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

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

FIGURE 1.1 – Apercu : Menu Principal

2 Interpolation

2.1 Méthode de Lagrange

2.1.1 Programme

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

FIGURE 2.1 – Code : lagrange.c

2.1.2 Résultats de tests

2.2 Méthode de Newton

2.2.1 Programme

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

void newton (double ** tab, int n)
{
  int i, k;
  double** t= (double**) malloc(n*sizeof(double*));
  for (i=0; i<n; i++)
  {
    t[i]= (double*) malloc(n*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 polynomes
  polynome** tabP= (polynome**) malloc(n*(sizeof(polynome*)));
  for (i=0; i<n; i++)
  {
```

```
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.),
            tabP[i-1]));
44    }
45    redimensionnerPoly(tabP[n-1]);
46
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

2.2.2 Résultats de tests

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 neuville (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(n*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]),
              1.), t[i-1][k-1]), mulPoly(creerPoly(2, "valeur", tab[0][i-k], -1.),t[i][k-1])));
28      }
29    }
30    //polynome àretourner
31    redimensionnerPoly(t[n-1][n-1]);
32
```

```
33    //affichage
34    menuAffichage(t[n-1][n-1]);
35    printf("\n");
36
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

2.4 Comparaison

3 Approximation

3.1 Regression linéaire

3.1.1 Programme

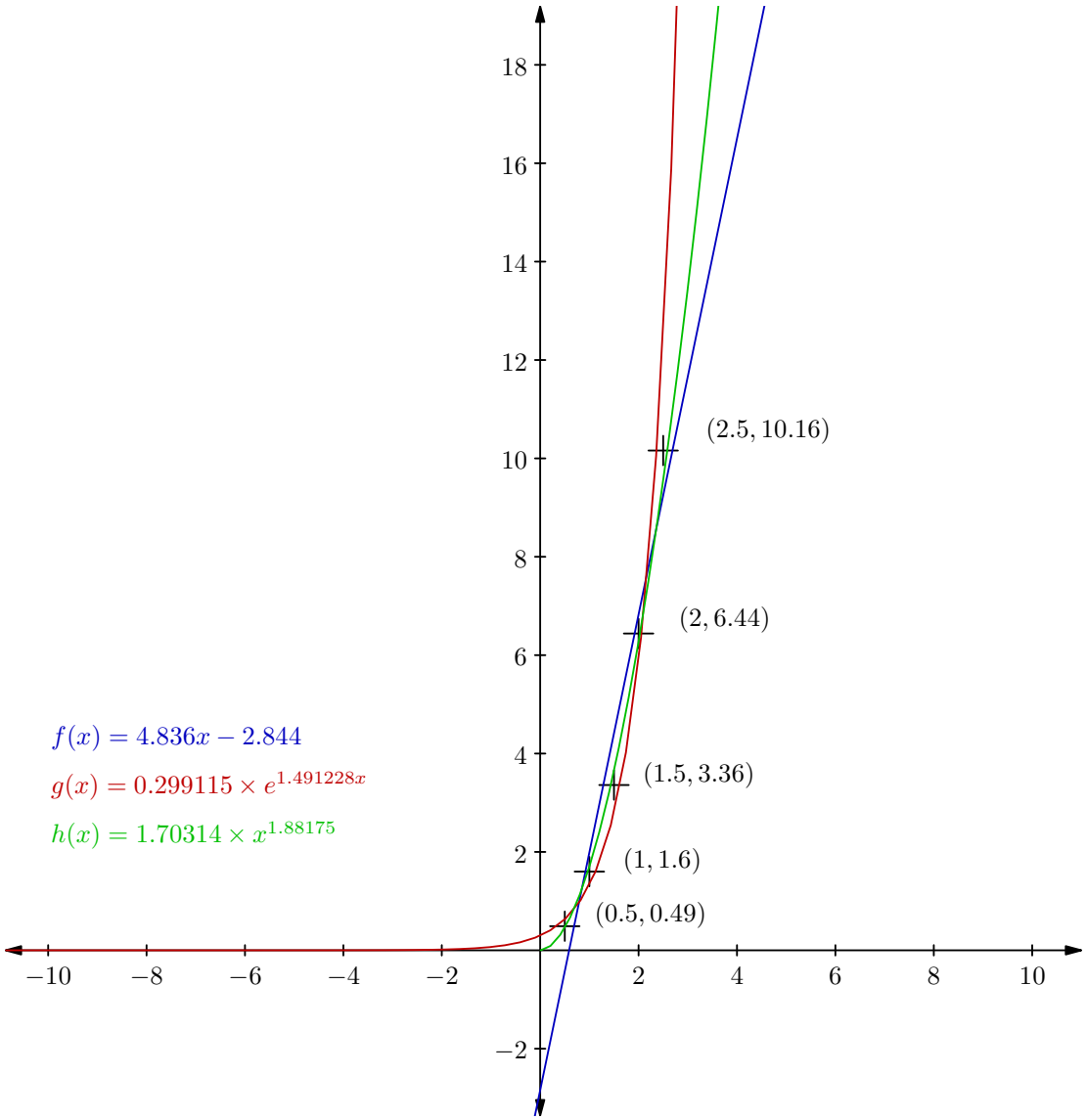
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "polynome.h"

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

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;
  double cpt = 0;
  int i;
```

```
  for(i=0;i<n;i++)
  {
    resultat = resultat + pow(tab[l][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");
  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 du polynome
  polynome *P = creerPoly(2,"valeur",a0,a1);
  menuAffichage(P);
}

reglinE(double** tab, int n) //y=c(e^(dx)) => ln(y)=ln(c)+xd => c=e^(a_0) & d=a1
{
  double c = 0;
  double d = 0;
  double a0 = 0;
  double a1 = 0;
  double xb, yb, xcb, xyb; // b pour barre et c pour carre
  printf("Nous cherchons une approximation sous la forme c*(e^(d*x)).\n");
  double** t = (double**) malloc(2*sizeof(double*)); // contiendra le mapping de tab
  int i;
  for(i=0;i<2;i++)
  {
    t[i] = (double*) malloc (n*sizeof(double));
  }
  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);
  printf("P(x) = %f*exp(%f*x)",c,d);
}

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