Application en Ingénierie et Programmation Numérique

"Rendu III - Interpolation et Approximation"

VILLEDIEU Maxance et BESQUEUT Corentin

7 novembre 2023

Table des matières

I Interpolation Polynomiale **2**
I.1 Interpolation par la méthode de Neville . . . 3
I.1.1 Présentation de la méthode . . . 3
I.1.2 Résolution Manuelle . . . 4
I.1.3 Algorithme . . . 5
I.1.4 Implémentation en C . . . 5
I.1.5 Exemples d'exécution . . . 7

II Approximation **10**
II.1 Méthode des moindres au carré . . . 11
II.2 Approximation selon une droite de régression . . . 11
II.2.1 Présentation . . . 11
II.2.2 Résolution manuelle . . . 11
II.2.3 Algorithme et Implémentation en C . . . 12
II.2.4 Exemples d'exécutions et graphiques . . . 13
II.3 Approximation selon un ajustement exponentiel . . . 15
II.3.1 Présentation . . . 15
II.3.2 Résolution manuelle . . . 15
II.3.3 Algorithme et Implémentation en C . . . 16
II.3.4 Exemples d'exécutions et graphiques . . . 17
II.4 Approximation selon un ajustement puissance . . . 17
II.4.1 Présentation . . . 17
II.4.2 Résolution manuelle . . . 18
II.4.3 Algorithme et Implémentation en C . . . 19
II.4.4 Exemples d'exécutions et graphiques . . . 20

Annexe **21**
Jeux d'essais . . . 21

Chapitre I

Interpolation Polynomiale

L'interpolation est une technique fondamentale en analyse numérique, qui s'illustre aussi bien dans la modélisation mathématique que dans la visualisation de données. Elle permet l'approximation d'un ensemble de données discrètes, à partir d'une fonction continue. L'objectif de ce TP est d'explorer deux méthodes d'interpolation, à savoir les méthodes de Newton et Neville. L'interpolation de Newton repose sur les différences divisées, tandis que la méthode de Neville utilise un schéma récursif pour construire un polynôme interpolateur. Dans ce rapport, nous explorerons en détail ces deux méthodes d'interpolation. Pour chacune d'entre-elles, nous commencerons par une présentation théorique, puis nous illustrerons nos propos avec un exemple pratique de résolution. Nous nous concentrerons ensuite sur leurs algorithmes respectifs, ainsi que sur leur implémentation en C. Nous analyserons enfin les résultats fournis par l'ordinateur des deux méthodes sur les 4 jeux d'essais présents en annexe de ce rapport, ainsi que les avantages et les limitations de chaque méthode.

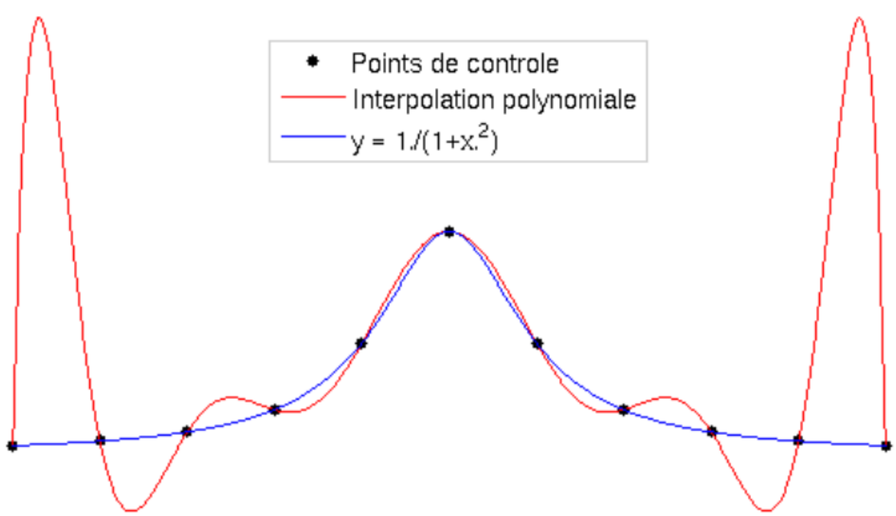


FIGURE I.1 – Exemple d'interpolation

I.1 Interpolation par la méthode de Neville

I.1.1 Présentation de la méthode

La méthode de Neville est une technique d'interpolation qui permet d'approximer une fonction inconnue à partir de données discrètes. Elle repose sur un processus récursif de construction d'un polynôme interpolateur à partir des données initiales. À chaque étape, deux polynômes voisins sont combinés pour former un nouveau polynôme qui passe par certains points données. Cette méthode devient rapidement imprécise au fur et à mesure que le nombre de points augmente. Elle est en revanche efficace pour l'interpolation de petits ensembles de données.

Considérons un ensemble de n points donnés, notés (x_i, y_i), où les x_i sont deux à deux distincts. Nous cherchons à déterminer un polynôme d'interpolation $p(x)$ de degré $n-1$ au maximum, qui satisfait la condition suivante :

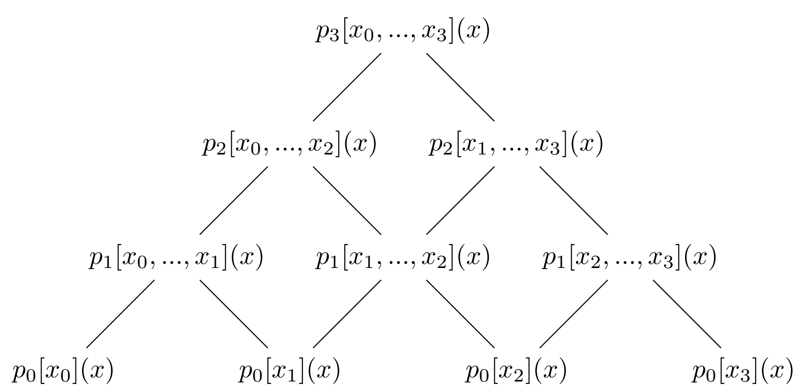
$$p(x_i) = y_i, \text{ avec } \ i = 0, ..., n-1$$

La méthode de Neville consiste à évaluer ce polynôme pour le point d'abscisse x.
Soit $p_k[x_i, ..., x_i + k](x)$ le polynôme de degré k qui passe par les points $(x_i, y_i), ..., (x_{i+k}, y_{i+k})$. Alors $p_k[x_i, ..., x_{i+k}](x)$ vérifie la relation de récurrence suivante :

$$\begin{cases} p_0[x_i](x) = y_i, \text{ avec } 0 \leqslant i < n \text{ et } k = 0 \\ p_k[x_i, ..., x_{i+k}](x) = \frac{(x - x_{i+k})p_{k-1}[x_i, ..., x_{i+k-1}](x) + (x_i - x)p_{k-1}[x_{i+1}, ..., x_{i+k}](x)}{x_i - x_{i+k}}, \text{ avec } 1 \leqslant k < n \text{ et } 0 \leqslant i < n \end{cases}$$

Cette relation de récurrence permet de calculer $p_{n-1}[x_0, ..., x_{n-1}](x)$, qui est le polynôme recherché.

Nous pouvons alors représenter les polynômes calculés par cette relation de récurrence dans un graphe. Soit une collection de 4 points (x, y). Alors le polynôme recherché, interpolateur de ces points, est $p_3[x_0, ..., x_3](x)$. Voici la représentation des polynômes calculés par la méthode :



I.1.2 Résolution Manuelle

Soient les points donnés dans le tableau ci-dessous.Déterminer le polynôme interpolateur $p(x)$ de ces points.

x_i	2	6	4
y_i	4	1.5	-2

Calcul des polynômes pour k=0

$$p_0[x_0](x) = y_0 = 4$$
$$p_0[x_1](x) = y_0 = 1.5$$
$$p_0[x_2](x) = y_0 = -2$$

Calcul des polynômes pour k=1

$$\begin{aligned}
p_1[x_0,x_1](x) &= \frac{(x-x_1)p_0[x_0](x)+(x_0-x)p_0[x_1](x)}{x_0-x_1} = \frac{(x-6)\times 4+(2-x)\times 1.5}{2-6} \\
&= \frac{2.5x-21}{-4} = -0.625x+5.25 \\
p_1[x_1,x_2](x) &= \frac{(x-x_2)p_0[x_1](x)+(x_1-x)p_0[x_2](x)}{x_1-x_2} = \frac{(x-4)\times 1.5+(6-x)\times(-2)}{6-4} \\
&= \frac{3.5x-18}{2} = 1.75x-9
\end{aligned}$$

Calcul du polynôme pour k=2

$$\begin{aligned}
p_2[x_0,x_1,x_2](x) &= \frac{(x-x_2)p_1[x_0,x_1](x)+(x_0-x)p_1[x_1,x_2](x)}{x_0-x_2} = \frac{(x-4)(-0.625x+5.25)+(2-x)(1.75x-9)}{2-4} \\
&= \frac{-2.375x^2+20.25x-39}{-2} = 1.1875x^2-10.125x+19.5
\end{aligned}$$

Nous avons donc bien notre polynôme interpolateur de degré $n-1=3-1=2$, qui passe par tous les points donnés :

$$p(x) = 1.1875x^2-10.125x+19.5$$

Nous pouvons vérifier cela :

$$\begin{aligned}
p(x_0) = p(2) = 1.1875\times 2^2-10.125\times 2+19.5 = 4 \\
p(x_1) = p(6) = 1.1875\times 6^2-10.125\times 6+19.5 = 1.5 \\
p(x_2) = p(4) = 1.1875\times 4^2-10.125\times 4+19.5 = -2
\end{aligned}$$

I.1.3 Algorithme

Nous allons poser l'algorithme suivant, qui reprend simplement la relation de récurrence vue dans la section I.1.1.

```
Fonction calculerPolynom(data, k, i, nbPoints):
    si k=0:
        renvoyer p_0[x_i](x)
    sinon:
        renvoyer ((x-x_{i+k}) × calculerPolynom(data,k-1,i,nbPoints) + (x_i-x) × calculerPolynom(data,k-1,i+1,nbPoints)) / (x_i-x_{i+k})
```

I.1.4 Implémentation en C

Pour l'implémentation en C de cet algorithme, plusieurs contraintes nous font obstacle. Avant de commencer à implémenter notre code, nous devons les citer et trouver un moyen de les franchir.

Nous devrons réfléchir à :
- De quelle manière stocker les points donnés ?
- Comment représenter un polynôme dans la mémoire ?
- Comment réaliser des opérations sur ces polynômes ?

Voici les décisions prises pour répondre à ces questions :

De quelle manière stocker les points donnés ?

Pour stocker les points fournis en entrée de notre programme, j'ai opté pour l'utilisation d'une matrice de flottants. La première ligne de cette matrice correspondra aux abscisses des points, tandis que la deuxième ligne contiendra leurs ordonnées. Sont alors mises en places des fonctions pour la création, le remplissage et l'affichage de la matrice, ainsi qu'une fonction pour libérer la mémoire allouée.

Ainsi, soient les points $(1,2),(2,4),(5,3)$, notre matrice contenant nos données sera alors $\begin{pmatrix} 1 & 2 & 5 \\ 2 & 4 & 3 \end{pmatrix}$.

Comment représenter un polynôme dans la mémoire ?

Pour représenter un polynôme en mémoire, j'ai choisi d'utiliser une nouvelle structure de données définissant le type **_polynom**. Chaque élément de cette structure se voit attribuer un entier **degree**, qui correspond au degré maximal du polynôme, ainsi qu'un tableau de nombres flottants **coefficients**. Les coefficients sont stockés dans l'ordre de la base canonique de l'espace vectoriel $\mathbb{R}_n[X]$. Autrement dit, le polynôme $3x^2+4x-2$ serait représenté par le tableau $[-2,4,3]$.

J'ai donc implémenté la structure **_polynom** ainsi que les fonctions suivantes :

- La fonction **createPolynom** crée un nouveau polynôme en initialisant le degré et les coefficients.
- La fonction **printPolynom** permet d'afficher le polynôme en console.
- La fonction **freePolynom** libère la mémoire allouée pour stocker le polynôme, évitant ainsi les fuites de mémoire.

Comment réaliser des opérations sur ces polynômes ?

Dans le but d'implémenter la méthode de Neville, nous avons besoin de pouvoir manipuler des polynômes. Pour cela, j'ai codé les opérations nécessaires : l'addition de polynômes, la multiplication de polynômes, et la division d'un polynôme par un flottant.

Addition de polynôme
L'opération d'addition de deux polynômes est relativement simple. Pour ce faire, il suffit d'additionner un à un les coefficients du polynôme de degré minimum, avec les coefficients de l'autre polynôme. Les résultats de ces additions sont ensuite stockés dans un nouveau polynôme, que nous renvoyons en sortie

de la fonction d'addition.

Multiplication de polynôme
La multiplication de polynôme, bien qu'aisée à effectuer manuellement, s'avère plus ardue lorsqu'il s'agit de l'implémenter de manière efficace. Heureusement ici, nous remarquons que la méthode de Neville n'utilise que la multiplication d'un polynôme de degré 1 avec un polynôme de degré k. Cela facilitera grandement notre travail.
Pour ce faire, nous prenons en paramètre de la fonction un nouveau polynôme de degré $k+1$. Ensuite, nous remplissons ce polynôme en multipliant chaque coefficient du polynôme de degré k par le coefficient associé à X de l'autre polynôme. Enfin, nous additionnons ces résultats avec les produits des coefficients du polynôme de degré k avec le coefficient restant du polynôme de degré 1. Nous stockons cela dans le nouveau polynôme.

Pour éclaircir tout cela, voici un exemple :

Supposons que l'on veuille réaliser le produit $(2X-1)(3X^2+2X+1)$.
Les polynômes sont respectivement représentés par les tableaux $[-1,2]$ et $[1,2,3]$.

1. Nous avons notre nouveau polynôme de degré 3 : $[.,.,.,.]$
2. Nous multiplions les coefficients $[1,2,3]$ par le coefficient devant X de l'autre polynôme, soit 2. On obtient : $[.,2,4,6]$.
3. Enfin, on multplie les coefficients $[1,2,3]$ par l'autre coefficient de l'autre polynôme, soit -1. On ajoute ces produits dans le nouveau polynôme, ce qui donne : $[-1,0,1,6]$.

Le produit de $(2X-1)(3X^2+2X+1)$ est donc $6X^3+X^2-1$.

Division de polynôme
Pour effectuer la division d'un polynôme par un flottant, nous stockons simplement, dans un nouveau polynôme, les quotients résultants de la division de chaque coefficient du polynôme par le diviseur fourni en paramètre.

I.1.5 Exemples d'exécution

Voici les différentes sorties du programme pour l'interpolation des jeux de données présents en annexe de ce document. Vous trouverez également un graphe représentant les points donnés, ainsi que le polynôme trouvé.

Listing I.1 – Annexe 1 data results

```
    Polynom
0.000000x^19 - 0.000000x^18 + 0.000000x^17 - 0.000000x^16 +
0.000000x^15 - 0.000000x^14 + 0.000000x^13 - 0.000000x^12 +
0.000001x^11 - 0.000025x^10 + 0.000361x^9 - 0.004044x^8 +
0.035004x^7 - 0.229974x^6 + 1.116527x^5 - 3.842630x^4 +
8.760551x^3 - 11.683978x^2 + 6.758180x^1 + 0.999870

Temps d'execution : 0.096877 secondes
```

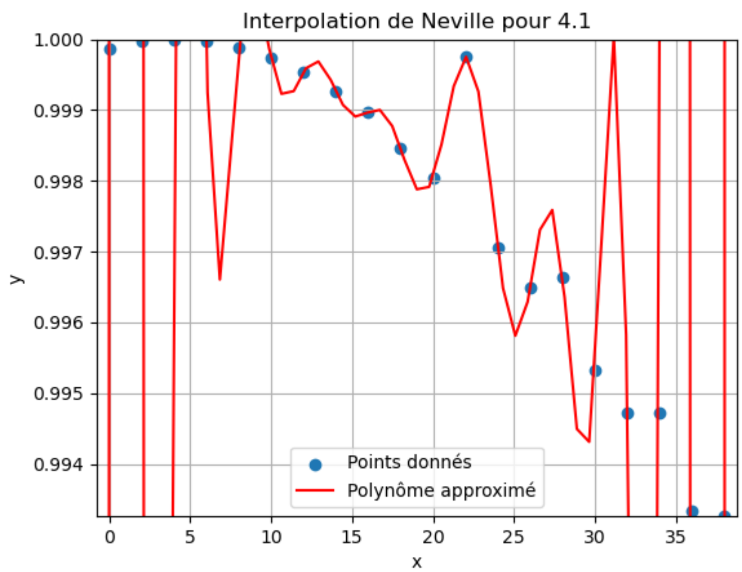


Figure I.2 – Interpolation du jeu de données 1 de l'annexe

Listing I.2 – Annexe 2 data results

```
    Polynom
0.000000x^20 - 0.000000x^19 + 0.000000x^18 - 0.000000x^17 +
0.000000x^16 - 0.000000x^15 + 0.000000x^14 - 0.000000x^13 +
0.000000x^12 - 0.000002x^11 + 0.001309x^10 - 0.860809x^9 +
465.832642x^8 - 206286.906250x^7 + 74020648.000000x^6 -
21189636096.000000x^5 + 4725708685312.000000x^4
- 791294909612032.000000x^3 + 93583403189796864.000000x^2 -
6969840492355780608.000000x^1 + 245843147369784279040.000000

Temps d'execution : 0.196364 secondes
```

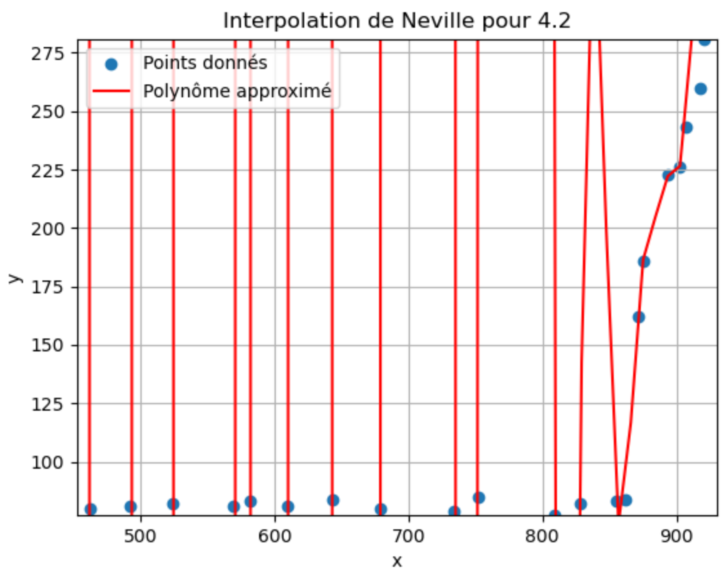


Figure I.3 – Interpolation du jeu de données 2 de l'annexe

Listing I.3 – Annexe 3 data results

```
    Polynom
0.000012x^10 - 0.001164x^9 + 0.049084x^8 - 1.220944x^7 +
19.789038x^6 - 217.668701x^5 + 1639.865601x^4 -
8326.726562x^3 + 27183.572266x^2 - 51370.457031x^1 + 42569.960938

Temps d'execution : 0.000282 secondes
```

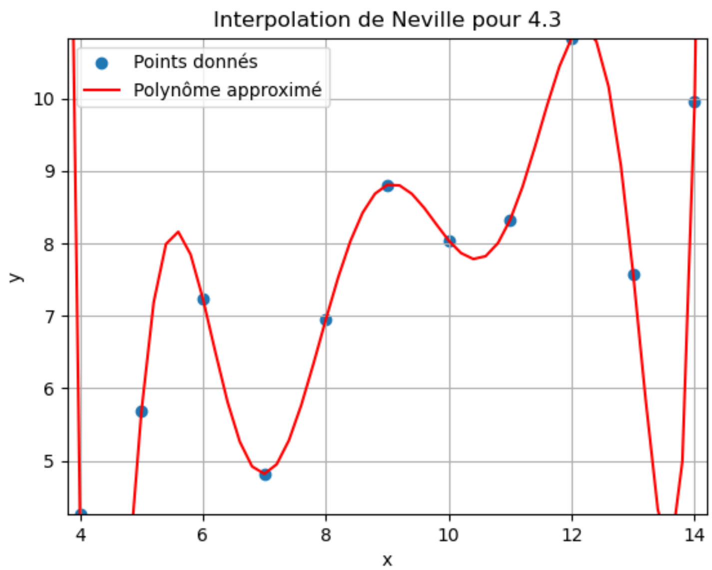


Figure I.4 – Interpolation du jeu de données 2 de l'annexe

Chapitre II

Approximation

L'approximation est également une méthode d'analyse s'illustrant dans la modélisation mathématique et la visualisation de données. Elle permet l'approximation d'un ensemble de données discrètes, à partir d'objets mathématiques. L'objectif de ce TP est d'explorer plusieurs méthodes d'approximation, à savoir l'approximation par une droite de regression, par un ajustement exponentiel, ou par un ajustement puissance. Vous trouverez dans ce chapitre la présentation des trois approches d'approximations, leur implémentation en C, ainsi que des exemples commentés, et illustrés par des graphiques.

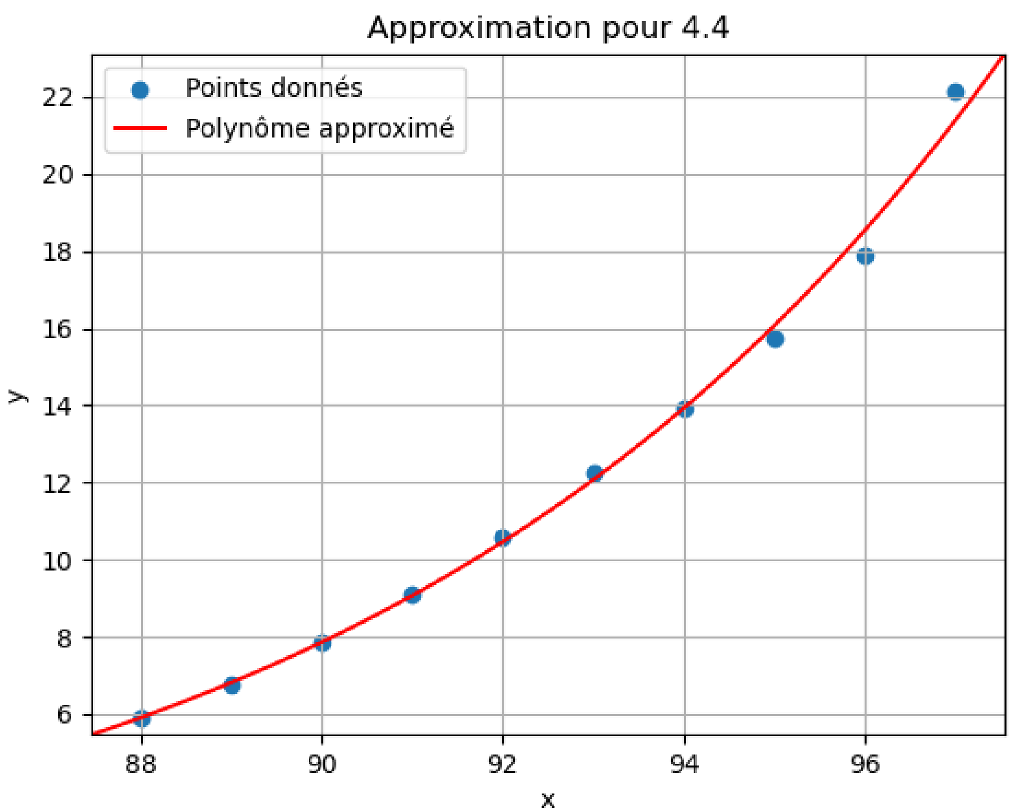


FIGURE II.1 – Exemple d'approximation

II.1 Méthode des moindres au carré

II.2 Approximation selon une droite de régression

II.2.1 Présentation

Pour approximer une collection de points à l'aide d'une droite de régression, nous allons utiliser le cas particulier de la méthode des moindres au carré. Dans ce cas, l'approximation se fait par une droite représentative du polynôme $f(x) = a_0x + a_1$ et on obtient a_0 et a_1 par les calculs suivants :

$$a_0 = \frac{\overline{y}\times\overline{x^2}-\overline{x}\times\overline{yx}}{\overline{x^2}-(\overline{x})^2},$$
$$a_1 = \frac{\overline{yx}-\overline{x}\times\overline{y}}{\overline{x^2}-(\overline{x})^2}$$

où $\overline{x} = \frac{\sum_{i=0}^{n-1} x_i}{n}$, $\overline{yx} = \frac{\sum_{i=0}^{n-1} x_i y_i}{n}$, $\overline{y} = \frac{\sum_{i=0}^{n-1} y_i}{n}$, $\overline{x^2} = \frac{\sum_{i=0}^{n-1} x_i^2}{n}$

II.2.2 Résolution manuelle

Soient les points $(1,3),(8,1),(1.5,-1)$. Approximer ces points par une droite de régression. Dans un premier temps, calculons $\overline{x},\overline{y},\overline{x^2},\overline{yx}$:

$$\overline{x} = (1+8+1.5)/3 = 3.5$$
$$\overline{y} = (3+1-1)/3 = 1$$
$$\overline{x^2} = (1^2+8^2+1.5^2)/3 \approx 22.42$$
$$\overline{yx} = (1\times 3+8\times 1+1.5\times(-1))/3 \approx 3.17$$

On calcule alors a_0 et a_1 :

$$a_0 = \frac{1\times 22.42 - 3.5\times 3.17}{22.42-3.5^2} \approx 1.11$$
$$a_1 = \frac{3.17-3.5\times 1}{22.42-3.5^2} \approx -0.03$$

Ainsi, la droite d'équation $y = -0.03x + 1.11$ approxime les points $(1,3),(8,1),(1.5,-1)$.

II.2.3 Algorithme et Implémentation en C

Note : pour l'implémentation de cette approximation, nous allons réutiliser la matrice de stockage des points, la structure _polynom, ainsi que leurs fonctions associées (c.f. section I.1.4).

Pour implémenter cette approche, nous avons donc besoin de calculer les moyennes blablabla . Pour cela, nous avons implémenté une fonction **getMeans**. Voici son pseudo-code :

```
Fonction getMeans:
    sumX = 0
    sumY = 0
    sumX2 = 0
    sumXY = 0;
    Pour i de 0 a n:
        sumX+ = x_i
        sumY+ = y_i
        sumX2+ = x_i × x_i
        sumXY+ = x_i + y_i
    renvoyer [sumX/nbPoints, sumY/nbPoints, sumX2/nbPoints, sumXY/nbPoints]
```

Ceci fait, ils nous reste simplement à coder la fonction qui retourne le bon polynôme. Voici son implémentation en C :

```
_polynom computeLinearPolynom(float** data, int n) {
    _polynom polynom = createPolynom(1);
    float* meanValues = getMeans(data, n);
    polynom.coefficients[0] = ((meanValues[1] * meanValues[2]) - (meanValues[0] * meanValues[3])) /
                              (meanValues[2] - (meanValues[0] * meanValues[0]));
    polynom.coefficients[1] = (meanValues[3] - (meanValues[0] * meanValues[1])) /
                              (meanValues[2] - (meanValues[0] * meanValues[0]));

    free(meanValues);
    return polynom;
}
```

Nous ne détaillerons pas ici les fonctions d'affichage telles que les fonctions pour afficher une matrice ou pour afficher un polynôme.

II.2.4 Exemples d'exécutions et graphiques

Vous trouverez en annexe de ce rapport plusieurs jeux de données qui peuvent être approximés par une droite. C'est notamment le cas des 3 premiers. Voici les polynômes retournés par l'exécution du programme avec ces collections, ainsi que la représentation graphique illustrant les résultats.

Premier Jeu de Données

Après exécution, notre programme retourne le polynôme : $y = -0.000186x + 1.001279$. Nous avons tracé le polynôme ainsi que les points fournis pour obtenir le graphique suivant :

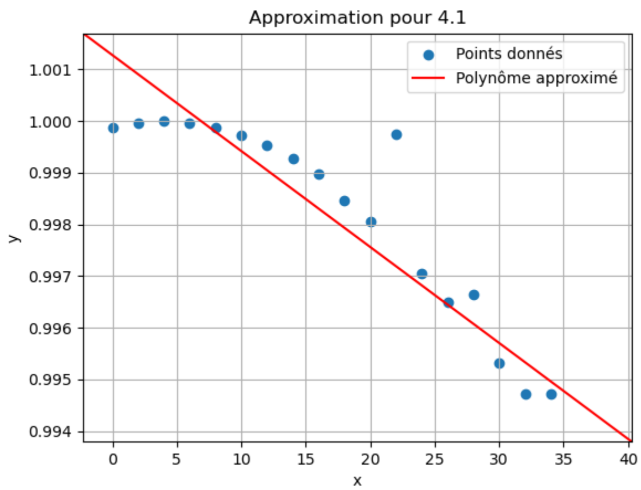


FIGURE II.2 – Approximation du premier jeu de données

Deuxième Jeu de Données

Après exécution, notre programme retourne le polynôme : $y = 0.324356x + -112.658463$. Nous avons tracé le polynôme ainsi que les points fournis pour obtenir le graphique suivant :

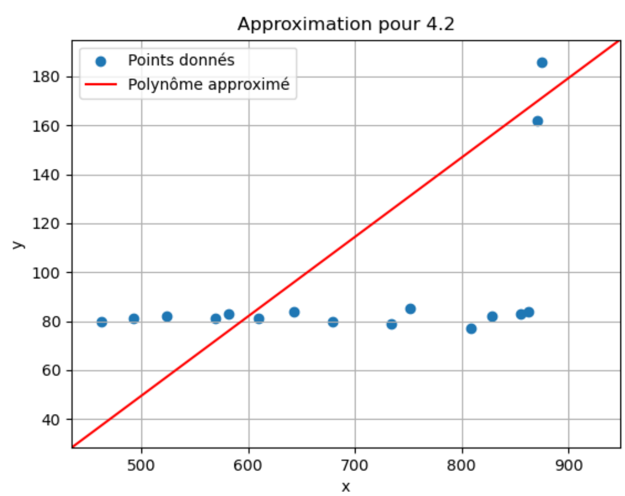


FIGURE II.3 – Approximation du deuxième jeu de données

Troisième Jeu de Données

Après exécution, notre programme retourne le polynôme : $y = 0.500092x + 3.000079$. Nous avons tracé le polynôme ainsi que les points fournis pour obtenir le graphique suivant :

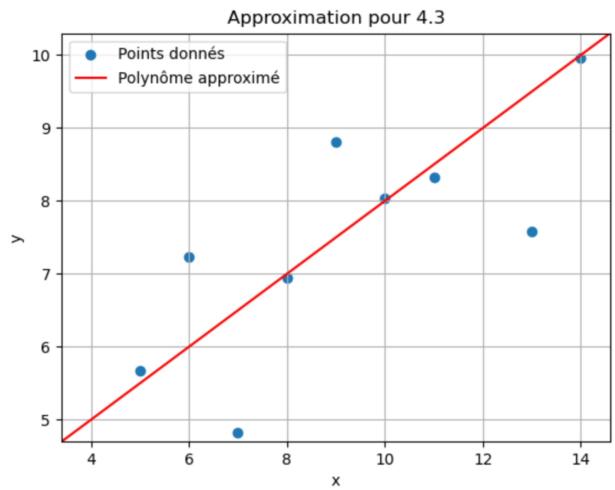


FIGURE II.4 – Approximation du deuxième jeu de données

II.3 Approximation selon un ajustement exponentiel

II.3.1 Présentation

L'objectif pour nous ici va être de trouver un moyen de revenir à une regression linéaire. Pour cela, il faut que l'on retrouve une forme $y = ax + b$ en partant d'une forme $y = ce^{dx}$. En manipulant l'expression grâce à l'exponentielle et au logarithme népérien, nous obtenons le résultat suivant :

$$
\begin{aligned}
y &= ce^{dx} \\
ln(y) &= ln(ce^{dx}) \\
ln(y) &= ln(c) + dx
\end{aligned}
$$

On pose $Y = ln(y)$, $B = ln(c)$ et $A = d$. On peut alors écrire $Y = Ax + B$. C'est la forme que l'on souhaitait. À partir des points initiaux, nous allons calculer $z_i = ln(y_i)$ pour $i = 0, ..., n-1$. Il suffira alors de calculer une droite de régression comme vue précedemment avec les points (x_i, z_i). Enfin, nous calculerons la valeur de c et d avec ce qui a été posé précedemment pour obtenir une forme exponentielle.

II.3.2 Résolution manuelle

Soient les points $(2,1), (3,4), (2.5,3)$. Approximer ces points par un ajustement exponentiel. Calculons $z_i = ln(y_i)$ pour $i = 0, 1, 2$:

$$
\begin{aligned}
z_0 &= ln(y_0) = ln(1) = 0 \\
z_1 &= ln(y_1) = ln(4) \approx 1.39 \\
z_2 &= ln(y_2) = ln(3) \approx 1.1
\end{aligned}
$$

Calculons alors une droite de régression qui approxime tous les points (x_i, z_i) :
Après calculs, nous obtenons :

$$
\begin{aligned}
\overline{x} &= 2.5 \\
\overline{z} &\approx 0.83 \\
\overline{x^2} &\approx 6.42 \\
\overline{zx} &\approx 2.31
\end{aligned}
$$

Nous pouvons alors calculer B et A, pour obtenir la droite $Y = 1.38x - 2.63$. Enfin, il ne nous reste plus qu'à exprimer la forme exponentielle, en faisant le chemin inverse que celui qui est présenté au début de cette partie.

$$
\begin{aligned}
Y &= Ax + B = 1.38x - 2.63 \\
\Leftrightarrow ln(y) &= ln(c) + dx \\
\Leftrightarrow y &= e^{ln(c)+dx} = e^{ln(c)} \times e^{dx} = e^{-2.63} \times e^{1.38x} = 0.072e^{1.38x}
\end{aligned}
$$

Nous avons donc comme approximation exponentielle $y = 0.072e^{1.38x}$.

II.3.3 Algorithme et Implémentation en C

Note : *pour l'implémentation de cette approximation, nous allons réutiliser la matrice de stockage des points, la structure _polynom, ainsi que leurs fonctions associées (c.f. section I.1.4). Nous réutiliserons également la fonction getMeans vu précédemment.*

Pour implémenter cette approximation, nous allons devoir créer une deuxième matrice de points, qui cette fois contiendra les points $(x_i, ln(y_i))$. Pour cela, nous allons coder une fonction *completeDataMatrix*, qui sera par ailleurs réutilisée dans l'approximation par un ajustement de puissance. Cette fonction va prendre en paramètre la matrice de donnée initiale, une nouvelle matrice de donnée vierge, le nombre de points donnés, et le type d'approximation (exponentiel ou puissance). Voici son implémentation en C :

```
void completeDataMatrix(float** data, float** matrix, int nbData, char type){
    for (int i = 0; i < nbData; i++) {
        if(type=='e'){
            matrix[0][i]=data[0][i];
            matrix[1][i]=logf(data[1][i]);
        }
        ...
    }
}
```

À partir de là, nous avons tous les outils nécessaires pour l'approximation. Voici le code source de la fonction qui calcule le polynôme recherché :

```
_polynom computeExponentialPolynom(float** data, int n) {
    //Creation d'une matrice pour les donnees necessaires a l'ajustement exponentiel
    float** dataForExp = createMatrix(2, n);
    completeDataMatrix(data, dataForExp, n, 'e');
    //Creation du polynome ln(y)=Bx+A
    _polynom polynom = computeLinearPolynom(dataForExp, n);
    //Transformation en y=exp(A)exp(Bx)
    polynom.coefficients[0]=expf(polynom.coefficients[0]);
    polynom.coefficients[1]=polynom.coefficients[1];
    freeMatrix(dataForExp, 2);
    return polynom;
}
```

II.3.4 Exemples d'exécutions et graphiques

Pour mettre en pratique le code précédemment présenté, nous pouvons l'exécuter avec le quatrième jeu de donnée présent en annexe du rapport. En effet, les points se pretent bien à une approximation exponentielle.
Voici le polynôme retourné par le programme : $y = 0.000020exp(0.143061x)$. Nous avons tracé le polynôme ainsi que les points fournis pour obtenir le graphique suivant :

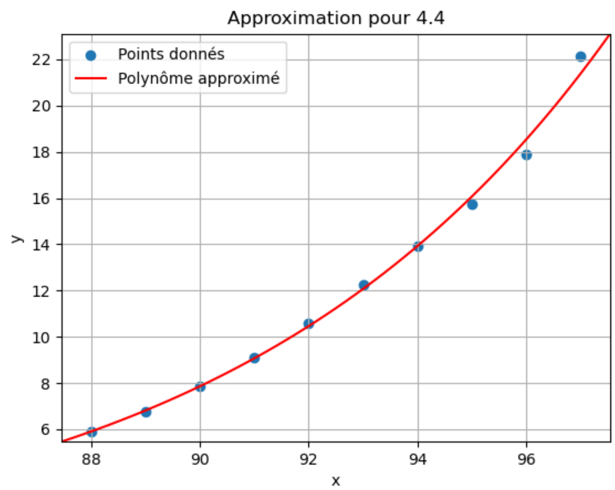


FIGURE II.5 – Approximation du quatrième jeu de données

II.4 Approximation selon un ajustement puissance

II.4.1 Présentation

Le but va ici être de trouver un moyen de revenir à une regression linéaire, de la même manière que pour l'approximation exponentielle. Pour cela, il faut que l'on retrouve une forme $y = ax + b$ en partant d'une forme $y = ax^b$. En manipulant l'expression grâce au logarithme, nous obtenons le résultat suivant :

$$\begin{aligned} y &= ax^b \\ log(y) &= log(ax^b) \\ log(y) &= log(a) + blog(x) \end{aligned}$$

On pose $Y = ln(y)$, $B = log(a)$, $A = b$ et $X = log(x)$. On peut alors écrire $Y = AX + B$. C'est la forme que l'on souhaitait. À partir des points initiaux, nous allons calculer $w_i = log(x_i)$ et $z_i = log(y_i)$ pour $i = 0, ..., n-1$. Il suffira alors de calculer une droite de régression comme vue précedemment avec les points (w_i, z_i). Enfin, nous calculerons la valeur de a et b avec ce qui a été posé précedemment pour obtenir un ajustement puissance.

II.4.2 Résolution manuelle

Soient les points $(1,3),(2,7),(5,1.3)$. Approximer ces points par un ajustement puissance. Calculons $w_i = log(x_i)$ et $z_i = log(y_i)$ pour $i = 0,1,2$:

D'une part :

$$w_0 = log(x_0) = log(1) = 0$$
$$w_1 = log(x_1) = log(2) \approx 0.3$$
$$w_2 = log(x_2) = log(5) \approx 0.7$$

D'autre part :

$$z_0 = log(y_0) = log(3) \approx 0.48$$
$$z_1 = log(y_1) = log(7) \approx 0.84$$
$$z_2 = log(y_2) = log(1.3) \approx 0.11$$

Calculons alors une droite de régression qui approxime tous les points (w_i, z_i) :
Après calculs, nous obtenons :

$$\overline{w} = 0.34$$
$$\overline{z} \approx 0.48$$
$$\overline{w^2} \approx 0.19$$
$$\overline{wz} \approx 0.11$$

Nous pouvons alors calculer B et A, pour obtenir la droite $Y = -0.71x + 0.71$. Enfin, il ne nous reste plus qu'à exprimer le polynôme, en faisant le chemin inverse que celui qui est présenté au début de cette partie.

$$Y = Ax + B = -0.71x + 0.71$$
$$\Leftrightarrow log(y) = log(a) + blog(x)$$
$$\Leftrightarrow y = 10^{log(a)+blog(x)} = 10^{log(a)} \times 10^{blog(x)} = 10^{0.71} \times x^{-0.71} = 5.12x^{-0.71}$$

Nous avons donc comme approximation exponentielle $y = 5.12x^{-0.71}$.

II.4.3 Algorithme et Implémentation en C

Note : *pour l'implémentation de cette approximation, nous allons réutiliser la matrice de stockage des points, la structure _polynom, ainsi que leurs fonctions associées (c.f. section I.1.4). Nous réutiliserons également la fonction getMeans et la fonction completeDataMatrix vues précédemment.*

Pour implémenter cette approximation, nous allons aussi devoir créer une deuxième matrice de points, qui cette fois contiendra les points $(log(x_i), log(y_i))$. Pour cela, nous allons compléter la fonction *completeDataMatrix*. Voici son implémentation en C :

```
void completeDataMatrix(float** data, float** matrix, int nbData, char type){
    for (int i = 0; i < nbData; i++) {
        ...
        if(type=='p'){
            matrix[0][i] = log10f(data[0][i]);
            matrix[1][i] = log10f(data[1][i]);
        }
    }
}
```

À partir de là, nous avons tous les outils nécessaires pour l'approximation. Voici le code source de la fonction qui calcule le polynôme recherché :

```
_polynom computePowerPolynom(float** data, int n) {
    //Creation d'une matrice pour les donnees necessaires a l'ajustement puissance
    float** dataForPow = createMatrix(2, n);
    completeDataMatrix(data, dataForPow, n, 'p');
    //Creation du polynome log(y)=Blog(x)+A
    _polynom polynom = computeLinearPolynom(dataForPow, n);
    //Transformation en y=(10^A)x^B
    polynom.coefficients[0]=pow(10, polynom.coefficients[0]);
    polynom.coefficients[1]=polynom.coefficients[1];
    freeMatrix(dataForPow, 2);
    return polynom;
}
```

II.4.4 Exemples d'exécutions et graphiques

Pour mettre en pratique le code précédemment présenté, nous pouvons l'exécuter avec le dernier jeu de donnée présent en annexe du rapport. En effet, les points se prêtent bien à une approximation polynomiale.
Voici le polynôme retourné par le programme : $y = 696595.051263x^{-}2.527187$. Nous avons tracé le polynôme ainsi que les points fournis pour obtenir le graphique suivant :

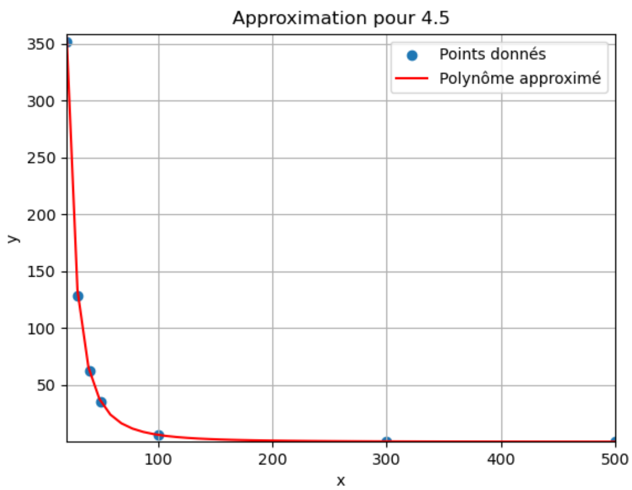


Figure II.6 – Approximation du cinquième jeu de données

Annexe

Jeux d'essais

Densité (D) de l'eau en fonction de la température (T)

T(C)	0	2	4	6	8	10	12	14	16	18
D(t/m^3)	0.99897	0.99846	0.99987	0.99997	1.00000	0.99997	0.99988	0.99973	0.99953	0.99927
T(C)	20	22	24	26	28	30	32	34	36	38
D(t/m^3)	0.99805	0.999751	0.99705	0.99650	0.99664	0.99533	0.99472	0.99472	0.99333	0.99326

Dépenses mensuelles et revenus

On s'intéresse à la relation qui existe entre les Dépenses de loisirs mensuelles D et les revenus R des employés d'une entreprise.

R	752	855	871	734	610	582	921	492	569	462	907
D	85	83	162	79	81	83	281	81	81	80	243
R	643	862	524	679	902	918	828	875	809	894	
D	84	84	82	80	226	260	82	186	77	223	

Série S dûe à Anscombe

x_i	10	8	13	9	11	14	6	4	12	7	5
y_i	8.04	6.95	7.58	8.81	8.33	9.96	7.24	4.26	10.84	4.82	5.68

Série chronologique avec accroissement exponentiel

x_i	88	89	90	91	92	93	94	95	96	97
y_i	5.89	6.77	7.87	9.11	10.56	12.27	13.92	15.72	17.91	22.13

Vérification de la loi de Pareto

Loi de Pareto : *"Entre le revenu x et le nombre y de personnes ayant un revenu supérieur à x, il existe une relation du type :*

$$y = \frac{A}{x^a} = Ax^{-a}$$

où a et A sont des constantes positives caractéristiques de la région considérée et de la période étudiée.

x_i	20	30	40	50	100	300	500
y_i	352	128	62.3	35.7	6.3	0.4	0.1