

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	5
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	26
3.4.6	Vérification de la loi de Pareto	27
3.5	Comparaison	27
4	Conclusion	28
5	Annexe	29
5.1	Menu principal	29
5.2	Fonctions associées au calcul polynômial	31

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- Neville
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 31 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`.



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

FIGURE 2.2 – Code : neville.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 :

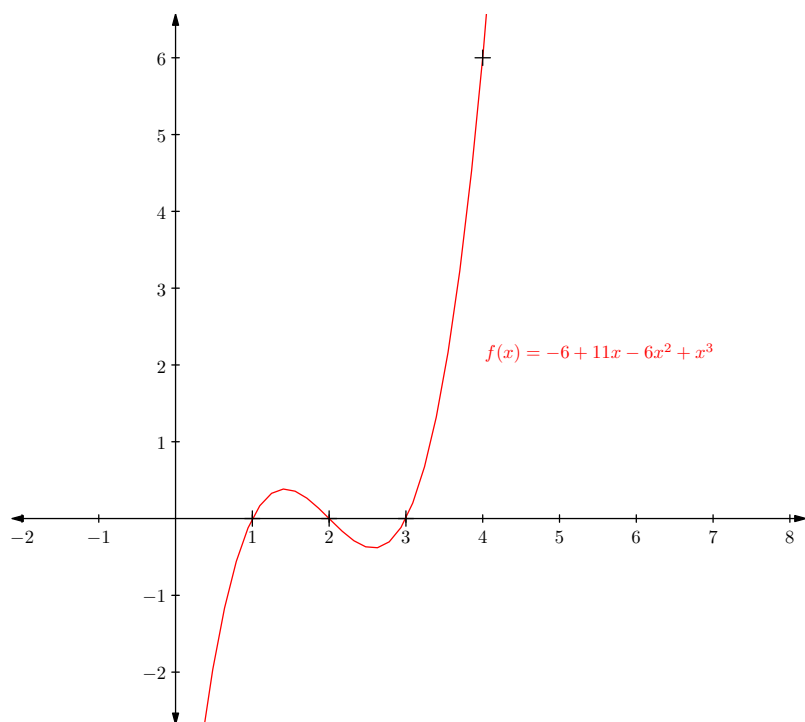
$$P(x) = -6.00 + 11.00 \cdot x - 6.00 \cdot x^2 + 1.00 \cdot x^3$$

Erreur : 0

Méthode de Neville :

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

Erreur : 0



GRAPHIQUE 2.3.1 – Interpolations de Newton et Neville – (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 :

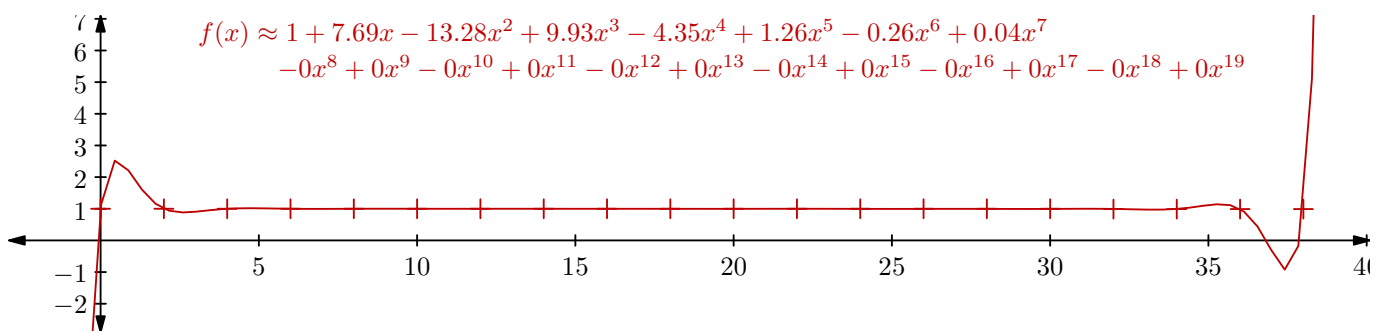
$$P(x) \approx 0.999870 + 7.693711 \cdot x - 13.276666 \cdot x^2 + 9.932303 \cdot x^3 - 4.345460 \cdot x^4 + 1.259124 \cdot x^5 - 0.258585 \cdot x^6 + 0.039240 \cdot x^7 - 0.004520 \cdot x^8 + 0.000402 \cdot x^9 - 0.000028 \cdot x^{10} + 0.000002 \cdot x^{11} - 0.000000 \cdot x^{12} + 0.000000 \cdot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

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 Neville :

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

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 Neville :

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

Erreur : 0.000000001033081661

Série 3 :

Méthode de Newton :

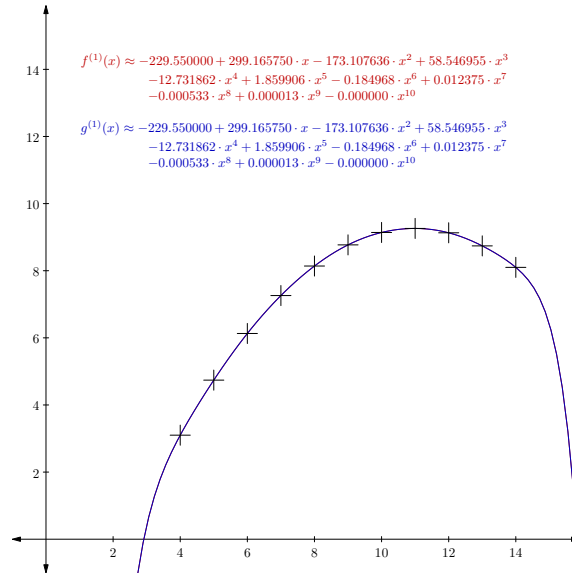
$$P(x) \approx -568559.640000 + 739678.381270 \cdot x - 424130.450858 \cdot x^2 + 141275.523224 \cdot x^3 - 30298.693006 \cdot x^4 + 4375.222059 \cdot x^5 - 431.155992 \cdot x^6 + 28.652640 \cdot x^7 - 1.229803 \cdot x^8 + 0.030806 \cdot x^9 - 0.000342 \cdot x^{10}$$

Erreur : 0.000000081067879843

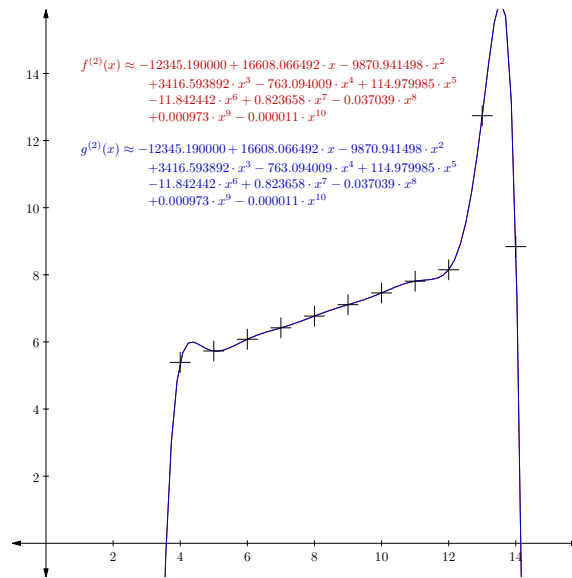
Méthode de Neville :

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

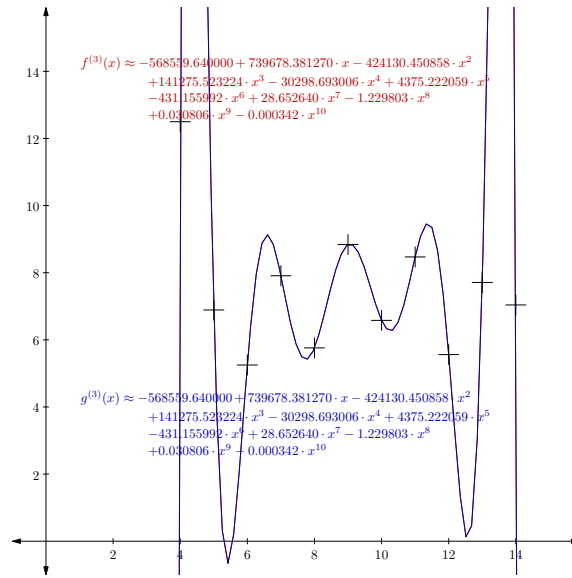
Erreur : 0.000000035876804073



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 Neville :

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

Erreur : 0.192499821369502971

Partie Haute :

Méthode de Newton :

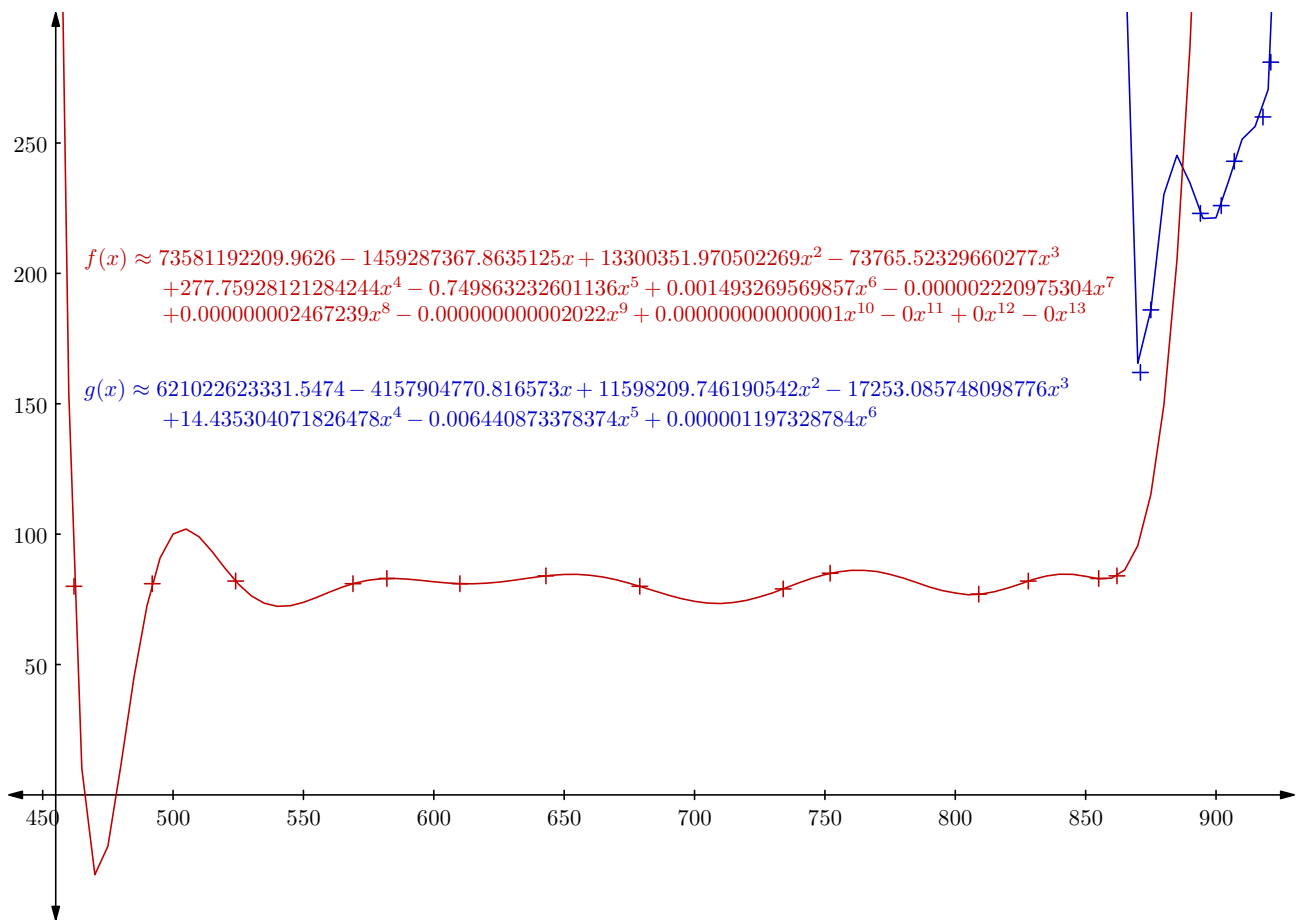
$$P(x) \approx 621022623331.547363 - 4157904770.816573 \cdot x + 11598209.746191 \cdot x^2 - 17253.085748 \cdot x^3 + 14.435304 \cdot x^4 - 0.006441 \cdot x^5 + 0.000001 \cdot x^6$$

Erreur : 0.000429349286215646

Méthode de Neville :

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

Erreur : 0.002443850040435791



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

2.4 Comparaison

Les méthodes d'interpolation permettent d'obtenir d'excellents résultats, avec des écarts particulièrement faibles, mais il est important de noter que ce sont des méthodes coûteuses en mémoire, et sensibles à la précision (raisons pour lesquelles nous avons choisi le compromis entre la précision et l'espace mémoire : un stockage double-précision 64 bits).

Pour N points, la méthode de Newton nécessitera $12N^2 + 32N + 56$ octets environ, et la méthode de Neville nécessitera $4N^3 + 6N^2 + 30N + 40$ octets environ. Il est donc évident que la quantité de mémoire croîtra très rapidement dès que le nombre de points augmentera.

La précision entre en compte dans les calculs car on recherche un polynôme dont le degré est fonction du nombre de points. Par conséquent, si les valeurs données en abscisse sont importantes et la précision des coefficients des grandes puissances trop faible, on aura des écarts très importants. Aussi nous avons choisi d'afficher les coefficients retournés par la machine de la manière la plus précise possible (plus d'une centaine de décimales) pour obtenir des résultats stables et des courbes correctes correspondant à l'écart calculé.

Les méthodes d'interpolation sont également coûteuses en calculs, elles passent par de nombreuses boucles et étapes intermédiaires, on a une complexité globale en $O(n^2)$ pour chacune de ces deux méthodes, elles sont donc peu efficaces en rapidité de calcul.

Finalement, on peut conclure que ces méthodes sont à restreindre à certains cadres d'utilisation, c'est-à-dire la recherche d'un polynôme absolument, ou bien la nécessité de passer par toute une liste de points tant qu'elle n'est pas trop conséquente. Sinon, il existe d'autres méthodes d'interpolation (interpolation polynomiale par segment / spline, ...), et des méthodes d'approximation qui permettent d'obtenir une équation dont la courbe s'approche des points considérés.

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*.

Note : On n'aura donc pas systématiquement à considérer une précision très importante pour comparer les tests, puisqu'il s'agit d'approximation, ce qui implique des écarts relativement importants.

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 pour a_0 et a_1 :

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

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

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) = %.18f*exp(%.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) = %.18f*x^(%.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 – Exemple

Régression linéaire :

$$P(x) = 4.836x - 2.844$$

Erreur : 0.732

Ajustement par une fonction exponentielle :

$$P(x) = 0.299115x \times e^{1.491228x}$$

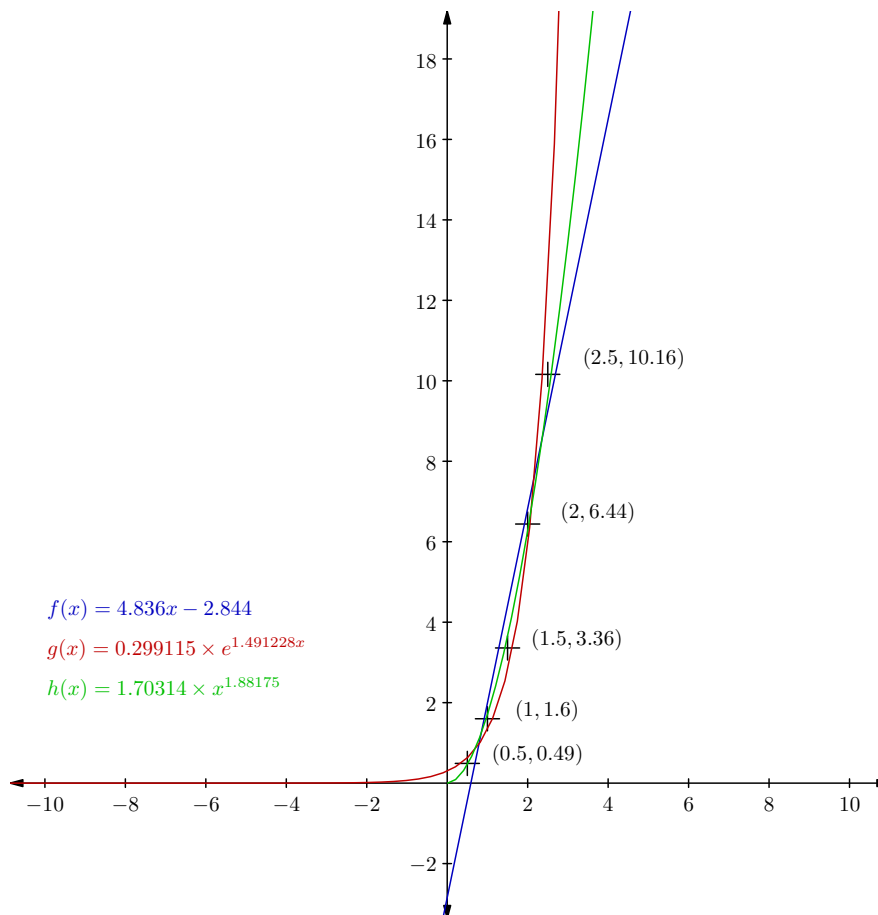
Erreur : 0.758042

Ajustement par une fonction “puissance” :

$$P(x) = 1.70314 \times x^{1.88175}$$

Erreur : 0.23913

On notera que pour ce jeu de points, c'est l'ajustement de type “puissance” qui est le plus approprié puisque l'erreur est la plus faible, et de loin, parmi les trois méthodes.



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

Régression linéaire :

$$P(x) = 0.500091x + 3.000091$$

Erreur : 0.837405

Ajustement exponentiel

$$P(x) = 3.804664 \cdot e^{0.071377 \cdot x}$$

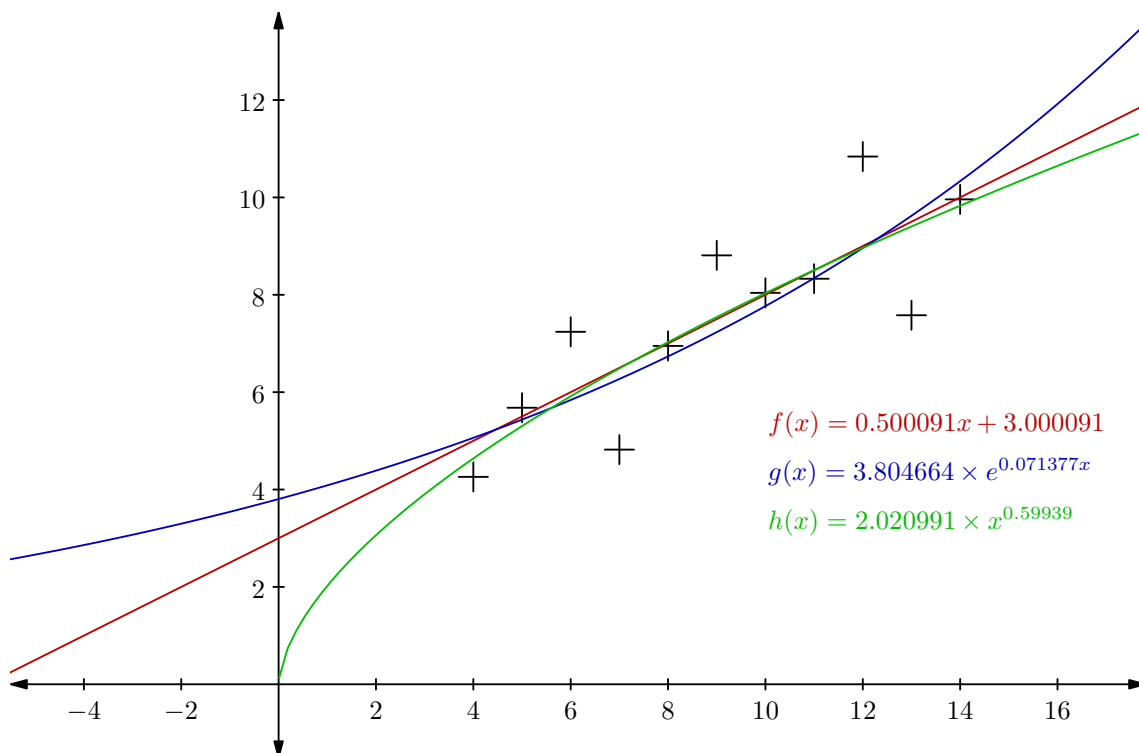
Erreur : 0.933867

Ajustement “puissance”

$$P(x) = 2.020991 \cdot x^{0.599390}$$

Erreur : 0.827968

Note : Un ajustement de type “puissance” permet d’obtenir une précision de 0.827968, ce qui est très proche de celle obtenue ici. L’ajustement exponentiel est moins adapté.



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

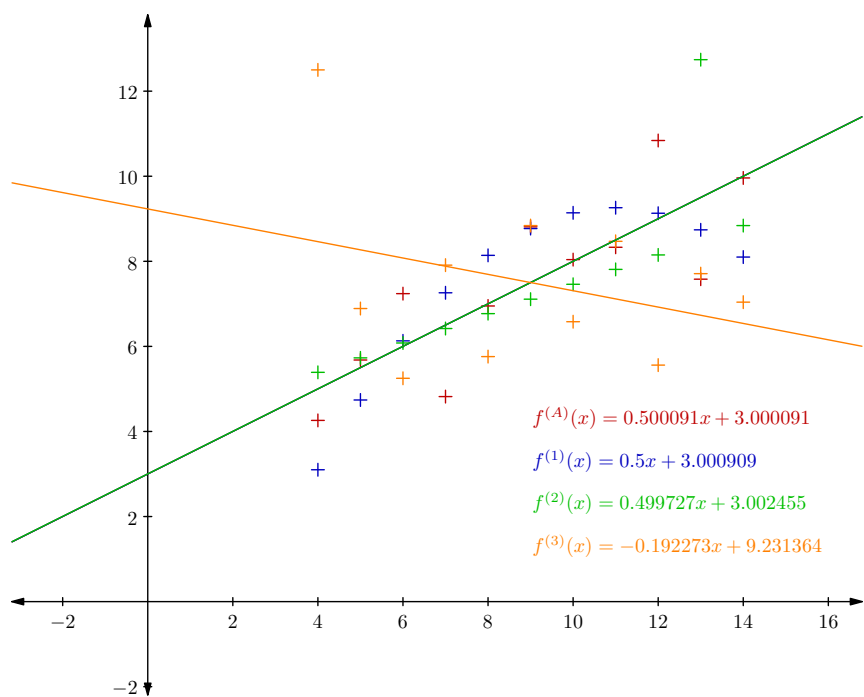
TABLEAU 3.4.3 – 3 séries $S^{(1)}$, $S^{(2)}$ et $S^{(3)}$ comparées à Anscombe

Série 1 : Régression linéaire : $P(x) = 3.000909 + 0.500000 \cdot x$ Erreur : 0.967934	Série 2 : Régression linéaire : $P(x) = 3.002455 + 0.499727 \cdot x$ Erreur : 0.715967	Série 3 : Régression linéaire : $P(x) = 9.231364 - 0.192273 \cdot x$ Erreur : 0.902727
Ajustement exponentiel : $P(x) = 3.417548 \cdot e^{0.082249 \cdot x}$ Erreur : 1.187786	Ajustement exponentiel : $P(x) = 4.100273 \cdot e^{0.063981 \cdot x}$ Erreur : 0.590601	Ajustement exponentiel : $P(x) = 8.564272 \cdot e^{-0.017989 \cdot x}$ Erreur : 1.468203
Ajustement “puissance” : $P(x) = 1.453451 \cdot x^{0.749910}$ Erreur : 0.950634	Ajustement “puissance” : $P(x) = 2.478570 \cdot x^{0.507328}$ Erreur : 0.682932	Ajustement “puissance” : $P(x) = 10.959075 \cdot x^{-0.192021}$ Erreur : 1.463448

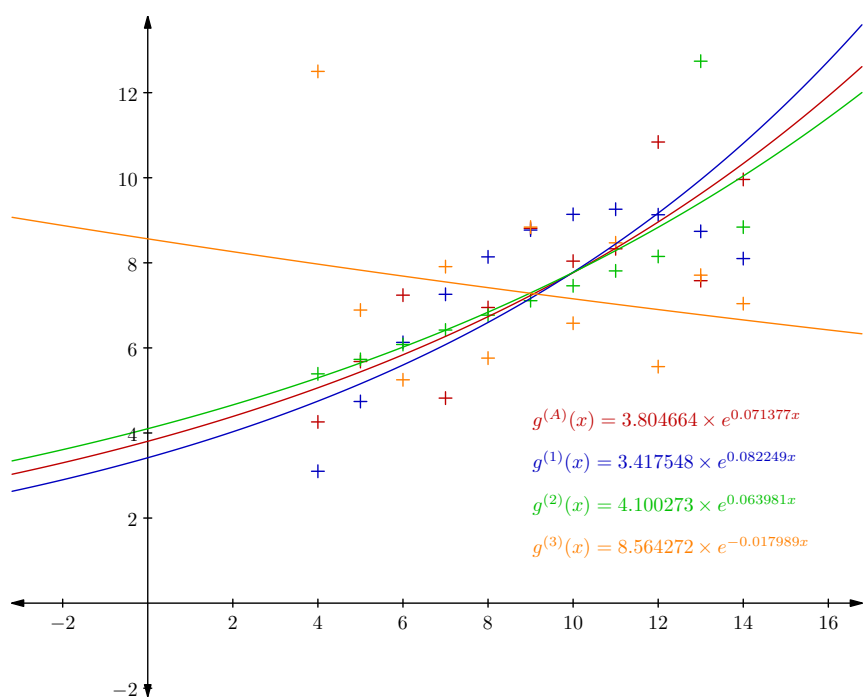
Pour la régression linéaire, il est important de remarquer que les deux premières séries donnent des équations de droites similaires et comparables à celle de la série d’Anscombe, tandis que la dernière varie complètement. On peut expliquer ceci par la présence de “points isolés” ((4, 12.50) pour la série $S^{(3)}$) qui influent sur le calcul de l’équation. Ainsi on obtient des équations similaires pour des séries de points très différentes.

On notera aussi que la série $S^{(2)}$ possède un point isolé (13, 12.74), et que tous les autres points semblent appartenir à la même droite. Ce point a tendance à “tirer” la droite vers le haut et donc à augmenter l’écart qui aurait pu être obtenu en l’absence de ce point.

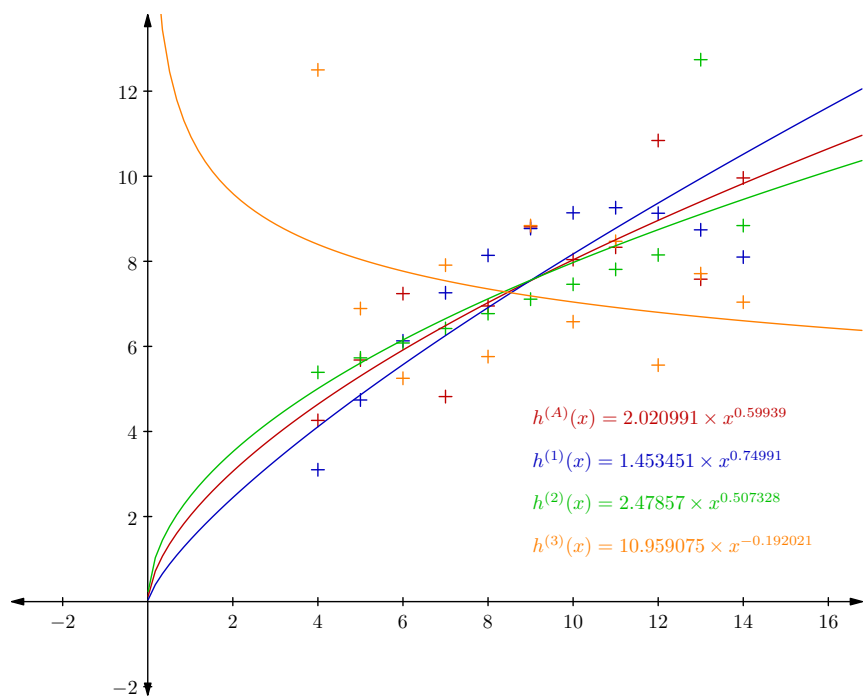
Les ajustements de types exponentiel et “puissance” donnent des écarts variables. Par exemple, il vaudra mieux approximer la série $S^{(1)}$ par puissance ou par régression, et la série $S^{(2)}$ avec un ajustement exponentiel. La série $S^{(3)}$ est composée d’un nuage de points, tout comme celle du test précédent (Anscombe), et les résultats montrent pour ces deux séries que des approximations par régression linéaire et parfois par ajustement “puissance” selon la disposition des points sont plus appropriés que l’ajustement exponentiel.



GRAPHIQUE 3.4.3 – Régression linéaire – (Tableau 3.4.3)



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



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
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 3.4.4 – Série non-triée

Comme pour l'interpolation, on peut trier les séries :

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 3.4.5 – 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 3.4.6 – Série triée : Partie Haute

Valeurs non-triées :

Régression Linéaire :

$$P(x) = -112.658491 + 0.324356 \cdot x$$

Erreur : 43.378231

Partie Basse :

Régression Linéaire :

$$P(x) = 80.112013 + 0.002173 \cdot x$$

Erreur : 1.607015

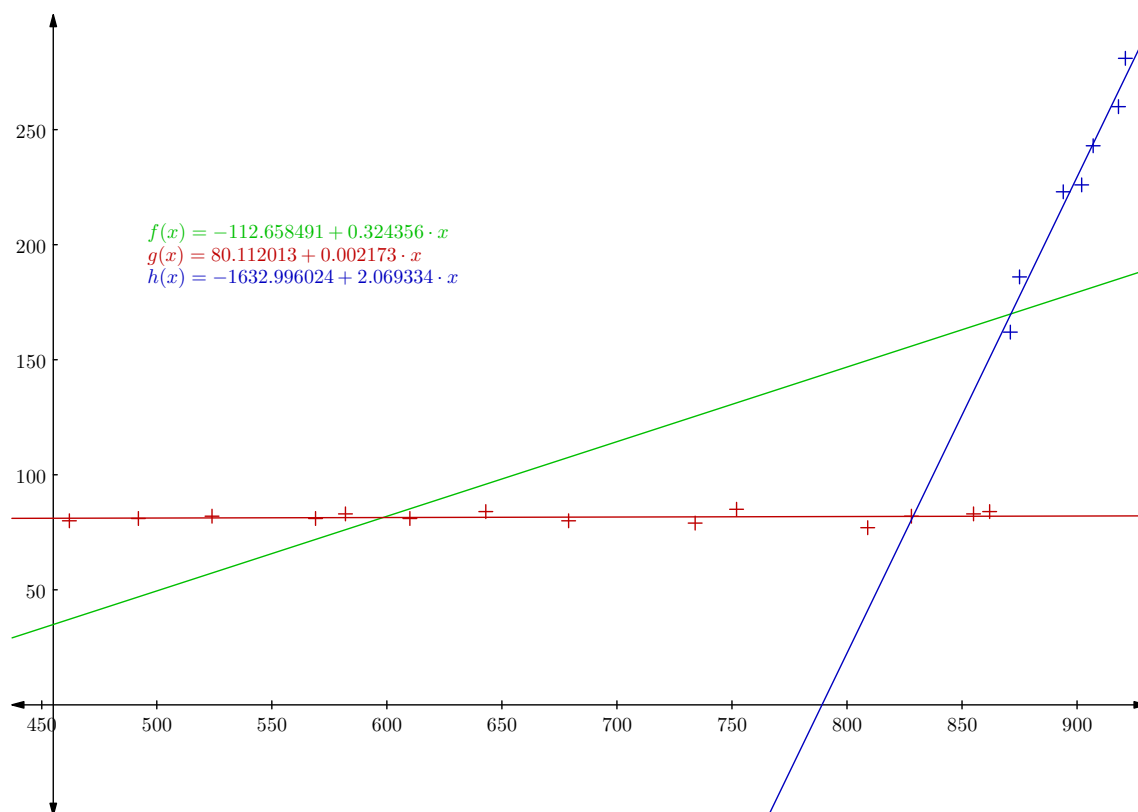
Partie Haute :

Régression Linéaire :

$$P(x) = -1632.996024 + 2.069334 \cdot x$$

Erreur : 6.422749

Ici, on constate de nouveau qu'une analyse préalable des points permet d'obtenir de meilleurs résultats. En effet, $\frac{2}{3}$ de ces points semblent appartenir à une droite horizontale, ce qui justifie le fait de scinder la série en deux sous-séries. Par conséquent, l'erreur obtenue pour chacune d'entre-elles est nettement plus faible. On peut ensuite considérer que la fonction qui régit la relation entre ces points change au-delà d'un certain seuil. En l'occurrence, il s'agirait d'un seuil de revenus.



GRAPHIQUE 3.4.6 – Régression Linéaire (Tableaux 3.4.4 à 3.4.6)

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 3.4.7 – Série

Dans le plan, ces points semblent appartenir à une courbe de type exponentiel, ce que justifient les résultats et le graphique suivants.

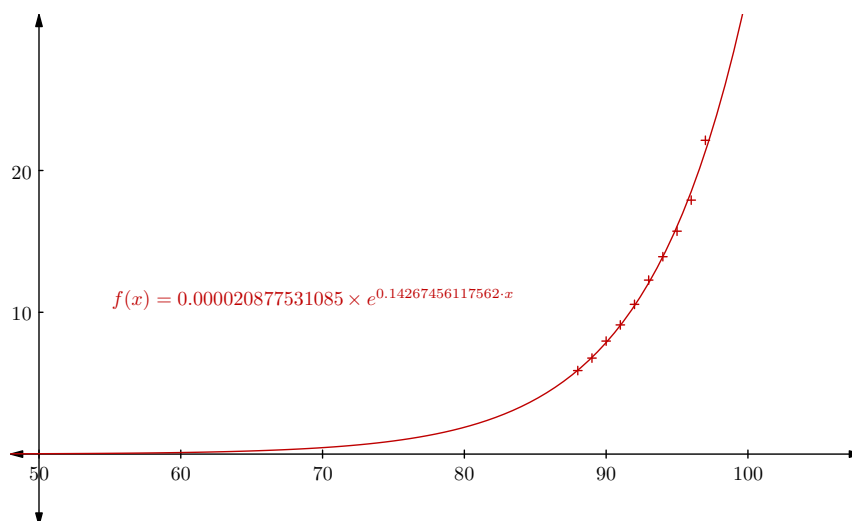
Ajustement exponentiel :

$$P(x) = 0.000020877531085468 \cdot e^{0.142674561175619524 \cdot x}$$

Erreur : 0.222680

On notera une erreur particulièrement faible, le type exponentiel est donc celui convient le mieux à ce jeu de points.

Cependant, on ne peut pas se contenter d'une faible précision. En effet, il faut dans ce cas au minimum une précision de 10^{-5} pour avoir une courbe exponentielle.



GRAPHIQUE 3.4.7 – Approximation par une fonction exponentielle (Tableau 3.4.7)

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 3.4.8 – Relation entre revenu et nombre de personnes ayant un revenu supérieur

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 :*

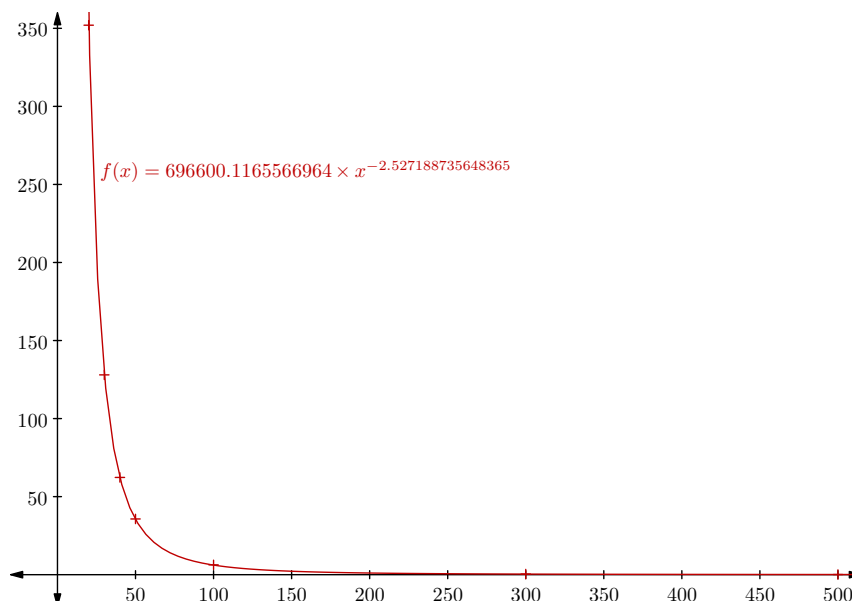
$$y = \frac{A}{x^a} = Ax^{-a}$$

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

Ajustement “puissance” :

$$P(x) = 696600.116557 \cdot x^{-2.527189}$$

Erreur : 1.179858



GRAPHIQUE 3.4.8 – Approximation par une fonction de type “puissance” (Tableau 3.4.8)

D’après les résultats, on aurait $a \approx 2.527189$ et $A \approx 696600.116557$ pour ce jeu de données et en effet, les constantes sont positives, comme l’indique la loi de Pareto.

On notera que ce type de courbe est plus approprié qu’un autre, et que l’erreur est faible au regard de la valeur moyenne des ordonnées : ≈ 83.5 .

3.5 Comparaison

On pourra remarquer que la qualité des résultats par approximation est variable et dépend en grande partie des données. Par conséquent, une analyse préalable semble pertinente pour déterminer l’approximation la plus adaptée, et éventuellement pour éliminer des points pouvant être le résultat d’erreurs de mesure.

Dans le cas d’un nuage de points, une régression linéaire donnera la plupart du temps de meilleurs résultats.

4 Conclusion

Le but des méthodes d'interpolation et d'approximation est d'obtenir une fonction mathématique dont la courbe passe au mieux par une série de points.

Les interpolations de Newton et Neville retournent des résultats très précis (les erreurs de calcul de la machine sont particulièrement faibles), mais sont coûteuses en mémoire et en calculs. De surcroît, la qualité des résultats est altérée lorsque le nombre de points augmente (effectivement, le degré augmente, la précision aussi, et la machine ne permet pas le calcul en précision infinie). Leur complexité moyenne est $O(n^2)$.

Lorsque ces interpolations ne permettent pas d'obtenir un résultat satisfaisant, on peut avoir recours à d'autres méthodes plus adaptées, ou bien se contenter de résultats moins fiables mais plus faciles à obtenir.

Les méthodes d'approximation abordées dans ce rapport permettent d'obtenir des résultats approchés, mais nécessitent une étude préalable des séries de points telle que le type de la fonction idéale, la vérification de la qualité des mesures (passant notamment par une suppression des points aberrants) ou la segmentation en sous-séries.

5 Annexe

5.1 Menu principal

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <math.h>
5  #include "neville.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         i=1;
38         while (i!=0 && i!=9)
39         {
40             clear(); //nettoyage écran
41             printf("Tableau de valeurs :\n");
42             for (i=0; i<2; i++)
43             {
44                 for (j=0; j<n; j++)
45                 {
46                     if (0==j && 0==i)
47                     {
48                         printf(" x ");
49                     }
50                     else if (0==j && 1==i)
51                     {
52                         printf(" y ");
53                     }
54                 }
55             }
56         }
57     }
```

```

54     printf("| %+4.5f ", tab[i][j]);
55 }
56 printf("\n");
57 }
58 printf("\nQuelle résolution utiliser ?\n");
59 printf("1- Newton\n");
60 printf("2- Neville\n");
61 printf("3- Régression Linéaire\n");
62 printf("4- Approximation par une fonction exponentielle\n");
63 printf("5- Approximation par une fonction puissance\n");
64 printf("9- Nouvelle série de points (Menu principal)\n");
65 printf("0- Quitter\n");
66 printf("Votre choix : ");
67 scanf("%d", &i);
68 c='0';
69 cleanBuffer(); //vidage buffer
70 switch (i)
71 {
72     case 1:
73         printf("Résolution par Newton ... \n");
74         newton(tab,n);
75         hitToContinue();
76         break;
77     case 2:
78         printf("Résolution par Neville ... \n");
79         neville(tab,n);
80         hitToContinue();
81         break;
82     case 3:
83         printf("Résolution par Régression linéaire ... \n");
84         reglinD(tab,n);
85         hitToContinue();
86         break;
87     case 4:
88         printf("Résolution par Approximation par une fonction exponentielle... \n");
89         reglinE(tab,n);
90         hitToContinue();
91         break;
92     case 5:
93         printf("Résolution par Approximation par une fonction puissance... \n");
94         reglinP(tab,n);
95         hitToContinue();
96         break;
97 }
98 printf("\n\n");
99 }
100
101 //libération mémoire
102 for (j=0; j<2; j++)
103 {
104     free(tab[j]);
105 }
106 free(tab);
107 clear(); //nettoyage écran
108 }
109 return 0;
110 }

```

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  $f(x)=c \cdot \exp(dx)$ .
25 double imagePui(double a, double b, double x); // La fonction calcule l'image de x par une fonction puissance
    de la forme  $f(x)=a \cdot x^b$ .
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  #include "useful.h"
8
9  polynome* creerPoly(int c, char* mode, ...)
10 {
11     int i;
12     polynome* P=(polynome*) malloc(sizeof(polynome));
13     P->d = c-1;
14     P->poln = (double*) malloc(c*sizeof(double));
15     if ((strcmp(mode, "valeur"))==0) //si mode = "valeur"
16     {
17         va_list ap;
18         va_start(ap, mode);
19         for (i=0; i<c; i++)
20         {
21             P->poln[i]=va_arg(ap, double);
22         }
23         va_end(ap);
24     }
25     else //si mode= "tableau"
26     {
27         va_list ap;
28         va_start(ap, mode);

```



```

29     double* tmp=va_arg(ap, double*);
30     for (i=0; i<c; i++)
31     {
32         P->poln[i]=tmp[i];
33     }
34     free(tmp);
35     va_end(ap);
36 }
37 redimensionnerPoly(P);
38 return P;
39 }
40
41 void menuAffichage(polynome* P)
42 {
43     FILE* fichier=fopen("resultat", "a+");
44     int choix; // permet de choisir les options voulues
45     printf("Voulez-vous afficher le polynome dans la sortie standard (1-Oui *-Non) ? ");
46     scanf("%d",&choix);
47     cleanBuffer();
48     if(choix ==1)
49     {
50         afficherPoly(P,"console");
51     }
52     else
53     {
54         afficherPoly(P,"console");
55         afficherPoly(P,"latex",fichier);
56     }
57     fclose(fichier);
58 }
59
60 void afficherPoly(polynome* P, char* mode, ...)
61 {
62     int i;
63     FILE* f;
64     int c = (P->d) +1 ;
65     if((strcmp(mode,"console")) == 0)
66     {
67         printf("P(x) = ");
68         for(i=0;i<c;i++)
69         {
70             if(P->poln[i] > 0)
71             {
72                 if(i != 0) { printf(" + "); }
73                 if(i == 0) { printf("%20.20f",P->poln[i]); }
74                 else if(i == 1) { printf("%20.20f * x",P->poln[i]); }
75                 else { printf("%20.20f * x^%d", P->poln[i],i); }
76             }
77             if(P->poln[i] < 0)
78             {
79                 printf(" - ");
80                 if(i == 0) { printf("%20.20f",-P->poln[i]); }
81                 else if(i == 1) { printf("%20.20f * x",-P->poln[i]); }
82                 else { printf("%20.20f * x^%d", -(P->poln[i]), i); }
83             }
84         }
85         printf("\n");
86     }
87     else
88     {
89         va_list ap;
90         va_start(ap,mode);
91         f = va_arg(ap,FILE*);
92         fprintf(f,"$P(x) \\approx ");
93         for(i=0; i<c; i++)
94         {
95             if(P->poln[i] > 0)
96             {
97                 if(i != 0) { fprintf(f," + "); }
98                 if(i == 0) { fprintf(f,"%20.6f",P->poln[i]); }
99                 else if(i == 1) { fprintf(f,"%20.6f \\cdot x",P->poln[i]); }

```

```

100     else { fprintf(f, "%.6f \\cdot x^{%d} ", P->poln[i], i); }
101     }
102     if(P->poln[i] < 0)
103     {
104         if(i == 0) { fprintf(f, "-%.6f", -(P->poln[i])); }
105         else if(i == 1) { fprintf(f, "-%.6f \\cdot x", -(P->poln[i])); }
106         else { fprintf(f, "- %.6f \\cdot x^{%d} ", -(P->poln[i]), i); }
107     }
108     }
109     fprintf(f, "$\\n\\n");
110     va_end(ap);
111 }
112 }
113
114 void redimensionnerPoly(polynome* P1)
115 {
116     int degre=P1->d;
117     while((P1->poln[degre])==0)
118     {
119         degre--;
120     }
121     if(degre!=P1->d)
122     {
123         P1->d=degre;
124         P1->poln= (double*) realloc(P1->poln, (degre+1)*sizeof(double));
125     }
126 }
127
128 polynome* addPoly(polynome* P1, polynome* P2)
129 {
130     // Rappel : Deg(P1+P2) <= max(Deg(P1), Deg(P2))
131     int i;
132     polynome* P=(polynome*)malloc(sizeof(polynome));
133     if(P1->d > P2->d) // Deg(P1) > Deg(P2)
134     {
135         P->d = P1->d;
136         P->poln = (double*) malloc((1+P1->d)*sizeof(double));
137         for(i=0; i <= P2->d; i++)
138         {
139             P->poln[i] = P1->poln[i] + P2->poln[i];
140         }
141         for(i= P2->d +1; i<= P1->d; i++)
142         {
143             P->poln[i] = P1->poln[i];
144         }
145     }
146     else if (P1->d < P2->d) // Deg(P2) > Deg(P1)
147     {
148         P->d = P2->d;
149         P->poln = (double*) malloc((1+P2->d)*sizeof(double));
150         for(i=0; i <= P1->d; i++)
151         {
152             P->poln[i] = P1->poln[i] + P2->poln[i];
153         }
154         for(i= P1->d +1; i<= P2->d; i++)
155         {
156             P->poln[i] = P2->poln[i];
157         }
158     }
159     else // Deg(P2) = Deg(P1)
160     {
161         P->d = P2->d;
162         P->poln = (double*) malloc((1+P2->d)*sizeof(double));
163         for(i=0; i <= P1->d; i++)
164         {
165             P->poln[i] = P1->poln[i] + P2->poln[i];
166         }
167     }
168     redimensionnerPoly(P);
169     return P;
170 }

```

```

171
172 polynome* mulSPoly(double s, polynome* P1)
173 {
174     int i;
175     polynome* P=(polynome*) malloc(sizeof(polynome));
176     P->d = P1->d;
177     P->poln = (double*) malloc((1+P1->d)*sizeof(double));
178
179     for(i=0;i <= P1->d;i++)
180     {
181         P->poln[i] = s*P1->poln[i] ;
182     }
183     redimensionnerPoly(P); //redimensionnement nécessaire si le scalaire est nul.
184     return P;
185 }
186
187 polynome* mulPoly(polynome * P1, polynome* P2)
188 {
189     int i,j;
190     polynome* P=(polynome*)malloc(sizeof(polynome));
191     P->d = P1->d + P2->d ;
192     P->poln = (double*) malloc((1+P->d)*sizeof(double));
193
194     for(i=0; i<=P->d; i++)
195     {
196         P->poln[i] = 0;
197     }
198
199     for(i=0; i<=(P2->d); i++)
200     {
201         for(j=0; j<=(P1->d); j++)
202         {
203             P->poln[i+j] = P->poln[i+j] + (P2->poln[i])*(P1->poln[j]);
204         }
205     }
206     redimensionnerPoly(P);
207     return P;
208 }
209
210 double imagePoly(polynome* P, double x)
211 {
212     int i;
213     double res = P->poln[0];
214     for(i=1;i<=P->d;i++)
215     {
216         res = res + P->poln[i]*pow(x,i);
217     }
218     return res;
219 }
220
221 double imageExpo(double c, double d, double x)
222 {
223     double res = 0.;
224     res = c*exp(d*x);
225     return res;
226 }
227
228 double imagePui(double a, double b, double x)
229 {
230     double res = 0.;
231     res = a*pow(x,b);
232     return res;
233 }
234
235 void ecartPoly(double** tab, int n, polynome* P)
236 {
237     int i;
238     double moyecart= 0.;
239     for(i=0;i<n;i++)
240     {
241         moyecart = moyecart + fabs((imagePoly(P,tab[0][i])-tab[1][i]));

```

```

242     }
243     moyecart = moyecart/n;
244     printf("Erreur moyenne : %.18f",moyecart);
245 }
246
247 void ecartExpo(double** tab, int n, double c, double d)
248 {
249     int i;
250     double moyecart= 0.;
251     for(i=0;i<n;i++)
252     {
253         moyecart = moyecart + fabs((imageExpo(c,d,tab[0][i])-tab[1][i]));
254     }
255     moyecart = moyecart/n;
256     printf("Erreur moyenne : %.18f",moyecart);
257 }
258
259 void ecartPui(double** tab, int n, double a, double b)
260 {
261     int i;
262     double moyecart= 0.;
263     for(i=0;i<n;i++)
264     {
265         moyecart = moyecart + fabs((imagePui(a,b,tab[0][i])-tab[1][i]));
266     }
267     moyecart = moyecart/n;
268     printf("Erreur moyenne : %.18f",moyecart);
269 }
270
271 void convertTabtoLatex(double** tab, int n, int m)
272 {
273     FILE* fichier = fopen("resultat","a+");
274     int i,j;
275     //déclaration de l'environnement et de n+1 colonnes
276     fprintf(fichier,"\\begin{tabular}{|");
277     for(i=0;i<=n;i++)
278     {
279         fprintf(fichier," c |");
280     }
281     fprintf(fichier,"}\\n \\hline \\n");
282
283     //remplissage des cases
284     for(i=0;i<2;i++)
285     {
286         if(i==0) {fprintf(fichier,"$x_{i}$ & ");}
287         if(i==1) {fprintf(fichier,"$y_{i}$ & ");}
288         for(j=0;j<n;j++)
289         {
290             if(j!=(n-1)) {fprintf(fichier,"$%f$ & ",tab[i][j]);}
291             else {fprintf(fichier,"$%f$ ",tab[i][j]);}
292         }
293         fprintf(fichier,"\\\\ \\n \\hline \\n");
294     }
295     fprintf(fichier,"\\end{tabular}\\n\\n");
296     fclose(fichier);
297 }

```

FIGURE 5.3 – Code : polynome.c