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

Programmation en C++ :

sujets des travaux pratiques

(version du 13 janvier 2021)

Damien Simon,
avec Cédric Boutillier et Thibaut Lemoine
Laboratoire de probabilités, statistique et modélisation,
Sorbonne Université

Année universitaire 2019–2020

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++`
- la bibliothèque `libstdc++5` (voire la version 6)
- la bibliothèque `libeigen3-dev` pour l'algèbre linéaire
- l'éditeur `geany`
- le logiciel `gnuplot` pour visualiser les données
- (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 Entrée/sortie

— 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 les sites suivants :

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

   ou bien

   https://fr.cppreference.com/w/cpp/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 :

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

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

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

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

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

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

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

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

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

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

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

2.1 Premiers pas : la fonction puissance

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

et elle repose sur le raisonnement suivant :

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

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

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

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

Question 2.2. Dans 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

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

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. 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 2.4. Écrire une fonction `puissance2` qui prend les mêmes types d'arguments d'entrée et de sortie que `puissance` mais fonctionne sur la récurrence ci-dessus.

Question 2.5. Test : comparer les temps de calcul pour la puissance 1000-ème [3] de la matrice B de taille 30×30 donnée dans le fichier `matrice.dat` [4] (disponible sur le site du cours) selon que l'on utilise `puissance` ou `puissance2` . Tester également l'effet de l'utilisation ou non de l'option de compilation `-O2` de `g++` .

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 :

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 `>>` .

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

Question 2.7. (À traiter plus tard dans le semestre) Fusionner les fonctions `puissance2` et `puissance_sparse` en un template de fonction `puissance<MatrixType>` compatible avec leurs deux types respectifs.

2.2 Résolution d'une EDP linéaire elliptique discrétisée sur le carré.

Nous souhaitons résoudre le problème d'analyse suivant sur le domaine $D = [0, 1]^2$:

$$\begin{cases} \Delta f(x) = u(x) & \text{pour } x \in]0,1[^2 \\ f(y) = f_0(y) & \text{pour } y \in \partial D \end{cases} \tag{2.1}$$

où u est une fonction $[0, 1]^2 \to \mathbb{R}$ fixée et f_0 est une fonction réelle sur le bord du domaine fixée. Δ désigne le laplacien et f est la fonction inconnue.

Numériquement, une manière de faire est de remplacer le carré par un quadrillage de pas $\epsilon = 1/(P + 1)$ avec P grand et de discrétiser le laplacien. Ainsi, on remplace une fonction réelle $u : [0, 1]^2 \to \mathbb{R}$ par son approximation $(\tilde{u}_{i,j})_{\substack{0\le i\le P+1,\\ 0\le j\le P+1}}$ définie par :

$$\tilde{u}_{i,j} = u(\epsilon i, \epsilon j) \tag{2.2}$$

Le laplacien est alors remplacé par le laplacien discret :

$$(\tilde{\Delta}\tilde{f})_{i,j} = \epsilon^{-2}\left(\tilde{f}_{i-1,j} + \tilde{f}_{i+1,j} + \tilde{f}_{i,j-1} + \tilde{f}_{i,j+1} - 4\tilde{f}_{i,j}\right) \tag{2.3}$$

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

Résoudre (2.1) de manière approchée revient donc à résoudre une équation linéaire sur le quadrillage en « inversant » la matrice du laplacien discret. Pour assurer l'inversibilité du laplacien discret, il faut se restreindre aux indices $1 \leq i, j \leq P$ et résoudre le problème suivant :

$$(\tilde{M}\tilde{f})_{i,j} = \epsilon^2 \tilde{u}_{i,j} - (Kf_0)_{i,j}, \qquad 1 \leq i, j \leq P \tag{2.4}$$

avec les définitions suivantes :

$$M_{(i,j),(k,l)} = \begin{cases} 1 & \text{si } (i,j) = (k \pm 1, l) \text{ et } 1 \leq i,j,k,l \leq P \\ 1 & \text{si } (i,j) = (k, l \pm 1) \text{ et } 1 \leq i,j,k,l \leq P \\ -4 & \text{si } (i,j) = (k,l) \text{ et } 1 \leq k,l \leq P \\ 0 & \text{sinon} \end{cases} \tag{2.5}$$

$$(Kf_0)_{i,j} = \begin{cases} f_0(i\epsilon, 0) & \text{si } j = 1 \\ f_0(i\epsilon, 1) & \text{si } j = P \\ f_0(0, j\epsilon) & \text{si } i = 1 \\ f_0(1, j\epsilon) & \text{si } i = P \\ 0 & \text{sinon} \end{cases} \tag{2.6}$$

Pour pouvoir utiliser une bibliothèque d'algèbre linéaire comme Eigen, il faut d'abord encoder une suite $(\tilde{u}_{i,j})_{1 \leq i,j \leq P}$ en un vecteur $(\tilde{U}_k)_{0 \leq k < P^2} \in \mathbb{R}^{(P^2)}$, indexé de 0 à $P^2 - 1$ sur lequel appliquer des matrices. Pour cela, nous utiliserons la correspondance suivante d'indices

$$(i,j) \leftrightarrow k = (j-1)P + (i-1) \tag{2.7}$$

et le coefficient $\tilde{u}_{i,j}$ sera ainsi la $(j-1)P + (i-1)$ coordonnée d'un vecteur U.

Question 2.8. Nous observons que la matrice M contient de nombreux zéros. Nous allons donc privilégier les matrices creuses. Écrire une fonction de prototype

```
Eigen::SparseMatrix<double> make_sparse_laplacian_matrix(int P) ;
```

qui crée une matrice creuse de taille P^2 initialisée comme dans (2.5) avec le codage d'indice. Vérifier que votre fonction est correcte en testant la fonction pour de petites valeurs de P.

Question 2.9. Nous nous intéressons ici au cas particulier suivant (choisi uniquement pour la beauté esthétique du résultat !) :

$$u(x,y) = 320x(1-x)y(1-y) \tag{2.8a}$$

$$f_0(x,0) = \mathbf{1}_{x>3/4} \qquad f_0(x,1) = \mathbf{1}_{x>1/4} \tag{2.8b}$$

$$f_0(0,y) = 0 \qquad f_0(1,y) = 1 \tag{2.8c}$$

Écrire une fonction de prototype

```
Eigen::VectorXd make_source_and_boundary_condition(int P);
```

qui crée `Eigen::VectorXd` [6] de taille P^2 et le remplit selon le membre de droite de (2.4).

(Attention, l'écriture `3/4` en `C++` est évalué à 0 car ce sont des entiers...)

6. Ce type est un raccourci pour `Eigen::Matrix<double,Eigen::Dynamic,1>` qui code des matrices de réels à une colonne et à nombre de lignes fixés à la construction.

Dans toutes les questions de cette section, on prêtera une attention particulière aux indices des vecteurs et à la correspondance précédente.

Question 2.10. Écrire une fonction de prototype

```
void squareprint_eigen_vector(
                const Eigen::VectorXd & f,
                std::ostream & output,
                int P, int Q);
```

qui écrit le contenu de `f` qui est un vecteur de taille PQ dans un flux de sortie `output` (terminal ou fichier) avec P valeurs par ligne et Q lignes :

```
f_0  f_1  f_2  ... f_{P-1}
f_P  f_{P+1} ...  f_{2P-1}
...
f_{P(Q-1)} ... ... f_{P*Q-1}
```

Justifier les choix de `const` et `&` dans les prototypes des arguments de cette fonction.

Question 2.11. Résoudre efficacement (2.4) demande de savoir résoudre une équation linéaire $Af = b$ avec A une matrice creuse symétrique et b un vecteur non-creux. Il existe de nombreux algorithmes pour cela mais l'un des plus efficaces est la décomposition de Cholesky qui repose sur la décomposition $A = LL^t$ où L est une matrice triangulaire inférieure ; en particulier, cela ne nécessite pas de calculer nécessairement l'inverse de A (qui est un calcul souvent lourd). Pour cela, on utilisera la syntaxe suivante

```
Eigen::VectorXd b(N); //taille N
Eigen::VectorXd f(N); //taille N
// ... remplissage de b ...
Eigen::SparseMatrix<double> A(N,N);
// ... remplissage de A ...
Eigen::SimplicialCholesky< Eigen::SparseMatrix<double> > Solver(A);
/* cela crée un objet qui contient la décomposition de Cholesky de A
 * et résout efficacement l'équation Af=b avec l'instruction: */
f = Solver.solve(b);
```

Écrire un programme complet qui utilise les fonctions précédentes, résout la discrétisation (2.4) de (2.1) par la méthode précédente pour $P = 50$ et écrit le résultat f selon le format de la question précédente dans un fichier `solution.dat` .

On pourra visualiser ce fichier avec le logiciel libre `gnuplot` en tapant, dans `gnuplot` , la commande `plot "solution.dat" matrix with image` . Cela devrait donner un résultat similaire figure 2.1.

Question 2.12. En reprenant les outils de mesure de temps de la section précédente, refaire la question précédente en faisant varier la valeur de P parmi 25, 50, 100 et 200 (si votre machine le permet pour cette dernière). Quelle complexité observez-vous ? Utiliser l'option de compilation `-O2` change-t-il quelque chose ?

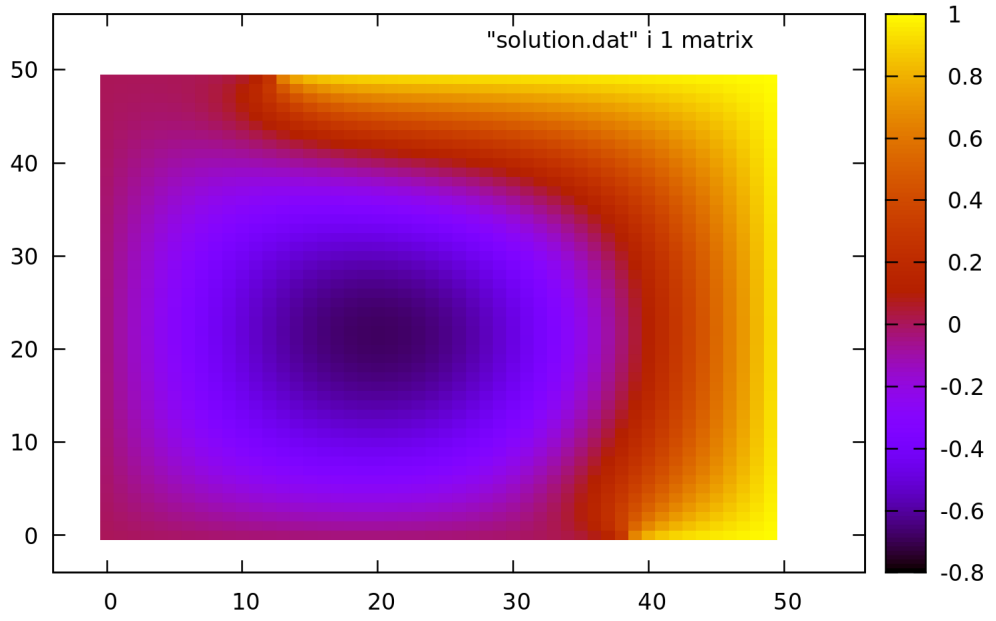


FIGURE 2.1 – Solution de $\Delta f = u$ avec u et les conditions aux bords données par les équations (2.8).

2.3 Ensemble gaussien orthogonal

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 [7]

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

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

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

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

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

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

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

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

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

1. On inclut les bibliothèques `<random>` [8] et `<ctime>` ;
2. On déclare un générateur aléatoire :

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

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

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

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

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

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

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

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

Question 2.13. Nous souhaitons à présent réaliser un histogramme à $M=20$ boîtes sur le segment $[-3,3]$ des valeurs propres normalisées $\lambda/(2\sqrt{N})$. Pour cela, on crée un vecteur `std::vector<double> hist(20,0)` dont chaque case `hist[k]` va contenir le nombre de valeurs propres normalisées qui tombent dans le segment $[-3+k(6/M),-3+(k+1)6/M[$.

On simule ensuite un nombre [9] $n=20$ de matrices indépendantes $(GOE_k)_{1\leq k\leq n}$ de taille $N=150$. Pour chacune de ces matrices, calculer ses valeurs propres (λ_i) et incrémenter `hist[k]` de $\frac{1}{nN}$ si la valeur propre normalisée $\lambda/(2\sqrt{N})$ tombe dans le segment $[-3+k(6/M),-3+(k+1)6/M[$. Si la valeur propre normalisée ne tombe pas dans $[-3,3[$, alors aucune case de `hist` n'est incrémentée.

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

8. La bibliothèque `<random>` nécessite de compiler avec l'option c++ 11 !
9. Si le temps le permet, ne pas hésiter à simuler un nombre plus grand, par exemple 50 !

Conseil : il y a beaucoup de choses à faire dans cette question. Vous pourrez déclarer autant de fonctions élémentaires que vous souhaitez pour réaliser chaque étape séparément (veillez néanmoins aux bons usages de `const` et `&`).

Question 2.14. 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.

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. Plusieurs façons de représenter ces objets sont proposées. Chacune a ses avantages et inconvénients, mais toutes présentent la même interface publique permettant de faire fonctionner le code de test.

3.1 Permutations

Une permutation σ de taille n est une bijection de $\{0, \ldots, n-1\}$ 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 l'élément au dessus :

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

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

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 croissant des premiers éléments. Cette permutation est donc d'ordre $3 \times 2 = 6$.

3.1.1 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 s'inspirant du code suivant. Les fichiers `file_s.dat` et `file_t.dat` seront à télécharger depuis le site du cours.

```
#include "permutation.hpp"
#include <iostream>
#include <fstream>
#include <algorithm> // pour for_each et max_element
#include <random> // pour le générateur std::mt19937
// donc compiler en -std=c++11

int main () {
// Premiere partie
    std::vector<int> v{2,4,5,3,1,0};//syntaxe C++11 avec { }
    Permutation a(v);
    Permutation b(6); //identite
    for(int i=0; i<=6; ++i) {
      std::cout << "a^" << i << std::endl << b << std::endl;
      b = b*a;
    }
    std::list<int> fp = a.fixed_points();
    auto affiche = [](int x) { std::cout << x ;};
    std::for_each(fp.begin(), fp.end(), affiche);

```

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

```
    // Deuxieme partie
22      std::ifstream fichier_s("./file_s.dat");
        std::ifstream fichier_s("./file_t.dat");
24      Permutation s(fichier_s);
        Permutation t(fichier_t);
26      fichier_s.close();
        fichier_t.close();
28
        Permutation u=s*t.inverse();
30      std::cout << "L'ordre de la permutation s*t^-1 est égal à "
            << u.order() << std::endl;
32      std::list<Cycle> l = u.cycles();
        std::cout << "Cette permutation a " << l.size() <<
34          "cycles, dont le plus grand a pour longueur " <<
            (*std::max_element(l.begin(), l.end())).size() <<std::endl;
36          //attention, cela utilise < sur des Cycle !
    // Troisieme partie
38      std::mt19937 g;
        unsigned n=100;
40      unsigned nb_echant = 10000;
        unsigned nb_derang = 0;
42      for(unsigned i = 0; i < nb_echant; ++i) {
            nb_derang += Permutation(n,g).is_derangement();
44      }
        std::cout << "La proportion de dérangements est environ "
46          << nb_derang/double(nb_echant) << std::endl;

48      return 0;
    }
```

Le but est d'essayer de faire fonctionner ce code avec une implémentation de la classe `Permutation` qui propose les opérations suivantes :

- Des constructeurs prenant en argument :
  - un entier n pour créer l'identité de $\mathfrak{S}_n$,
  - un vecteur représentant la seconde ligne du tableau,
  - (plus tard) un flux d'entrée qui permet de lire une permutation depuis le clavier ou un fichier,
  - (encore plus tard) un entier et un générateur de nombres pseudo-aléatoires pour générer des permutations aléatoires.
- une méthode constante `size()` qui renvoie n pour $\sigma \in \mathfrak{S}_n$,
- une méthode constante `extend(int) const` qui plonge canoniquement une permutation de $\mathfrak{S}_n$ dans $\mathfrak{S}_m$ en complétant les images manquantes par l'identité.
- des accesseurs `operator[]` pour lire l'image d'un élément (on pourra mettre en privé des mutateurs correspondants),
- le produit comme une fonction globale amie de la classe `Permutation`, avec la signature `Permutation operator*(const Permutation &, const Permutation &)`,
- la méthode calculant l'inverse `Permutation inverse() const`,
- la méthode constante `fixed_point()` qui renvoie la liste des points fixes sous la

forme de `std::list<int>` ,
— une fonction booléenne `is_derangement()` qui dit si oui ou non une permutation est un dérangement,
— une fonction globale qui permet d'afficher ou de sauvegarder dans un fichier une permutation avec l'opérateur `<<` ,
— une méthode `cycles()` qui renvoie la décomposition en produit de cycles disjoints sous forme d'une liste de cycles,
— une méthode constante `order()` qui renvoie l'ordre d'une permutation.

Ceci constitue l'interface publique de notre classe `Permutation` . On aura aussi besoin d'implémenter la classe `Cycle` .

3.2 Permutation comme vecteur d'images

Une première façon de décrire pour un ordinateur une permutation σ est d'utiliser un vecteur où la case numero i contient $\sigma(i)$.

1. Écrire le code test dans le fichier `test_permutation.cpp` dans lequel on inclura les en-têtes nécessaires. On commentera toutes les instructions de la fonctions `main` et les décommentera au fur et à mesure des questions pour tester l'implémentation par compilation et exécution.
2. Dans un fichier `permutation.hpp` déclarer une classe `Permutation` avec :
   — un champ privé `n` qui est la taille de la permutation,
   — un vecteur `images` de `int` qui contiendra les images.
   On écrira les définitions des méthodes qui ne sont pas *en ligne* et autres fonctions globales amies dans le fichier `permutation.cpp` .
3. Écrire le constructeur `Permutation::Permutation(int)` qui crée l'identité, et le constructeur `Permutation::Permutation(const std::vector<int> &)` qui crée une permutation à l'aide d'un vecteur d'images.
4. Écrire la méthode `size()` ainsi que les accesseurs et mutateurs aux éléments de `images` à l'aide des crochets. *Il est souhaitable mathématiquement de placer les mutateurs des éléments dans la partie privée pour éviter qu'un utilisateur étourdi fasse n'importe quoi : comprenez-vous pourquoi ?*
5. On a parfois envie de considérer une permutation de $\mathfrak{S}_n$ comme une permutation de $\mathfrak{S}_m$, avec $m > n$. Pour cela, écrire une méthode privée `extend(int) const` qui renvoie une nouvelle permutation, étendue par des points fixes pour atteindre la taille passée en argument (si l'argument est plus petit que la taille actuelle, ne rien faire).
6. Écrire le produit, en étendant au besoin les permutations pour que le produit ait un sens. Donner la complexité de cette fonction : étant données deux permutations de $\mathfrak{S}_n$, quel est l'ordre de grandeur du nombre d'opérations élémentaires nécessaires au calcul de leur produit ?
7. Écrire `std::ostream & operator<<(std::ostream &, const Permutation &)` , fonction globale qui affiche sur une ligne la valeur de n, puis sur la ligne suivante le contenu du vecteur `images` séparées par un espace.
8. Écrire la méthode `std::list<int> fixed_points() const` qui renvoie la liste des points fixes d'une permutation σ, i.e. les entiers k tel que $\sigma(k) = k$. (version éventuellement plus facile : renvoyer un objet de type `std::vector<int>`).

Vous devriez être en mesure d'exécuter la totalité de la première partie du programme de test.

9. Écrire la méthode calculant l'inverse et indiquer sa complexité.
10. Déclarer une classe `Cycle` avec un champ privé `elem` de type `std::list<int>`, un constructeur public prenant en argument une liste qui sera copiée dans `elem` et une méthode `order()` qui renvoie la taille d' `elem`.
11. Petit intermède arithmétique : nous allons avoir besoin de calculer le pgcd (gcd en anglais) et le ppcm (lcm en anglais) d'une liste de nombres. Rappel : pour deux entiers a et b, le pgcd d de a et b peut se calculer récursivement par l'algorithme de la division euclidienne : si $b = 0$, on renvoit a, sinon, on renvoie le $pgcd(b, r)$ où `r=a % b` est le reste dans la division euclidienne de a par b. Le ppcm est alors $m = ab/d$. Pour une liste d'entiers $a_0, a_1, \cdots, a_k - 1$, avec $k \geq 3$, on utilise l'associativité et la commutatité du pgcd pour calculer $P_k = pgcd(a_0, a_1, \cdots, a_k - 1)$ par récurrence en posant $P_0 = 0$ et $P_{n+1} = pgcd(P_n, a_n)$ pour $n \geq 0$ (cela fonctionne de la même manière pour le ppcm). Dans un fichier `arithm.cpp` écrire les fonctions[3] `gcd`, `lcm`, `gcd_list` et `lcm_list` permettant de calculer les pgcd et ppcm de deux nombres, ainsi que d'une liste de nombres[4]. Ces fonctions seront déclarées dans le fichier `arithm.hpp`.
12. Écrire la méthode `cycles()` de la classe `Permutation`. 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`. Puis retirer de `S` les images itérées de cet élément par la permutation et les stocker dans une liste jusqu'à retomber sur `x`. Créer alors un cycle et l'ajouter à la liste `L` sauf si `x` est un point fixe, puis essayer à nouveau d'enlever le plus petit élement de `S`, etc. Une fois que `S` est vide, renvoyer `L`.
13. Écrire la méthode `order()` calculant l'ordre d'une permutation.
14. Écrire un opérateur de comparaison $<$ sous forme d'une fonction globale amie de la classe `Cycle` permettant de comparer les longueurs de deux cycles : elle renvoie `true` si la taille du 1er argument est plus petite que celle du 2e. En cas d'égalité des tailles, on renvoie `true` si les éléments du cycle courant sont plus petits dans l'ordre lexicographique[5] (comparaison des champs `elem`).
    Cette méthode est nécessaire pour `std::max_element` en ligne 35 du code test.
15. Écrire une fonction amie permettant l'affichage de cycles. Tester la décomposition en cycles sur l'exemple de la permutation `a` de la première partie.
16. Écrire un constructeur pour une permutation prenant un flux d'entrée en paramètre au même format que l'affichage (le premier champs en particulier servira pour fixer la taille).
17. Exécuter la deuxième partie du programme de test.

3. si votre compilateur prend en charge la norme C++17, le pgcd/ppcm de deux nombres peut etre calculé avec les fonctions `std::gcd` et `std::lcm` dans l'en-tête `<numeric>`.

4. Les plus astucieux d'entre vous reconnaîtront peut-être une utilisation possible de `std::accumulate` !

5. Pour éviter de dangereuses boucles, on pourra si on le souhaite se référer à `std::mismatch` dans `<algorithm>`, voire encore mieux à `std::lexicographical_compare`.

18. Écrire[6] `Permutation::Permutation(int n, std::mt19937 & g)` , un constructeur permettant de renvoyer une permutation choisie uniformément dans $\mathfrak{S}_n$. On se propose pour cela d'utiliser l'algorithme de Fisher-Yates-Knuth[7] : 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 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.
19. Écrire la méthode `is_derangement()` de la classe `Permutation` . Elle pourra être très courte en utilisant les opérations existantes sur les permutations.
20. Exécuter la troisième partie du programme de test.
21. Une action naturelle pour une permutation est de... permuter. Pour les éléments d'un vecteur, cette action s'écrirait $(\sigma \cdot v)[i] = v[\sigma(i)]$. Écrire une méthode "modèle" (*template*) `permute` qui prendra un argument `v` de type `std::vector<T>` (où `T` est un type quelconque) passé en référence, et qui, si la taille de la permutation est inférieure à celle du vecteur, permute (en place) les éléments du vecteur `v` : à la position `i` , on retrouvera l'élément qui était initialement à la position `s[i]` . la déclaration de cette méthode dans la classe aura la forme suivante :

```
    template <typename T>
2   permute(std::vector<T> &) const;
```

On pourra par exemple pour chaque cycle, stocker dans une variable tampon l'élément avec l'indice i_0 le plus petit du cycle, puis stocker dans cette case l'élément d'indice $\sigma[i_0]$, dans lequel on stockera celui d'indice $\sigma^2[i_0]$, etc. jusqu'au dernier indice du cycle qui est celui de la case dans laquelle on stockera la valeur du tampon. Appliquer cette méthode pour le vecteur de `Fiche` s du TP1 et une partition aléatoire de même taille.

3.3 Permutation comme liste de cycles

Une autre façon de représenter les permutations, comme nous l'avons dit, est sous forme de liste (on d'ensemble) de cycles. Une telle permutation aura deux champs privés : l'entier n et la liste `cycles` des cycles.

Le but de cette section est de reprendre les questions précédentes en réimplémentant la classe `Permutation` avec cette description interne. Il s'agit donc de réécrire l'intégralité des fonctions et méthodes publiques présentées au dessus, dans des fichiers `permutation2.hpp` et `permutation2.cpp` . Le fichier d'en-tête sera quasiment identique au fichier `permutation.hpp` modulo la définition des champs privés de `Permutation` . Les points délicats sont :

6. Le générateur `std::mt19937` est déclaré dans `<random>` et c'est donc du standard C++11. N'oubliez pas l'option de compilation associée !

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

— les accesseurs `operator[]` (les mutateurs encore plus !)
— La gestion de la multiplication pour laquelle il faut adapter une stratégie semblable à la décomposition en cycle précédente car il faut calculer les images itérées.
— L'écriture et la lecture vers/depuis un flux.

En revanche, les calculs d'inverse, et de l'ordre sont aisés, la décomposition en cycles, triviale.

Il faut aussi un algorithme de génération aléatoire pour la troisième partie du code test, adapté à la représentation en cycle. Il s'agit du joli algorithme dit du « restaurant chinois » : On ajoute les éléments 0 à $n-1$ successivement dans les cycles de la manière suivante : lorsque les éléments jusqu'à $j-1$ ont été insérés, il y a j intervalles possibles entre deux éléments successifs des cycles déjà crées. On choisit uniformément une de ces positions avec probabilité $\frac{1}{j+1}$ pour y insérer j, ou avec probabilité restante $\frac{1}{j+1}$, on met j dans un nouveau cycle tout seul.

Évaluer les avantages et les inconvénients de cette implémentation pour l'exécution du code test.

3.4 Autres développements

On peut imaginer des développements dans plusieurs directions, comme l'implémentation des permutations comme matrices creuses, avec `Eigen` par exemple. On rappelle qu'à toute permutation $\sigma \in \mathfrak{S}_n$ est naturellement associé la matrice $M_\sigma = (m_{ij})_{1\leq i,j\leq n}$ telle que

$$m_{ij} = \begin{cases} 1 & \text{si } j = \sigma(i) \\ 0 & \text{sinon} \end{cases}$$

Le produit et l'inverse pour les permutations coïncident avec les opérations matricielles correspondantes (on dit qu'on a affaire à une représentation du groupe $\mathfrak{S}_n$ dans $\mathbb{C}^n$).

1. Plutôt que de réimplémenter toute la classe, écrire dans un fichier `perm_matrix.cpp` une fonction globale `perm_to_mat` qui prend une `Permutation` par référence (constante) et renvoie la matrice de permutation associée sous la forme d'une matrice creuse de `Eigen` .
2. Vérifier que la matrice du produit est le produit des matrices, et que celle de l'inverse est l'inverse de la matrice, en utilisant les exemples `file_s.dat` et `file_t.dat` ou d'autres plus petits générés aléatoirement.
3. Vérifier que l'ordre de la permutation est la plus petite puissance de la matrice associée qui est égale à l'identité.

Une fois que vous aurez vu les *templates* on peut imaginer réécrire la classe Permutation de manière générique pour représenter des permutations avec plusieurs types d'entiers (`long int` , `unsigned int` , ...) avec le même code de classe template.

T.P. 4

Une rapide exploration de la bibliothèque standard

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 Les nombres complexes avec `<complex>` : un exemple avec le fractal de Mandelbrot

Une documentation parmi d'autres est présentée ici :

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

Question 4.1. Écrire un programme intitulé `mandelbrot.cpp` qui écrit dans un fichier `fractal.dat` la quantité $K_M(c)$ définie ci-dessous pour N^2 points c régulièrement espacés dans $R_{a,b} = \{z \in \mathbb{C}; Re(a) < Re(z) < Re(b) \text{ et } Im(a) < Im(z) < Im(b)\}$. On choisira les paramètres $a = -1.5 - 1.5i$ et $b = 1.5 + 1.5i$ et les valeurs $M = 256$ et $N = 1000$. Pour les passages à la ligne, on respectera la structure en ligne et colonne des N^2 points.

Soit $M \geq 0$. Pour chaque $c \in R_{a,b}$, on introduit la suite $(z_n)_{n\in\mathbb{N}}$ définie par $z_0 = 0$ et $z_{n+1} = z_n^2 + c$ et on définit :

$$K_M(c) = \min\left(M, \inf\left\{n \geq 0; |z_n| > 2\right\}\right)$$

Question 4.2. Utiliser `gnuplot` dans le terminal avec l'instruction

```
plot "fractal.dat" matrix with image
```

N'hésitez pas ensuite à faire varier les valeurs de a, b (et éventuellement M) pour zoomer sur des régions du fractal de Mandelbrot.

4.2 L'aléatoire avec <random> (C++11) : un exemple 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 :

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

Question 4.3. 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;
   public:
      RandomWalk(int s0,double p);
      ... val(...) ...  //accesseur à s
      ... time(...) ...  //accesseur à n;
      ... reset() ... // redémarrage à l'état initial
      ... void update( ... G) ... ;
            // passage de n à n+1 avec générateur G
};
```

pour implémenter l'évolution de la marche aléatoire (S_n).

Question 4.4. Tester votre classe avec un court programme qui affiche plusieurs réalisations des 10 premiers pas d'une marche.

Question 4.5. Écrire une courte classe

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

qui hérite de la précédente et ajoute le calcul du minimum absolu courant (M_n).

Question 4.6. Écrire un programme `RW.cpp` qui utilise la classe précédente pour réaliser une simulation de longueur $N = 10000$ et écrit les valeurs de (S_n) dans un fichier `RW.dat` et celles de (M_n) dans un fichier `RWmin.dat`, en mettant à chaque fois une valeur par ligne.

Question 4.7. 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.

4.3 Les conteneurs et les itérateurs

4.3.1 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 peut différer

Plusieurs opérations essentielles sont communes à tous les conteneurs :

1. compter le nombre d'éléments du conteneur,
2. effacer tout le conteneur,
3. ajouter ou enlever un élément (éventuellement à un endroit précis si le conteneur est ordonné),
4. la possibilité de le parcourir entièrement sans oublier d'éléments (de manière ordonnée si le conteneur l'est, aléatoirement sinon) ;
5. trouver si un élément est présent et récupérer l'emplacement où il est stocké.

Cela se traduit informatiquement par l'existence d'une méthode `unsigned size() const` pour chaque conteneur pour le premier point, l'existence d'une méthode `void clear()` pour le deuxième point, l'existence de méthode `insert`, `erase`, `pop_front` (enlever au début), `pop_back` (enlever à la fin), `push_front` (ajouter au début), `push_back` (ajouter à la fin) pour le troisième point. Nous vous encourageons à consulter le bas de la page

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

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*. Très grossièrement, 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 indirectement à chaque fois que vous avez utilisé une fonction de `<algorithm>` .

4.3.2 Performances sur différents conteneurs ordonnés

Question 4.8. (performance de quelques algorithmes sur les vecteurs) Écrire un programme qui fait les étapes suivantes :
- crée deux vecteurs `c1` et `c2` de `double` de taille $N = 10000000$ (dix millions),
- remplit le premier avec des nombres aléatoires indépendants de loi exponentielle réduite,
- compte le nombre d'éléments supérieurs à 10 dans le premier,
- élève au carré les 15 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` .

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.

Faire tourner le même programme en multipliant N par 2, 4 et 8. **Discuter** les résultats obtenus.

Question 4.9. Refaire le programme en utilisant `std::deque` à la place `std::vector` . **Discuter** les différences de temps avec l'exemple précédent.

Question 4.10. Refaire le programme 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 4.11. (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.

4.3.3 Le conteneur `std::map`

Les questions qui suivent sont indépendantes des précédentes.

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. Cela fonctionne de la manière suivante :

```cpp
#include <map>
...
std::map< T1, T2 > M;// déclaration de M
unsigned n=M.size(); // taille de M
M[x]=t; // associe la valeur t de type T2
        // à la valeur x de type T1
M[x]=u; // associe la nouvelle valeur u de type T2
        // à la valeur x de type T1
        // t est alors effacée.
T2 u=M[y]; // donne la valeur associée à x s'il y en a une
        // renvoie la valeur par défaut de T2 sinon
// Affichage de toutes les associations stockées par T1 croissant:
for(std::map<T1,T2>::iterator it=M.cbegin(); it != M.cend(); it++) {
   std::cout << it->first << " " << it->second << "\n";
}
// Equivalent en C++11
std::for_each(
         M.cbegin(),M.cend(),
         [&](const std::pair<T1, T2> & x){
            std::cout << x.first << " " << x.second << "\n";
         });
```

Question 4.12. 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 utilise

```
std::map<std::string,unsigned>
```

qui compte le nombre d'apparitions de chaque mot.

Dans toutes les question suivent de cette section sauf la dernière, on complètera ce programme.

Question 4.13. Écrire dans un fichier `stats.dat` chaque mot avec sa fréquence en ordre lexicographique.

Question 4.14. Combien y a-t-il de mots différents ? Combien y a-t-il de mots différents de plus de 7 lettres ou plus ? Quel est le mot le plus fréquent et combien de fois apparaît-il ? Combien y a-t-il de lettres au total dans la déclaration ?

Question 4.15. (plus difficile) Afficher 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" (indice : vous pourrez consulter la documentation de < string> pour décrire les deux derniers critères).

Question 4.16. (bonus) Le conteneur `std::set` : en lisant la documentation de ce conteneur, écrire une fonction

```
std::set<std::string> filter(
        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.

Question 4.17. (bonus) Reprendre la marche aléatoire de la section 4.2 et, à l'aide de `std::map<int,unsigned>` , réaliser l'histogramme de la position finale de la marche (S_n) pour un grand nombre de réalisations indépendantes de la marche et l'écrire dans des fichiers (visualiser avec gnuplot). Quel théorème de probabilité cela illustre-t-il ?

4.4 (Bonus) Les retours multiples avec `<tuple>` (C++11)

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;
...
X=std::make_tuple(x,n,s); // définition de X à partir de x, n, s
...
std::tie(x,n,s)=X; // 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

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

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 p.

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_(n-1) echantillon[n-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.

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.

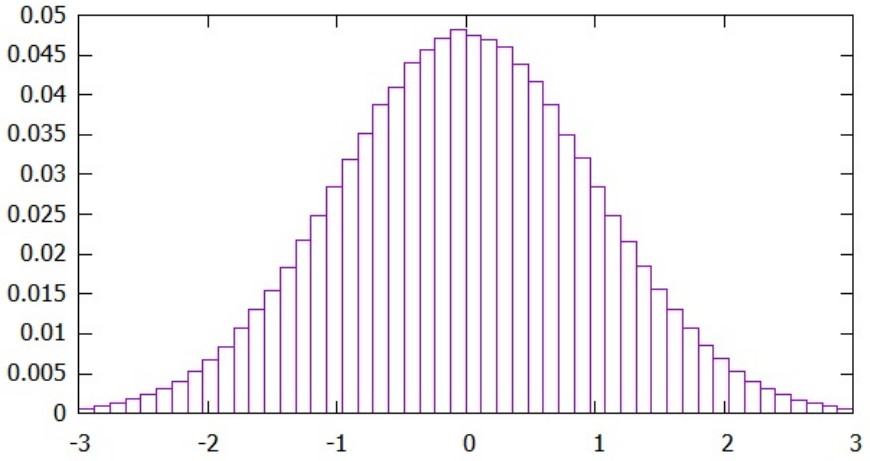


FIGURE 5.1 – Affichage de l'histogramme de la loi $\mathcal{N}(0,1)$ sous `gnuplot`

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 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`

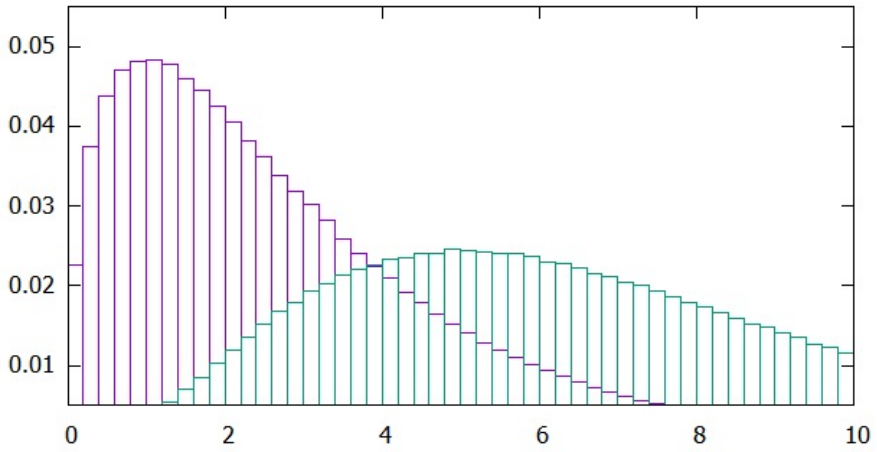


FIGURE 5.2 – Affichage de l'histogramme de la loi χ_2 avec paramètres respectifs 3 et 6

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:
```

```
6       ....
};
```

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 :

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

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 : un peu de *backtracking*

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

Création d'un conteneur : arbres binaires auto-équilibrants

Le but de ce TP est d'écrire de zéro une classe représentant une structure de données (c'est à dire un *conteneur*), qui présente une interface similaire aux conteneur de la bibliothèque standard. Cela permettra en effet de pouvoir utiliser les algorithmes de la bibliothèque standard sur ces structures de données.

Nous présentons ici les *arbres AVL*, nommés après leurs inventeurs Adelson-Velsky et Landis. Il s'agit d'arbres binaires de recherche, qui ont la propriété de s'équilibrer automatiquement au cours de leur utilisation (nous détaillerons plus tard cette propriété). Ce genre de structure est adaptée à la représentation d'ensembles : une valeur particulière ne peut être insérée au plus qu'une fois. Les données sont supposées d'un type ayant un ordre total $<$. Le stockage utilise efficacement cet ordre pour garantir des coûts (ici en moyenne, et dans le pire des cas) d'insertion, de recherche et de suppression en $O(\log(n))$, pour un arbre contenant n éléments.

Un arbre binaire peut être vu comme un graphe dirigé, avec un sommet privilégié, appelé racine. Chaque sommet a au plus deux arêtes (gauche/droite) sortantes. Chaque sommet hormis la racine a exactement une arête entrante, la racine n'en a aucune.

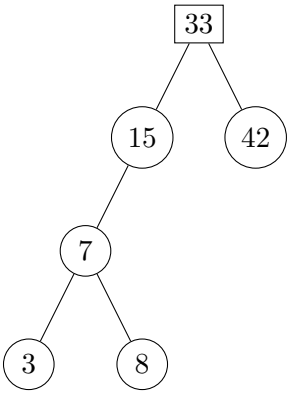


FIGURE 6.1 – Arbre binaire de recherche contenant les entiers 3, 7, 8, 15, 33 et 42. La racine est représenté par le nœud carré, contenant 33. Les autres nœuds ont exactement un parent, situé au dessus d'eux, auquel ils sont reliés par une arête. Cet arbre binaire n'est pas équilibré : le sous-arbre gauche de 15 a une hauteur 2 alors que le sous-arbre droit a une hauteur nulle.

Un arbre binaire de recherche est un arbre binaire dont les nœuds sont étiquetés avec

des données, de telle sorte que les étiquettes dans le sous-arbre gauche (resp. droit) sont strictement inférieures (resp. supérieures) à l'étiquette du nœud.

Dans un arbre de recherche classique, les données sont insérées au fur et à mesure en respectant l'ordre des données : on part de la racine, et on descend dans l'arbre à gauche ou à droite suivant que la donnée à insérer est plus petite ou plus grande que l'étiquette du nœud. Arrivé à une feuille, on l'insère à gauche ou à droite pour respecter l'ordre.

Pour des données dans un ordre typique, l'arbre ainsi construit est plutôt équilibré : les branches ont plus ou moins le même nombre de branche, ce qui fait que la hauteur de l'arbre (distance de la racine à la feuille la plus loin) pour n données est de l'ordre de $O(\log(n))$. En moyenne, les opérations de recherche et d'insertion sont donc très efficaces. Par contre, si par exemple, les données à insérer arrivent triées dans l'ordre croissant, alors elles seraient alors ajoutées au fur et à mesure comme sous-arbre droit du dernier nœud ajouté : la hauteur est alors n et la structure arborescente n'est pas exploitée.

Pour les arbres AVL, après insertion (ou suppression) d'un nœud, une étape d'*équilibrage* est ajoutée, pour assurer que la différence de hauteur entre les sous-arbres gauche et droit est toujours dans $\{-1, 0, 1\}$. Cela évite les situations similaires à celle représentée sur la figure 6.1 et garantit que la hauteur est toujours logarithmique en le nombre de données contenus dans l'ordre (même dans le pire des cas), assurant une grande performance pour l'insertion et la recherche de données.

Le but du TP est d'écrire une classe générique (template) `Tree<T>` pour des données d'un type quelconque `T` ayant un ordre total. Le sujet présente d'abord une implémentation pour les `long int` puis propose de la transformer en classe générique. Vous pouvez choisir d'écrire directement la version template [1].

6.1 Les arbres AVL pour les `long`

L'implémentation de la structure de données necessitera trois classes : la classe `Node` représentant les nœuds de l'arbre, la class `Tree` pour l'arbre de recherche à proprement parler, et une classe `TreeIterator` pour fournir l'interface compatible avec la bibliothèque standard et ses algorithmes. La partie sur les itérateurs fait l'objet de la fin de la section.

Dans un fichier `avl_tree.hpp`, déclarer les classes de la façon suivante. Les champs et méthodes seront étoffés au fur et à mesure de la section.

```
#ifndef _AVL_TREE_HPP
#define _AVL_TREE_HPP

class Tree;

class Node {
  private:
    long data; // étiquette du nœud
    int height; // hauteur du nœud dans un arbre
    Node * left;
    Node * right;
  public:
    Node(long x);
```

1. dans ce cas, l'implémentation de toutes les méthodes et fonctions génériques devront être écrites dans le fichier entête de la classe.

```
14      friend class Tree;

16  };

18  class Tree {
      private:
20      Node * root; // racine de l'arbre

22    public:
        Tree(Node * r);
24      Node * insert(long x); // plus tard : TreeIterator insert(long x);
        Node * search(long x); // plus tard : TreeIterator search(long x);
26
    };
28
    #endif
```

Le code des méthodes et fonctions un peu longues seront placées dans un fichier `avl_tree.cpp` .

6.1.1 Construction, insertion et affichage

Question 6.1. Écrire *inline* le constructeur de la classe `Node` créant un nœud « nu », de hauteur 1, avec `left` et `right` égaux au pointeur nul[2] `nullptr` . Écrire un accesseur `value` pour le champ `data` de la classe `Node` .

Question 6.2. Écrire *inline* le constructeur pour la classe `Tree` , affectant le pointeur `r` à la racine de l'arbre. Modifier sa définition pour en faire un constructeur par défaut, donnant un arbre vide, dont la racine est le pointeur nul.

L'ajout de données dans l'arbre entraînera l'allocation de mémoire pour stocker les nœuds ajoutés dynamiquement. Il faut que cette mémoire soit libérée une fois l'arbre inutile. C'est le travail du destructeur.

Question 6.3. Écrire le destructeur de la classe `Node` , qui libère les sous-arbres gauche et droit s'ils sont non-vides. On fera bien attention plus tard que si on veut supprimer un nœud d'un arbre, il faudra penser à décrocher les enfants, c'est à dire de stocker le contenu des champs `left` et `right` dans des variables, et mettre ces champs à `nullptr` , sous peine de perdre tous les sous-arbres. Écrire le destructeur de la classe `Tree` qui libère la mémoire allouée pour la racine, si celle-ci n'est pas le pointeur nul. Le code à proprement parler de ces destructeurs devrait faire entre 3 et 5 lignes.

Question 6.4. Définir une méthode privée `Node * Tree::balance(Node * r)` qui pour l'instant renvoie uniquement `r` . Elle sera modifiée dans la suite pour se charger de l'équilibrage de l'arbre.

2. L'usage de `nullptr` au lieu de `NULL` nécessite le standard C++11,

Question 6.5. L'insertion dans un arbre binaire de recherche (en mettant pour l'instant de coté la question du rééquilibrage) d'une valeur `x` se fait en parcourant l'arbre à partir de la racine, en descendant le long des branches. Si l'arbre est vide, on rajoute un noeud racine avec étiquette `x`. Sinon, on part de la racine et procède itérativement ainsi : si l'étiquette du nœud `n` qu'on visite est égale à `x`, on ne fait rien (on ajoute une valeur au plus une fois). Si elle est plus petite, on essaie de l'insérer dans le sous-arbre gauche : soit il est vide et on y place un nouveau nœud avec l'étiquette `x`, soit il est non vide, et on visite le nœud pointé par `n.left`. On procède similairement si x est plus grand que l'étiquette de `n` dans le sous-arbre droit commençant à `n.right`. Mettre en application cet algorithme *à la main* pour l'insertion des valeurs successives 2, 1, 3 et 4 dans un arbre initialement vide. On présentera les différentes étapes, ainsi qu'un schéma de la mémoire.

Question 6.6. Déclarer une méthode publique[3] `insert` dans la classe `Tree` dont le code sera :

```
Node * Tree::insert(long x) {
  Node * ins=nullptr;
  root = insert_and_balance(root, ins, x);
  return ins;
}
```

Tout le travail est donc délégué à la méthode

```
Node * Tree::insert_and_balance(Node * r, Node * ins, long x)
```

qu'on déclarera comme privée. Le premier argument est un pointeur vers le nœud à partir duquel on veut faire l'insertion, le deuxième est un pointeur passé en argument qui pointera à la fin de l'appel vers le nœud qui a été inséré. La valeur de retour d' `insert_and_balance` est un pointeur vers le sommet du sous-arbre dans lequel on fait l'insertion. Il pourra être différent du premier argument, après la phase d'équilibrage (*balance*). C'est pour cela qu'on a besoin de réaffecter `root` pour éviter qu'une partie de l'arbre se « décroche ». L'insertion d'un nombre x dans le sous-arbre partant du nœud pointé par `r` se fait de la façon suivante :

- Si `r` est le pointeur nul, on crée un nouveau nœud avec x comme étiquette, et on en fait la racine de l'arbre : `r=new Node(x);`. On affecte `r` à `ins` et on renvoie `r`.
- Si l'étiquette x est strictement plus petite que celle attachée à `*r`, deux possibilités se présentent à nous :
  - soit `r->left` est nul, dans ce cas, on y insère un nouveau nœud :

    ```
    r->left=new Node(x);
    ins=r->left;
    ```

  - soit `r->left` est non nul, alors

3. On changera plus tard le type de renvoi à `TreeIterator` quand cette classe sera implémentée.

```
r->left=insert_and_balance(r->left, ins, x);
```

Pareil à droite si l'étiquette de `r` est strictement plus grande que x.

— Si jamais x est égal à l'étiquette de `r`, on ne fait rien (on ne veut pas de duplication.

À la fin, on renvoie ensuite l'opération d'équilibrage, `return balance(r)` au nœud `r`, qui sera appliqué dans tous les cas sauf le tout premier (correspondant à l'ajout du premier nœud dans un arbre).

Question 6.7. Ajouter un champ `taille` à la classe `Tree`, l'initialiser correctement dans le constructeur et le mettre à jour lors de l'insertion. Écrire un accesseur `size()` pour ce champ[4].

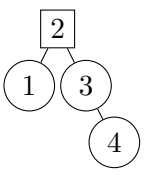
6.1.2 Affichage et équilibrage

Question 6.8. Dans le fichier `main_tree.cpp`, écrire une fonction `main` qui insère dans un arbre `t` les nombres 2, 1, 3 et 4. Vérifier qu'il n'y a ni erreur de syntaxe à la compilation, ni erreur de segmentation.

Question 6.9. On veut pouvoir visualiser l'arbre. Pour cela, on définit une méthode `void Tree::affiche(std::ostream & o) const`, qui délègue le travail à une méthode privée `void Tree::affiche_rec(std::ostream &o, Node *p, unsigned n)`, en l'appelant avec comme paramètre `o`, `root` et `0`. La méthode `affiche_rec` affiche sur une ligne la valeur de l'étiquette du noeud pointé par `p` décalé par `2*n` espaces (si `p` est `nullptr`, la ligne reste vide). On pourra utiliser le fait que le constructeur `std::string(unsigned k, char c)` crée une chaine de caractères répétant k fois le caractère `c`. Ensuite, elle s'appelle elle-même sur les nœuds gauche et droit avec le paramètre `n` $+1$. On pourra choisir d'écrire 'g' et 'd' au début des lignes pour les sous-arbres gauche et droit, avant les espaces, pour faciliter l'identification des sous-arbres. L'affichage de `t` de la fonction `main` devrait donner quelque chose comme cela :

```
2
g|  1
g|
d|
d|  3
g|
d|    4
g|
d|
```

ce qui doit correspondre à l'arbre suivant :



4. On pourrait écrire `size()` pour calculer le nombre de nœuds en parcourant l'arbre, mais cela aurait un coût linéaire en le nombre de données dans l'arbre plutot que constant avec un champ dédié.

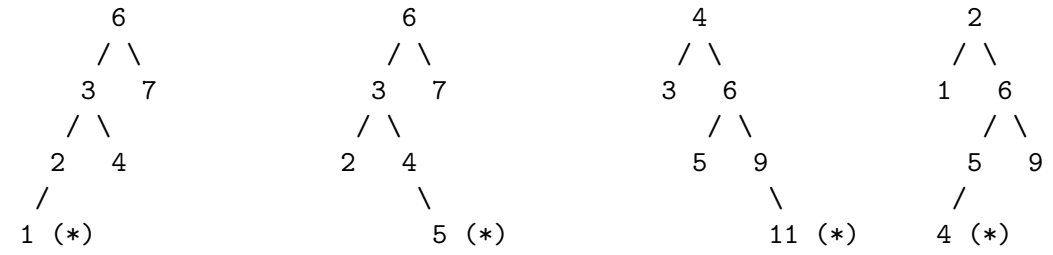


FIGURE 6.2 – Exemples des 4 situations dans lesquelles un équilibrage est nécessaire après insertion du dernier élément (noté par une étoile *). Chacun de ces cas de figures se résout par une rotation. De gauche à droite : `rotateR` , `rotateLR` , `rotateL` , `rotateRL` .

Dans la fonction `main` , créer un arbre `u` dans lequel seront insérés 25 nombres entiers longs aléatoires entre 0 et 10000, et l'afficher [5].

Pour l'instant, les nœuds sont insérés dans l'arbre, mais leur champ height est toujours à 1, valeur attribué à la construction. D'autre part, suivant l'ordre d'arrivée des éléments à insérer, l'équilibre (la différence de hauteur entre les sous-arbres gauche et droit) peut devenir grand. Nous allons corriger ces deux points dans les questions qui suivent.

Question 6.10. Écrire une méthode privée `unsigned Tree::height(Node *p) const` qui renvoie la hauteur du noeud pointé par `p` s'il est non nul, et renvoie 0 si `p` est `nullptr` . Écrire une méthode privée `void Tree::set_height(Node *p)` pour recalculer la hauteur du noeud pointé par `p` , comme 1 plus le maximum des hauteurs des noeuds gauche et droit. Modifier `balance` pour appeler `set_height` sur le paramètre avant de retourner. Modifier la fonction d'affichage pour que la hauteur d'un nœud soit écrite après la valeur de son étiquette entre parenthèses.

Nous allons étayer notre méthode `balance` pour qu'une fois la hauteur du nœud pointé par `p` recalculée, on rééquilibre l'arbre si l'insertion d'un noeud a rompu cette équilibre, à l'aide de rotations d'arbres. L'équilibre est rompu si la valeur absolue de la différence entre les hauteurs des sous-arbres gauche et droit devient égale à 2 après insertion (on souhaite qu'elle soit toujours entre -1 et 1). Cela arrive dans 4 situations, représentées sur la figure 6.2, qui peuvent chacune être résolue par une rotation.

La rotation à gauche `rotateL` , décrite sur la figure 6.3, peut être écrite de la manière suivante :

```
Node * Tree::rotateL(Node * p) {
  Node * q = p->right;
  p->right = q->left;
  q->left = p;

  set_height(p);
  set_height(q);
```

5. Cette méthode d'affichage, pratique pour la visualisation, est temporaire. Dès qu'on aura les itérateurs, on préférera une boucle `for` ou `for_each` , comme pour les listes.

```
8     return q;
    }
```

La rotation droite `rotateR` a une écriture analogue.

Question 6.11. Déclarer et écrire le code des méthodes privées `rotateL` et `rotateR`. Pour la rotation gauche-droite `rotateLR`, on applique d'abord `rotateL` sur `p->left`, puis on renvoie `rotateR(p)`. La rotation droite-gauche `rotateRL` est définie similairement en échangeant le rôle de gauche et droite. Déclarer et définir ces deux méthodes privées. Tester sur papier l'équilibrage proposé par ces 4 méthodes sur les exemples de la figure 6.2 pour lesquels on devrait obtenir des arbres « à 3 étages » au lieu de 4.

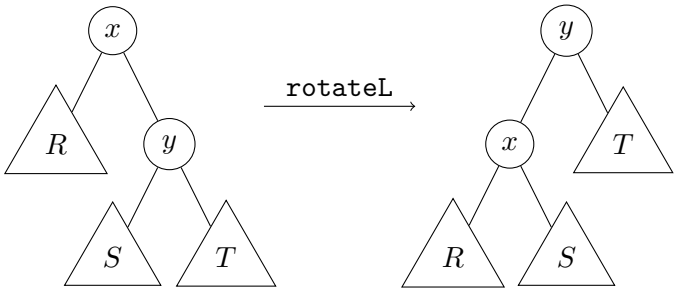


FIGURE 6.3 – Rotation gauche (`rotateL`) au nœud contenant x, lorsque la hauteur du sous-arbre droit du sous-arbre droit T devient strictement plus grande que R. Lors de l'insertion dans un arbre équilibré cela arrive lorsque R et S sont vides, et T est réduit à un nœud contenant la valeur insérée. La version plus générale peut être nécessaire pour la suppression.

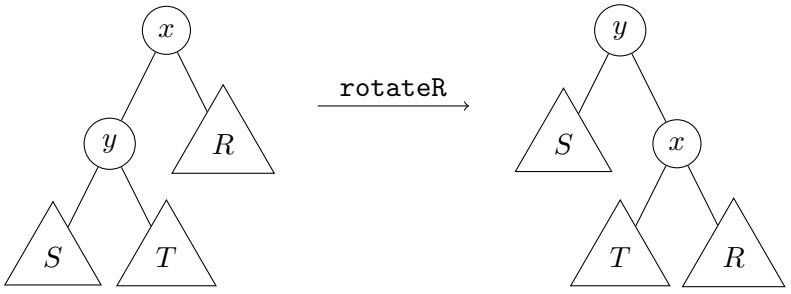


FIGURE 6.4 – Rotation droite (`rotateR`) au nœud contenant x, lorsque la hauteur du sous-arbre gauche du sous-arbre gauche T devient strictement plus grande que R.

Question 6.12. La méthode `balance` doit être adaptée de la manière suivante : après avoir recalculé la hauteur du paramètre `p`, on fait les tests suivants pour savoir si une rotation est à faire (et laquelle) :

- Si la hauteur de `p->left` est égale à 2 + celle de `p->right` :
  - si `p->left->right` est plus haut strictement que `p->left->left`, on renvoie le résultat de la rotation LR sur `p`

- sinon on renvoie le résultat de la rotation R sur `p`
- Si la hauteur de `p->right` est égale à 2 + celle de `p->left` :
  - si `p->right->left` est plus haut strictement que `p->right->right`, on renvoie le résultat de la rotation RL sur `p`
  - sinon on renvoie le résultat la rotation L sur `p`

Dans les autres cas, le sous-arbre en `p` est équilibré, il n'y a donc rien à faire, et on renvoie juste `p`.

Question 6.13. Réafficher les arbres `t` et `u` et vérifier qu'on obtient des arbres bien équilibrés.

Question 6.14. Écrire une méthode publique `Tree::search(long x)` qui renvoie pour l'instant un `Node *`. Elle appelle une méthode privée[6] `search_rec(long, Node *p)`, renvoyant un pointeur vers un nœud, avec les paramètres `x` et `root`. Lors de l'appel à `search_rec(x,p)`, si `p` est `nullptr` ou l'étiquette de `p` égale `x`, on renvoie `p` (correspondant aux cas où on ne trouvera pas `x` ou au fait qu'on a trouvé `x`). Sinon, on cherche dans le sous-arbre gauche ou droit de `p` suivant la comparaison de `x` avec l'étiquette de `p`.

Question 6.15. Comme il y a des opérations d'allocation de mémoire dans gestion des objets `Tree`, les opérateurs d'affectation et constructeurs par copie fournis par le compilateur ne font pas ce qu'il faut (non libération des anciens nœuds lors de l'affectation, partage des nœuds avec des pointeurs différents au lieu de copie des nœuds lors de la copie,...). Écrire explicitement un constructeur par copie et un opérateur d'affectation gérant correctement la mémoire.

6.2 Itérateurs

On a maintenant les fonctionnalités de base pour notre conteneur. On souhaite désormais proposer une interface similaire aux conteneurs de la STL (en particulier, `std::set` dont `Tree` est le plus proche). On veut donc que nos méthodes `search` et `insert` renvoient non pas un pointeur, mais un *itérateur* vers un nœud de la structure de données, et que chaque arbre propose un itérateur de début `begin()` et de fin `end()` pour pouvoir appliquer des algorithmes sur tous les éléments de la structure.

On veut écrire une nouvelle classe (toujours dans le fichier `avl_tree.hpp`) nommée `TreeIterator`, contenant un champ privé `p` de type `Node *p`. Les objets servant d'itérateurs doivent avoir les méthodes `long operator*() const` permettant d'accéder à la valeur pointée par l'itérateur, `TreeIterator & operator++()` (incrémentation préfixe) pour passer à l'élément suivant, et `bool operator!=(const TreeIterator &)` pour comparer (et par exemple tester si on est à la fin du conteneur). Voici la déclaration de la classe (qui nécessite l'entête `iterator` :

6. on aurait pu avoir ressort à une valeur par défaut `p=root` et utiliser une seule fonction pour search. Ce ne sera plus possible quand on aura les itérateurs car `search` en renverra un alors que `search_rec` continuera à renvoyer un pointeur.

```
class TreeIterator {
  private:
    Node * p;
  public:
    // types demandés par certains algorithmes de la STL
    // remplacer long par T pour la classe générique TreeIterator<T>
    typedef std::forward_iterator_tag iterator_category;
    typedef long value_type;
    typedef long* pointer;
    typedef long& reference;
    typedef std::ptrdiff_t difference_type;

    TreeIterator(Node *r);
    TreeIterator & operator++();
    bool operator==(const TreeIterator &) const;
    bool operator!=(const TreeIterator &) const;
    long operator*() const;
};
```

6.2.1 Opérations sur les itérateurs

Question 6.16. Écrire un constructeur *inline* pour la classe `TreeIterator` initialisant `p` à partir d'un pointeur passé en paramètre.

Question 6.17. Le début d'un arbre non vide est le nœud obtenu en allant le plus possible à gauche depuis la racine (avant de toucher le pointeur nul). La « fin » représente un élément qui serait après le dernier nœud : il sera représenté par le `TreeIterator` construit à partir du pointeur nul. Écrire les méthodes `begin()` et `end()` pour la classe `Tree`.

Question 6.18. Écrire la méthode `long TreeIterator::operator*() const` et les opérateurs d'égalité et de non-égalité entre itérateurs. L'appel l' `operator*` sur `t.end()` ou sur tout autre itérateur initialisé par le pointeur nul est indéfini. Dans ce cas, renvoyer par exemple la valeur par défaut du type de donnée (long ou `T`) pour éviter les erreurs de segmentation.

6.2.2 Prendre soin des parents

Afin de se promener dans l'arbre librement, (en particulier pour trouver le successeur d'un nœud dans l'arbre), il faut pouvoir non seulement descendre dans les branches (ce qu'on peut faire avec `left` et `right`, mais il faut aussi pouvoir remonter ! Il nous faut donc enrichir notre classe `Node` avec un champ `parent` qui pointera vers le nœud juste au dessus (et sera nul pour la racine de l'arbre).

Question 6.19. Ajouter le champ `parent` à la classe. Modifier le constructeur de `Node` pour prendre deux paramètres, dont le deuxième, de type `Node *`, servira à initialiser le

champ `parent` et aura la valeur `nullptr` par défaut s'il n'est pas précisé (pour imiter le comportement précédent).

Question 6.20. Modifier l'appel au constructeur `Node` dans la méthode privée d'insertion `Tree::insert_and_balance` et vérifier l'affectation des champs `parent` lors des rotations L et R. On pourra avoir besoin d'une méthode `void Node::fix_childrens_parent()` qui vérifie que le nœud sur lequel elle est appelée est bien le parent de ses nœuds gauche et droit (s'ils sont non vides), qui pourra être utile à la fin de `rotateL` et `rotateR`.

Question 6.21. Modifier la fonction d'affichage des arbres pour ajouter sur la ligne correspondant à chaque nœud la valeur de son parent, et vérifier sur les exemples `t` et `u` que tout fonctionne correctement.

Question 6.22. Le parcours des éléments de l'arbre se fait dans l'ordre croissant. Le nœud suivant un nœud `n` dans l'arbre est trouvé de la manière suivante :

— le nœud le plus à gauche dans le sous-arbre droit s'il y a un sous-arbre droit
— sinon, on remonte les parents jusqu'à en trouver un qui a une étiquette plus grande que celle de `n`.

Si on n'en trouve pas (la dernière recherche finit par `nullptr`), on est à la fin. Écrire l'opérateur `operator++()` qui modifie l'itérateur courant en l'itérateur construit à partir du pointeur vers le nœud suivant dans l'arbre.

6.2.3 Application d'algorithmes

Question 6.23. En utilisant `std::for_each` ou une boucle `for` dont la variable est un itérateur `TreeIterator`, afficher tous les nœuds des exemples `t` et `u` sur une seule ligne, en les séparant par un espace.

Question 6.24. Ecrire un constructeur de la class `Tree` prenant un flux d'entrée en paramètre, et construisant un arbre obtenu en insérant tous les nombres lus depuis le flux. Télécharger le fichier `tp6_nombres.txt` depuis le site du cours. Dans le fichier `main_tree.cpp`, construire l'arbre correspondant à ces entiers, vérifier que 106045 et 107809 sont dans l'arbre et afficher tous les éléments entres ces deux valeurs.

6.3 Réécriture pour des templates

Maintenant que notre conteneur fonctionne correctement et efficacement pour le type `long`, on voit que la totalité des algorithmes sous-jacents sont complètement génériques, et n'utilisent pas la spécificité du type `long` à part pour l'existence de l'ordre total. Dans un sous-répertoire template, reprendre le fichier d'entête pour les arbres AVL pour long, et le modifier pour la programmation générique avec des classes et fonctions templates, pour un type de données `T`. Attention, pour des classes templates, les méthodes doivent être définies dans le fichier entête, pour que le compilateur puisse savoir à la compilation quelles sont les versions qui doivent être générées.

Question 6.25. Réécrire entièrement les trois classes précédente en classes génériques `Node<T>`, `Tree<T>` et `TreeIterator<T>` dan un autre fichier entête `avl_tree2.hpp` qui contiendra toutes les méthodes et fonctions génériques (pas de fichier d'implémentation séparé). Copier et adapter le contenu de `main_tree.cpp` pour le faire fonctionner avec les nouvelles classes, et vérifier que tout fonctionne encore.

Question 6.26. Les distributions Linux localisées pour le français fournissent un fichier `/usr/share/dict/french` qui contient une liste d'environ 300 000 mots français (un par ligne). Insérer tous ces mots dans un arbre. Compter combien de mots entre "hôpital" et "prisme" s'écrivent avec au moins deux 'y'.

Question 6.27. Comparer les temps de recherche et d'insertion de "priflachou" dans l'arbre, dans un `std::set`, une `std::list` et un `std::vector` à l'aide de `<chrono>`.