T.P. 2

Un exemple de bibliothèque d'algèbre linéaire : Eigen

L'objectif principal de ce TP est de se familiariser avec l'utilisation de bibliothèques externes, en l'occurrence ici la bibliothèque `Eigen` qui est une boîte à outils très complète pour l'algèbre linéaire, et d'en voir deux applications possibles en mathématiques. Toute la documentation nécessaire figure sur le site http://eigen.tuxfamily.org/dox/. **Aller lire la documentation pour comprendre les prototypes et les modes d'emploi des différentes fonctions fait pleinement partie de l'exercice !**

Nous rappelons les bases suivantes :

— Il faut installer la bibliothèque `Eigen` sur son ordinateur (facile sous Ubuntu ou Linux Mint : il suffit d'installer le paquet `libeigen3-dev` par la commande `apt install libeigen3-dev` dans un terminal).

— Il faut compiler avec `g++` et l'option `-I /usr/include/eigen3` sur un système Linux. Pour tirer pleinement parti de la bibliothèque, l'option d'optimisation `-O2` est également conseillée (essayez avec et sans !).

— Il faut déclarer la bibliothèque `<Eigen/Dense>` ou `<Eigen/Sparse>` dans les en-têtes de fichiers selon le type de matrices souhaité.

— Une matrice à coefficients réels et de taille $N \times M$ fixée à l'écriture du programme se déclare par

```
Eigen::Matrix<double, N, M> A;
```

— Si la taille n'est pas fixée à l'écriture du programme mais à son exécution,il faut utiliser

```
Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic> A(N,M);
```

Pour alléger le code, on pourra enregistrer ce type sous la forme

```
typedef Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic>
        MatrixDouble;
```

— On accède au coefficient (i, j) par `A(i,j)` et la numérotation commence à 0 (et non à 1).

— On écrit une matrice avec `o << A` où `o` est un flux d'écriture de type `std::ostream`.
— on additionne (resp. multiplie) des matrices avec l'opérateur `+` (resp. `*`) et on les affecte avec `=`.

2.1 Premiers pas : la fonction puissance

Attention : les sections suivantes vous demandent de coder plusieurs fonctions de puissance : n'utilisez pas le même nom à chaque fois ! ! !

2.1.1 Un prototype de fonction récursive

On se propose d'écrire une fonction récursive[1] permettant de calculer la puissance d'une matrice. Elle se déclare comme suit :

```
MatrixDouble puissance_lente(const MatrixDouble & M, int n){
        if (n==0){  ...  }
        else if (n==1){  ...  }
        else {
                MatrixDouble N(...,...);
                N=puissance_lente(M,n-1);
                return ...
        }
}
```

et elle repose sur le raisonnement suivant :

si $n = 0$ alors $M^n = Id$, sinon $M^n = M(M^{n-1})$.

Question 2.1. Créer un fichier "matrice.cpp" et compléter le code ci-dessus. La matrice identité peut s'obtenir directement par

```
MatrixDouble::Identity(N,N) // pour une matrice carrée de taille N par N
```

qui renvoie la matrice identité. *Bonus : réécrire avec un* `switch` *au lieu d'une suite de test* `if`.

Question 2.2. Dans votre code, combien de multiplications sont effectuées ? Qu'en pensez-vous ? Par ailleurs, pourquoi ajoute-t-on le test `n==1` alors que, mathématiquement, il n'est pas nécessaire ?

Question 2.3. Dans le code `main()` de ce même fichier, déclarer la matrice

$$A = \begin{pmatrix} 0.4 & 0.6 & 0 \\ 0.75 & 0.25 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

1. Une fonction récursive est une fonction qui fait appel à elle-même en modifiant ses arguments jusqu'à atteindre une condition de sortie. Typiquement, une telle fonction f peut être décrite par $f(x, 0) = f_0(x)$, et $f(x, n) = g(x, f(n-1)(x))$ pour tout $n \geq 1$, avec g une fonction donnée.

puis calculer A^{100} et afficher le résultat dans le terminal. Pour déclarer une matrice ligne par ligne et sans devoir écrire `A(i,j) =...` , il est possible d'utiliser la *comma-initialization* décrite sur la page suivante de la documentation : https://eigen.tuxfamily.org/dox/group__TutorialMatrixClass.html.

Il est **fortement déconseillé** de passer aux questions suivantes avant d'avoir le bon résultat pour le calcul de A^{100}, à savoir

$$A^{100} = \begin{pmatrix} 0.555556 & 0.444444 & 0 \\ 0.555556 & 0.444444 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Question 2.4. Pourquoi y a-t-il une esperluette `&` dans le prototype de la fonction `puissance` ? Combien de copies sont réalisées lors du calcul ? Combien de copies seraient réalisées si la référence `&` était absente ? Faire le test sans `&` pour des puissances 100, 1000 et 10000 [2] et comparer les temps de calcul approximatifs (pour l'instant approximativement, voir plus bas pour des mesures précises).

2.1.2 Perspectives d'optimisation

La matrice A définie dans la section précédente est de petite taille, aussi est-il rapide de calculer ses puissances. Mais pour calculer la puissance n-ème d'une matrice de taille $N \times N$, il faudra n appels à la multiplication de matrices $N \times N$ qui quant à elle correspond à N^2 opérations, ce qui fait nN^2 calculs (on dit alors que l'algorithme qui sous-tend la fonction est de *complexité* $O(nN^2)$). On peut améliorer l'algorithme de la question 2 de la façon suivante :

> Si $n = 0$ alors $M^n = Id$. Sinon, si n est pair, alors $M^n = M^{n/2}M^{n/2}$, et si n est impair, alors $M^n = M(M^{(n-1)/2})(M^{(n-1)/2})$.

Question 2.5. Écrire une fonction `puissance_rapide` qui prend les mêmes types d'arguments d'entrée et de sortie que `puissance_rapide` mais fonctionne sur la récurrence ci-dessus.

Question 2.6. Combien de multiplications utilise votre code (en fonction de n) ? Qu'en pensez-vous ?

Question 2.7. Test : comparer les temps de calcul pour la puissance 1000-ème [3] de la matrice B de taille 30×30 donnée dans le fichier `matrice.dat` [4] (disponible sur le site du cours) selon que l'on utilise `puissance` ou `puissance2` . Tester également l'effet de l'utilisation ou non de l'option de compilation `-O2` de `g++` ainsi que de l'option `-DNDEBUG` (associée à Eigen).

2. Le choix de l'exposant est arbitraire et dépend surtout de votre machine : sur une machine plutôt ancienne et peu puissante, des différences apparaissent dès les petits exposants ; sur une machine récente, les différences de temps ne deviennent sensibles que pour des exposants grands. Nous vous laissons augmenter les exposants jusqu'à observer des différences notables.

3. Là encore, l'exposant est arbitraire et, selon la puissance de votre machine, nous vous laissons l'augmenter jusqu'à voir des différences significatives.

4. On rappelle que l'on peut lire les éléments successifs d'un fichier, séparés par une tabulation, en utilisant l'opérateur `>>` .

Pour cela, on pourra utiliser la bibliothèque `<chrono>` de `C++11` qui est plus précise que `time()`. Pour déterminer le temps effectué par un calcul donné, il suffit de procéder comme suit :

```
auto t1 = std::chrono::system_clock::now();
... // Calcul dont on veut mesurer la durée
auto t2 = std::chrono::system_clock::now();
std::chrono::duration<double> diff = t2-t1;
std::cout << "Il s'est ecoule " << diff.count() << "s." << std::endl;
```

Par ailleurs, la matrice B n'a pas été choisie complètement au hasard : il s'agit d'une *matrice creuse*, ou *sparse matrix* en anglais, c'est-à-dire une matrice qui possède un nombre limité de coefficients non nuls. La bibliothèque `Eigen` possède une façon d'encoder ce type de matrice qui permet de réduire drastiquement les temps de calcul. Pour cela, il faut déclarer `<Eigen/Sparse>` en tête de fichier, et utiliser le type `Eigen::SparseMatrix<double>` au lieu du type `Eigen::Matrix<double, N,M>`. La déclaration d'une matrice creuse se fait de manière similaire à celle d'une matrice dont la taille n'est pas connue à l'avance :

```
Eigen::SparseMatrix<double> Mat(N,M);
```

où N et M sont de type `int`. Les opérations $+$, $-$ et $\times$ s'utilisent de la même façon pour les matrices creuses que pour les matrices classiques, et il est également possible d'effectuer des opérations entre des matrices creuses et classiques :

```
Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic> A(N,N);
Eigen::SparseMatrix<double> B(N,N);
Eigen::Matrix<double,Eigen::Dynamic, Eigen::Dynamic> C1(N,N);
Eigen::Matrix<double,Eigen::Dynamic, Eigen::Dynamic> C2(N,N);
C1 = A*B;
C2 = A+B;
```

En revanche, contrairement au cas des matrices denses, l'accès et la modification du coefficient (i, j) d'une matrice creuse se fait avec `A.coeffRef(i,j)`.

On se référera à la page pour une documentation rapide sur les matrices creuses.

Une matrice creuse `M` prédéfinie peut être mise égale à la matrice identité avec la commande `M.setIdentity();`.

Question 2.8. Écrire une fonction `puissance_rapide_sparse` qui calcule la puissance n-ème d'une matrice creuse. Calculer[5] B^{1000} en écrivant B comme une matrice creuse et en lui appliquant cette fonction, puis comparer le temps de calcul à ceux des questions précédentes.

Question 2.9. (À traiter plus tard dans le semestre) Fusionner les fonctions `puissance_rapide` et `puissance_rapide_sparse` en un unique template de fonction `puissance_rapide<MatrixType>` compatible avec leurs deux types respectifs.

5. Là encore, choisir l'exposant suffisamment grand selon votre machine.

2.2 Matrices aléatoires et leur spectre

En théorie des matrices aléatoires, l'*ensemble gaussien orthogonal* (ou GOE, pour *gaussian orthogonal ensemble*) est l'ensemble des matrices symétriques $A \in \mathcal{M}_N(\mathbb{R})$ dont les coefficients de la diagonale et au-dessus de la diagonale sont indépendants et tels que $a_{ii} \sim \mathcal{N}(0,1)$ et $a_{ij} \sim \mathcal{N}(0,2)$ pour tout $1 \leq i < j \leq N$. En tant que matrices symétriques réelles, elles sont diagonalisables avec des valeurs propres réelles. On peut montrer que presque-sûrement (par rapport à la mesure de Lebesgue sur les matrices) ces valeurs propres $(\lambda_1, ..., \lambda_N)$ sont toutes distinctes. On peut alors définir la mesure empirique spectrale des valeurs propres [6]

$$\mu_N = \frac{1}{N} \sum_{i=1}^{N} \delta_{\frac{\lambda_i}{2\sqrt{N}}}.$$

Un résultat classique est que cette mesure converge étroitement, lorsque N tend vers l'infini, vers la mesure du demi-cercle

$$\mathrm{d}\sigma(x) = \frac{1}{\pi}\sqrt{4-x^2}\mathbf{1}_{[-2,2]}(x)\mathrm{d}x \tag{2.1}$$

dont la densité est un demi-cercle centré en 0 et de rayon 2. En particulier, la mesure limite est à support compact.

Diagonaliser avec `Eigen`. La bibliothèque `<Eigen/Eigenvalues>` permet de calculer les valeurs propres d'une matrice : étant donné une matrice `MatrixDouble M(N,N)`, on déclare l'algorithme de calcul de ses valeurs propres par

```
Eigen::EigenSolver<MatrixDouble> Solver(M);
```

et `auto spectrum=Solver.eigenvalues()` renvoie un vecteur `spectrum` de taille N à coefficients complexes contenant les différentes valeurs propres. Un nombre complexe $x+iy$ est modélisé comme un couple (x,y) et n'est donc pas de type `double`, mais comme on sait que dans le cas du GOE celles-ci sont réelles, elles sont égales à leur partie réelle. Aussi, on obtient la i-ème valeur propre réelle par `spectrum[i].real()`.

Générer des nombres aléatoires en `C++11`. On (rappelle qu'on) peut simuler une loi uniforme sur $[a,b[$ de la façon suivante en C++11 :

1. On inclut les bibliothèques `<random>` [7] et `<ctime>` ;
2. On déclare un générateur aléatoire :

```
std::mt19937_64 G(time(NULL));
```

3. On déclare la loi uniforme sur $[a,b[$:

6. modulo une renormalisation en $1/2\sqrt{N}$ pour des raisons de convergence.
7. La bibliothèque `<random>` nécessite de compiler avec l'option c++ 11 !

```
uniform_real_distribution<double> Loi(a,b);
```

ou encore la loi normale $\mathcal{N}(m, s)$ par :

```
std::normal_distribution<double> Loi(m,s);
```

4. On simule une variable aléatoire X qui suit cette loi cette loi via

```
double X;
X = Loi(G);
```

Tous les appels successifs de `Loi(G)` produisent des variables aléatoires indépendantes.

Afin de tester la convergence de μ_N vers la loi du demi-cercle décrite en (2.1), nous souhaitons réaliser un histogramme des valeurs propres. Cela signifie diviser le segment $[-3, 3]$ (par exemple) qui contient le support de σ en K segments consécutifs de même taille et compter le nombre de valeurs propres qui tombent dans chaque segment.

Question 2.10. Nous souhaitons à présent réaliser un histogramme à $K = 20$ boîtes sur le segment $[-3, 3]$ des valeurs propres normalisées $\lambda/(2\sqrt{N})$. Pour cela, on crée un vecteur `std::vector<double> hist(K,0)` dont chaque case `hist[k]` va contenir le nombre de valeurs propres normalisées qui tombent dans le segment $[-3+k(6/K), -3+(k+1)6/K[$.

On simule ensuite un nombre[8] $n = 20$ de matrices indépendantes $(GOE_k)_{1\leq k\leq n}$ de taille $N = 150$. Pour chacune de ces matrices, calculer ses valeurs propres (λ_i) et incrémenter `hist[k]` de $\frac{1}{nN}$ si la valeur propre normalisée $\lambda/(2\sqrt{N})$ tombe dans le segment $[-3 + k(6/M), -3 + (k + 1)6/M[$. Si la valeur propre normalisée ne tombe pas dans $[-3, 3[$, alors aucune case de `hist` n'est incrémentée.

Dans un fichier `"eigenvalues.dat"` , stocker dans deux colonnes séparées par une tabulation `"\t "` les centres de chaque segment $[-3 + k(6/M), -3 + (k + 1)6/M[$ et la valeur de `hist` correspondant à ce segment.

Remarque : si vous n'y parvenez pas, passez cette question et celle qui suit.

Question 2.11. En utilisant gnuplot, afficher le résultat de la question précédente avec la commande `plot "eigenvalues.dat" with boxes` . On pourra au besoin adapter l'échelle des ordonnées à l'aide de la commande `set yrange[a:b]` avec `a` et `b` respectivement les valeurs minimale et maximale des ordonnées que l'on veut afficher.

Question 2.12. Créer une fonction

```
auto generate_random_spectrum(std::mt19937 & G, N)
```

qui génère le spectre (cf. ci-dessus) d'une matrice du GOE de taille `N` en utilisant le générateur `G` . Tester.

8. Si le temps le permet, ne pas hésiter à simuler un nombre plus grand, par exemple 50 !

Question 2.13. Pour l'histogramme, le plus simple est de créer une classe assez simpliste qui va permettre d'organiser les choses. Nous vous proposons la déclaration suivante :

```
class Histogramme {
   private:
      double a;
      double b;
      double delta;
      std::vector<int> bars;
      int nb_out;
   public:
      Histogramme(double a, double b, int N):
         a(a), b(b), delta((b-a)/N), bars(N,0), nb_out(0) {}
      bool operator+=(double x) ; //ajoute un point dans la bonne barre
      double lower_bound() const; //accède à a
      double upper_bound() const; //accède à b
      double nb_boxes() const; //accède au nombre de barres de l'histo.
      int out_of_domain() const; //accède à nb_out
      void print(std::ostream & out) const;// affiche sur le flux out
      void reset(); //réinitialise à 0 en conservant les paramètres
};
```

où `a` et `b` sont les extrêmités du segment $[a, b]$ sur lequel on réalise un histogramme à N boîtes, chacune de largeur $\delta = (b - a)/N$. Chaque case `bars[k]` indique le nombre de points qui tombent dans $[a + k\delta, a + (k + 1)\delta]$ avec $0 \leq k < N$. Si un point x tombe en dehors du segment $[a, b]$, c'est le compteur `nb_out` qu'on incrémente.

1. Que signifient les différentes parties des lignes 9 et 10 ?
2. Comprendre les différentes méthodes (et les `const` et références associés).
3. Pourquoi ajouter le champ privé `delta` qui est a priori redondant avec `a` et `b`
4. Écrire le code des accesseurs.
5. Écrire le code de la méthode `print` . Pour cela, on écrira sur chaque ligne deux nombres séparés par une espace : le centre de la k-ème boîte et la hauteur de la barre associées dans l'histogramme.

Question 2.14. Reprendre la question 2.10 à l'aide de cette classe. Constater la grande amélioration de la conception et de la lisibilité du programme.