Rapport de Projet

Elaboré par

KCHAOU Amira

Projet du cours "Analyse Numérique"

Responsable :

Julien Ah-Pine

Réalisé au sein de

SIGMA Clermont

Mastère Spécialisé Expert en Science des Données

Année Universitaire 2024 - 2025

Table des matières

Table des figures

Introduction générale

Ce dossier entre dans le cadre des modalités de contrôle des connaissances du cours d'Analyse Numérique. L'objectif est de présenter et d'étudier plusieurs algorithmes numériques essentiels, en expliquant leur principe, leur objectif, et en les illustrant par des implémentations en Python.

Dans ce rapport, nous explorons divers algorithmes utilisés pour résoudre des problèmes mathématiques courants tels que la recherche de racines, la résolution de systèmes d'équations linéaires, l'ajustement de courbes, et l'intégration numérique. Chaque section présente un algorithme, explique son fonctionnement, et fournit un exemple pratique en Python pour illustrer son application.

Les algorithmes étudiés dans ce dossier sont les suivants :

- **Méthode de Jacobi** : utilisée pour résoudre des systèmes d'équations linéaires.
- **Méthode de la puissance itérée** : permet de trouver les valeurs propres dominantes d'une matrice.
- **Régression polynomiale)** : pour ajuster des données par des polynômes, en utilisant la méthode de Jacobi pour résoudre le système d'équations associé.
- **Interpolation par spline cubique** : pour créer une courbe lisse passant par un ensemble de points.
- **Méthode de dichotomie** : utilisée pour trouver les racines d'une fonction en divisant récursivement un intervalle.
- **Méthode de la fausse position modifiée** : une variante améliorée de la méthode de la fausse position pour la recherche de racines.

— **Méthode de Newton** : une méthode itérative pour la recherche de racines, utilisant l'approximation linéaire de la fonction.

— **Méthode du trapèze composite** : pour l'intégration numérique en divisant un intervalle en segments de taille égale.

— **Méthode de Simpson composite** : une méthode d'intégration numérique plus précise en utilisant un polynôme de degré 2 sur chaque segment.

— **Intégration de Romberg** : combine la méthode de trapèze composite avec l'extrapolation de Richardson pour améliorer la précision de l'intégration.

Chacun de ces algorithmes est expliqué et illustré dans les sections qui suivent. Nous détaillons leur fonctionnement théorique, les conditions de leur utilisation, et leurs implémentations pratiques.

Chapitre 1

Les algorithmes étudiés

1.1 Algorithme 1 : Méthode de Jacobi

1.1.1 Objet de l'algorithme

La méthode de Jacobi permet de résoudre des systèmes linéaires $Ax = b$ où $A \in R^{nn}$. L'idée centrale est de décomposer la matrice A en trois matrices structurées : une matrice triangulaire inférieure L, une matrice diagonale D, et une matrice triangulaire supérieure U de sorte que

$$A = L + D + U$$

En se concentrant sur la matrice diagonale D, définie comme suit pour $A = (a_{ij})$:

$$D = \text{diag}(a_{11}, a_{22}, \ldots, a_{nn}) \quad \text{où} \quad d_{ij} = \begin{cases} a_{ii} & \text{si } i = j \\ 0 & \text{si } i \neq j \end{cases}$$

La méthode de Jacobi utilise l'inverse de la matrice diagonale D comme approximation de l'inverse de A :

$$C = D^{-1} = \text{diag}\left(\frac{1}{a_{11}}, \frac{1}{a_{22}}, \ldots, \frac{1}{a_{nn}}\right)$$

Une itération de la méthode de Jacobi est définie par :

$$x^{(k+1)} = x^{(k)} + D^{-1}(b - Ax^{(k)})$$

La convergence de cette méthode est assurée si la matrice A est à diagonale strictement dominante, ce qui signifie :

$$|a_{ii}| > \sum_{j=1, j\neq i}^{n} |a_{ij}|, \quad \forall i = 1, \ldots, n$$

Dans ce cas, on peut montrer que la norme matricielle infinie de $\|I_n - D^{-1}A\|_\infty < 1$, garantissant ainsi la convergence de la méthode de Jacobi.

1.1.2 Exemples pratiques d'application

La méthode de Jacobi est utilisée dans divers contextes pratiques :

— **Résolution de grands systèmes d'équations linéaires** : La méthode de Jacobi est particulièrement utile pour résoudre des systèmes de grande taille où les méthodes directes sont coûteuses en calcul. Elle permet d'obtenir une solution approximative par des itérations successives.

— **Calcul scientifique et ingénierie** : Dans des domaines tels que le calcul scientifique et l'ingénierie, la méthode de Jacobi est employée pour obtenir des solutions approchées dans des problèmes de modélisation où des systèmes linéaires apparaissent fréquemment.

— **Modélisation numérique et simulations** : La méthode est souvent utilisée en dynamique des fluides, en électromagnétisme et dans les méthodes de différences finies, où elle permet d'approcher des solutions dans des simulations numériques de phénomènes physiques.

Ces exemples montrent l'importance de la méthode de Jacobi pour résoudre

des systèmes d'équations linéaires, en particulier dans les cas où une solution exacte n'est pas nécessaire, mais où une convergence vers une solution approchée est suffisante.

1.1.3 Pseudo-code de l'algorithme

Nous rappelons ci-dessous le pseudo-code de l'algorithme de Jacobi.

Algorithm: Jacobi

Input: $\mathbf{A} \in \mathbb{R}^{n \times n}$, $\mathbf{b} \in \mathbb{R}^{n}$, $\mathbf{x}^{(0)} = \mathbf{0}_n$, $\epsilon = 0.01$, $K = 10$
Output: $\mathbf{x}^* \in \mathbb{R}^{n \times 1}$ (approximation)

$k \leftarrow 0$
$\mathbf{D}^{-1} = \text{Diag}(\{1/a_{ii}\}_{i=1,\ldots,n})$
repeat
    $k \leftarrow k + 1$
    $\mathbf{x}^k = \mathbf{x}^{k-1} + \mathbf{D}^{-1}\left(\mathbf{b} - \mathbf{A}\mathbf{x}^{k-1}\right)$
until $\|\mathbf{x}^k - \mathbf{x}^{k-1}\| < \epsilon$ or $k < K$ (Distance Euclidienne -voir slide 105-)
return $\mathbf{x}_k$

Figure 1.1 : Pseudo-code de l'algorithme de Jacobi

1.1.4 Code Python de l'algorithme

Voici ci-dessous le code Python de notre implémentation de l'algorithme présenté dans le paragraphe précédent.

```python
def Jacobi(A, b, epsilon=0.01, K=10):

    # Initialisation
    n = A.shape[0]
    x0 = np.zeros(n)
    D_inv = np.diag(1 / np.diag(A))
    k = 0

    # Boucle itérative
    while k < K:
        k = k + 1
        x_new = x0 + D_inv @ (b - A @ x0)

        # Condition d'arrêt
        if (np.linalg.norm(x_new - x0) < epsilon):
            return x_new

        x0 = x_new  # Mise à jour de x^(k)

    return x0  # Retourne x_k si le nombre max d'itérations est atteint
```

Figure 1.2 : Code python de l'algorithme de Jacobi

1.1.5 Illustration sur un exemple

Nous illustrons le bon fonctionnement de l'algorithme implémenté en l'exécutant sur l'exemple suivant :

$$\mathbf{A} = \begin{pmatrix} 2 & 1 \\ 3 & 5 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 3 \\ 8 \end{pmatrix}$$

L'exécution du code Python présenté dans la sous-section précédente donne le résultat suivant :

```
# Exemple
A = np.array([[2, 1], [3, 5]], dtype=float)
b = np.array([3, 8], dtype=float)

solution = Jacobi(A, b, epsilon=0.01, K=10)

print("Solution approchée :", solution)

✓ 0.0s
Solution approchée : [0.99757 0.99757]
```

Figure 1.3 : Exemple de l'algorithme de Jacobi

La solution approchée est : [0.99757 0.99757]

1.2 Algorithme 2 : Méthode de la puissance itérée

1.2.1 Objet de l'algorithme

— Supposons que $\mathbf{A} \in \mathcal{M}_n(R)$ ait n valeurs propres distinctes et qu'on les ordonne par ordre décroissant de leur valeur absolue :

$$|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|.$$

— La **méthode de puissance** que nous étudierons vise à déterminer la plus grande valeur propre λ_1 et le vecteur propre $\mathbf{v}_1$ associé.

— Notons en préambule les relations suivantes :

— Par définition du couple $(\lambda_1, \mathbf{v}_1)$, nous avons :

$$\mathbf{v}_1 \neq \mathbf{0}_n \text{ et } \mathbf{A}\mathbf{v}_1 = \lambda_1 \mathbf{v}_1.$$

— Nous pouvons prémultiplier l'équation par $\mathbf{v}_1^\top$ et il vient :

$$\mathbf{v}_1^\top \mathbf{A}\mathbf{v}_1 = \lambda_1 \mathbf{v}_1^\top \mathbf{v}_1 \Rightarrow \lambda_1 = \frac{\mathbf{v}_1^\top \mathbf{A}\mathbf{v}_1}{\mathbf{v}_1^\top \mathbf{v}_1}.$$

— Remarque : cette dernière expression est valable pour tout couple $(\lambda_i, \mathbf{v}_i)$, $i = 1, \ldots, n$.

1.2.2 Exemples pratiques d'application

La méthode de la puissance itérée est utilisée dans divers domaines, notamment :

— **Analyse de réseaux** : Pour déterminer l'importance des nœuds dans les graphes, par exemple dans l'algorithme PageRank de Google.

— **Mécanique et physique** : Pour trouver le mode de vibration fondamental dans les systèmes vibratoires.

— **Traitement d'images et apprentissage automatique** : Pour réduire la dimensionnalité en trouvant les directions principales dans les données.

1.2.3 Pseudo-code de l'algorithme

Nous rappelons ci-dessous le pseudo-code de l'algorithme de la puissance itérée.

Algorithm: PowerMethod

Input: $\mathbf{A} \in \mathbb{R}^{n \times n}$, $\mathbf{x}^{(0)} = \text{None}$, $\epsilon = 0.01$, $K = 10$
Output: $(\lambda_1, \mathbf{v}_1)$ (approximation)

```
if x^(0) is None then
    Générer aléatoirement un vecteur x^(0) de taille n
v^(0) ← x^(0) / ||x^(0)||
μ^(0) = 0
for k = 1, ..., K do
    x^(k) ← A v^(k−1)
    v^(k) ← x^(k) / ||x^(k)||
    μ^(k) = [v^(k)]^⊤ A v^(k)
    if |μ^(k) − μ^(k−1)| / |μ^(k)| < ε then
        return (μ^(k), v^(k))
return (μ^(k), v^(k))
```

Figure 1.4 : Pseudo-code de l'algorithme de la puissance itérée

1.2.4 Code Python de l'algorithme

Voici ci-dessous le code Python de notre implémentation de l'algorithme présenté dans le paragraphe précédent.

```python
def PowerMethod(A, x0=None, epsilon=0.01, K=10):
    n = A.shape[0]
    if x0 is None:
        x0 = np.random.rand(n)

    #on va normaliser x0
    v = x0 / np.linalg.norm(x0)

    mu_old = 0

    for i in range(K):
        x= np.dot(A,v) #on multiplie A par v

        v= x / np.linalg.norm(x) #on normalise x

        mu = np.dot(v.T, np.dot(A,v))

        if abs((mu - mu_old) / mu ) < epsilon:  #on verifie le critere de convergence
            return mu,v

        mu_old=mu # mettre à jour mu(k-1)

    return mu,v
```

Figure 1.5 : Code python de l'algorithme de la puissance itérée

1.2.5 Illustration sur un exemple

Nous illustrons le bon fonctionnement de l'algorithme implémenté en l'exécutant sur l'exemple suivant :

$$A = \begin{pmatrix} 2 & 0 & 4 \\ 3 & -4 & 12 \\ 1 & -2 & 5 \end{pmatrix}$$

L'exécution du code Python présenté dans la sous-section précédente donne le résultat suivant :

```
A = np.array([[2, 0, 4],
              [3, -4, 12],
              [1, -2, 5]])

mu, v = PowerMethod(A)

print("Valeur propre approximée:", mu)
print("Vecteur propre approximé:", v)
```

✓ 0.0s

```
Valeur propre approximée: 2.0188179642486905
Vecteur propre approximé: [0.89014018 0.45565605 0.00529298]
```

Figure 1.6 : Exemple de l'algorithme de la puissance itérée

La valeur propre approximée est : 2.0188179642486905

Le vecteur propre approximé est : [0.89014018 0.45565605 0.00529298]

1.3 Algorithme 3 : La régression polynomiale

1.3.1 Objet de l'algorithme

— **Définition du problème :** La régression polynomiale est une méthode qui permet de modéliser une relation entre les variables indépendantes et dépendantes par un polynôme de degré $m > 1$. Cela permet de capturer des relations non linéaires entre x et y.

— **Formulation mathématique :** Soit $f(x)$ le modèle que nous supposons être de la forme :

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_m x^m$$

avec $a = (a_0, a_1, \ldots, a_m) \in R^{m+1}$ représentant le vecteur des coefficients du polynôme.

— **Objectif :** L'objectif de l'algorithme est de minimiser l'erreur quadratique Scr(a) entre les valeurs observées et les valeurs prédites. Le critère de Moindres Carrés Ordinaires (MCO) s'écrit donc :

$$\text{Scr}(a_0, \ldots, a_m) = \sum_{i=1}^{n} (y_i - (a_0 + a_1 x_i + a_2 x_i^2 + \cdots + a_m x_i^m))^2$$

— **Notations :** On utilise les notations suivantes :

— $\varphi_j(x) = x^j, \forall j = 0, \ldots, m$

— $\varphi(x) = (\varphi_0(x), \varphi_1(x), \ldots, \varphi_m(x))$

— $X \in R^{n(m+1)}$ est la matrice des données, où chaque ligne X_i correspond à $(x_i^0, x_i^1, \ldots, x_i^m)$

— **Forme matricielle du problème :** En notation matricielle, le problème de minimisation de Scr(a) s'exprime par :

$$\min_{a \in R^{m+1}} \text{Scr}(a) = \sum_{i=1}^{n} (y_i - X_i a)^2$$

— **Solution du problème :** Les conditions de premier ordre de minimisation (CNPO) nous amènent à résoudre le système suivant pour trouver le vecteur optimal a :

$$X^T X a = X^T y$$

où $X^T X \in R^{(m+1)(m+1)}$ et $X^T y \in R^{(m+1)}$.

1.3.2 Exemples pratiques d'application

La régression polynomiale est utilisée dans divers domaines :

— **Analyse de tendances** : Pour ajuster une courbe aux données temporelles et détecter des tendances non linéaires.

— **Modélisation de phénomènes physiques** : Pour modéliser des relations complexes en physique, chimie ou biologie.

— **Finance et économie** : Pour ajuster des courbes aux données économiques et financières lorsque les relations entre les variables ne sont pas linéaires.

1.3.3 Pseudo-code de l'algorithme

Nous rappelons ci-dessous le pseudo-code de l'algorithme de de la régression polynomiale.

Algorithm: DetermineCoefficient

Input: $\{x_i, y_i\}_{i=0,\dots,n}$, m
Output: $\{a_0, \dots, a_m\}$
1 n ← taille du vecteur **x**
2 **A** ← matrice nulle de taille $(m+1) \times (m+1)$
3 **b** ← vecteur nul de taille $(m+1) \times 1$
4 **for** $j = 0, \dots, m$ **do**
5 **for** $i = 0, \dots, n-1$ **do**
6 $b_j \leftarrow b_j + y_i x_i^j$
7 **for** $k = 0, \dots, m$ **do**
8 $a_{jk} \leftarrow a_{jk} + x_i^j x_i^k$
9 Résoudre **Aa** = **b**
10 **return** $\{a_0, \dots, a_m\}$

Algorithm: EvaluatePolynomial

Input: $\{a_j\}_{j=0,\dots,m}$, $\{x_i\}_{i=0,\dots,n}$
Output: $\{y_i\}_{i=0,\dots,n}$
1 **for** $i = 0, \dots, n$ **do**
2 $y_i \leftarrow 0$
3 **for** $j = 0, \dots, m$ **do**
4 $y_i \leftarrow y_i + a_j x_i^j$
5 **return** $\{y_1, \dots, y_n\}$

Figure 1.7 : Pseudo-code de l'algorithme de la régression polynomiale

1.3.4 Code Python de l'algorithme

Voici ci-dessous le code Python de notre implémentation de l'algorithme présenté dans le paragraphe précédent.

```python
def DetermineCoefficient(x, y, m):
    # Normalisation des données
    x_norm = (x - np.mean(x)) / np.std(x)
    n = len(x_norm)
    A = np.zeros((m + 1, m + 1))
    b = np.zeros(m + 1)

    # Remplissage de la matrice A et du vecteur b
    for j in range(m + 1):
        for i in range(n):
            b[j] += y[i] * (x_norm[i]**j)
            for k in range(m + 1):
                A[j, k] += x_norm[i]**(j + k)

    # Utilisation de la méthode de Jacobi pour résoudre Aa = b
    coeffs =Jacobi(A, b, epsilon=1e-10, K=10000)

    return coeffs
```

Figure 1.8 : Fonction de régression polynomiale avec Jacobi

```python
# Fonction pour évaluer le polynôme
def EvaluatePolynomial(coeffs, x):
    # Normalisation des valeurs de x utilisées pour l'évaluation
    x_norm = (x - np.mean(x)) / np.std(x)
    n = len(x_norm)
    m = len(coeffs) - 1
    y = np.zeros(n)

    for i in range(n):
        for j in range(m + 1):
            y[i] += coeffs[j] * (x_norm[i]**j)

    return y
```

Figure 1.9 : Fonction pour évaluer le polynôme

1.3.5 Illustration sur un exemple

Pour cet exemple, nous utilisons les valeurs suivantes pour x et y :

$$x = \begin{pmatrix} -0.04 & 0.93 & 1.95 & 2.90 & 3.83 & 5.0 & 5.98 & 7.05 & 8.21 & 9.08 \\ 10.09 & & & & & & & & & \end{pmatrix}$$

$$y = \begin{pmatrix} -8.66 & -6.44 & -4.36 & -3.27 & -0.88 & 0.87 & 3.31 & 4.63 & 6.19 & 7.4 \\ 8.85 & & & & & & & & & \end{pmatrix}$$

Le degré du polynôme est défini comme suit :

$$m = 3$$

L'exécution du code Python présenté dans la sous-section précédente donne le résultat suivant :

```python
# Données d'exemple
x = np.array([-0.04, 0.93, 1.95, 2.90, 3.83, 5.0, 5.98, 7.05, 8.21, 9.08, 10.09])
y = np.array([-8.66, -6.44, -4.36, -3.27, -0.88, 0.87, 3.31, 4.63, 6.19, 7.4, 8.85])

m = 3  # Degré du polynôme

# Calcul des coefficients du polynôme de degré 3
coeffs = DetermineCoefficient(x, y, m)
print("Coefficients du polynôme de degré 3 :", coeffs)

# Évaluation du polynôme aux points x
y_eval = EvaluatePolynomial(coeffs, x)

# Tracer les deux nuages de points
plt.figure()

# Nuage de points originaux (x, y)
plt.plot(x, y, 'o', label='Points originaux')

# Nuage de points pour le polynôme de degré 3 (x, p(x))
plt.plot(x, y_eval, 'x', label='Polynôme de degré 3')

# Ajouter des labels et une légende
plt.xlabel('x')
plt.ylabel('y')
plt.legend()

# Afficher le graphique
plt.show()
```

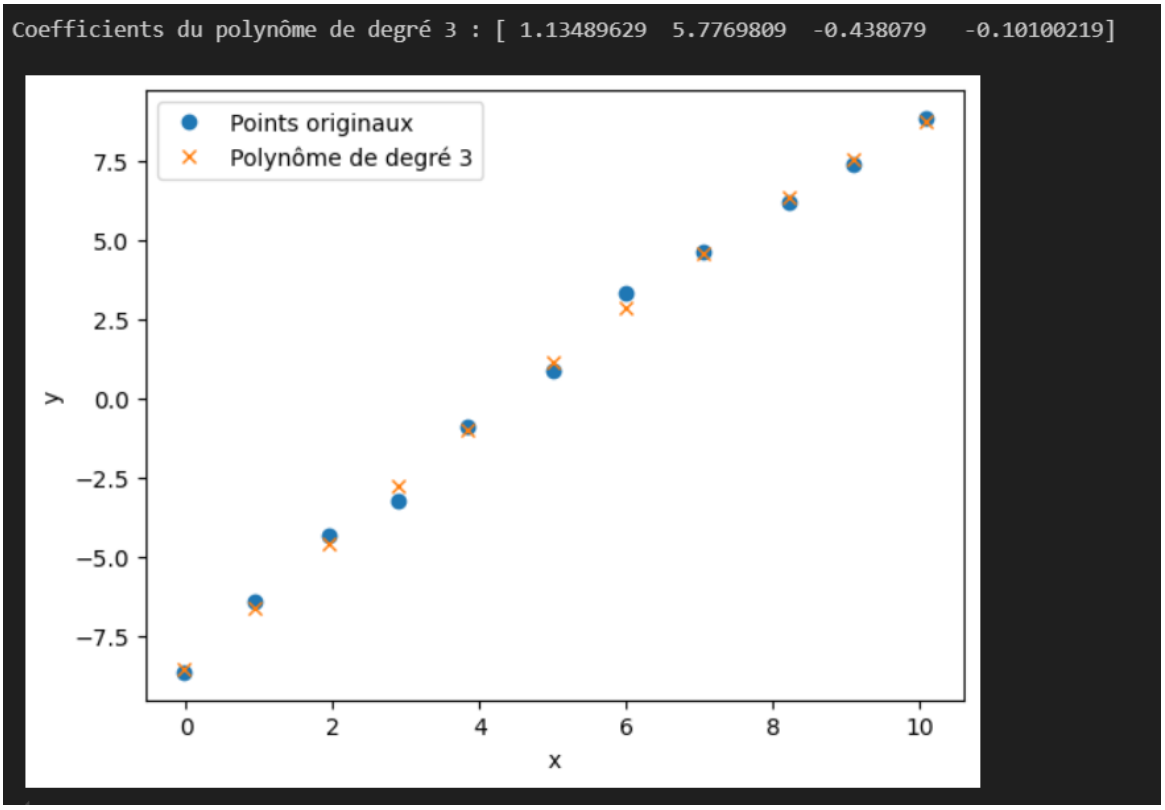

Figure 1.10 : Exemple de l'algorithme de la régression polynomiale

1.4 Algorithme 4 : Interpolation par splines cubiques

1.4.1 Objet de l'algorithme

L'interpolation par splines cubiques est une méthode d'interpolation qui vise à approximer une fonction $y = f(x)$ à partir d'un ensemble de points de données connus $\{(x_i, y_i)\}_{i=0}^{n}$. Contrairement à l'interpolation polynomiale globale, qui utilise un unique polynôme de degré n, l'interpolation par splines cubiques divise l'intervalle $[a, b]$ des données en sous-intervalles et associe à chacun d'eux un polynôme de degré 3. Ces polynômes sont appelés **splines cubiques** et sont définis de manière à garantir la continuité de la fonction interpolée, ainsi que celle de ses premières et deuxièmes dérivées.

Soit un ensemble de $n+1$ points $\{(x_i, y_i)\}_{i=0,\dots,n}$ où l'on souhaite interpoler une fonction. On suppose que les points x_i sont ordonnés tels que $a = x_0 < x_1 <$

$\cdots < x_n = b$. Ainsi, l'intervalle $[a, b]$ peut être subdivisé en n sous-intervalles :

$$[a, b] = [a_0, x_1] \cup [x_1, x_2] \cup \cdots \cup [x_{n-1}, x_n]$$

On définit alors une fonction spline $s(x)$ sur chaque sous-intervalle $[x_i, x_{i+1}]$, avec $i = 0, \ldots, n-1$, telle que :

$$s(x) = p_i(x) \quad \text{si } x \in [x_i, x_{i+1}] \quad \text{et} \quad s(x_n) = y_n \quad \text{si } x = x_n$$

où chaque $p_i(x)$ est un polynôme de degré 3 qui satisfait les conditions suivantes pour garantir la continuité et la régularité de la fonction interpolée : - $s(x_i) = y_i$ pour chaque point $i = 0, \ldots, n$, assurant que la courbe passe par tous les points de données. - Les dérivées premières et secondes des polynômes adjacents sont égales aux points de jonction x_{i+1}, assurant la continuité des pentes et des courbures :

$$p_i(x_{i+1}) = p_{i+1}(x_{i+1}), \quad p_i'(x_{i+1}) = p_{i+1}'(x_{i+1}), \quad p_i''(x_{i+1}) = p_{i+1}''(x_{i+1}).$$

Dans le cas des splines cubiques, chaque $p_i(x)$ est de la forme suivante :

$$p_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3$$

Les coefficients a_i, b_i, c_i, et d_i sont déterminés en résolvant un système d'équations linéaires qui garantit les conditions d'interpolation et de continuité.

L'interpolation par splines cubiques est largement utilisée car elle permet une approximation lisse et évite les oscillations souvent observées dans les interpolations polynomiales de haut degré.

1.4.2 Exemples pratiques d'application

Les splines cubiques sont couramment utilisées dans plusieurs domaines :

— **Graphique et animation** : Pour tracer des courbes lisses dans les logiciels de graphisme et d'animation.

— **Analyse de données** : Pour ajuster des courbes aux données expérimentales en sciences.

— **Physique et ingénierie** : Pour modéliser des trajectoires et des courbes dans des simulations.

— **Biologie et médecine** : Pour modéliser des courbes de croissance et des tendances biomédicales.

— **Finance** : Pour interpoler des courbes de rendement et des taux d'intérêt.

1.4.3 Pseudo-code de l'algorithme

Nous rappelons ci-dessous le pseudo-code de l'algorithme de l'interpolation par splines cubiques.

Algorithm: DeterminePsecond

Input: $\{x_i, y_i\}_{i=0,\dots,n}$
Output: $\{p_i\}_{i=0,\dots,n}$

$p_i \leftarrow 0$
$p_n \leftarrow 0$
Calculer $a_i, i = 1, \dots, n-2$
Calculer $b_i, i = 1, \dots, n-1$
Calculer $c_i, i = 1, \dots, n-1$
Former la matrice tri-diagonale $\mathfrak{A}$
Former le vecteur c
Résoudre pour $p = (p_1, \dots, p_{n-1})$ le sel. $\mathfrak{A}p = c$

Algorithm: ComputeSplineInterpolation

Input: $\{x_i, y_i\}_{i=0,\dots,n}, \{p_i\}_{i=0,\dots,n}, \{x'_{i'}\}_{i'=0,\dots,n'}$
Output: $\{x'_{i'}, y'_{i'}\}_{i'=0,\dots,n'}$

Calculer $h_i, i = 0, \dots, n-1$
Calculer $\alpha_i, i = 0, \dots, n-1$
Calculer $\beta_i, i = 0, \dots, n-1$
for $i' = 0, \dots, n'$ **do**
    for $i = 0, \dots, n-1$ **do**
        if $x'_{i'} \in [x_i, x_{i+1}]$ **then**
            $y'_{i'} = \frac{1}{6h_i}\left(p_{i+1}(x'_{i'} - x_i)^3 + p_i(x_{i+1} - x'_{i'})^3\right) + \alpha_i(x'_{i'} - x_i) + \beta_i$
            break

Figure 1.11 : Pseudo-code de l'algorithme de l'interpolation par splines cubiques

1.4.4 Code Python de l'algorithme

Voici ci-dessous le code Python de notre implémentation de l'algorithme présenté dans le paragraphe précédent.

```python
def DeterminePSecond(x, y):
    n = len(x) - 1  # Nombre de segments
    h = np.diff(x)   # Calcul des intervalles h_i = x_{i+1} - x_i

    # Initialiser les vecteurs alpha, beta, gamma et le vecteur c
    alpha = np.zeros(n-1)
    beta = np.zeros(n-1)
    gamma = np.zeros(n-1)
    c = np.zeros(n-1)

    # Calcul des coefficients alpha, beta, gamma
    for i in range(1, n):
        alpha[i-1] = h[i-1]
        beta[i-1] = 2 * (h[i-1] + h[i])
        gamma[i-1] = h[i]
        c[i-1] = 6 * ((y[i+1] - y[i]) / h[i] - (y[i] - y[i-1]) / h[i-1])

    # Construire la matrice tridiagonale A et résoudre A*p = c
    A = np.zeros((n-1, n-1))
    for i in range(n-1):
        if i > 0:
            A[i, i-1] = alpha[i-1]
        A[i, i] = beta[i-1]
        if i < n-2:
            A[i, i+1] = gamma[i]

    # Résoudre le système Ap = c
    p_internal = np.linalg.solve(A, c)

    # Ajouter les contraintes p_0 = 0 et p_n = 0 (pentes nulles aux extrémités)
    p = np.zeros(n+1)
    p[1:n] = p_internal

    return p
```

Figure 1.12 : Fonction DeterminePSecond

```python
def ComputeSplineInterpolation(x, y, p, x_new):
    n = len(x) - 1  # Nombre de segments
    y_new = np.zeros(len(x_new))

    h = np.diff(x)   # Calcul des intervalles h_i = x_{i+1} - x_i

    # Interpolation pour chaque point dans x_new
    for i_new, xi_prime in enumerate(x_new):
        for i in range(n):
            if x[i] <= xi_prime <= x[i+1]:
                hi = h[i]
                alpha_i = (y[i+1] - y[i]) / hi - (hi * (p[i+1] - p[i])) / 6
                beta_i = y[i]

                y_new[i_new] = (p[i+1] * (xi_prime - x[i])**3) / (6 * hi) \
                               - (p[i] * (xi_prime - x[i+1])**3) / (6 * hi) \
                               + alpha_i * (xi_prime - x[i]) + beta_i
                break

    return y_new
```

Figure 1.13 : Fonction ComputeSplineInterpolation

1.4.5 Illustration sur un exemple

Considérons l'ensemble de points suivants pour l'interpolation par splines cubiques :

$$x = \begin{pmatrix} 0.15 & 2.3 & 3.15 & 4.85 & 6.25 & 7.95 \end{pmatrix}$$

$$y = \begin{pmatrix} 4.79867 & 4.49013 & 4.2243 & 3.47313 & 2.66674 & 1.51909 \end{pmatrix}$$

L'exécution du code Python présenté dans la sous-section précédente donne le résultat suivant :

```
# Données d'exemple
x = np.array([0.15, 2.3, 3.15, 4.85, 6.25, 7.95])
y = np.array([4.79867, 4.49013, 4.2243, 3.47313, 2.66674, 1.51909])

# Calculer les dérivées secondes
p = DeterminePSecond(x, y)

# Points d'interpolation
x_new = np.linspace(min(x), max(x), 100)

# Calculer les valeurs interpolées
y_new = ComputeSplineInterpolation(x, y, p, x_new)

# Tracer les résultats
plt.plot(x, y, 'o', label='Points d\'origine')
plt.plot(x_new, y_new, '-', label='Spline cubique')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```

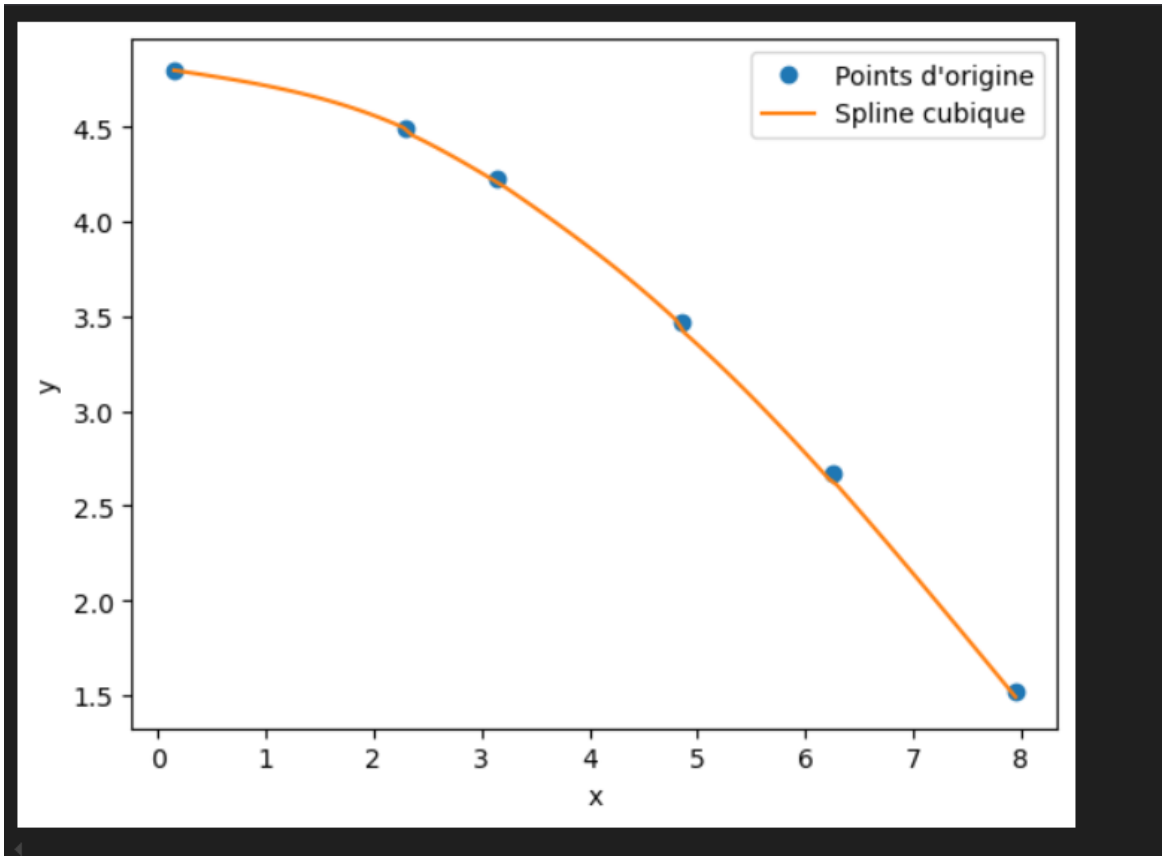

Figure 1.14 : Exemple de l'algorithme l'nterpolation par splines cubiques

1.5 Algorithme 5 : Méthode de dichotomie

1.5.1 Objet de l'algorithme

La méthode de dichotomie consiste à couper en deux l'intervalle $[a_k, b_k]$ en prenant comme point de séparation $c_k = \frac{a_k+b_k}{2}$, le point central de l'intervalle.

On se demande alors lequel des deux sous-intervalles, à droite $[c_k, b_k]$ ou à gauche $[a_k, c_k]$ du point central c_k, faut-il retenir ? On utilise à nouveau la condition qui indique que si les valeurs de f aux extrémités du sous-intervalle considéré sont de signes opposés, alors ce sous-intervalle contient une racine.

On itère ce découpage en deux jusqu'à obtenir une précision suffisamment bonne !

La procédure génère une séquence $c_0, c_1, \ldots, c_k, \ldots$ de points centraux telle que $\lim_{k\to\infty} c_k \to x^*$, où $x^* \in [a_0, b_0]$ est une racine de f.

À l'itération n, la racine x^* étant dans $[a_n, b_n]$ et c_n étant le point au milieu de cet intervalle, on a :

$$|x^* - c_n| \leq \frac{1}{2}(b_n - a_n)$$

Par ailleurs :

$$b_n - a_n = \frac{1}{2}(b_{n-1} - a_{n-1}) = \frac{1}{2^2}(b_{n-2} - a_{n-2}) = \cdots = \frac{1}{2^n}(b_0 - a_0)$$

Ainsi, nous avons :

$$|x^* - c_n| \leq \frac{1}{2^{n+1}}(b_0 - a_0)$$

ce qui montre la convergence de la méthode quand n augmente.

On peut, de plus, déterminer à l'avance le nombre d'itérations n nécessaire afin d'atteindre une précision ε donnée ! En effet :

$$\frac{1}{2^{n+1}}(b_0 - a_0) < \varepsilon \Leftrightarrow n > \log_2\left(\frac{b_0 - a_0}{\varepsilon}\right) - 1$$

Par exemple, si on prend $\varepsilon = 10^{-5}$, alors $n > 15.6$.

1.5.2 Exemples pratiques d'application

La méthode de dichotomie est employée dans plusieurs domaines :

— **Informatique** : Pour rechercher des valeurs dans des structures de données triées.

— **Physique et ingénierie** : Pour résoudre des équations dans des modèles physiques, comme le calcul de points d'équilibre.

— **Finance** : Pour trouver des taux d'intérêt ou de rendement qui équilibrent certaines équations financières.

1.5.3 Pseudo-code de l'algorithme

Nous rappelons ci-dessous le pseudo-code de l'algorithme de la méthode de dichotomie.

Algorithm: BisectionMethod

Input: f, ϵ, a, b tel que $f(a)f(b) < 0$
Output: $\overline{x}$ tel que $|\overline{x} - x^*| < \epsilon$ où $f(x^*) = 0$

$a_0 \leftarrow a$, $b_0 \leftarrow b$, $c_0 \leftarrow (a_0 + b_0)/2$
$n \leftarrow \lceil \ln((b_0 - a_0)/\epsilon) / \ln(2) \rceil$
for $k = 1, \ldots, n$ **do**
    if $f(a_{k-1})f(c_{k-1}) \leq 0$ **then**
        if $f(c_{k-1}) = 0$ **then**
            return $\overline{x} \leftarrow c_{k-1}$
        $a_k \leftarrow a_{k-1}$, $b_k \leftarrow c_{k-1}$
    else
        $a_k \leftarrow c_{k-1}$, $b_k \leftarrow b_{k-1}$
    $c_k \leftarrow (a_k + b_k)/2$
return $\overline{x} \leftarrow c_n$

Figure 1.15 : Pseudo-code de l'algorithme de la méthode de dichotomie

1.5.4 Code Python de l'algorithme

Voici ci-dessous le code Python de notre implémentation de l'algorithme présenté dans le paragraphe précédent.

```python
def BisectionMethod(f, a, b, epsilon):

    # Vérifier que f(a) et f(b) sont de signes opposés
    if f(a) * f(b) >= 0:
        raise ValueError("f(a) et f(b) doivent avoir des signes opposés")


   # Initialisation des variables
    a_k = a
    b_k = b
    c_k = (a_k + b_k) / 2
    n = int(np.ceil(np.log((b - a) / epsilon) / np.log(2)))

    # Boucle pour n itérations
    for k in range(1, n+1):
        # Vérifier la condition d'arrêt
        if f(a_k) * f(c_k) <= 0:
            if f(c_k) == 0:
                return c_k
            b_k = c_k
        else:
            a_k = c_k

        c_k = (a_k + b_k) / 2

    return c_k
```

Figure 1.16 : Code python méthode de dichotomie

1.5.5 Illustration sur un exemple

Soit la fonction définie par :

$$f(x) = x^3 + x - 1$$

Nous cherchons la racine de cette fonction dans l'intervalle $[a, b] = [0, 1]$ avec une précision $\varepsilon = 0.001$.

```python
# Définir la fonction donnée
def f(x):
    return x**3 + x - 1  # Fonction f(x) = x^3 + x - 1

# Intervalle [a, b] et précision epsilon
a = 0
b = 1
epsilon = 0.001

# Appeler la méthode de dichotomie
root = BisectionMethod(f, a, b, epsilon)
print("Racine trouvée :", root)
```

✓ 0.0s

```
Racine trouvée : 0.68212890625
```

Figure 1.17 : Exemple de la méthode de dichotomie

La racine trouvée est : 0.68212890625

1.6 Algorithme 6 : Méthode de la fausse position modifiée

1.6.1 Objet de l'algorithme

Afin de garantir que l'amplitude de l'intervalle contenant la racine tend vers 0, on peut ajouter un poids qui diminue l'importance d'une des bornes. Cela permet de forcer c_k à être du côté de cette borne modifiée relativement à la racine. Ainsi, on peut éviter d'avoir toujours la même borne qui est mise à jour par c_k.

Plus spécifiquement, on utilise les règles suivantes :

— Si c'est la borne inférieure qui est modifiée deux fois consécutives, alors on applique un poids de $\frac{1}{2}$ à $f(b_k)$ et on définit c_k comme suit :

$$c_k = \frac{\frac{1}{2}a_k f(b_k) - b_k f(a_k)}{\frac{1}{2}f(b_k) - f(a_k)}$$

— Si c'est la borne supérieure qui est modifiée deux fois consécutives, alors on

applique un poids de $\frac{1}{2}$ à $f(a_k)$:

$$c_k = \frac{a_k f(b_k) - \frac{1}{2} b_k f(a_k)}{f(b_k) - \frac{1}{2} f(a_k)}$$

1.6.2 Exemples pratiques d'application

La méthode de la fausse position modifiée est utilisée dans divers domaines où une convergence rapide vers une racine est souhaitée :

— **Physique et ingénierie** : Pour trouver les points d'équilibre dans les systèmes mécaniques ou électriques.

— **Économie** : Pour résoudre des équations non linéaires dans les modèles économiques.

— **Sciences de la terre** : Dans les calculs de géophysique, pour déterminer des racines dans des modèles de propagation d'ondes ou de déplacement tectonique.

1.6.3 Pseudo-code de l'algorithme

Nous rappelons ci-dessous le pseudo-code de la méthode de la fausse position modifiée.

Algorithm: ModifiedRegulafalsiMethod

Input: f, ϵ, a, b tel que $f(a)f(b) < 0$
Output: $\overline{x}$ tel que $|f(\overline{x})| < \epsilon$

$k \leftarrow 0$, $a_0 \leftarrow a$, $b_0 \leftarrow b$, $c_0 \leftarrow (a_0 f(b_0) - b_0 f(a_0))/(f(b_0) - f(a_0))$
repeat
  $k \leftarrow k + 1$
  $\alpha \leftarrow 1$, $\beta \leftarrow 1$
  if $f(a_{k-1})f(c_{k-1}) \leq 0$ **then**
    if $f(c_{k-1}) = 0$ **then**
      return $\overline{x} \leftarrow c_{k-1}$
    $a_k \leftarrow a_{k-1}$, $b_k \leftarrow c_{k-1}$
    if $k > 1$ and $f(c_{k-2})f(c_{k-1}) > 0$ **then**
      $\alpha = 1/2$
  else
    $a_k \leftarrow c_{k-1}$, $b_k \leftarrow b_{k-1}$
    if $k > 1$ and $f(c_{k-2})f(c_{k-1}) > 0$ **then**
      $\beta = 1/2$;
  $c_k \leftarrow (\beta a_k f(b_k) - \alpha b_k f(a_k))/(\beta f(b_k) - \alpha f(a_k))$
until $|f(c_k)| < \epsilon$
return $\overline{x} \leftarrow c_k$

Figure 1.18 : Pseudo-code de l'algorithme de la fausse position modifiée

1.6.4 Code Python de l'algorithme

Voici ci-dessous le code Python de notre implémentation de l'algorithme présenté dans le paragraphe précédent.

```python
def ModifiedRegulafalsiMethod(f, a, b, epsilon):
    # Vérifier que f(a) et f(b) sont de signes opposés
    if f(a) * f(b) >= 0:
        raise ValueError("f(a) et f(b) doivent avoir des signes opposés")

    # Initialisation des variables
    k = 1
    a_k = a
    b_k = b
    c_k = (a_k * f(b_k) - b_k * f(a_k)) / (f(b_k) - f(a_k))
    prev_c_k = None  # Pour stocker la valeur précédente de c_k

    # Boucle principale
    while abs(f(c_k)) > epsilon:
        k = k + 1
        alpha = 1
        beta = 1

        if f(a_k) * f(c_k) < 0:
            b_k = c_k
            if prev_c_k is not None and f(prev_c_k) * f(c_k) > 0:
                alpha = 1 / 2
            # Mettre à jour c_k en utilisant les valeurs pondérées de a_k et b_k
            c_k = (beta * b_k * f(a_k) - alpha * a_k * f(b_k)) / (beta * f(a_k) - alpha * f(b_k))
        else:
            a_k = c_k
            if prev_c_k is not None and f(prev_c_k) * f(c_k) > 0:
                beta = 1 / 2
            # Mettre à jour c_k en utilisant les valeurs pondérées de a_k et b_k
            c_k = (beta * b_k * f(a_k) - alpha * a_k * f(b_k)) / (beta * f(a_k) - alpha * f(b_k))

        # Stocker la valeur précédente de c_k pour la prochaine itération
        prev_c_k = c_k

    return c_k
```

Figure 1.19 : Code python de l'algorithme de la fausse position modifiée

1.6.5 Illustration sur un exemple

Nous illustrons le bon fonctionnement de l'algorithme implémenté en l'exécutant sur l'exemple suivant : Définissons la fonction :

$$f(x) = x^3 + x - 1$$

Prenons l'intervalle $[a, b] = [0, 1]$ et la précision $\epsilon = 0.001$.

Nous appliquons la méthode de la fausse position modifiée pour trouver la racine de $f(x)$ dans cet intervalle :

```python
# Définir la fonction f(x)
def f(x):
    return x**3 + x - 1

# Définir l'intervalle et la précision
a = 0
b = 1
epsilon = 0.001

# Appeler la méthode de la fausse position modifiée
root = ModifiedRegulafalsiMethod(f, a, b, epsilon)
print("Racine trouvée :", root)
```

✓ 0.0s

```
Racine trouvée : 0.6819735421424331
```

Figure 1.20 : Exemple de l'algorithme de la fausse position modifiée

La racine trouvée est : 0.6819735421424331

1.7 Algorithme 7 : Méthode de Newton

1.7.1 Objet de l'algorithme

La méthode de Newton emploie, dans l'idée, une stratégie d'approximation linéaire, similaire à celle de la méthode de la fausse position. Cependant, elle utilise comme approximation linéaire la tangente à f en x_k. Dans ce cas, l'approche repose sur une hypothèse supplémentaire par rapport aux méthodes précédentes : elle suppose que f est différentiable sur tout l'intervalle et que f', sa dérivée, est également continue.

L'approximation linéaire d'une fonction dans le voisinage d'un point est donnée par le développement de Taylor à l'ordre 1. Si f est différentiable dans le voisinage de x_k, alors la meilleure approximation linéaire est donnée par (cf. cours

MNO) :

$$f(x) \approx f(x_k) + f'(x_k)(x - x_k)$$

Ainsi, au lieu de résoudre directement $f(x) = 0$, la méthode de Newton remplace $f(x)$ par son approximation linéaire et résout :

$$f(x_k) + f'(x_k)(x - x_k) = 0$$

Ce qui mène à la mise à jour suivante :

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

Cette stratégie repose donc sur la *méthode du point fixe*, en utilisant la fonction g suivante :

$$x_{k+1} = g(x_k) = x_k - \frac{f(x_k)}{f'(x_k)}$$

1.7.2 Exemples pratiques d'application

La méthode de Newton est couramment utilisée dans les domaines suivants :

— **Physique et ingénierie** : pour résoudre des équations non linéaires complexes apparaissant dans les modèles physiques.

— **Économie et finance** : pour trouver des points d'équilibre ou résoudre des modèles économiques non linéaires.

— **Analyse numérique** : comme une méthode rapide pour l'approximation des racines d'équations différentiables.

1.7.3 Pseudo-code de l'algorithme

Nous rappelons ci-dessous le pseudo-code de l'algorithme de la méthode de Newton.

Algorithm: NewtonMethod

Input: f, f', x_0 tel que $f'(x_0) \neq 0, \epsilon, \delta, \mathrm{K}$
Output: $\overline{x}$ tel que $|f(\overline{x})| < \epsilon$

$k \leftarrow 0$
repeat
  $k \leftarrow k + 1$
  $x_k = x_{k-1} - \frac{f(x_{k-1})}{f'(x_{k-1})}$
  if $k \geq \mathrm{K}$ ou $f'(x_k) = 0$ **then**
    return Pas de racine trouvée
until $|f(x_k)| < \epsilon$ ou $|x_k - x_{k-1}| < \delta$
return x_k

Figure 1.21 : Pseudo-code de la méthode de Newton

1.7.4 Code Python de l'algorithme

Voici ci-dessous le code Python de notre implémentation de l'algorithme présenté dans le paragraphe précédent.

```python
def NewtonMethod(f, f_prime, x0, epsilon, delta, K):
    k = 0
    x_k = x0

    while True:
        k += 1
        # Calculer la nouvelle valeur de x_k
        x_k_new = x_k - f(x_k) / f_prime(x_k)

        # Vérifier la condition d'arrêt
        if abs(f(x_k_new)) < epsilon or abs(x_k_new - x_k) < delta:
            return x_k_new

        # Si la dérivée est nulle, arrêter
        if k >= K or f_prime(x_k) == 0:
            return "Pas de racine trouvée"

        # Mettre à jour x_k pour la prochaine itération
        x_k = x_k_new
```

Figure 1.22 : Code python de la méthode de Newton

1.7.5 Illustration sur un exemple

Nous illustrons le bon fonctionnement de l'algorithme implémenté en l'exécutant sur l'exemple suivant : On considère la fonction f définie par :

$$f(x) = x^3 + x - 1$$

et sa dérivée :

$$f'(x) = 3x^2 + 1$$

Le code Python pour résoudre cet exemple est le suivant :

```python
# Définir la fonction et sa dérivée
def f(x):
    return x**3 + x - 1  # Fonction f(x) = x^3 + x - 1

def f_prime(x):
    return 3*x**2 + 1  # Dérivée de f(x)

# Paramètres de la méthode
x0 = 0.5  # Point de départ
epsilon = 0.001  # Précision sur la fonction
delta = 0.001  # Précision sur la différence entre x_k et x_k-1
K = 10  # Nombre maximal d'itérations

# Appeler la méthode de Newton
root = NewtonMethod(f, f_prime, x0, epsilon, delta, K)
print("Racine trouvée :", root)
```

✓ 0.0s

```
Racine trouvée : 0.6823284233045783
```

Figure 1.23 : Exemple de la méthode de Newton

La racine trouvée est : 0.6823284233045783

1.8 Algorithme 8 : Méthode du trapèze composite

1.8.1 Objet de l'algorithme

La méthode du trapèze composite est une technique d'intégration numérique utilisée pour approximer l'intégrale d'une fonction continue sur un intervalle donné $[a, b]$. L'idée principale est de diviser l'intervalle $[a, b]$ en m segments de même longueur et d'appliquer la méthode du trapèze sur chaque sous-intervalle. En appliquant cette méthode de manière répétée sur chaque segment, on obtient une meilleure approximation de l'intégrale totale.

Pour un intervalle $[a, b]$ divisé en m segments de taille égale $h = \frac{b-a}{m}$, les points de division sont donnés par $x_0, x_1, \ldots, x_m$. La fonction f est ensuite approximée par morceaux, en utilisant un polynôme de degré 1 sur chaque intervalle $[x_i, x_{i+1}]$.

On en déduit que :

$$\int_a^b f(x)\,dx \approx \frac{h}{2}\left(f(x_0) + 2f(x_1) + \cdots + 2f(x_{m-1}) + f(x_m)\right)$$

1.8.2 Exemples pratiques d'application

La méthode du trapèze composite est largement utilisée dans les domaines suivants :

— **Physique** : Pour estimer des intégrales dans des problèmes de mécanique ou d'électromagnétisme.

— **Finance** : Pour le calcul d'intégrales dans la modélisation de flux de trésorerie.

— **Ingénierie** : Dans les analyses de structures ou de vibrations, où l'intégration numérique est nécessaire pour estimer les réponses d'un système.

1.8.3 Pseudo-code de l'algorithme

Nous rappelons ci-dessous le pseudo-code de l'algorithme la méthode de trapèze composite.

Algorithm: CompositeTrapezoidalRule

Input: f, m, a, b
Output: Valeur approchée de $\int_a^b f(x)dx$

```
1 Découper [a, b] en m intervalles de même taille h
2 I ← 0
3 for i = 0, . . . , m do
4 |   if i == 0 ou i == m then
5 |   |   I ← I + f(x_i)
6 |   else
7 |   |   I ← I + 2f(x_i)
8 return I h/2
```

Figure 1.24 : Pseudo-code de la méthode de trapèze composite

1.8.4 Code Python de l'algorithme

Voici ci-dessous le code Python de notre implémentation de l'algorithme présenté dans le paragraphe précédent.

```python
def compositeTrapezoidalRule(f,m,a,b):
    h=(b-a)/m    #m taille de segment

    l=0
    for i in range(m+1):
        x_i=a+ i*h #calcul du point x_i
        if (i==0) or (i==m):
            l=l+ f(x_i)
        else:
            l=l+2*f(x_i)

    return l*(h/2)
```

✓ 0.0s

Figure 1.25 : Code python de la méthode de trapèze composite

1.8.5 Illustration sur un exemple

Nous illustrons le bon fonctionnement de l'algorithme implémenté en l'exécutant sur l'exemple suivant :

Soit la fonction $f(x) = \sin(x)$ que nous voulons intégrer sur l'intervalle $[0, \pi]$. Nous appliquerons la méthode du trapèze composite avec différents nombres de segments m pour estimer l'intégrale de cette fonction.

Le code Python pour résoudre cet exemple est le suivant :

```python
def f(x):
    return np.sin(x)

a=0
b=np.pi

#test avec 8 seg
appox_8_seg=compositeTrapezoidalRule(f,8,a,b)
print ("Appro avec 8 seg :",appox_8_seg)

#test avec 16 seg
appox_16_seg=compositeTrapezoidalRule(f,16,a,b)
print ("Appro avec 16 seg :",appox_16_seg)
```

✓ 0.0s

```
Appro avec 8 seg : 1.9742316019455508
Appro avec 16 seg : 1.9935703437723395
```

Figure 1.26 : Exemple de la méthode de trapèze composite

Approximation avec 8 seg : 1.9742316019455508

Approximation avec 16 seg : 1.9935703437723395

1.9 Algorithme 9 : La méthode de Simpson composite

1.9.1 Objet de l'algorithme

La méthode de Simpson composite est une approche numérique pour évaluer l'intégrale d'une fonction $f(x)$ sur un intervalle $[a, b]$ en divisant cet intervalle en m sous-intervalles de même longueur $h = \frac{b-a}{m}$. On suppose ici que m est pair.

Chaque sous-intervalle $[x_i, x_{i+1}]$ est alors approximé par un polynôme de degré 2. En appliquant la méthode de Simpson sur chaque sous-intervalle et en sommant les résultats, on obtient l'approximation de l'intégrale sur $[a, b]$.

La formule pour cette méthode est donnée par :

$$\int_a^b f(x)dx \approx \frac{h}{3}\left(f(x_0) + 4\sum_{i=1}^{m/2} f(x_{2i-1}) + 2\sum_{i=1}^{m/2-1} f(x_{2i}) + f(x_m)\right)$$

où $x_0 = a$, $x_m = b$, et les points intermédiaires x_i sont définis comme $x_i = a + i \cdot h$.

1.9.2 Exemples pratiques d'application

La méthode de Simpson composite est utilisée dans de nombreux domaines pour calculer des intégrales de fonctions lorsqu'une solution analytique est difficile ou impossible à obtenir :

— **Physique :** pour calculer l'aire sous une courbe de distribution de probabilités ou des trajectoires de particules en mécanique.

— **Ingénierie :** dans l'analyse des signaux pour déterminer l'énergie totale d'un signal continu.

— **Économie :** pour estimer l'aire sous une courbe de densité de probabilité dans les modèles financiers.

1.9.3 Pseudo-code de l'algorithme

Nous rappelons ci-dessous le pseudo-code de l'algorithme de la méthode de Simpson composite.

Algorithm: CompositeSimpsonRule

Input: f, m, a, b (m doit être paire)
Output: Valeur approchée de $\int_a^b f(x)dx$

```
1  Découper [a, b] en m intervalles de même taille h
2  I ← 0
3  for i = 0, . . . , m do
4  |   if i == 0 ou i == m then
5  |   |   I ← I + f(x_i)
6  |   else
7  |   |   if i est pair then
8  |   |   |   I ← I + 2f(x_i)
9  |   |   else
10 |   |   |   I ← I + 4f(x_i)
11 return I h/3
```

Figure 1.27 : Pseudo-code de la méthode de Simpson composite

1.9.4 Code Python de l'algorithme

Voici ci-dessous le code Python de notre implémentation de l'algorithme présenté dans le paragraphe précédent.

```python
def compositeSimpsonRule(f,m,a,b):
    if m % 2 != 0 :
        print('m doir etre paire ')
        raise ValueError ('m doir etre paire')

    h= (b-a) / m
    l=0

    for i in range (m+1):
        x_i=a+ i*h #calcul du point x_i

        if (i==0) or (i==m):
            l = l+f(x_i)
        elif i % 2 ==0 : #si i paire
            l= l + 2*f(x_i)
        else :
            l= l + 4*f(x_i)  #si i impaire

    return l*(h/3)
```

Figure 1.28 : Code python de la méthode de Simpson composite

1.9.5 Illustration sur un exemple

Nous allons illustrer l'utilisation de la règle de Simpson composite sur la fonction $g(x) = \sin(x)$ dans l'intervalle $[0, \pi]$.

Le code Python pour résoudre cet exemple est le suivant :

```python
def g(x):
    return np.sin(x)

a=0
b=np.pi

#test avec 8 seg
appox_8_seg=compositeSimpsonRule(g,8,a,b)
print ("Appro avec 8 seg :",appox_8_seg)

#test avec 16 seg
appox_16_seg=compositeSimpsonRule(g,16,a,b)
print ("Appro avec 16 seg :",appox_16_seg)
```

✓ 0.0s

```
Appro avec 8 seg : 2.000269169948388
Appro avec 16 seg : 2.0000165910479355
```

Figure 1.29 : Exemple de la méthode de Simpson composite

Approximation avec 8 seg : 2.000269169948388

Approxiamtion avec 16 seg : 2.0000165910479355

1.10 Algorithme 10 : La méthode de Romberg

1.10.1 Objectif de l'algorithme

— L'**extrapolation de Richardson** est une technique simple permettant de **booster la précision de diverses méthodes numériques**, y compris l'intégration. En particulier, elle est utilisée pour améliorer l'approximation de l'intégrale d'une fonction f sur un intervalle donné $[a, b]$.

— Dans le cas de la méthode de Romberg, on combine l'**intégration par la méthode du trapèze composite** avec l'extrapolation de Richardson pour

obtenir une approximation plus précise de l'intégrale. La méthode consiste à calculer des valeurs approchées de l'intégrale avec des intervalles de plus en plus petits, et ensuite à appliquer l'extrapolation pour éliminer les erreurs principales.

— En introduisant des approximations successives $r_{11}, r_{21}, r_{31}, \ldots$ obtenues en divisant l'intervalle en segments de taille de plus en plus petite, l'extrapolation de Richardson permet d'atteindre un ordre d'erreur plus élevé à chaque étape de l'algorithme.

1.10.2 Exemples pratiques d'application

Nous illustrons le bon fonctionnement de la méthode de Romberg en l'appliquant à l'exemple suivant :

Soit la fonction $f(x) = \sin(x)$ sur l'intervalle $[0, \pi]$. Nous allons utiliser la méthode de Romberg pour calculer l'intégrale

$$\int_0^{\pi} \sin(x)\, dx.$$

En appliquant l'algorithme, nous obtenons une suite de valeurs de plus en plus précises qui convergent vers la valeur exacte de l'intégrale.

1.10.3 Pseudo-code de l'algorithme

Nous rappelons ci-dessous le pseudo-code de l'algorithme de la méthode Romberg.

Algorithm: RichardsonExtrapolation

Input: a, b, j
Output: Valeur de l'extrapolation de Richardson entre a et b
1 **return** $\frac{4^{j-1}a-b}{4^{j-1}-1}$

Algorithm: RombergIntegration

Input: $f, a, b, \epsilon, K = 10$ (nombre d'itérations maximales)
Output: Valeur approchée de $\int_a^b f(x)dx$
1 Initialiser **R** par une matrice nulle de taille $K \times K$
2 $r_{11} \leftarrow$ CompositeTrapezoidalRule(f, 1, a, b)
3 **for** $i = 2, \ldots, K$ **do**
4 $\quad r_{i1} \leftarrow$ CompositeTrapezoidalRule(f, 2^{i-1}, a, b)
5 $\quad r_{\text{old}} \leftarrow r_{i1}$
6 $\quad$ **for** $j = 2, \ldots, i$ **do**
7 $\quad\quad r_{ij} \leftarrow$ RichardsonExtrapolation($r_{i(j-1)}, r_{(i-1)(j-1)}, j$)
8 $\quad\quad$ **if** $|r_{ij} - r_{\text{old}}| < \epsilon$ **then**
9 $\quad\quad\quad$ **return** r_{ij}
10 $\quad\quad$ **else**
11 $\quad\quad\quad r_{\text{old}} \leftarrow r_{ij}$
12 **return** r_{old}

Figure 1.30 : Pseudo-code de la méthode de Romberg

1.10.4 Code Python de l'algorithme

Voici ci-dessous le code Python de notre implémentation de l'algorithme présenté dans le paragraphe précédent.

```python
# Fonction d'extrapolation de Richardson
def RichardsonExtrapolation(a, b, j):
    return (4**(j-1) * a - b) / (4**(j-1) - 1)

# Intégration de Romberg
def RombergIntegration(f, a, b, eps, K):
    # Initialisation de la matrice R avec la taille K x K
    R = np.zeros((K, K))
    R[1, 1] = compositeTrapezoidalRule(f, 1, a, b)

    for i in range(2, K+1):
        # Calcul de R[i,0] avec 2^i segments
        R[i, 1] = compositeTrapezoidalRule(f, 2**i, a, b)
        Rold = R[i, 1]

        for j in range(2, i + 1):
            # Application de l'extrapolation de Richardson
            R[i, j] = RichardsonExtrapolation(R[i, j-1], R[i-1, j-1], j)

            # Vérification de la convergence
            if np.abs(R[i, j] - Rold) < eps:
                return R[i, j]
            else:
                Rold = R[i, j]

    return Rold
```

✓ 0.0s

Figure 1.31 : Code python de la méthode de Romberg

1.10.5 Illustration sur un exemple

Nous appliquons notre code d'intégration de Romberg pour approximer l'intégrale de la fonction $f(x) = \sin(x)$ sur l'intervalle $[0, \pi]$, avec une précision de $\epsilon = 0.001$ et un nombre maximal d'itérations $K = 10$.

Le code Python pour résoudre cet exemple est le suivant :

```python
# Définition de la fonction à intégrer
def f(x):
    return np.sin(x)

# Paramètres de l'intégration
a = 0
b = np.pi
epsilon = 0.001
K = 10  # Nombre d'itérations

# Calcul de l'intégrale avec la méthode de Romberg
result_romberg = RombergIntegration(f, a, b, epsilon, K)
print("Approximation de l'intégrale avec Romberg :", result_romberg)
```

✓ 0.0s

```
Approximation de l'intégrale avec Romberg : 1.9999997524545725
```

Figure 1.32 : Exemple de la méthode de Romberg

Approximation de l'intégrale avec Romberg : 1.9999997524545725

Conclusion générale

Dans ce travail, nous avons implémenté plusieurs algorithmes d'analyse numérique en Python, en abordant des méthodes variées pour résoudre des problèmes de calcul, tels que la recherche de racines, l'ajustement de courbes, la résolution de systèmes d'équations linéaires, et l'intégration numérique.

Ce travail nous a permis de mieux comprendre les fondements théoriques de chaque algorithme et leur utilité pratique en analyse numérique. En appliquant ces méthodes, nous avons développé des compétences dans la traduction de concepts mathématiques en code et dans l'optimisation des calculs numériques.

Les principales difficultés rencontrées ont été liées à l'adaptation des méthodes théoriques aux exigences de précision numérique et à la gestion de la convergence pour certains algorithmes itératifs. L'implémentation de certaines méthodes, comme l'extrapolation de Richardson pour l'intégration de Romberg, a également présenté des défis en termes de stabilité numérique.

Certaines améliorations peuvent être apportées dans le futur, notamment en optimisant les algorithmes pour de plus grands ensembles de données et en explorant des variantes des méthodes de base pour améliorer la précision et la vitesse de convergence.

Dans l'ensemble, ce dossier nous a permis de renforcer notre compréhension des techniques de calcul numérique et de mieux saisir leur importance dans la résolution de problèmes complexes en sciences et en ingénierie.