Sorbonne Université
Master 1 de mathématiques et applications
Unité d'enseignement MU4MA056

Programmation en C++ :

sujets des travaux pratiques

(version du 9 février 2024)

Année universitaire 2023–2024

Table des matières

Information importante : Vous trouverez sur la page web du cours un fichier `fichiersTP.zip` qui contient tous les fichiers de données et les codes à trous nécessaires pour les TP de cette brochure :

Paquets nécessaires sur un système Linux Debian/Ubuntu/Mint :

- compilateur `g++` et la bibliothèque `libstdc++5` (ou la version 6), cette dernière étant en général automatiquement installée
- la bibliothèque `libeigen3-dev` pour l'algèbre linéaire
- l'éditeur `geany`
- visualisation des données : Python avec `matplotlib` ou bien le logiciel `gnuplot`
- (optionnel) l'utilitaire de compilation `make`
- (optionnel) le compilateur `clang` (si vous souhaitez tester un autre compilateur)

T.P. 1

Premiers programmes, entrées/sorties, statistiques

1.1 Création et compilation d'un premier programme

1.1.1 Un premier programme

Considérons un programme très simple :

```
#include <iostream>
#include <cmath>

double aire_du_cercle(double x) {return M_PI*x*x;}

int main () {
        double r;
        std::cout << "Entrez le rayon du cercle:" << std::endl;
        std::cin >> r;
        std::cout << "Son aire est " << aire_du_cercle(r) << std::endl;
        return 0;
}
```

1. Créez un répertoire `TP1` sur votre bureau de travail.
2. Ouvrez un terminal et tapez `cd Desktop/TP1` ou `cd Bureau/TP1` (selon la langue du système) pour vous déplacer dans le répertoire créé. Laissez le terminal ouvert.
3. Ouvrez le logiciel Geany, créez un nouveau fichier, copiez le code ci-dessus et sauvegardez le fichier sous le nom `premierprog.cpp` dans le répertoire `TP1`. *Nous vous conseillons de le recopier à la main en essayant de comprendre chaque ligne.*
4. Revenez dans le terminal puis tapez

   ```
   g++ premierprog.cpp -o premierexec
   ```

   (selon la machine, il faut éventuellement ajouter `-lm` à la fin). Si un message apparaît, c'est qu'une erreur a été commise en recopiant le programme : corrigez-la.

5. Tapez à présent `./premierexec` dans le terminal. Il ne vous reste plus qu'à interagir avec votre programme !
6. Ajouter une fonction au programme précédent pour qu'il puisse calculer également l'aire d'un carré.
7. Modifier la fonction `main()` pour demander la longueur du côté d'un carré et afficher son aire.

1.1.2 Un deuxième programme

Considérons le code suivant :

```
#include __________
#include _________
#include ________

___ _____() {
        ___ n;
        _________ << "Entrez un nombre entier <100:" << std::endl;
        std::cin __ n;
        std::vector<int> tableau(n);
        for(__________________) {
                tableau[i]=i*i;
        }
        _____ofstream fichier("donnees.dat");
        fichier << "Voici les carrés des entiers:" << std::endl;
        for(__________________) {
                fichier << i <<": " <<tableau[i] << std::endl;
        }
        fichier._______;
        return 0;
}
```

1. Écrire ce programme dans un fichier `programme2.cpp` dans le répertoire `TP1`. Remplacer tous les `_____` par ce qu'il faut. Le compiler et l'exécuter. Que voyez-vous apparaître dans le répertoire `TP1` ?
2. Modifier ce programme pour avoir, dans le fichier `donnees.dat` sur les mêmes lignes, également les cubes des entiers.
3. Modifier le programme pour que les données soient écrites dans l'ordre décroissant dans le fichier.

1.2 Un brève présentation des entrées/sorties vers les fichiers

— L'écriture dans le terminal se fait avec `std::cout` et l'opérateur `<<`. Cet objet est défini dans la bibliothèque `iostream`.
— La lecture dans le terminal se fait avec `std::cin` et l'opérateur `>>`. Cet objet est défini dans la bibliothèque `iostream`.

— Un saut de ligne se fait avec l'opérateur `std::endl` .
— La déclaration d'un fichier en écriture se fait par

```
std::ofstream F ("Nom du fichier");
```

Cette commande est définie dans la bibliothèque `<fstream>` . On alimente le fichier en contenu avec l'opérateur d'injection `<<` selon la commande `F << CONTENU` où `CONTENU` est une valeur ou une variable.
— La déclaration d'un fichier en lecture se fait par

```
std::ifstream F ("Nom du fichier");
```

Cette commande est définie dans la bibliothèque `<fstream>` . On lit les données entre deux espaces avec `>>` selon `F >> x` où `x` est la variable de stockage des données lues.
— Avant la fin du programme, tout fichier doit être fermé avec `F.close();` .
— Il est possible de se débarrasser de tous les préfixes `std::` en écrivant :

```
using namespace std;
```

juste après les `include` .
— Toute la documentation des classes de la STL (vecteurs, listes, etc) est disponible sur http://www.cplusplus.com/reference/stl/ et sur http://www.cplusplus.com/reference/std/ pour celle sur les algorithmes, l'aléatoire, les complexes, etc.

1.3 Quelques calculs statistiques simples

1.3.1 Sans la bibliothèque `<algorithm>`

L'archive de données `fichiersTP.zip` contient un fichier `smalldata.txt` . Récupérez-le et placez-le dans votre répertoire de travail pour ce TP.

Ce fichier contient 2500 personnes, décrites par leur prénom, leur ville, leur âge et leur temps de course à l'épreuve du 100 mètres. Pour décrire une personne, nous introduisons la structure suivante :

```
struct Fiche {
        std::string prenom;
        std::string ville;
        int age;
        double temps;
};
```

Le type `std::string` permet de stocker des chaînes de caractères et est défini dans la bibliothèque `<string>` .

1. Créez dans `TP1` un programme `analyse.cpp` qui contient les bibliothèques nécessaires, la définition de la structure ci-dessus, une fonction `main()` qui ouvre le fichier `smalldata.txt` en lecture.
2. Déclarez dans ce programme un tableau `vdata` de taille 2500 et contenant des objets de type `Fiche`. Remplir ce tableau avec les données du fichier.
3. En utilisant uniquement des boucles `for`, des tests logiques `if` et en déclarant des variables, écrivez un programme (ou des programmes si vous préférez faire le travail en plusieurs fois) qui répond aux questions suivantes :
   (a) Combien de personnes habitent Lyon ? Quelle est le pourcentage de Lyonnais ?
   (b) Combien de personnes habitent Lyon et ont strictement moins de 30 ans ?
   (c) Existe-t-il un Toulousain dont le prénom commence par la lettre A ?
   (d) Quel est l'âge minimal ? L'âge maximal ? Comment s'appelle le plus jeune ? Le plus âgé ?
   (e) Quel est l'âge moyen des personnes du fichier ? Quel est l'écart-type de leur âge ?
   (f) Les Parisiens sont-ils en moyenne plus rapides au 100 mètres que les Marseillais ?
   (g) Produire un fichier `toulousains.txt` qui contient toutes les informations sur les personnes qui habitent Toulouse. On remplacera dans ce fichier leur âge par leur date de naissance (on supposera que les âges ont été déclarés en 2018).
   (h) Quelle est la covariance empirique entre âge et temps à l'épreuve du 100 mètres sur cet échantillon de Toulousains ?
   (i) Afficher dans le terminal la liste des villes représentées. *Attention, ce n'est pas si facile ! Vous pouvez utiliser si vous le souhaitez le conteneur* `std::set` *pour avoir une solution rapide ou sinon tout refaire à la main.*
4. (bonus) Supposons à présent que nous n'ayons pas donné initialement le nombre de personnes du fichier : cela empêcherait la déclaration du tableau statique `individu` à la bonne taille. Réécrire le début du programme en utilisant à présent un tableau `individu` de type `std::vector<Fiche>` de la classe `<vector>`. *Indication : le remplir avec* `push_back()` *en parcourant le fichier.*

1.3.2 Avec la bibliothèque `<algorithm>`

1. Refaire intégralement toutes les questions précédentes 3.(a) jusqu'à 3.(i) **sans écrire une seule boucle** `for` et en utilisant intensivement la bibliothèque standard `<algorithm>` dont une documentation est disponible sur le site suivant :

   http://www.cplusplus.com/reference/algorithm/

   Vous vous inspirerez des exemples décrits sur cette page pour chaque fonction. Pour certaines questions, vous pourrez également utiliser la fonction `std::accumulate` de la bibliothèque `<numeric>`.

   La plupart des fonctions de `<algorithm>` prennent en argument une fonction de test ou de comparaison : vous pourrez, au choix, soit déclarer ces fonctions dans le préambule de votre programme, soit dans le corps de la fonction `main()` en utilisant des lambda-fonctions du standard C++11.

   Exemple : pour la question (3)-a, il suffit d'écrire :

```
auto is_from_Lyon=[](Fiche f) {
                return (f.ville=="Lyon");}
int nb_Lyon=std::count_if(vdata.begin(),vdata.end(),is_from_Lyon);
std::cout << "Il y a " << nb_Lyon << " Lyonnais.\n";
```

2. Dans `<algorithm>` , il existe une fonction de tri `std::sort` qui fonctionne de la manière suivante. Si `v` est un vecteur d'objets de type `T` et `compare` une fonction de prototype :

```
bool compare(T x, T y)
```

qui renvoie `true` si `y` est plus grand que `x` et `false` sinon, alors l'instruction

```
std::sort(v.begin(),v.end(),compare)
        // pour v de type std::vector ou std::list
```

trie le tableau par ordre croissant. Produire un fichier `data_tri.txt` qui contient les 100 personnes les plus rapides au 100 mètres triées par vitesse décroissante.

1.3.3 Quelques questions additionnelles plus difficiles pour plus de réflexion

Vous pourrez traiter ces questions à la fois en écrivant vous même les boucles nécessaires et en définissant les variables nécessaires au calcul et à la fois en vous appuyant sur les outils de la bibliothèque standard.

1. Quel est le plus petit écart entre les temps de courses au 100 mètres de deux personnes (indice en note de bas de page [1]) ?
2. Créer deux vecteurs `jeunes` et `moinsjeunes` contenant respectivement les fiches des personnes de moins de 40 ans et de strictement plus de 40 ans (indice en bas de page [2]).
3. Écrire dans un fichier `ordre.dat` la liste des 2500 personnes classées selon l'ordre suivant :
   — par ordre alphabétique des prénoms
   — en cas d'égalité, par ordre alphabétique des villes
   — en cas d'égalité à nouveau, de la plus âgée à la plus jeune
   — en cas d'égalité à nouveau, de la plus lente à la plus rapide.
   et, bien sûr, vérifier que le fichier produit est correct.
4. Nous souhaitons établir l'histogramme des âges qui permette de connaître la répartition des âges. En utilisant le conteneur `std::map<int,int>` , calculer l'histogramme et l'afficher ligne par ligne dans le terminal. Quelle est la classe d'âge la plus nombreuse ?

1. Vous pouvez utiliser `std::sort` , `std::adjacent_difference` et rechercher un extremum. Nous vous conseillons tout d'abord d'extraire les temps de courses dans un `std::vector<double>` avant d'appliquer `std::adjacent_difference` à des `double` et non des `Fiche` .

2. Vous pourrez utiliser `std::partition_copy` et les itérateurs `std::back_inserter` de `<iterator>` si vous souhaitez utiliser `<algorithm>` pleinement et ne pas avoir à calculer au préalable le nombre de personnes de chaque catégorie.

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 :

— Sur vos ordinateurs personnels, 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). *Sur les ordinateurs de l'université, cette installation est* ***déjà*** *faite.*

— 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` et l'option `-DNDEBUG` sont également conseillées (essayez avec et sans !).

— Il faut déclarer la bibliothèque `<Eigen/Dense>` ou `<Eigen/Sparse>` dans les entêtes de fichiers selon le type de matrices souhaité, dense ou creuse (voir ci-dessous les détails)

— 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

```
using MatrixDouble =  Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic>;
```

— On accède au coefficient (i, j) par `A(i,j)` et la numérotation commence à 0 (et non à 1) pour une matrice dense et par `A.coeffRef(i,j)` pour une matrice creuse.
— On accède aux nombres de lignes et de colonnes par `A.rows()` et `A.cols()`.
— 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 tests* `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 ?

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.

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

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

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

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 `>>` .

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).

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.

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

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.

2.2 Matrices aléatoires et leur spectre

2.2.1 But de cet exercice

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. *Le but des questions qui suivent est d'illustrer ce phénomène.*

Pour cela, nous allons réaliser les tâches suivantes :

1. générer des matrices aléatoires indépendantes de grande taille N
2. calculer leur spectre
3. faire l'histogramme de leurs valeurs propres, c'est-à-dire partager un segment $[a, b]$ en K segments de même longueur et compter combien de valeurs propres tombent dans chaque segment.

2.2.2 Quelques indications en C++

2.2.2.1 Diagonaliser avec Eigen .

La sous-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, b);
        // Si b vaut true, les vecteurs propres sont aussi calculés.
        // SI b vaut false, seules les valeurs propres sont calculées.
```

6. modulo une renormalisation en $1/2\sqrt{N}$ pour des raisons de convergence.

Cet objet `Solver` prend une matrice `M` et, dès la construction, fait toutes les opération nécessaires pour la diagonaliser. Une fois cet objet construit et les calculs faits, l'accesseur `Solver.eigenvalues()` donne accès en lecture au vecteur des valeurs propres (complexes) (qu'on peut parcourir comme un conteneur de la STL). Par exemple,

```
for( std::complex<double> lambda : Solver.eigenvalues() ) {
        std::cout << lambda.real() << " ";        // lambda.imag() pour les parties imag
}
```

affiche toutes les parties réelles des valeurs propres.

2.2.3 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 la bibliothèque `<random>` (documentation : https://en.cppreference.com/w/cpp/numeric/random)
2. On déclare un générateur aléatoire au début de la fonction `main()` :

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

3. On déclare par exemple la loi uniforme réelle sur $[a, b[$ ou une loi normale $\mathcal{N}(m, s)$ par :

```
uniform_real_distribution<double> Loi1(a,b);
std::normal_distribution<double> Loi2(m,s);
```

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

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

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

2.2.3.1 Réalisation d'un histogramme

On considère un segment $[a, b[$ que l'on partage en K segments successifs de même taille $B_k = [a + k\delta, a + (k + 1)\delta[$ avec $0 \leq k < K$ et $\delta = (b - a)/K$: ils vont constituer les "boîtes" de l'histogramme.

On considère ensuite une suite de valeurs réelles $(\lambda_n)_{0\leq n<N}$. Réaliser l'histogramme des valeurs consiste à calculer les nombres $(h_k)_{0\leq k<N}$ tels que, pour chaque k, h_k est un entier indiquant le nombre de valeurs λ_n qui appartiennent à la boîte B_k. On peut éventuellement normaliser ensuite h_k par N pour obtenir la fraction de nombre de valeurs λ_n qui appartiennent à la boîte B_k.

Pour cela, il suffit de parcourir l'ensemble des valeurs $(\lambda_n)_{0\leq n<N}$, de trouver à quel B_k appartient chaque λ_n et d'incrémenter la valeur h_k correspondante.

2.2.4 Les questions

2.2.4.1 La mauvaise voie : tout en même temps

Question 2.10. Nous souhaitons à présent réaliser un histogramme à $K = 20$ boîtes sur le segment $[-3, 3]$ des valeurs propres normalisées $\lambda/(2\sqrt{N})$ d'un nombre $S = 50$ de matrices aléatoires de taille $N = 150$. Pour cela, peut on créer 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[7] $N = 50$ 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 $B_k = [-3+k(6/M), -3+(k+1)6/M[$. 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. Ce serait tout à fait légitime car on vous demande beaucoup de choses de natures très variées en même temps et, à la fin, le programme est peu lisible.

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.

2.2.4.2 La bonne voie : découper en tâches élémentaires

Question 2.12. Créer une fonction

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

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

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 K):
```

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

```
10          a(a), b(b), delta((b-a)/K), bars(K,0), nb_out(0) {}
              //accesseurs
12        double lower_bound() const; //accède à a
          double upper_bound() const; //accède à b
14        int nb_boxes() const; //accède au nombre de barres de l'histo.
          int nb_out_of_domain() const; //accède à nb_out
16      // autre méthode
          bool operator+=(double x) ;
18        //ajoute une donnée selon h += x en incrémentant la bonne case
          void print(std::ostream & out) const;// affiche sur le flux out
20        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 K boîtes, chacune de largeur $\delta = (b-a)/K$. Chaque case `bars[k]` indique le nombre de points qui tombent dans $[a + k\delta, a + (k+1)\delta]$ avec $0 \leq k < K$. 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.
6. Écrire le code de l'opérateur `+=` .

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.

T.P. 3

Classes et permutations

L'objectif de ce TP est d'écrire une classe représentant une famille d'objets mathématiques, les permutations, et de l'utiliser pour calculer numériquement certaines propriétés de ces objets.

Après quelques rappels théoriques pour fixer les définitions, on propose quelques questions auxquelles on essaiera de répondre numériquement pour deviner la réponse (en attendant une démonstration mathématique que vous pourrez chercher sur votre temps libre)

3.1 Permutations

3.1.1 Introduction mathématique

Une permutation σ de taille n est une bijection de $\{0, \dots, n-1\}$ dans lui-même. On note $\mathfrak{S}_n$ l'ensemble des permutations de taille n. Cet ensemble est de cardinal $n!$.

L'ensemble $\mathfrak{S}_n$ muni de la composition des applications a une structure de groupe, dont l'identité $id : i \mapsto i$ est l'élément neutre. Le produit $\sigma \cdot \tau$ de deux permutations est donc la permutation qui envoie tout élément i vers $\sigma(\tau(i))$. Attention, ce n'est pas commutatif.

L'*ordre* d'une permutation σ est le plus petit entier strictement positif k tel que $\sigma^k = id$. Les éléments $\{id, \sigma, \cdots, \sigma^{k-1} = \sigma^{-1}\}$ forment un sous-groupe de $\mathfrak{S}_n$ (le groupe engendré par σ) de cardinal k. Son action sur $\{0, \cdots, n-1\}$ définit naturellement une relation d'équivalence : $i \sim_\sigma j \Leftrightarrow \exists l \in \mathbb{Z} : i = \sigma^l j$. Les classes d'équivalence s'appellent les *orbites* de σ. Lorsqu'une orbite est un singleton, de la forme $\{i\}$, on dit que l'orbite est triviale et que i est un point fixe.

Un *cycle* non trivial est une permutation dont exactement une orbite est de longueur supérieure ou égale à deux. Cette orbite est appelée *support du cycle.*[1] La longueur d'un cycle est la longueur de sa plus grande orbite. L'ordre d'un cycle est la longueur du cycle. Une *transposition* est un cycle de longueur 2, qui échange donc deux éléments.

Un théorème dit que toute permutation se décompose en produit de cycles de supports disjoints. Cette décomposition est unique à l'ordre près des facteurs, qui commutent. L'ordre d'une permutation est alors le ppcm des ordres de tous les cycles qui la composent.

Une permutation σ peut être représentée de plusieurs façons : comme un tableau dont la première ligne liste les éléments de 0 à $n-1$, et la deuxième liste les images de

1. Par extension, pour inclure l'identité comme cycle trivial, on peut dire qu'un cycle a au plus une orbite non triviale.

l'élément au dessus :

$$\sigma = \begin{pmatrix} 0 & 1 & \cdots & n-1 \\ \sigma(0) & \sigma(1) & \cdots & \sigma(n-1) \end{pmatrix}$$

On donne parfois uniquement la seconde ligne du tableau, écrite comme un « mot dans les lettres $0, \ldots, n-1$ ».

Un cycle est parfois donné par la suite des images d'un élément de l'orbite non triviale (en oubliant les points fixes). On peut alors représenter une permutation en juxtaposant les cycles qui la composent.

Par exemple, la permutation σ de $\mathfrak{S}_6$ qui envoit respectivement 0,1,2,3,4,5 sur 2,4,5,3,1,0 peut être représenté par

$$\sigma = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 5 & 3 & 1 & 0 \end{pmatrix}$$

ou simplement $\sigma =$ "245310". Par itérations successives de σ, on a $0 \mapsto 2 \mapsto 5 \mapsto 0$, et $1 \mapsto 4 \mapsto 1$ alors que 3 est un point fixe. σ est donc le produit d'un cycle de longueur 2 (transposition) (14) et d'un cycle de longueur 3 (025). On écrit alors

$$\sigma = (025)(14).$$

en prenant la convention d'écrire chaque cycle avec le plus petit élément en premier, et de ranger les cycles dans l'ordre décroissant des premiers éléments (cette écriture est alors unique). Cette permutation est donc d'ordre $3 \times 2 = 6$.

3.1.2 Questions d'intérêt

Une fois introduites toutes ces définitions, on voudrait avoir une idée des nombres qui apparaissent pour toutes les grandeurs qu'on a introduites : combien y a-t-il de permutations sans point fixe[2] ? Avec 1, 2,..., k points fixes ? Quelle est la distribution de l'ordre des permutations parmi les éléments de $\mathfrak{S}_n$? du nombre de cycles ?

On essaiera de répondre à ces questions en cherchant à faire fonctionner le code suivant écrit dans un fichier `test_perm.cpp` . Il s'agit donc de se placer du côté du développeur mathématicien qui va développer une bibliothèque C++ `my_permutation` dans deux fichiers `my_permutation.hpp` et `my_permutation.cpp` où tous les formules mathématiques seront implémentées.

Les fichiers `file_s.dat` et `file_t.dat` seront à télécharger depuis le site du cours.

```
// ------ CONTENU DU FICHIER test_perm.cpp -------
#include "my_permutation.hpp"
#include <iostream>
#include <fstream>
#include <algorithm>
#include <iterator>
#include <random> // pour le générateur aléatoire std::mt19937
int main () {
// *******Premiere partie: bases de la bibliothèque
// Déclaration de deux permutations de deux manières différentes:
    Permutation b(6); //identite
```

2. Les permutations sans point fixe sont appelées des *dérangements*.

```
12      std::vector<int> v{2,4,5,3,1,0};//syntaxe C++11 avec { }
        Permutation a(v);
14  // Calcul des itérées d'une permutation
        for(int i=0; i<=6; ++i) {
16        std::cout << "a^" << i << "\n" << b << "\n";
          b = b*a;
18      }
    // Calcul des points fixes d'une permutation
20      std::list<int> fp = a.fixed_points();
        std::copy( fp.begin(), fp.end(),
22                 std::ostream_iterator<int>(std::cout," ") );
        std::cout << "\n";
24
    // ******* Deuxieme partie: un peu plus d'algorithmique
26  // lecture de deux permutations dans deux fichiers
        std::ifstream fichier_s("./file_s.dat");
28      std::ifstream fichier_t("./file_t.dat");
        Permutation s,t;
30      fichier_s >> s;
        fichier_t >> t;
32      fichier_s.close();
        fichier_t.close();
34   // Calcul de cycles et ordres
       std::list<Cycle> la = a.cycles();
36     cout << "Les cycles de a sont: ";
       for (const auto & c: la) cout << c << ";";
38     std::cout << "\nL'ordre de la permutation a est égal à "
                 << a.order() << "\n";//est-ce cohérent ?
40     Permutation u=s*t.inverse();
       std::list<Cycle> lu = u.cycles();
42     cout << "Les cycles de u ont pour longueurs: ";
       for(const Cycle & c:lu) cout << c.order() << " ";
44     long int order_u = u.order();
       std::cout << "\nL'ordre de la permutation u est égal à "
46               << order_u << "\n";//est-ce cohérent ?

48  // ******* Troisieme partie: génération aléatoire et Monte-Carlo
        std::mt19937 g(time(nullptr));
50      unsigned n=100;
        unsigned nb_echant = 10000;
52      unsigned nb_derang = 0;
        for(unsigned i = 0; i < nb_echant; ++i) {
54          nb_derang += Permutation(n,g).is_derangement();
        }
56      std::cout << "La proportion de dérangements est environ "
            << nb_derang/double(nb_echant) << "\n";
58      return 0;
    }
```

Pour cela, nous allons procéder pas à pas et nous vous recommandons dans le développement de n'importe quelle bibliothèque de procéder de même.

3.2 Implémentation de la classe

3.2.1 Préparation et outils élémentaires

Question 3.1. Créer deux fichiers `my_permutation.hpp` (en-tête) et `my_permutation.cpp` (code). Faire les inclusions de bibliothèques nécessaires.

Question 3.2. Définir dans le fichier d'en-tête une classe `Permutation` avec comme champs privés :

- un entier `n` qui est la taille de la permutation
- un vecteur d'entiers `images` qui contiendra plus tard les images de chaque point (i.e. `images[i]` est l'image de `i` par la permutation objet).

Question 3.3. Faire la liste dans le code précédent de *tous les constructeurs, méthodes et opérateurs nécessaires* pour faire marcher le code du fichier de test `test_perm.cpp` . Écrire leurs prototypes dans la partie publique de la classe en faisant bien attention aux `const` et aux passages des arguments par référence.

Il va falloir maintenant écrire dans le fichier `.cpp` les codes de chacun et les tester au fur et à mesure : il est donc *important* de s'y prendre dans le bon ordre. Nous vous conseillons donc la progression suivante.

3.2.2 Initialisation et entrées-sorties

Question 3.4. Écrire le code du constructeur qui permet de déclarer une variable en l'initialisant à l'identité de taille n (ligne 10 du code ci-dessus). Donner une valeur par défaut 1 à l'argument du constructeur.

Question 3.5. Surcharger l'opérateur d'affichage `<<` (ligne 15 du code ci-dessus). On souhaite qu'une permutation de taille n s'affiche sur une seule ligne avec la taille n sur la première ligne puis une espace puis deux points `:` puis une espace et enfin les images successives séparées par des espaces. La permutation `a` précédente s'affiche alors sous la forme :

```
6 : 2 4 5 3 1 0
```

Question 3.6. Tester les deux dans un court programme de quelques lignes pour vérifier que vous êtes capable d'afficher une permutation égale à l'identité.

Question 3.7. Surcharger l'opérateur de lecture `>>` (lignes 28 et 29). Le tester dans un court programme en lisant le fichier `file_s.dat` dans une permutation et en la réécrivant dans un fichier `test_file_s.dat` (il faut alors vérifier que les deux sont identiques !).

Question 3.8. Ajouter un accesseur `size()` au champ `n` (taille de la permutation).

Question 3.9. Ajouter un accesseur à la i-ème case du vecteur `images` via l'opérateur `[]` de sorte que `a[i]` renvoie l'image de `i` par la permutation `a`. Le tester.

Question 3.10. À quoi ressemblerait le mutateur associé ? Pourquoi ne faut-il surtout pas le mettre ?

Question 3.11. Écrire le constructeur utilisé à la ligne 11 qui crée une permutation telle que les images de chaque point sont les nombres lus dans le vecteur donné en argument. On supposera que le contenu du vecteur de départ contient une et une seule fois chaque entier.

Question 3.12. Faut-il définir un constructeur par copie ? Un opérateur d'affectation `=` ? Justifier.

3.2.3 Méthodes et opérateurs supplémentaires

Nous pouvons à présent commencer à coder le reste des méthodes et opérateurs en utilisant le travail précédent pour vérifier rapidement des résultats.

Question 3.13. Écrire le code de la composition de deux permutations comme surcharge de `*` (utilisé par exemple ligne 16). On pourra le faire soit par une méthode, soit par une fonction. Tester les lignes 14 à 17. On prendra les conventions suivantes :

- si les deux permutations sont de même taille, alors c'est la composition usuelle définie ci-dessus.
- si les deux permutations ont des tailles n et m avec $n < m$, alors on considérera la "petite permutation" sur $\{1, \dots, n\}$ comme une permutation σ sur $\{1, \dots, m\}$ telle que $\sigma(i) = i$ pour $i > n$. Le résultat sera alors de taille m.

Question 3.14. Écrire le code de la méthode `fixed_points`. Tester les lignes 19 à 22. Écrire également la méthode `is_derangement` qui renvoie vraie ou faux selon que la permutation est un dérangement ou non.

Question 3.15. Écrire la méthode `inverse` telle que `a.inverse()` ne modifie pas `a` et renvoie l'inverse de `a` pour la composition.

3.2.4 Décomposition en cycles

Pour décrire un cycle qui est un exemple particulier de permutation, nous introduisons la classe suivante

```
class Cycle {
   private:
      std::vector<int> elem;
      void add_last_point(int);//ajoute un point à la fin du cycle
      //std::vector<int>::iterator find(int) const;
   public:
      long int order() const;
      //int operator[](int ) const; //uses find
```

```
    //Cycle inverse() const;
};
```

Le champ `elem` contient les éléments du cycle de sorte que `elem[k+1]` est l'image par la permutation de `elem[k]` et `elem.front()` est l'image de `elem.back()`. Les éléments qui n'apparaissent pas dans `elem` sont considérés comme des points fixes.

Question 3.16. Est-il nécessaire de définir le constructeur par défaut ?

Question 3.17. Compléter la méthode *privée* `add_last_point` qui ajoute un point dans le cycle sans aucune vérification. Pourquoi mettre cette méthode privée ?

Question 3.18. Compléter l'accesseur `order()` à l'ordre du cycle.

Question 3.19. Surcharger `<<` pour les cycles, de telle sorte que l'affichage soit le suivant

```
[ elem[0] ..... elem[k] ]
```

Question 3.20. Écrire la méthode `cycles()` de la classe `Permutation` qui renvoie une liste d'éléments de type `Cycle`. On propose pour cela l'algorithme suivant : créer une liste de cycles vide `L`. Mettre dans un ensemble `S` tous les éléments de 0 à $n-1$. Tant que l'ensemble `S` n'est pas vide, retirer le plus petit `x` (accessible par l'itérateur `begin()` de la classe `std::set`). Si `x` n'est pas un point fixe, commencer un cycle en ajoutant `x` à un cycle vide. Retirer (utiliser `find` et `erase` dans `std::set`) de `S` les images itérées de `x` et les ajouter au cycle commencé jusqu'à retomber sur `x`. Ajouter alors le cycle ainsi formé à la liste `L`. Une fois que `S` est vide, renvoyer `L`.

Remarque : observer que le cycle ainsi construit commence bien par son plus petit élément car `S` est ordonné.

On observe également que pour cela il faut que la classe `Permutation` soit amie de `Cycle` afin de pouvoir utiliser `add_last_point`. Faites le nécessaire.

Question 3.21. Nous allons avoir besoin de calculer le PGCD (GCD en anglais) et le PPCM (LCM en anglais) d'une liste de grands nombres. Rappel : pour deux entiers a et b, le pgcd d de a et b peut se calculer itérativement par l'algorithme de la division euclidienne. On pose $(u_1, u_2) = (a, b)$. Si $u_2 = 0$, on renvoie u_1, sinon, tant que $u_2 \neq 0$, on remplace (u_1, u_2) par (u_2, r) où `r=u_1 % u_2` est le reste dans la division euclidienne de u_1 par u_2. Le PPCM est alors $m = ab/d$ où d est le PGCD de a et b. Écrire deux fonctions

```
long int pgcd(long int a,long int b);
long int ppcm(long int a,long int b);
```

qui calculent ces PGCD et PPCM.

Question 3.22. Les PGCD et PPCM sont deux opérations associatives et commutatives de telle sorte que le PGCD (resp. PPCM) de n nombres $(a_1, \ldots, a_n)$ peut se faire de la manière suivante : on pose $P_0 = 0$ puis $P_{k+1} = pgcd(P_k, a_k)$ (resp. $ppcm(P_k, a_k)$, pour $1 \leq k \leq n$ et le PGCD (resp. PPCM) est donné par le dernier terme P_n. Écrire le code de la méthode

```
long int Permutation::order() const;
```

en combinant la méthode `Cycle`, la fonction `ppcm` et `std::algorithm`. Le code ne doit pas prendre plus de deux ou trois instructions (sinon, faites à votre manière).

Question 3.23. Tester les lignes 34 à 46 sur le calcul de les ordres de `a` et `u`. Cela vous semble-t-il normal ? Où est le problème ? Avez-vous des idées pour le résoudre ?

Question 3.24. Ajouter un opérateur de comparaison `<` entre deux cycles qui fonctionne de la manière suivante : si les deux cycles ont des longueurs différentes alors on compare leurs longueurs ; en cas d'égalité de longueurs, on regarde l'ordre de leurs premiers éléments puis, en cas d'égalité on regarde leurs deuxièmes éléments, etc (on pourra si on le souhaite se référer à `std::mismatch` dans `<algorithm>`, voire encore mieux à `std::lexicographical_compare`). Cette méthode est nécessaire pour `std::max_element` en ligne 43 du code. Tester les lignes 39 à 44 sur `u`.

Questions supplémentaires sur les cycles non-nécessaires pour le code précédent mais nécessaires pour la section 3.3 ci-dessous.

Question 3.25. Ajouter une méthode `find(i)` qui cherche un élément `i` dans `elem`, renvoie l'itérateur sur son emplacement si `i` est présent et renvoie `elem.end()` sinon.

Question 3.26. Ajouter un accesseur `[i]` à l'image de `i` par un cycle. On pourra utiliser `find` ci-dessus pour distinguer les points fixes pour lesquels il faut renvoyer `i`.

Question 3.27. Ajouter une méthode `inverse()` à la classe `Cycle`.

Question 3.28. Ajouter une méthode `cycles()` à `Cycle` sur le modèle de la même méthode de la classe permutation (attention, c'est trivial).

3.2.5 Génération aléatoire

Nous allons maintenant coder le constructeur utilisé ligne 54. Il prend comme argument un entier qui est la taille de la permutation et un générateur aléatoire de la bibliothèque standard et construit une permutation tirée uniformément sur le groupe symétrique $\mathfrak{S}_n$. On suit l'algorithme suivant de Fisher-Yates-Knuth [3] : on commence par remplir le vecteur `images` par les éléments de 0 à $n-1$. Puis pour tout i entre 0 et $n-2$, on génère un entier j uniforme entre i et $n-1$ inclus (avec `std::uniform_int_distribution<int>`) et échanger les valeurs `images[i]` et `images[j]` (si i=j, on a un point fixe). On pourra

3. Remarque : cet algorithme de mélange de n éléments est également implémenté par `std::shuffle` dans `<algorithm>`.

utiliser `std::swap` pour l'échange sans tampon intermédiaire explicite (c'est géré en interne par `std::swap`). Estimer la complexité d'une construction d'une permutation aléatoire de taille n par cet algorithme.

Question 3.29. Écrire le code du constructeur pour un générateur de type `std::mt19937`. Pourquoi faut-il passer ce dernier par référence ?

Question 3.30. Tester la dernière partie du programme.

Question 3.31. Remplacer le constructeur précédent par un template de constructeur qui marche pour tout générateur de la bibliothèque standard. Tester à nouveau la dernière partie du programme.

3.3 Héritage virtuel

On se rend compte que la classe `Cycle` est un cas particulier de permutation et on souhaite s'assurer que les prototypes des méthodes sont bien les mêmes entre les deux classes. De plus, on souhaite pouvoir écrire des fonctions qui utilisent indifféremment un objet de type `Cycle` ou `Permutation` (tant que, bien sûr, les différences entre les deux objets ne sont pas pertinentes). Pour cela, tant que nous n'avons pas vu les templates, une solution historique consiste à utiliser l'héritage pur.

Pour cela, nous définissons la classe virtuelle pure suivante :

```
class Non_modifiable_Permutation {
        public:
                virtual int size() const = 0;
                virtual int operator[](int) const = 0;
                virtual long int order() const = 0;
                //...
};
```

Question 3.32. Faire hériter `Permutation` et `Cycle` de cette classe.

Question 3.33. Compléter toutes les méthodes possibles dans la classe-mère, communes aux deux classes-filles.

Question 3.34. Que manque-t-il pour `Cycle` ? (compilez et vous saurez) Pour associer une taille au cycle, on pourra par exemple regarder le plus grand élément du cycle et le prendre comme taille.

Question 3.35. Vérifier que tout fonctionne bien.

3.4 Une implémentation alternative pour les permutations avec beaucoup de points fixes

L'implémentation précédente de `Permutation` est nécessaire lorsque la permutation a peu de points fixes : il faut stocker les images de presque tous les points et ajouter quelques points fixes ne change rien (voire permet de simplifier certains algorithmiques). Supposons à présent que nous soyons dans une situation très différente avec n très grand et la plupart des points sont des points fixes. Il devient alors avantageux de stocker seulement l'image des points qui ne sont pas fixes.

Pour cela, on introduit la classe suivante

```
class SparsePermutation {
        private:
                int n;
                std::map<int,int> non_trivial_images;
        public:
                int operator[] (int i) const;
                //...
};
```

(qu'on pourra plus tard si on le souhaite faire hériter de `Non_modifiable_Permutation`). Cela nous donne l'occasion d'introduire le conteneur `std::map<T1,T2>` de la bibliothèque `std::map` (cf. cours). Il permet de décrire une application $f : E_1 \to E_2$ où E_1 et E_2 sont des ensembles *finis* d'objets de type `T1` et `T2` respectivement où `T1` est un ensemble ordonné et `T2` est muni d'une valeur par défaut. D'un point de vue informatique, cette fonction est encodée sous la forme d'une p_1-uplet de valeurs $((x_1, f(x_1)), \ldots, (x_{p_1}, f(x_{p_1})$ où $x_1 < x_2 < \ldots < x_{p_1}$ est une énumération croissante des éléments de E_1.

Si l'on sait que x fait partie de l'ensemble E_1, on peut accéder à l'image $f(x)$ via la syntaxe `f[x]` . Si $x \notin E_1$ alors, l'appel `f[x]` modifie l'objet f en *ajoutant à* E_1 l'élément x et l'élément (x, d) au p_1-uplet de valeur où d est la valeur par défaut du type `T2` . Si on cherche seulement à savoir si $x \in E_1$ et, dans ce cas seulement, à utiliser $f(x)$ alors il faudra faire

```
T1 x= ??? ;
auto where_x = f.find(x);
T2 value_x;
if ( where_x != f.end() ) {
        value_x = where_x->second;
}// si where_x == f.end() alors x n'est pas dans la map.
```

Ici `where_x` est un itérateur vers la case où est écrit la valeur `x` et la valeur `*where_x` est une paire $(x, f(x))$ de type `std::pair<T1,T2>` , de telle sorte que `where_x->first` et `where_x->second` donnent x et $f(x)$.

Nous prendrons alors l'exemple suivant :

```
int SparsePermutation::operator[](int i) const {
   auto where_i = non_trivial_images.find(i);
```

```
    if ( where_i != non_trivial_images.end() )
4       return where_i->second;
    else
6       return i;
}
```

Question 3.36. Ajouter toutes les méthodes nécessaires pour que `SparsePermutation` fonctionne comme `Permutation` ainsi que les opérateurs d'écriture, de lecture et le produit.

Question 3.37. On pourra tester les temps de calcul avec le programme `test_sparse.cpp` sur les fichiers de données `fichier_u.dat` et `fichier_sparse_u.dat` où la même grande permutation est écrite dans deux formats différents.

La deuxième partie des TP viendra plus tard...

T.P. 4

Une exploration de la bibliothèque standard

fencour Ce TP est une revue assez sommaire de bibliothèques importantes incluses dans la bibliothèque standard. Certaines d'entre elles n'ont été introduites qu'à partir du standard C++11 mais sont déjà bien ancrées dans la tradition.

Le TP1 vous a permis de découvrir brièvement `<vector>` et une partie de la bibliothèque `<algorithm>`. N'hésitez pas à (ré)utiliser autant que possible cette dernière dans ce TP.

4.1 Exercice avec `<random>` : un exemple d'héritage avec la marche aléatoire

Soit la marche aléatoire simple $(S_n)_{n\geq 0}$ définie pour tout $n \geq 0$ par $S_n = \sum_{k=1}^{n} X_k$ où les X_k sont des v.a. indépendantes telles que $\mathbb{P}(X_k = 1) = p$ et $\mathbb{P}(X_k = -1) = 1-p$. Nous souhaitons réaliser une simulation de cette marche aléatoire et du processus $(M_n)_{n\in\mathbb{N}}$ défini par $M_n = \min(S_0, \ldots, S_n)$.

Pour cela, vous utiliserez la bibliothèque `<random>` et un générateur de nombres pseudo-aléatoires de type `std::mt19937`. Une documentation de la classe est disponible à l'adresse :

https://en.cppreference.com/w/cpp/numeric/random

Question 4.1. Écrire une courte classe

```
class RandomWalk {
   protected:
      unsigned n;//temps courant n
      int s;// valeur de S_n
      int s_init;//valeur de S_0
      std::bernoulli_distribution U;//paramètre p
   public:
      RandomWalk(int s0,double p);
      ... val(...) ...  //accesseur à s
      ... time(...) ...  //accesseur à n;
      ... reset() ... // redémarrage à l'état initial
```

```
12      template .... void update( ... G) ... ;
            // mise à jour de s: passage de n à n+1 un générateur G de type arbitraire
14  };
```

pour implémenter l'évolution de la marche aléatoire $(S_n)_{n\in\mathbb{N}}$.

Question 4.2. Tester votre classe avec un court programme qui affiche plusieurs réalisations des 10 premiers pas d'une marche.

Question 4.3. Écrire une courte classe

```
class RandomWalk_with_Min: public RandomWalk {
2   protected:
       int m;//valeur de M_n
4   public:
       RandomWalk_with_Min(int s0,double p): ...
6      ... minimum() ...; //accesseur à m
       ...//compléter/modifier les méth. de la classe mère si nécessaire.
8   };
```

qui hérite de la précédente et ajoute le calcul du minimum absolu courant (M_n). *Aucune modification de* `RandomWalk` *ne doit être faite et aucune répétition de code par rapport à* `RandomWalk` *ne doit être présente.*

Question 4.4. Écrire un programme `RW.cpp` qui utilise la classe précédente pour réaliser une simulation de longueur $T = 10000$ et écrit les valeurs successives de $(S_n)_{0\leq n<T}$ dans un fichier `RW.dat` et celles de $(M_n)_{0\leq n<T}$ dans un fichier `RWmin.dat` , en mettant sur chaque ligne le temps n, la valeur de S_n et celle de M_n, toutes séparées par des espaces.

Question 4.5. Visualisez les fichiers avec `gnuplot` en tapant dans ce dernier :

```
plot "RW.dat" with lines, "RWmin.dat" with lines
```

Relancer plusieurs fois le programme et observer le changement graphique. Cela vous semble-t-il cohérent ?

4.2 Exercice sur conteneurs et itérateurs : mesure de performance

4.2.1 Rappels

Le mot *conteneur* désigne en C++ une classe (ou plus précisément un *template* de classe) qui permet de décrire une collection d'objets de même type. Selon le *conteneur*, cette collection peut être ordonnée ou non et le codage en mémoire et les complexités peuvent différer. Plusieurs opérations essentielles sont communes à tous les conteneurs :

1. compter le nombre d'éléments du conteneur (`size()`),
2. effacer tout le conteneur (`clear()`),
3. ajouter ou enlever un élément (éventuellement à un endroit précis si le conteneur est ordonné) : selon l'endroit et/ou le conteneur, `insert` , `erase` , `pop_front` (enlever au début), `pop_back` (enlever à la fin), `push_front` (ajouter au début), `push_back` (ajouter à la fin)
4. la possibilité de le parcourir entièrement sans oublier d'éléments (de manière ordonnée si le conteneur l'est, aléatoirement sinon) via

```
for(auto x : conteneur) //ou bien const auto & ou bien auto &
```

   ou toute fonction de `<algorithm>` via les itérateurs de début et de fin,
5. trouver si un élément est présent et récupérer l'emplacement où il est stocké (`find`)

Nous vous encourageons à consulter l'immense tableau de la page

https://en.cppreference.com/w/cpp/container

pour voir quelles opérations sont disponibles sur quels conteneurs.

Les deux derniers points de la liste ci-dessus sont gérés par le concept d'*itérateur*. Un *itérateur* est un objet qui permet :

— de pointer un emplacement d'un conteneur
— de se déplacer intelligemment dans un conteneur.

Étant donné un itérateur `it` , on peut :

— accéder à la valeur sur l'emplacement indiqué par `*it` ,
— passer à l'élément suivant par `it++` ou `++it` ,

Chaque itérateur existe sous une version en lecture et écriture et une version en lecture seule. Les prototypes sont donnés par

```
std::CONTENEUR<TYPE>::iterator
std::CONTENEUR<TYPE>::const_iterator
```

Chaque conteneur `X` possède deux méthodes

— `X.begin()` qui pointe vers son premier élément (`X.cbegin()` pour la version en lecture seule),
— `X.end()` qui pointe vers un élément fantôme indiquant qu'on a déjà franchi le dernier élément (`X.cend()` pour la version en lecture seule),

Remarque : vous avez déjà utilisé les itérateurs à chaque fois que vous avez utilisé une fonction de `<algorithm>` .

4.2.2 Performances sur différents conteneurs ordonnés

Question 4.6. (performance de quelques algorithmes sur les vecteurs) Écrire une fonction

```
void measure_complexity_on_vector(long int N)
```

qui fait les étapes suivantes :

- crée deux vecteurs `c1` et `c2` de `double` de taille N,
- remplit le premier avec des nombres aléatoires indépendants de loi exponentielle de paramètre 1,
- compte le nombre d'éléments supérieurs à 10 dans le premier,
- élève au carré les 100 premiers éléments du premier,
- copie le premier dans le deuxième,
- trie la première moitié du premier avec `std::sort`
- échange le contenu des deux vecteurs via `std::swap` .
- écrit le deuxième dans un fichier `chocolatine.cpp`

Pour chaque étape, vous chronométrerez (avec une procédure similaire à ce qui a été fait dans le TP2 en utilisant `<chrono>`) le temps de calcul nécessaire et l'afficherez dans la console.

Question 4.7. Utilisez la fonction pour $N = M = 10000000$ (dix millions), puis $N = 2M$, $N = 4M$ et $N = 8M$. **Discuter** l'évolution des temps obtenus le plus précisément possible..

Question 4.8. Faire une deuxième fonction en utilisant `std::deque` à la place de `std::vector` . **Discuter** les différences de temps avec l'exemple précédent.

Question 4.9. Faire une troisième fonction en utilisant `std::list` à la place `std::vector` *sans faire l'étape de tri.* **Discuter** les différences de temps avec les exemples précédents.

Question 4.10. (bonus) Si vous avez utilisé la bibliothèque `<algorithm>` , refaites le programme en écrivant toutes les boucles à la main (sauf pour le tri) et comparez les temps obtenus. Si vous avez tout fait avec des boucles, refaites le programme avec `<algorithm>` et comparez les temps obtenus.

4.3 Exercice sur le conteneur `std::map` : analyse d'un texte

Le conteneur `std::map` (ainsi que `std::unordered_map`) fournit l'équivalent C++ des dictionnaires de Python : il permet de stocker l'association d'une valeur à une autre valeur. Une documentation restreinte est disponible dans le polycopié de cours et la documentation générale avec maints exemples est disponible à l'adresse

https://en.cppreference.com/w/cpp/container/map

Question 4.11. Le fichier `declaration.txt` contient le préambule de la déclaration des droits de l'homme en français, sans accent ni ponctuation ni majuscule. Écrire un programme `texte_analyse.cpp` qui fait les choses suivantes :

— ouvre le fichier `declaration.txt` en lecture,
— déclare un objet `S` de type `std::map<std::string,unsigned>` vide,
— lit le texte et, à chaque mot, incrémente `S` afin qu'à la fin `S[mot]` indique le nombre de fois que `mot` apparaît dans le texte.

Dans toutes les question suivent de cette section sauf la dernière, on complètera ce programme. Les questions avec une (*) sont plus difficiles

Question 4.12. Écrire dans un fichier `stats.dat` chaque mot avec sa fréquence en ordre lexicographique.

Question 4.13. (bonus) Écrire dans un fichier `stats2.dat` chaque mot avec sa fréquence par ordre décroissant de fréquence.

Question 4.14. Répondre numériquement aux questions suivantes :

1. Combien y a-t-il de mots différents ?
2. Combien y a-t-il de mots différents de plus de 7 lettres ou plus ?
3. Quel est le mot le plus fréquent et combien de fois apparaît-il ? Donnez-en un. Bonus : donnez les tous.
4. Combien y a-t-il de lettres au total dans la déclaration ?
5. (*) Combien y a-t-il de mots avec exactement deux voyelles ?
6. (*) Quels mots contiennent le plus de fois la lettre 'e' ?
7. (*) Quelle est la corrélation empirique entre la longueur d'un mot et sa fréquence ?

Question 4.15. (*) Afficher tous les mots qui ont plus de 12 lettres, apparaissent au moins 13 fois, ne contiennent pas la lettre "e" et ne terminent pas par la lettre "s".

Question 4.16. (bonus) Écrire une fonction

```
std::set<std::string> filter(
2       const std::map<string,unsigned> & M,
       char first_letter, unsigned k);
```

qui construit l'ensemble des mots qui commencent par `first_letter` et apparaissent au moins `k` fois.

Application : écrire dans un fichier `h2.dat` l'ensemble des mots qui commencent par "h" et apparaissent au moins 2 fois.

Un retour sur la première section de cette feuille.

Question 4.17. (bonus+) Reprendre la marche aléatoire de la section 4.1 et, à l'aide de `std::map<int,unsigned>` , réaliser l'histogramme de la position finale de la marche (S_n) pour $n = 10000$ et un grand nombre de réalisations indépendantes ($M = 10^7$) de la marche et l'écrire dans des fichiers Visualiser avec l'outil de votre choix (gnuplot, matplotlib, etc). Quel théorème de probabilité cela illustre-t-il ?

4.4 Exercice : les retours multiples avec `<tuple>`

L'une des grandes frustrations du langage C et du langage C++ jusqu'au standard C++03 est l'impossibilité pour une fonction de renvoyer plusieurs objets. Pour contrer cela, il fallait soit renvoyer des structures définies *ad hoc*, soit passer les variables de sorties en argument via des pointeurs ou des références (ce qui rend la lecture des prototypes difficile sans une documentation correcte). L'introduction des `std::tuple` dans la bibliothèque `<tuple>` à partir du standard C++11 permet de contourner cela.

Mode d'emploi : un objet déclaré selon

```
std::tuple<double, int, std::string> X;
```

contient trois objets auquel on accède en lecture comme en écriture par `std::get<0>(U)` , `std::get<1>(U)` et `std::get<2>(U)` . On a droit aux opérations suivantes :

```
double x;
int n;
std::string s;
...
auto X=std::make_tuple(x,n,s); // définition de X à partir de x, n, s
...
std::tie(x,n,s)=X; // et réciproquement remplissage de x,n,s à partir de X
...
auto [y,m,ss]=X; // définition et initialisation de y,m,ss à partir de X
                 // (uniquement à partir du standard c++17)
```

Question 4.18. Reprendre l'exemple de la section précédente avec le programme intitulé `texte_analyse.cpp` et le compléter en écrivant une fonction :

```
std::tuple< double, unsigned, std::vector<std::string>,  >
basic_statistics(const std::map<std::string,unsigned> & M);
```

qui renvoie simultanément le nombre moyen de lettres par mots (pondéré par la fréquence des mots), le nombre de lettres du mot le plus long et la liste des mots les plus longs.

La tester ensuite et afficher le résultat.

T.P. 5

Templates et méthodes de Monte-Carlo

Les *méthodes de Monte-Carlo* regroupent un ensemble de techniques et d'algorithmes permettant d'approcher la valeur numérique de certaines intégrales en simulant un grand nombre de variables aléatoires indépendantes et identiquement distribuées. L'objectif de ce TP est de comprendre le principe de ces algorithmes de Monte Carlo et d'utiliser les templates pour en faire un outil adaptable à un maximum de modèles possibles.

5.1 La fonction génerique `MonteCarlo`

5.1.1 Le principe de l'algorithme...

Soit $(\Omega, \mathcal{F}, \mathbb{P})$ un espace de probabilité, E et F deux espace mesurable, $X : \Omega \to E$ une variable aléatoire telle que $\mathbb{E}(|X|) < \infty$, $f : E \to F$ une fonction mesurable. On suppose que $X_1, ..., X_n$ sont n v.a. indépendantes et de même loi que X. On définit le k-ème *moment empirique* de $f(X)$

$$\widehat{m}_n^k = \frac{1}{n} \sum_{i=1}^{n} f(X_i)^k.$$

Si $f(X)^k$ est intégrable, la loi forte des grands nombres dit que

$$\widehat{m}_n^k \xrightarrow[n\to\infty]{p.s., L^1} \mathbb{E}(f(X)^k) \tag{5.1}$$

5.1.2 ... Et sa mise en pratique

On se propose d'implémenter une méthode de Monte-Carlo permettant d'approcher $\mathbb{E}(f(X))$ avec l'équation (5.1) sous forme de template de fonction

```
template <class Statistique, class RandomVariable, class Measurement,
class RNG>
void MonteCarlo(Statistique & res, RandomVariable & X,
      const Measurement & f, RNG & G, long unsigned int n);
```

Les paramètres du template sont :

1. La classe `Statistique` qui correspond à l'estimateur que l'on veut calculer à l'aide de la simulation, par exemple la moyenne empirique ;
2. La classe `RandomVariable` qui correspond à la loi des variables aléatoires que l'on simule (par exemple `std::uniform_real_distribution<double>`) ;
3. La classe `Measurement` qui correspond à l'ensemble des fonctions f possibles ;
4. La classe `RNG` qui donne le type du générateur utilisé (dans tout le TP on se restreindra au type `std::mt19937`).

Pour résumer, `MonteCarlo(res,X,f,G,n)` stocke dans `res` le résultat du calcul

$$\frac{1}{n}\sum_{k=1}^{n} f(X_k) \tag{5.2}$$

où $f : E \to F$ est une fonction mesurable dont le type d'argument correspond au type de sortie des X_k, qui sont des v.a. iid d'une loi déterminée par la classe `RandomVariable` et qui sont simulées à l'aide du générateur aléatoire G.

On part du principe que :

— la classe `RandomVariable` possède un template de méthode

```
template<class RNG> TYPE RandomVariable::operator()(RNG & G)
```

qui renvoie une simulation de v.a. X_k [1].

— la fonction `f` possède un opérateur `()` appelable sur le type de retour de l'opérateur précédent de `RandomVariable` .

— la classe `Statistique` possède un opérateur `+=` qui permet d'incorporer une nouvelle valeur dans les statistiques, ainsi qu'un opérateur de normalisation `/=` par un `double` .

Question 5.1. Dans un fichier `"monte_carlo.hpp"` , déclarer le template de la fonction `MonteCarlo` et écrire le code correspondant : le code de `MonteCarlo(res,X,f,G,n)` doit contenir la simulation de `n` variables aléatoires de loi donnée par `X` à l'aide du générateur `G` , applique la fonction `f` à ces variables aléatoires, et met à jour `res` selon la formule (5.2). *Attention :* il est inutile de stocker les `n` valeurs des variables (et c'est souvent impossible en pratique).

5.2 Exemples d'applications

5.2.1 Approximation de π

Le premier exemple d'algorithme de Monte Carlo que l'on va implémenter consiste à estimer la valeur de π. On tire des points au hasard $(x, y) \in [0, 1]^2$ (selon la loi uniforme), et on compte la proportion de ces points tombant dans le disque unité $\{(x, y) : x^2+y^2 \leq 1\}$. Cette proportion converge vers le rapport des aires, soit $\frac{\pi}{4}$, lorsque le nombre de points tend vers l'infini. Pour modéliser cette expérience, on introduit la classe de variables aléatoires des variables de Bernoulli décrivant si un point uniforme du carré unité tombe dans le quart de cercle unité.

1. C'est notamment le cas de toutes les distributions de probabilités disponibles dans la bibliothèque `<random>` .

```
class SquareInDisk {
    private:
        std::uniform_real_distribution<double> U;
};
```

Question 5.2. Créer un fichier `"pi.hpp"` et recopier le code de `SquareInDisk` en prenant soin d'ajouter un constructeur par défaut qui initialise `U` à $(0, 1)$.

Question 5.3. Ajouter un template de méthode de la classe `SquareInDisk`

```
template<class RNG> double operator()(RNG & G)
```

qui simule un couple (x, y) de variables aléatoires indépendantes de loi `U`, et renvoie $\mathbf{1}_{\{x^2+y^2\leq 1\}}$.

Question 5.4. Créer un fichier `"simulations.cpp"`, et déclarer dans le code `main()` un élément `SquareInDisk A`, ainsi qu'un élément `double pi_approx`. Appliquer la fonction `MonteCarlo` à `pi_approx`, `A` et à la fonction $x \mapsto 4x$ pour `n = 1000`, `10000` et `100000` afin d'estimer la valeur de π.

Calcul simultané de la variance empirique : On rappelle que la variance empirique $\widehat{Var}(Z)$ d'une variable aléatoire est définie par

$$\widehat{\mathrm{Esp}}(Z) = \frac{1}{n}\sum_{k=1}^{n} Z_k$$

$$\widehat{\mathrm{Var}}(Z) = \frac{1}{n}\sum_{k=1}^{n}\left(Z_k - \widehat{\mathrm{Esp}}(Z)\right)^2 = \left(\frac{1}{n}\sum_{k=1}^{n} Z_k^2\right) - \widehat{\mathrm{Esp}}(Z)^2 = \widehat{\mathrm{Esp}}(Z^2) - \widehat{\mathrm{Esp}}(Z)^2$$

En changeant la classe de l'argument `res` de sorte à surcharger l'opérateur `+=`, il est possible de calculer la moyenne empirique et la variance empirique simultanément.

Question 5.5. Dans le fichier `monte_carlo.hpp`, créer une classe

```
class DoubleMeanVar{
protected:
    double m;//Moyenne
    double v;//utile pour la Variance
};
```

et la munir d'un constructeur `DoubleMeanVar(double x=0.)` qui initialise `m` à `x` et `v` à zéro.

Question 5.6. Surcharger l'opérateur `operator+=` de sorte que lorsqu'on a `DoubleMeanVar MV` et `double x`, l'opération `MV+=x` ajoute `x` à `m` et `x*x` à `v`. De même, surcharger l'opérateur `operator/=` de sorte qu'il permette de normaliser simultanément `m` et `v`.

Question 5.7. Écrire un accesseur de `m` et `v` adapté à l'utilisation de `MonteCarlo`. **Attention** : le lecteur observateur aura remarqué que si l'on applique les opérateurs `+=` et `/=` dans la formule (5.2) cela ne donne pas la variance empirique : il faut alors corriger la valeur de `v` dans l'accesseur en la modifiant de manière appropriée.

Question 5.8. Lorsqu'on veut connaître la précision de l'approximation d'une moyenne empirique, on peut en déterminer son intervalle de confiance. Lorsque la population suit une loi normale (ce qui est le cas lorsque la population est de taille suffisante et ses individus indépendants, par le théorème central limite), on a l'encadrement asymptotique suivant :

$$\widehat{\mathrm{Esp}}(X) - k\sqrt{\frac{\widehat{\mathrm{Var}}(X)}{n}} \leq \mathbb{E}[X] \leq \widehat{\mathrm{Esp}}(X) + k\sqrt{\frac{\widehat{\mathrm{Var}}(X)}{n}},$$

avec k le quantile qui détermine le niveau de confiance. Par exemple, pour un degré de confiance de 95%, on a $k = 1.96$. Écrire sur le terminal l'intervalle de confiance à 95% de la valeur de π donnée par la simulation de la question 4 en utilisant la classe `DoubleMeanVar` de la question 5.

5.2.2 Approximation d'intégrales

Question 5.9. Estimer, à l'aide de `MonteCarlo`, les intégrales suivantes :

$$\int_0^1 \ln(1+x^2)\mathrm{d}x$$
$$\int_{\mathbb{R}_+\times[0,1]} \ln(1+xy)e^{-x}\mathrm{d}x\mathrm{d}y.$$

Pour la seconde intégrale, on notera que

$$\int_{\mathbb{R}_+\times[0,1]} f(x,y)e^{-x}\mathrm{d}x\mathrm{d}y = \mathbb{E}[f(X,Y)]$$

où X suit une loi exponentielle de paramètre 1 et Y une loi uniforme sur $[0,1]$. Pour éviter l'usage de classes, on pourra introduire la λ-fonction

```
auto CoupleXY = [&](std::mt19937 & G) {return std::make_pair(X(G),Y(G));};
```

et une autre λ-fonction `Function_to_evaluate` qui prend une `std::pair<double,double>` en argument et applique la fonction à intégrer pour renvoyer un `double`.

5.2.3 L'histogramme, ou l'art d'approcher une densité de probabilité

Dans les TPs précédents, on a déja abordé les histogrammes de façon ponctuelle, l'objectif ici est de systématiser le processus en utilisant la fonction `MonteCarlo`. Si l'on simule un ensemble de variables aléatoires indépendantes et de même loi $\mathcal{L}$ de densité ρ, alors l'histogramme de cette population est une approximation graphique de la courbe de ρ sur un intervalle donné $[a,b]$. Cela fonctionne comme suit :

1. On découpe l'intervalle $[a,b]$ en p sous-intervalles (ou boîtes) de même largeur $\frac{b-a}{p}$;

2. On effectue la simulation d'un nombre n de variables aléatoires indépendantes et de même loi ;
3. Pour chaque simulation, si sa valeur tombe dans la boîte i, on incrémente de 1 la i-ème coordonnée de l'histogramme (vu comme un vecteur de taille p).
4. Une fois toutes les simulations terminées, on renormalise les coordonnées du vecteur en les divisant par le nombre total de points de l'échantillon.

Question 5.10. Dans le fichier `monte_carlo.hpp`, déclarer la classe suivante :

```
class Histogramme{
protected:
    std::vector<double> echantillon;
    unsigned int nb_boxes;//nombre de boîtes
    double lbound;//borne inférieure de l'intervalle
    double ubound;//borne supérieure de l'intervalle
    double box_width;//largeur des boîtes
};
```

et écrire un constructeur

```
Histogramme(double min_intervalle, double max_intervalle,unsigned int n);
```

qui initialise un histogramme à n boîtes sur [`min_intervalle` , `max_intervalle`].

Question 5.11. Surcharger les opérateurs `+=` et `/=` pour que `H+=x` incrémente `H.echantillon[i]` si `x` est dans l'intervalle `i`, et `H/=n` divise toutes les entrées de `H.echantillon` par `n`.

Question 5.12. Surcharger l'opérateur `<<` pour qu'il affiche l'histogramme sous la forme

```
a_0     echantillon[0]
a_1     echantillon[1]
...
a_(nb_boxes-1) echantillon[nb_boxes-1]
```

où `a_i` est la borne inférieure de la boîte `i`, *i.e.* `a_0=a`, `a_1 = a+(b-a)/p`, etc.

Question 5.13. En utilisant la classe `Histogramme` et la fonction `MonteCarlo` et sans utiliser la moindre boucle supplémentaire, construire un histogramme à 50 boîtes de la loi normale standard $\mathcal{N}(0,1)$ sur $[-3,3]$ à partir d'un échantillon de 100000 simulations. On rappelle que la loi normale de la bibliothèque `random` s'écrit `std::normal_distribution<double>`.

Pour afficher l'histogramme sous `gnuplot`, écrire par exemple :

`plot "histogramme.dat" using 1:2 with boxes`, ce qui doit donner la figure 5.1.

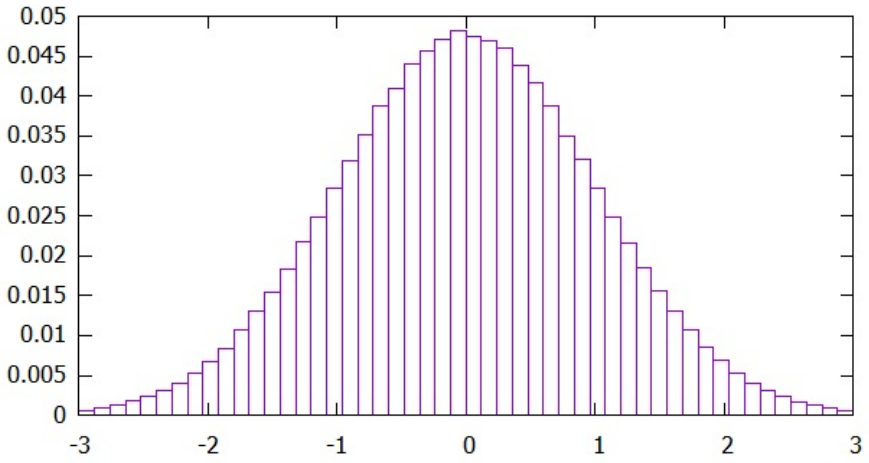

FIGURE 5.1 – Affichage de l'histogramme de la loi $\mathcal{N}(0,1)$ sous `gnuplot`

Application à une loi un peu moins connue. Si $X_1, ..., X_k$ sont k variables aléatoires gaussiennes centrées réduites indépendantes, alors $Y = X_1^2 + ... + X_k^2$ suit la loi du χ_2 à k degrés de liberté.

Question 5.14. Dans un fichier `chi_deux.hpp` , créer un template de classe

```
template<class REAL, int k>
class Chi2_distribution
{
        private:
                std::normal_distribution<REAL> N;
        public :
                Chi2_distribution();
                template <class RNG> REAL operator()(RNG & G);
};
```

où le constructeur initalise et `N` à $\mathcal{N}(0,1)$, l'opérateur `operator()` simule $X_1, ..., X_k$ et renvoie $Y = X_1^2 + ... + X_k^2$.

Question 5.15. À l'aide de `MonteCarlo` , `Histogramme` et `Chi2_distribution` , afficher simultanément sur `gnuplot` les histogrammes des lois $\chi_2(3)$ et $\chi_2(6)$ sur $[0, 10]$. Cela doit donner la figure 5.2.

5.3 Généralisation à la méthode MCMC (Monte Carlo Markov Chain)

La méthode d'approximation d'une intégrale par la loi forte des grands nombres (5.1) qui utilise des v.a.i.i.d. peut s'étendre à l'approximation de mesures invariantes de chaînes de Markov par le théorème ergodique (cf. le cours de probabilités approfondies). Soit $(X_n)_{n\geq 0}$ une chaîne de Markov irréductible, récurrente positive sur un ensemble E, avec

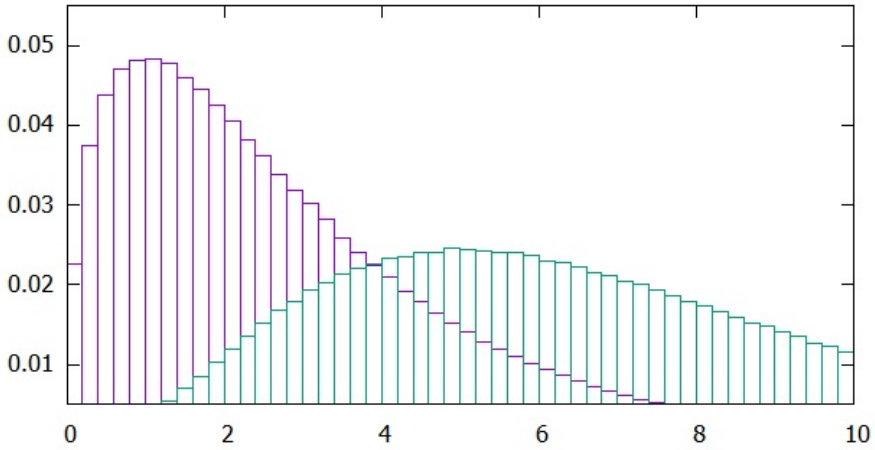

FIGURE 5.2 – Affichage de l'histogramme de la loi χ_2 avec paramètres respectifs 3 et 6

probabilité invariante π. Alors, pour toute fonction $f : E \to \mathbb{R}$ mesurable et π-intégrable,

$$\frac{1}{n}\sum_{k=1}^{n} f(X_k) \xrightarrow[n\to\infty]{p.s., L^1} \int_E f(x)d\pi(x) \tag{5.3}$$

La conclusion numérique est donc qu'on peut encore utiliser la fonction `MonteCarlo` pour des classes `RandomVariable` qui ne génèrent pas seulement des v.a.i.i.d mais également une trajectoire d'une chaîne de Markov à chaque appel du template de méthode `operator()(RNG & G)` .

La chaîne de Markov à deux états $E = \{1, 2\}$. On considère la chaîne de Markov suivante : à chaque pas de temps, si $X_n = 1$, alors $X_{n+1} = 2$ avec probabilité a et $X_{n+1} = 1$ avec probabilité $1 - a$; si $X_n = 2$, alors $X_{n+1} = 1$ avec probabilité b et $X_{n+1} = 2$ avec probabilité $1 - b$. On considère ainsi la classe suivante :

```
class Markov2states {
protected:
   int x;
   std::bernoulli_distribution Ua;
   std::bernoulli_distribution Ub;
public:
   Markov2states(int x0=1,double a0=0.5,double b0=0.5);
   ....
};
```

Question 5.16. Coder la classe entièrement. Le template de méthode pour `operator()` met à jour le champ `x` selon le modèle mathématique précédent.

Question 5.17. La mesure invariante est donnée par $\pi(1) = b/(a+b)$ et $\pi(2) = a/(a+b)$. Écrire une classe `Stat2states` qui compte le nombre de visites des états 1 et 2 selon le modèle :

```
class Stat2states {
protected:
        long unsigned visit1;
        long unsigned visit2;
public:
        ....
};
```

et vérifier le résultat du théorème ergodique pour cette chaîne de Markov.

Modèle d'Ising à une dimension. Soit $N \geq 1$, $\beta > 0$ et $h \in \mathbb{R}$. On considère l'ensemble fini $E = \{-1, 1\}^N$ muni de la probabilité :

$$\pi(x_0, \ldots, x_{N-1}) = \frac{1}{Z_N(h, \beta)} \exp\left(\beta \sum_{k=0}^{N-2} x_k x_{k+1} + h \sum_{k=0}^{N-1} x_k\right) \tag{5.4}$$

Étant donné la forme compliquée de π et le fait que la constante $Z_N(h, \beta)$ soit difficile à calculer, il n'est pas possible de générer facilement des réalisations i.i.d. de v.a. de loi π. En revanche, il existe une chaîne de Markov très simple dont π est la mesure invariante ! Elle est définie de la manière suivante :

- à chaque pas de temps, on choisit l'une des N variables x_k uniformément.
- on calcule $p = \min(1, \exp(-2\beta(x_{k-1} + x_{k+1})x_k - 2hx_k)))$ (si $k = 0$ ou $k = N - 1$, le terme x_{k-1} ou x_{k+1} non défini est tout simplement absent)
- avec probabilité p, x_k devient $-x_k$ et, avec probabilité $1 - p$, x_k ne change pas.

Question 5.18. Écrire une classe `Ising1D` qui implémente ce modèle, de telle sorte à pouvoir être utilisée ensuite dans `MonteCarlo` .

Question 5.19. On souhaite estimer la valeur moyenne de x_{500} pour $N = 1000$ lorsque β et h varient. Écrire un programme qui réalise cette estimation en utilisant tout le travail déjà fournit en prenant un nombre d'itérations grand par rapport à N.

5.4 Bonus : retour sur des TP antérieurs

En programmation, le *backtracking* désigne une méthode qui consiste à revenir sur une décision effectuée précédemment pour sortir d'un blocage, par exemple par un principe d'essai-erreur ; ici le titre veut simplement dire que l'on reprend les TPs précédents avec des outils plus évolués pour répondre plus efficacement aux questions !

Question 5.20. Traiter les questions du TP 2 sur le *GOE* en créant une classe `GOE` et en se servant de la fonction `MonteCarlo` et de la classe `Histogramme` .

Question 5.21. Refaire la partie du TP 3 sur la simulation de permutations aléatoires et le nombre de dérangements à l'aide de `MonteCarlo` en écrivant une classe `random_permutation` .

T.P. 6

Templates et polynomes

L'objectif de ce TP est de travailler sur différents aspects des modèles de classes et de fonctions. On se propose d'écrire une classe représentant la famille des polynômes à coefficients dans un anneau abstrait, et de coder plusieurs opérations sur ces polynômes.

6.1 Les polynômes

Soit $(\mathbf{A}, +, \times)$ un anneau commutatif.

On appelle *polynôme* à une variable à coefficients dans l'anneau $\mathbf{A}$ toute suite infinie $A = (a_0, a_1, \ldots, a_n, \ldots)$ d'éléments de $\mathbf{A}$ tous nuls à partir d'un certain rang :

$$\exists N \in \mathbb{N}, \quad \forall n \geq N \quad a_n = 0.$$

Tout polynôme $A = (a_0, a_1, \ldots, a_n, 0, \ldots)$ peut s'écrire :

$$A = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n = \sum_{i=0}^{n} a_i x^i.$$

Le *degré* d'un polynôme A non nul est le *plus grand* entier n tel que le coefficient a_n de x^n dans A est non nul. Ce coefficient a_n s'appelle alors le coefficient *dominant*.

L'ensemble des polynômes à une variable à coefficients dans $\mathbf{A}$ est noté $\mathbf{A}[X]$.

6.2 Premiers pas : les éléments neutres

Nous aurons besoin de vérifier régulièrement si certains coefficients des polynômes sont égaux à 0 ou à 1. Pour la majorité des classes utilisées, nous supposerons que la comparaison directe avec les entiers 0 et 1 est possible.

Question 6.1. Écrire dans le fichier `polynome.hpp` deux modèles de fonction (*templates*) `is_zero` et `is_one` qui testent si le paramètre `a` de type générique `T` est égal respectivement à 0 ou à 1.

```
template<typename T>
bool is_zero(T a){
        ...
}
```

```
6  template<typename T>
   bool is_one(T a){
8          ...
   }
```

Pour certains types flottants, à cause des erreurs d'arrondis, il se pourrait qu'après plusieurs opérations, un coefficient soit non pas égal à 0 comme il devrait, mais de l'ordre de 10^{-18}. Le test strict alors échouerait. Une manière de remédier à ce problème est d'écrire une version spécifique de ces templates pour les types concernés.

Question 6.2. Écrire une version spécifique de `is_zero` et `is_one` pour le type `double` .

Question 6.3. Écrire une version spécifique (template) de `is_zero` et `is_one` pour les types `std::complex<T>` .

6.3 La classe `Polynome<T>`

Un polynôme avec des coefficients dans `T` est représenté par un entier `n` représentant le degré, et un vecteur `coeff` de longueur `n+1` permettant de stocker les coefficients. L'élément `coeff[i]` est le coefficient de x^i si i est inférieur au degré. Par convention un degré strictement négatif (avec une préférence pour la valeur -1 pour continuer la correspondance entre la valeur de `n` et la taille de `coeff`) correspondra au polynôme nul[1].

6.3.1 Champs, constructeurs accesseurs et mutateurs

Question 6.4. Définir dans le fichier `polynome.hpp` le modèle de classe `Polynome<T>` et déclarer les champs privés `n` et `coeff` .

6.3.2 Constructeurs

Question 6.5. Écrire un constructeur par défaut qui construit le polynôme nul. Écrire un constructeur à deux paramètres qui prend en argument un coefficient `a` en référence constante et un entier `m` pour représenter le monôme ax^m. Écrire un dernier constructeur qui prend en argument un vecteur d'éléments de type `T` et produit le polynôme avec ces coefficients (on supposera que le dernier élément du vecteur est non nul).

Question 6.6. Définir une méthode accesseur `degre` renvoyant le degré du polynôme représenté par l'objet courant.

Question 6.7. Définir un accesseur à partir de l'opérateur `operator[]` pour accéder aux coefficients du polynôme. Si l'entier passé en argument est négatif ou strictement plus grand que le degré, on fera bien attention à ne pas provoquer une erreur d'accès mémoire.

1. En mathématique, la convention est que le degré du polynôme nul est $-\infty$.

6.3.3 Affichage

On veut écrire une fonction `operator<<` qui permet d'afficher un polynôme dans un flux de sortie. Cette fonction globale (template) doit être amie avec la classe `Polynome<T>` pour pouvoir accéder aux champs privés de cette classe [2]. Cette fonction d'affichage délègue une partie du travail à une fonction auxiliaire `affiche_monome` qui, comme son nom l'indique, affiche un monôme.

Question 6.8. Définir une fonction template `affiche_monome`, prenant trois arguments :
- un flux de sortie en référence `s`,
- un coefficient de type `T` (en référence constante) `a`,
- un entier `n`,

et qui affiche dans `s` le monôme de la même façon que sur les exemples suivants : `7*x^2`, `-5*x`, `42`. Rien n'est affiché si le coefficient `a` est nul.

Question 6.9. On souhaite modifier l'affichage de cette fonction en fonction du type `T`. Si ce type représente des complexes, on préférerait que la variable soit `z` plutôt que `x`. On pourrait écrire comme pour `is_zero` précédemment une spécialisation particulière de cette fonction pour ces types. On propose d'utiliser un autre mécanisme, introduit avec le standard C++ 11. La classe template `std::is_same<typename A, typename B>` définie dans l'entête `<type_traits>` a un champ booléen `value` qui vaut `true` (resp. `false`) si les types `A` et `B` sont les mêmes (resp. différents). Consultez la page de la référence de C++ consacrée à cette commande :

https://www.cplusplus.com/reference/type_traits/is_same/

Rajouter une condition dans la fonction `affiche_monome` avec ce mécanisme pour adapter la variable au type des coefficients.

On passe maintenant à la fonction d'affichage à proprement parler pour laquelle on montre plusieurs implémentations possibles.

Question 6.10. *Version extravertie.* Déclarer dans la classe `Polynome<T>` une relation d'amitié avec une fonction d'affichage template

```
template <typename U>
std::ostream & operator<<(std::ostream &, const Polynome<U> &)
```

Notez bien que les paramètres de templates de `Polynome` est différent dans le nom de la classe et le paramètre de ce `std::operator<<`. Définir cette fonction à l'extérieur de la classe. C'est la version la plus simple à écrire, mais qui ne respecte pas le principe d'encapsulation : une version détournée pourrait en principe accéder (en lecture et écriture !) aux champs privés de toutes les autres variantes de `Polynome<T>`.

On souhaite maintenant écrire une version plus stricte de cette fonction template amie : seule la version avec le paramètre `<T>` sera amie avec la classe `Polynome<T>`.

2. Grâce aux accesseurs écrits précédemment, on aurait pu faire une fonction d'affichage n'ayant pas besoin des champs privés. La question ici sert de prétexte pour travailler le point délicat de la relation d'amitié pour des classes templates.

Question 6.11. *Version intravertie.* Déclarer une relation d'amitié avec la fonction globale de même paramètre :

```
template <typename T>
class Polynome {
[...]
    friend std::ostream& operator<< <T>(std::ostream& o, const Polynome<T>&);
[...]
};
```

et écrire la définition à l'extérieur de la classe. Notez que pour que la compilation réussisse, il faut déclarer en amont, c'est à dire avant la définition de la classe `Polynome<T>` (on parle en anglais de *forward declaration*) la fonction globale `std::ostream& operator<<`, ce qui implique aussi de devoir déclarer l'existence du modèle de classe juste avant (donc deux déclarations en amont).

```
template <typename T> class Polynome;
```

Remarquons que si nous avions pu/voulu écrire le code de la fonction globale `inline` dans la classe, il n'y aurait pas eu besoin de :

— indiquer le paramètre `<T>` dans le nom de la fonction ;
— faire les déclarations en amont.

Pour plus de détails, consulter la réponse à cette question sur stackoverflow.

https://stackoverflow.com/questions/4660123/overloading-friend-operator-for-template-class/4661372#4661372

Question 6.12. Écrire dans un fichier `main.cpp` le code suivant et vérifier ainsi le bon fonctionnement de la classe `Polynome` et de méthodes jusqu'à présent définies en compilant et en exécutant la première partie de ce code (en mettant en commentaire la partie que l'on ne souhaite pas encore exécuter).

```
#include "polynome.hpp"
#include <iostream>

int main(){
//Première partie
    Polynome<double> q;
    std::cout << "Degre de q : " << q.degre() << endl;

    vector<int> v1{6, 3, 0, 1, 5};
    Polynome<int> p1(v1);

    vector<int> v2{1,0,1};
    Polynome<int> p2(v2);

    std::complex<double> a (2.0,1.0);
    std::complex<double> b (0.0,1.0);
```

```
    vector<std::complex<double>> vc{a,b};

    Polynome<std::complex<double>> pc(vc);

    std::complex<int> one = 1;
    std::cout << "Is one one : " <<  is_one(one) << endl;

    std::cout << is_zero(a) << endl;
    affiche_monome(std::cout, a, 3);
    std::cout << std::endl;

    std::cout << "p1 : " << p1 << std::endl;

//Deuxième partie
    // Somme, différence, produit
    Polynome<int> sum = p1+p2;
    Polynome<int> diff = p1-p2;
    Polynome<int> prod = p1*p2;
    std::cout << "Somme : " << sum << std::endl;
    std::cout << "Différence : " << diff << std::endl;
    std::cout << "Produit : " << prod << std::endl;

    // Division et reste
    Polynome<int> div = p1/p2;
    Polynome<int> reste = p1%p2;
    std::cout << "Quotient : " << div << std::endl;
    std::cout << "Reste : " << reste << std::endl;

    // Evaluation en un point
    std::cout << "p1(2) : "  <<  p1(2) << std::endl;

    return 0;
}
```

6.3.4 Opérations arithmétiques

Étant donnés deux polynômes à coefficients dans un même anneau (de même type `Polynome<T>`) , nous désirons en un premier temps calculer leur somme, leur différence et leur produit.

Question 6.13. Ajouter à la classe `Polynome` une méthode privée `extend` qui prend en argument un entier `m` et, si `m` est plus grand que le degré du polynôme, renvoie un « polynôme » dont le vecteur de coefficients est obtenu en copiant celui de l'objet courant, et complété avec des zéros pour avoir la taille `m+1` (c'est à dire avoir virtuellement degré `m`), sinon, il renvoie juste l'objet courant. Attention : le dernier coefficient de ce polynôme est potentiellement nul, et dans ce cas, n'est pas le coefficient dominant. Les propriétés supposées sur les objets pour garantir la cohérence de la classe ne sont donc

plus satisfaites, et on ne peut pas utiliser les polynômes ainsi créés de manière sûre sans précaution. C'est pour cela que cette méthode doit être privée.

Question 6.14. Ajouter à la classe `Polynome` une méthode privée `adjust` qui modifie un polynôme de telle sorte que son coefficient dominant ne soit pas nul. Le degré du polynôme ansi que son vecteur de coefficients sont ainsi ajustés.

Question 6.15. Déclarer et définir, en utilisant les méthodes `extend` et `adjust`, les opérateurs de somme et de différence, écrites comme des fonctions globales.

```
template <typename T>
Polynome<T> operator+(const Polynome<T> &, const Polynome<T> &);
template <typename T>
Polynome<T> operator-(const Polynome<T> &, const Polynome<T> &);
```

Remarque : comme ces opérateurs vont faire appel aux méthodes privées `extend` et `adjust`, il faudra les déclarer amis de la classe `Polynome`.

Soient $A = (a_0, a_1, \ldots, a_n, 0, \ldots)$ un polynôme de degré n et $B = (b_0, b_1, \ldots, b_m, 0, \ldots)$ un polynôme de degré m, à coefficients dans le même anneau **A**. Le produit $C = AB = (c_0, c_1, \ldots, c_l, 0, \ldots)$ est un polynôme à coefficients dans **A** de degré $l = n + m$ tel que pour tout $k \in \{0, \ldots, l\}$,

$$c_k = \sum_{\substack{i=0,\ldots,n \\ j=0,\ldots,m \\ i+j=k}} a_i b_j$$

Question 6.16. Déclarer et définir la fonction globale opérateur de produit.

```
template <typename T>
Polynome<T> operator*(const Polynome<T> &, const Polynome<T> &);
```

Nous nous intéressons maintenant à la division euclidienne de deux polynômes. Si A et B sont deux polynômes à coefficients de type `T` et que B, qui est supposé non nul, a un coefficient dominant inversible dans `T` (si `T` est un type entier comme `int`, cela signifie qu'il est egal à 1 ou -1), alors il existe un unique couple de polynômes (Q, R) tels que $A = BQ + R$ et $\deg(R) < \deg(B)$. Q est alors appelé le *quotient* et R le *reste* de la division euclidienne de A par B. Par analogie avec les calculs pour les entiers, nous souhaitons écrire les opérateurs `operator/` et `operator %` renvoyant respectivement le quotient et le reste de la division du premier paramètre par le second.

Question 6.17. Écrire près de `is_zero` et `is_one` une fonction template `is_invertible` qui renvoie true si l'argument x est inversible, c'est à dire -1 ou 1 si x est de type entier, et juste non nul en général. On pourra utiliser `std::is_integral<decltype(x)>::value` qui renvoie vrai si et seulement si le type de `x` est entier (`int`, `long`, `char`, etc.)

https://www.cplusplus.com/reference/type_traits/is_integral/

Question 6.18. Écrire une fonction d'aide `euclid_div` qui prend deux polynômes A et B en argument comme référence constante et qui renvoie sous forme de paire les polynômes Q et R. Q et R sont calculés itérativement, en posant $R_0 = A$. Ã chaque itération, on calcule le monôme Q_i pour que $R_{i+1} = R_i - BQ_i$ soit de degré strictement inférieur à R_i en éliminant les termes dominants. On arrête les itérations à l'étape n telle que $\deg R_n < \deg B$. On a alors $R = R_n$ et $Q = Q_0 + \cdots + Q_{n-1}$. La fonction affiche un avertissement et renvoie la paire $(0, A)$ si le coefficient de B n'est pas inversible.

Question 6.19. Écrire les deux fonctions globales `operator/` et `operator %` qui renvoient respectivement la première et la deuxième composante de la paire renvoyée par `euclid_div` .

6.3.5 Évaluation en un point

Soit $A = (a_0, a_1, \ldots, a_n, 0, \ldots) \in \mathbf{A}[X]$ un polynôme de degré n et $x_0 \in \mathbf{A}$. Nous souhaitons évaluer le polynôme A en x_0, ce qui correspond à calculer

$$A(x_0) = \sum_{i=0}^{n} a_i x_0^i.$$

Une approche extrêmement naïve consiste à calculer les puissances de x_0, multiplier chaque puissance par son coefficient et faire ensuite la somme. Le nombre de produits nécessaires à effectuer ce calcul est $n + (n-1) + \cdots + 2 + 1 = O(n^2)$.

La *méthode de Horner* permet de réduire considérablement cette complexité, en se ramenant à une complexité en $O(n)$, en effectuant le calcul comme suit :

$$A(x_0) = ((\cdots((a_n x_0 + a_{n-1})x_0 + a_{n-2})x_0 + \cdots)x_0 + a_1)x_0 + a_0.$$

Question 6.20. Écrire un opérateur parenthèse () qui permet d'évaluer un polynôme en un point, sous forme de méthode de la classe `Polynome` .

Remarque : une possible implémentation de la méthode de Horner peut s'écrire en faisant appel à la fonction `std::accumulate` de la bibliothèque `numeric` . Il faut alors parcourir le vecteur des coefficient à l'envers. On pourra faire appel aux `reverse_iterator` , auxquels on peut accéder via les méthodes `rbegin()` et `rend()` . Pour plus de détails sur les `reverse_iterator` , consulter la documentation en ligne :

https://www.cplusplus.com/reference/iterator/reverse_iterator/

Question 6.21. Vous avez peut-être écrit la fonction d'évaluation pour un argument x qui aurait le même type que les coefficients du polynôme. Or on pourrait vouloir évaluer un polynôme à coefficients entiers sur un réel, ou un polynôme à coefficients réels sur une matrice carrée à coefficients complexes. Ce qui compte, c'est que si `x` est de type `U` et les coefficients sont de type `T` , il y ait des fonctions pour multiplier des éléments de type `T` et de type `U` pour obtenir quelquechose de type `U` , ce qui est le cas dans les deux exemples ci-dessus. Modifier si besoin la fonction d'évaluation. Utilisez la bibliothèque Eigen utilisée au TP2 pour calculer `p1(M)` où $M = \begin{pmatrix} 4 & 1+i \\ -2 & \sqrt{3} \end{pmatrix}$.

6.4 Autres développements

Lorsque un polynôme est représenté par le vecteur de ses coefficients, il se peut que l'occupation de la mémoire ne soit pas optimale. Pour représenter le polynôme $x^{1000} + 1$, par exemple, nous allons utiliser un vecteur de 1001 cases, alors que l'information sur ce polynôme n'est contenue que dans la première et la dernière case du vecteur et dans son degré.

Une autre façon d'encoder un polynôme alors est de le représenter comme une somme de monômes.

Question 6.22. Écrire une court modèle de classe `Monome` , avec les constructeurs et accesseurs nécessaires.

```
template <typename T>
class Monome {
private:
    int n; // degree
    T coeff; //coefficient

public:
    Monome() : ... {}
    Monome(const T& a, int m=0): ... // le monome a*x^m

    // comparaison
    friend bool operator< <T>(const Monome<T>& u, const Monome<T>& v);
    ...
}
```

Question 6.23. Écrire le modèle de classe `Polynome`

```
template <typename T>
class Polynome {
private:
    std::list<Monome> monomes;
    ...
public:
    ...
```

et y ajouter les opérations de somme, différence, produit, division, ainsi que l'évaluation du polynôme en un point.