T.P. 5

Templates et méthodes de Monte-Carlo

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

5.1 La fonction génerique `MonteCarlo`

5.1.1 Le principe de l'algorithme...

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

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

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

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

5.1.2 ... Et sa mise en pratique

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

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

Les paramètres du template sont :

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

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

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

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

On part du principe que :

— la classe `RandomVariable` possède un template de méthode

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

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

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

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

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

5.2 Exemples d'applications

5.2.1 Approximation de π

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.2.2 Approximation d'intégrales

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

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

Pour la seconde intégrale, on notera que

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

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

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

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

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

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

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

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

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

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

et écrire un constructeur

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

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

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

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

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

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

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

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

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

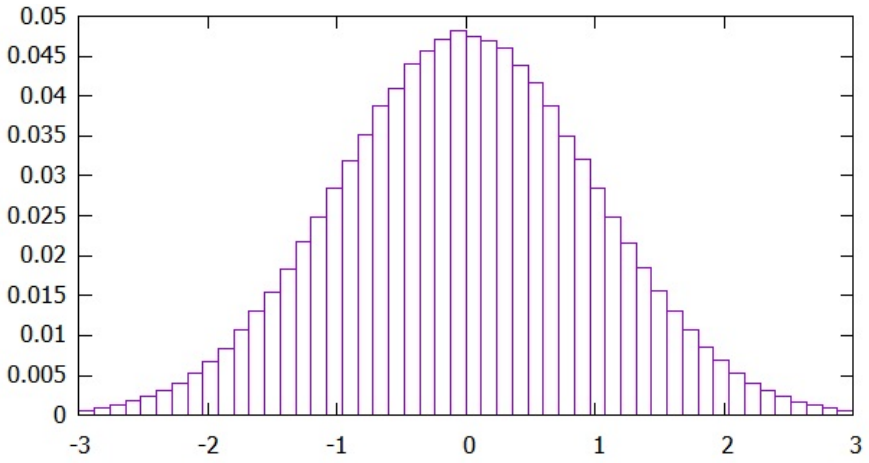

FIGURE 5.1 – Affichage de l'histogramme de la loi $\mathcal{N}(0,1)$ sous `gnuplot`

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

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

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

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

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

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

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

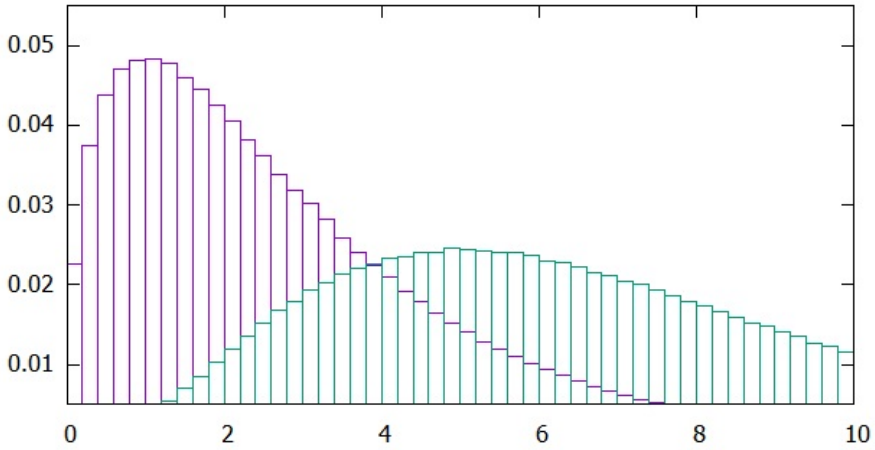

FIGURE 5.2 – Affichage de l'histogramme de la loi χ_2 avec paramètres respectifs 3 et 6

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.4 Bonus : retour sur des TP antérieurs

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

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

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