



PROGRAMMATION LINÉAIRE

Interpolation polynomiale

Avec les méthodes de Newton et Lagrange



Clément PAYARD
Mathieu LAURENÇOT

Encadrant : M. SUZANNE ÉLODIE

Table des matières

1	Rappel rapide des méthodes	1
1.1	Cas général des méthodes	1
1.2	Méthode de Lagrange	1
1.3	Méthode de Newton	2
2	Présentation des programmes commentés	2
2.1	Nos structures	2
2.2	Présentation de la méthode de Lagrange	4
2.3	Présentation de la méthode de Newton	5
3	Présentation des Jeux d'essais (avec graphique)	5
4	Commentaires des jeux d'essais	5
5	Conclusion sur les méthodes	5

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 $k + 1$ de la forme :

$$P(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$$

Où l'on a $n \in [0, 1, \dots, k]$.

1.2 Méthode de Lagrange

La méthode de Lagrange se base sur la formule suivante :

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

Que l'on peut aussi écrire sous cette forme :

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

Avec l_i définie tel que :

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

1.3 Méthode de Newton

https://fr.wikipedia.org/wiki/Interpolation_newtonienne#Remarque

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

Avec les polynômes de Newton définis de la manière suivante

$$n_i(x) = \prod_{0 \leq j < i} (x - x_j) \quad j = 0, \dots, k$$

Conclusion :

$$N(x) = [y_0] + [y_0, y_1](x - x_0) + \dots + [y_0, \dots, y_k](x - x_0) \dots (x - x_{k-1}).$$

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 TODO, nous avons donc besoin de deux structures supplémentaires : la structure *Maillon* et la structure *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. *destruireListe*
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 créer 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 degrés 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 L_i , 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 L_i , avec $i \in [0, 1, \dots, k]$. 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*.

```

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;

```

2.3 Présentation de la méthode de Newton

3 Présentation des Jeux d'essais (avec graphique)

4 Commentaires des jeux d'essais

1. unicité du polynôme obtenu
2. adéquation du polynôme aux jeux d'essais
3. influence de la modification d'un ou plusieurs points donnés sur le poly
4. évaluation des coûts (complexité, efficacité)

5 Conclusion sur les méthodes

<https://math-linux.com/mathematiques/interpolation/article/interpolation-polynomia>

Malgré la différence de programme et de méthode utilisé, les méthodes nous amènent au même résultat. En effet,

Il existe une fonction sur scilab mdr Scilab : calcul de polynome d'interpolation de Newton

La fonction Scilab `newton.sci` permet de déterminer le polynome d'interpolation de Newton. X contient les points d'interpolations et Y les valeurs d'interpolation, P

est le polynome d'interpolation de Newton calculé à l'aide des différences divisées.