

Algorithmes numériques – Rapport
Interpolation et Approximation

Axel Delsol, Pierre-Loup Pissavy

Décembre 2013

Table des matières

1	Préambule	2
1.1	Structure du programme	2
1.2	Compilation et Logiciels utilisés	3
2	Interpolation	4
2.1	Méthode de Newton	4
2.1.1	Présentation	4
2.1.2	Programme	4
2.2	Méthode de Neville	6
2.2.1	Présentation	6
2.2.2	Programme	6
2.3	Résultats de tests	7
2.3.1	Exemple tiré d'un TD	7
2.3.2	Densité de l'eau en fonction de la température	8
2.3.3	3 séries	9
2.3.4	Dépenses et Revenus	12
2.4	Comparaison	14
3	Approximation	15
3.1	Régression linéaire	15
3.1.1	Présentation	15
3.1.2	Programme	15
3.2	Ajustement exponentiel	17
3.2.1	Présentation	17
3.2.2	Programme	17
3.3	Ajustement de type "puissance"	18
3.3.1	Présentation	18
3.3.2	Programme	18
3.4	Résultats de tests	19
3.4.1	Exemple tiré d'un TD	19
3.4.2	Série d'Anscombe	20
3.4.3	3 séries	21
3.4.4	Dépenses mensuelles et revenus	24
3.4.5	Série chronologique avec accroissement exponentiel	25
3.4.6	Vérification de la loi de Pareto	26
4	Conclusion	27
5	Annexe	28
5.1	Menu principal	28
5.2	Fonctions associées au calcul polynômial	29

1 Préambule

1.1 Structure du programme

Nous avons conçu un programme principal avec menus, présenté sous la forme suivante :

```
Menu principal : Interpolation et Approximation

Entrez n le nombre de points :
(Saisie de la série de points...)

(Affichage du tableau correspondant...)
Quelle résolution utiliser ?
1- Newton
2- Neuville
3- Régression Linéaire
4- Approximation par une fonction exponentielle
5- Approximation par une fonction "puissance"
9- Nouvelle série de points (Menu principal)
0- Quitter
Votre choix :
```

FIGURE 1.1 – Aperçu : Menu Principal

Au lancement, le programme demande la saisie des valeurs, qu'il stocke dans un tableau, puis affiche le menu. Après chaque résolution, il est possible de réutiliser le jeu de données (chaque méthode qui doit modifier les valeurs utilise un duplicata).

Le menu principal est codé dans le fichier source `main.c`. Les méthodes sont codées dans des fichiers individuels à l'exception des méthodes d'approximation qui sont toutes codées dans le même fichier puisqu'elles présentent de nombreuses similarités. Les prototypes des fonctions sont écrits dans les headers correspondants. Enfin, un fichier source présenté en annexe page 30 regroupe toutes les fonctions de manipulation de polynômes. La liste de tous ces fichiers est présentée figure 1.3.

Le stockage des valeurs se fait en double précision (type `double`, 64 bits) afin d'obtenir des résultats suffisamment précis pour tracer les courbes.

Et nos fonctions utilisent une structure de polynôme (composée du degré et des coefficients), présentée figure 1.2, pour faciliter la compréhension du code.


```
4 | typedef struct polynome
5 | {
6 |     int d; //degree
7 |     double* poln; //coefficients
8 | } polynome;
```

FIGURE 1.2 – Code : Structure de Polynôme

Note : Pour des raisons de lisibilité, les polynômes résultats sont arrondis dans ce rapport. En revanche, les graphiques sont tracés avec les valeurs calculées par la machine (précision maximale possible pour le type de données).

Les écarts donnés sont calculés par la machine juste après la résolution (on calcule la distance moyenne entre les points et la courbe).

Les arrondis affichés dans le rapport sont retournés à la demande par le programme, au format \LaTeX , dans un fichier intitulé `resultat`.



```
main.c
neuville.c
newton.c
polynome.c
reglin.c
lagrange.h
neuville.h
newton.h
polynome.h
reglin.h
makefile
```

FIGURE 1.3 – Aperçu : Arborescence des fichiers C et `makefile`

1.2 Compilation et Logiciels utilisés

La compilation est gérée par un `makefile`.

Le compilateur utilisé est `GCC`. Il suffit de taper `make` pour lancer la compilation, puis `./main` pour lancer le programme.

Pour nettoyer les fichiers temporaires, il faudra taper `make clean`.

Ce `makefile` permet également de générer ce rapport ainsi que quelques fichiers qui y sont intégrés.

Les représentations graphiques sont générées avec `Asymptote`, générateur vectoriel de graphiques.

Les polynômes résultats sont vérifiés avec `GeoGebra`, qui permet ensuite de générer une trame de fichier source pour `Asymptote`.

2 Interpolation

L'interpolation, en analyse numérique, est un ensemble de méthodes permettant d'obtenir une équation mathématique passant par tous les points d'une liste donnée.

Pour cette partie, les équations mathématiques recherchées sont des polynômes.

Notation pour la suite :

- La liste comporte N éléments (x_i, y_i) .
- Les polynômes recherchés sont de la forme

$$P_{N-1}(x) = \sum_{i=0}^{N-1} (a_i \cdot x^i)$$

2.1 Méthode de Newton

2.1.1 Présentation

La forme du polynôme par la méthode de Newton est la suivante :

$$P_{N-1}(x) = \sum_{i=0}^{N-1} \left(a_i \cdot \prod_{j=1}^i (x - x_j) \right)$$

Pour ce faire, on utilise une méthode de recherche de coefficients récursive appelée méthode des *différences divisées*. Le calcul des valeurs des différences divisées se fait à l'aide de fonctions :

La différence divisée de degré 0 est : $\forall i \in \{1, \dots, N\}, \quad \nabla_{y_i}^0 = y_i$.

La différence divisée de degré k est : $\forall i \in \{k+1, \dots, N\}, \quad \nabla_{y_i}^k = \frac{\nabla_{y_i}^{k-1} - \nabla_{y_k}^{k-1}}{x_i - x_k}$.

Ensuite, on a directement les coefficients du polynôme de Newton par la relation $\forall i \in \{1, \dots, N-1\}, \quad a_i = \nabla_{y_{(i+1)}}^i$.

Enfin, on peut retrouver la forme développée du polynôme à l'aide de la relation suivante :

$$\forall i \in \{0, \dots, N\}, \quad P_i(x) = \begin{cases} a_{N-1} & \text{si } i=0 \\ a_{N-1-i} + (x - x_{N-i}) \cdot P_{i-1}(x) & \text{sinon} \end{cases}.$$

2.1.2 Programme

```
1 | #include <stdio.h>
2 | #include <stdlib.h>
3 | #include <string.h>
4 | #include <math.h>
5 | #include "polynome.h"
6 |
```

```

7 void newton (double ** tab, int n)
8 {
9     int i, k;
10    polynome* pol1=(polynome*) malloc(sizeof(polynome));
11    polynome* pol2=(polynome*) malloc(sizeof(polynome));
12    double** t= (double**) malloc(n*sizeof(double*));
13    for (i=0; i<n; i++)
14    {
15        t[i]= (double*) malloc((i+1)*sizeof(double));
16    }
17    //initialisation des valeurs : on récupère les y.
18    for (i=0; i<n; i++)
19    {
20        t[i][0]=tab[1][i];
21    }
22    //calcul des differences divisees
23    for (k=1; k<n; k++)
24    {
25        for (i=k; i<n; i++)
26        {
27            t[i][k]=(t[i][k-1]-t[k-1][k-1])/(tab[0][i]-tab[0][k-1]);
28        }
29    }
30    //tableau de poly
31    polynome** tabP= (polynome**) malloc(n*(sizeof(polynome*)));
32    for (i=0; i<n; i++)
33    {
34        tabP[i]=(polynome*) malloc(sizeof(polynome));
35    }
36    tabP[0]->d=0;
37    tabP[0]->poln=(double*) malloc(sizeof(double));
38    tabP[0]->poln[0]=t[n-1][n-1];
39    for (i=1; i<n; i++)
40    {
41        pol1=creerPoly(2,"valeur",-tab[0][n-1-i], 1.);
42        pol2=mulPoly(pol1,tabP[i-1]);
43        free(pol1->poln); free(pol1);
44        pol1=creerPoly(1,"valeur",t[n-1-i][n-1-i]);
45        tabP[i]=addPoly(pol1,pol2);
46        free(pol1->poln); free(pol2->poln);
47        free(pol1); free(pol2);
48    }
49    redimensionnerPoly(tabP[n-1]);
50    //affichage
51    menuAffichage(tabP[n-1]);
52    ecartPoly(tab,n,tabP[n-1]);
53    printf("\n");
54    //libération mémoire
55    for(i=0;i<n;i++)
56    {
57        free(tabP[i]->poln);
58        free(tabP[i]);
59        free(t[i]);
60    }
61    free(tabP);
62    free(t);
63 }

```

FIGURE 2.1 – Code : newton.c

2.2 Méthode de Neville

2.2.1 Présentation

Cette méthode permet d'exprimer le polynôme $P_{N-1}[x_1, \dots, x_N]$ sur les points $\{1, \dots, N\}$ en fonction des polynômes $P_{N-2}[x_1, \dots, x_{N-1}]$ et $P_{N-2}[x_2, \dots, x_N]$ sur l'ensemble des points $\{1, \dots, N-1\}$ et $\{2, \dots, N\}$.

L'expression est donnée sous la forme suivante :

$$P_k[x_i, \dots, x_{i+k}](x) = \frac{(x-x_{i+k}) \cdot P_{k-1}[x_i, \dots, x_{i+k-1}](x) + (x_i-x) \cdot P_{k-1}[x_{i+1}, \dots, x_{i+k}](x)}{x_i-x_{i+k}}, \forall x, \forall k = 2, \dots, N-1$$

2.2.2 Programme

```
1 | #include <stdio.h>
2 | #include <stdlib.h>
3 | #include <string.h>
4 | #include <math.h>
5 | #include "polynome.h"
6 |
7 | void neuville (double ** tab, int n)
8 | {
9 |     int i, k;
10 |    polynome* pol1=(polynome*) malloc(sizeof(polynome));
11 |    polynome* pol2=(polynome*) malloc(sizeof(polynome));
12 |    polynome* pol3=(polynome*) malloc(sizeof(polynome));
13 |    polynome*** t= (polynome***) malloc(n*sizeof(polynome**));
14 |    for (i=0; i<n; i++)
15 |    {
16 |        t[i]= (polynome**) malloc((i+1)*sizeof(polynome*));
17 |    }
18 |    //initialisation des valeurs : on récupère les y.
19 |    for (i=0; i<n; i++)
20 |    {
21 |        t[i][0]=creerPoly(1,"valeur", tab[1][i]);
22 |    }
23 |    //calcul des differences divisees
24 |    for (k=1; k<n; k++)
25 |    {
26 |        for (i=k; i<n; i++)
27 |        {
28 |            pol1=creerPoly(2, "valeur", tab[0][i-k], -1.); pol2=mulPoly(pol1,t[i][k-1]);
29 |            free(pol1->poln); free(pol1);
30 |            pol1=creerPoly(2, "valeur", -(tab[0][i]), 1.); pol3=mulPoly(pol1, t[i-1][k-1]);
31 |            free(pol1->poln); free(pol1);
32 |            pol1=addPoly(pol3, pol2);
33 |            t[i][k]=mulSPoly((1/((tab[0][i-k])-(tab[0][i]))),pol1);
34 |            free(pol1->poln); free(pol2->poln); free(pol3->poln); free(pol1); free(pol2); free(pol3);
35 |        }
36 |    }
37 |    //poly à retourner
38 |    redimensionnerPoly(t[n-1][n-1]);
39 |    //affichage
40 |    menuAffichage(t[n-1][n-1]);
41 |    ecartPoly(tab,n,t[n-1][n-1]);
42 |    printf("\n");
43 |    //libération mémoire
44 |    for(i=0;i<n;i++)
45 |    {
46 |        for(k=0;k<i;k++)
47 |        { free(t[i][k]->poln); free(t[i][k]); }
48 |        free(t[i]);
49 |    }
50 |    free(t);
51 | }
```

FIGURE 2.2 – Code : neuville.c

2.3 Résultats de tests

2.3.1 Exemple tiré d'un TD

x_i	1	2	3	4
y_i	0	0	0	6

TABLEAU 2.3.1 – Série 1

Méthode de Newton :

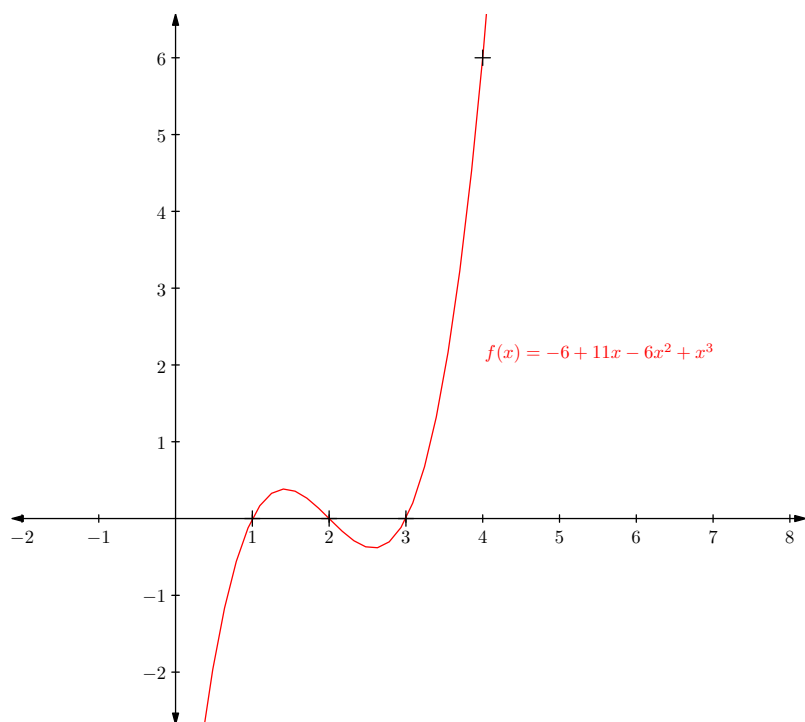
$$P(x) = -6.00 + 11.00 \cdot x - 6.00 \cdot x^2 + 1.00 \cdot x^3$$

Erreur : 0

Méthode de Neuville :

$$P(x) = -6.00 + 11.00 \cdot x - 6.00 \cdot x^2 + 1.00 \cdot x^3$$

Erreur : 0



GRAPHIQUE 2.3.1 – Interpolations de Newton et Neuville – (Tableau 2.3.1)

2.3.2 Densité de l'eau en fonction de la température

x_i	0	2	4	6	8	10	12	14	16	18
y_i	0.999870	0.999970	1.000000	0.999970	0.999880	0.999730	0.999530	0.999530	0.998970	0.998460
x_i	20	22	24	26	28	30	32	34	36	38
y_i	0.998050	0.999751	0.997050	0.996500	0.996640	0.995330	0.994720	0.994720	0.993330	0.993260

TABLEAU 2.3.2 – Mesures

Méthode de Newton :

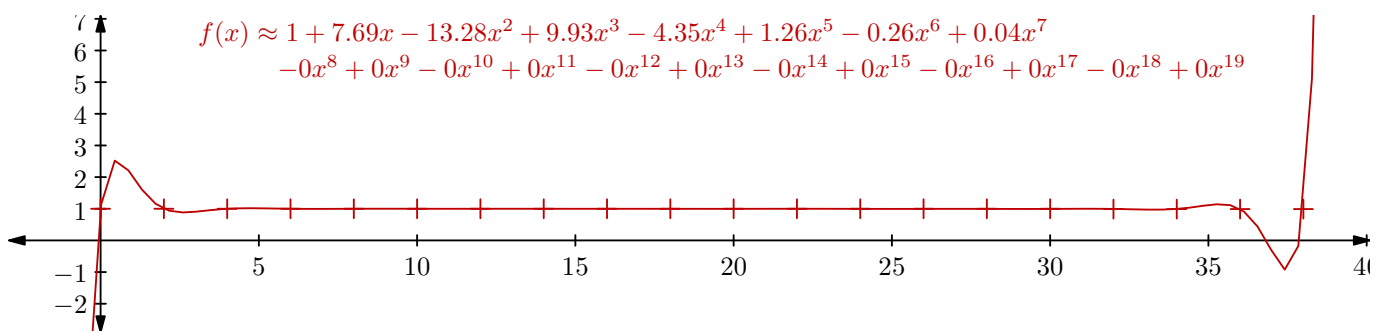
$$P(x) \approx 0.999870 + 7.693711 \cdot x - 13.276666 \cdot x^2 + 9.932303 \cdot x^3 - 4.345460 \cdot x^4 + 1.259124 \cdot x^5 - 0.258585 \cdot x^6 + 0.039240 \cdot x^7 - 0.004520 \cdot x^8 + 0.000402 \cdot x^9 - 0.000028 \cdot x^{10} + 0.000002 \cdot x^{11} - 0.000000 \cdot x^{12} + 0.000000 \cdot x^{13} - 0.000000 \cdot x^{14} + 0.000000 \cdot x^{15} - 0.000000 \cdot x^{16} + 0.000000 \cdot x^{17} - 0.000000 \cdot x^{18} + 0.000000 \cdot x^{19}$$

Erreur : 0.000002166566117323

Méthode de Neville :

$$P(x) \approx 0.999870 + 7.693711 \cdot x - 13.276666 \cdot x^2 + 9.932303 \cdot x^3 - 4.345460 \cdot x^4 + 1.259124 \cdot x^5 - 0.258585 \cdot x^6 + 0.039240 \cdot x^7 - 0.004520 \cdot x^8 + 0.000402 \cdot x^9 - 0.000028 \cdot x^{10} + 0.000002 \cdot x^{11} - 0.000000 \cdot x^{12} + 0.000000 \cdot x^{13} - 0.000000 \cdot x^{14} + 0.000000 \cdot x^{15} - 0.000000 \cdot x^{16} + 0.000000 \cdot x^{17} - 0.000000 \cdot x^{18} + 0.000000 \cdot x^{19}$$

Erreur : 0.000028505775100296



GRAPHIQUE 2.3.2 – Interpolation de Newton et Neville – (Tableau 2.3.2)

2.3.3 3 séries

x_i	10	8	13	9	11	14	6	4	12	7	5
$y_i^{(1)}$	9.14	8.14	8.74	8.77	9.26	8.10	6.13	3.10	9.13	7.26	4.74
$y_i^{(2)}$	7.46	6.77	12.74	7.11	7.81	8.84	6.08	5.39	8.15	6.42	5.73
$y_i^{(3)}$	6.58	5.76	7.71	8.84	8.47	7.04	5.25	12.50	5.56	7.91	6.89

TABLEAU 2.3.3 – Trois séries S1, S2, S3

Série 1 :

Méthode de Newton :

$$P(x) \approx -229.550000 + 299.165750 \cdot x - 173.107636 \cdot x^2 + 58.546955 \cdot x^3 - 12.731862 \cdot x^4 + 1.859906 \cdot x^5 - 0.184968 \cdot x^6 + 0.012375 \cdot x^7 - 0.000533 \cdot x^8 + 0.000013 \cdot x^9 - 0.000000 \cdot x^{10}$$

Erreur : 0.000000000016217532

Méthode de Neuville :

$$P(x) \approx -229.550000 + 299.165750 \cdot x - 173.107636 \cdot x^2 + 58.546955 \cdot x^3 - 12.731862 \cdot x^4 + 1.859906 \cdot x^5 - 0.184968 \cdot x^6 + 0.012375 \cdot x^7 - 0.000533 \cdot x^8 + 0.000013 \cdot x^9 - 0.000000 \cdot x^{10}$$

Erreur : 0.000000000012119518

Série 2 :

Méthode de Newton :

$$P(x) \approx -12345.190000 + 16608.066492 \cdot x - 9870.941498 \cdot x^2 + 3416.593892 \cdot x^3 - 763.094009 \cdot x^4 + 114.979985 \cdot x^5 - 11.842442 \cdot x^6 + 0.823658 \cdot x^7 - 0.037039 \cdot x^8 + 0.000973 \cdot x^9 - 0.000011 \cdot x^{10}$$

Erreur : 0.0000000000774325735

Méthode de Neuville :

$$P(x) \approx -12345.190000 + 16608.066492 \cdot x - 9870.941498 \cdot x^2 + 3416.593892 \cdot x^3 - 763.094009 \cdot x^4 + 114.979985 \cdot x^5 - 11.842442 \cdot x^6 + 0.823658 \cdot x^7 - 0.037039 \cdot x^8 + 0.000973 \cdot x^9 - 0.000011 \cdot x^{10}$$

Erreur : 0.000000001033081661

Série 3 :

Méthode de Newton :

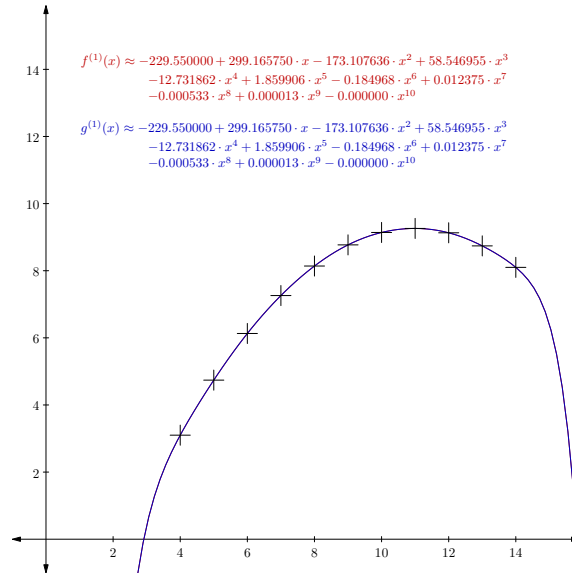
$$P(x) \approx -568559.640000 + 739678.381270 \cdot x - 424130.450858 \cdot x^2 + 141275.523224 \cdot x^3 - 30298.693006 \cdot x^4 + 4375.222059 \cdot x^5 - 431.155992 \cdot x^6 + 28.652640 \cdot x^7 - 1.229803 \cdot x^8 + 0.030806 \cdot x^9 - 0.000342 \cdot x^{10}$$

Erreur : 0.000000081067879843

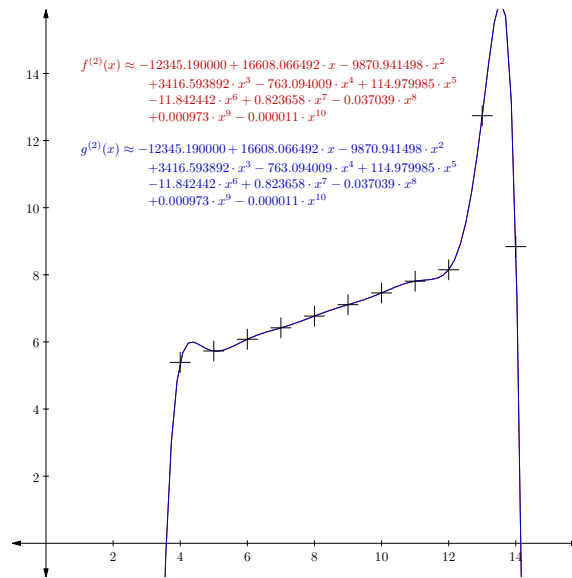
Méthode de Neuville :

$$P(x) \approx -568559.640000 + 739678.381270 \cdot x - 424130.450858 \cdot x^2 + 141275.523224 \cdot x^3 - 30298.693006 \cdot x^4 + 4375.222059 \cdot x^5 - 431.155992 \cdot x^6 + 28.652640 \cdot x^7 - 1.229803 \cdot x^8 + 0.030806 \cdot x^9 - 0.000342 \cdot x^{10}$$

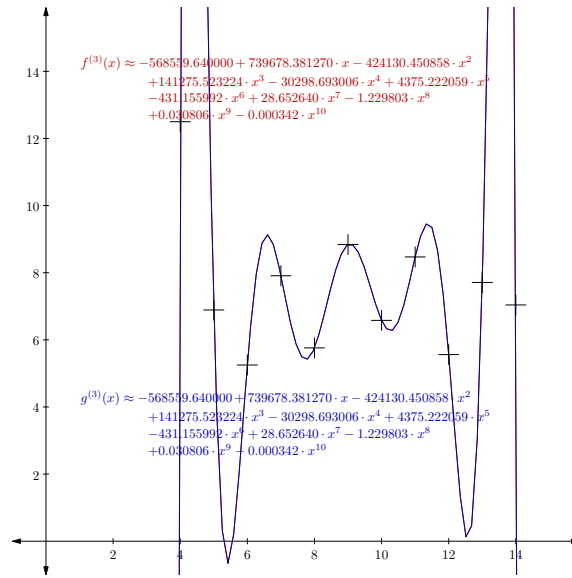
Erreur : 0.000000035876804073



GRAPHIQUE 2.3.3 – Interpolations de Newton et Neville – Série 1 (Tableau 2.3.3)



GRAPHIQUE 2.3.4 – Interpolations de Newton et Neville – Série 2 (Tableau 2.3.3)



GRAPHIQUE 2.3.5 – Interpolations de Newton et Neville – Série 3 (Tableau 2.3.3)

2.3.4 Dépenses et Revenus

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

TABLEAU 2.3.4 – Série non triée

x_i (R)	752	855	828	734	809	610	582	492	569	462	643	862	524	679
y_i (D)	85	83	82	79	77	81	83	81	81	80	84	84	82	80

TABLEAU 2.3.5 – Série triée : Partie Basse

x_i (R)	902	918	871	875	921	907	894
y_i (D)	226	260	162	186	281	243	223

TABLEAU 2.3.6 – Série triée : Partie Haute

Partie Basse :

Méthode de Newton :

$$P(x) \approx 73581192209.962601 - 1459287367.863513 \cdot x + 13300351.970502 \cdot x^2 - 73765.523297 \cdot x^3 + 277.759281 \cdot x^4 - 0.749863 \cdot x^5 + 0.001493 \cdot x^6 - 0.000002 \cdot x^7 + 0.000000 \cdot x^8 - 0.000000 \cdot x^9 + 0.000000 \cdot x^{10} - 0.000000 \cdot x^{11} + 0.000000 \cdot x^{12} - 0.000000 \cdot x^{13}$$

Erreur : 0.018460432587224723

Méthode de Neuville :

$$P(x) \approx 73581192209.952133 - 1459287367.863373 \cdot x + 13300351.970501 \cdot x^2 - 73765.523297 \cdot x^3 + 277.759281 \cdot x^4 - 0.749863 \cdot x^5 + 0.001493 \cdot x^6 - 0.000002 \cdot x^7 + 0.000000 \cdot x^8 - 0.000000 \cdot x^9 + 0.000000 \cdot x^{10} - 0.000000 \cdot x^{11} + 0.000000 \cdot x^{12} - 0.000000 \cdot x^{13}$$

Erreur : 0.192499821369502971

Partie Haute :

Méthode de Newton :

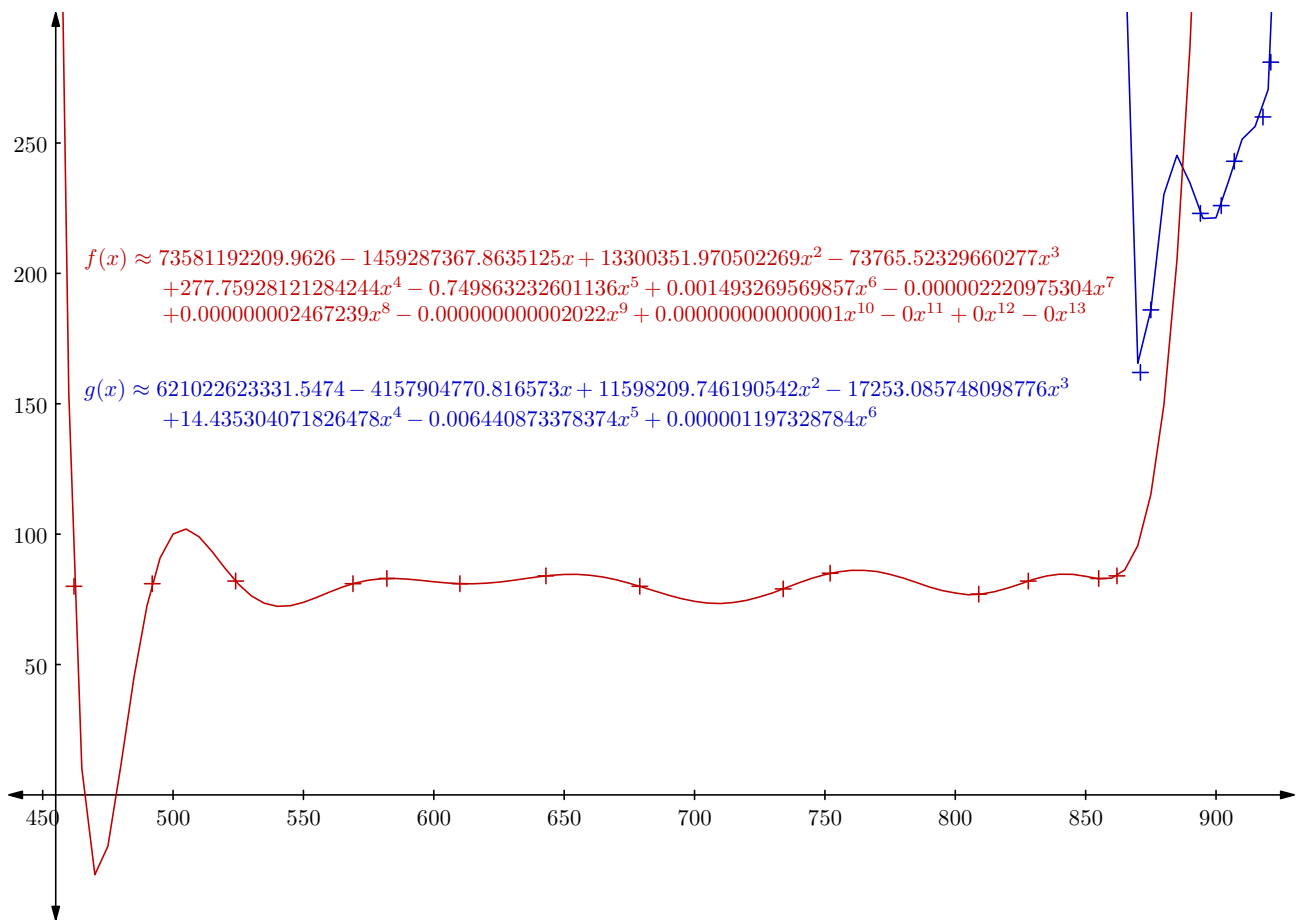
$$P(x) \approx 621022623331.547363 - 4157904770.816573 \cdot x + 11598209.746191 \cdot x^2 - 17253.085748 \cdot x^3 + 14.435304 \cdot x^4 - 0.006441 \cdot x^5 + 0.000001 \cdot x^6$$

Erreur : 0.000429349286215646

Méthode de Neuville :

$$P(x) \approx 621022623331.548340 - 4157904770.816572 \cdot x + 11598209.746191 \cdot x^2 - 17253.085748 \cdot x^3 + 14.435304 \cdot x^4 - 0.006441 \cdot x^5 + 0.000001 \cdot x^6$$

Erreur : 0.002443850040435791



GRAPHIQUE 2.3.6 – Interpolation de Newton et Neville – (Tableaux 2.3.5 & 2.3.6)

2.4 Comparaison

3 Approximation

Contrairement aux interpolations, les équations obtenues ne passent pas forcément par les N points. On cherche uniquement à minimiser la distance moyenne entre les N points mesurés et la courbe. Cette méthode est appelée *méthode des moindres carrés*.

3.1 Régression linéaire

3.1.1 Présentation

La régression linéaire est un cas particulier de la méthode des moindres carrés. L'équation recherchée ici est une droite d'équation $y = a_0 + a_1 \cdot x$. On obtient alors le système :

$$\begin{bmatrix} \sum_{i=1}^N x_i^0 & \sum_{i=1}^N x_i^1 \\ \sum_{i=1}^N x_i^1 & \sum_{i=1}^N x_i^2 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^N y_i x_i^0 \\ \sum_{i=1}^N y_i x_i^1 \end{bmatrix}$$

On a enfin une expression de a_0 et a_1 :

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

$$a_0 = \bar{y} - a_1 \cdot \bar{x}$$

3.1.2 Programme

```
1 | #include <stdio.h>
2 | #include <stdlib.h>
3 | #include <string.h>
4 | #include <math.h>
5 | #include "polynome.h"
6 |
7 | void mapping(double** from, double** to, int n, char* fn);
8 |
9 | double moyenneElements(double** tab, int l, int n)
10 | {
11 |     double resultat = 0.;
12 |     double cpt = 0.;
13 |     int i;
14 |     for(i=0; i<n; i++)
15 |     {
16 |         resultat = resultat + tab[l][i];
17 |         cpt = cpt + 1.;
18 |     }
19 |     resultat = resultat/cpt;
20 |     return resultat;
21 | }
22 |
23 | double moyenneElementsCarres(double** tab, int l, int n)
24 | {
25 |     double resultat = 0;
```



```

26     double cpt = 0;
27     int i;
28     for(i=0;i<n;i++)
29     {
30         resultat = resultat + pow(tab[1][i],2);
31         cpt = cpt + 1;
32     }
33     resultat = resultat/cpt;
34     return resultat;
35 }
36
37 double moyenneProduitElements(double** tab, int n)
38 {
39     double resultat = 0;
40     double cpt = 0;
41     int i;
42     for(i=0;i<n;i++)
43     {
44         resultat = resultat + tab[0][i]*tab[1][i];
45         cpt = cpt + 1;
46     }
47     resultat = resultat/cpt;
48     return resultat;
49 }
50
51 reglinD(double** tab, int n)
52 {
53     double a0 = 0;
54     double a1 = 0;
55     double xb, yb, xcb, xyb; // b pour barre et c pour carre
56     printf("Nous cherchons le Polynome de degré 1 sous la forme a0 + a1*x.\n");
57
58     //calculs
59     xb = moyenneElements(tab,0,n);
60     yb = moyenneElements(tab,1,n);
61     xcb = moyenneElementsCarres(tab,0,n);
62     xyb = moyenneProduitElements(tab,n);
63
64     a1 = (xyb-xb*yb)/(xcb-pow(xb,2));
65     a0 = yb-xb*a1;
66
67     // creation et affichage
68     polynome *P = creerPoly(2,"valeur",a0,a1);
69     menuAffichage(P);
70
71     //statistiques
72     ecartPoly(tab,n,P);
73     printf("\n");
74
75     //libération mémoire
76     free(P->poln);
77     free(P);
78 }

```

FIGURE 3.1 – Code : reglin.c

3.2 Ajustement exponentiel

3.2.1 Présentation

On doit trouver une équation sous la forme $y = ce^{dx}$.

On a donc $\ln(y) = \ln(c) + dx \ln(e) \Leftrightarrow \ln(y) = \ln(c) + dx$.

Cela revient à effectuer une régression linéaire sous la forme :

$Y = a_0 + a_1x$ avec $Y = \ln(y)$, $a_0 = \ln(c)$ et $a_1 = d$.

Finalement, on obtient $c = e^{a_0}$ et $d = a_1$.

3.2.2 Programme

```
80 reglinE(double** tab, int n) //y=c(e^(dx)) <=> ln(y)=ln(c)+xd => c=e^(a0) & d=a1
81 {
82     int i;
83     double c = 0;
84     double d = 0;
85     double a0 = 0;
86     double a1 = 0;
87     double xb, yb, xcb, xyb; // b pour barre et c pour carre
88     double** t = (double**) malloc(2*sizeof(double*)); // contiendra le mapping de tab
89     for(i=0;i<2;i++)
90     {
91         t[i] = (double*) malloc (n*sizeof(double));
92     }
93     printf("Nous cherchons une approximation sous la forme c*(e^(d*x)).\n");
94
95     //calculs
96     mapping(tab, t, n, "exponentielle");
97     xb = moyenneElements(t,0,n);
98     yb = moyenneElements(t,1,n);
99     xcb = moyenneElementsCarres(t,0,n);
100    xyb = moyenneProduitElements(t,n);
101
102    a1 = (xyb-xb*yb)/(xcb-pow(xb,2.));
103    a0 = yb-xb*a1;
104    d = a1;
105    c = exp(a0);
106
107    //affichage
108    printf("P(x) = %20.18f*exp(%20.18f*x)\n",c,d);
109
110    //statistiques
111    ecartExpo(tab,n,c,d);
112    printf("\n");
113
114    //libération mémoire
115    for (i=0; i<2; i++)
116    {
117        free(t[i]);
118    }
119    free(t);
120 }
```

FIGURE 3.2 – Code : reglin.c

3.3 Ajustement de type “puissance”

3.3.1 Présentation

On doit trouver une équation sous la forme $y = ax^b$.

On a donc $\ln(y) = \ln(a) + b \ln(x)$.

Cela revient à effectuer une régression linéaire sous la forme :

$Y = a_0 + a_1 X$ avec $Y = \ln(y)$, $X = \ln(x)$, $a_0 = \ln(a)$, et $a_1 = b$.

Finalement, on obtient $a = e^{a_0}$ et $b = a_1$.

3.3.2 Programme

```
122 reglinP(double ** tab, int n) //y=a(x^b) <=> ln(y)=ln(a)+b*ln(x) => a=e^(a0) & b=a1
123 {
124     int i;
125     double a = 0.; double b = 0.; double a0 = 0.; double a1 = 0.;
126     double xb, yb, xcb, xyb; // b pour barre et c pour carre
127     double** t = (double**) malloc(2*sizeof(double*)); // contiendra le mapping de tab
128     for(i=0;i<2;i++)
129     {
130         t[i] = (double*) malloc (n*sizeof(double));
131     }
132     printf("Nous cherchons une approximation sous la forme a*(x^(b)).\n");
133
134     //calculs
135     mapping(tab, t, n, "puissance");
136     xb = moyenneElements(t,0,n);
137     yb = moyenneElements(t,1,n);
138     xcb = moyenneElementsCarres(t,0,n);
139     xyb = moyenneProduitElements(t,n);
140
141     a1 = (xyb-xb*yb)/(xcb-pow(xb,2));
142     a0 = yb-xb*a1;
143     b = a1;
144     a = exp(a0);
145
146     //affichage
147     printf("P(x) = %20.18f*x^(%20.18f)\n",a,b);
148
149     //statistiques
150     ecartPui(tab,n,a,b);
151     printf("\n");
152
153     //libération mémoire
154     for (i=0; i<2; i++)
155     {
156         free(t[i]);
157     }
158     free(t);
159 }
160
161 void mapping(double** from, double** to, int n, char* fn)
162 {
163     int i, j;
164     if (strcmp(fn,"exponentielle")==0)
165     {
166         for (j=0; j<n; j++){to[0][j]=from[0][j];}
167         for (j=0; j<n; j++){to[1][j]=log(from[1][j]);}
168     }
169     else if (strcmp(fn,"puissance")==0)
170     {
171         for (i=0; i<2; i++){for (j=0; j<n; j++){to[i][j]=log(from[i][j]);}}
172     }
173 }
```

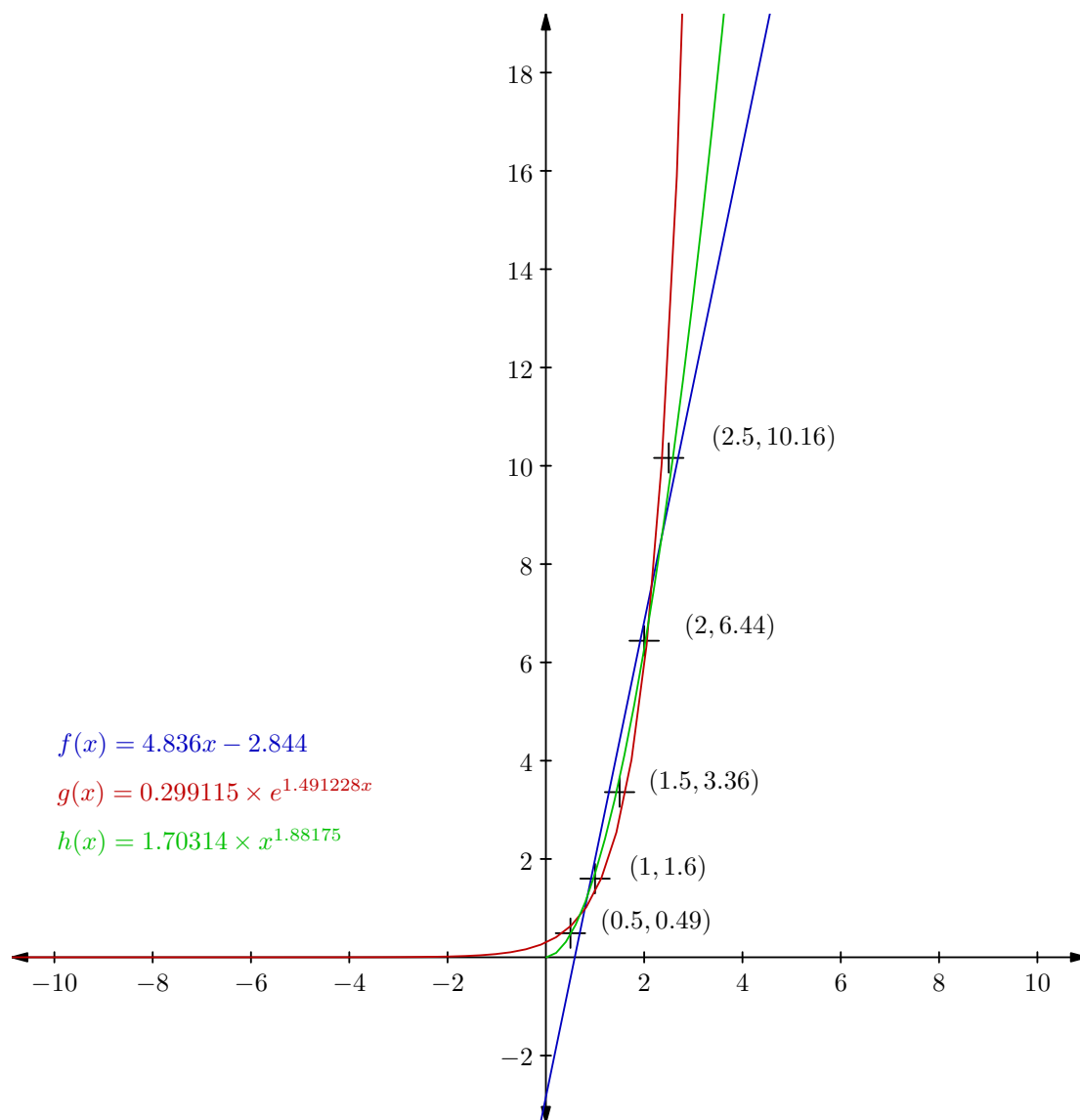
FIGURE 3.3 – Code : reglin.c

3.4 Résultats de tests

3.4.1 Exemple tiré d'un TD

x_i	0.5	1	1.5	2	2.5
y_i	0.49	1.6	3.36	6.44	10.16

TABLEAU 3.4.1 – Série 1

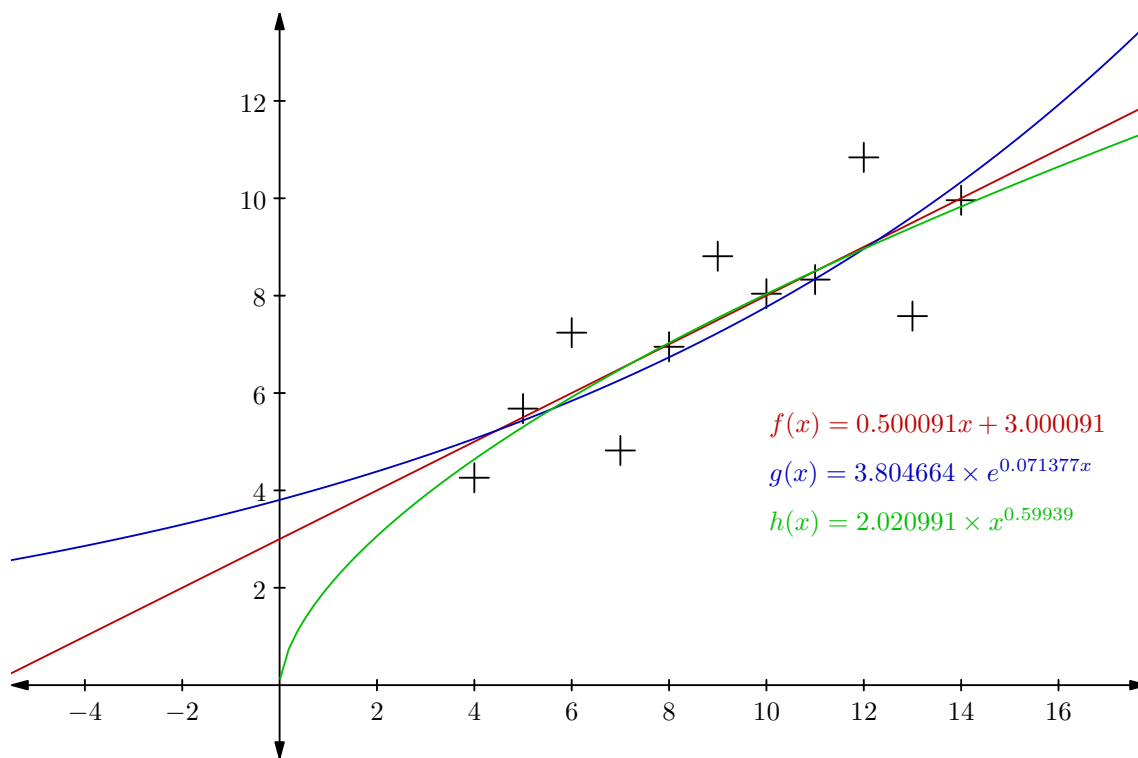


GRAPHIQUE 3.4.1 – Régression linéaire – (Tableau 3.4.1)

3.4.2 Série d'Anscombe

x_i	10	8	13	9	11	14	6	4	12	7	5
$y_i^{(A)}$	8.04	6.95	7.58	8.81	8.33	9.96	7.24	4.26	10.84	4.82	5.68

TABLEAU 3.4.2 – Série due à Anscombe



GRAPHIQUE 3.4.2 – Régression linéaire – (Tableau 3.4.2)

3.4.3 3 séries

x_i	10	8	13	9	11	14	6	4	12	7	5
$y_i^{(1)}$	9.14	8.14	8.74	8.77	9.26	8.10	6.13	3.10	9.13	7.26	4.74
$y_i^{(2)}$	7.46	6.77	12.74	7.11	7.81	8.84	6.08	5.39	8.15	6.42	5.73
$y_i^{(3)}$	6.58	5.76	7.71	8.84	8.47	7.04	5.25	12.50	5.56	7.91	6.89
$y_i^{(A)}$	8.04	6.95	7.58	8.81	8.33	9.96	7.24	4.26	10.84	4.82	5.68

TABLEAU 3.4.3 – 3 séries $S^{(1)}$, $S^{(2)}$ et $S^{(3)}$ comparées à Anscombe

Série (1) :

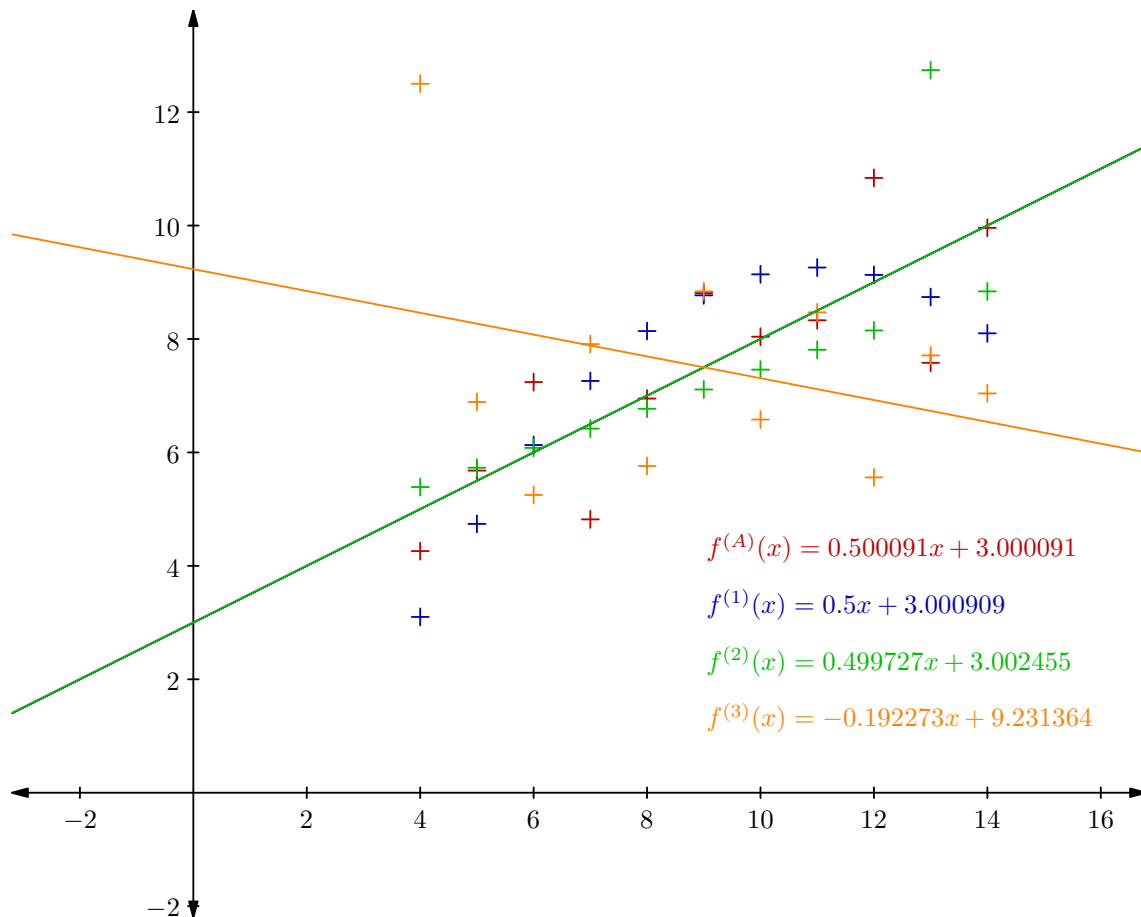
Regression linéaire par une droite : $P(x) = 3.000909 + 0.500000 \cdot x$

Erreur moyenne : 0.967934

Série (2) :

Regression linéaire par une droite : $P(x) = 3.002455 + 0.499727 \cdot x$

Erreur moyenne : 0.715967



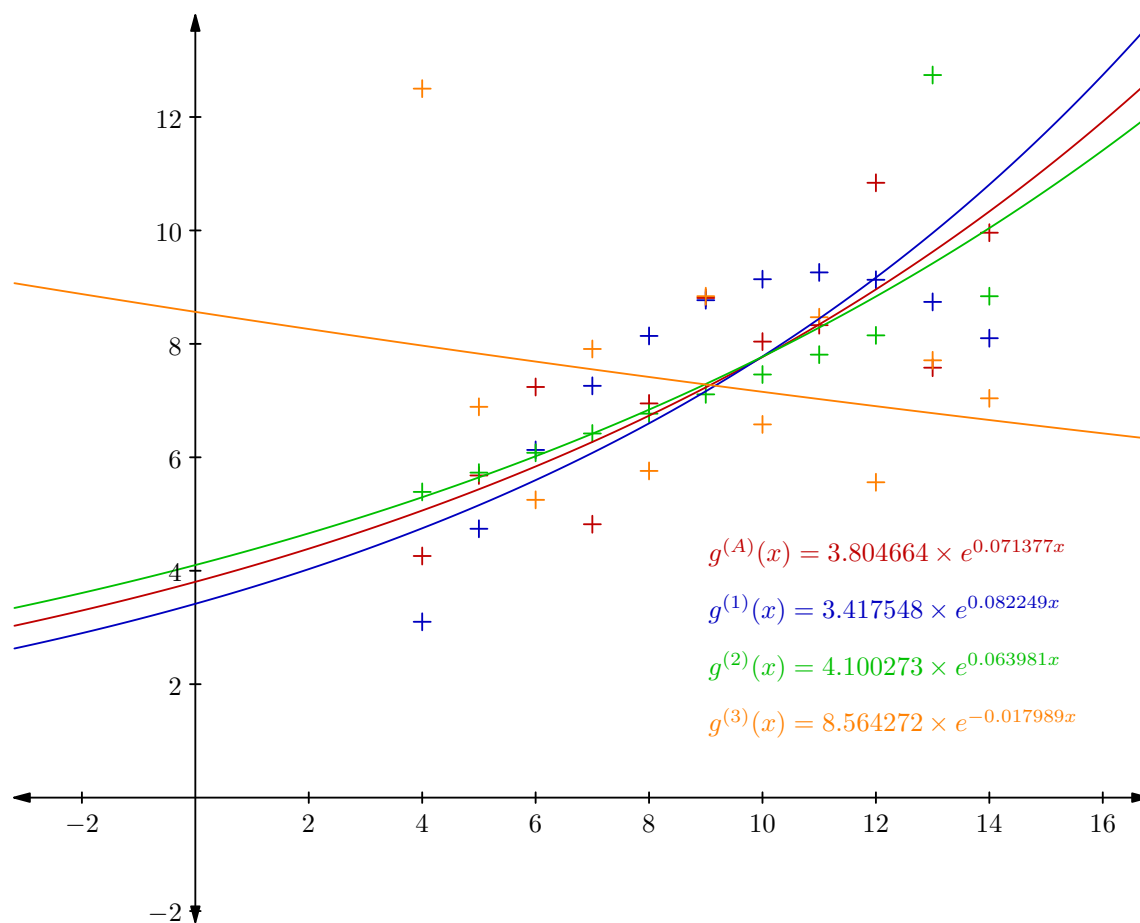
GRAPHIQUE 3.4.3 – Régression linéaire – (Tableau 3.4.3)

Regression linéaire par une exponentielle : $P(x) = 3.417548 \cdot \exp(0.082249 \cdot x)$

Erreur moyenne : 1.187786

Regression linéaire par une exponentielle : $P(x) = 4.100273 \cdot \exp(0.063981 \cdot x)$

Erreur moyenne : 0.590601



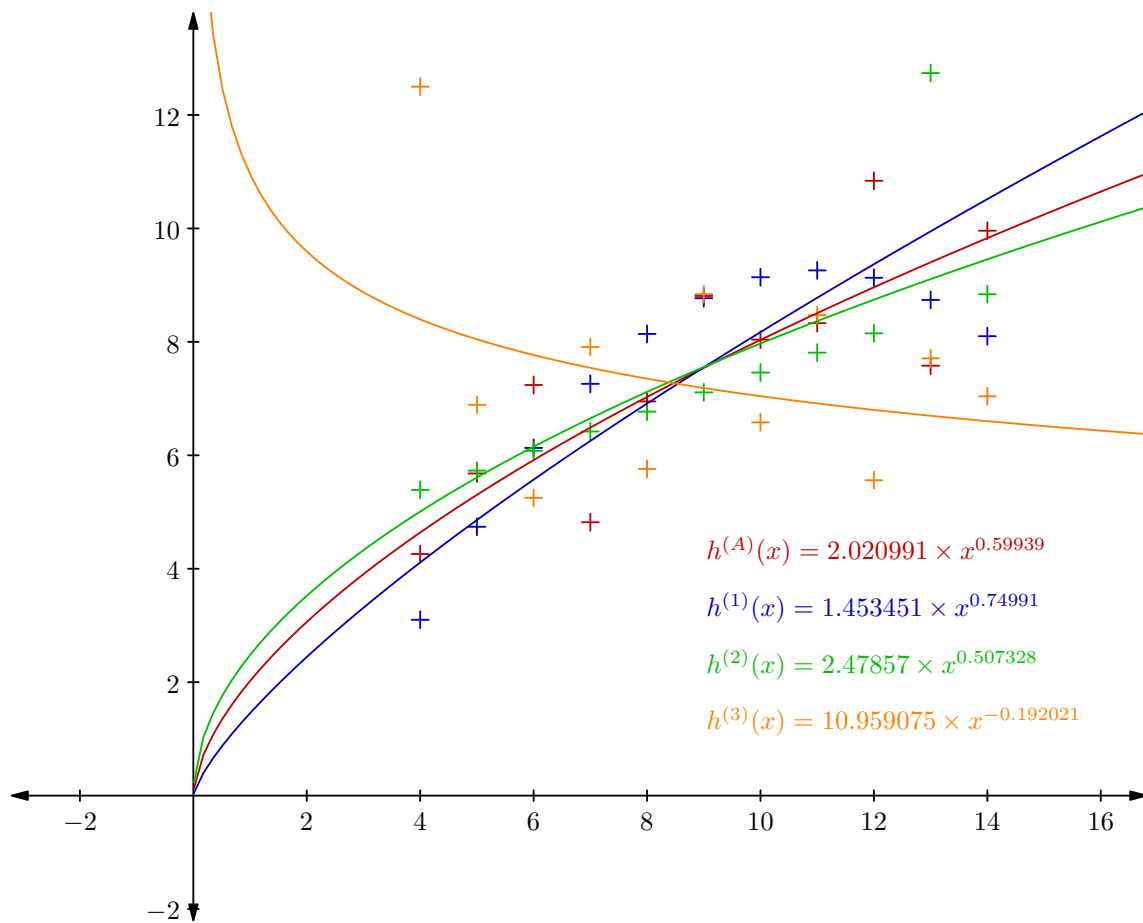
GRAPHIQUE 3.4.4 – Approximation par ajustement exponentiel – (Tableau 3.4.3)

Regression linéaire par une puissance : $P(x) = 1.453451 \cdot x^{0.749910}$

Erreur moyenne : 0.950634

Regression linéaire par une puissance : $P(x) = 2.478570 \cdot x^{0.507328}$

Erreur moyenne : 0.682932



GRAPHIQUE 3.4.5 – Approximation par ajustement “puissance” – (Tableau 3.4.3)

3.4.4 Dépenses mensuelles et revenus

x_i (R)	752	855	871	734	610	582	921	492	569	462	907
y_i (D)	85	83	162	79	81	83	281	81	81	80	243

TABLEAU 3.4.4 – Série 1

x_i (R)	643	862	524	679	902	918	828	875	809	894
y_i (D)	84	84	82	80	226	260	82	186	77	223

TABLEAU 3.4.5 – Série 2

Série 1 :

Regression linéaire par une droite : $P(x) = -98.368005 + 0.312192 \cdot x$

Erreur moyenne : 38.488186

Regression linéaire par une exponentielle : $P(x) = 24.011644 \cdot \exp(0.002124 \cdot x)$

Erreur moyenne : 33.486916

Regression linéaire par une puissance : $P(x) = 0.015258 \cdot x^{1.356482}$

Erreur moyenne : 36.660388

Série 2 :

Regression linéaire par une droite : $P(x) = -164.266162 + 0.381480 \cdot x$

Erreur moyenne : 46.455702

Regression linéaire par une exponentielle : $P(x) = 14.780629 \cdot \exp(0.002657 \cdot x)$

Erreur moyenne : 45.099159

Regression linéaire par une puissance : $P(x) = 0.000785 \cdot x^{1.793989}$

Erreur moyenne : 47.682118

3.4.5 Série chronologique avec accroissement exponentiel

x_i	88	89	90	91	92	93	94	95	96	97
y_i	5.89	6.77	7.97	9.11	10.56	12.27	13.92	15.72	17.91	22.13

TABLEAU 3.4.6 – Série

3.4.6 Vérification de la loi de Pareto

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

TABLEAU 3.4.7 – Relation entre revenu et nombre de personnes ayant un revenu supérieur

4 Conclusion

5 Annexe

5.1 Menu principal

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <math.h>
5  #include "neuville.h"
6  #include "newton.h"
7  #include "polynome.h"
8  #include "reglin.h"
9  #include "useful.h"
10
11 int main (int argc, char ** argv)
12 {
13     int n, j;
14     int i=1;
15     char c;
16     while (i!=0)
17     {
18         printf("Menu principal : Interpolation et Approximation\n\n");
19         n=0;
20         while (n<2) //minimum 2 points
21         {
22             printf("Entrez n le nombre de points : ");
23             scanf("%d", &n);
24         }
25         double** tab= (double**) malloc(2*sizeof(double*));
26         for (i=0; i<2; i++)
27         {
28             tab[i]=(double*) malloc(n*sizeof(double));
29         }
30         for (i=0; i<n; i++)
31         {
32             printf("Entrez x[%d] : ", i+1);
33             scanf("%lf", &tab[0][i]);
34             printf("Entrez y[%d] : ", i+1);
35             scanf("%lf", &tab[1][i]);
36         }
37         //fonction temporaire, utile uniquement pour la rédaction du rapport
38         convertTabtoLatex(tab,n,1);
39         i=1;
40         while (i!=0 && i!=9)
41         {
42             clear(); //nettoyage écran
43             printf("Tableau de valeurs :\n");
44             for (i=0; i<2; i++)
45             {
46                 for (j=0; j<n; j++)
47                 {
48                     if (0==j && 0==i)
49                     {
50                         printf(" x ");
51                     }
52                     else if (0==j && 1==i)
53                     {
```

```

54     printf(" y ");
55     }
56     printf("| %.5f ", tab[i][j]);
57     }
58     printf("\n");
59 }
60 printf("\nQuelle résolution utiliser ?\n");
61 printf("1- Newton\n");
62 printf("2- Neuville\n");
63 printf("3- Régression Linéaire\n");
64 printf("4- Approximation par une fonction exponentielle\n");
65 printf("5- Approximation par une fonction puissance\n");
66 printf("9- Nouvelle série de points (Menu principal)\n");
67 printf("0- Quitter\n");
68 printf("Votre choix : ");
69 scanf("%d", &i);
70 c='0';
71 cleanBuffer(); //vidage buffer
72 switch (i)
73 {
74     case 1:
75         printf("Résolution par Newton ... \n");
76         newton(tab,n);
77         hitToContinue();
78         break;
79     case 2:
80         printf("Résolution par Neuville ... \n");
81         neuville(tab,n);
82         hitToContinue();
83         break;
84     case 3:
85         printf("Résolution par Régression linéaire ... \n");
86         reglinD(tab,n);
87         hitToContinue();
88         break;
89     case 4:
90         printf("Résolution par Approximation par une fonction exponentielle... \n");
91         reglinE(tab,n);
92         hitToContinue();
93         break;
94     case 5:
95         printf("Résolution par Approximation par une fonction puissance... \n");
96         reglinP(tab,n);
97         hitToContinue();
98         break;
99     }
100     printf("\n\n");
101 }
102
103 //libération mémoire
104 for (j=0; j<2; j++)
105 {
106     free(tab[j]);
107 }
108 free(tab);
109 clear(); //nettoyage écran
110 }
111 return 0;
112 }

```

FIGURE 5.1 – Code : main.c

5.2 Fonctions associées au calcul polynômial

```

1  #ifndef POLYNOME__H
2  #define POLYNOME__H
3
4  typedef struct polynome
5  {

```

```

6   int d; //degree
7   double* poln; //coefficients
8 } polynome;
9
10  //fonctions sortie LaTeX
11 void convertTabtoLatex(double** tab, int n, int m); //sortie du tableau en LaTeX
12 void menuAffichage(polynome* P); //choix entre LaTeX et terminal
13 void afficherPoly(polynome* P, char* mode, ...); //mode:"console"|"latex"; console->terminal; latex->fichier
    (FILE* opt) format maths LaTeX.
14
15  //fonctions de manipulation des poly
16 polynome* creerPoly(int c, char* mode, ...); //c: nbre coefs; mode: "tableau"|"valeur"; mode tableau -> paramè
    tre optionnel : tableau des coefs.
17 void redimensionnerPoly(polynome* P1); // Enleve les 0 inutiles dans le poly.
18 polynome* addPoly(polynome* P1, polynome* P2); // addition de 2 poly entre eux.
19 polynome* mulSPoly(double s, polynome* P1); // multiplication d'un poly par un scalaire.
20 polynome* mulPoly(polynome* P1, polynome* P2); // multiplication de 2 poly entre eux.
21
22  // fonctions de calcul d'images de fonctions
23 double imagePoly(polynome* P, double x); // La fonction calcule l'image de x par un poly.
24 double imageExpo(double c, double d, double x); // La fonction calcule l'image de x par une exponentielle de
    la forme  $f(x)=c*\exp(dx)$ .
25 double imagePui(double a, double b, double x); // La fonction calcule l'image de x par une fonction puissance
    de la forme  $f(x)=a*x^b$ .
26
27  //fonctions de statistiques
28 void ecartPoly(double** tab, int n, polynome* P);
29 void ecartExpo(double** tab, int n, double c, double d);
30 void ecartPui(double** tab, int n, double a, double b);
31
32 #endif

```

FIGURE 5.2 – Code : polynome.h

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include <stdarg.h>
5  #include <string.h>
6  #include "polynome.h"
7
8  polynome* creerPoly(int c, char* mode, ...)
9  {
10     int i;
11     polynome* P=(polynome*) malloc(sizeof(polynome));
12     P->d = c-1;
13     P->poln = (double*) malloc(c*sizeof(double));
14     if ((strcmp(mode, "valeur"))==0) //si mode = "valeur"
15     {
16         va_list ap;
17         va_start(ap, mode);
18         for (i=0; i<c; i++)
19         {
20             P->poln[i]=va_arg(ap, double);
21         }
22         va_end(ap);
23     }
24     else //si mode= "tableau"
25     {
26         va_list ap;
27         va_start(ap, mode);
28         double* tmp=va_arg(ap, double*);
29         for (i=0; i<c; i++)
30         {
31             P->poln[i]=tmp[i];
32         }
33         free(tmp);
34         va_end(ap);
35     }
36     redimensionnerPoly(P);

```

```

37     return P;
38 }
39
40 void menuAffichage(polynome* P)
41 {
42     FILE* fichier=fopen("resultat", "a+");
43     int choix; // permet de choisir les options voulues
44     printf("Voulez-vous afficher le polynome dans la sortie standard (1-Oui *-Non) ? ");
45     scanf("%d",&choix);
46     if(choix ==1)
47     {
48         afficherPoly(P,"console");
49     }
50     else
51     {
52         afficherPoly(P,"console");
53         afficherPoly(P,"latex",fichier);
54     }
55     fclose(fichier);
56 }
57
58 void afficherPoly(polynome* P, char* mode, ...)
59 {
60     int i;
61     FILE* f;
62     int c = (P->d) +1 ;
63     if((strcmp(mode,"console")) == 0)
64     {
65         printf("P(x) = ");
66         for(i=0;i<c;i++)
67         {
68             if(P->poln[i] > 0)
69             {
70                 if(i != 0) { printf(" + "); }
71                 if(i == 0) { printf("%.200f",P->poln[i]); }
72                 else if(i == 1) { printf("%.200f * x",P->poln[i]); }
73                 else { printf("%.200f * x^%d", P->poln[i],i); }
74             }
75             if(P->poln[i] < 0)
76             {
77                 printf(" - ");
78                 if(i == 0) { printf("%.200f",-(P->poln[i])); }
79                 else if(i == 1) { printf("%.200f * x",-(P->poln[i])); }
80                 else { printf("%.200f * x^%d", -(P->poln[i]), i); }
81             }
82         }
83         printf("\n");
84     }
85     else
86     {
87         va_list ap;
88         va_start(ap,mode);
89         f = va_arg(ap,FILE*);
90         fprintf(f,"$P(x) \\approx ");
91         for(i=0; i<c; i++)
92         {
93             if(P->poln[i] > 0)
94             {
95                 if(i != 0) { fprintf(f, " + "); }
96                 if(i == 0) { fprintf(f, "%.6f",P->poln[i]); }
97                 else if(i == 1) { fprintf(f, "%.6f \\cdot x",P->poln[i]); }
98                 else { fprintf(f, "%.6f \\cdot x^{%d} ", P->poln[i],i); }
99             }
100             if(P->poln[i] < 0)
101             {
102                 if(i == 0) { fprintf(f, "-%.6f", -(P->poln[i])); }
103                 else if(i == 1) { fprintf(f, "-%.6f \\cdot x", -(P->poln[i])); }
104                 else { fprintf(f, "- %.6f \\cdot x^{%d} ", -(P->poln[i]), i); }
105             }
106         }
107         fprintf(f,"$\\n\\n");

```



```

108     va_end(ap);
109 }
110 }
111
112 void redimensionnerPoly(polynome* P1)
113 {
114     int degre=P1->d;
115     while((P1->poln[degre])==0)
116     {
117         degre--;
118     }
119     if(degre!=P1->d)
120     {
121         P1->d=degre;
122         P1->poln= (double*) realloc(P1->poln, (degre+1)*sizeof(double));
123     }
124 }
125
126 polynome* addPoly(polynome* P1, polynome* P2)
127 {
128     // Rappel : Deg(P1+P2) <= max(Deg(P1), Deg(P2))
129     int i;
130     polynome* P=(polynome*) malloc(sizeof(polynome));
131     if(P1->d > P2->d) // Deg(P1) > Deg(P2)
132     {
133         P->d = P1->d;
134         P->poln = (double*) malloc((1+P1->d)*sizeof(double));
135         for(i=0; i <= P2->d; i++)
136         {
137             P->poln[i] = P1->poln[i] + P2->poln[i];
138         }
139         for(i= P2->d +1; i<= P1->d; i++)
140         {
141             P->poln[i] = P1->poln[i];
142         }
143     }
144     else if (P1->d < P2->d) // Deg(P2) > Deg(P1)
145     {
146         P->d = P2->d;
147         P->poln = (double*) malloc((1+P2->d)*sizeof(double));
148         for(i=0; i <= P1->d; i++)
149         {
150             P->poln[i] = P1->poln[i] + P2->poln[i];
151         }
152         for(i= P1->d +1; i<= P2->d; i++)
153         {
154             P->poln[i] = P2->poln[i];
155         }
156     }
157     else // Deg(P2) = Deg(P1)
158     {
159         P->d = P2->d;
160         P->poln = (double*) malloc((1+P2->d)*sizeof(double));
161         for(i=0; i <= P1->d; i++)
162         {
163             P->poln[i] = P1->poln[i] + P2->poln[i];
164         }
165     }
166     redimensionnerPoly(P);
167     return P;
168 }
169
170 polynome* mulSPoly(double s, polynome* P1)
171 {
172     int i;
173     polynome* P=(polynome*) malloc(sizeof(polynome));
174     P->d = P1->d;
175     P->poln = (double*) malloc((1+P1->d)*sizeof(double));
176
177     for(i=0; i <= P1->d; i++)
178     {

```

```

179     P->poln[i] = s*P1->poln[i] ;
180 }
181 redimensionnerPoly(P); //redimensionnement nécessaire si le scalaire est nul.
182 return P;
183 }
184
185 polynome* mulPoly(polynome * P1, polynome* P2)
186 {
187     int i,j;
188     polynome* P=(polynome*)malloc(sizeof(polynome));
189     P->d = P1->d + P2->d ;
190     P->poln = (double*) malloc((1+P->d)*sizeof(double));
191
192     for(i=0; i<=P->d; i++)
193     {
194         P->poln[i] = 0;
195     }
196
197     for(i=0; i<=(P2->d); i++)
198     {
199         for(j=0; j<=(P1->d); j++)
200         {
201             P->poln[i+j] = P->poln[i+j] + (P2->poln[i])*(P1->poln[j]);
202         }
203     }
204     redimensionnerPoly(P);
205     return P;
206 }
207
208 double imagePoly(polynome* P, double x)
209 {
210     int i;
211     double res = P->poln[0];
212     for(i=1;i<=P->d;i++)
213     {
214         res = res + P->poln[i]*pow(x,i);
215     }
216     return res;
217 }
218
219 double imageExpo(double c, double d, double x)
220 {
221     double res = 0.;
222     res = c*exp(d*x);
223     return res;
224 }
225
226 double imagePui(double a, double b, double x)
227 {
228     double res = 0.;
229     res = a*pow(x,b);
230     return res;
231 }
232
233 void ecartPoly(double** tab, int n, polynome* P)
234 {
235     int i;
236     double moyecart= 0.;
237     for(i=0;i<n;i++)
238     {
239         moyecart = moyecart + fabs((imagePoly(P,tab[0][i])-tab[1][i]));
240     }
241     moyecart = moyecart/n;
242     printf("Erreur moyenne : %20.18f",moyecart);
243 }
244
245 void ecartExpo(double** tab, int n, double c, double d)
246 {
247     int i;
248     double moyecart= 0.;
249     for(i=0;i<n;i++)

```

```

250 | {
251 |     moyecart = moyecart + fabs((imageExpo(c,d,tab[0][i])-tab[1][i]));
252 | }
253 | moyecart = moyecart/n;
254 | printf("Erreur moyenne : %20.18f",moyecart);
255 | }
256 |
257 | void ecartPui(double** tab, int n, double a, double b)
258 | {
259 |     int i;
260 |     double moyecart= 0.;
261 |     for(i=0;i<n;i++)
262 |     {
263 |         moyecart = moyecart + fabs((imagePui(a,b,tab[0][i])-tab[1][i]));
264 |     }
265 |     moyecart = moyecart/n;
266 |     printf("Erreur moyenne : %20.18f",moyecart);
267 | }
268 |
269 | void convertTabtoLatex(double** tab, int n, int m)
270 | {
271 |     FILE* fichier = fopen("resultat", "a+");
272 |     int i,j;
273 |     //déclaration de l'environnement et de n+1 colonnes
274 |     fprintf(fichier, "\\begin{tabular}{|");
275 |     for(i=0;i<=n;i++)
276 |     {
277 |         fprintf(fichier, " c |");
278 |     }
279 |     fprintf(fichier, "}\\n \\hline \\n");
280 |
281 |     //remplissage des cases
282 |     for(i=0;i<2;i++)
283 |     {
284 |         if(i==0) {fprintf(fichier, "$x_{i}$ & ");}
285 |         if(i==1) {fprintf(fichier, "$y_{i}$ & ");}
286 |         for(j=0;j<n;j++)
287 |         {
288 |             if(j!=(n-1)) {fprintf(fichier, "$%f$ & ",tab[i][j]);}
289 |             else {fprintf(fichier, "$%f$ ",tab[i][j]);}
290 |         }
291 |         fprintf(fichier, "\\ \\n \\hline \\n");
292 |     }
293 |     fprintf(fichier, "\\end{tabular}\\n\\n");
294 |     fclose(fichier);
295 | }

```

FIGURE 5.3 – Code : polynome.c