

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 Lagrange	3
2.1.1	Programme	3
2.2	Méthode de Newton	3
2.2.1	Programme	3
2.3	Méthode de Neuville	4
2.3.1	Programme	4
2.3.2	Résultats de tests	5
2.4	Comparaison	5
3	Approximation	6
3.1	Régression linéaire	6
3.1.1	Programme	6
3.1.2	Résultats de tests	9
4	Conclusion	10

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

Quelle résolution utiliser ?
1- Lagrange
2- Newton
3- Neuville
4- Régression Linéaire
0- Quitter
```

FIGURE 1.1 – Aperçu : Menu Principal

2 Interpolation

2.1 Méthode de Lagrange

2.1.1 Programme

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

FIGURE 2.1 – Code : lagrange.c

2.2 Méthode de Newton

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 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 | //affichage
48 | menuAffichage(tabP[n-1]);
49 | printf("\n");
50 |
51 | for(i=0;i<n;i++)
52 | {
53 |     free(tabP[i]->poln);
54 |     free(tabP[i]);
55 |     free(t[i]);
56 | }
57 | free(tabP);
58 | free(t);
59 | }

```

FIGURE 2.2 – Code : newton.c

2.3 Méthode de Neville

2.3.1 Programme

```

1 | #include <stdio.h>
2 | #include <stdlib.h>
3 | #include <string.h>
4 | #include <math.h>
5 | #include "polynome.h"
6 |
7 | void newville (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 |    printf("\n");
37 |    //libération mémoire

```

```

38 | for(i=0;i<n;i++)
39 | {
40 |     for(k=0;k<i;k++)
41 |     {
42 |         free(t[i][k]->poln);
43 |         free(t[i][k]);
44 |     }
45 |     free(t[i]);
46 | }
47 | free(t);
48 | }

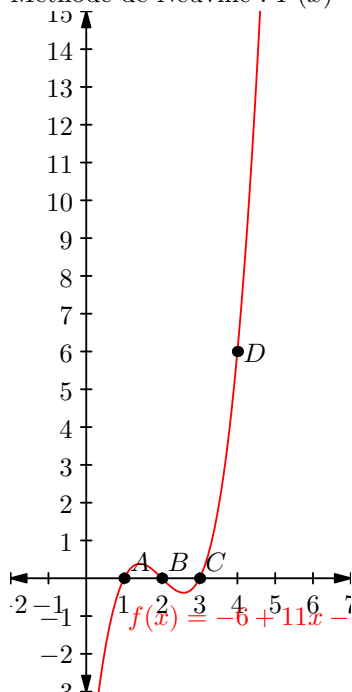
```

FIGURE 2.3 – Code : neuville.c

2.3.2 Résultats de tests

x_i	1.000000	2.000000	3.000000	4.000000
y_i	0.000000	0.000000	0.000000	6.000000

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



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     xb = moyenneElements(tab,0,n);
82     yb = moyenneElements(tab,1,n);
83     xcb = moyenneElementsCarres(tab,0,n);
84     xyb = moyenneProduitElements(tab,n);
85
86     a1 = (xyb-xb*yb)/(xcb-pow(xb,2));
87     a0 = yb-xb*a1;
88
89     // creation et affichage du polynome
90     polynome *P = creerPoly(2,"valeur",a0,a1);
91     menuAffichage(P);
92 }
93
94 reglinE(double** tab, int n) //y=c(e^(dx)) => ln(y)=ln(c)+xd => c=e^(a_0) & d=a1
95 {
96     double c = 0;
97     double d = 0;
98     double a0 = 0;
99     double a1 = 0;
100    double xb, yb, xcb, xyb; // b pour barre et c pour carre
101    printf("Nous cherchons une approximation sous la forme c*(e^(d*x)).\n");
102    double** t = (double**) malloc(2*sizeof(double*)); // contiendra le mapping de tab
103    int i;
104    for(i=0;i<2;i++)
105    {
106        t[i] = (double*) malloc (n*sizeof(double));
107    }
108    mapping(tab, t, n, "exponentielle");
109    xb = moyenneElements(t,0,n);
110    yb = moyenneElements(t,1,n);
111    xcb = moyenneElementsCarres(t,0,n);
112    xyb = moyenneProduitElements(t,n);
113
114    a1 = (xyb-xb*yb)/(xcb-pow(xb,2.));
115    a0 = yb-xb*a1;
116
117    d = a1;
118    c = exp(a0);
119    printf("P(x) = %f*exp(%f*x)",c,d);
120 }
121
122 reglinP(double ** tab, int n) //y=a(x^b) => ln(y)=ln(a)+b*ln(x) => a=e^(a_0) & b=a1

```



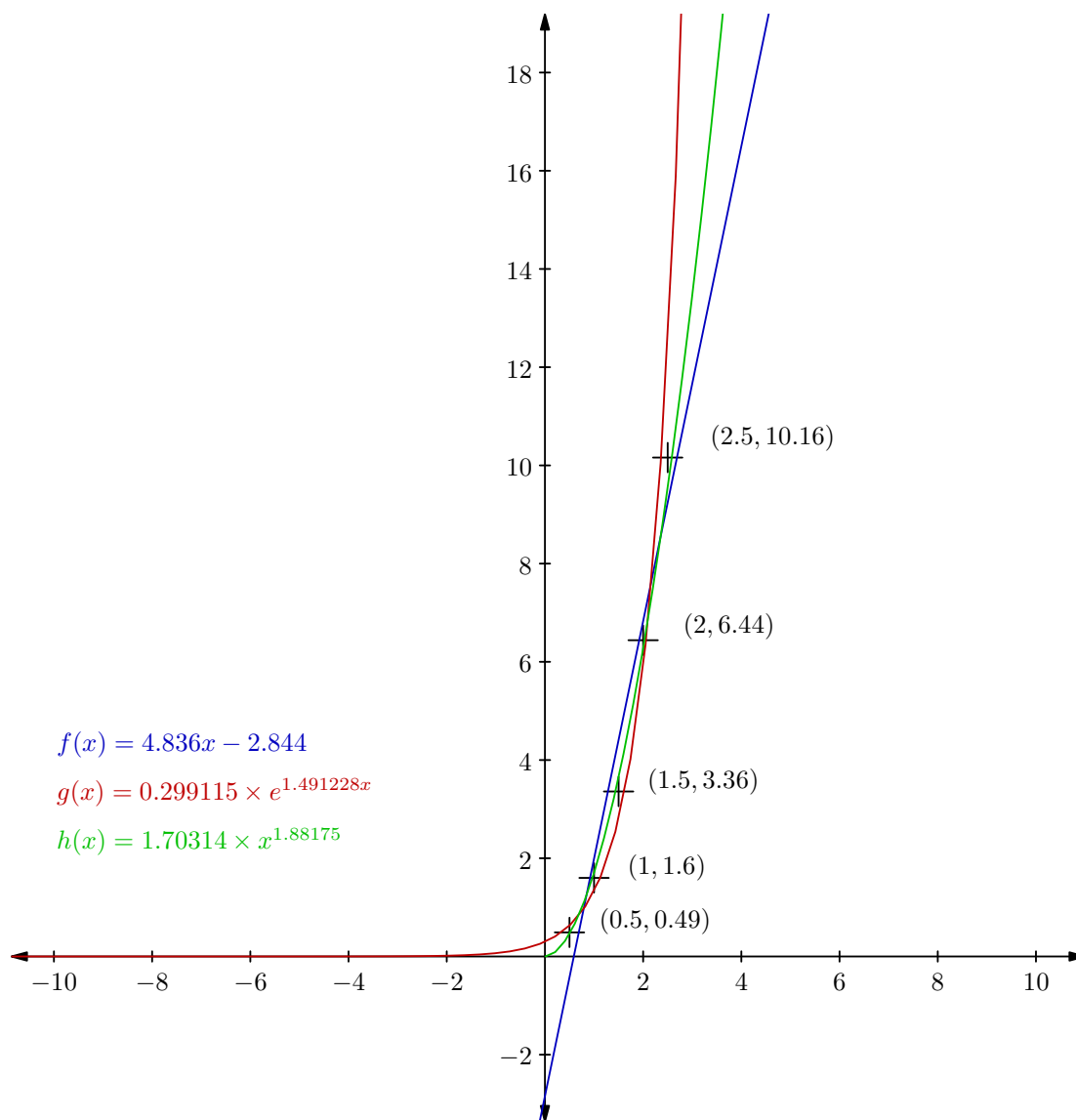
```

123 {
124     double a = 0.;
125     double b = 0.;
126     double a0 = 0.;
127     double a1 = 0.;
128     double xb, yb, xcb, xyb; // b pour barre et c pour carre
129     printf("Nous cherchons une approximation sous la forme a*(x^(b)).\n");
130     double** t = (double**) malloc(2*sizeof(double*)); // contiendra le mapping de tab
131     int i;
132     for(i=0;i<2;i++)
133     {
134         t[i] = (double*) malloc (n*sizeof(double));
135     }
136     mapping(tab, t, n, "puissance");
137     xb = moyenneElements(t,0,n);
138     yb = moyenneElements(t,1,n);
139     xcb = moyenneElementsCarres(t,0,n);
140     xyb = moyenneProduitElements(t,n);
141
142     a1 = (xyb-xb*yb)/(xcb-pow(xb,2));
143     a0 = yb-xb*a1;
144
145     b = a1;
146     a = exp(a0);
147     printf("P(x) = %f*x^(%f)",a,b);
148 }

```

FIGURE 3.1 – Code : reglin.c

3.1.2 Résultats de tests



4 Conclusion