TP4 : La programmation orientée objet (2)

Objectifs et conseils

Qu'est ce que la POO ?

Qu'est ce qu'un objet ? Il s'agit d'un mélange de plusieurs variables et fonctions. Imaginez que vous avez créé un programme qui permet de résoudre des équations différentielles ordinaires : vous pouvez afficher vos solutions, calculer des ordres de convergence, comparer deux méthodes ... Le code est complexe : il aura besoin de plusieurs fonctions qui s'appellent entre elles, ainsi que de variables pour mémoriser la solution au cours du temps, la méthode utilisée, le pas de temps choisi ... Au final, votre code est composé de plusieurs fonctions et variables. Votre code sera difficilement accessible par quelqu'un qui n'est pas un expert du sujet : Quelle fonction il faut appeler en premier ? Quelles valeurs doit-on envoyer à quelle fonction pour afficher la solution ? etc ... Votre solution est de concevoir votre code de *manière orientée objet*. Ce qui signifie que vous placerez tout votre code dans une grande boîte. Cette boîte c'est ce qu'on appelle un **objet**. L'objet contient toutes les fonctions et variables mais elles sont masquées pour l'utilisateur. Seulement quelques outils sont proposés à l'utilisateur comme par exemple : définir mon pas de temps, mon intervalle de calcul et ma méthode, calculer l'ordre de la méthode, afficher la solution ...

En quelques lignes :

- La programmation orientée objet est une façon de concevoir son code dans laquelle on manipule des objets.
- Les objets peuvent être complexes mais leur utilisation est simplifiée. C'est un des avantages de la programmation orientée objet.
- Un objet est constitué d'attributs et de méthodes, c'est-à-dire de variables et de fonctions membres.
- On appelle les méthodes de ces objets pour les modifier ou obtenir des informations.

Qu'est ce qu'une classe ? Pour créer un objet, il faut d'abord créer une classe. Créer une classe consiste à définir les plans de l'objet. Une fois que la classe est faite (le plan), il est possible de créer autant d'objets du même type. Vocabulaire : on dit qu'un objet est une **instance** d'une classe.

Objectifs

- Surcharges d'opérateur et templates à travers l'implémentation d'une classe de vecteurs creux,
- Utiliser une librairie extérieure pour les vecteurs denses et creux et les matrices denses et creuses : `Eigen`.

Exercices

Exercice 1 - Une classe de vecteurs creux

1. On dit qu'un vecteur est creux quand la plupart de ses composantes sont nulles. Quand c'est le cas nous ne souhaitons pas stocker toutes les valeurs mais seulement celles qui sont non nulles. Par exemple pour :

$$V = (0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1.3 \quad 0 \quad 2.1 \quad 0 \quad 0 \quad 3.3 \quad 2.5 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \ ),$$

on stocke que les valeurs non-nulles et les indices de colonnes correspondants (commençant à 0) :

$$\begin{aligned} \text{index} &= (\ 5 \quad 7 \quad 10 \quad 11\ ) \\ \text{values} &= (1.3 \quad 2.1 \quad 3.3 \quad 2.5\ ). \end{aligned}$$

1. Dans un fichier *SparseVector.h*, on écrira la déclaration de la classe d'un vecteur creux, qui contiendra en attribut ces deux tableaux, par exemple :

```
1  class SparseVector {
2    private:
3      std::vector<double> _values;
4      std::vector<int> _index;
5    public: // Méthodes et opérateurs de la classe
6  };
```

2. On implémentera les méthodes suivantes (dont 3 constructeurs) dans la classe SparseVector :

- SparseVector() : constructeur par défaut d'un objet de la classe SparseVector.
- SparseVector(SparseVector const& u) : constructeur par copie.
- SparseVector(vector<double> const& vec_dense) : constructeur à partir d'un vecteur dense.
- GetNumOfNonZeroElem() : renvoie le nombre d'éléments non-nuls stockés.
- Resize(int n) : réalloue les tableaux pour stocker un nouveau nombre n d'éléments non-nuls.
- Index(int i) : renvoie le numéro de colonne associé à l'élément i.
- Value(int i) : renvoie la valeur associée à l'élément i.
- GetIndex() et GetValue() : renvoient les vecteurs _index et _values.

Index et Value doivent renvoyer des références de telle sorte qu'on pourra utiliser ces fonctions pour modifier le vecteur creux et donc faire dans votre fonction *main* (lignes 5-8) :

```
1  SparseVector u; // Constructeur par défaut
2  // on alloue u avec 4 éléments non-nuls:
3  u.Resize(4);
4  // on peut modifier u avec Index/Value (on a pris ici l'exemple donné au-dessus)
5  u.Index(0) = 5;  u.Value(0) = 1.3;
6  u.Index(1) = 7;  u.Value(1) = 2.1;
7  u.Index(2) = 10; u.Value(2) = 3.3;
8  u.Index(3) = 11; u.Value(3) = 2.5;
9  cout << "Nombre d'elements non-nuls " << u.GetNumOfNonZeroElem() << endl;
10 SparseVector w(u); // Constructeur par copie
11 vector<double> xx = {-2.1, 0., 5.6, 0., 36.1};
12 SparseVector x(xx);
```

3. Les 3 fonctions *Get* ne modifient pas le *SparseVector*. Afin de protéger le code nous allons rajouter un *const* dans leurs prototypes (dans le *h* et dans le *.cpp*), par exemple pour *GetIndex* :

```
std::vector<int> GetIndex() const;
```

Pour éviter de renvoyer un vecteur qui sera ensuite copié, on pourra renvoyer un *const&* :

```
std::vector<int> const& GetIndex() const;
```

4. Nous allons voir une fonctionnalité très importante du C++ : "la surcharge des opérateurs". Cette technique permet de réaliser des opérations mathématiques entre les objets en utilisant les symboles : $+, -, *, ==, <$ et d'afficher un objet à l'aide de $<<$. Pour notre classe SparseVector nous allons surcharger les opérateurs $==$ et $<<$ afin d'utiliser les commandes suivantes :

```
cout << u << endl; //Affiche le vecteur u
//Compare les vecteurs u et v
if(u == v)
  cout << "Les deux vecteurs sont égaux !" << endl;
```

L'affichage d'un vecteur creux doit écrire sur chaque ligne un élément non-nul, ce qui donne pour le vecteur u :

```
 5  1.3
 7  2.1
10  3.3
11  2.5
```

Les surcharges d'opérateur se font à l'extérieur de la classe : les prototypes dans le fichier .h se mettent après la fermeture de l'accolade et du point virgule et les définitions dans le .cpp se mettent à la fin du fichier pour plus de clarté.

```
//Surchage de ==
bool operator==(Objet const& o1, Objet const& o2)
{
   if (.....) // Effectuer la comparaison.
      return true;
   else
      return false;
}
//Surchage de <<
ostream& operator<<(ostream& out, Objet const& o)
{
    // Écrire le vecteur creux dans le flux "out" puis le renvoyer.
    return out;
}
```

Implémenter ces 2 surcharges d'opérateur et les tester dans la fonction *main* en ajoutant :

```
v.Resize(2);
v.Index(0) = 3; v.Value(0) = 3.7; v.Index(1) = 5; v.Value(1) = 2.4;
cout << "u = " << u << endl;
cout << "v = " << v << endl;
cout << "w = " << w << endl;
if(u == v)
```

```
  cout << "Les vecteurs u et v sont égaux !" << endl;
else
  cout << "Les vecteurs u et v ne sont pas égaux !" << endl;
if(u == w)
  cout << "Les vecteurs u et w sont égaux !" << endl;
else
  cout << "Les vecteurs u et w ne sont pas égaux !" << endl;
```

5. Pour finir cet exercice, nous allons voir une autre fonctionnalité du C++ : les "templates". L'objectif est d'utiliser le même code pour un vecteur creux d'entiers, de réels ou de complexes. La force des templates est d'autoriser une fonction ou une classe à utiliser des types différents. Vous pouvez les reconnaitre par les chevrons < et > comme le fait la STL par exemple pour les vecteurs :

```
vector<int> vecteur_int;
vector<double> vecteur_double;
// Inclure la librairie "complex" (encore un objet qui est aussi "templaté" !)
vector<complex <double> > vecteur_complex;
```

Nous souhaitons pouvoir faire pareil avec notre classe de vecteurs creux :

```
SparseVector<int> vecteur_creux_int;
SparseVector<double> vecteur_creux_double;
SparseVector<complex <double> > vecteur_creux_complex;
```

Il faut modifier la déclaration de votre classe (dans le .h) comme ceci :

```
template<class T>
class SparseVector {
  private:
    std::vector<T> _values;
    std::vector<int> _index;
  public:
  // méthodes de la classe : remplacer tous les double par T
};
```

Et ensuite pour la définition de vos méthodes et vos opérateurs (dans le .cpp), l'adaptation est rapide :

```
template<class T>
SparseVector<T>::SparseVector() {}

template<class T>
int SparseVector<T>::GetNumOfNonZeroElem() const
{
   // Corps de la fonction x
}

template<class T>
ostream& operator<<(ostream& out, SparseVector<T> v)
{
   // Corps de la fonction
}
```

Modifier votre code. À la compilation, vous avez une erreur lors de la création de liens :

```
Undefined symbols for architecture x86_64:
  "SparseVector<double>::Index(int)", referenced from:
      _main in main-99d7d6.o
               .
ld: symbol(s) not found for architecture x86_64
```

C'est une erreur classique pour les templates. En effet, le compilateur a besoin de connaître "T" au moment de la compilation pour générer la spécialisation. Plusieurs stratégies existent dans la littérature et dépendent du contexte. Ici nous allons en voir deux. Copier le code dans deux dossiers différents afin de pouvoir tester les deux stratégies.

6. La première stratégie est très utile quand on souhaite pouvoir utiliser n'importe quel T : int, double, complex etc ... sans avoir à préciser la liste. Il suffit de tout mettre dans le ".h". Copier donc le code qui est dans le ".cpp" à la fin de votre fichier ".h". Attention : si vous avez utilisé

```
using namespace std;
```

dans le ".cpp" vous devez modifier votre code. En effet il ne faut pas les inclure dans le ".h" (page 4, TP 3). Supprimer le fichier ".cpp" et compiler seulement votre fichier ".cc".
Les deux surchages d'opérateur n'étant pas des méthodes de la classe, vous n'avez pas besoin de conserver leurs prototypes dans le ".h". Les supprimer.
Créer un vecteur de réels, un vecteur d'entiers et un vecteur de complexes (ne pas oublier la librairie *complex*). Les afficher. Pour le vecteur de complexe, vous pouvez par exemple faire :

```
complex<double> z(3,1); // z = 3 + i
SparseVector<complex<double> > w;
w.Resize(1);
w.Index(0) = 2; w.Value(0) = z;
cout << "w = " << w << endl;
```

7. La deuxième stratégie consiste à définir à la fin du ".cpp" la liste des types que nous souhaitons :

```
template class SparseVector<double>;
template class SparseVector<int>;
template class SparseVector<complex <double> >; // ajouter #include <complex>
```

En ajoutant ces quelques lignes, il est possible de conserver les définitions des méthodes et des opérateurs dans le ".cpp".
Cependant si vous ne voulez pas avoir à créer pour chaque type vos deux surcharges d'opérateur, il faut les placer dans le ".h" (et supprimer alors leurs prototypes).

Créer un vecteur de réels, un vecteur d'entiers et un vecteur de complexes. Les afficher.

Exercice 2 - Une librairie d'algèbre linéaire : `Eigen`

Notre classe de vecteur creux est encore loin d'être finie. Il faudrait ajouter le produit par un scalaire, les produits scalaire et vectoriel, la transposition, les maximum et minimum ... Et puis pour faire du calcul scientifique il faudrait faire une classe de matrice puis une classe de matrice creuse et implémenter toutes les méthodes : addition, produit par un scalaire, produit de matrices, calcul du déterminant, inverse d'une matrice, calcul des valeurs et des vecteurs propres etc ... Et si on souhaite faire du parallèle et travailler sur plusieurs processeurs en même temps, il faudrait implémenter toutes ses méthodes en parallèle. Heureusement des librairies d'algèbre linéaires existent déjà et peuvent être utilisées en C++, de la même manière que vous avez déjà utilisé plusieurs librairies de la STL : iostream, string, vector, complex etc ... Voici une liste non exhaustive de librairies d'algèbre linéaire qui sont compatibles avec le C++ (attention cela ne signifie pas qu'elles sont codées en C++) :

- `Eigen` (C++). Matrices denses et creuses. En séquentiel. Facile à utiliser.
- `Armadillo` (C++). Matrices denses et creuses. En séquentiel. Facile à utiliser.
- `LAPACK` (Fortran). Matrices denses. En séquentiel. Difficile à utiliser.
- `PetSc` (C). Matrices creuses. En parallèle. Moyennement difficile à utiliser.
- `Trilinos` (C++). Matrices creuses. En parallèle. Difficile à utiliser.

Nous utiliserons dans ces TPs la librairie d'algèbre linéaire `Eigen`. Télécharger la librairie `Eigen` en suivant ce lien : Téléchargement `Eigen`. Extraire tous les fichiers et les placer dans un dossier *EigenLibrary* qui peut être par exemple dans votre dossier *TPC++*.
Pour compiler un fichier qui fait appel à *Eigen* il faut dire au compilateur où se situe le dossier *Eigen* qui est situé dans le dossier *EigenLibrary*. Pour trouver le chemin aller dans ce dossier, taper la commande linux *pwd* et récupérer le chemin. Ensuite il est possible de compiler un fichier *main.cc* par exemple rajoutant cette option :

```
g++ -std=c++11 -I /mon_chemin/EigenLibrary/Eigen -o run main.cc
```

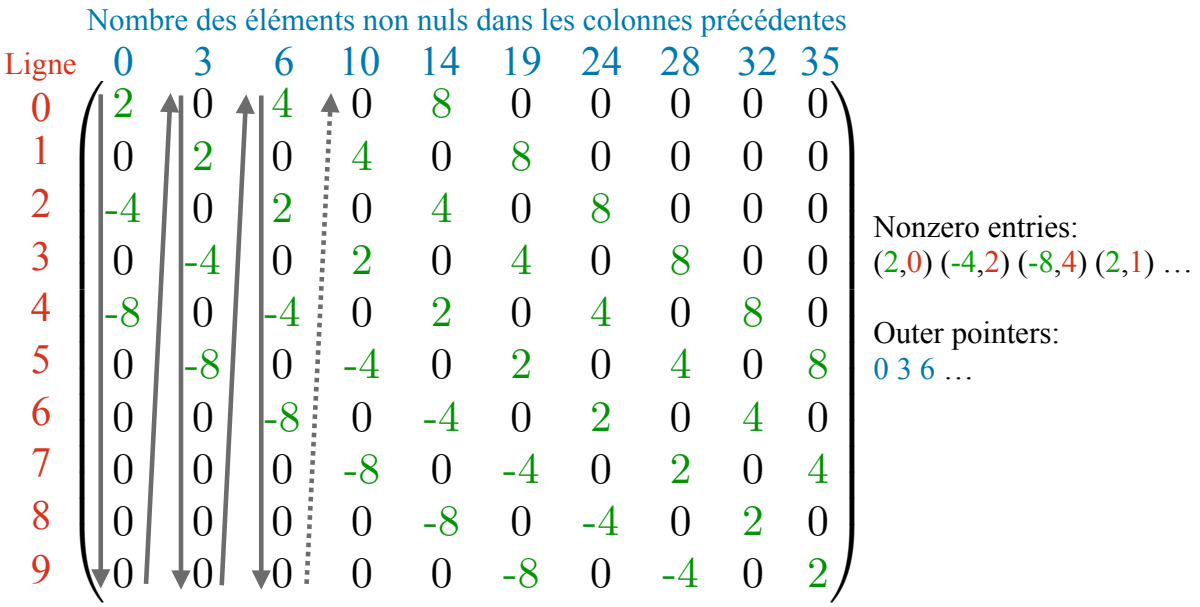
Télécharger deux fichiers en suivant ce lien : lien. Regarder ce que nous devons inclure pour pouvoir utiliser `Eigen` en haut des fichiers.

1. Compiler le fichier *main_ dense.cc* et l'exécuter. Regarder les commandes et leurs résultats dans le terminal. Cette étape est très importante pour comprendre comment construire une matrice dense. Des exemples de fonctionnalité sont aussi proposés. Bien sûr cette liste n'est pas exhaustive donc il ne faut pas hésiter à chercher dans la documentation d'`Eigen` quand on souhaite faire quelque chose qui n'est pas listé ici.

2. Compiler le fichier *main_ sparse.cc* et l'exécuter. Regarder attentivement ce fichier pour comprendre comment définir une matrice creuse. Les dernières commandes qui sont proposées vous montrent comment passer d'une matrice dense à une matrice creuse et vice-et-versa.
`Eigen` affiche une matrice creuse de deux manières différentes : tout d'abord une version creuse et ensuite une version dense. L'affichage creux correspond au mode de stockage de la matrice par `Eigen`. Par exemple la matrice T est stockée ainsi :

```
Nonzero entries:
(2,0) (-4,2) (-8,4) (2,1) (-4,3) (-8,5) (4,0) (2,2) (-4,4) (-8,6) (4,1) (2,3)
(-4,5) (-8,7) (8,0) (4,2) (2,4) (-4,6) (-8,8) (8,1) (4,3) (2,5) (-4,7) (-8,9)
(8,2) (4,4) (2,6) (-4,8) (8,3) (4,5) (2,7) (-4,9) (8,4) (4,6) (2,8) (8,5)
(4,7) (2,9)
Outer pointers:
0 3 6 10 14 19 24 28 32 35 $
```

La figure suivante explique ce stockage. Réfléchir à comment reconstruire la matrice à partir de *Nonzero entries* et *Outer pointers*.



Exercice 3 - Résolution de l'équation de Laplace en 1D avec `Eigen`

Nous allons à présent résoudre l'équation de Laplace en 1D avec conditions aux bord de Dirichlet homogène :

$$\left\{ \begin{array}{rcll} -u''(x) & = & f(x), & x \in]x_{min}, x_{max}[, \\ u(x) & = & 0 & x = x_{min}, \text{ et } x = x_{max}. \end{array} \right.$$

On discrétise l'intervalle $[x_{min}, x_{max}]$ par

$$(x_i)_{i=0,N+1} \text{ avec } x_i = x_{min} + ih \text{ et } h = \frac{x_{max} - x_{min}}{N+1}.$$

Soit u_i une approximation de $u(x_i)$ et on note $f_i = f(x_i)$. On considère le schéma suivant :

$$\left\{ \begin{array}{c} \dfrac{2u_i - u_{i+1} - u_{i-1}}{h^2} = f_i, \quad i = 1, N, \\ u_0 = u_{N+1} = 0 \end{array} \right.$$

En définissant les deux vecteurs suivants : $U = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_N \end{pmatrix}$ et $F = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_N \end{pmatrix}$ le problème s'écrit : $HU = F$ avec H la matrice suivante :

$$H = \frac{1}{h^2} \begin{pmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & -1 & 2 & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{pmatrix}.$$

1. Construire une classe *Laplacian1D* avec la structure suivante (fichier *Laplacian1D.h*) :

```
class Laplacian1D {
  private: // Les attributs de la classe
    const double _x_min;                  // x_min
    const double _x_max;                  // x_max
    const int _N;                         // N
    const double _h;                      // h = (x_max - x_min)/(N + 1)
    Eigen::SparseMatrix<double> _LapMat; // Matrice du Laplacien 1D (CREUSE)
    Eigen::VectorXd _x;                   // Vecteur (x_i)_{i=0..N-1} (DENSE)
    Eigen::VectorXd _f;                   // Vecteur source F (DENSE)
    Eigen::VectorXd _sol;                 // Vecteur solution U (DENSE)

  public: // Méthodes et opérateurs de la classe
    // Constructeur. Initialiser _x_min, _x_max, _N, _h et _x.
    Laplacian1D(const double x_min, const double x_max, const int N);
    // Construit la matrice _LapMat.
    void LaplacianMatrix();
    // Construit le vecteur source _f.
    void SourceTerm();
    // Résout le système _LapMat * _sol = _f avec un solveur direct.
    void DirectSolver();
    // Résout le système _LapMat * _sol = _f avec un solveur itératif.
    void IterativeSolver();
    // Calculer l'erreur L^∞ : sup|U - U_exact|  et la renvoyer
    double GetError();
    // Écrit les solutions dans le fichier "name_file".
    // Colonne 1 : _x ; Colonne 2 : _sol ; Colonne 3 : solution exacte.
    void SaveSol(std::string name_file);
};
```

Quelques remarques et/ou conseils :

- Dans le fichier *Laplacian1D.h* nous ne devons pas utiliser les *namespace*, en particulier le *namespace Eigen*. C'est pourquoi il faut rajouter "Eigen : :" devant les matrices et les vecteurs.
- Comme $\_x\_min$, $\_x\_max$, $\_N$ et $\_h$ sont constants ils doivent être initialisés dans la liste d'initialisation :

```
Laplacian1D::Laplacian1D(const double x_min, const double x_max, const int N)
: _x_min(x_min), _x_max(x_max), _N(N), _h((x_max-x_min)/(N+1.))
{
        ....
}
```

- Ne pas oublier de diviser par h^2 la matrice du Laplacien.
- Pour vérifier nos calculs nous allons définir le vecteur source à partir d'une solution exacte :

$$u(x) = (x - x_{min})(x - x_{max})e^{-x}$$

(attention la fonction choisie doit bien vérifier les conditions aux bord !). Il faut donc prendre :

$$f(x) = -e^{-x}(x^2 - (4 + x_{min} + x_{max})x + 2 + 2(x_{min} + x_{max}) + x_{min}x_{max})$$

- Concernant le *solveur*, `Eigen` en propose plusieurs dont une description est donnée sur cette page où il est expliqué comment choisir son solveur en fonction de la matrice : carrée ou rectangulaire, symétrique ou non, très creuse ou non, pattern irrégulier ou non etc ... Il est important de remarquer qu'il est aussi possible d'utiliser des solveurs extérieurs à `Eigen` en faisant appel à d'autres librairies.
  Nous allons proposer **à l'utilisateur de choisir** entre : un solveur direct *Cholesky* (classe : SimplicialLLT) et un solveur itératif *Gradient conjugué* (classe : ConjugateGradient).
  Quelque soit le solveur, voici la marche à suivre :

```
//Adapter SolverClass en fonction du solveur
SolverClass <SparseMatrix<double> > solver;
solver.compute(_LapMat);
_sol = solver.solve(_f);
```

- L'erreur correspond à la norme infinie entre la solution approchée et la solution exacte :

$$\sup_{i=1,N} |u_i - u(x_i)|.$$

- Ne pas oublier de rajouter u_0 et u_{N+1} au vecteur solution.
- Pour $N = 100$, $x_{min} = 0$ et $x_{max} = 1$, vous devez obtenir une erreur proche de 7.811×10^{-6}.

2. Afficher la solution approchée et la solution exacte avec `Gnuplot` pour $N = 10, 100, 500$ avec :

```
plot "solution.txt" using 1:2 title "solution approchee" with linespoints
replot "solution.txt" using 1:3 title "solution exacte" with linespoints
```

3. Nous souhaitons comparer les deux solveurs : *Cholesky* et *Gradient conjugué*. Vérifier que nous obtenons les mêmes résultats quel que soit le solveur. À présent, nous allons les comparer du point de vue du temps de calcul. Pour cela nous allons utiliser la librairie "chrono" (ne pas oublier de l'ajouter) qui s'utilise de la façon suivante :

```
#include <chrono> // Au début du fichier
auto start = chrono::high_resolution_clock::now(); // Démarrage du chrono
//Action que je souhaite chronométrer
auto finish = chrono::high_resolution_clock::now(); // Fin du chrono

// Différence entre les deux (affichage en microsecondes demandé)
double t = chrono::duration_cast<chrono::microseconds>(finish-start).count();
cout << "Cela a pris "<< t << " microsecondes" << endl; // Affichage du résultat
```

Conclure sur le solveur le plus adapté ici. Est-ce cohérent avec les conseils d'`Eigen` ?

Remarque : Vous avez sans doute remarqué que le type de *start* et *finish* est un type que vous n'avez encore jamais vu. En fait *auto* n'est pas un type. Cela signifie automatique : on laisse automatiquement le compilateur fixer le type de *start* et de *finish*. Pour cela les variables définies avec *auto* doivent être absolument initialisées. Si nous ne souhaitons pas utiliser *auto*, il faut aller voir dans la documentation de la librairie *chrono* quel est le type à utiliser (page) :

```
chrono::high_resolution_clock::time_point
```

Remplacer *auto* par ce dernier et vérifier que cela compile bien. En pratique, il ne faut pas abuser du *auto* mais quand il n'y a aucune ambiguïté comme c'est le cas ici, il ne faut pas hésiter.

4. Calculer l'ordre de la méthode :
   a. Calculer l'erreur pour un certain N. On la note e_N.
   b. Faire de même avec $2N$. On la note e_{2N}.
   c. Calculer l'erreur de la méthode par :

$$\text{ordreConv} = \log 2 \left(e_N \,/\, e_{2N} \right).$$

Exercice 4 - Résolution du problème aux valeurs propres de l'équation de Laplace en 1D avec `Eigen`

Nous allons continuer à compléter notre classe avec la résolution du problème aux valeurs propres suivant :

$$\begin{cases} -u''(x) &= \lambda u(x), & x \in]x_{min}, x_{max}[, \\ u(x) &= 0 & x = x_{min}, \text{ et } x = x_{max}. \end{cases}$$

1. **Rajouter** les variables privées et les méthodes suivantes dans votre fichier .h :

```
class Laplacian1D {
  private: // Les attributs de la classe
    .
    .
    Eigen::VectorXd _eigenvalues;

  public: // Méthodes et opérateurs de la classe
    .
    .
    // Résout le système _LapMat*_sol=_eigenvalues*_sol avec un solveur Eigen
    void EigenSolver();
    // Résout le système _LapMat*_sol=_eigenvalues*_sol avec un solveur Lapack
    void LapackEigenSolver();
    // Écrit les valeurs propres dans le fichier "name_file".
    // Colonne 1 : vecteur de 1 à N ; Colonne 2 : valeurs propres calculées ;
    // Colonne 3 : valeurs propres exactes.
    void SaveEigenSol(std::string name_file);
};
```

2. Nous allons proposer à l'utilisateur de choisir entre deux méthodes pour le calcul des valeurs propres. Implémenter les deux méthodes suivantes :

- **EigenSolver()** : Nous allons utiliser le solveur *SelfAdjointEigenSolver* (page d'Eigen) puisque notre matrice est symétrique. Il faut l'appliquer sur la version dense de la matrice de Laplace. Ne pas oublier de remplir le vecteur *_eigenvalues*.
- **LapackEigenSolver()** : Nous allons utiliser la librairie `Lapack` (Fortran 90) qui est une bibliothèque d'algèbre linéaire qui contient de nombreuses routines : résolution de systèmes linéaires, ajustement par des moindres carrés linéaires, diagonalisation, méthodes spectrales, nombreuses opérations de factorisation de matrices (QR, LU ...). Nous pouvons utiliser des routines `Lapack` directement avec les matrices `Eigen`. Ici nous allons utiliser la routine *dsyev* (pour les matrices symétriques) dont une explication est donnée ici : Lien `Lapack` (le nom des routines rend parfois son utilisation difficile ...). Pour utiliser une routine il faut d'abord exporter le prototype "C" de la fonction. Pour cela ajouter dans le fichier .cpp après les *include* :

```
extern "C" void dsyev_(char *jobz, char *uplo, const int* n, const double* A,
  int* LDA, double* VR, double* WORK, int* LWORK, int* info);
```

Voici ensuite comment s'utilise la routine `Lapack` :

```
// "N" = slmt les valeurs propres, "V" = valeurs propres + vecteurs propres
char p('N');
// "U" = partie supérieure de la matrice est stockée, "L" = partie inférieure
// Comme nous avons stocké toute la matrice cela nous est égal
char q('U');
// Plus grande dimension de la matrice (dans notre cas = _N)
int LDA = LapMatDense.outerStride();
// Adapter la taille du vecteur _eigenvalues
_eigenvalues.resize(_N);
// Lire la documentation de Lapack pour les deux suivants
int LWORK = 3*_N-1;
VectorXd WORK;
WORK.resize(LWORK);
// INFO est un entier = 0: tout s'est bien passé
//                    < 0:  si INFO = -i, l'argument ième n'est pas correct
//                    > 0:  l'algo ne converge pas
int INFO = 0;
// Appel de la routine
dsyev_(&p, &q, & _N, LapMatDense.data(), & LDA, _eigenvalues.data(),
             WORK.data(), & LWORK, & INFO);
```

Pour pouvoir compiler vous devez créer un lien vers la librairie `Lapack`. Sous Linux il faut ajouter la commande :

```
-llapack
```

lors de la compilation.

3. Implémenter la méthode *SaveEigenSol*. Les valeurs propres exactes sont données par :

$$\lambda_k = \frac{k^2\pi^2}{(x_{max} - x_{min})^2}, \quad k = 1, \cdots, \infty.$$

4. Nous allons maintenant pouvoir comparer les deux méthodes :
   a. Vérifier que vous obtenez les mêmes résultats quelle que soit la routine.
   b. Il est facile de remarquer que l'on approxime relativement bien les premières valeurs propres mais que rapidement il faut augmenter N pour pouvoir bien approximer des valeurs propres plus grandes. Par exemple pour approximer correctement les 200 premières valeurs propres nous devons prendre $N = 1000$. La rapidité de la routine devient donc très importante. Comparer les temps de calcul entre les deux stratégies et conclure.