Tp de C++.
Résolution du problème de Monty Hall par simulation de Monte-Carlo

V.Doyeux

Septembre 2012

1 Presentation du problème

Nous allons dans ce TP résoudre un petit *casse tête* de probablités. Il s'agit d'un problème inspiré par un jeu télé américain dont le présentateur s'appellait Monty Hall. Vous trouverez tous les détails, l'historique et la résolution formelle de ce problème sur la page wikipedia problème de Monty Hall et ses références. Voici l'énoncé original du problème :
Supposez que vous êtes sur le plateau d'un jeu télévisé, face à trois portes et que vous devez choisir d'en ouvrir une seule, en sachant que derrière l'une d'elles se trouve une voiture et derrière les deux autres des chèvres. Vous choisissez une porte, disons la numéro 1, et le présentateur, qui lui sait ce qu'il y a derrière chaque porte, ouvre une autre porte, disons la numéro 3, porte qui une fois ouverte découvre une chèvre. Il vous demande alors : « désirez-vous ouvrir la porte numéro 2 ? ». À votre avis, est-ce à votre avantage de changer de choix et d'ouvrir la porte 2 plutôt que la porte 1 initialement choisie ?

Le problème se déroule donc en 2 temps :
- Choisir une des 3 portes
- Une chèvre a été éliminée d'une des deux portes non selectionnée. On a le choix de garder notre première sélection ou de la changer.

Le problème consiste à savoir ce qu'il faut faire au moment du second choix. Doit on garder ou changer notre sélection initiale ?

La première idée qui vient chez la plupart des gens est qu'il est exactement équivalent de changer ou non notre sélection initiale, en effet, il y a une chèvre et une voiture derrière les portes, donc les chances de gagner sont 50/50.

Si on regarde le problème dans son ensemble, on obtient une réponse différente. Pour gagner, il y a deux possibilités :

1) on tombe initialement sur la voiture et on **garde** notre choix initial
2) on tombe initialement sur une chèvre et on **change** notre choix initial

Or il y a 1/3 chance de tomber initialement sur la voiture et 2/3 de tomber sur une chèvre. Donc en changeant tout le temps notre choix initial on a 2/3 de chance d'avoir la voiture. Cette conclusion a fait débat pendant quelques temps dans la communauté des mathématiciens et amateurs de jeux de probabilité (voir l'historique sur wikipedia et ses références). Aujourd'hui, la véracité de la solution "$2/3 - 1/3$" est communément admise et vous pouvez trouver une démonstration détaillée sur internet.

Une des validations possible de la solution, est de faire la simulation d'un très grand nombre de jeux en choisissant l'une ou l'autre des stratégies et de compter le pourcentage de victoires, soit la probabilité de gagner lorsqu'on choisit une stratégie. Nous proposons dans ce TP de faire la simulation du problème de Monty Hall et de déterminer la probabilité de victoire suivant la stratégie que l'on utilise. Pour cela, on écrira une classe nous permettant de jouer au jeu. Puis nous ferons un très grand nombre de tirage en utilisant les deux stratégies. Les méthodes de simulations dans lesquelles on fait intervenir du hasard sont regroupées sous le terme de simulations de Monte-Carlo. Ce terme est très général et il est fort probable que vous l'entendiez dans des contextes très différents de simulation de problèmes physiques. Par exemple en physique nucléaire où un code de calcul de Monte Carlo a été développé lors du projet Manathan (MCNP) et est aujourd'hui encore très utilisé dans d'autres domaines (radioprotection, dosimétrie, imagerie médicale, criticité, instrumentation). Vous trouvez des simulations de Monte Carlo dans bien d'autres domaines (physique statistique, jeux d'échecs, finance etc ...)

2 Création d'une classe permettant de jouer

Nous allons créer la classe `Jeu` qui nous permettra de simuler le jeu de Monty Hall pour un seul joueur. Dans un fichier `jeu.hpp`, on définit la classe suivante :

```
#include <cstdlib>

class Jeu {

public :

  // constructeur
  Jeu() { nb_victoires = 0; }

  void initialiser();
  void premiereSelection(int choix);
  bool choixFinal(bool changer);

  int nombreVictoires()
  {return nb_victoires;}

private :
```

```
  bool PremierChoix[3];
  bool SecondChoix[2];
  int nb_victoires;
};
```

L'idée est que pour l'utilisateur de notre classe, une séquence de jeu se résume à appeller successivement les trois fonctions `initialiser`, `premiereSelection` et `choixFinal`. La première fonction servira à tirer les 3 choix aléatoirement. Dans la seconde fonction, on sélectionne le premier choix que l'on fait. Enfin, dans la dernière fonction, on dit au jeu si l'on décide de changer le choix fait précedemment ou non. Lorsqu'on veut voir le nombre de victoires que l'on a fait depuis le début, il nous suffit d'appeller `nombreVictoires`. En résumé, l'utilisation de notre classe peut se faire comme ceci par exemple si l'on veut faire 1000 tirages en ne changeant jamais notre premier choix :

```
    Jeu unjeu;

    for (int i=0; i<1000; i++)
    {
      unjeu.initialiser();
      unjeu.premiereSelection(1);
      unjeu.choixFinal(false);
    }

    std::cout<<"Vous avez gagne "<<jeu.nombreVictoires()<<" fois.";
```

On notera que l'utilisateur de la classe n'a pas accès directement au contenu des rideaux (`PremierChoix` et `SecondChoix`) qui sont privés. De même, le nombre de victoire n'est accessible que par l'intermédiaire de la fonction `nombreVictoires()`, l'utilisateur peut donc connaitre son nombre de victoires mais pas le modifier !

Voyons maintenant comment doit marcher la classe `Jeu`. Le principe, est que les choix possibles de la première et la seconde étape sont modélisés par des tableaux de booléens. Chaque rideau est un élément du tableau. Dans la première étape, on a le choix entre 3 rideaux donc `PremierChoix` contient 3 éléments et le second en contient 2. Le contenu d'un rideau n'a que deux choix possibles : *chèvre* ou *voiture*, c'est pourquoi le type booléen est bien adapté. On choisit de représenter *chèvre* = `false` et *voiture* = `true`.

- La méthode `initialiser()` fonctionne comme ceci :
  - initialiser `PremierChoix` avec 3 booléens
  - initialiser `SecondChoix` avec 2 booléens
  - tirer au hasard une configuration pour remplir `PremierChoix` (3 configurations possibles : `(true,false,false)`, `(false,true,false)` ou `(false,false,true)`
- La méthode `premiereSelection(int choix)` consiste a remplir `SecondChoix` en fonction du choix que l'utilisateur a fait. Le choix passé en argument est l'indice du vecteur `PremierChoix` que l'utilisateur veut garder. `premiereSelection(int`

`choix)` se comporte donc comme ceci :
  - remplire le premier élément de `SecondChoix` avec la valeur `PremierChoix` à l'indice `choix`
  - si une des deux autres valeurs de `PremierChoix` contient `true`, alors mettre le second élément de `SecondChoix` à `true`, sinon le mettre à `false`
- Enfin, la méthode `choixFinal(bool changer)` consiste à regarder si l'utilisateur veut changer (`changer=true`) ou garder (`changer=false`) son premier choix. Puis si son choix final contient `true`, on augmente `nb_victoires` de 1.

3 Utilisation de la classe Jeu

3.1 Jeu interactif

Servons nous maintenant de la classe Jeu. En vous inspirant de son modèle d'utilisation (cf section précédente), créez une application qui simule le jeu de façon interactive, c'est à dire en laissant l'utilisateur rentrer au clavier les deux choix qu'il a à faire. Au final, pour l'utilisateur, votre programme doit ressembler à quelque chose comme ceci :

```
Tirage aleatoire fait. Choisissez un rideau (0, 1, 2) :
> 2
On garde le rideau 2, elimination d une chevre dans un des autres rideau.
Voulez vous changer votre choix ? (o,n)
> n
Vous avez gagne. Votre score est de 10 victoires sur 30. Continuer a jouer ? (o,n)
>
```

3.2 Simulation

Nous allons maintenant déterminer la probabilité de gagner suivant la stratégie que l'on utilise. L'idée est de faire un très grand nombre de tirage et de comparer les scores de 3 stratégies possibles pour voir si statistiquement, une stratégie est meilleure que les autres. Le premier choix n'a pas d'importance. C'est le choix de garder ou non son choix initial qui est le sujet du probleme de Monty Hall. Les trois différentes stratégies sont les suivante :

1) toujours garder son choix initial
2) toujours changer son choix initial
3) choisir aléatoirement de garder ou changer

Vous créerez donc une application, créant 3 jeux en même temps :

```
Jeu jeu1;
Jeu jeu2;
Jeu jeu3;
```

Puis faites un grand nombre de tirage en forçant la stratégie de chacun des joueurs. Au final, comparez les trois scores, affichez les en pourcentage, et déterminez la probabilité de

gagner suivant la stratégie utilisée.

En bonus, si vous avez du temps, vérifiez que le pourcentage de réussite converge vers une valeur fixe quand le nombre de tirage augmente. Pour cela, modifiez votre programme pour que le jeu enregistre la valeur des pourcentages de réussite à chaque nouveau tirage dans un fichier. Enfin, vous pourrez tracer la variation de chaque pourcentage en fonction du nombre de tirage. Qu'obtenez vous si vous faites 10 tirages ? 1000000 tirages ?

4 Annexe

4.1 Tirer un nombre aléatoire

Pour tirer un nombre aléatoire, il faut inclure l'entête `cstdlib` et mettre la ligne suivante au début du main :

```
srand ( time(NULL) );
```

Puis chaque fois que l'on veut un entier aléatoire compris entre 0 et N-1, on appelle l'instruction suivante :

```
(int) ( rand() % N );
```