Sorbonne Université **M1 Mathématiques**
2018–2019 **4M056 : Programmation en C++**

TP noté 2 : Algorithme du recuit simulé

À l'issue du TP, chaque binôme envoie par email ses fichiers `.hpp` et `.cpp`. **Il est impératif de mettre en commentaire, dans tous les fichiers, les nom, prénom et n° d'étudiant de chacun des membres du binôme.**

Description La méthode du recuit simulé permet d'estimer des minima locaux (mais qu'on espère globaux en général) d'une fonction $\phi : E \to \mathbb{R}$. Elle est basée sur l'idée suivante : pour tout $T > 0$ (appelé température), on introduit la mesure π_T sur E définie[1] par :

$$\pi_T(x) = \frac{1}{Z_T} \exp(-\phi(x)/T)$$

où Z est une constante de normalisation qui n'aura aucun intérêt pour nous. Lorsque $T \to 0$, la mesure π_T se concentre sur les minima de ϕ. Lorsque $T \to +\infty$ au contraire, π_T ressemble à la mesure uniforme sur E.

La méthode du recuit simulé consiste à simuler une suite de v.a. X_n dont la loi est *proche*[2] de π_{T_n} avec (T_n) une suite de température qui tend vers 0.

Algorithme. Soit (T_n) une suite déterministe de température telle que $T_n \to 0$. Soit $x_0 \in E$ fixé. Soit $(Q_x)_{x\in E}$ une famille de lois de probabilité indexée par x. On génère une suite (X_n) de v.a. à valeurs dans E de la manière suivante :

- $X_0 = x_0$ p.s.
- pour chaque $n \in \mathbb{N}$, on choisit Y_{n+1} aléatoirement dans E selon la loi Q_{X_n} et on choisit aléatoirement U_{n+1} selon une loi uniforme sur $[0, 1]$.
- si $U_{n+1} \leq \min(1, \exp((\phi(X_n) - \phi(Y_{n+1}))/T_{n+1}))$, alors on pose $X_{n+1} = Y_{n+1}$, sinon on pose $X_{n+1} = X_n$.

1. Écrire dans un fichier `recuit.hpp` un modèle de fonction :

```
template <class E,class Func, class TempSeq, class RandomY, class RNG>
E recuit_simule(const Func & phi, E x0, const TempSeq & T,
                const RandomY & Y,RNG & G, long unsigned N);
```

avec les contraintes suivantes :

- `phi` possède une méthode `double operator()(const E & x) const` qui calcule $\phi(x)$.
- `T` possède une méthode `double operator()(long unsigned n) const` qui renvoie T_n.
- `Y` possède une méthode `E operator()(const E & x, RNG & G) const` qui renvoie une réalisation d'une v.a. Y à valeurs dans E de loi Q_x.
- `G` est n'importe quel générateur de nombres pseudo-aléatoires de la STL.
- `N` est un nombre d'itérations à réaliser
- le résultat final est la valeur de X_N.

Premier test. Pour un premier test, nous souhaitons estimer un minimum de la fonction $\phi : \mathbb{R} \to \mathbb{R}$ définie par $\phi(x) = x^6 - 48x^2$. Pour cela, on prendra $T_n = 10 \cdot (0.9)^n$, $x_0 = 0$, $Y_{n+1} = X_n + W_n$ où W_n est une v.a. de loi normale centrée réduite et $N = 1000$.

1. Plus précisément : si E est discret, alors la formule est celle proposée ; si $E = \mathbb{R}^d$, l'expression donnée est la densité par rapport à la mesure de Lebesgue.
2. Il y a très peu de justification théorique derrière cette méthode mais les applications montrent qu'avec de bons choix de T_n cette méthode donne de très bons résultats.

2. Dans un fichier `test1.cpp`, écrire des fonctions (ou des λ-fonctions, ou des classes) pour `phi`, `T` et `Y` ainsi choisis avec `E` valant `double`.

3. Dans le même fichier `test1.cpp`, écrire un programme complet qui utilise les fonctions précédentes et `recuit_simule` pour estimer l'un des deux minima[3] de ϕ. Vous afficherez les estimations dans le terminal.

4. Dans le même fichier `test1.cpp`, compléter le programme pour estimer la moyenne et la variance empiriques sur $K = 100$ échantillons de l'estimation $|X_N|$ pour $N = 200$ et $N = 400$. Nous vous rappelons que la moyenne et la variance empiriques sur K échantillons d'une v.a. Z est définie comme :

$$M_K = \frac{1}{K}\sum_{k=1}^{K} Z_{(k)}$$
$$V_K = \frac{1}{K}\sum_{k=1}^{K} Z_{(k)}^2 - M_K^2$$

où les $Z_{(k)}$ sont des réalisations indépendantes de Z. *Indication :* il s'agit de faire appel K fois à `recuit_simule` et de moyenner sur les résultats obtenus.

Le problème du voyageur de commerce. Étant données M villes numérotées de 0 à $M-1$, on se donne toutes les distances $d_{i,j}$ entre deux villes. On souhaite trouver un itinéraire $x = (x_0, \ldots, x_{M-1})$ qui passe par chaque ville une unique fois et minimise la distance totale parcourue :

$$\phi(x) = \sum_{k=0}^{M-2} d_{x_k, x_{k+1}} \tag{1}$$

Pour implémenter cela en C++, il faut décrire à la fois la fonction de distance et l'itinéraire. Pour l'itinéraire, on prendra pour `E` la classe `std::vector<int>` avec des vecteurs de taille M qui contiennent chaque nombre entre 0 et $M-1$ une unique fois : la case i décrit le numéro de la i-ème ville visitée.

Pour la fonction à minimiser, on introduit :

```
class DistanceTotale {
protected:
        int M;
        double * dist; // dist[i+M*j] donne la distance entre les villes i et j.
public:
        DistanceTotale(int M0);//alloue le tableau et met les distances à zéro.
        //Accesseur et mutateur
        double distance(int i,int j) const;
        double & distance(int i,int j);
        double operator()(const std::vector<int> & x) const;
};
```

5. Dans des fichiers `voyageur.hpp` et `voyageur.cpp`, écrire les codes du constructeur, de l'accesseur et du mutateur.

6. Écrire le code de `operator()` qui calcule la distance totale parcourue pour la trajectoire `x`.

3. ϕ atteint ses minima en -2 et 2.

7. (*rule of three*) Écrire les codes des trois méthodes nécessaires à une bonne gestion de la mémoire.

Optimisation par recuit simulé sur 50 villes. Afin d'éviter d'avoir à charger un fichier de données pour ce TP noté, nous prendrons comme distance entre les $M = 50$ villes $d_{i,j} = \cos^2(4i+3j)$. Pour `Y`, nous utiliserons le modèle de fonction à ajouter dans `voyageur.hpp` :

```
template <class RNG>
std::vector<int> Y_transposition(const std::vector<int> & x, RNG & G) {
        std::vector<int> y(x);
        std::uniform_int_distribution<int> U(0,y.size()-1);
        std::swap(y[U(G)],y[U(G)]);
        return y;
}
```

8. Dans un fichier `test_commerce.cpp`, déclarer un objet de type `DistanceTotale` pour $M = 50$ villes, remplir les distances grâce au mutateur avec la formule précédente. Déclarer également une trajectoire initiale qui parcourt les villes dans leur ordre de numérotation.

9. Dans le même fichier, en commentaire de la définition de `Y_transposition`, expliquer en commentaire du code fourni ce que génère cette fonction.

10. Dans le même fichier, en utilisant $T_n = 10 \cdot (0.95)^n$ et `recuit_simule` pour $N = 100000$ itérations, afficher dans le terminal à la fois la longueur de la trajectoire initiale, la trajectoire obtenue après optimisation et la longueur correspondante [4].

4. Vous devriez passer d'autour de 24 à autour de 2 pour les longueurs.