Algorithmes numériques – Rapport

Interpolation et Approximation

Axel Delsol, Pierre-Loup Pissavy

Décembre 2013

Table des matières

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 – Apercu : 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.

```c
typedef struct polynome
{
  int d; //degree
  double* poln; //coefficients
} 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 – Apercu : 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 : $\nabla^0_{y_i} = y_i, \forall i = 1, \cdots, N$.

La différence divisée de degré k est : $\nabla^k_{y_i} = \frac{\nabla^{k-1}_{y_i} - \nabla^{k-1}_{y_k}}{x_i - x_k}, \forall i = k+1, \cdots, N$.

Ensuite, on a directement les coefficients du polynôme de Newton par la relation $a_i = \nabla^i_{y_{(i+1)}}, \forall i = 1, \cdots, N-1$

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

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

2.1.2 Programme

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "polynome.h"

void newton (double ** tab, int n)
```

```
{
  int i, k;
  polynome* pol1=(polynome*) malloc(sizeof(polynome));
  polynome* pol2=(polynome*) malloc(sizeof(polynome));
  double** t= (double**) malloc(n*sizeof(double*));
  for (i=0; i<n; i++)
  {
    t[i]= (double*) malloc((i+1)*sizeof(double));
  }
  //initialisation des valeurs : on récupère les y.
  for (i=0; i<n; i++)
  {
    t[i][0]=tab[1][i];
  }
  //calcul des differences divisees
  for (k=1; k<n; k++)
  {
    for (i=k; i<n; i++)
    {
      t[i][k]=(t[i][k-1]-t[k-1][k-1])/(tab[0][i]-tab[0][k-1]);
    }
  }
  //tableau de poly
  polynome** tabP= (polynome**) malloc(n*(sizeof(polynome*)));
  for (i=0; i<n; i++)
  {
    tabP[i]=(polynome*) malloc(sizeof(polynome));
  }
  tabP[0]->d=0;
  tabP[0]->poln=(double*) malloc(sizeof(double));
  tabP[0]->poln[0]=t[n-1][n-1];
  for (i=1; i<n; i++)
  {
    pol1=creerPoly(2,"valeur",-tab[0][n-1-i], 1.);
    pol2=mulPoly(pol1,tabP[i-1]);
    free(pol1->poln); free(pol1);
    pol1=creerPoly(1,"valeur",t[n-1-i][n-1-i]);
    tabP[i]=addPoly(pol1,pol2);
    free(pol1->poln); free(pol2->poln);
    free(pol1); free(pol2);
  }
  redimensionnerPoly(tabP[n-1]);
  //affichage
  menuAffichage(tabP[n-1]);
  ecartPoly(tab,n,tabP[n-1]);
  printf("\n");
  //libération mémoire
  for(i=0;i<n;i++)
  {
    free(tabP[i]->poln);
    free(tabP[i]);
    free(t[i]);
  }
  free(tabP);
  free(t);
}
```

Figure 2.1 – Code : newton.c

2.2 Méthode de Neuville

2.2.1 Présentation

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

L'expression est donnée sous la forme suivante :
$P_k[x_i, \cdots, x_{i+k}](x) = \frac{(x-x_{i+k})\cdot P_{k-1}[x_i,\cdots,x_{i+k-1}](x)+(x_i-x)\cdot P_{k-1}[x_{i+1},\cdots,x_{i+k}](x)}{x_i-xi+k}, \forall x, \forall k = 2, \cdots, N-1$

2.2.2 Programme

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "polynome.h"

void neuville (double ** tab, int n)
{
  int i, k;
  polynome* pol1=(polynome*) malloc(sizeof(polynome));
  polynome* pol2=(polynome*) malloc(sizeof(polynome));
  polynome* pol3=(polynome*) malloc(sizeof(polynome));
  polynome*** t= (polynome***) malloc(n*sizeof(polynome**));
  for (i=0; i<n; i++)
  {
    t[i]= (polynome**) malloc((i+1)*sizeof(polynome*));
  }
  //initialisation des valeurs : on récupère les y.
  for (i=0; i<n; i++)
  {
    t[i][0]=creerPoly(1,"valeur", tab[1][i]);
  }
  //calcul des differences divisees
  for (k=1; k<n; k++)
  {
    for (i=k; i<n; i++)
    {
      pol1=creerPoly(2, "valeur", tab[0][i-k], -1.); pol2=mulPoly(pol1,t[i][k-1]);
      free(pol1->poln); free(pol1);
      pol1=creerPoly(2,"valeur", -(tab[0][i]), 1.); pol3=mulPoly(pol1, t[i-1][k-1]);
      free(pol1->poln); free(pol1);
      pol1=addPoly(pol3, pol2);
      t[i][k]=mulSPoly((1/((tab[0][i-k])-(tab[0][i]))),pol1);
      free(pol1->poln); free(pol2->poln); free(pol3->poln); free(pol1); free(pol2); free(pol3);
    }
  }
  //poly à retourner
  redimensionnerPoly(t[n-1][n-1]);
  //affichage
  menuAffichage(t[n-1][n-1]);
  ecartPoly(tab,n,t[n-1][n-1]);
  printf("\n");
  //libération mémoire
  for(i=0;i<n;i++)
  {
    for(k=0;k<i;k++)
    { free(t[i][k]->poln); free(t[i][k]); }
    free(t[i]);
  }
  free(t);
}
```

FIGURE 2.2 – Code : neuville.c

2.3 Résultats de tests

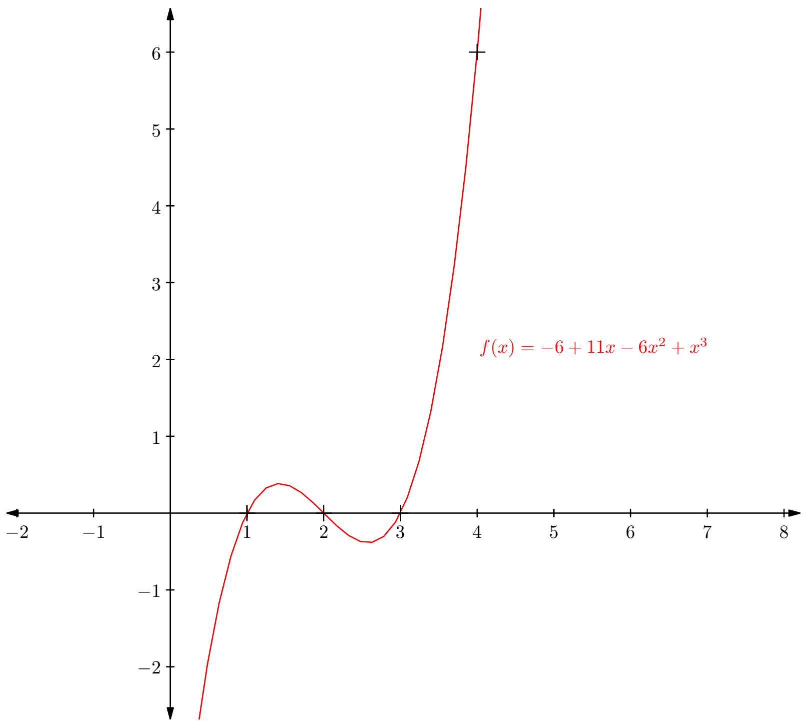
2.3.1 Exemple tiré d'un TD

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)

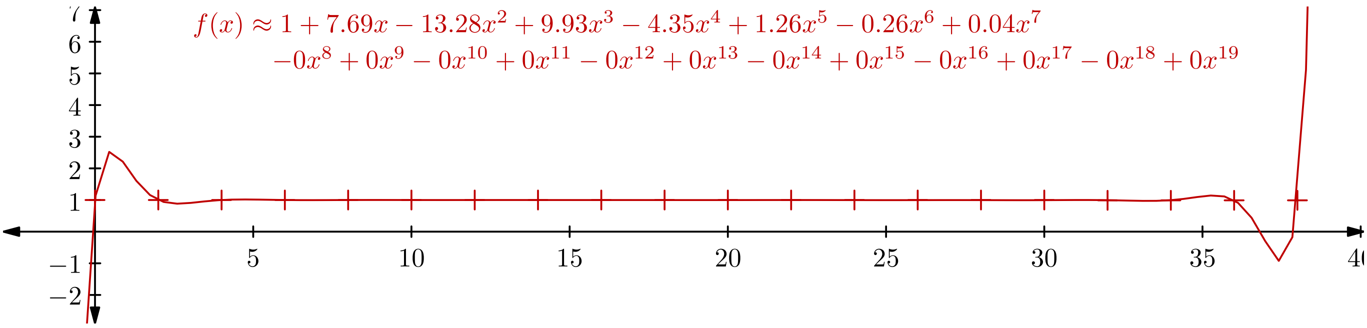
2.3.2 Densité de l'eau en fonction de la température

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 Neuville :
$P(x) \approx 0.999870 + 7.693711 \cdot x - 13.276666 \cdot x^2 + 9.932303 \cdot x^3 - 4.345460 \cdot x^4 + 1.259124 \cdot x^5 - 0.258585 \cdot x^6 + 0.039240 \cdot x^7 - 0.004520 \cdot x^8 + 0.000402 \cdot x^9 - 0.000028 \cdot x^{10} + 0.000002 \cdot x^{11} - 0.000000 \cdot x^{12} + 0.000000 \cdot x^{13} - 0.000000 \cdot x^{14} + 0.000000 \cdot x^{15} - 0.000000 \cdot x^{16} + 0.000000 \cdot x^{17} - 0.000000 \cdot x^{18} + 0.000000 \cdot x^{19}$
Erreur : 0.000028505775100296



Graphique 2.3.2 – Interpolation de Newton et Neuville – (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 :

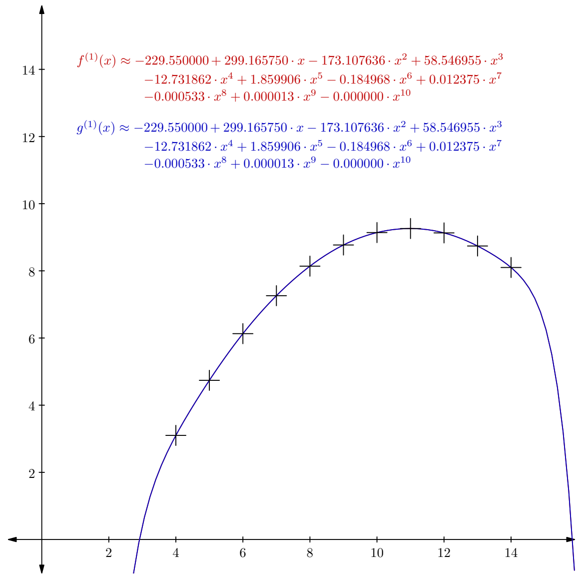
Méthode de Newton :
$P(x) \approx -12345.190000 + 16608.066492 \cdot x - 9870.941498 \cdot x^2 + 3416.593892 \cdot x^3 - 763.094009 \cdot x^4 + 114.979985 \cdot x^5 - 11.842442 \cdot x^6 + 0.823658 \cdot x^7 - 0.037039 \cdot x^8 + 0.000973 \cdot x^9 - 0.000011 \cdot x^{10}$
Erreur : 0.000000000774325735

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

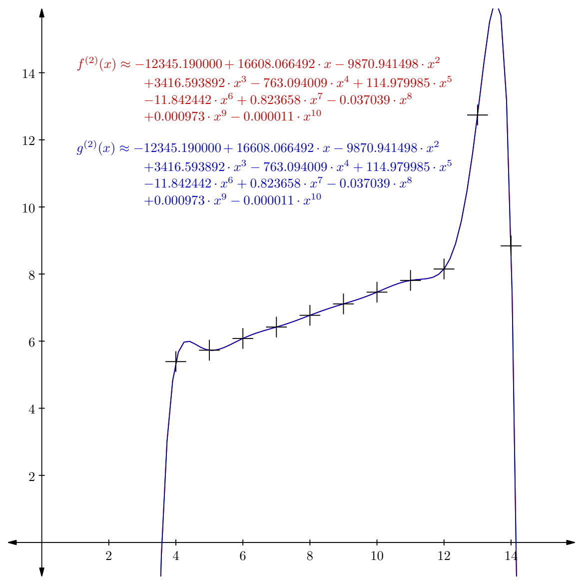
Série 3 :

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

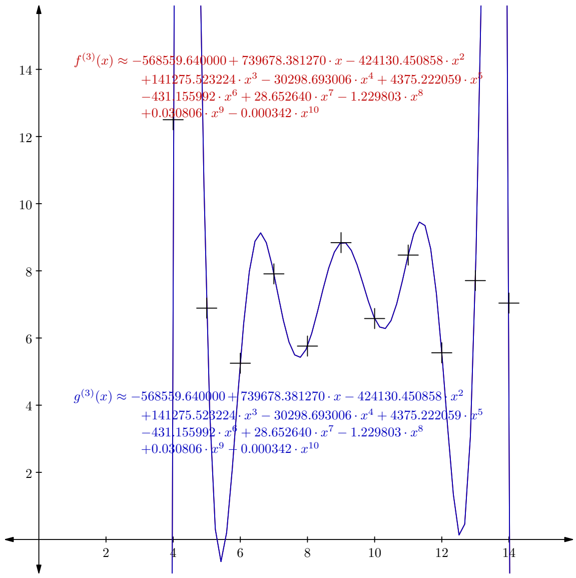
Méthode de Neuville :
$P(x) \approx -568559.640000 + 739678.381270 \cdot x - 424130.450858 \cdot x^2 + 141275.523224 \cdot x^3 - 30298.693006 \cdot x^4 + 4375.222059 \cdot x^5 - 431.155992 \cdot x^6 + 28.652640 \cdot x^7 - 1.229803 \cdot x^8 + 0.030806 \cdot x^9 - 0.000342 \cdot x^{10}$
Erreur : 0.000000035876804073



Graphique 2.3.3 – Interpolations de Newton et Neuville – Série 1 (Tableau 2.3.3)



Graphique 2.3.4 – Interpolations de Newton et Neuville – Série 2 (Tableau 2.3.3)



GRAPHIQUE 2.3.5 – Interpolations de Newton et Neuville – 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 :

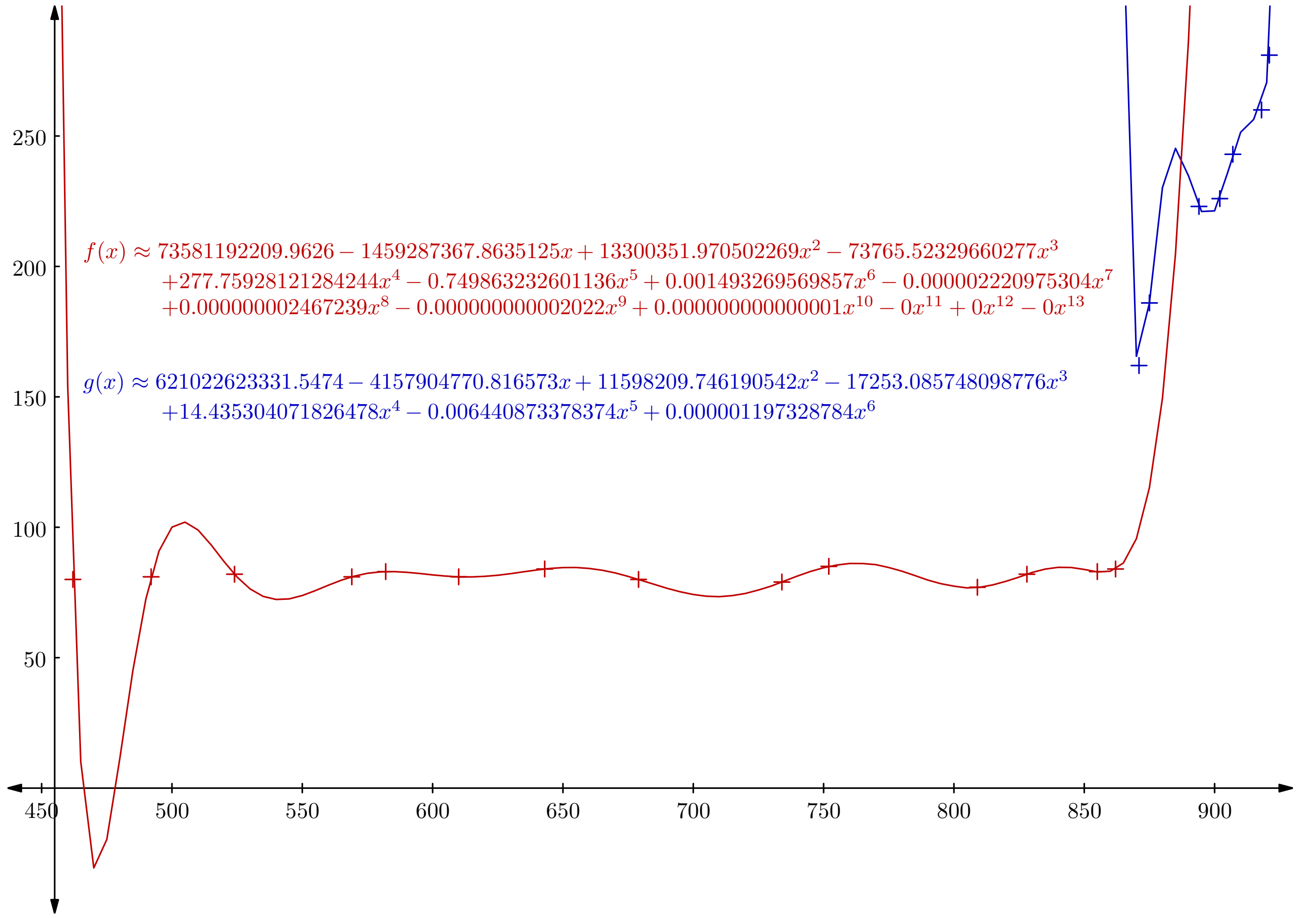
Méthode de Newton :
$P(x) \approx 73581192209.962601 - 1459287367.863513 \cdot x + 13300351.970502 \cdot x^2 - 73765.523297 \cdot x^3 + 277.759281 \cdot x^4 - 0.749863 \cdot x^5 + 0.001493 \cdot x^6 - 0.000002 \cdot x^7 + 0.000000 \cdot x^8 - 0.000000 \cdot x^9 + 0.000000 \cdot x^{10} - 0.000000 \cdot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

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

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

3.1.2 Programme

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "polynome.h"

void mapping(double** from, double** to, int n, char* fn);

double moyenneElements(double** tab,int l, int n)
{
  double resultat = 0.;
  double cpt = 0.;
  int i;
  for(i=0;i<n;i++)
  {
    resultat = resultat + tab[l][i];
    cpt = cpt + 1.;
  }
  resultat = resultat/cpt;
  return resultat;
}

double moyenneElementsCarres(double** tab,int l, int n)
{
  double resultat = 0;
```

```c
  double cpt = 0;
  int i;
  for(i=0;i<n;i++)
  {
    resultat = resultat + pow(tab[1][i],2);
    cpt = cpt + 1;
  }
  resultat = resultat/cpt;
  return resultat;
}

double moyenneProduitElements(double** tab, int n)
{
  double resultat = 0;
  double cpt = 0;
  int i;
  for(i=0;i<n;i++)
  {
    resultat = resultat + tab[0][i]*tab[1][i];
    cpt = cpt + 1;
  }
  resultat = resultat/cpt;
  return resultat;
}

reglinD(double** tab, int n)
{
  double a0 = 0;
  double a1 = 0;
  double xb, yb, xcb, xyb; // b pour barre et c pour carre
  printf("Nous cherchons le Polynome de degré 1 sous la forme a0 + a1*x.\n");

  //calculs
  xb = moyenneElements(tab,0,n);
  yb = moyenneElements(tab,1,n);
  xcb = moyenneElementsCarres(tab,0,n);
  xyb = moyenneProduitElements(tab,n);

  a1 = (xyb-xb*yb)/(xcb-pow(xb,2));
  a0 = yb-xb*a1;

  // creation et affichage
  polynome *P = creerPoly(2,"valeur",a0,a1);
  menuAffichage(P);

  //statistiques
  ecartPoly(tab,n,P);
  printf("\n");

  //libération mémoire
  free(P->poln);
  free(P);
```

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 :
$Y = a_0 + a_1 x$ 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

```
}

reglinE(double** tab, int n) //y=c(e^(dx)) <=> ln(y)=ln(c)+xd => c=e^(a0) & d=a1
{
  int i;
  double c = 0;
  double d = 0;
  double a0 = 0;
  double a1 = 0;
  double xb, yb, xcb, xyb; // b pour barre et c pour carre
  double** t = (double**) malloc(2*sizeof(double*)); // contiendra le mapping de tab
  for(i=0;i<2;i++)
  {
    t[i] = (double*) malloc (n*sizeof(double));
  }
  printf("Nous cherchons une approximation sous la forme c*(e^(d*x)).\n");

  //calculs
  mapping(tab, t, n, "exponentielle");
  xb = moyenneElements(t,0,n);
  yb = moyenneElements(t,1,n);
  xcb = moyenneElementsCarres(t,0,n);
  xyb = moyenneProduitElements(t,n);

  a1 = (xyb-xb*yb)/(xcb-pow(xb,2.));
  a0 = yb-xb*a1;
  d = a1;
  c = exp(a0);

  //affichage
  printf("P(x) = %20.18f*exp(%20.18f*x)\n",c,d);

  //statistiques
  ecartExpo(tab,n,c,d);
  printf("\n");

  //libération mémoire
  for (i=0; i<2; i++)
  {
    free(t[i]);
  }
```

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

```c
}

reglinP(double ** tab, int n) //y=a(x^b) <=> ln(y)=ln(a)+b*ln(x) => a=e^(a0) & b=a1
{
  int i;
  double a = 0.; double b = 0.; double a0 = 0.; double a1 = 0.;
  double xb, yb, xcb, xyb; // b pour barre et c pour carre
  double** t = (double**) malloc(2*sizeof(double*)); // contiendra le mapping de tab
  for(i=0;i<2;i++)
  {
    t[i] = (double*) malloc (n*sizeof(double));
  }
  printf("Nous cherchons une approximation sous la forme a*(x^(b)).\n");

  //calculs
  mapping(tab, t, n, "puissance");
  xb = moyenneElements(t,0,n);
  yb = moyenneElements(t,1,n);
  xcb = moyenneElementsCarres(t,0,n);
  xyb = moyenneProduitElements(t,n);

  a1 = (xyb-xb*yb)/(xcb-pow(xb,2));
  a0 = yb-xb*a1;
  b = a1;
  a = exp(a0);

  //affichage
  printf("P(x) = %20.18f*x^(%20.18f)\n",a,b);

  //statistiques
  ecartPui(tab,n,a,b);
  printf("\n");

  //libération mémoire
  for (i=0; i<2; i++)
  {
    free(t[i]);
  }
  free(t);
}

void mapping(double** from, double** to, int n, char* fn)
{
  int i, j;
  if (strcmp(fn,"exponentielle")==0)
  {
    for (j=0; j<n; j++){to[0][j]=from[0][j];}
    for (j=0; j<n; j++){to[1][j]=log(from[1][j]);}
  }
  else if (strcmp(fn,"puissance")==0)
  {
    for (i=0; i<2; i++){for (j=0; j<n; j++){to[i][j]=log(from[i][j]);}}
  }
```

```
}
```

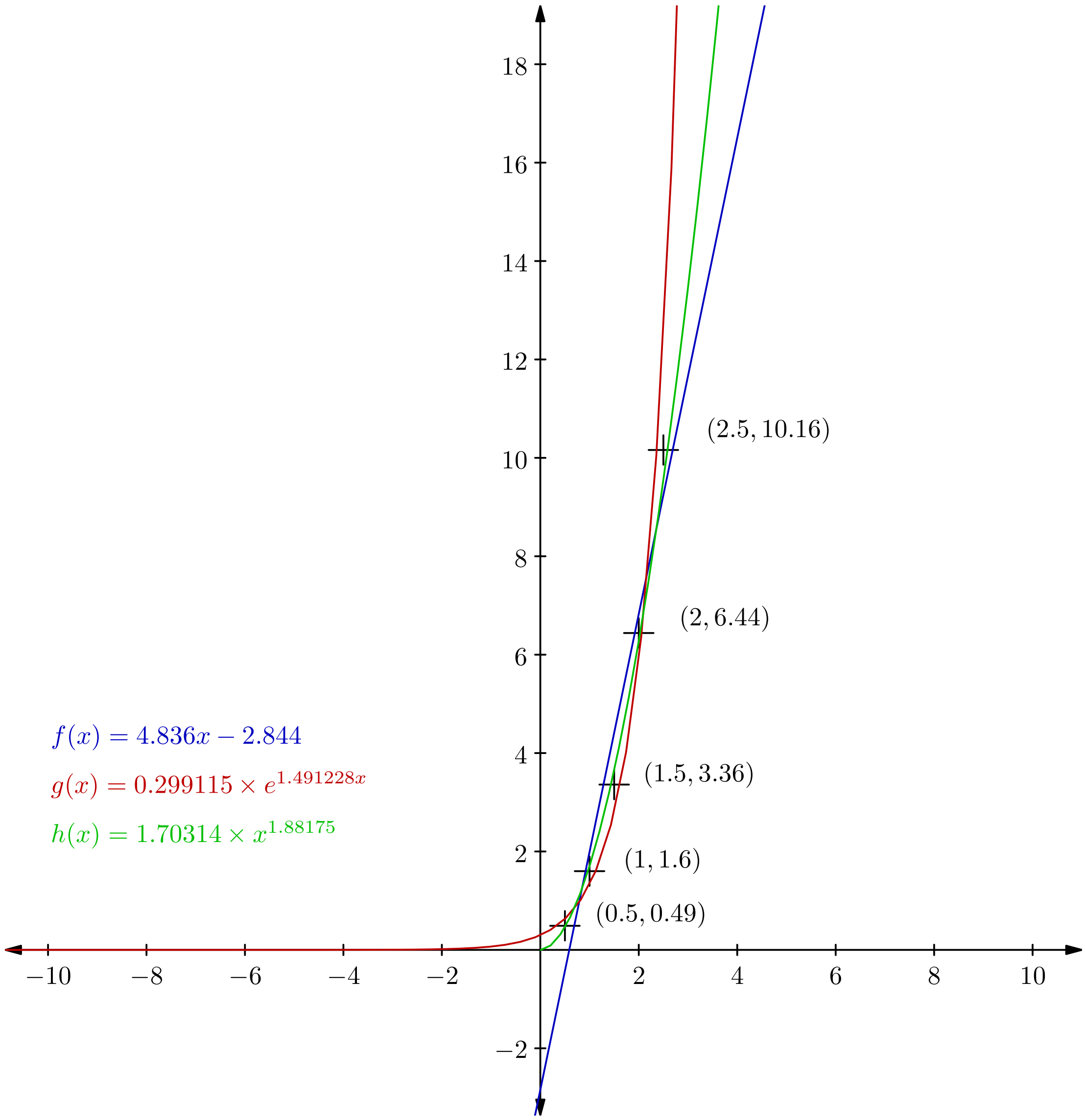
Figure 3.3 – Code : reglin.c

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

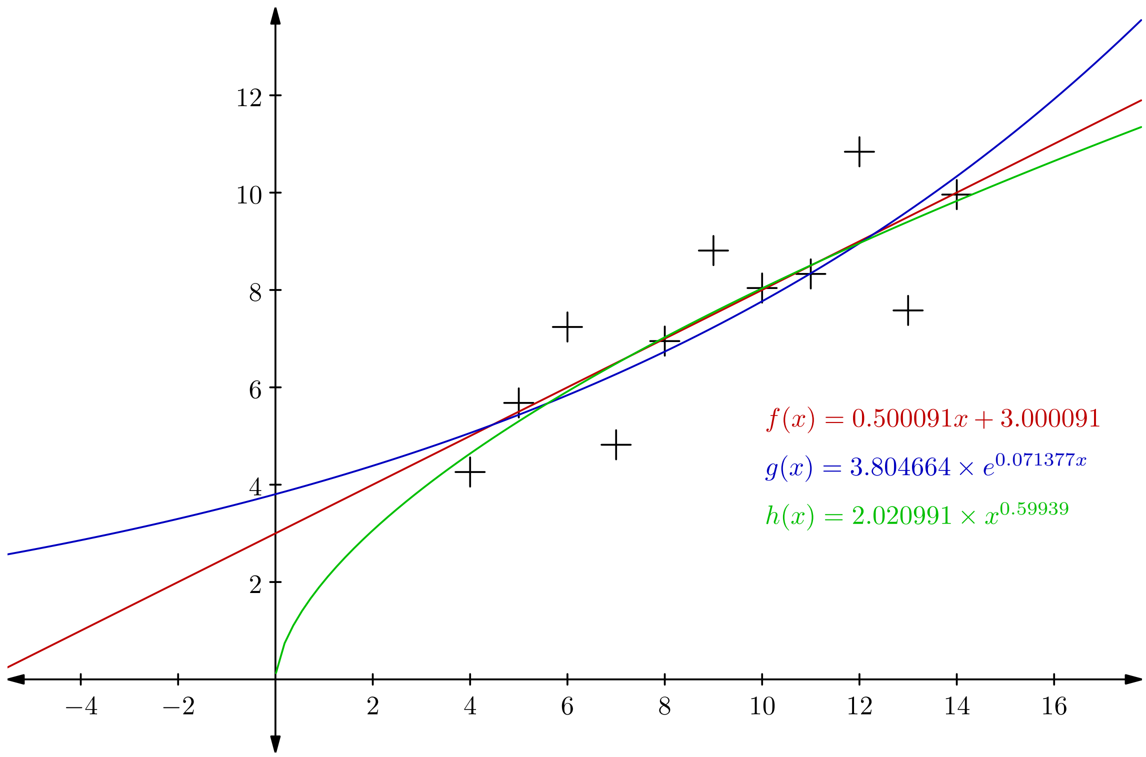


GRAPHIQUE 3.4.1 – Régression linéaire – (Tableau 3.4.1)

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 dûe à Anscombe



GRAPHIQUE 3.4.2 – Régression linéaire – (Tableau 3.4.2)

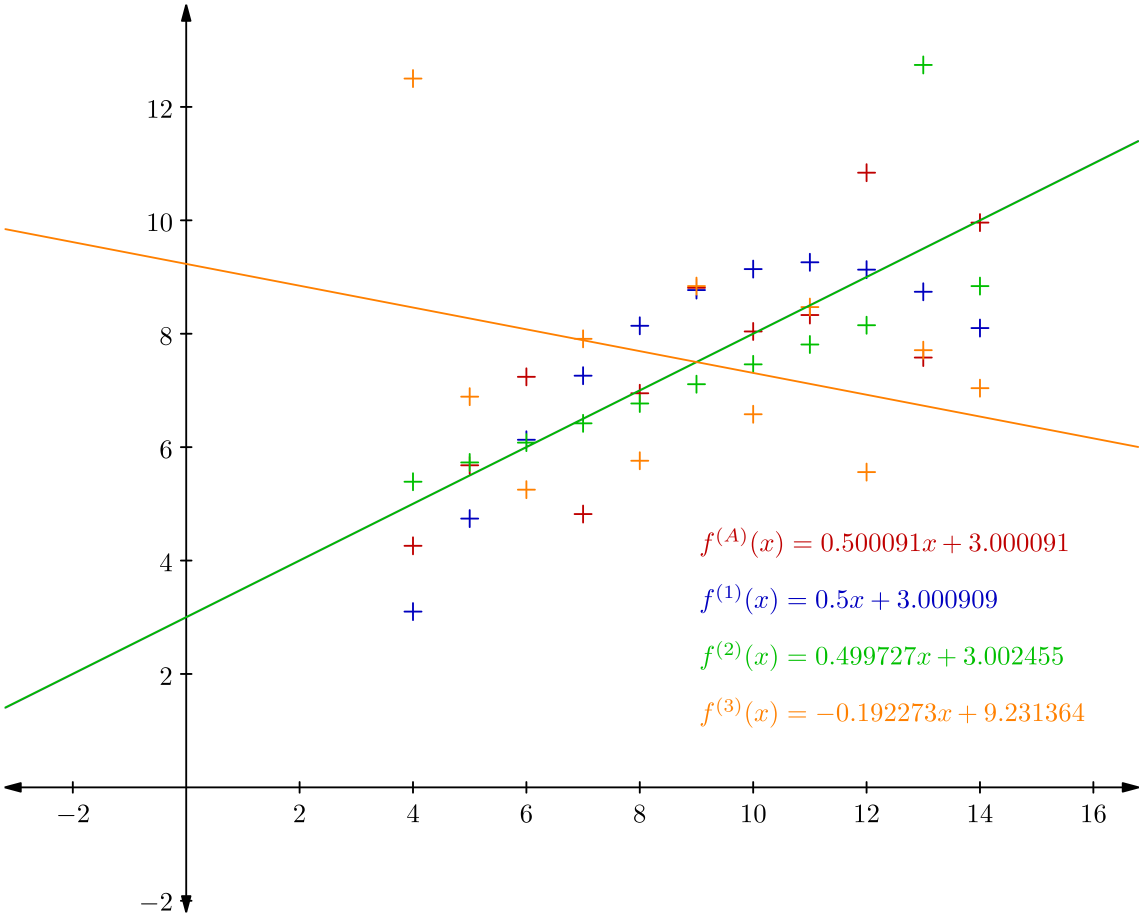
3.4.3 3 séries

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

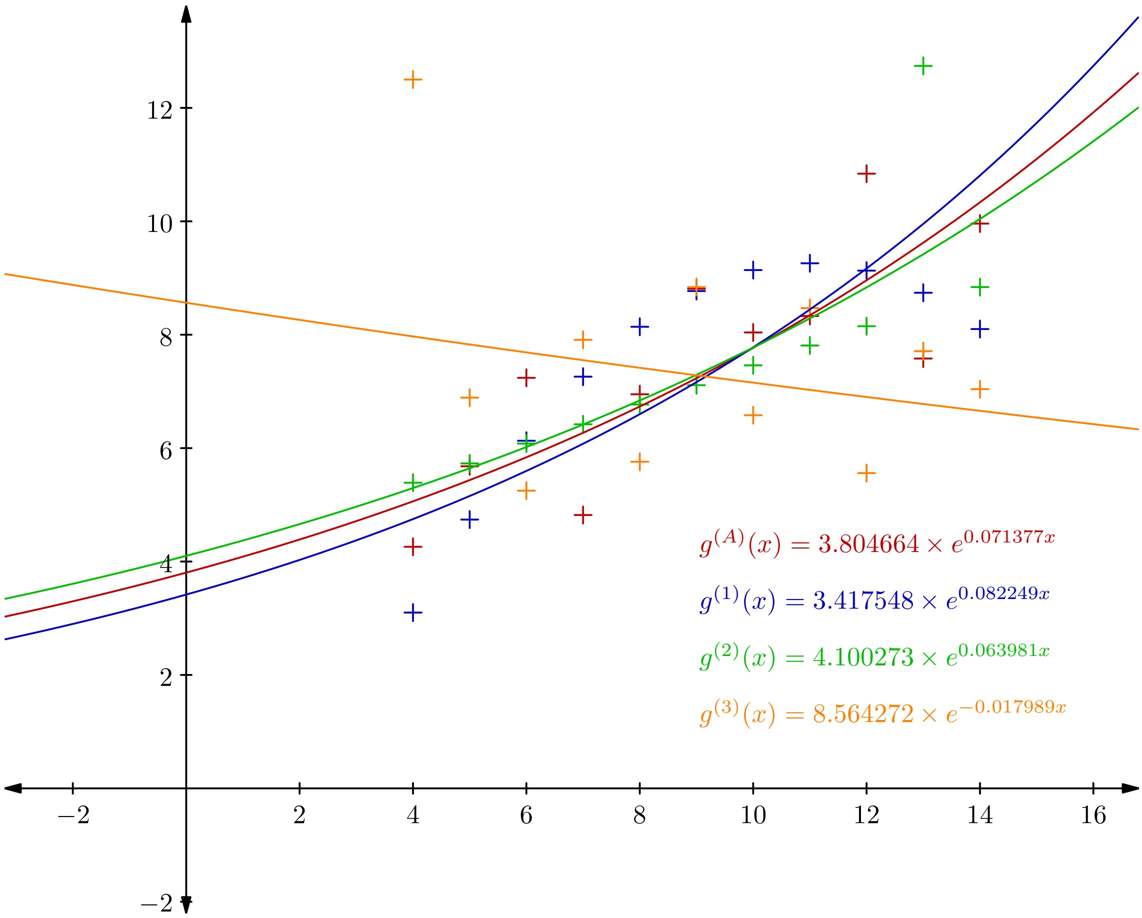
Série (1) :
Regression linéaire par une droite : $P(x) = 3.000909 + 0.500000 \cdot x$
Erreur moyenne : 0.967934

Série (2) :
Regression linéaire par une droite : $P(x) = 3.002455 + 0.499727 \cdot x$
Erreur moyenne : 0.715967



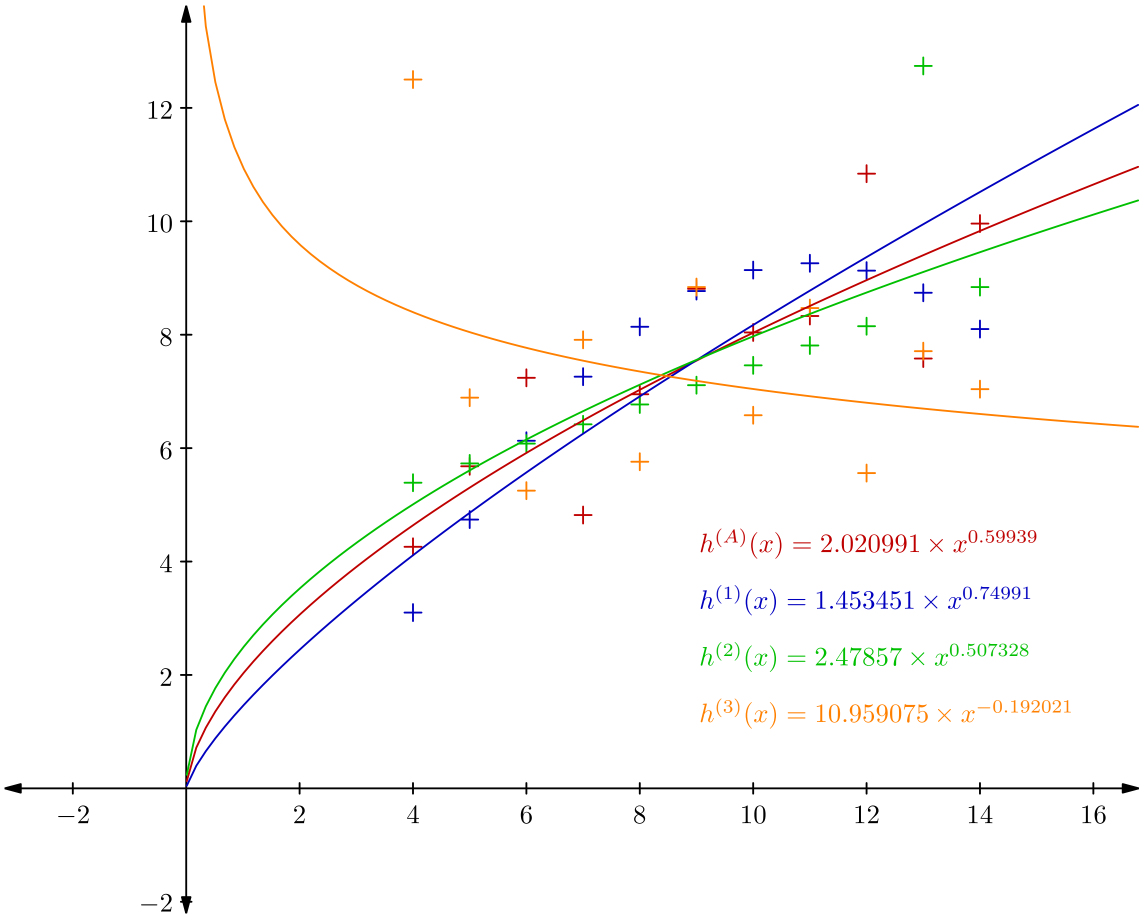
GRAPHIQUE 3.4.3 – Régression linéaire – (Tableau 3.4.3)

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

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "neuville.h"
#include "newton.h"
#include "polynome.h"
#include "reglin.h"
#include "useful.h"

int main (int argc, char ** argv)
{
  int n, j;
  int i=1;
  char c;
  while (i!=0)
  {
    printf("Menu principal : Interpolation et Approximation\n\n");
    n=0;
    while (n<2) //minimum 2 points
    {
      printf("Entrez n le nombre de points : ");
      scanf("%d", &n);
    }
    double** tab= (double**) malloc(2*sizeof(double*));
    for (i=0; i<2; i++)
    {
      tab[i]=(double*) malloc(n*sizeof(double));
    }
    for (i=0; i<n; i++)
    {
      printf("Entrez x[%d] : ", i+1);
      scanf("%lf", &tab[0][i]);
      printf("Entrez y[%d] : ", i+1);
      scanf("%lf", &tab[1][i]);
    }
    //fonction temporaire, utile uniquement pour la rédaction du rapport
    convertTabtoLatex(tab,n,1);
    i=1;
    while (i!=0 && i!=9)
    {
      clear(); //nettoyage écran
      printf("Tableau de valeurs :\n");
      for (i=0; i<2; i++)
      {
        for (j=0; j<n; j++)
        {
          if (0==j && 0==i)
          {
            printf(" x ");
          }
          else if (0==j && 1==i)
          {
```

```
            printf(" y ");
          }
          printf("| %+.5f ", tab[i][j]);
        }
        printf("\n");
      }
      printf("\nQuelle résolution utiliser ?\n");
      printf("1- Newton\n");
      printf("2- Neuville\n");
      printf("3- Régression Linéaire\n");
      printf("4- Approximation par une fonction exponentielle\n");
      printf("5- Approximation par une fonction puissance\n");
      printf("9- Nouvelle série de points (Menu principal)\n");
      printf("0- Quitter\n");
      printf("Votre choix : ");
      scanf("%d", &i);
      c='0';
      cleanBuffer(); //vidage buffer
      switch (i)
      {
        case 1:
          printf("Résolution par Newton ... \n");
          newton(tab,n);
          hitToContinue();
          break;
        case 2:
          printf("Résolution par Neuville ... \n");
          neuville(tab,n);
          hitToContinue();
          break;
        case 3:
          printf("Résolution par Régression linéaire ... \n");
          reglinD(tab,n);
          hitToContinue();
          break;
        case 4:
          printf("Résolution par Approximation par une fonction exponentielle... \n");
          reglinE(tab,n);
          hitToContinue();
          break;
        case 5:
          printf("Résolution par Approximation par une fonction puissance... \n");
          reglinP(tab,n);
          hitToContinue();
          break;
      }
      printf("\n\n");
    }

    //libération mémoire
    for (j=0; j<2; j++)
    {
      free(tab[j]);
    }
    free(tab);
    clear(); //nettoyage écran
  }
  return 0;
}
```

Figure 5.1 – Code : main.c

5.2 Fonctions associées au calcul polynômial

```
#ifndef POLYNOME__H
#define POLYNOME__H

typedef struct polynome
{
```

```
  int d; //degree
  double* poln; //coefficients
} polynome;

//fonctions sortie LaTeX
void convertTabtoLatex(double** tab, int n, int m); //sortie du tableau en LaTeX
void menuAffichage(polynome* P); //choix entre LaTeX et terminal
void afficherPoly(polynome* P, char* mode, ...); //mode:"console"||"latex"; console->terminal; latex->fichier
     (FILE* opt) format maths LaTeX.

//fonctions de manipulation des poly
polynome* creerPoly(int c,char* mode, ...); //c: nbre coefs;mode: "tableau"||"valeur";mode tableau -> paramè
     tre optionnel : tableau des coefs.
void redimensionnerPoly(polynome* P1); // Enleve les 0 inutiles dans le poly.
polynome* addPoly(polynome* P1, polynome* P2); // addition de 2 poly entre eux.
polynome* mulSPoly(double s, polynome* P1); // multiplication d'un poly par un scalaire.
polynome* mulPoly(polynome * P1, polynome* P2); // multiplication de 2 poly entre eux.

// fonctions de calcul d'images de fonctions
double imagePoly(polynome* P, double x); // La fonction calcule l'image de x par un poly.
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).
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.

//fonctions de statistiques
void ecartPoly(double** tab, int n, polynome* P);
void ecartExpo(double** tab, int n, double c, double d);
void ecartPui(double** tab, int n, double a, double b);

#endif
```

FIGURE 5.2 – Code : polynome.h

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdarg.h>
#include <string.h>
#include "polynome.h"

polynome* creerPoly(int c,char* mode, ...)
{
  int i;
  polynome* P=(polynome*) malloc(sizeof(polynome));
  P->d = c-1;
  P->poln = (double*) malloc(c*sizeof(double));
  if ((strcmp(mode,"valeur"))==0)//si mode = "valeur"
  {
    va_list ap;
    va_start(ap, mode);
    for (i=0; i<c; i++)
    {
      P->poln[i]=va_arg(ap, double);
    }
    va_end(ap);
  }
  else //si mode= "tableau"
  {
    va_list ap;
    va_start(ap, mode);
    double* tmp=va_arg(ap, double*);
    for (i=0; i<c; i++)
    {
      P->poln[i]=tmp[i];
    }
    free(tmp);
    va_end(ap);
  }
  redimensionnerPoly(P);
```

```
  return P;
}

void menuAffichage(polynome* P)
{
  FILE* fichier=fopen("resultat", "a+");
  int choix; // permet de choisir les options voulues
  printf("Voulez-vous afficher le polynome dans la sortie standard (1-Oui *-Non) ? ");
  scanf("%d",&choix);
  if(choix ==1)
  {
    afficherPoly(P,"console");
  }
  else
  {
          afficherPoly(P,"console");
    afficherPoly(P,"latex",fichier);
  }
  fclose(fichier);
}

void afficherPoly(polynome* P, char* mode, ...)
{
  int i;
  FILE* f;
  int c = (P->d) +1 ;
  if((strcmp(mode,"console")) == 0)
  {
    printf("P(x) = ");
    for(i=0;i<c;i++)
    {
      if(P->poln[i] > 0)
      {
        if(i != 0) { printf(" + "); }
        if(i == 0) { printf("%20.200f",P->poln[i]); }
        else if(i == 1) { printf("%20.200f * x",P->poln[i]); }
        else { printf("%20.200f * x^%d", P->poln[i],i); }
      }
      if(P->poln[i] < 0)
      {
        printf(" - ");
        if(i == 0) { printf("%20.200f",-(P->poln[i])); }
        else if(i == 1) { printf("%20.200f * x",-(P->poln[i])); }
        else { printf("%20.200f * x^%d", -(P->poln[i]), i); }
      }
    }
    printf("\n");
  }
  else
  {
    va_list ap;
    va_start(ap,mode);
    f = va_arg(ap,FILE*);
    fprintf(f,"$P(x) \\approx ");
    for(i=0; i<c; i++)
    {
      if(P->poln[i] > 0)
      {
        if(i != 0) { fprintf(f," + "); }
        if(i == 0) { fprintf(f,"%.6f",P->poln[i]); }
        else if(i == 1) { fprintf(f,"%.6f \\cdot x",P->poln[i]); }
        else { fprintf(f,"%.6f \\cdot x^{%d} ", P->poln[i],i); }
      }
      if(P->poln[i] < 0)
      {
        if(i == 0) { fprintf(f,"-%.6f", -(P->poln[i])); }
        else if(i == 1) { fprintf(f,"-%.6f \\cdot x", -(P->poln[i])); }
        else { fprintf(f,"- %.6f \\cdot x^{%d} ", -(P->poln[i]), i); }
      }
    }
    fprintf(f,"$\n\n");
```

```
    va_end(ap);
  }
}

void redimensionnerPoly(polynome* P1)
{
  int degre=P1->d;
  while((P1->poln[degre])==0)
  {
    degre--;
  }
  if(degre!=P1->d)
  {
    P1->d=degre;
    P1->poln= (double*) realloc(P1->poln, (degre+1)*sizeof(double));
  }
}

polynome* addPoly(polynome* P1, polynome* P2)
{
  // Rappel : Deg(P1+P2) <= max(Deg(P1),Deg(P2))
  int i;
  polynome* P=(polynome*)malloc(sizeof(polynome));
  if(P1->d > P2->d) // Deg(P1) > Deg(P2)
  {
    P->d = P1->d;
    P->poln = (double*) malloc((1+P1->d)*sizeof(double));
    for(i=0;i <= P2->d;i++)
    {
      P->poln[i] = P1->poln[i] + P2->poln[i];
    }
    for(i= P2->d +1; i<= P1->d; i++)
    {
      P->poln[i] = P1->poln[i];
    }
  }
  else if (P1->d < P2->d) // Deg(P2) > Deg(P1)
  {
    P->d = P2->d;
    P->poln = (double*) malloc((1+P2->d)*sizeof(double));
    for(i=0;i <= P1->d;i++)
    {
      P->poln[i] = P1->poln[i] + P2->poln[i];
    }
    for(i= P1->d +1; i<= P2->d; i++)
    {
      P->poln[i] = P2->poln[i];
    }
  }
  else // Deg(P2) = Deg(P1)
  {
    P->d = P2->d;
    P->poln = (double*) malloc((1+P2->d)*sizeof(double));
    for(i=0;i <= P1->d;i++)
    {
      P->poln[i] = P1->poln[i] + P2->poln[i];
    }
  }
  redimensionnerPoly(P);
  return P;
}

polynome* mulSPoly(double s, polynome* P1)
{
  int i;
  polynome* P=(polynome*) malloc(sizeof(polynome));
  P->d = P1->d;
  P->poln = (double*) malloc((1+P1->d)*sizeof(double));

  for(i=0;i <= P1->d;i++)
  {
```

```
    P->poln[i] = s*P1->poln[i] ;
  }
  redimensionnerPoly(P); //redimensionnement nécessaire si le scalaire est nul.
  return P;
}

polynome* mulPoly(polynome * P1, polynome* P2)
{
  int i,j;
  polynome* P=(polynome*)malloc(sizeof(polynome));
  P->d = P1->d + P2->d ;
  P->poln = (double*) malloc((1+P->d)*sizeof(double));

  for(i=0; i<=P->d; i++)
  {
    P->poln[i] = 0;
  }

  for(i=0; i<=(P2->d); i++)
  {
    for(j=0; j<=(P1->d); j++)
    {
      P->poln[i+j] = P->poln[i+j] + (P2->poln[i])*(P1->poln[j]);
    }
  }
  redimensionnerPoly(P);
  return P;
}

double imagePoly(polynome* P, double x)
{
  int i;
  double res = P->poln[0];
  for(i=1;i<=P->d;i++)
  {
    res = res + P->poln[i]*pow(x,i);
  }
  return res;
}

double imageExpo(double c, double d, double x)
{
  double res = 0.;
  res = c*exp(d*x);
  return res;
}

double imagePui(double a, double b, double x)
{
  double res = 0.;
  res = a*pow(x,b);
  return res;
}

void ecartPoly(double** tab, int n, polynome* P)
{
  int i;
  double moyecart= 0.;
  for(i=0;i<n;i++)
  {
    moyecart = moyecart + fabs((imagePoly(P,tab[0][i])-tab[1][i]));
  }
  moyecart = moyecart/n;
  printf("Erreur moyenne : %20.18f",moyecart);
}

void ecartExpo(double** tab, int n, double c, double d)
{
  int i;
  double moyecart= 0.;
  for(i=0;i<n;i++)
```

```
  {
    moyecart = moyecart + fabs((imageExpo(c,d,tab[0][i])-tab[1][i]));
  }
  moyecart = moyecart/n;
  printf("Erreur moyenne : %20.18f",moyecart);
}

void ecartPui(double** tab, int n, double a, double b)
{
  int i;
  double moyecart= 0.;
  for(i=0;i<n;i++)
  {
    moyecart = moyecart + fabs((imagePui(a,b,tab[0][i])-tab[1][i]));
  }
  moyecart = moyecart/n;
  printf("Erreur moyenne : %20.18f",moyecart);
}

void convertTabtoLatex(double** tab, int n, int m)
{
  FILE* fichier = fopen("resultat","a+");
  int i,j;
  //déclaration de l'environnement et de n+1 colonnes
  fprintf(fichier,"\\begin{tabular}{|");
  for(i=0;i<=n;i++)
  {
    fprintf(fichier," c |");
  }
  fprintf(fichier,"}\n \\hline \n");

  //remplissage des cases
  for(i=0;i<2;i++)
  {
  if(i==0) {fprintf(fichier,"$x_{i}$ & ");}
  if(i==1) {fprintf(fichier,"$y_{i}$ & ");}
  for(j=0;j<n;j++)
  {
    if(j!=(n-1)) {fprintf(fichier,"$%f$ & ",tab[i][j]);}
    else {fprintf(fichier,"$%f$ ",tab[i][j]);}
  }
  fprintf(fichier,"\\\\ \n \\hline \n");
  }
  fprintf(fichier,"\\end{tabular}\n\n");
  fclose(fichier);
}
```

FIGURE 5.3 – Code : polynome.c