

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
2	Interpolation	3
2.1	Méthode de Newton	3
2.1.1	Programme	3
2.2	Méthode de Neuville	5
2.2.1	Programme	5
2.3	Résultats de tests	6
2.4	Comparaison	8
3	Approximation	9
3.1	Régression linéaire	9
3.1.1	Programme	9
3.2	Résultats de tests	12
4	Conclusion	13

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 : 2
(Saisie de la série de points...)

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

FIGURE 1.1 – Aperçu : Menu Principal

2 Interpolation

2.1 Méthode de Newton

2.1.1 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     double** t= (double**) malloc(n*sizeof(double*));
11     for (i=0; i<n; i++)
12     {
13         t[i]= (double*) malloc((i+1)*sizeof(double));
14     }
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
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
31     //tableau de polynomes
32     polynome** tabP= (polynome**) malloc(n*(sizeof(polynome*)));
33     for (i=0; i<n; i++)
34     {
35         tabP[i]=(polynome*) malloc(sizeof(polynome));
36     }
37     tabP[0]->d=0;
38     tabP[0]->poln=(double*) malloc(sizeof(double));
39     tabP[0]->poln[0]=t[n-1][n-1];
40
41     for (i=1; i<n; i++)
42     {
43         tabP[i]=addPoly(creerPoly(1,"valeur",t[n-1-i][n-1-i]),mulPoly(creerPoly(2,"valeur",-tab[0][n-1-i], 1.),
44             tabP[i-1]));
45     }
46     redimensionnerPoly(tabP[n-1]);
47
48     //affichage
49     menuAffichage(tabP[n-1]);
50     ecartPoly(tab,n,tabP[n-1]);
51     printf("\n");
```

```

51 |
52 | //libération mémoire
53 | for(i=0;i<n;i++)
54 | {
55 |     free(tabP[i]->poln);
56 |     free(tabP[i]);
57 |     free(t[i]);
58 | }
59 | free(tabP);
60 | free(t);
61 | }

```

FIGURE 2.1 – Code : newton.c

2.2 Méthode de Neville

2.2.1 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*** t= (polynome***) malloc(n*sizeof(polynome**));
11     for (i=0; i<n; i++)
12     {
13         t[i]= (polynome**) malloc((i+1)*sizeof(polynome*));
14     }
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
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]=mulSPoly((1/((tab[0][i-k])-(tab[0][i]))),addPoly(mulPoly(creerPoly(2,"valeur", -(tab[0][i]),
28                 1.), t[i-1][k-1]), mulPoly(creerPoly(2, "valeur", tab[0][i-k], -1.),t[i][k-1])));
29         }
30     }
31     //polynome à retourner
32     redimensionnerPoly(t[n-1][n-1]);
33
34     //affichage
35     menuAffichage(t[n-1][n-1]);
36     ecartPoly(tab,n,t[n-1][n-1]);
37     printf("\n");
38
39     //libération mémoire
40     for(i=0;i<n;i++)
41     {
42         for(k=0;k<i;k++)
43         {
44             free(t[i][k]->poln);
45             free(t[i][k]);
46         }
47         free(t[i]);
48     }
49     free(t);
50 }
```

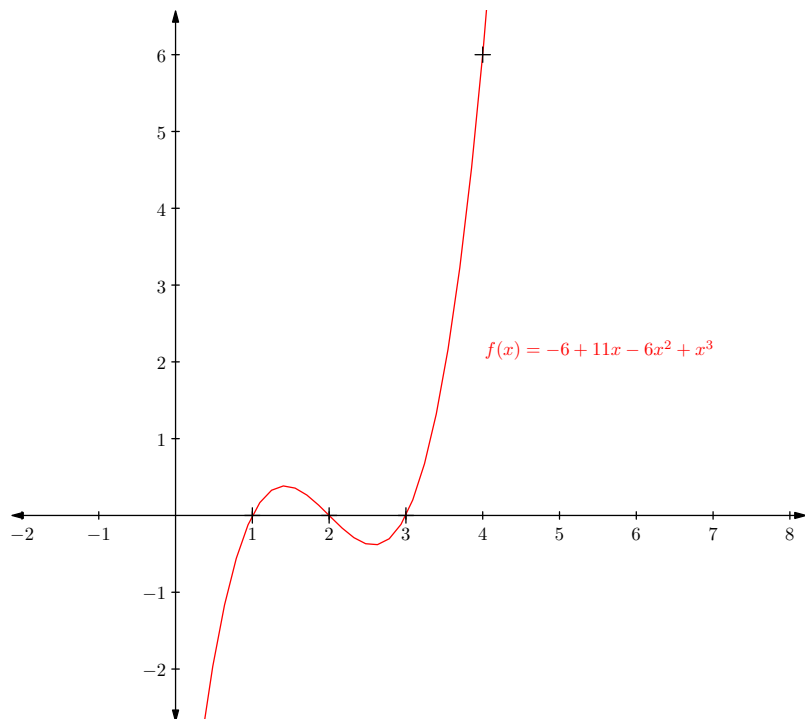
FIGURE 2.2 – Code : neville.c

2.3 Résultats de tests

x_i	1	2	3	4
y_i	0	0	0	6

TABLEAU 2.3.1 – Série 1

Méthode de Newton : $P(x) = -6.000000 + 11.000000 \cdot x - 6.000000 \cdot x^2 + 1.000000 \cdot x^3$
Méthode de Neuville : $P(x) = -6.000000 + 11.000000 \cdot x - 6.000000 \cdot x^2 + 1.000000 \cdot x^3$



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

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

TABLEAU 2.3.2 – Sériel dépense

Méthode de Newton : $P(x) = 524166663.148073 - 8065079.743259 \cdot x + 55479.774423 \cdot x^2 - 224.689243 \cdot x^3 + 0.593281 \cdot x^4 - 0.001067 \cdot x^5 + 0.000001 \cdot x^6 - 0.000000 \cdot x^7 + 0.000000 \cdot x^8 - 0.000000 \cdot x^9 + 0.000000 \cdot x^{10}$

Méthode de Neville : $P(x) = 524166663.148078 - 8065079.743259 \cdot x + 55479.774423 \cdot x^2 - 224.689243 \cdot x^3 + 0.593281 \cdot x^4 - 0.001067 \cdot x^5 + 0.000001 \cdot x^6 - 0.000000 \cdot x^7 + 0.000000 \cdot x^8 - 0.000000 \cdot x^9 + 0.000000 \cdot x^{10}$

Pourcentage d'erreur : 0.000046

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

2.4 Comparaison

3 Approximation

3.1 Régression linéaire

3.1.1 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 |     int i, j;
10 |    if (strcmp(fn,"exponentielle")==0)
11 |    {
12 |        for (j=0; j<n; j++)
13 |        {
14 |            to[0][j]=from[0][j];
15 |        }
16 |        for (j=0; j<n; j++)
17 |        {
18 |            to[1][j]=log(from[1][j]);
19 |        }
20 |    }
21 |    else if (strcmp(fn,"puissance")==0)
22 |    {
23 |        for (i=0; i<2; i++)
24 |        {
25 |            for (j=0; j<n; j++)
26 |            {
27 |                to[i][j]=log(from[i][j]);
28 |            }
29 |        }
30 |    }
31 | }
32 |
33 | double moyenneElements(double** tab,int l, int n)
34 | {
35 |     double resultat = 0.;
36 |     double cpt = 0.;
37 |     int i;
38 |     for(i=0;i<n;i++)
39 |     {
40 |         resultat = resultat + tab[l][i];
41 |         cpt = cpt + 1.;
42 |     }
43 |     resultat = resultat/cpt;
44 |     return resultat;
45 | }
46 |
47 | double moyenneElementsCarres(double** tab,int l, int n)
48 | {
49 |     double resultat = 0;
50 |     double cpt = 0;
51 |     int i;
```

```

52     for(i=0;i<n;i++)
53     {
54         resultat = resultat + pow(tab[1][i],2);
55         cpt = cpt + 1;
56     }
57     resultat = resultat/cpt;
58     return resultat;
59 }
60
61 double moyenneProduitElements(double** tab, int n)
62 {
63     double resultat = 0;
64     double cpt = 0;
65     int i;
66     for(i=0;i<n;i++)
67     {
68         resultat = resultat + tab[0][i]*tab[1][i];
69         cpt = cpt + 1;
70     }
71     resultat = resultat/cpt;
72     return resultat;
73 }
74
75 reglinD(double** tab, int n)
76 {
77     double a0 = 0;
78     double a1 = 0;
79     double xb, yb, xcb, xyb; // b pour barre et c pour carre
80     printf("Nous cherchons le polynome de degré 1 sous la forme a0 + a1*x.\n");
81
82     //calculs
83     xb = moyenneElements(tab,0,n);
84     yb = moyenneElements(tab,1,n);
85     xcb = moyenneElementsCarres(tab,0,n);
86     xyb = moyenneProduitElements(tab,n);
87
88     a1 = (xyb-xb*yb)/(xcb-pow(xb,2));
89     a0 = yb-xb*a1;
90
91     // creation et affichage du polynome
92     polynome *P = creerPoly(2,"valeur",a0,a1);
93     menuAffichage(P);
94
95     //statistiques
96     ecartPoly(tab,n,P);
97     printf("\n");
98
99     //libération mémoire
100    free(P->poln);
101    free(P);
102 }
103
104 reglinE(double** tab, int n) //y=c(e^(dx)) <=> ln(y)=ln(c)+xd => c=e^(a0) & d=a1
105 {
106     int i;
107     double c = 0;
108     double d = 0;
109     double a0 = 0;
110     double a1 = 0;
111     double xb, yb, xcb, xyb; // b pour barre et c pour carre
112     double** t = (double**) malloc(2*sizeof(double*)); // contiendra le mapping de tab
113     for(i=0;i<2;i++)
114     {
115         t[i] = (double*) malloc (n*sizeof(double));
116     }
117     printf("Nous cherchons une approximation sous la forme c*(e^(d*x)).\n");
118
119     //calculs
120     mapping(tab, t, n, "exponentielle");
121     xb = moyenneElements(t,0,n);
122     yb = moyenneElements(t,1,n);

```

```

123   xcb = moyenneElementsCarres(t,0,n);
124   xyb = moyenneProduitElements(t,n);
125
126   a1 = (xyb-xb*yb)/(xcb-pow(xb,2.));
127   a0 = yb-xb*a1;
128   d = a1;
129   c = exp(a0);
130
131   //affichage
132   printf("P(x) = %f*exp(%f*x)",c,d);
133
134   //statistiques
135   ecartExpo(tab,n,c,d);
136   printf("\n");
137
138   //libération mémoire
139   for (i=0; i<2; i++)
140   {
141       free(t[i]);
142   }
143   free(t);
144 }
145
146 reglinP(double ** tab, int n) //y=a(x^b) <=> ln(y)=ln(a)+b*ln(x) => a=e^(a0) & b=a1
147 {
148     int i;
149     double a = 0.;
150     double b = 0.;
151     double a0 = 0.;
152     double a1 = 0.;
153     double xb, yb, xcb, xyb; // b pour barre et c pour carre
154     double** t = (double**) malloc(2*sizeof(double*)); // contiendra le mapping de tab
155     for(i=0;i<2;i++)
156     {
157         t[i] = (double*) malloc (n*sizeof(double));
158     }
159     printf("Nous cherchons une approximation sous la forme a*(x^(b)).\n");
160
161     //calculs
162     mapping(tab, t, n, "puissance");
163     xb = moyenneElements(t,0,n);
164     yb = moyenneElements(t,1,n);
165     xcb = moyenneElementsCarres(t,0,n);
166     xyb = moyenneProduitElements(t,n);
167
168     a1 = (xyb-xb*yb)/(xcb-pow(xb,2));
169     a0 = yb-xb*a1;
170     b = a1;
171     a = exp(a0);
172
173     //affichage
174     printf("P(x) = %f*x^(%f)",a,b);
175     //statistiques
176     ecartPui(tab,n,a,b);
177     printf("\n");
178
179     //libération mémoire
180     for (i=0; i<2; i++)
181     {
182         free(t[i]);
183     }
184     free(t);
185 }

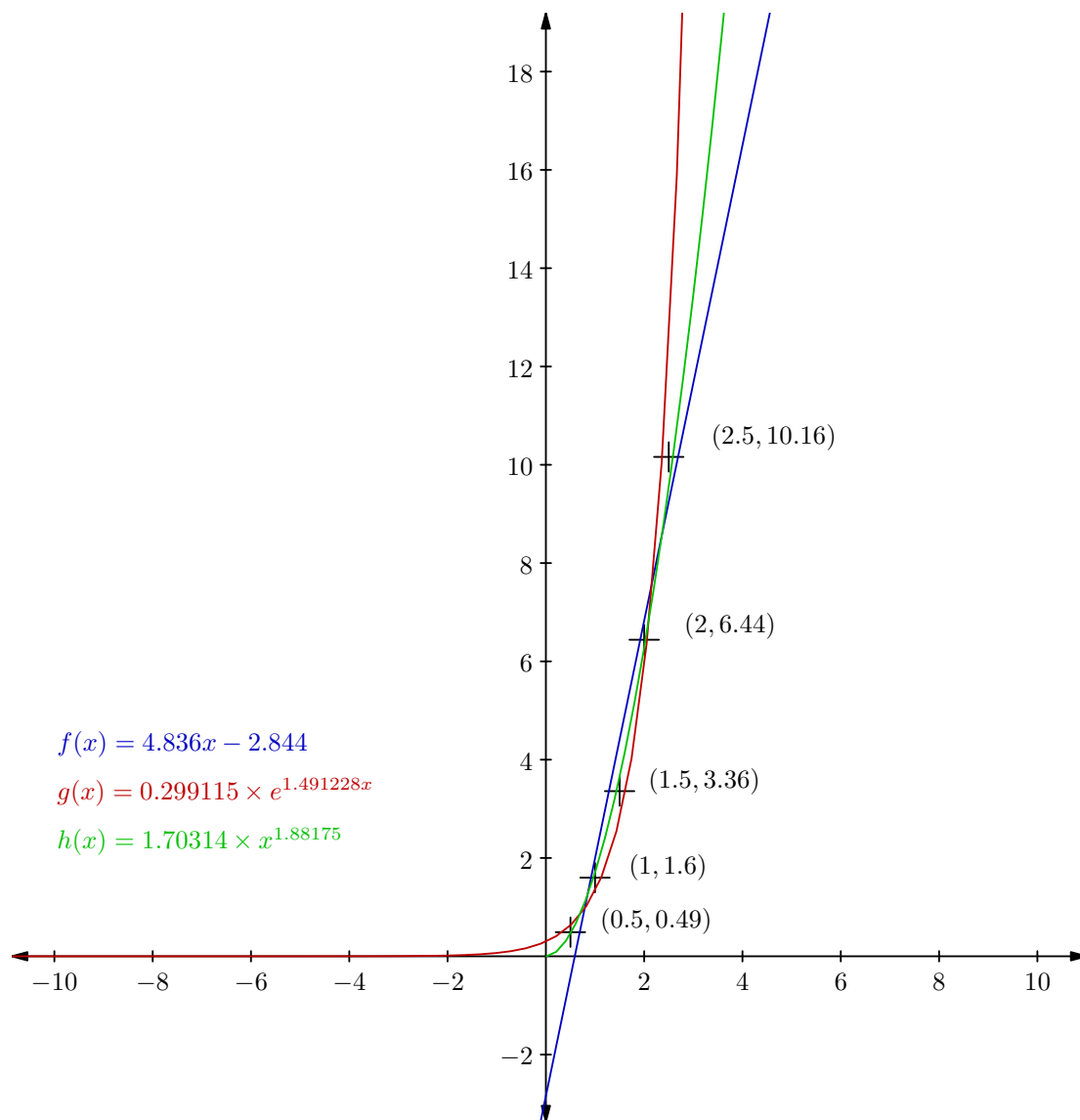
```

FIGURE 3.1 – Code : reglin.c

3.2 Résultats de tests

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

TABLEAU 3.2.1 – Série 1



GRAPHIQUE 3.2.1 – Régression linéaire – (Tableau 3.2.1)

4 Conclusion