Prép'ISIMA

Programmation Linéaire

Interpolation polynomiale

Avec les méthodes de Newton et Lagrange

Clément Payard
Mathieu Laurençot

Encadrant : M. Suzanne ÉLODIE

2021

Table des matières

1 Rappel rapide des méthodes

1.1 Cas général des méthodes

Pour les deux méthodes, le but est le même, trouver une équation avec un nombre de points $n+1$ de la forme :

$$P(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$$

1.2 Méthode de Lagrange

La méthode de Lagrange se base sur la formule suivante :

$$L(x) = \sum_{i=0}^{n} y_i \left(\prod_{i=0, i \neq j}^{n} \frac{x - x_j}{x_i - x_j} \right)$$

Que l'on peut aussi écrire sous cette forme :

$$L(X) = \sum_{i=0}^{n} y_i l_i(X)$$

Avec li définie tel que :

$$li = \prod_{i=0, i \neq j}^{n} \frac{x - x_j}{x_i - x_j}$$

1.3 Méthode de Newton

Pour la méthode de newton, nous devons appliquer cette formule :

$$N(x) = \sum_{i=0}^{k} a_i n_i(x)$$

Où n sont les polynômes de Newton définis de la manière suivante :

$$n_i(x) = \prod_{0 \leq j < i} (x - x_j) \qquad i = 0, \ldots, n$$

et a_i les différences divisées définit tel que

$$\begin{aligned} a_0 &= y_1 \\ a_1 &= \frac{y_2 - y_1}{x_2 - x_1} \\ &\ldots \\ a_k &= \nabla^k y_{k+1} \\ &\ldots \\ a_{n-1} &= \nabla^{n-1} y_n \end{aligned}$$

$$\text{Avec pour } j > i \text{ ,} \nabla^k y_j = \frac{\nabla^{k-1} y_j - \nabla^{k-1} y_k}{x_j - x_k}$$

Conclusion : Nous devons effectuer cette formule :

$$N(x) = [y_0] + [y_0, y_1](x - x_0) + \ldots + [y_0, \ldots, y_n](x - x_0) \ldots (x - x_n).$$

2 Présentation des programmes commentés

Nous ne mettrons pas l'intégralité du code des différentes résolutions, mais ils sont consultables ici pour la méthode de Lagrange, ainsi qu'ici pour la méthode Newton.

2.1 Nos structures

Nous avons choisis d'utiliser plusieurs structures pour plusieurs cas :

1. La première, la structure *point*, qui est utilisé partout. En effet, cette structure nous permet de stocker les différents points qui nous sont nécessaire pour trouver un polynôme respectant l'interpolation.
2. Puis, la structure *Maillon*. Cette structure est essentielle pour une chose : former des listes, ce qui nous permet d'ajouter un nombre de points indéfinis au départ, puis d'en rajouter, supprimer, etc, sans perdre de la mémoire.
3. La structure *Liste*, qui est la suite logique de structure *Maillon*.
4. La structure polynôme : celle-ci est particulière, une partie lui est entièrement consacré dans la suite du document.
5. Enfin, même si ce n'est pas vraiment une structure, une fonction s'appelant *transformefloatenpoly* transforme une liste de points en un tableau, permettant alors d'y accéder plus rapidement et facilement.

2.1.1 Listes/Points

Le code pour les listes ainsi que pour les points est trouvable ici.

Pour former un point, nous avons tout simplement besoin d'un x et d'un y associé à celui-ci. La structure des points est donc la suivante :

```
typedef struct point
{
  float x;
  float y;
} point;
```

Puis, nous voulons rassembler ses points en des listes, nous avons donc besoin de deux structures supplémentaires : la structure *Maillon* et la struture *Liste*. Ces deux structures permettent de structurer nos données, les utiliser, et les afficher dans des graphiques.

```
typedef struct Maillon{
  struct Maillon *suiv;
  point val;
}Maillon;

typedef struct Liste{
  struct Maillon *first;
}Liste;
```

De plus, les fonctions usuelles des listes sont également créé :

1. *creerListe*
2. *detruireListe*
3. ajouteDebut
4. ajouteFin
5. *afficheListePoints*
6. ListLenght
7. supprDebut

8. supprFin
9. supprValeur
10. supprMaillon
11. ListeToTabsPoints, qui transforme une liste en un tableau.
12. ViderListe

2.1.2 Polynômes

Le code pour les polynôme est trouvable ici.

Comme vous le savez, le C n'inclue pas de type "polynôme". Nous avons donc dû creer la structure suivante :

```
typedef struct Polynome{
  float *p;
  int maxDeg;
}polynome;
```

Cette structure à pour but de prendre un tableau, où l'indice du tableau nous permet de trouver le degrès de x, et les valeurs stockés dans le tableau sont les différents coefficients pour chaque x du polynôme.

Nous avons également fait les diverses fonctions de bases :

1. *polynome *creePolynome(int maxDeg)*, qui nous permet de créer un pointeur sur un polynôme.
2. void destroyPolynome(polynome *p), qui supprime un polynôme.
3. void affichepolynome(polynome *p), comme son nom l'indique, affiche le polynôme passé en paramètre.
4. polynome *transformefloatenpoly(float unfloat), qui convertie un flottant en un polynôme.
5. polynome *addPolynome(polynome *p1, polynome *p2), qui nous permet d'additionner (et dans le même temps de soustraire) deux polynôme entre eux

```
int i;
polynome *poly =
  creePolynome((p1->maxDeg > p2->maxDeg) ? p1->maxDeg : p2->maxDeg);

for (i = 0; i < p1->maxDeg + 1; ++i)
  {
    /* printf("Degrèse de p1 %d \n", p1->maxDeg); */
    poly->p[i] = p1->p[i];
  }

for (i = 0; i < p2->maxDeg + 1; ++i)
  {
    poly->p[i] += p2->p[i];
  }
destroyPolynome(p1);
destroyPolynome(p2);
return poly;
```

6. polynome *multPolynome(polynome *p1, polynome *p2). Cette fonction multiplie deux polynômes entre eux.

```
int i, j;
polynome *poly = creePolynome(p1->maxDeg + p2->maxDeg);

for (i = 0; i < p1->maxDeg + 1; ++i)
  {
    for (j = 0; j < p2->maxDeg + 1; ++j)
```

```
        {
    poly->p[i + j] += p1->p[i] * p2->p[j];
        }
      }
    destroyPolynome(p1);
    destroyPolynome(p2);
    return poly;
```

2.2 Présentation de la méthode de Lagrange

Pour aborder ce problème, nous avons décidé d'aborder ce problème en deux étapes :

1. D'abord, calculer Li, avec une fonction *calculLi*
2. Puis, grâce à la fonction *calculLi* que l'on appelle dans la fonction *calculLagrange*, on renvoie le polynôme correspondant à l'interpolation.

2.2.1 Présentation de la fonction *calculLi*

Suite à l'initialisation des variables nécessaires, cette fonction permet le calcul de Li, avec $i \in [0, 1, ..., k]$. Les deux principales difficultés sont les suivantes :

1. Prendre en compte le cas de la division par 0, lorsque $x_i - x_j = 0$.
2. Réinitialiser x à chaque tour de boucle. En effet, nos fonction renvoie un nouveau pointeur, ce qui supprime x à chaque fois. Nous devons donc le réinitialiser à chaque tour dans la boucle, ce qui nous permet d'enlever la fameuse erreur "segmentation fault" !

```
for (i = 0; i < ListLenght(points); ++i)
{
  if (i == numero)
  {
    /* si i = numero, alors il y aurait une division par 0. On ne
       fait donc rien */
  }
  else
  {
    /* Création du polynôme pour les calculs */
    polynome *x = creePolynome(1);
    x->p[1] = 1;

    /* Calcul de la première différence */
    polynome *y = addPolynome(x, transformefloatenpoly(-pointstab[0][i]));

    /* calcul de la multiplication par l'inverse de xnum - xi */
    polynome *tmp = multPolynome(
y,
transformefloatenpoly(1 / (pointstab[0][numero] - pointstab[0][i])));

    /* multiplication par le polynôme précédemment calculé */
    Li = multPolynome(Li, tmp);
  }
}

return Li;
```

2.2.2 Présentation de la fonction *calculLagrange*

Cette fonction permet de faire la somme des différents Li trouvé dans la fonction *calculLi*. En effet, pour compléter la méthode de Lagrange, il faut faire une boucle *for* qui fait une somme des y_i multiplier par *calculLi* pour l'itération i.

```
  int i;

/* création du polynôme du résultatde l'interpolation  */
polynome *fonction = creePolynome(ListLenght(points));

/* Initialisation d'un tableau à partir d'une liste (accéder au
   valeur plus facilement) */
float **pointstab;
pointstab = ListeToTabsPoints(points);

/* boucle for pour multiplier par yi les Li calculer dans la
   fonction calculLi */
for (i = 0; i < ListLenght(points); ++i)
{
  fonction = addPolynome(fonction,
 multPolynome(transformefloatenpoly(pointstab[1][i]),
      calculLi(i, points)));
}

return fonction;
```

Il ne reste plus qu'à retourner le polynôme trouver pour pouvoir l'afficher ou bien même l'utiliser dans SDL.

2.3 Présentation de la méthode de Newton

Le code pour l'interpolation newtonienne est disponible ici.

Pour la résolution par Newton, nous commençons tout d'abord par initialiser un tableau de la taille adéquate pour nous permettre de stocker les différences divisées.

```
long double **triangle = (long double **)malloc(sizeof(long double *) * (pointNB));
for(int i = 0; i < pointNB; i++){
  triangle[i] = (long double *)malloc(sizeof(long double) * (pointNB-i));
}
```

Puis, nous remplissons la première colonne avec les ordonnées des points.

```
for(int i = 0; i < pointNB; i++)
  {
    triangle[0][i] = Points[1][i];
  }
```

Subséquemment, nous pouvons effectuer le calcul des différences divisées en appliquant la formule du cours

```
for(int i = 1; i < pointNB; i++){
  for(int j = 0; j < pointNB-i; j++){
    triangle[i][j] = (triangle[i-1][j+1] - triangle[i-1][0]) / (Points[0][i+j] - Poin
  }
 }
```

Enfin, nous pouvons passer à la résolution résolution finale, qui elle aussi est issue du cours.

```
for(int i = 0; i < pointNB-1; i++){
  /* créer le polynome "constant" */
  tmp = creePolynome(2);
  tmp->p[1] = 1;

  /* résolution de la méthode */
  tmp->p[0] = -Points[0][pointNB - 2 - i];
  Solution = multPolynome(Solution, tmp);
  Solution->p[0] += triangle[pointNB-i-2][0];
}
Solution = AdaptePoly(Solution);
```

Il ne nous reste plus qu'à retourner le polynôme obtenu pour pouvoir l'afficher dans le terminal, où sous une courbe avec SDL.

3 Présentation des Jeux d'essais avec commentaires

Les différents résultats sont disponibles en faisant les test grâces au fonctionnalité implémenter dans le programme.

3.1 Densité de l'eau en fonction de la température de l'air :

Liste de points avec une précision de l'ordre de 10^{-5}. Difficultés potentielles : précisions.

Résultats : Les deux polynômes sont les mêmes et les calculs sont les bons.

3.2 Les trois séries :

Suites de points normaux. Difficultés potentielles : aucune, sauf pour le dernier qui ne pourra sûrement pas être calculé (car ce n'est pas une suite de points avec différents x)

Résultats :

1. Le polynôme est le bon, aucune difficulté
2. Le polynôme est le bon, mais des erreurs d'approximations sont présentes. Les polynômes ne sont pas exactement les mêmes.
3. Ne peut pas se calculer : en effet, les points sont sur le même X. L'interpolation est donc incalculable (et infaisable).

3.3 Dépenses mensuelles et revenus :

Pour ces données, il y a un grand nombre de données, ainsi qu'un axe des x qui commence avec de "grandes valeurs".

Difficultés potentielles : précisions dû aux résultats importants obtenus.

Résultats :Les polynômes ont soit des coefficients très importants, soit des coefficients presque négligeable. En revanche, on peut constater qu'il y a des approximations de calcul dans les deux méthodes. En effet, les deux polynômes finaux ne sont pas exactement les mêmes, même si ils sont tous les deux du même ordre de grandeur.

3.4 Commentaire global

Pour ses différents jeux d'essais, on peut constater plusieurs choses :

1. L'unicité du polynôme obtenu
   En effet, ce dernier est souvent le même pour les deux méthodes. Évidemment, des approximations de calcul sont présentes.
2. Influence de la modification d'un ou plusieurs points donnés sur le polynôme : Lorsque l'on donne des points aléatoires au fur et à mesure grâce à l'implémentation de notre méthode "placer un point" pour mettre un point, on peut voir que les deux polynômes s'adaptent bien.

3. Évaluation des coûts : Pour Newton, la complexité est de l'ordre $(o)n^2$, tandis que pour Lagrange, elle est de $(o)n^3$. Vous pouvez, grâce au programme, afficher les polynômes avec une grande ou une petite précision. Si vous choisissez la méthode précise, vous aurez alors la moyenne des différences avec les points originaux, ce qui nous permet d'avoir une idée précise sur l'efficacité des méthodes.

4 SDL

Le code qui nous permet de gérer la fenêtre SDL est disponible ici.

4.1 Début

Pour démarrer SDL, nous devons initialiser de nombreuses variables, comme par exemple :

— La variable *Stape*, qui nous permet de fermer SDL si elle est égale à 0,
— *size*, qui va nous permettre de gérer la taille de l'écran,
— des variables permettant de garder un nombre d'image par seconde (fps) constant et agréable
— des variables permettant de détecter où le curseur de la souris se trouve sur l'écran
— les varibles permettant de dessiner le graphique
— etc.

4.2 Affichage et fonctionnalité

4.2.1 Affichage

Pour effectuer l'affichage d'une fenêtre SDL, nous devons passer par une boucle *while*.

Puis, nous distinguerons trois cas grâce à un *if* (et *else if*).

1. Dans le premier cas, SDL dessinera l'écran, s'il n'a pas été dessiné depuis un certain temps
2. Sinon, nous vérifierons également si les courbes sont en adéquation avec les polynômes. Si ce n'est pas le cas, nous entrons alors dans le *else if* qui va nous permettre d'écraser l'image précédente. Enfin,
3. si nous passons les deux conditions précédentes, nous devons **absolument** endormir le Central Processing Unit (CPU). Cela nous permet de ne pas utiliser tout le processeur de l'ordinateur.

Puis, nous avons aussi un cas de débogage. En effet, si l'on n'est pas entré dans le while depuis une seconde ou plus, il peut y avoir un problème. On recommence alors une seconde "propre", en mettant certaines variables à 0.

4.2.2 Fonctionnalités :

Divers fonctionnalités sont présentes : Vous pouvez afficher la fenêtre grâce à la touche "g". Vous pouvez désormais voir la liste des points, les courbes représentant l'interpolation lagrangienne et newtonienne, ainsi que la liste de points à droite.

De plus, si jamais vous voulez rajouter des points à la liste, cette fonctionnalité est disponible grâce au bouton gauche de la souris. Le bouton droit aura pour effet de supprimer le point sélectionné.

Le curseur aura alors une position (x et y) qui sera automatiquement ajouté dans la liste des points. Les courbes ainsi que les polynômes vont s'adapter automatiquement !

Enfin, vous pouvez zoomer et dézoomer grâce à la molette de la souris.

4.3 Fin de SDL

La fonction *end-sdl* nous permet de fermet la fenêtre SDL proprement, ainsi que faire les opérations nécessaires pour vider la mémoire qui a besoin d'être libéré.

5 Conclusion sur les méthodes

Nous pourrions sûrement améliorer notre programme pour Newton : en effet, d'après la définition des différences divisées, lorsque l'on ajoute de nouveaux points, nous ne sommes pas obligés de recalculer l'ensemble des coefficients du polynôme.

En revanche, malgré la différence de programme et de méthode utilisé, les méthodes nous amènent au même résultat (hors approximation des calculs). En effet, les polynômes, grâce aux jeux de données, sont trés proches et passent par les différents points donnés (voir fenêtre graphique SDL).