de changement de signe, la ligne 13 doit alors déplacer la borne `a` vers `m` avec `a=m` ; sinon il y a un changement de signe, la ligne 15 doit alors déplacer la borne `b` vers `m` avec et `b=m`.

Le script 6 effectue des calculs similaires et donne le même résultat que le script 5, mais en utilisant la fonction `optimize.fsolve` pour le calcul d'une racine de la fonction nonlinéaire `f(x)` autour de la valeur initiale 2. La fonction `optimize.fsolve` est disponible dans le module `optimize` de la librairie `scipy`.

SCRIPT 5
`bissection` : Cherche une racine de $f(x)$ entre $a = 0$ et $b = 3$

```
import numpy as np

# Methode de la bissection
def bissection(f, a, b, tol=1e-6, maxiter=100):
    if f(a)*f(b)>=0:
        raise ValueError("La fonction est le meme signe aux bornes a et b!")
    i = 0
    while (b-a)/2 > tol and i < maxiter:
        m = (b + a) / 2   # Calcule le point milieu
        if f(m) == 0:
            return m
        elif f(a)*f(m) > 0: # Si vrai, il n'y a pas de changement de signe
            a = m          # Alors on deplace la borne de gauche
        else:              # Si faux, il y a un changement de signe
            b = m          # Alors on deplace la borne de droit
        i += 1
    return m               # Retourne la racine m

def f(x):                  # Fonction f(x) = x**2 - 2
    return x**2-2

r = bissection(f,0,3)      # Cherche la racine de f entre 0 et 3
print('Racine de f(x): r = %f, f(r) = %f' % (r, f(r)))
```

Racine de f(x) : r = 1.414214, f(r) = 0.000000

SCRIPT 6
`optimize.fsolve` : Cherche une racine de $f(x)$ autour de $x = 2$

```
import numpy as np
from scipy import optimize

def f(x):                  # Fonction f(x) = x**2 - 2
    return x**2-2

r = optimize.fsolve(f,2) # Cherche la racine de f autour de 2
print('Racine de f(x): r = %f, f(r) = %f' % (r, f(r)))
```

Racine de f(x) : r = 1.414214, f(r)= 0.000000

2.3.2 Intersection de fonctions nonlinéaires

L'intersection de fonctions nonlinéaires peut aussi être reformuler en un problème de racine de la différence entre les deux fonctions. Supposons l'intersection des fonctions $f(x) = x^2 - 2$ et $g(x) = 3\cos(0.9\pi x)$. Ainsi, nous avons $h(x) = f(x) - g(x)$, dont les racines sont les points d'intersections. Le script 7 définit des fonctions `f`, `g` et `h` et utilise `optimize.fsolve` pour calculer les racines de $h(x)$ autour de $x = 1$, $x = 2$ et $x = 2.5$. La figure 2.3 montre les intersections entre $f(x)$ et $g(x)$ (points noirs), ainsi les racines de $h(x)$ (points verts).

Script 7
`optimize.fsolve` : Intersection entre $f(x)$ et $g(x)$

```
import numpy as np
from scipy import optimize
def f(x):                  # Fonction f(x) = x**2 - 2
    return x**2-2
def g(x):                  # Fonction g(x) = 3*np.cos(0.9*np.pi*x)
    return 3*np.cos(0.9*np.pi*x)
def h(x):                  # Fonction h(x) = f(x) - g(x)
    return f(x)-g(x)

r1 = optimize.fsolve(h,1)    # Cherche la racine autour de x=1
r2 = optimize.fsolve(h,2)    # Cherche la racine autour de x=2
r3 = optimize.fsolve(h,2.5)  # Cherche la racine autour de x=2.5
print('Racines de h(x): r1 = %f, r2 = %f, r3 = %f' % (r1, r2, r3))
```

Racines de h(x) : r1 = 0.735241, r2 = 1.833112, r3 = 2.235589

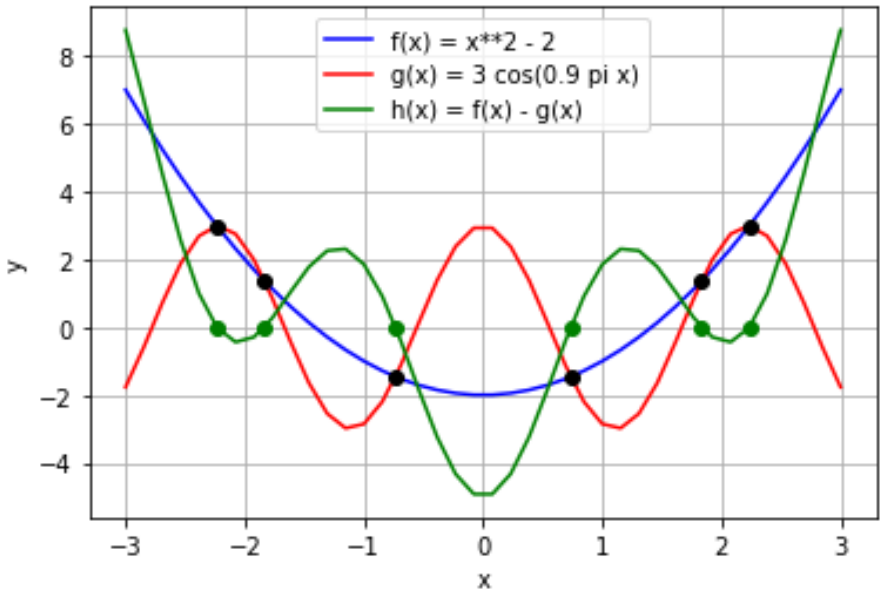

Figure 2.3 – Points noirs d'intersection entre $f(x)$ (ligne bleu) et $g(x)$ (ligne rouge)

Chapitre 3

Calculer une dérivée

3.1 Introduction

Lorsque la fonction $f(x)$ est connue, il est facile d'effectuer symboliquement sa dérivée, puis de l'utiliser dans nos calculs. Cependant, il est fréquent que cette l'expression ne soit pas connue. Dans ce cas, l'ingénieur peut disposer d'observations numériques, telles qu'un ensemble de n pairs de données d'entrées-sorties $\{x_i, y_i\}_0^{n-1}$. Les différences entre les points permettent d'approximer les dérivées du système dynamique observé.

3.2 Dérivée première

La dérivée première $f'(x)$ de $f(x)$ au point $x = a$ est défini comme :

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

c'est-à-dire la pente à $x = a$. Avec une grille numérique à pas $h = x_{i+1} - x_i$ constant, la méthode des différences finies permet d'approximer la pente $f'(x_i)$ avec 2 points dans le voisinage de x_i selon trois formules différentes (voir figure 3.1) :

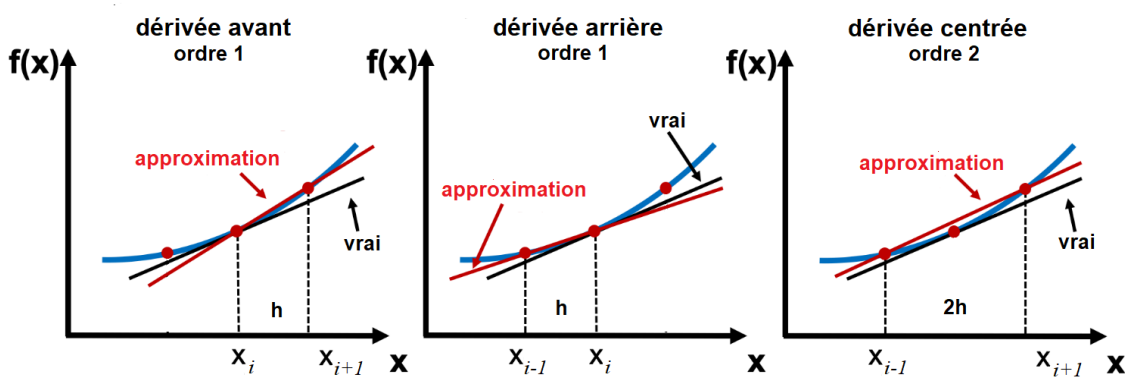

FIGURE 3.1 – Dérivée première avant, arrière et centrée

1. dérivée première avant (ordre 1)

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{h}$$

2. dérivée première arrière (ordre 1)

$$f'(x_i) = \frac{f(x_i) - f(x_{i-1})}{h}$$

3. dérivée première centrée (ordre 2)

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1})}{2h}$$

3.3 Dérivée seconde

Alors que la dérivée première décrit la pente d'une courbe, la dérivée seconde décrit la concavité de cette courbe. Pour un ensemble de données de la position, la dérivée première décrit la vitesse et la dérivée seconde l'accélération. Avec une grille numérique à pas $h = x_{i+1} - x_i$ constant, la méthode des différences finies permet d'approximer $f''(x_i)$ avec 3 points dans le voisinage de x_i selon trois formules différentes :

1. dérivée seconde avant (ordre 1)

$$f''(x_i) = \frac{f(x_i) - 2f(x_{i+1}) + f(x_{i+2})}{h^2}$$

2. dérivée seconde arrière (ordre 1)

$$f''(x_i) = \frac{f(x_i) - 2f(x_{i-1}) + f(x_{i-2})}{h^2}$$

3. dérivée seconde centrée (ordre 2)

$$f''(x_i) = \frac{f(x_{i+1}) - 2f(x_i) + f(x_{i-1})}{h^2}$$

3.4 Fonction `MaDerivee`

Selon les données numériques disponibles, nous allons construire une fonction Python afin de calculer la dérivée première d'un ensemble de données en utilisant la formule de dérivée première centrée d'ordre 2 pour tous les points, sauf le premier et le dernier point. Pour ces derniers, nous utiliserons plutôt les formules de dérivée première avant et arrière selon la disponibilité des données.

Le script 8 définit la fonction `MaDerivee` pour calculer la dérivée première. Dans ce script : les lignes 4-14 définissent la fonction `MaDerivee` ; la ligne 16 définit les coefficients du polynôme $f(x)$; la ligne 17 créer une grille x uniforme de -3 à 3 avec 15 points ; la ligne 18 calcule le pas h de la grille ; la ligne 19 évalue $f(x)$ aux valeurs de x pour le graphique ; la ligne 20 demande l'exécution de la fonction `MaDerivee` avec le pas h ; la ligne 21 définit les coefficients de la dérivée exacte de $f(x)$, soit $f'(x)$; et finalement, la ligne 22 évalue $f'(x)$ pour le graphique. Dans la fonction `MaDerivee` : les lignes 10-11 permettent d'utiliser le formule de dérivée centrée pour tous les points, sauf le premier $i = 0$ et le dernier $i = n - 1$; les lignes 8-9 permettent d'utiliser la formule de dérivée avant pour le premier point x_0 ; et finalement, les lignes 12-13 permettent d'utiliser la formule de dérivée arrière pour le dernier point x_{n-1}.

La figure 3.2 montre en noir $f(x)$, en rouge la dérivée exacte $f'(x)$, en bleu l'approximation numérique de `MaDerivee`, et finalement, en jaune tireté l'approximation numérique avec la formule de dérivée avant qui n'est que d'ordre 1 de précision.

SCRIPT 8
`MaDerivee` : Dérivée numérique de $f(x) = -2x^3 + 4x^2 + 8x - 2$

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

def MaDerivee(y, h=1): # derivee premiere
    n = len(y)
    dy = np.zeros(n)
    for i in range(n):
        if i==0:       # derivee premiere avant
            dy[i] = (y[i+1] - y[i])/h
        elif i==n-1:   # derivee premiere arriere
            dy[i] = (y[i]  - y[i-1])/h
        else:          # derivee premiere centree
            dy[i] = (y[i+1] - y[i-1])/(2*h)
    return dy

a = np.array([-2, 4, 8, -2]) # Coefficients du polynome
x = np.linspace(-3,+3,15)    # Calcul 15 points de x de -3 a +3
h = x[1]-x[0]                # Pas h de la grille
y = np.polyval(a,x)
dy = MaDerivee(y,h)
b = np.array([-6, 8, 8])     # Coefficients de la derivee exacte
z = np.polyval(b,x)
...
```

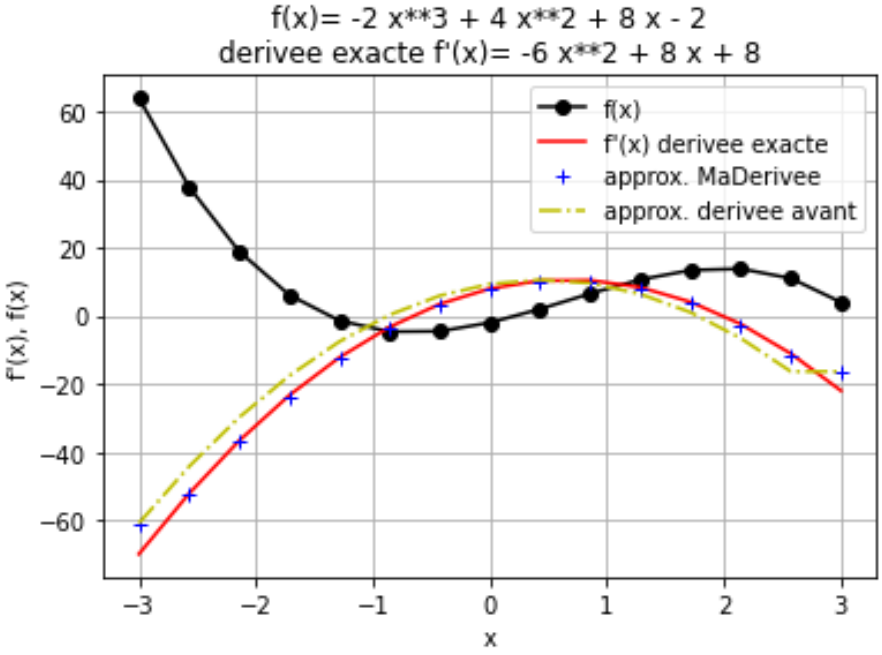

FIGURE 3.2 – Dérivée exacte (rouge) de $f(x)$ (noir), approximation `MaDerivee` (bleu) et approximation dérivée avant (jaune)

Chapitre 4

Calculer une intégrale

4.1 Introduction

Comme pour la dérivation numérique, l'intégration numérique est une technique importante pour l'ingénieur. Il s'agit de calculer l'aire de $f(x)$ sur l'intervalle a et b en utilisant uniquement les n valeurs de $f(x)$ évaluées sur la grille $\{x_i\}_0^{n-1}$, où $x_0 = a$ et $x_{n-1} = b$ avec un pas $h = x_{i+1} - x_i$ constant. L'intervalle $[x_i, x_{i+1}]$ est appelé un sous-interval.

4.2 Méthode de Riemann

Tel que montré à la figure 4.1, l'aire de chaque sous-intervalle de i à $i+1$ peut être approximée par des rectangles de largeur h et de différente hauteur selon les trois cas de la méthode de Riemann :

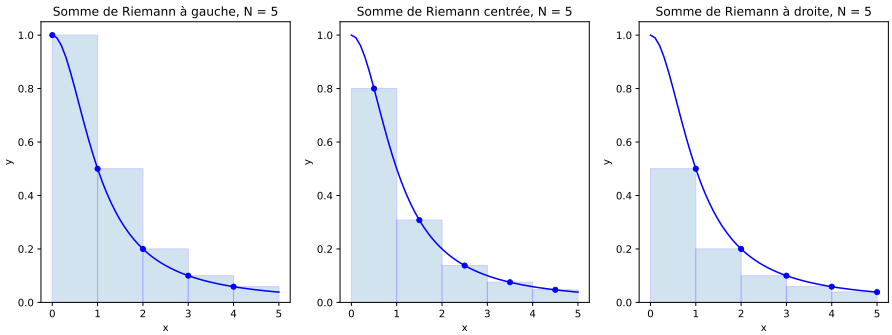

FIGURE 4.1 – Différences entre les trois méthodes de sommes de Riemann

1. Reimann à gauche
$$\int_a^b f(x)dx \approx h \sum_{i=0}^{n-2} f(x_i)$$
2. Riemann centrée
$$\int_a^b f(x)dx \approx h \sum_{i=0}^{n-2} f((x_i + x_{i+1})/2)$$
3. Riemann à droite
$$\int_a^b f(x)dx \approx h \sum_{i=1}^{n-1} f(x_i)$$

Le script 9 calcul les trois sommes de Riemann (gauche, droite et centrée) afin d'estimer l'intégrale de $sin(x)$ de 0 à $\pi/2$. Selon le nombre de points d'approximation, le résultat doit s'approcher de la valeur exacte de 1.0.

SCRIPT 9
Méthode de Riemann : Intégrale de $sin(x)$ de 0 à $\pi/2$ doit donner 1.0

```
import numpy as np

a = 0                  # Borne (a,b) de l'integral
b = np.pi/2
n = 17                 # Nombre de points
h = (b-a)/(n-1)        # Pas d'integration
x = np.linspace(a,b,n) # Calcul la grille x
y = np.sin(x)          # Calcul la fonction sin(x)

RG = h*sum(y[0:n-1])  # Riemann gauche
RD = h*sum(y[1:n])     # Riemann droit

xc = x + h/2           # Decale la grille de h/2
xc = xc[0:n-1]         # Enleve le dernier point
yc = np.sin(xc)        # calcul la grille centree
RC = h*sum(yc[0:n-1])  # Riemann centree
print('Integral de f(x): RG = %f, RC = %f, RD = %f' % (RG, RC, RD))
```

Integral de f(x) : RG = 0.950109, RC = 1.000402, RD = 1.048284

4.3 Méthode des trapèzes

La méthode des trapèzes composés ressemble beaucoup à la somme de Riemann centrée. La différence étant que la forme choisie est le trapèze au lieu du rectangle. La formule de base de la méthode des trapèzes est

$$\int_a^b f(x)\mathrm{d}x \approx h\sum_{i=0}^{n-2}\frac{f(x_i)+f(x_{i+1})}{2} = \frac{h}{2}\left[f(x_0)+2\left(\sum_{i=1}^{n-2}f(x_i)\right)+f(x_{n-1})\right]$$

Contrairement aux sommes de Riemann, cette méthode ne contient pas de variation directionnelle, car les informations des 2 points sont utilisées. L'application de cette méthode est donc très rapide. Le script 10 montre un exemple d'intégrale numérique par la méthode des trapèzes. Il montre également l'utilisation de la fonction d'intégration numérique `np.trapz` de `numpy`.

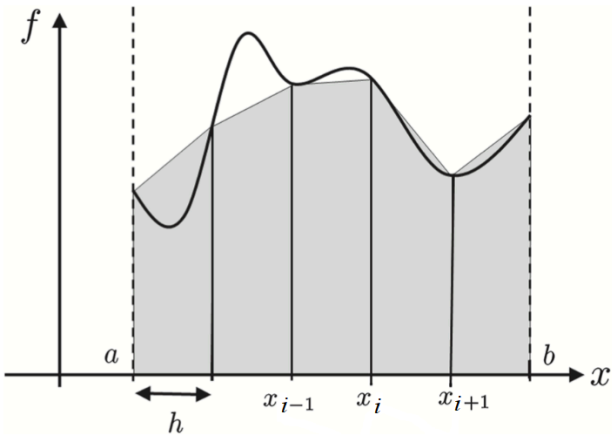

FIGURE 4.2 – Intégrale numérique par la Méthode des trapèzes

SCRIPT 10
Intégrale numérique par la méthode des trapèzes

```
import numpy as np

a = 0                  # Borne (a,b) de l'integral
b = np.pi/2
n = 9                  # Nombre de sous-interval
h = (b-a)/(n-1)        # Pas d'integration
x = np.linspace(a,b,n) # Calcul la grille x
y = np.sin(x)          # Calcul la fonction sin(x)

T1 = (h/2)*(y[0]+ 2*sum(y[1:n-1]) + y[n-1]) # Methode trapeze
T2 = np.trapz(y,dx=h)                       # Methode trapz() de numpy
print('Integrale de f(x): Exacte = %f, Trapeze = %f, trapz = %f' % (1.0, T1, T2))
```

Integrale de f(x) : Exacte = 1.000000, Trapeze = 0.996785, trapz = 0.996785