PROJET : RECHERCHE DE POINTS PARTICULIERS PAR INTERPOLATION

1. Objectif

Les objectifs de ce projet sont multiples :

- Approfondir la maîtrise de la programmation en langage C : pointeurs, fichiers, mémoire dynamique.
- S'approprier les outils vus en technique scientifique : Lagrange, Horner…
- Découvrir les limites des méthodes, trouver des techniques pour contourner ces difficultés.
- Mettre en œuvre une démarche raisonnée consistant à construire des outils numérique, les valider puis les évaluer en critiquant les résultats obtenus pour améliorer les outils et obtenir, *in fine*, le résultat souhaité.

Pour ce faire, ce projet va vous conduire à résoudre un problème que tout scientifique est amené à résoudre régulièrement : compléter un relevé expérimental composé de n+1 points $\{x_k, y_k\}$ avec $k \in [0; n]$. Pour simplifier la suite de l'exposé, on appelle F, la fonction reliant les abscisses aux ordonnées : $y_k = F(x_k)$

Le problème choisi sera de déterminer avec précision la plus haute valeur d'une courbe : déterminer l'abscisse d'un maximum local (ou sommet). Cela implique d'être à même de calculer des points entre ceux que comporte le relevé expérimental. La détermination de ces ordonnées n'est pas compliquée si l'on connait la fonction *F(x)*. Bien entendu, cette fonction n'est pas connue, et l'on remplace localement la fonction F(x) par le polynôme d'interpolation.

Deux approches seront utilisées : la dichotomie et la recherche de la racine de F'(x).

1.i. Dichotomie.

Cette approche utilise un algorithme dérivé de celui de la dichotomie.

L'idée est de déplacer une fenêtre de largeur α : [x_k - α ; x_k + α] en diminuant la largeur α cette fenêtre lorsque le maximum se trouve dans cette fenêtre en suivant l'algorithme de la dichotomie. Le plus simple est une illustration pour comprendre cet algorithme.

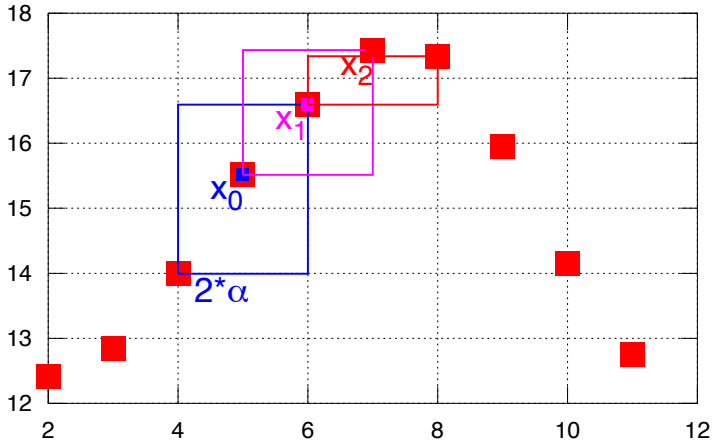

Fig. 1 : méthode de dichotomie, phase d'approche de la zone du sommet entre deux phases de dichotomie

L'algorithme est le suivant. On calcule $F(x_k - \alpha)$, $F(x_k)$ et $F(x_k + \alpha)$.

Si la plus grande des trois valeur est $F(x_k - \alpha)$ on calcule $x_{k+1} = x_k - \alpha$.

Si la plus grande des trois valeurs est $F(x_k + \alpha)$ on calcule $x_{k+1} = x_k + \alpha$

Si la plus grande des trois valeurs est $F(x_k)$ on divise α par 2.

On recommence le processus tant que $\alpha > 2\epsilon$, la précision souhaitée.

Cette méthode est simple à programmer, c'est celle que vous réaliserez en premier.

1.ii. Racine de la dérivée

Cette méthode utilise le fait qu'en un maximum local, la dérivée d'une fonction s'annule (cf. cours de maths L1-Semestre 2).

La difficulté de cette seconde approche est qu'il va falloir chercher la racine de la fonction *F'(x)* dans l'intervalle [a, b], or *F(x)* n'est pas connue. Aussi c'est la racine de la dérivée du polynôme d'interpolation dans l'intervalle [a, b] qui sera déterminée.

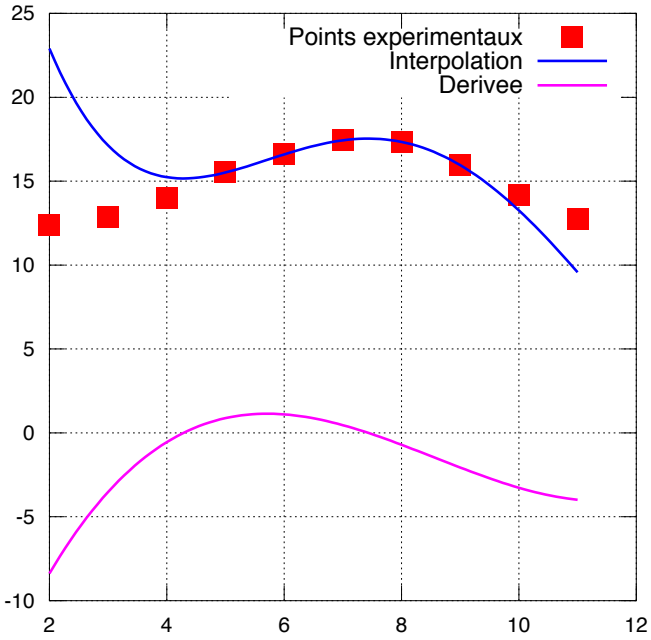

Fig. 2 : Points expérimentaux et polynôme d'interpolation. Noter la zone d'interpolation. La dérivée est calculée à partir du polynôme d'interpolation. Noter qu'elle a deux racines dont une, hors de la zone d'interpolation, est à rejeter.

Pour déterminer les valeurs prises par la dérivée du polynôme d'interpolation, nous faisons le choix d'utiliser la méthode d'Horner, il nous faut donc l'expression littérale du polynôme d'interpolation. Dans ce cas l'expression du polynôme d'interpolation à utiliser fait appel à $\Phi(x) = \prod_{k=0}^{n}(x - x_k)$.

Il faut donc être capable de calculer $\Phi(x)$, la méthode de Leverrier-Newton permet ce calcul. Cette méthode est décrite plus loin. Une fois que $\Phi(x)$ est connu, il faut calculer les coefficients de Lagrange :

$$L_k(x) = \frac{\Phi(x)}{(x - x_k)\Phi'(x_k)}$$

Les $L_k(x)$ sont le résultat de la division de $\Phi(x)$ par $(x\text{-}x_k)$, ce que l'on peut réaliser à l'aide d'un schéma de Horner, puis par $\Phi'(x_k)$, que l'on peut calculer à l'aide d'un second schéma de Horner.

Une fois les $L_k(x)$ calculés on peut calculer $P(x) = \sum_{k=1}^{n} L_k(x)Y_k$

La racine de *F'(x)*, donc de *P(x)* sera déterminée par la méthode de Newton.

En résumé pour cette seconde méthode, nous aurons besoin de savoir :

- Déterminer l'expression du polynôme d'interpolation par la méthode de Lagrange, ce qui impose de savoir :
  - calculer $\Phi(x)$ à l'aide de la méthode de Leverrier-Newton (cf ci-dessous)
  - savoir calculer les $L_k(x)$.
- Calculer la dérivée de ce polynôme d'interpolation : méthode d'Horner.
- Déterminer la racine de la dérivée de ce polynôme : méthode de Newton.

Pour chacun de ces points le plus simple est d'écrire une fonction dédiée.

2. Rappels

2.i. Gestion dynamique de la mémoire.

Pour gérer dynamiquement la mémoire et créer des variables dimensionnées, on utilisera la fonction malloc. Par exemple pour créer une variable dimensionnée, a une dimension de 10 éléments, de façon statique on déclare :

```
int a[10] ;
```

ou

```
typedef int vect[10]
…
vect a;
```

De manière dynamique :

```
int *p;
p = (int *) malloc( 10*sizeof( int ) );
// pointeur = ( transtypage type du pointeur) malloc( nombre*sizeof( type éléments ))
```

Pour des variables a une dimension, cela est donc simple. Il suffit de ne pas oublier de libérer la mémoire en fin de programme.

Pour des variables a deux dimensions, cela est plus complexe (encore que...)

Avec un typedef, on conseille de procéder en deux étapes :

```
typedef int vect[10];
typedef vect mat[10];
…
mat a;
```

Dans ce cas, a[3] est un vect, c'est à aussi un pointeur… puisque c'est le nom d'une variable dimensionnée. Le type de
a[3] est donc int *. Par conséquent, le type de a est int **.

Dans le cas d'une variable dimensionnée, on peut créer une variable à deux dimensions[1] de la manière suivante :

```
int **p;
p = ( int **) malloc ( Nligne*sizeof( int *) );
// chaque ligne est une variable dimensionnée de type int *, p est de type int **;
```

Puis pour chaque ligne k :

```
p[k] = ( int *) malloc( ncolone*sizeof( int ) );
// p[k] est de type int *, p[k][l] est de type int
```

Notez qu'il est alors possible d'avoir des variables dimensionnées ayant un nombre de colonnes différent pour chaque ligne…

2.ii. Polynôme d'interpolation : Méthode de Newton

Soient n+1points {x,y}, régulièrement espacés d'un pas h :

x_0	$x_0 + h$	…	$x_0 + i*h$	…	$x_0 + n*h$
$y_0=f(x_0)$	$y_1=f(x_1)$	…	$y_i=F(x_i)$	…	$y_n=F(x_n)$

On cherche à déterminer pour un α donné $\{\alpha \in [x_0, x_0+n*h]\}$ la valeur de $y = f(\alpha)$ sans connaître l'expression de $f(x)$. Cette fonction $f(x)$ peut être représentée par un polynôme $P_n(x)$ de degré n passant par les n+1 points {x,y}. $P_n(\alpha)$ s'écrit :

$$P_n(u) = \overset{0}{\Delta} y_0 + \frac{u}{1!}\overset{1}{\Delta} y_0 + \frac{u(u-1)}{2!}\overset{2}{\Delta} y_0 + \cdots + \frac{u(u-1)\cdots(u-(n-1))}{n!}\overset{n}{\Delta} y_0$$

où u est la variable réduite : $u = \frac{x-x_0}{h}$

et les $\overset{i}{\Delta} y_0$ sont les différences finies construites avec les relations :

$$\overset{0}{\Delta} y_k = y_k \text{ et } \overset{i}{\Delta} y_k = \overset{i-1}{\Delta} y_{k+1} - \overset{i-1}{\Delta} y_k$$

[1] On se reportera avec profit au cours du mercredi 7 avril dernier.

2.iii. Racine d'une fonction : Méthode de Newton.

Le développement en série limitée (dl) de Taylor d'une fonction *f(x)* s'écrit :

$$f(x)_{x_0} = f(x_0) + (x - x_0)f'(x_0) + \frac{(x - x_0)^2}{2!}f''(x_0) + \dots$$

Si calcule un DL en x_n, au point x_n+h tel que x_n+h = α une racine de f(x) et en tronquant ce DL à l'ordre 1 :

$$f(x_n + h) = f(x_n) + ((x_n + h) - x_n)f'(x_n) + O((x_n + h) - x_n)^2 = f(x_n) + hf'(x_n) + O(h)^2$$

Or α est racine de f(x), donc f(α) = f(x_n+h) = 0. Il vient en négligeant $O(h)^2$: $h \approx -\frac{f(x_n)}{f'(x_n)}$

Ou, en intégrant l'erreur commise dans h, que l'on note alors $\widehat{h}$, il vient : $\widehat{h} = -\frac{f(x_n)}{f'(x_n)}$. On calcule alors non pas la valeur d'α, mais une nouvelle valeur de x_n que l'on notre x_{n+1} : $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$ qui est une meilleure approximation d'α.

2.iv. Méthode de Leverrier-Newton

Cette méthode permet de calculer l'expression de la forme canonique d'un polynôme à partir de la connaissance de sa forme factorisée. Un même polynôme peut en effet s'écrire :

$$P(x) = a_0x^n + a_1x^{n-1} + \dots + a_{n-1}x + a_n = (x - x_0)(x - x_1)\cdots(x - x_{n-1}) = \prod_{k=0}^{n-1}(x - x_k)$$

Notez bien qu'un polynôme de degré n, a n+1 coefficients et n racines numérotées, ici, de 0 à n-1.

La méthode consiste à calculer un vecteur contenant la somme des puissances des racines : $S_j = \sum_{k=0}^{n-1} x_k^j$

A partir de ce vecteur, on calcule les coefficients du polynôme sous sa forme canonique en appliquant l'algorithme suivant :

$$\begin{cases} a_0 = 1 \\ a_0S_1 + 1 * a_1 = 0 \\ a_0S_2 + a_1S_1 + 2a_2 = 0 \\ a_0S_3 + a_1S_2 + a_2S_1 + 3a_3 = 0 \\ \vdots \\ a_0S_n + a_1S_{n-1} + a_2S_{n-2} + \dots + a_{n-1}S_1 + na_n = 0 \end{cases}$$

On pose a_0=1 par convention. Le fait de choisir une autre valeur pour a_0 ne change pas la valeur des racines du polynôme, mais si l'on développe le produit des racines on aboutit à cette valeur.

Il suffit de calculer a_0, pour pouvoir calculer a_1, puis a_2 etc. La fonction peut ainsi s'écrire :

```
void Leverrier(double *X, int n, double *a)
{  int i,j;
   double *S,p; // p auxiliaire de calcul pour le calcul des puissances
   /* n est le nombre de racines  */
   S = (double *) malloc( (n+1)*sizeof(double) );
   // Attention il y a n éléments de S1 à Sn, donc n+1 éléments de 0 à n.

   /* Calcul des Sommes de puissances */
   for (i=0 ; i<n ; i++)         /* Il y n racines... */
   {  p = X[i];
      for (j=1 ; j<=n ; j++)  /* On ne calcule pas S[0], début à 1... */
      {  S[j] = S[j]+p;
         p    = p*X[i];     /* p = puissances successives X[i] */
      }
   }
   /* remontee de la solution */
   a[0]= 1;
```

```
    for (i=1 ; i<=n ; i++)     /* Pn(x) a n+1 coefs */
    {  a[i] = 0;
       for ( j=0 ; j<=i-1 ; j++ )
       {  a[i] = a[i] - a[j]*S[i-j];
       }
       a[i] = a[i]/i;
    }
    free(S);
}
```

Notez dans ce code l'usage fait de **`malloc`**, et surtout l'utilisation d'un variable p pour calculer les puissances successives. Noter de même l'utilisation de **`S[i-j]`** pour croiser les indices. En effet dans l'expression permettant de calculer le terme a_k :

$$a_k = \frac{-1}{n}\left(a_0 S_k + a_1 S_{k-1} + a_2 S_{k-2} + \cdots + a_{k-1} S_1\right)$$

On remarque que les indices des termes de la somme sont croisés, ainsi, le terme *j* de la somme est le produit de $a_j * S_{k-j}$

Vous recopierez cette fonction telle quelle dans votre programme pour déterminer le polynôme $\Phi(x)$

3. Travail à effectuer

➡ **Remarque générale** : toutes les variables dimensionnées seront crées à l'aide de pointeurs et en utilisant la gestion dynamique de la mémoire. Le fichier de données contient 10 points.

➡ **Attention ! :** La méthode de Leverrier amène à calculer la somme des abscisses jusqu'à la puissance 10 (il y a 10 points). Par conséquent, il n'est pas possible d'utiliser des variables en simple précision (float).
C'est pourquoi toutes les fonctions utiliseront des variables de type double précision.
Dans ce cas, le type de champs à utiliser dans les saisies (scanf, fscanf) et les sorties (printf, fprintf) est %lf (la lettre 'L' et non le chiffre 1) et non %f.

3.i. Lecture du fichier

Avant toute choses, il faut bien entendu lire les points contenus dans un fichier appelé : projet_L2.txt, pour ce faire :

★ Ecrivez une fonction qui lise le contenu du fichier projet_L2.txt. L'interface est laissée libre.

3.ii. Calcul du polynôme d'interpolation : méthode de Newton

Ensuite, il faut être capable d'évaluer le polynôme d'interpolation.

★ Ecrivez une fonction qui calcule la valeur en un point quelconque la valeur prise par le polynôme d'interpolation calculé par la méthode de Newton. L'interface de cette fonction sera :

```
double newton_poly(double *Delta, int n, double u);
/* *Delta pointe vers une variable dimensionnée contenant les différences finies
 * en y0, n est le nombre de différences, et u le point de calcul exprimé en variable
 * réduite.
 */
```

✦ Testez votre fonction avec les différences pour Y_0 : $\overset{0}{\Delta} Y_0 = 1, \overset{1}{\Delta} Y_0 = 3, \overset{2}{\Delta} Y_0 = 14, \overset{3}{\Delta} Y_0 = 36, \overset{4}{\Delta} Y_0 = 24$
vous devez trouver : P(3) = 88 et P(3,5) ≈ 158,06

★ Ecrivez une fonction qui calcule la valeur de la variable réduite :

```
double varred(double xo, double h, double x)
/* Calcule la valeur de la variable réduite u au point x. x0 et h ont leur sens
 * habituels
 */
```

✦ Testez votre fonction.

★ Ecrivez une fonction qui calcule toutes les différences finies au point Y_0 :

```
void diffinies( double *y, int n, double *d);
/* *y pointe sur une variable dimensionnée contenant les Yi
 * n est le nombre de points du vecteur pointé par *y, et *d pointe
 * vers une variable contenant en sortie de fonction les différences finies en y0.
 */
```

- ✦ Testez votre fonction avec le vecteur Y suivant : Y = {1, 4, 21, 88, 265} vous devez trouver les différences finies en y_0 données deux point au dessus.

★ A l'aide de ces trois fonctions, calculez la valeur prise par le polynôme d'interpolation lorsqu'on interpole les points suivants :

X	-2	-1	0	1	2
Y	-2	3	2	1	6

- ✦ Vous devez trouver P(1.5) = 2,375.

3.iii. Méthode Dichotomie.

Pour cette méthode nous allons écrire une fonction dédiée. Cette fonction aura comme paramètres d'entrée les pivots (points lus) et leur nombre, le point de départ de la méthode, et la largeur de la fenêtre au départ.

★ Ecrivez une fonction appelée dichotomie qui mette en œuvre l'algorithme proposé au point 1.i. Le prototype de la fonction sera :

```
double dichotomie(  double *px,  double *py, int n,  double alpha,  double x );
/* px et py pointent vers les points, n est le nombre de points utilisés pour
 * l'interpolation. Alpha est le paramètre décrit au § 1.1, la largeur de la
 * fenêtre. x est le point de départ de l'algorithme.
 */
```

➡ **Remarque 1 :** Cette fonction dichotomie lancera le calcul des différences finies afin de déterminer le polynôme d'interpolation qui sera utilisé pour déterminer les valeurs de la fonction F(x).

➡ **Remarque 2 :** Il n'est pas utile d'interpoler sur les 10 points du fichier (cf la figure 2) On transmettra au pointeurs px et py les adresses des premiers points de la zone d'interpolation. De même la valeur transmise pour n sera le nombre de points utilisés pour l'interpolation. Afin de permettre à l'utilisateur de facilement choisir ces valeurs, les points lus seront affichés dans la fonction **`main()`**.

➡ **Remarque 3 :** La précision des calculs sera de **`1e-5`**, elle sera codée directement dans le programme.

★ Testez cette fonction avec plusieurs valeurs d'α { 2, 1, 0.5…}, la largeur de l'intervalle et en prenant comme point de départ le deuxième point lu. Affichez à l'écran les valeurs successives trouvées pour x et les valeurs d'α dans la boucle de calcul. Que constatez-vous ?

Afin de corriger ce problème l'algorithme est complété comme suivant :

```
x2=x1;
x1=x0;
it++;
if (it>1)
{  if (x0 == x2)
   {  alpha = alpha/2;
      it = 0;
   }
}
```

★ Expliquez la signification de chaque variable, sachant qu'**`x0`** est la valeur calculée dans la boucle en cours. Expliquez le rôle de la variable **`it`**, et des variables **`x1`** et **`x2`** au cours des boucles de la méthode mise en œuvre.

Exemple d'exécution du programme :

```
PhiL:Projet phil$ ./a.out

*** Points lus ***

0   2.000000   12.416590
1   3.000000   12.846458
2   4.000000   13.992534
3   5.000000   15.515667
4   6.000000   16.593288
5   7.000000   17.432881
6   8.000000   17.338295
7   9.000000   15.952573
8   10.000000      14.150546
9   11.000000      12.754494

Entrez l'indice du point de départ : 3
Entrez la valeur de départ d'Alpha : 1
```

```
Nombre de pts pour l'interpolation : 5
Valeur trouvée : 7.431915
```

★ Faites un bilan de cette méthode. Est-elle pleinement satisfaisante ?

3.iv. Méthode racine de la dérivée.

★ Pour cette méthode il faut déterminer l'expression du polynôme d'interpolation (exercice demandé à préparer en TD). Pour cette méthode, il faut tout d'abord déterminer à l'aide de la fonction Leverrier (cf 2.iv) l'expression de $\Phi(x)$. On utilisera **npi** pivots pour déterminer un polynôme autour d'une zone et non sur l'ensemble des pivots (comme pour la méthode de Dichotomie).

✦ Insérez la fonction Leverrier permettant de déterminer les coefficients de Φ(x) dans votre programme :

```
void Leverrier(double *X, int n, double *a)
/* X est un pointeur vers les racines de la forme factorisée du polynôme Φ(x)
 * n est le nombre de racines (ici ce sera 10...) et a un pointeur vers une variable
 * qui contiendra les coefficients du polynôme Φ(x). Attention, s'il y a n racines
 * alors Φ(x) est de degré n, il a donc n+1 coefficients !!
 */
```

✦ Testez la fonction avec les points suivants : $x_0=1$, $x_1=2$, $x_2=3$, Vous devez obtenir le polynôme de degré 3 (avec 4 coefficients donc) : $P(x) = x^3 - 6x^2 + 11x - 6$.

★ Ecrivez une fonction permettant le calcul du reste et des coefficients du polynôme quotient de P(x)/(x-α). Le prototype de cette fonction sera le suivant :

```
double horner(  double *a,  double *b, int n,  double alfa );
/* a : coefs du polynôme, b : coefs du polynome quotient, n : degre de a
 * alpha, point de calcul
 */
```

★ Ecrivez une fonction servant à déterminer l'expression littérale du polynôme d'interpolation. Cette fonction utilisera la fonction horner pour calculer chaque coefficient de Lagrange.

```
void detpoly( double *X,  double *Y, int n,  double *a)
/* X est un pointeur vers les abscisses et Y vers les ordonnées des points
 * pour lesquels on cherche à déterminer l'expression du polynôme d'interpolation
 * n est le nombre de points, et a un pointeur vers une zone mémoire suffisante pour
 * contenir les coefficients d'un polynôme de degré n-1.
 */
```

★ Dans la fonction principale, calculez la valeur des coefficients du polynôme dérivé. On rappelle que l'on a $(a_k x^n)' = n * a_k x^{n-1}$.

★ Ecrivez une fonction qui, en utilisant la méthode de Newton et la méthode d'Horner, détermine une racine d'un polynôme à partir d'un point de départ **x0**. Le prototype de cette fonction sera :

```
double Newton( double *a,  int n, double x0 );
/* a = coefs du polynôme, n degre du polynôme, x0 : point de départ de la méthode
 * La précision des calculs sera de 1e-6 en précision relative et non absolue.
 */
```

★ En utilisant le polynôme dérivé, et la fonction Newton, déterminez une valeur approché de l'abscisse du sommet.

➡ Exemple de calcul :

```
Entrez l'indice du point de départ : 3
Nombre de points pour l'interpolation : 5
*** Points Interpolés ****
0  Py = 12.416590   22.904047
1  Py = 12.846458   17.126913
2  Py = 13.992534   15.235363
3  Py = 15.515667   15.515668
4  Py = 16.593288   16.593288
5  Py = 17.432881   17.432881
6  Py = 17.338295   17.338295
7  Py = 15.952573   15.952573
8  Py = 14.150546   13.257953
9  Py = 12.754494   9.575864
```

```
Entrez le point de départ : 7

*** Polynôme derivé ****
x^4   0.014133 *4 -> 0.056532
x^3   -0.483485 *3 -> -1.450455
x^2   5.516838 *2 -> 11.033675
x^1   -25.093760 *1 -> -25.093760
x^0   54.665967 *0 -> 0.000000
Valeur Interpolation : 7.431922
```

3.v. Bilan

Quelle est d'après-vous la méthode la plus efficace ? Pouvait-on envisager d'autres méthodes ?