Sorbonne Université — **M1 Mathématiques**

2022–2023 — **MU4M056 : Programmation en C++**

TP noté 1 : Polynômes d'interpolation de Lagrange

L'objectif de ce TP est d'implémenter en C++ les polynômes d'interpolation de Lagrange sous forme de classe, et d'étudier leur application dans l'approximation de fonctions. On veillera à inclure dans chaque fichier toutes les bibliothèques nécessaires et les options de compilation nécessaires. **Il est également impératif de mettre en commentaire, dans tous les fichiers, les nom, prénom et n° d'étudiant.**

1 Définition des polynômes de Lagrange

1.1 Mathématiques

Soit $u = ((x_0, y_0), \dots, (x_{N-1}, y_{N-1}))$ un N-uplet de couples de nombres réels tels que les $(x_i)_{0\leq i<N}$ soient deux à deux distincts. On notera $x = (x_0, \dots, x_{N-1})$ le N-uplet des abscisses (x_i). On définit pour tout $0 \leq j \leq N$ le polynôme

$$R_j^{(x)}(X) = \prod_{\substack{i=0\\ i\neq j}}^{N-1} \frac{X - x_i}{x_j - x_i} = \frac{(X-x_0)(X-x_1)...(X-x_{j-1})(X-x_{j+1})...(X-x_N)}{(x_j-x_0)(x_j-x_1)...(x_j-x_{j-1})(x_j-x_{j+1})...(x_j-x_N)}. \quad (1)$$

On peut montrer que $R_j^{(x)}$ est de degré N pour tout j, et que $R_j^{(x)}(x_i) = \delta_{ij}$ pour tout $0 \leq i, j < N$. Par conséquent, le polynôme

$$P^{(u)}(X) = \sum_{j=0}^{N-1} y_j R_j^{(x)}(X) \quad (2)$$

est un polynôme de degré au plus $N-1$ qui vérifie $P^{(u)}(x_i) = y_i$ pour tout i, et on peut vérifier que c'est le seul qui vérifie ces deux propriétés. Ce polynôme permet d'avoir une interpolation polynomiale d'une fonction $f : \mathbb{R} \to \mathbb{R}$ suffisamment régulière[1] sur l'intervalle $[\min x_i, \max x_i]$ en posant $y_i = f(x_i)$ pour tout $0 \leq i \leq N$.

1.2 Implémentation

Remarque : dans le texte ci-dessous, le symbole **(T)** *indique un fichier à télécharger sur Moodle avant le début du partiel.*

Nous allons implémenter les polynômes d'interpolation de Lagrange de la façon suivante :

```cpp
//fichier interpolation.hpp
class LagrangeInterpolation{
  private:
        std::vector< double > x_values;//contient les x_i
        std::vector< double > y_values;//contient les y_i
  public:
        LagrangeInterpolation(double, double, const std::vector<double> &);
        ... nb_points() ...;
        ... operator()(double) ...;//évaluation du polynome en un point
        ... lower_bound() ...;
        ... upper_bound() ...;
        ... add_point(double, double) ...;
};
```

1. La régularité de f contrôle la précision de l'approximation de l'interpolation.

— `LagrangeInterpolation(u, delta, y)` est un constructeur qui construit le polynôme dont les points $(y_i)_{0\leq i<N}$ sont lus dans le vecteur `y` et les abscisses x_i sont données par la formule $x_i = u + i\delta$ pour $0 \leq i < N$.
— `nb_points()` donne le nombre de points d'interpolation.
— `add_point(x,y)` est une méthode qui permet d'ajouter un couple (x, y) aux points d'interpolation ;
— `operator()` est un opérateur qui calcule une évaluation du polynôme d'interpolation de Lagrange $P^{(u)}(z)$ au point z donné en argument selon les formules (1) et (2).
— `lower_bound()` et `upper_bound()` donnent respectivement $\min x_i$ et $\max x_i$.

2 Fonctionnalités élémentaires

Voici un programme minimal qui doit fonctionner.

```cpp
//fichier test1.cpp
using namespace std;
int main() {
   const LagrangeInterpolation P(0., 0.25,{1., 3., 2., 4.});
   cout << "Voici le polynome lu dans points.dat:\n" << P << "-----\n";
   cout << "Le polynome P contient " << P.nb_points() << " couples \n";
   cout << "L'abscisse minimale est : " << P.lower_bound() << "\n";//0.
   cout << "L'abscisse maximale est : " << P.upper_bound() << "\n";//0.75
   cout << "L'évaluation en z=0.75 donne : " << P(0.75) << "\n";//4.
   cout << "L'évaluation en z=0.4 donne : " << P(0.4) << "\n";//2.376
   cout << "L'évaluation en z=0.1 donne : " << P(0.1) << "\n";//2.544
   LagrangeInterpolation Q(0., 0.25,{1., 3., 2., 4.});
   cout << "Ajout de (0.5,7) à Q ? " << Q.add_point(0.5,7.) << "\n";//0
        //message d'erreur et n'ajoute pas de point
   cout << "Ajout de (0.4,1.2) à Q? " << Q.add_point(0.4,1.2) << "\n";//1
   cout << "Nouvel objet Q: "<< Q;
   cout << "L'évaluation de Q en z=0.75 donne: " << Q(0.75) << "\n";//4
   cout << "L'évaluation de Q en z=0.4 donne: " << Q(0.4) << "\n";//1.2
   cout << "L'évaluation de Q en z=0.1 donne: " << Q(0.1) << "\n";//4.728
   return 0;
}
```

Vous mettrez toutes les lignes en commentaires et les décommenterez au fur et à mesure de l'avancée des questions pour les tester.

1. Dans le fichier `"interpolation.hpp"` **(T)**, compléter le code en ajoutant les lignes nécessaires pour :
— s'assurer qu'il n'y ait pas de problème de redéfinition en cas d'inclusions multiples
— ajouter les prototypes du constructeur et des 5 méthodes sus-mentionnées (ne pas oublier les `const` nécessaires)
— inclure les bibliothèques nécessaires
— ajouter la déclaration des opérateurs `<<` et `>>`
Compléter les fichiers `test1.cpp` **(T)** et `interpolation.cpp` **(T)** avec les en-têtes nécessaires.

2. Écrire le code du constructeur de la classe.

3. Écrire le code de l'opérateur d'écriture `<<`. On souhaite le format suivant : sur la première ligne apparaît le nombre de points, puis sur chaque ligne apparaissent les nombres x_i et y_i séparées par une espace. Le polynôme `P` de `test1.cpp` doit alors apparaître comme :

```
4
0. 1.
0.25 3.
0.5 2.
0.75 4.
```

4. Écrire le code de l'accesseur `nb_points()` au nombre de points stockées dans le champ `points`.

5. Écrire les codes de `lower_bound()` et `upper_bound()`.
Attention : Vous vérifierez bien que toutes vos méthodes marches.

6. Écrire le code de `operator()` qui permet les évaluations des polynômes $P^{(u)}$ en un point arbitraire z.

7. Écrire le code de la méthode `add_point(x,y)` qui tente d'ajouter un nouveau couple (x, y) l'ensemble de points déjà présents :

— si `x` correspond à une valeur déjà présente, rien n'est ajouté, un message d'erreur (laissé à votre choix) est affiché et la méthode renvoie `false`

— sinon, le couple (x, y) est ajouté et la méthode renvoie `true` ;

Indication : on pourra utiliser `std::find` de `<algorithm>` et `std::distance` de ? ? ? qui fonctionnent de la manière suivante :

```cpp
auto it = std::find( u.begin(), u.end(), x);
//si x est dans u alors it est un itérateur vers la case où est x
int k = std::distance( u.begin(), it); // k est le numéro de la
// case indiquée par it, i.e. u[k] est la case indiquée par it.
```

8. Vérifier à présent que l'intégralité du code de `test1.cpp` compile et s'exécute correctement. Si ce n'est pas le cas, ce n'est pas la peine de passer à la suite.

3 Fonctionnalités plus avancés et programme plus complet

3.1 Lecture des données dans un fichier

Nous souhaitons à présent lire les données des couples (x_i, y_i) dans un fichier.

9. Faut-il écrire un constructeur par défaut ? Si oui, faites-le ; si non, expliquer pourquoi en commentaire dans la classe.

10. Écrire le code de l'opérateur de lecture `>>` qui lit un polynôme à partir du **même** format que celui imposé pour `<<`.

11. Écrire un programme `test2.cpp` **(T)** qui lit les points d'interpolation dans un fichier `points.dat` **(T)**, construit l'objet de type `LagrangeInterpolation` correspondant grâce à `>>`, l'affiche dans le terminal, donnent les deux abscisses extrêmales (vérifier vous-même) et l'évalue en $z = 0.5$ et $z = 1$. *Indication : on doit obtenir les valeurs* -1.23435 *et* 0.297951*.*

3.2 Approximation d'une fonction connue

Nous considérons approcher la fonction sinus sur $[0, \pi]$ par des polynômes de Lagrange évalués en des points uniformément répartis sur $[0, \pi]$ et nous nous intéressons à la qualité de l'évaluation. Pour cela, nous introduisons pour tout entier $N \geq 2$ la suite de points $(u^{(N)})_{0 \leq k < N}$ définie par :

$$x_k^{(N)} = \frac{k\pi}{N}$$

Nous introduisons également la suite $v_{0 \leq k < N}^{(N)}$ définie par

$$t_k^{(N)} = \frac{(2k+1)\pi}{2N}$$

et on remarquera l'entrelacement

$$x_0^{(N)} < t_0^{(N)} < x_1^{(N)} < t_1^{(N)} < \ldots < x_{N-1}^{(N)} < t_{N-1}^{(N)}$$

On considère le polynôme de Lagrange définie par la suite $u = (x_k^{(N)}, f(x_k^{(N)}))_{0 \leq k < N}$ et on évalue sa qualité par

$$\Delta = \max_{0 \leq k < N} |\sin(t_k^{(N)}) - P^{(u)}(t_k^{(N)})|$$

12. Dans un fichier `approx.cpp` **(T)**, définir deux fonctions

```
std::vector<double> generate_x(int N);
std::vector<double> generate_t(int N);
```

qui génèrent deux vectors qui contiennent les suites $x^{(N)}$ et $t^{(N)}$ et les stockent dans des vecteurs de taille N.

13. Dans une fonction `main()` , calculer Δ pour la fonction sinus en utilisant la classe `LagrangeInterpolation` pour $N \in \{4, 16, 64, 256, 512\}$. Indiquer en commentaire comment vous pensez que Δ diminue asymptotiquement avec N.

4 Addition

Étant donnés deux polynômes P et P' de Lagrange qui interpolent respectivement les points $(x_i, y_i)_{0 \leq i < N}$ et les points $(x'_i, y'_i)_{0 \leq i < N'}$ *tous distincts*, la somme des deux polynômes $P + P'$ est l'unique polynôme qui interpole les points $(x''_i, y''_i)_{0 \leq N + N'}$ donnés par :

$$(x''_i, y''_i) = \begin{cases} (x_i, y_i + P'(x_i)) & \text{pour } i < N \\ (x'_{i-N}, y'_{i_N} + P(x'_{i-N})) & \text{pour } N \leq i < N + N' \end{cases} \tag{3}$$

Vous êtes obligés d'utiliser `std::copy` et `std::transform` et aucune boucle `for` ne doit apparaître dans les questions ci-dessous.

14. Écrire l'opérateur `+` sur la classe `LagrangeInterpolation` qui code la construction de $P + P'$ à partir de la construction précédente.

15. Écrire un programme complet `test_add.cpp` **(T)** qui teste votre opérateur sur un exemple.

16. si deux points (x_i, y_i) et (x'_j, y'_j) satisfont $x_i = x'_j$, alors il ne faut garder que l'un des points et considérer l'ordonnée $y_i + y'_j$. Améliorer votre opérateur `+` pour gérer ce cas-là.