L'une des méthodes les plus utilisées actuellement est **la méthode des congruences linéaires**.

Là encore, des grands mots pour quelque chose de pas si compliqué que ça ! Décortiquons ensemble cette expression barbare.
Congruence : dans notre cas, cela se réfère à ce qui a rapport avec le reste de la division entière (vous savez, le signe '%' qui ne sert à rien ?).
Linéaire : en simplifiant, cela revient à dire que l'on ne fait que des opérations de type addition et multiplication. Le type d'opération "mise au carré" et "logarithme" n'existe pas (ce n'est pas très grave si vous ne comprenez pas cette phrase, c'est juste pour la culture).

De façon plus concrète, voici une formule permettant de calculer une suite de nombres aléatoires :

Code : Autre

```
nouveau_nombre= (a*ancien_nombre + b) % m
```

Petit rappel important pour la suite, "*a%n*" signifie le **reste** de la division de *a* par *n*. Par exemple 11%3=2 : 11 divisé par 3 égal 3, il reste 2 ; et 12%3=0 : 12 divisé par 3 égal 4, il reste 0.

Pour traduire cela avec des mots, ça donne : pour trouver le nouveau nombre aléatoire, il faut :
1. Prendre celui que l'on vient de calculer, le multiplier par a puis ajouter b ;
2. Si le nombre obtenu est plus grand que m, lui enlever m jusqu'à ce qu'il soit plus petit.

Rien ne vous choque ? Pas un petit problème de variable non définie ? Ah si quand même ! La formule dit qu'il faut prendre le nombre que l'on vient de calculer, mais si on n'a encore rien calculé, on prend quoi ? C'est là un sujet épineux dont on pourrait débattre longtemps, mais nous n'allons pas nous étendre sur ce sujet. En général, comme première valeur, on prend le timestamp ; cette première valeur s'appelle **la graine** (**seed** en anglais).

Cela vous explique pourquoi, en C, vous devez initialiser rand par srand (qui doit vouloir dire seedrand), en passant en argument le timestamp actuel ! Si vous ne le faites pas, la suite de nombres générée est toujours la même... c'est fâcheux pour un jeu de hasard...

Dans des langages plus évolués, comme Java ou Python, on n'a pas à faire cette initialisation, le langage s'en charge lui-même !

Un premier code

Maintenant que vous avez la théorie, je vous propose de coder votre propre générateur aléatoire. L'exercice consiste à écrire un programme (tout dans le main, on s'embêtera plus tard avec des jolies structures) qui affiche 15 nombres aléatoires.
Vous devez réfléchir comment coder la formule que j'ai donnée au-dessus (pour les valeurs de a, b et m, mettez un peu au hasard pour l'instant, en bidouillant pour que ça marche mieux si besoin, mais gardez-les relativement petits).

Attention, prêts ? C'est parti !

Secret (cliquez pour afficher)

Code : C++

```
#include <iostream>
#include <ctime>
using namespace std;

main(){
 unsigned int a=3,b=3,m=5;                // Définition des valeurs
 unsigned int nombre = time(NULL);     // Définition de la graine
```

```
        for(int i=0;i<15;i++){
         nombre = (a*nombre+ b) % m;   // Calcul effectif
         cout<<nombre<<" ";             // Affichage
        }
       }
```

Voilà, je pense que ça ne présentait pas de grandes difficultés ; si oui, déroulez le programme sur papier (par exemple 1e boucle : nombre=10, nombre*a=30..., 2e boucle : nombre =...,) en ayant la "formule" donnée plus haut sous les yeux. Pour ceux qui ont utilisé un tableau pour stocker toutes les valeurs, ce n'était pas utile, et je dirais même une erreur. Là, ça va, il n'y avait que 15 nombres à générer. Mais imaginez deux secondes que vous produisiez des nombres pour une application qui demande des milliards de nombres aléatoires ? La mémoire ne sera jamais assez grande pour conserver tous ces nombres !

Maximisons la période !

Maintenant, analysons la sortie de ce programme :

Code : Console

```
4 0 3 2 4 0 3 2 4 0 3 2 4 0 3
```

J'avais pris m = 5, donc logiquement, toutes mes valeurs sont entre 0 et 4 (inclus).
On peut déjà constater 2 choses :

- les nombres suivent une certaine séquence qui se répète (ici 4,0,3,2). Quelles que soient les valeurs que l'on prend pour a, b et m, nous aurons toujours ce comportement ; cela peut paraître gênant, vu que le générateur n'est du coup plus du tout aléatoire. En pratique m est très grand, donc la séquence est également très longue ;
- il manque un nombre dans la séquence : 1 ; le générateur est donc clairement faussé ! En effet, pour un générateur produisant des nombres entre 0 et 4, on doit s'attendre à une moyenne de 2 ; ici, si on fait le calcul, on a (0+2+3+4)/4 = 2.25 comme moyenne !

Cela m'amène à vous parler de la **période**. La période correspond à la longueur de la séquence générée; dans notre exemple, la période est de 4. Evidemment, plus cette période est grande, meilleur est le générateur.
Dans le cas de cette méthode, on dit que la **période est maximale si elle est égale à m**. Vous remarquerez facilement qu'avec cette méthode, chaque nombre n'est "visité" qu'une fois au plus ; si le calcul "repasse" par le même nombre, cela signifie qu'on a fait le tour de la séquence.
Avoir une période maximale signifie donc que **tous** les nombres entre 0 et m-1 (inclus) sont "visités" **une seule fois**. Si la période est maximale, la moyenne des nombres sortis sera de (m-1)/2 (je ne ferai pas le calcul précis ici, mais on peut voir "intuitivement" que ça marche), on a donc un bon argument en faveur de la validité du générateur.

> Ceci est un des aspects fondamentaux de la génération algorithmique de nombres aléatoires. Si vous devez retenir quelque chose, c'est **ça** ! Les formules que je donne, vous pourrez les utiliser sans rien comprendre à tout ça, mais dans ce cas-là, rien ne sert de lire ce tuto. Les fonctions rand et compagnie feront très bien l'affaire.

Au passage, vous remarquerez que si la première valeur générée est 1, toutes les valeurs suivantes seront les mêmes (= 1) ! On voit donc clairement ici une limite de ce générateur.

Ok ! Donc pour avoir une période maximale, il suffit que je teste des valeurs a, b et m et que je regarde si tous les chiffres sont "visités" ?

On pourrait faire comme ça... Mais ça serait trèèèèèèès long si m est très grand (ce qui est le cas en pratique). Heureusement des gens très intelligents ont réfléchi au problème et nous ont donné un **critère** pour que la période soit maximale. Ce critère s'écrit sous la forme de trois conditions.

La période est maximale si et seulement si :

- b et m sont premiers entre eux ;

- si m est un multiple de 4, alors $a\%4 = 1$;
- pour tous les nombres premiers p diviseurs de m, on a $a\%p = 1$.

Rappel :

- Nombres premiers : tous les nombres entiers qui ne peuvent pas se décomposer en facteurs autres que le nombre lui-même ou 1. Par exemple, on peut décomposer 15 (= 3*5), donc il n'est pas premier. Par contre 13 est premier, on ne peut pas trouver 2 entiers a et b (autres que 1 et 13) tels que a*b=13.
- Nombres premiers entre eux : soient deux nombres entiers x et y. On peut décomposer ces nombres en facteurs : x=1*d*e*f*... et y=1*n*o*p*... On dit que x et y sont premiers entre eux s'ils n'ont aucun facteur en commun autre que 1. Par exemple 24 = 1*3*2*2*2 est premier avec 35 = 1*5*7, ils n'ont que 1 en commun. En revanche, 35 = 1*5*7 n'est pas premier avec 15 = 1*5*3 : ils ont 5 comme facteur commun.

Mais la 2e condition, elle ne sert à rien ! Ce n'est qu'un cas particulier de la 3e !

Erreur ! La troisième nous parle des diviseurs premiers de m. La deuxième est effective si 4 est un diviseur de m. Mais 4 n'est pas premier (eh oui 4=2*2).

Si vous n'avez absolument jamais entendu parler de nombres premiers, ne vous attardez pas dessus, retenez juste qu'il suffit de vérifier certaines conditions sur a, b et m pour avoir une période maximale.
Pour les autres qui en savent un peu plus, essayez vraiment de voir à quoi ces conditions correspondent, et tentez de montrer que les valeurs que l'on a prises au-dessus ne donnent pas une période maximale.
Pour les très forts en arithmétique, vous pouvez essayer de démontrer ce résultat (sans aller voir la démo sur Wikipédia ou équivalent, cela va sans dire...).

Si vous voulez voir comment vérifier effectivement ce critère, jetez un coup d'oeil au QCM, la correction est assez complète.

Maintenant, nous pouvons former un générateur à période maximale. Par exemple les valeurs : $a=5$, $b=3$, $m=8$ vérifient les 3 conditions ci-dessus et produisent la sortie suivante (c'est exactement le même programme, il suffit de changer les valeurs dans la première ligne du main) :

Code : Console

```
0 3 2 5 4 7 6 1 0 3 2 5 4 7 6 1
```

Tous les chiffres sont bien parcourus, la période est maximale, YOUHOU !!! Nous avons notre premier générateur aléatoire efficace !!!

Mais attendez, ce n'est pas fini ! Les nombres que nous utilisons là sont très petits. Quand on veut utiliser des nombres plus grands, d'autres problèmes apparaissent.

Minimisons la corrélation !

Encore un mot compliqué, mais comme d'hab, une signification pas bien sorcière : la corrélation entre deux nombres aléatoires peut être vue comme la capacité, connaissant un nombre, de déterminer le second.

Un exemple pour comprendre : on prend $a=2^{30}+1$, $b=3$ et $m=2^{31}$. La sortie donne un truc du genre :

Code : Console

```
1073741832 1073741835 14 17 1073741844 1073741847 26 29 1073741856 1073741859 38 41
```

Si on fait attention, on remarque qu'on passe d'une valeur à l'autre en ajoutant 3 (écrivez à la main ce que ça donne, vous verrez). Les nombres sont donc fortement corrélés.

Il va donc falloir encore une fois trouver des critères qui nous permettent d'avoir une faible corrélation. Et encore une fois, on dit merci aux matheux :

- m doit être le plus grand possible (en général, on veut profiter de tous les entiers disponibles, donc on prend 2^{32}-1 ou 2^{31}). C'est un truc qui nous arrange bien car cela permet d'avoir une période très longue ;
- b doit être "petit" par rapport à a et m ;
- a doit être proche de la racine carrée de m.

Dans l'exemple que j'ai pris juste au-dessus ($a=2^{30}+1$, $b=3$ et $m=2^{31}$), a est très éloigné de la racine de m ; grâce à ce critère, nous aurions pu voir immédiatement que ce générateur était pourri !

Si on prend $m=2^{32}-1$, il n'est pas nécessaire d'effectuer le modulo explicitement car ce nombre correspond au plus grand entier non signé en C (sur une machine 32 bits). Le modulo se fera donc de lui-même ! Cependant, $2^{32}-1$ n'est pas forcément très pratique à utiliser, et on préfère des fois prendre 2^{31} qui n'a qu'un seul diviseur (2) ce qui simplifie l'étude du générateur. Nous garderons donc le `%m` ; de plus la définition des entiers varie d'une plate-forme à l'autre, on ne peut pas supposer froidement que les entiers sont codés sur 32 bits !

Quelques bons générateurs

Voilà, maintenant, vous savez pratiquement tout sur les générateurs congruentiels linéaires.
Donc pour résumer, pour avoir un bon générateur, il faut avoir une période maximale **et** il faut que les coefficients respectent quelques règles. Si vous oubliez une des conditions, le générateur sera médiocre.

Je vais terminer cette partie en vous invitant à aller voir quelques "bons" générateurs utilisés par certaines librairies (j'ai choisi l'article anglais car l'article français est beaucoup moins bien fait) : article anglais sur les générateurs congruentiels linéaires sur Wikipédia (le coefficient que j'ai appelé 'b' est appelé 'c' dans cet article).

Une classe Aleatoire

Dans cette partie, nous allons mettre en pratique ce que nous avons appris. Comme je l'annonce depuis le début, nous allons tous ensemble construire cette classe Aleatoire.

Je vais être assez spécial vu que je vais vous demander de faire un générateur "paramétrable". Je veux que n'importe qui puisse définir un générateur congruentiel linéaire en testant différentes valeurs. Je veux en plus que cette classe soit indépendante, dans la mesure où elle devra elle-même initialiser la graine.

Essayez de la faire par vous-mêmes, il n'y a aucun "piège" particulier. Il s'agit juste de mettre dans une classe le code que l'on a tapé dans le main un peu plus haut.

Voici une correction possible.

Le fichier Aleatoire.h :

Code : C++

```
#ifndef _ALEATOIRE_H
#define _ALEATOIRE_H
#include <ctime>
class Aleatoire{
    private :
        unsigned int m_a,m_b,m_m; // Les coefficients du générateur
        unsigned int m_nombre;  // le nombre actuel généré

    public :
        Aleatoire(unsigned int a=16807,
                  unsigned int b=0,
                  unsigned int m=0x7FFFFFFF);
            // par défaut, le générateur prendra les valeurs du
            // compilateur Apple CarbonLib
        int generer();
};
```

```
#endif
```

Et le fichier Aleatoire.cpp :

Code : C++

```
#include "aleatoire.h"

Aleatoire::Aleatoire(unsigned int a,
                     unsigned int b,
                     unsigned int m) : m_a(a),m_b(b),m_m(m){
    m_nombre = time(NULL);
}

int Aleatoire::generer(){
    m_nombre = (m_a*m_nombre + m_b) % m_m;
    return m_nombre;
}
```

Et voilà ! Une classe toute simple, prête à l'emploi !

Un petit main pour l'utiliser :

Code : C++

```
#include <cstdlib>
#include <iostream>
#include "aleatoire.h"
using namespace std;

int main(int argc, char *argv[])
{
    Aleatoire a(3,3,5);
    for(int i=0;i<15;i++){
        cout << a.generer()<<" ";
    }
    return 0;
}
```

Maintenant, vous pouvez tester très facilement les différents paramètres grâce au constructeur. Si vous voulez un générateur fonctionnel, laissez les paramètres par défaut.
Voilà, le tuto est fini ; voici quelques pistes pour approfondir le sujet !

Pour commencer, vous pouvez consulter l'article sur les générateurs de nombres pseudo-aléatoires.

Vous pouvez ensuite améliorer la classe Aleatoire (des pistes sur ce tuto):

- créer une méthode generer(int min,int max) qui génère un nombre entre min et max ;
- créer une méthode double genererDouble() qui génère un nombre réel entre 0 et 1 ;
- vérifier que les coefficients donnent bien une période maximale (attention, ça demande un peu plus de temps que les deux autres améliorations).

Si vous êtes en Terminale S (ou que vous savez ce qu'est une loi), vous pouvez aussi regarder comment générer une loi aléatoire comme la loi normale, binomiale, de Poisson... et aussi une loi quelconque ! (Pour ça, voir la méthode du rejet et la méthode de l'inverse.) Ces méthodes se basent sur la génération de nombres qu'on fait ici (c'est-à-dire la loi uniforme), vous verrez. C'est par contre un peu plus poussé en maths (une piste ici) !

Si vous voulez aller vraiment plus loin (en dessous de bac +1 en maths, s'abstenir), vous pouvez essayer de comprendre la méthode de Mersen Twister qui permet d'avoir une génération encore meilleure.

Si vous voulez des applications de ce que l'on a fait, vous pouvez aller voir mon tuto sur la méthode de Monte Carlo qui est très utilisée en physique nucléaire.

L'aléatoire en C et C++ : se servir de rand

Envie de coder un Yams ? Besoin de simuler un phénomène physique ? Enfin bref, vous voulez utiliser des nombres aléatoires dans votre programme en C ?

Ce tutoriel est là pour vous !

Après avoir décrypté la fonction rand, nous verrons comment générer simplement des nombres aléatoires, entiers ou en virgule flottante. Je vous montrerai ensuite une technique pour générer une suite d'entiers sans doublons, utile pour simuler des cartes tirées dans un paquet par exemple.

Ce tutoriel a été écrit pour le langage C, mais si vous utilisez le langage C++, ce tutoriel est encore valable. Il suffit juste de modifier les "include" : vous devez retirer le ".h" à la fin du nom de la bibliothèque et insérer un "c" au début. Par exemple `#include <stdlib.h>` deviendra `#include <cstdlib>` ; et `#include <time.h>` deviendra `#include <ctime>` .

Le cœur de l'aléatoire en C : rand
Des nombres pas si aléatoires que ça !

Avant de commencer, il faut que vous ayez en tête que l'on ne peut pas avoir de nombres parfaitement aléatoires avec la méthode utilisée ici. Vous aurez l'**impression** qu'ils le sont, mais en réalité, il ne s'agit que de nombres "pseudo-aléatoires". Si vous voulez en savoir plus, je vous renvoie à l'article traitant de ce sujet. Cela n'est en général pas très gênant, sauf si vous faites de la cryptographie ou d'autres applications où la sécurité est primordiale, car on pourra "deviner" les nombres qui vont sortir. Dans la suite, j'emploierai "aléatoire" à la place de "pseudo-aléatoire" par abus de langage.

Votre première expérience avec l'aléatoire

Vous l'attendez tous, alors voici la fonction qui permet de générer des nombres aléatoires :

Code : C - La fonction rand

```
int rand(void)
```

Elle fait partie de la bibliothèque `stdlib` , vous devrez donc l'inclure (`"stdlib.h"` si vous êtes en C, `"cstdlib"` en C++).

"rand" est la troncature du mot "random" qui en anglais veut tout simplement dire "aléatoire".

Dans la foulée, un exemple pour s'en servir ; le code suivant affiche cinq nombres aléatoires :

Code : C - Génération de 5 nombres aléatoires

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h> //Ne pas oublier d'inclure le fichier time.h

int main(void){
 int i = 0;
 int nombre_aleatoire = 0;
 for(i=0; i<5; i++){
  nombre_aleatoire = rand();
  printf("%d ",nombre_aleatoire);
 }
 return 0;
}
```

Comme vous le voyez, chaque nouvel appel à `rand` génère un nouveau nombre aléatoire.

Mais elle est nulle ta fonction ! J'ai relancé le programme et j'ai eu exactement les mêmes nombres !

Oui, cela illustre bien ce que je vous disais au début : `rand` va en réalité toujours renvoyer la même séquence de nombres. En l'occurrence pour le programme ci-dessus, on aura :

Code : Console

```
41 18467 6334 26500 19169
```

Donc à priori, pas du tout aléatoire ! En pratique, cette séquence est très (très) longue, c'est ce qui va nous permettre de "faire croire" que les nombres sont tirés au hasard. Mais si on part toujours du même point, ce n'est pas intéressant. On va donc "dire" à rand de se placer dès le départ à une autre position dans la séquence. Mais là, on se mort un peu la queue : il faut choisir "aléatoirement" la position initiale.

Le plus courant est d'utiliser la fonction `int time(int*)` qui est définie dans `time.h` ; cette fonction donne le nombre de secondes écoulées depuis le premier janvier 1970 (il est très rare qu'on lance le programme deux fois dans la même seconde). On ne se servira pas du paramètre que l'on mettra donc à `NULL`, je vous renvoie à la doc officielle si ça vous intéresse. On l'appellera donc comme ceci : `time(NULL);` .

Il me reste à vous dévoiler comment "dire à rand de se mettre à telle position" : on le fait grâce à la fonction `void srand(int start)` , où `start` indiquera où se placer dans la séquence.

Le code ci-dessus devient alors :

Code : C - Génération de 5 nombres aléatoires

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h> //Ne pas oublier d'inclure le fichier time.h

int main(void){
 int i = 0;
 int nombre_aleatoire = 0;
 srand(time(NULL)); // initialisation de rand
 for(i=0; i<5; i++){
  nombre_aleatoire = rand();
  printf("%d ",nombre_aleatoire);
 }
 return 0;
}
```

Comme vous le voyez, `srand` n'a été appelée qu'une seule fois. En général, il est inutile de la rappeler, c'est `rand` qui se chargera du boulot ensuite.

Restez entre les bornes

Avec des entiers

Je vous vois déjà râler : "Moi, je veux faire un Yams, mais rand me renvoie que des chiffres super grands, je pourrais jamais simuler un dé !"

Nous allons voir ici comment résoudre ce problème en écrivant la fonction `int rand_a_b(int a, int b)` qui renvoie un entier au hasard entre a (inclus) et b (exclu).

Vous ne vous en rendez peut-être pas compte, mais c'est déjà le comportement de `rand` : elle renvoie des entiers entre 0 et `RAND_MAX` , qui est une constante définie dans `stdlib.h` . La valeur de `RAND_MAX` peut varier suivant les compilateurs, mais elle est forcément d'au moins 32767.

Pour revenir à notre problème, on va commencer par générer un nombre entre 0 et un nombre `c`. Pour cela, nous allons utiliser l'opérateur '%' qui renvoie le reste de la division entière ; effectivement, je vous rappelle que le reste de la division de `x` par `c` est toujours compris entre `0` et `c` (exclu). Donc nous avons juste à faire :

Code : Autre

```
rand()%c
```

Maintenant, mettons que vous vouliez tirer un entier dans l'intervalle [a,b[(c'est à dire a inclus et b exclu), il vous suffit de tirer un nombre entre 0 et la longueur de cet intervalle (qui est égale à b-a), puis d'ajouter a. Le nombre sera bien dans [a,b[. Au final, la fonction ressemblera à :

Code : Autre

```
// On suppose a<b
int rand_a_b(int a, int b){
    return rand()%(b-a) +a;
}
```

Un tirage vraiment uniforme ?

Le standard C nous garantit que la répartition des valeurs de rand() entre 0 et `RAND_MAX` est uniforme, c'est à dire qu'on a exactement la même probabilité de tomber sur l'un ou l'autre des entiers contenus dans cet intervalle. Et on doit se poser la question de savoir si cette propriété est respectée par rand_a_b, ce qui n'est pas évident.

Effectivement, pour prendre un exemple, si `RAND_MAX=5`, et qu'on cherche à tirer à pile ou face (donc, soit 0 soit 1) :

- 0 sera obtenu à partir de 0,2 et 4, donc aura une probabilité d'apparition de 3/5=0,6 ;
- 1 sera obtenu à partir de 1 et 3, donc aura une probabilité d'apparition de 2/5=0,4.

La répartition n'est donc pas uniforme.

Bon, alors tout ce qu'on a lu, ça n'a servi à rien ?

Eh bien non. Car ici, on a pris un `RAND_MAX` très petit. Rappelez-vous, il est d'au moins 32767. Dans notre exemple, on aura donc 32767/2 = 16383 apparitions du 1, et 16383+1 apparitions du 0. Je vous laisse faire le calcul pour vous apercevoir que la différence de fréquence d'apparition est ridicule. Sans faire le calcul précis, on se rend bien compte que, tant que la longueur de l'intervalle est "petite" devant `RAND_MAX`, on peut faire l'approximation que rand_a_b est bien uniforme (c'est en réalité un peu plus complexe qu'une simple histoire de taille d'intervalles, il y a des histoires de divisibilité qui entrent en jeu également ... avis aux fous furieux de l'arithmétique).

La question est après de savoir ce que veut dire "petit". Tout dépend des applications que vous voulez en faire. Si vous faites des simulations physiques/financières, il faut étudier le cas assez précisément et trouver des solutions adaptées. Si vous implémentez une animation simplement esthétique utilisant l'aléatoire (par exemple, une chute de flocons neige), est-ce bien la peine de se prendre la tête ?

Et les flottants ?

Pour générer un nombre entre deux nombres réels a et b (représentés par des `float` et des `double` en C), c'est à peu près la même idée. On a vu que `rand()` était compris entre 0 et `RAND_MAX`. Donc mathématiquement parlant, `rand()/RAND_MAX` est compris entre 0 et 1. Il suffit maintenant de multiplier par `(b-a)` puis d'ajouter `a` pour avoir un nombre compris entre a et b.

Il y a tout de même un petit point technique que je me sens obligé de souligner : par "défaut" en C, la division est entière, c'est à dire que `a/b` est en réalité égal au nombre (entier) maximum de fois que l'on peut "mettre" b dans a. Dans notre cas, vu que `rand()` est inférieur ou égal à `RAND_MAX`, `rand()/RAND_MAX` vaudra 0 dans la plupart des cas, et 1 seulement si `rand()=RAND_MAX`.

On doit donc effectuer une conversion d'une des deux opérandes ; cela forcera la division à se faire comme on l'espère. Pour ce

faire, il suffit d'ajouter `(double)` devant la variable que l'on veut convertir.

> En réalité, vous ne générerez pas toutes les valeurs entre a et b ; la plus petite distance entre deux nombres tirés sera de `(b-a)/RAND_MAX`. Si (b-a) est petit devant `RAND_MAX`, pas trop de problèmes. Sinon, il faudra trouver d'autres méthodes.

Voilà donc la fonction qui génère des nombres flottants entre 2 bornes :

Code : Autre

```
double frand_a_b(double a, double b){
    return ( rand()/(double)RAND_MAX ) * (b-a) + a;
}
```

La génération sans doublons

On est parfois amené à devoir générer une suite de nombres sans doublons ; c'est à dire que si un certain nombre a déjà été tiré, il n'est pas possible de le tirer à nouveau. Les exemples sont multiples : tirer des cartes dans un tas, établir un ordre de passage aléatoire à un examen pour une liste d'étudiants, faire un diaporama aléatoire, ... bref les applications ne manquent pas.

Pour expliquer l'algo, je générerai une liste sans doublons d'entiers de 0 à `MAX` ; la modification à apporter pour avoir une génération entre `MIN` et `MAX` étant minime, elle sera apportée tout à la fin.

Une première idée

La première idée, naïve, qu'on pourrait avoir serait de tirer un nombre au hasard, regarder si ce nombre a déjà été tiré, et en retirer un autre tant que ce nombre n'a pas été tiré.

Mais comment peut-on savoir si un nombre a déjà été tiré ?

On peut procéder ainsi : les nombres tirés sont stockés dans un tableau de taille `MAX`, dans leur ordre d'apparition. La première case contiendra le premier nombre tiré, la deuxième case le deuxième, et ainsi de suite.

Lorsqu'on tire un nouveau nombre, on parcourt le tableau pour voir s'il y est déjà présent ; si c'est le cas, on ne touche pas au tableau et on recommence en tirant un nouveau nombre ; sinon, on ajoute ce nombre à la dernière case libre du tableau. Pour connaître l'indice de cette case, on peut utiliser une variable "compteur" qu'on initialisera à 0 au début, puis qu'on augmentera de 1 à chaque fois qu'on ajoute un nombre.

On a donc potentiellement un algo qui fonctionne. Maintenant, si vous avez du temps à perdre, vous pouvez l'implémenter et tester différentes valeurs de `MAX` : 200, 2 000, 20 000, 200 000 ... Personnellement, ma bécane traite le cas 20 000 en une dizaine de secondes et ne s'arrête pas en temps raisonnable (inférieur à la minute) pour 200 000. C'est dommage, j'ai 200 000 candidats qui vont passer le Bac (à sable ) et je dois leur donner un ordre de passage aléatoire...

Ce qui est très long dans cette façon de faire, c'est qu'on parcourt toute la partie du tableau déjà remplie pour savoir si un nombre a été tiré. Si on pouvait connaître instantanément cette information, on gagnerait beaucoup en efficacité. L'idée est donc de la stocker dans un deuxième tableau `test` : si le nombre `i` a déjà été tiré, alors `test[i]` est à `1`, sinon il est à `0`.

> L'intuition nous pousse à penser que cet algo peut ne jamais s'arrêter. C'est faux ! Prenons par exemple le cas où il ne reste plus qu'un nombre à tirer : la probabilité de NE PAS tirer ce nombre au premier coup est extrêmement élevée, mais tout de même strictement inférieure à 1. Notons la `p`. Pour que ce nombre ne soit pas apparu au deuxième coup, il ne doit pas être sorti au premier coup, ET ne doit pas être sorti au deuxième.
>
> 
>
> Si vous avez déjà manipulé un peu de probas, vous devez savoir que la probabilité de l'événement "A ET B" est égale à la probabilité de l'événement "A" multipliée par celle de l'événement "B" (si les événements sont indépendants, ce qui est le cas). Donc ici la probabilité de "le nombre n'est pas apparu au deuxième coup" est de : $p*p=p^2$.
>
> Et ainsi de suite, vous comprenez sans peine que la probabilité de "le nombre n'est pas apparu au `n`-ième coup" est de p^n. Mais `p` est strictement inférieur à 1. Donc lorsque n devient très grand, p^n devient aussi proche de 0 que l'on veut. Donc l'événement "le nombre n'apparaît jamais" a une probabilité de 0, tout pile.
>
> En conclusion, le nombre de coups nécessaires pour avoir le dernier nombre est bien fini ; il peut en revanche être très

grand.

Une approche plus efficace !

L'idée

La dernière version tourne assez rapidement (MAX=200 000 est traité chez moi en un clin d'?il), mais il subsiste deux points noirs :

- comme indiqué dans l'encadré juste au-dessus, même si le nombre de tirages est fini, il peut être très grand, ce qui prendra beaucoup de temps ;
- on a besoin d'un tableau annexe (`test`), ce qui prend de la place.

Pour comprendre le nouvel algo, je vais faire l'analogie avec un jeu de cartes ; remarquez que distribuer toutes les cartes aux joueurs revient à générer sans doublons toutes les cartes et les donner aux joueurs au fur et à mesure. Pour faire cela, est-ce que vous utilisez l'algo que j'ai décrit au dessus lorsque vous jouez aux cartes en famille ? Je ne pense pas (ou alors allez vite consulter un médecin, c'est grave); non, vous **mélangez** vos cartes !

C'est exactement la même idée ici. Nous allons encore avoir besoin d'un tableau (mais d'un seul cette fois). Pour l'initialiser, nous le remplissons "dans l'ordre" : à la case n° 0, on y met 0, à la n°1, on y met 1 et ainsi de suite jusqu'à `MAX`. Ensuite il suffit de le mélanger.

Alors un paquet de cartes, je vois comment le mélanger, mais un tableau stocké dans la `RAM`, j'ai plus de mal ...

Pas de panique ! L'idée est toute simple : vous tirez un nombre aléatoire `a` entre 0 et `MAX` ; puis vous échangez le contenu de la case n°`0` avec la case n°`a`. Et vous faites ça pour toutes les cases. Cet algo corrige bien les deux écueils que j'avais relevés à propos du premier et génère bien une suite sans doublons.

Il reste une petite broutille : pour l'instant, l'algorithme génère des entiers entre 0 et un nombre. Pour générer une suite à valeurs dans `[a,b[`, il n'y a qu'une modification à faire, lors de la génération du tableau : à la case n°0, on met `a`, à la case n°1, on met `a+1` et ainsi de suite. C'est tout ! Le reste du code se contentera de mélanger les cases, sans se préoccuper de leurs valeurs (il faut tout de même bien penser à tirer les valeurs dans l'intervalle `[0,b-a[` pour que les nombres tirés soient bien des indices du tableau).

L'implémentation

On pourrait écrire une fonction qui prend en argument `a` et `b`, les bornes de l'intervalle dans lequel on veut générer notre suite, et qui alloue un tableau, lui applique l'algo décrit au-dessus et le renvoie.

Problème : si on veut générer plusieurs suites aléatoires du même intervalle, on devra allouer et initialiser un nouveau tableau à chaque fois. Ce qui prend du temps. Et vous l'aurez remarqué, je n'aime pas en perdre !

On va donc écrire deux fonctions : une qui alloue et initialise le tableau et une qui se contente de mélanger un tableau passé en paramètre. Si on veut une nouvelle suite, il nous suffira de re-mélanger le tableau, sans avoir à le réallouer.

La première ne devrait pas vous poser de problèmes ; elle prend deux arguments entiers `a` et `b` qui sont les bornes de l'intervalle et renvoie un pointeur sur un tableau de taille `(b-a)` contenant `{a, a+1, a+2, ..., b-1}`. La voici :

Code : C

```
int* init_sans_doublons(int a, int b){
 int taille = b-a;
 int* resultat=malloc((taille)*sizeof (int));
 int i=0;
 // On remplit le tableau de manière à ce qu'il soit trié
 for(i = 0; i< taille; i++){
  resultat[i]=i+a;
 }
 return resultat;
```

```
}
```

Pour la deuxième, la seule difficulté se situe peut-être dans l'échange des valeurs de deux cases ; pour ce faire, on est obligé d'utiliser une variable `temp`. Cette fonction prend deux arguments : un pointeur vers un tableau et la taille de celui-ci.

Code : C

```
void melanger(int* tableau, int taille){
 int i=0;
 int nombre_tire=0;
 int temp=0;

 for(i = 0; i< taille;i++){
  nombre_tire=rand_a_b(0,taille);
  // On échange les contenus des cases i et nombre_tire
  temp = tableau[i];
  tableau[i] = tableau[nombre_tire];
  tableau[nombre_tire]=temp;
 }
}
```

Mais elle ne renvoie rien cette fonction !?

Effectivement, elle va directement modifier le tableau qu'on lui passe en paramètre, il n'y donc rien besoin de renvoyer !

Voilà, tout est prêt, il ne me reste plus qu'à vous donner un `main` qui permet de tester ces fonctions : le programme suivant génère une suite de nombres compris entre deux nombres entrés par l'utilisateur. Note importante : la fonction `init_sans_doublons` alloue un tableau, **il faut impérativement penser à libérer l'espace par la suite !**

Code : C

```
int main(){
        // A ne pas oublier !
        srand(time(NULL));
        int a=0;
        int b=0;
        int i =0;
        int* t=NULL; // Va contenir le tableau de nombres

        do{
                printf("Rentrez le premier nombre : ");
                scanf("%d",&a);
                printf("Rentrez le second, plus grand que le premier : ");
                scanf("%d",&b);
        }while(b<=a);

        // On commence pour de vrai ici :
        t=init_sans_doublons(a,b);
        melanger(t,b-a);

        printf("La suite aléatoire est : ");
        for(i=0; i<b-a; i++){
                printf("%d ",t[i]);
        }
        printf("\n");

        // Ne pas oublier de libérer le tableau
        free(t);
        return 0;
}
```

Ici, j'ai choisi de passer la taille du tableau en second argument de la fonction `melanger`. C'est assez désagréable : on est obligé de la calculer (alors que `init` le fait déjà) puis de se souvenir de cette taille pour appeler cette fonction. La solution habituelle est de créer un `struct` qui contient deux champs : le tableau et la taille de celui-ci. `init_sans_doublons` renverrait alors un pointeur vers une structure de ce type, calculant la taille une fois pour toute. Et `melanger` prendrait alors un seul argument : (un pointeur vers) la structure considérée.

Pour des raisons de simplicité, je n'ai pas opté pour cette solution pour le tuto ; dans la vraie vie, si vous devez vous servir à de nombreuses reprises de ces fonctions, je ne peux que vous conseiller de l'implémenter.

Voici un premier aperçu de l'utilisation des nombres aléatoires en C. Si vous voulez vous entrainer un peu, vous pouvez vous amuser à coder des jeux comme un yams ou un Master Mind, une animation en SDL qui simule une chute de neige ou encore plein d'autres applications auxquelles je ne pense pas.

Vous pouvez aussi vous amuser à vérifier "numériquement" ceci : si on a une file de personnes placées aléatoirement contenant Pierre et Paul, le nombre de personnes **le plus probable** entre les deux comparses est de 0 (autrement dit, Pierre et Paul sont côte à côte). Cela vous fera utiliser la génération sans doublon  . Cela paraît surprenant, mais c'est pourtant vrai, un simple calcul de dénombrement permet d'arriver au résultat.

Pour une utilisation un peu plus surprenante de l'aléatoire, vous pouvez allez voir le tuto sur la méthode Monte Carlo.

Bonne continuation et à bientôt pour un autre tuto !

Simuler une distribution de cartes géométrique

Dis, pourquoi y'a que 32 cartes dans ton jeu ?

Voici la question que m'a posé Titi, alors que nous terminions une partie endiablée de "bataille".

Je lui ai alors répondu que j'avais sous la main un jeu de 54 cartes, mais ça ne lui suffisait pas : "*Pourquoi **QUE** 54 cartes ?*" ; commençant à m'impatienter, j'espérais lui clouer le bec avec les 78 cartes du jeu de tarot, mais c'était peine perdu : "*Pourquoi **QUE** 78 cartes ? Pourquoi est-ce qu'on ne peut pas avoir autant de cartes différentes que l'on veut, ça serait plus marrant non ?*".

Au début, en tant que grande personne, j'étais sceptique, mais Titi a été très indulgent avec moi ; il m'a donné et redonné ses explications... Je vous transcris ici ce que j'en ai compris.

Son idée était d'inventer une variante au jeu de la bataille : au lieu de tirer les cartes dans le tas devant soi, on prend les cartes délivrées par une machine, avec une infinité de cartes différentes. Et ensuite de programmer une telle machine.

Pour comprendre tout cela, il m'a d'abord expliqué comment simuler un simple jeu de cartes, puis comment simuler un dé pipé. Il en a profité pour m'expliquer ce qu'étaient un espace de probabilité et une loi de probabilité, et comment simuler ces lois.

Il m'a ensuite emmené dans le monde de l'infini, j'ai alors compris que réaliser cette machine n'était pas une sinécure !

Un brave jeu de 32 cartes

Une machine "équilibrée"

D'entrée de jeu, Titi m'a déstabilisé en changeant le nom des cartes. Il m'a dit que cela n'avait pas d'importance ; au lieu d'avoir "As de cœur", "Roi de pique" ... Nous aurons simplement des entiers : 1, 2, 3, et ce jusqu'à 32. Il suffira d'avoir une liste dans un coin où l'on a noté que 1 correspond à "As de cœur", 2 à "As de carreau", etc.

Du coup, j'ai proposé moi-même une méthode pour simuler cette machine : tirer un nombre aléatoirement entre 1 et 32 et écrire le nombre obtenu sur un carton ! J'ai testé ça dans mon langage favori... et ça marche !

Évidement, Titi n'a pas pu s'empêcher de faire son intéressant :

Euh oui, ça marche, mais quel est ton espace de probabilité ? Quelle est ta loi de probabilité ?

Devant mes yeux grands comme des soucoupes, il se sentit obligé de m'expliquer. Au début, je ne voyais pas l'intérêt de se compliquer la vie avec de nouveaux noms, mais très vite, je me suis rendu compte ces notions nous facilitaient grandement la tâche ; soyez donc patients et attentifs, je vous promets que le résultat sera convaincant !

Espace de probabilité

C'est tout simplement **l'ensemble de tous les événements possibles**. C'est pourquoi on l'appelle aussi **"univers des possibles"**. Dans notre cas c'est très simple, il s'agit de {1, 2, 3, ..., 30, 31, 32}.

Probabilité

La probabilité d'un événement, c'est un **nombre** qui caractérise la fréquence d'un événement. Si ce nombre est très proche de 0, l'événement aura très peu de chances de se produire ; s'il est très proche de 1, l'événement aura au contraire beaucoup de chance de se produire.

Par exemple, quand on tire à pile ou face, la probabilité de tomber sur face est de 0,5 (=1/2) : on a 1 chance sur 2 que cela arrive. Quand on lance un dé, la probabilité de tomber sur le 3 est de 0,166... (=1/6) : on a 1 chance sur 6 que cela arrive. La probabilité de tirer la dame de pique (ou toute autre carte) dans notre jeu de 32 cartes est de 1/32.

Retenez bien : si la probabilité d'un événement est 0 (tout pile), l'événement ne se produira **jamais** ; si c'est 1, l'événement se produira **toujours**. Une probabilité est **toujours** comprise entre 0 et 1.

Loi de probabilité

Une loi de probabilité, c'est ce qui détermine la probabilité de **chaque événement**.

C'est quoi la différence avec une probabilité ?

C'est complètement différent. Après de longues réflexions, j'ai compris qu'en fait, connaître une loi de probabilité, ça revient à connaître un tableau qui indique à quelle probabilité apparaît chaque élément. Par exemple, pour notre jeu, la loi est donnée par :

Loi de probabilité d'une machine "équilibrée"

Carte n°	1	2	3	...	30	31	32
Probabilité	$\frac{1}{32}$	$\frac{1}{32}$	$\frac{1}{32}$	...	$\frac{1}{32}$	$\frac{1}{32}$	$\frac{1}{32}$

Comme vous pouvez le voir sur le tableau, la loi de probabilité pour notre jeu de carte est assez simple : chaque élément a la même probabilité d'apparition, à savoir 1/32.

Pour faire un peu plus sérieux, je me suis permis d'introduire une notation pour cette loi de probabilité : dans notre cas par exemple, je noterai la probabilité que le "3" apparaisse avec : P({3}). Ici, on a donc P({3})= 1/32. De même, pour n'importe quel entier n entre 1 et 32, on a P({n})=1/32. J'appellerai {3} un "*événement élémentaire*", et P({3}) une "*probabilité élémentaire*".

Vous avez donc une deuxième manière de connaître une loi, c'est de connaître chaque probabilité élémentaire ; autrement dit, pour tous les événements élémentaires **e** de l'univers des possibles, vous connaissez P(**e**).

Vous avez donc deux manières de représenter une loi : soit grâce à un tableau, soit grâce à P. Vous devez vous assurer que vous avez bien compris que ces deux notations servent à la même chose. Dans la suite, on utilisera soit l'un soit l'autre suivant ce qui est le plus adapté : il faut pouvoir jongler entre les deux sans problèmes !

Au passage, vous aurez remarqué que je dis maintenant simplement "loi" au lieu de "loi de probabilité" car je suis un gros fainéant, et que tout le monde sait bien de quoi on parle. Si un plouc pense que je parle de la "loi française", je lui conseille d'arrêter la fac de droit>

Ne voulant pas en rester là, j'ai eu envie de connaître la probabilité que le 3 ou le 7 apparaisse. En gardant la même notation, cette probabilité sera représentée par P({3,7}) qui est clairement égale à P({3}) + P({7}) = 2/32 (ou 1/16 pour les puristes).

Les plus malins d'entre vous tenteront de calculer P({3,3}). On peut le faire bien sûr, mais cela ne vaut pas P({3}) + P({3}) ; {3, 3} représente l'événement : "le 3 sort" OU "le 3 sort"... Vous admettrez qu'il s'agit tout simplement de l'événement "le 3 sort" et donc que sa probabilité est égale à P({3}).

Parti sur ma lancée, j'ai cherché la probabilité que 5 cartes données apparaissent, puis 10, et puis tant qu'à faire 32 (c'est à dire, la totalité du jeu). Cette probabilité est de : P({1, 2, ..., 31, 32}) = P({1}) + P({2}) + ... + P({31}) + P({32}) = 32/32 = 1. Vous allez me dire que j'aurais pu m'en douter : l'événement que je considère se produit tout le temps (lorsqu'on tire une carte, on tire soit le 1, soit le 2, ..., soit le 32). Or un événement qui se produit tout le temps a une probabilité de 1 !

Titi m'a alors fait remarquer la chose suivante : *la somme de toutes les probabilités élémentaires est égale à 1* !

Ça ne marche que pour notre jeu de cartes ça non ?

Eh bien non, c'est en fait très général. Si vous reprenez le paragraphe au-dessus, vous vous rendez compte que vous pouvez le modifier pour l'adapter à n'importe quel univers des possibles. On peut donc caractériser une loi de probabilité : **il suffit que lorsqu'on additionne toutes les probabilités élémentaires, on obtienne 1.**

Après m'avoir expliqué tout ça, Titi n'était pas encore très satisfait : "Ta machine elle marche, mais elle est pas drôle ; et puis les cartes, j'en ai ma claque, j'ai envie de jouer aux dés maintenant ! Je veux une machine qui simule un dé pipé : plein de 4 et pas

beaucoup d'autres chiffres !"

Une machine déséquilibrée

Faites votre loi !

Pour faire cela, Titi m'a dit qu'il suffisait de définir une nouvelle loi de probabilité.

Ce que veut Titi, c'est avoir beaucoup de "4". Pour cela, je mets donc une grosse probabilité sur "4", disons 0,6. Une fois que j'ai fixé cette probabilité, je ne peux pas prendre ce que je veux pour les autres probabilités élémentaires. Souvenez vous, il faut que la somme de toutes les probabilités élémentaires soit égale à 1. Il me reste alors 0,4 (= 1 - 0,6) à répartir entre les valeurs restantes.

Par exemple, je peux avoir la loi suivante :

Loi de probabilité d'une machine "déséquilibrée"

Face n°	1	2	3	4	5	6
Probabilité	0.08	0.08	0.08	0.6	0.08	0.08

Évidemment, je ne suis pas obligé de mettre la même probabilité pour "1", "2", "3" ,"5" et "6". Je peux aussi prendre :

Loi de probabilité d'une machine "déséquilibrée"

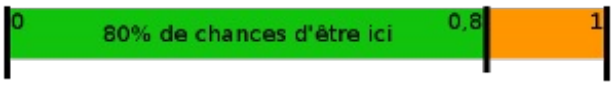
Face n°	1	2	3	4	5	6
Probabilité	0.05	0.05	0.1	0.6	0	0.2

Vous remarquez ici que la probabilité de tirer un 5 est de 0 ; cela veut dire que le 5 ne sortira jamais !

Je vous laisse le soin de vérifier que la somme des probabilités fait bien 1. Et coup de chance, cette loi plaît beaucoup à Titi !

Construire la machine

Mais cette loi est un peu plus dure à simuler que la première : chaque élément a une probabilité différente de sortir. Tirer "au hasard" ne rendra pas compte de cette loi.



L'approche ici est totalement différente. Elle part du constat suivant : si on tire un nombre au hasard dans l'intervalle [0;1], ce nombre aura par exemple beaucoup plus de chances de se trouver dans le sous-intervalle [0;0,8] que dans [0,8;1]. De façon générale, plus le sous-intervalle sera grand, plus la probabilité de tirer un nombre à l'intérieur sera grande.

On peut même être plus précis  : la probabilité de tirer un nombre dans le sous-intervalle [a,b] est égale à la longueur de ce dernier, c'est à dire (b-a). Et c'est là que va se situer toute l'astuce de la technique : à un événement élémentaire de probabilité p, on associe un sous-intervalle de [0,1] de longueur p.

En se débrouillant pour que les sous-intervalles ne se chevauchent pas, on aura donc complétement recouvert notre segment [0,1] ! Pour m'assurer que j'avais bien compris, Titi m'a demandé de lui faire un dessin ; il fut relativement déçu de mes talents artistiques :

Comme vous pouvez le constater, j'ai construit mes sous-intervalles pour coller avec la loi ci-dessus. La partie verte est associée à l'événement "le 1 sort", la partie violette à l'événement "le 4 sort", et ainsi de suite. On constate aussi que chaque sous-intervalle a bien la bonne longueur : la partie verte mesure 0.05, la violette 0.6 (=0.8 - 0.2), etc.

Maintenant, pour simuler notre machine, il suffira de tirer un nombre au hasard entre 0 et 1. Sa position dans le schéma ci-dessus déterminera quelle carte doit sortir : si le nombre est dans la partie bleue, on sortira le 3, s'il est dans la partie rouge, on sortira le 6. De cette manière, le 1 et le 2 sortiront avec une probabilité de 0.05 (les longueurs des parties verte et orange), le 3 avec une probabilité de 0.1, etc. Donc la machine suivra bien la loi que Titi avait choisie !

Certains se demandent peut-être s'il faut prendre des sous-intervalles fermés ou ouverts (c'est à dire, en incluant ou pas les bornes de l'intervalle) ; en théorie, cela n'a en fait aucune importance. Effectivement, la probabilité de tomber dans un intervalle est égale à sa longueur. Ici, la "longueur" d'un intervalle constitué d'un seul nombre est de 0, et donc la probabilité de tomber dessus tout pile est également 0 ! Donc, même si en pratique le générateur aléatoire n'est pas parfait, il s'agit d'un événement qui se produit extrêmement peu souvent. En résumé, faites comme vous voulez !

Pour l'implémenter, pas d'astuces : on tire un nombre aléatoire x entre 0 et 1. On regarde si x est inférieur à la première valeur dans le tableau ; dans ce cas, on renvoie 1 ; sinon, on regarde s'il est inférieur à la somme des deux premières valeurs du tableau auquel cas on renvoie 2, et ainsi de suite ...

Hein ? Pourquoi la **somme** des deux premières valeurs ? Pourquoi pas la deuxième valeur directement ?

Revenez au petit schéma ci-dessus : on veut d'abord savoir si x est dans la zone verte, donc on le compare à 0,05. Après, on veut savoir s'il est dans la zone bleue, autrement dit, s'il est

- supérieur à 0,05 : de ce côté là, pas de problème, si on arrive à cet endroit du code, c'est que x n'est pas inférieur à 0,05  ;
- **ET** inférieur à 0,1 : autrement dit s'il est inférieur à 0,05 + 0,05, ce qui correspond aux 2 premières valeurs du tableau. Si c'est le cas, on renvoie 2.

Et on continue ainsi jusqu'à ce que x soit inférieur à la somme de tous les éléments parcourus.

Pour faire plaisir à Titi, j'ai implémenté cet algo en C, il fut cette fois très content du résultat (pour une fois !) :

Secret (cliquez pour afficher)

Code : C

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/* Renvoie un nombre aléatoire entre 0 et 1 */
double rand_0_1(){ return rand()/(double)RAND_MAX; }

/*Génère un entier entre 1 et taille suivant la loi modélisée par
le tableau
passé en paramètre */
int alea_perso(double loi[], int taille){
 int i=0;
 double x=rand_0_1();
 double somme=0;
 /* Dans le premier passage dans la boucle, on testera
si x est inférieur à somme, c'est à dire loi[0]. Si ce n'est pas
le cas
on relance la boucle : somme vaudra alors loi[0] + loi[1]
... et ainsi de suite.
*/
 do{
  somme += loi[i];
  i++;
 }while( somme < x && i < taille  );
 return i;
}

int main(){
 const int NB_EVT=5;
 double loi[]={0.05,0.05,0.1,0.6,0,0.2};
```

```
    int nb_tirages=0;
    int i=0;
    int carte=0;
    srand(time(NULL));

    do{
     printf("Combien de tirages voulez-vous effectuer ? ");
     scanf("%d",&nb_tirages);
    }while(nb_tirages <= 0);

    printf("Voici la liste des tirages : \n");
    for(i=0; i<nb_tirages;i++){
     carte=alea_perso(loi,NB_EVT);
     printf("%d ",carte);
    }
    printf("\n");
    return 0;
   }
```

Attention, la méthode que Titi vous a présentée ici ne simule pas un paquet fini de cartes. Effectivement, si la carte n°1 est déjà sortie, elle peut sortir encore une fois. Pour simuler un paquet "réel" de manière correcte, il faudrait modifier la loi après chaque tirage.

Un jeu infini !

Maintenant, on passe à la vitesse supérieure, on va construire une machine qui sort un nombre infini de cartes (Titi s'est vite lassé des dés) ! Pour utiliser le vocabulaire que Titi m'a appris, cela signifie que **l'univers des possibles sera infini**. Vous allez voir que ce n'est pas une mince affaire.

Un problème épineux

Comme le dit Titi, "il faut réutiliser ce qu'on sait déjà faire" : dans la partie précédente, on a vu que si on connaît une loi, on a alors une méthode pour simuler une machine qui suit cette loi. Je pensais être sorti d'affaire : "Il me suffit de déterminer cette loi et le tour est joué !".

Mais si l'univers des possibles est infini, comment stockeras-tu ta loi ? Tu ne pourras pas utiliser un tableau !

Il est énervant à toujours avoir raison ce Titi ! Ma solution est de remplacer notre tableau par une gentille fonction qui prend un entier en paramètre et qui renvoie sa probabilité d'apparaître. Par exemple, j'ai eu l'idée d'utiliser le même type de loi que pour la machine équilibrée : toutes les cartes ont la même probabilité, et je choisis cette proba à 0,05 ! Ce qui nous fait une fonction du type :

Code : C

```
double loi(int n){
    if(n>0)
        return 0.05;
    else
        return 0; /* La numérotation des cartes commence à 1, donc
la probabilité de tirer un nombre inférieur
ou égal à 0 est nulle ! */
}
```

C'est un peu mieux... Mais cette fonction ne représente absolument pas une loi de probabilité : en additionnant les 20 premières probabilités élémentaires, on obtient 1, et si on continue, on dépasse le allégrement ; c'est même pire que ça, vu qu'il y en a un nombre infini, la somme de toutes les probabilités sera ... infinie !

Et prendre une probabilité plus petite ne changera rien, on mettra juste "plus de temps à atteindre l'infini".

Gare à celui qui me propose d'alterner des nombres positifs et négatifs pour faire en sorte que certains termes en annulent d'autres : une probabilité négative n'a aucun sens !

J'ai alors pensé à ne mettre une probabilité que sur un nombre fini de cartes, le reste des cartes ayant une probabilité nulle. Ce qui correspondrait à :

Code : C

```
double loi(int n){
    if(n>0 && n<=20)
       return 0.05;
    else
       return 0;
}
```

C'est bien une loi, mais on n'a rien gagné par rapport à la première partie ! En effet, on simule ici une machine qui ne tire qu'un nombre fini de carte ... Retour à la case départ !

Au lieu de chercher à construire une loi à tâtons, on va plutôt tenter de modéliser une situation "réelle" ; on s'assurera ensuite que ce qu'on a trouvé est bien une loi.

Un jeu de pile ou face

Le jeu

Titi m'a proposé le jeu suivant : je lance une pièce jusqu'à ce que je tombe sur pile. Mon score est alors le nombre de lancers que j'ai effectués. Par exemple, si je tire face, face puis pile, mon score est de 3 ; si je tire pile du premier coup, mon score est de 1.

Bien sûr, ce qu'on va essayer de faire, c'est simuler ce jeu par une machine : on lance la machine, elle calcule notre score puis l'inscrit sur une carte.

L'algorithme le plus naïf serait de tirer au hasard 0 (pour pile) ou 1 (pour face), compter le nombre de fois qu'on le fait et de s'arrêter lorsqu'on tire un face. De façon plus concise, cela donne :

Code : Autre

```
compteur=0
faire
    nombre = tirer 0 ou 1 au hasard
    compteur ++
tant que nombre = 0
afficher compteur
```

Mais je me suis vite rendu compte que cela n'allait pas du tout : on tire potentiellement pleins de nombres aléatoires, ce qui n'est pas du tout économique ! Cherchons maintenant à trouver la loi de cette machine pour pouvoir appliquer l'algo de la première partie, et être plus efficace.

La probabilité que je tire pile du premier coup est de 1/2 (Titi n'est pas encore assez âgé pour avoir la fourberie de me donner une pièce déséquilibrée). On connaît donc déjà une probabilité élémentaire : P({1})=1/2 (c'est la traduction de la phrase d'avant : la probabilité que mon score soit 1 est de 1/2).

Pour que je tire pile au deuxième coup (sans l'avoir fait au premier), il faut que je tire:

- face au premier coup, ce qui a une probabilité de 1/2;
- ET pile au deuxième, ce qui a aussi une probabilité de 1/2.

Croyez moi sur parole, nous avons : P({face au premier lancer} ET {pile au second lancer}) = P({face au premier lancer})*P({pile au second}) = 1/2 * 1/2 = 1/4.

Et donc P({2}) = 1/2² = 1/4. De la même manière, vous pouvez trouver que P({3}) = 1/2 * 1/2 * 1/2 = $1/2^3$=1/8, P({4})=$1/2^4$, ... On peut alors généraliser : P({n}) = $1/2^n$. Nous connaissons donc bien notre loi vu que pour chaque événement élémentaire, on a une probabilité.

Mais avons nous bien une loi ? Autrement dit, la somme de toutes les probabilités élémentaires fait-elle bien 1 ?

La loi du gâteau idéal

Déjà, quel est l'univers des possibles ? Le score du jeu peut être 1, 2, 3, 42,512, ... en fait, n'importe quel entier positif non nul. Donc on a bien un univers infini.

Et on veut vérifier que la somme des probabilités élémentaires soit égale à 1. Autrement dit que :

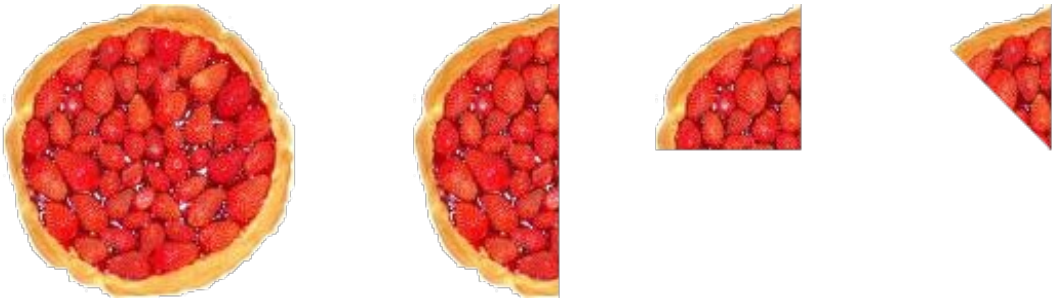
$$P(1)+P(2)+P(3)+\ldots = \frac{1}{2}+\left(\frac{1}{2}\right)^2+\left(\frac{1}{2}\right)^3+\ldots = \frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\ldots = 1$$

Vous remarquerez que j'ai utilisé cette fois "..." sans rien mettre derrière ; cela signifie qu'on continue la somme "à l'infini". Si j'avais mis "0.1+ 0.2 +0.3 +... + 0.9" cela signifierait que j'ai une somme finie.

Pour m'expliquer ce problème, Titi m'a raconté ce qu'il a fait à son dernier anniversaire. Comme à tous les anniversaires, il y avait un beau gros gâteau, comme on les aime de par chez nous, un gâteau "idéal" : on peut en couper des parts aussi fines qu'on veut, sans en perdre une miette.

Et Titi a beaucoup de chance, il a beaucoup d'amis. Quand je dis beaucoup, ce n'est pas 10, 100 ou 100 000. Non, beaucoup. Une infinité !

Et comme Titi est à moitié généreux, à chaque nouvel arrivant, il offrait la moitié restante du gâteau. Donc pour le premier invité, le gâteau était entier, il a donné la moitié du gâteau ; donc 1/2. Il restait alors la moitié du gâteau ; le deuxième invité a donc reçu la moitié de la moitié, autrement dit un quart (1/4). Et ainsi de suite, le troisième a reçu 1/8, le quatrième 1/16 ...
La question est : quand il aura servi tous ses amis, restera-t-il du gâteau pour Titi ?



Pour déjà faire le lien avec notre problème, on peut remarquer que la quantité de gâteau donnée par Titi correspond à la somme qui nous intéresse.

On remarque immédiatement que cette somme ne dépassera jamais 1 : Titi ne peut pas donner plus de gâteau que le gâteau lui même ! Et en regardant les images ci-dessus, on voit bien qu'à la fin, Titi aura distribué tout le gâteau et qu'il n'en aura pas ; pas de chance, il fallait se servir avant ! Tout ça pour dire que la somme qui nous intéresse fait bien 1.

Pourquoi n'a-t'il pas la moitié de la dernière part ?

Il y a une infinité d'ami... Du coup, pas de dernière part !

Nous sommes maintenant sûr d'avoir affaire à une loi de probabilité, on peut maintenant essayer de la simuler !

Une machine rapide !

Maintenant qu'on a une loi, on peut réadapter le programme qu'on avait fait pour simuler ce jeu dans la première partie. Je vous rappelle la méthode : on tire un nombre x compris entre 0 et 1, puis on regarde dans quelle "zone" il se situe. Ici le découpage des zones se fait ainsi :

Pour "regarder dans quelle zone x se situe", je vous rappelle (n'hésitez pas à relire la première partie) qu'en gros, on teste si x est dans la zone "1", puis s'il est dans la zone "2", et ainsi de suite.

Pour appliquer le même algo, il faut alors définir une fonction qui représentera notre loi :

Code : C

```c
#include <math.h>
double loi(int n){
    if(n<=0) return 0;
    else return pow(0.5,n); // = (1/2)^n
}
```

Ce qui au final donnerait un programme de ce type :

Secret (cliquez pour afficher)

Code : C

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

/* Renvoie un nombre aléatoire entre 0 et 1 */
double rand_0_1(){ return rand()/(double)RAND_MAX; }

double loi(int n){
    if(n<=0) return 0;
    else return pow(0.5,n); // = (1/2)^n
}

/*Génère un entier suivant la loi modélisée par la fonction "loi"
*/
int alea_perso(){
 int i=0;
 double x=rand_0_1();
 double somme=0;

 do{
  somme += loi(i);
  i++;
 }while( somme < x );
 return i-1;
}

int main(){
 int nb_tirages=0;
 int i=0;
 int carte=0;
 srand(time(NULL));

 do{
  printf("Combien de tirages voulez-vous effectuer ? ");
  scanf("%d",&nb_tirages);
 }while(nb_tirages <= 0);

 printf("Voici la liste des tirages : \n");
 for(i=0; i<nb_tirages;i++){
  carte=alea_perso();
  printf("%d ",carte);
```

```
        printf("%d ",carte);
    }
    printf("\n");
    return 0;
}
```

Mais en fait, ce n'est pas beaucoup mieux que la première version. Je vous rappelle que dans la première version, on tirait un nombre 0 ou 1 tant que ce nombre n'était pas 1 ; donc si le score valait 5, on avait donc 5 "tours de boucles". Ici, si on regarde de la même façon le "nombre de tours", on se rend compte très vite qu'il est aussi égal au score ; donc niveau efficacité, c'est la loose .

On va donc essayer d'être plus efficace : on va réfléchir pour trouver directement dans quelle zone se situe x sans faire aucune boucle. Autrement dit, on va chercher une formule mathématique qui nous donnera la zone directement à partir de x.

Pour certaines valeurs, c'est très facile de dire "à l'œil nu" dans quelle région est x. Par exemple, si x=0.2, il est clairement dans la région "1" ; si x=0.6, il est dans la région "2". Mais si x=0.9876, vous serez bien en peine de me dire - sans calculs ! - dans quelle région est x ! C'est cette "recherche de zone" que l'on va automatiser dans ce qui suit.

La partie qui suit, même si expliquée en détail, est relativement technique ; savoir ce qu'est une inéquation est fortement recommandé. Si vous voyez que vous ne comprenez rien à mon charabia, sautez à la formule finale sans états d'âmes.

Pour bien comprendre, mettons que x est dans la zone "3". Regardez le schéma au-dessus, cela veut dire que x est compris entre P({1})+P({2}) et P({1})+P({2})+P({3}). Autrement dit :

$$P(\{1\}) + P(\{2\}) \ < \ x \ \leq \ P(\{1\}) + P(\{2\}) + P(\{3\}).$$

On connaît très bien P({1}),P({2}) et P({3}), on peut donc écrire :

$$\frac{1}{2} + \left(\frac{1}{2}\right)^2 \ < \ x \ \leq \ \frac{1}{2} + \left(\frac{1}{2}\right)^2 + \left(\frac{1}{2}\right)^3$$

Mais la zone dans laquelle est x, c'est pas ce qu'on cherche à trouver ? On se mord la queue là, non ?

Oui, c'était un exemple pour bien comprendre la formule que je vais vous balancer. Maintenant, mettons que x est dans la zone "n" (que nous ne connaissons pas); on peut donc écrire :

$$\frac{1}{2} + \left(\frac{1}{2}\right)^2 + \cdots + \left(\frac{1}{2}\right)^{n-1} \ < \ x \ \leq \ \frac{1}{2} + \left(\frac{1}{2}\right)^2 + \cdots + \left(\frac{1}{2}\right)^n.$$

Nous avons donc une superbe inéquation, et le but du jeu, **c'est de trouver n** (nous connaissons déjà la valeur de x) !

Dans un premier temps, nous allons trouver une forme plus sympathique aux grosses sommes de part et d'autre des inégalités. Vous remarquerez que :

$$\frac{1}{2} + \left(\frac{1}{2}\right)^2 = \frac{3}{4} = 1 - \left(\frac{1}{2}\right)^2$$

$$\frac{1}{2} + \left(\frac{1}{2}\right)^2 + \left(\frac{1}{2}\right)^3 = \frac{7}{8} = 1 - \left(\frac{1}{2}\right)^3$$

...

$$\frac{1}{2} + \left(\frac{1}{2}\right)^2 + \cdots + \left(\frac{1}{2}\right)^{n-1} = 1 - \left(\frac{1}{2}\right)^{n-1}$$

$$\frac{1}{2} + \left(\frac{1}{2}\right)^2 + \cdots + \left(\frac{1}{2}\right)^n = 1 - \left(\frac{1}{2}\right)^n$$

Donc notre grosse inégalité se re-écrit : $1 - \left(\frac{1}{2}\right)^{n-1} < x \leq 1 - \left(\frac{1}{2}\right)^n$. Ce qui me fait déjà beaucoup moins peur. En enlevant 1 à chaque bout, puis en multipliant par (-1) on a (multiplier par un nombre négatif change le sens d'une inégalité) : $\left(\frac{1}{2}\right)^{n-1} > 1 - x \geq \left(\frac{1}{2}\right)^n$

Gloups ... et comment je fais "sauter" la puissance ?

Alors pour faire ça, rappelez vous de la façon dont vous faites "sauter" un carré : vous prenez la racine carré !

Mais qu'est-ce que la racine carré ? Vous connaissez surement très bien la fonction "carré" f(x)=x² qui a pour effet de multiplier un nombre par lui même. Eh bien la racine carré est la fonction qui fait le boulot inverse de la fonction carré.

Dans notre cas, nous avons la fonction "puissance de 1/2" $f(n) = \left(\frac{1}{2}\right)^n$; eh bien il existe une fonction qui fait le boulot inverse la fonction "puissance de 1/2". Cette fonction s'appelle le **logarithme de base 1/2** (évidement, si on considère la fonction "puissance de a", on aura affaire au "logarithme de base a"). Je le noterai $\log_{1/2}$. Ainsi, on aura : $\log_{1/2}\left(\left(\frac{1}{2}\right)^n\right) = n$

Vous pouvez remarquer que $\log_{1/2}$ est décroissante, donc quand je l'applique, les inégalités "changent de sens".

Revenons à notre calcul, j'applique le $\log_{1/2}$  :

$$n - 1 < \log_{1/2}(1 - x) \leq n$$

Donc il n'y a plus qu'une seule possibilité pour n : c'est l'entier qui est juste au-dessus (fonction `ceil` en C) de $\log_{1/2}(1-x)$.

Une dernière petite optimisation (c'est vraiment du bout de chandelles ...) : x est un nombre aléatoire compris entre 0 et 1 ... c'est aussi le cas de 1-x, donc dans l'argument du $\log_{1/2}$, on peut se contenter de prendre x au lieu de 1-x.

Dernière note technique : pour calculer $\log_{1/2}(x)$ concrètement en C, vous devrez faire le calcul suivant : `log(x)/log(0.5)` (en ayant inclus `math.h` évidemment).

Pfiou, j'ai enfin fini ! Bah ... pourquoi tout le monde est parti ? 

La formule finale

Tout ceci nous mène à ce superbe résultat ; pour savoir dans quelle zone se situe x, il suffit de faire le calcul :

```
ceil( log(x) / log(0.5) );
```

Cela correspond au score que j'ai obtenu au jeu de pile ou face. Il ne me reste plus qu'à l'écrire sur une carte !

Tout ce patacaisse pour une pauvre petite malheureuse formule ? Mais qu'est-ce qu'on a gagné ?

Vous avez gagné sur plusieurs points :

- efficacité du code : le calcul du log est très rapide, optimisé par des grosses brutes des mathématiques, beaucoup plus efficace que notre boucle !
- simplicité du code : nous n'avons plus besoin de définir une fonction "loi" auxiliaire, la simulation du score se fait en 1

ligne (voir code plus loin).
- vous avez appris quelques techniques mathématiques qui vous resserviront peut-être .

Pour simuler le score à ce jeu, il suffit de faire une fonction ressemblant à (je vous laisse faire le main qui va autour) :

Code : C

```
int score_pile_face(){
        double x=rand_0_1();
        int score=ceil(log(x)/log(0.5));
        return score;
}
```

Que l'on peut condenser en un ligne `return ceil(log(rand_0_1())/log(0.5))` . Le simple appel à cette fonction renverra le score qu'on obtient en jouant à ce jeu ; l'entier qu'elle renverra pour donc ne pas être toujours le même !

Des variantes

Titi est content, il a sa machine qui lui permet de simuler un jeu infini de cartes ; mais nous pouvons encore trouver d'autres lois. Par exemple, nous pouvons prendre une pièce "déséquilibrée" qui tombe sur face avec une probabilité de 0,7. Vous pouvez tenter de refaire les calculs ci-dessus pour obtenir une formule pour cette nouvelle machine.

Nous pouvons aussi modifier un peu le jeu : on tire 100 fois à pile ou face, et notre score est le nombre de "piles" que nous avons obtenus.

On pourrait faire le même genre d'étude que juste au-dessus, mais malheureusement, nous ne pourrons pas nous servir du logarithme, ni d'aucune fonction "simple" pour inverser l'expression que nous allons trouver (Titi a tenté de m'expliquer le calcul, je n'y ai rien compris ...). Le plus efficace est alors d'appliquer un algo "naïf" :

Code : Autre

```
compteur=0
faire 100 fois
    nombre = tirer 0 ou 1 au hasard
    si nombre = 0
        compteur ++
afficher compteur
```

Je l'ai répété assez de fois, pour avoir une loi, il suffit que la somme des probabilités élémentaires soit égale à 1 ; mais, vu qu'il y a une infinité de termes, cela force chacun de ces termes à devenir de plus en plus petit.

Pour construire une loi, vous pouvez donc essayer de trouver une suite (infinie) de nombres qui tend vers 0 et faire en sorte que leur somme vaille 1 ; le problème n'est pas évident car si les nombres ne tendent pas "assez vite" vers 0, la somme explose et tend vers l'infini ! Mais comme je suis gentil, je ferai un petit article à ce sujet dans pas longtemps...
Nous avons donc rempli notre objectif initial : construire une machine délivrant une infinité de cartes.

En réalité, pour employer le vocabulaire mathématique, Titi vous a initié à la simulation de "variables aléatoires" (que j'avais appelées "machines" dans le tuto pour ne pas vous surcharger en vocabulaire), et vous a fait découvrir une loi bien connue des probabilistes : la loi géométrique.

Parfois on parle de "distribution" géométrique à la place de "loi" géométrique. Ce n'est pas exactement la même chose mais revient au même. Vous pouvez donc maintenant comprendre le double sens du titre de ce tuto !

Nous avons aussi effleuré le problème de la loi binomiale : il s'agit du jeu où l'on lance 100 fois une pièce et où l'on compte le nombre de "faces". Je vous ai dit qu'il n'y avait pas de méthodes "malines" pour simuler cette loi ; mais lorsque le nombre de tirages est grand, on peut approcher la loi binomiale par une autre - la loi de Poisson - qui elle est facile à simuler.

Il y a un point que nous n'avons absolument pas abordé dans ce tuto : les lois continues. Effectivement, nous n'avons parlé que de lois discrètes (c'est le mot mathématique pour désigner ce qui n'est pas continu), or il en existe d'autres ! Prenez par exemple le

retard de votre bus sur son horaire : l'univers des possibles est [0,+infini[, ce qui est un ensemble infini, mais "beaucoup plus grand" (et donc compliqué à appréhender) que ce que nous avons vu (c'est à dire l'ensemble des entiers : {0,1,2,3,...}).

Voici donc ce que vous avez à creuser si le sujet vous intéresse... Mais comme vous avez pu le sentir, avant de pouvoir vraiment faire des probabilités intéressantes, il faut d'abord avoir un bagage mathématique convenable pour pouvoir se dépêtrer d'inégalités tordues et de sommes gigantesques ! 

Partie 2 : La méthode de Monte Carlo

Avez-vous entendu parler du "Projet Manhattan" ? Non ? Bon, le bombardement de Nagasaki et Hiroshima, peut-être ? Ah déjà ça vous parle !

Mais quel peut bien être le rapport entre 2 villes japonaises, un quartier New-yorkais et un quartier de Monaco ?

Hiroshima et Nagasaki ont été bombardées avec des bombes atomiques (ça, vous êtes sensés savoir !) ; bombes atomiques qui ont été développées par les américains à la fin de la Seconde Guerre mondiale, dans le cadre du "Projet Manhattan". Projet qui, pour la première fois de l'histoire, a utilisé massivement ces fameuses méthodes dites de Monte Carlo pour effectuer des calculs.

Mais pourquoi Monte Carlo me direz vous ? Eh bien tout simplement car ces méthodes sont basées sur la simulation de nombres aléatoires ; qui dit "nombre aléatoire" dit "hasard", et qui dit "hasard" fait fortement penser aux casinos... D'où le nom !

Dans cette partie, vous allez donc comprendre comment fonctionnent ces méthodes et quels sont leurs avantages par rapport aux méthodes plus "classiques" (à propos desquelles nous serons obligés de dire quelques mots).

La partie se découpe en 3 :

- Un premier exemple introductif accessible à tout le monde où l'on découvre une méthode originale pour calculer pi ;
- des explications un peu plus théoriques sur la méthode, son utilisation et ses avantages avec des méthodes plus "classiques". Cette section, décomposée en deux chapitres, est un peu plus corsée mathématiquement ; j'essaierai de la rendre accessible à tout le monde mais un niveau de seconde (système français) me semble un minimum ;
- un exemple d'application finale utilisant la méthode décrite précédemment : le calcul de la valeur du champ électrique autour d'une plaque chargée (à venir).

Calculons pi !

Bon, je t'arrête tout de suite, pi je connais depuis que j'ai 10 ans, ça vaut 3.14159... !

Je suis d'accord avec vous ... Mais comment connaissez-vous cette valeur ? Grâce à la maîtresse qui vous a demandé d'apprendre par cœur ce nombre ? Mais cette maîtresse, comment elle l'a su ? Grâce à sa propre maîtresse ?

On voit bien que si l'on continue le raisonnement, il faut bien que quelqu'un, un jour, ait trouvé cette valeur. Ce calcul a toujours été un grand problème des mathématiques et plusieurs méthodes ont été imaginées ; on peut notamment citer celle d'Archimède dans la Grèce antique. Aujourd'hui encore, certains fous furieux tentent de découvrir des méthodes encore plus efficaces afin de connaître encore plus de décimales.

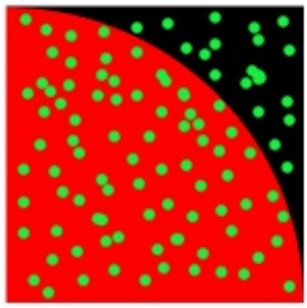
Ici, nous n'allons rien faire de compliqué, la méthode sera très simple à comprendre (satisfait ou remboursé !).

Titi à la rescousse

Un jeu de fléchettes

Je vais vous montrer ici une technique très simple, tellement simple que c'est Titi, votre petit frère de 3 ans et demi, qui va nous servir de "calculateur". On va demander à Titi de jouer à un jeu, lui aussi très simple, consistant à envoyer des fléchettes sur la cible représentée ci-dessous :

On ne va pas lui demander de viser, il en est incapable ; la fléchette doit juste arriver dans le carré. On donne à Titi 100 fléchettes et on le laisse jouer une heure ou deux. A la fin, la cible ressemblera à :



Refléchissons un peu

Voilà, le plus gros du travail est fait (pas très fatigant, et en plus, on a occupé Titi tout l'après-midi) ! Bon, laissons Titi tranquille pour le moment et commençons à réfléchir... Lorsque Titi lançait ses fléchettes, le jet était complètement aléatoire, donc la probabilité qu'elles atterrissent sur la partie rouge est proportionnelle à la surface de cette partie rouge. Cela se comprend intuitivement : plus une partie est grande, plus on a de chance de "tomber" dedans. Plus précisément, on a :

$$p = \text{"Probabilité que la fléchette soit dans le rouge"} = \frac{Surface\ rouge}{Surface\ totale}$$

On cherche à connaitre *p* (vous comprendrez pourquoi dans un instant). Pour cela, répondons à quelques questions : comment est constituée la cible ? Réponse : un quart de disque rouge dans un carré noir. Maintenant, un peu plus dur : Quelle est l'aire de la partie rouge (on considère que le carré est de côté 1) ? Vous séchez ? Alors prenons calmement : un disque de rayon R a une surface de πR^2. Ici nous avons un quart de disque, donc sa surface est $\frac{\pi R^2}{4}$; et *R=1*, donc, la surface est de $\frac{\pi}{4}$.

La surface totale de la cible est de *1x1=1* (c'est un carré de côté 1).

Donc nous avons : $p = \frac{Surface\ rouge}{Surface\ totale} = \frac{\pi/4}{1} = \frac{\pi}{4}$.

Mais nous avons un autre moyen **d'estimer** *p*. Cette estimation est toute simple, elle consiste à considérer que *p* est à peu près égale au rapport $\frac{nombre\ de\ points\ dans\ le\ rouge}{nombre\ de\ points\ total}$. Évidemment, plus le nombre de points est important, meilleure sera cette estimation. En m'amusant à compter combien de points sont dans la partie rouge, je trouve : 82. En remplaçant dans les formules trouvées plus tôt, on a donc : $p = \frac{82}{100} = \frac{\pi}{4}$. Une petite manipulation très simple donne : $\pi = \frac{82}{100} * 4 = 3.28$.

Mais ton résultat est faux ! Elle n'est pas juste ta méthode !

Je dirais plutôt que mon résultat comporte une erreur ; souvenez vous, on a **estimé** la valeur p comme étant le rapport "nombre de points dans le rouge"/"nombre de points total". Il est donc normal que l'on ait qu'une **estimation** de pi. Pour réduire l'erreur, il suffit d'augmenter le nombre de tirages ; mais malheureusement, Titi est fatigué et ne peut plus rien faire... Nous allons donc demander à nos ordinateurs de lancer des fléchettes à notre place.

Un piètre tireur : votre ordinateur

Nous allons donc implémenter ici un algorithme permettant de reproduire l'expérience que l'on vient de faire avec Titi ; grossièrement, il ressemblera à ceci :

Code : Autre

```
initialiser un compteur c à 0
lancer N fléchettes dans un carré de côté 1 :
    si la fléchette est dans le quart de cercle, on augmente le compteur de 1
    sinon, on ne fait rien
```

```
    on retourne c*4/N
```

Les seuls points un peu compliqués sont les lignes 2 et 3.

Lancer une fléchette

Pour "lancer une fléchette dans un carré de côté 1", il suffit de tirer au hasard deux nombres compris entre 0 et 1. Pour ça, il faut implémenter cette fonction à partir de rand (référez-vous au tutoriel de natim sur le sujet pour tout comprendre) :

Code : C

```c
double rand_0_1(void)
{
    return rand()/(double)RAND_MAX;
}
```

Maintenant, pour tirer effectivement notre fléchette, c'est tout simple :

Code : C

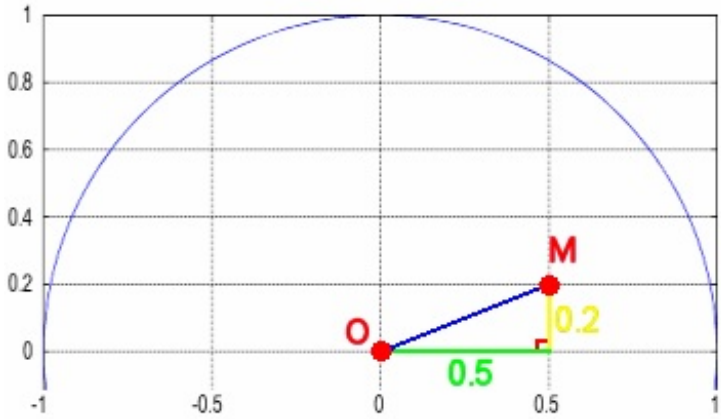
```c
double x = rand_0_1();
double y = rand_0_1();
```

Les variables *x* et *y* contiennent alors respectivement l'abcisse et l'ordonnée de la fléchette.

Sommes-nous dans le disque ?

Pour répondre à cette question, il suffit de définir précisément ce qu'est un disque de rayon *1* et de centre *O*. On peut prendre la définition suivante :

Un disque de rayon 1 et de centre O est l'ensemble des points M tels que la distance OM soit inférieure ou égale à 1.



Pour mieux comprendre cette définition un peu abstraite, un petit exemple : prenons le point *M(0.5,0.2)* ; *M* est-il dans le cercle ?

Pour répondre, utilisons la définition : il faut que $MO \leq 1$. Pour simplifier les calculs, prenons le carré de cette inégalité : $MO^2 \leq 1^2 = 1$.
Or d'après notre cher Pythagore : $MO^2 = (0.5-0)^2+(0.2-0)^2 = 0.5^2 + 0.2^2 = 0.25 + 0.04 = 0.29$.

0.29 est bien inférieur à *1* donc *M* est dans le disque.

Donc dans notre problème, la condition *"le point (x,y) est dans le disque"* s'exprimera :

Code : C

```c
( x*x + y*y ) <= 1;
```

À vous de jouer !

Vous avez maintenant tous les ingrédients pour implémenter l'algorithme. Je vous propose donc de faire un programme qui demande à l'utilisateur combien de lancers il veut faire, puis qui affiche une valeur approchée de pi.

Voici un programme possible :

Secret (cliquez pour afficher)

Code : C

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

double rand_0_1(void)
{
 return rand()/(double) RAND_MAX;
}

int main(int argc, char *argv[])
{
 unsigned int n=0; // le nombre de jets
 unsigned int compteur=0; // le compteur de flechettes dans la
cible
 unsigned int i=0; // le compteur de boucle
 double x,y;
 double pi;
 printf("Combien de jets voulez vous effectuer ? ");
 scanf("%d",&n);

 printf("Simulation en cours, cela peut prendre quelques minutes
... \n");

 for(i=0 ; i< n; i++)
 {
  // Jet de la flechette
  x = rand_0_1();
  y = rand_0_1();

  // La flechette est-elle dans le disque ?
  if( x*x + y*y <= 1 )
  {
   compteur++;
  }
 }

 pi = compteur*4 / (double) n; // "(double)" permet de convertir
n d'un entier vers un double ;
                               // si il n'y est pas, c'est la
division entière qui est effectuée (le résultat est un entier)
 printf("La simulation donne pour valeur de pi : %lf.\n",pi);
 return 0;
}
```

Vous remarquerez que je n'ai pas initialisé rand avec srand ; la suite de nombres générés sera donc toujours la même pour les différentes exécutions du programme. Cela n'a aucune importance ici et cela permet même de comparer les programmes. Par exemple pour 200 000 lancers (plus que Titi ne pourra jamais en faire de toute sa vie), j'obtiens 3.139080, ce que vous devriez obtenir également.

Sans le savoir, vous venez d'implémenter votre première méthode de Monte Carlo ! Vous avez pu vous en apercevoir par vous-mêmes, cette méthode est terriblement simple, le cœur même de l'algorithme tient en 4 lignes. En revanche, elle est relativement peu efficace pour calculer pi. Effectivement, la théorie des Séries de Fourier permet de montrer que :

$$\frac{1}{1}+\frac{1}{4}+\frac{1}{9}+\frac{1}{16}+\ldots+\frac{1}{n^2}+\ldots=\frac{\pi^2}{6}$$

Grâce à cette formule, en s'arrêtant à n = 100, on a 3.1302 comme valeur de pi, puis avec n = 3000, on obtient 3.14127, et les temps de calcul restent imperceptibles sur un ordinateur actuel. Monte Carlo est donc largement dépassé, d'autant qu'il existe d'autres formules donnant pi encore plus rapidement. Ceci étant, personne n'aurait le courage (moi y compris) de justifier et d'expliquer ces méthodes sur le SdZ. Justifier Monte Carlo est relativement simple (du moins dans l'idée), et c'est cette simplicité qui va être un des atouts de cette méthode.

Cette simplicité de l'algorithme a son importance, car même si les méthodes classiques sont plus performantes, il vaut mieux passer 4 heures de moins à coder plutôt que gagner 3 minutes à l'exécution. Ce qui compte c'est la rapidité (temps de codage compris) avec laquelle on obtient le résultat (évidement cette remarque n'a aucun sens dans un contexte "temps réel").

Mais une méthode n'est en général pas choisie uniquement pour sa simplicité, mais aussi pour son efficacité. Ici, le problème était en réalité trop simple pour qu'il soit intéressant d'y appliquer cette méthode. Pour faire une analogie avec le bricolage : pour visser des grosses vis, vous prenez un gros tournevis ; si vous prenez le même tournevis pour des toutes petites vis, vous allez avoir beaucoup de mal et ne serez pas très efficace. C'est exactement le même problème ici : Monte Carlo est un très gros tournevis, et pi, une toute petit vis !

Nous allons donc voir dans quels cas les méthodes "classiques" (des PMT : Petits et Moyens Tournevis) ne peuvent pas résoudre efficacement un problème et donc dans quels cas nous devrons utiliser Monte Carlo. Mais pour cela, il faut comprendre comment fonctionnent ces méthodes "classiques" !

Savez-vous intégrer ?

Nous l'avons vu dans le chapitre précédent, les méthodes de Monte Carlo peuvent servir à calculer des surfaces (il existe d'autres méthodes dites de "Monte Carlo" dont je ne parlerai pas ici). En réalité, nous avons calculé ce qu'on appelle une ***intégrale***... Non, ne partez pas ! Bon tant pis, je continue tout seul. Mais qu'est-ce qu'une intégrale ?

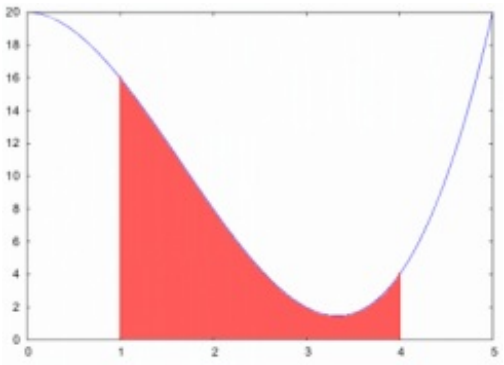
Nous allons voir donc dans cette partie ce qu'est une intégrale, une méthode "classique" de calcul d'une intégrale, puis un calcul par Monte Carlo. Cela nous permettra de comparer les deux méthodes.

Pour aborder ce tutoriel tranquillement, vous devez parfaitement maitriser les notions de fonction et de représentation d'une fonction.

Ce chapitre renvoie à d'autres articles du Web (notamment un tutoriel du SdZ) ou techniques mathématiques. Cependant, le tutoriel a été pensé pour se suffire à lui-même. Admettez donc les choses en première lecture pour ne pas vous embrouiller ; mais plus tard, n'hésitez pas à suivre les liens ou faire des recherches sur les sujets vous intéressant.

Définissons les choses

De façon simple, l'intégrale d'une fonction positive f entre deux points a et b peut-être vue comme la surface sous la courbe représentant f entre $x=a$ et $x=b$. On note cette surface : $\int_{[a,b]} f(x)dx$ (lire : "l'intégrale de f entre a et b" ; on dira aussi que l'on "intègre f sur l'intervalle $[a,b]$"). Sur l'image de droite, la partie rose saumon correspond à l'intégrale de 1 à 4 de la fonction $f(x) = x^3-5x^2+20$ représentée en bleu. Dans ce cas, on note alors : $\int_{[1,4]} (x^3 - 5x^2 + 20)dx$.



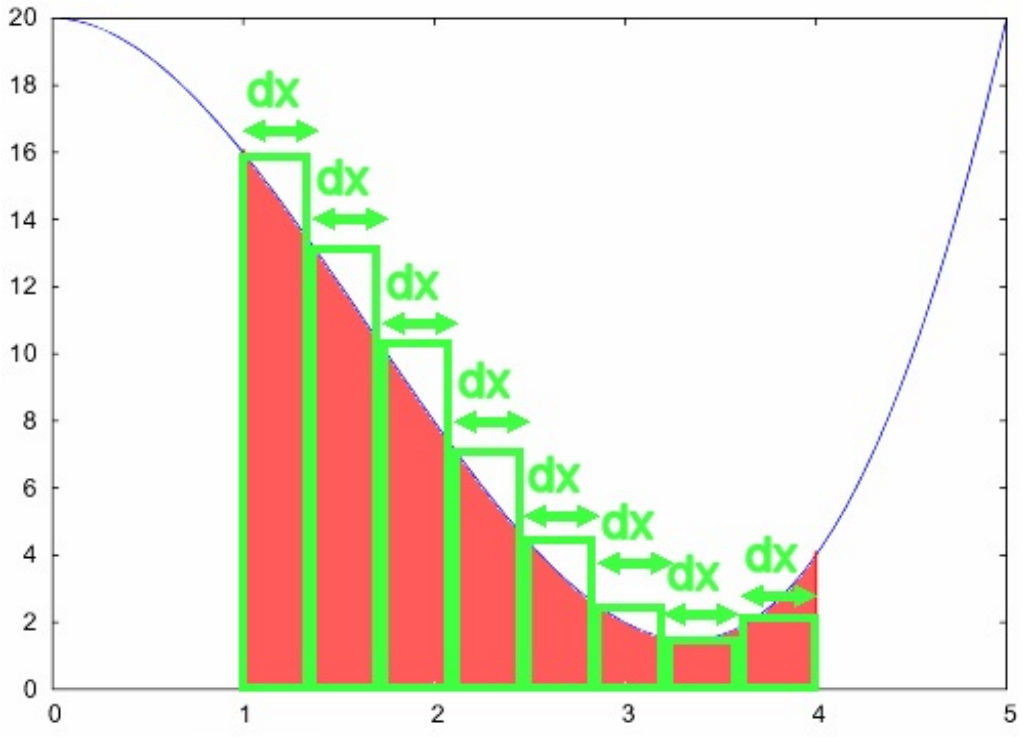
En Terminale, on utilise plus la notation : $\int_a^b f(x)dx$. Cela revient exactement au même, mais lorsque nous verrons les intégrales doubles un peu plus loin, cette notation ne sera pas très commode à utiliser.

Mais si la fonction est négative, ce n'est pas défini ?

Si, mais cela ne nous sera pas utile pour ce tutoriel. Si vous voulez en savoir plus, soit vous attendez patiemment la Terminale, soit vous vous renseignez sur Wikiversité. Sachez tout de même que tout ce je vais dire par la suite est vrai quel que soit le signe de f.

Revenons maintenant à la notation $\int_{[a,b]} f(x)dx$. Elle va nous aider à mieux cerner le problème. Le dx représente une variation infinitésimale (=infiniment petite) de x. L'idée est de considérer que sur un intervalle $[x,x+dx]$ la fonction est constante (vu que c'est un intervalle infiniment petit, ça se comprend). Dans ce cas-là, la "surface sous la courbe" entre x et $x+dx$ est un rectangle de hauteur $f(x)$ et de largeur dx dont l'aire est donnée par $f(x)dx$ (comprendre $f(x)$ fois dx).

Le symbole $\int$ est en réalité une sorte de S un peu stylisé, pour *"Somme"*. Donc lorsqu'on le met devant l'expression $f(x)dx$, cela signifie que l'on va sommer (additionner) l'aire de tous les petits rectangles :



Cette figure n'est qu'un schéma, elle ne reflète pas vraiment la réalité de l'intégrale. Il faudrait pour cela diminuer *dx "à l'infini"* ; le nombre de rectangles deviendrait lui, infini (vous comprendrez pourquoi je ne peux donc pas le représenter correctement sur un dessin).

Une méthode de calcul classique

Cependant, cette représentation va nous servir pour calculer ces intégrales. Nous allons calculer l'aire de chacun des rectangles, puis les additionner. Comme vous le voyez sur la figure ci-dessus, nous commettrons bien sûr une erreur car si f décroît entre x et $x+dx$, nous allons compter de la surface "en trop" ; à l'inverse, si elle croît, nous allons en oublier. Cette erreur peut facilement être réduite en prenant dx plus petit (il y aura donc plus de rectangles, donc le calcul prendra plus longtemps).

Seule petite contrainte "pratique", il faut qu'un nombre entier de rectangles recouvre l'intervalle. Donc au lieu de choisir arbitrairement dx et avoir toutes les chances que "ça ne tombe pas juste", nous allons choisir le nombre de rectangles et en déduire dx.

Tentons donc de calculer l'aire de la surface rose saumon dont je vous bassine depuis le début. La longueur de l'intervalle est de 3 (= 4-1). Donc si je le découpe en n parties, on aura $dx=3/n$.

À la main

Pour rendre le calcul simple, prenons $n=4$. Comme cela, on pourra écrire tous les détails du calcul. Nous avons donc : $dx=3/4=0.75$. On calcule l'aire du premier rectangle en $x=1$: sa hauteur est $f(1)=16$, sa largeur 0.75, son aire est donc de $16*0.75=12$. Et de même pour les trois autres rectangles.

Au final, on a donc :

$$\int_1^4 (x^3-5x^2+20)dx \simeq f(1)\times 0.75+f(1.75)\times 0.75+f(2.5)\times 0.75+f(3.25)\times 0.75 = 23.95$$

> Remarquez bien qu'on n'évalue pas $f(4)$, chaque multiplication correspond aux aires des rectangles situés respectivement : entre 1 et 1.75, 1.75 et 2.5, 2.5 et 3.25, 3.25 et 4. Tout l'intervalle est donc bien couvert, pas besoin d'aller plus loin !

Vous allez le voir un peu plus bas, notre calcul comporte une assez grosse imprécision, que nous voulons évidemment réduire. L'un des moyens est d'augmenter considérablement le nombre de rectangles ; mais vous vous doutez bien qu'on ne va pas s'amuser à calculer ces intégrales à la main lorsqu'on a 2000 rectangles (même votre prof de maths ne serait pas assez fou pour

ça).

Grâce à notre ordinateur

Je vous laisse chercher comment coder ça. Pour faire simple, contentez-vous de demander à l'utilisateur le nombre de rectangles à utiliser pour le calcul. Ne tentez pas de faire saisir une fonction à l'utilisateur, cela vous demanderait trop de travail en C (par contre si vous utilisez un langage de script, avec la fonction *eval*, c'est un jeu d'enfant).

Secret (cliquez pour afficher)

Code : C

```c
#include <stdio.h>
#include <stdlib.h>

// La fonction a intégrer : f(x)=x^3 - 5x^2 +20
double f(double x){return x*x*x - 5*x*x + 20; }

int main(int argc, char *argv[])
{
  double surface = 0;
  double dx=0;
  double x=0;
  int n=0;

  printf("En combien de parties voulez-vous decouper l'intervalle [1,4]? ");
  scanf("%d",&n);

  // Détermination du dx
  dx = 3./n;
  // On intègre de 1 a 4 avec un pas de dx
  for( x=1 ; x<4 ; x+=dx )
  {
 // Calcul de l'aire de chaque petit rectangle que l'on ajoute à la surface totale
 surface += f(x)*dx;
  }

  printf("L'integrale de f entre 1 et 4 est approximee par %lf \n",surface);

  return 0;
}
```

Les lecteurs attentifs auront remarqué que j'ai utilisé le verbe *"estimer"* pour la méthode de Monte Carlo et *"approximer"* pour cette méthode. Même si ces deux mots semblent proches, il y a une différence fondamentale entre les deux : lorsqu'on "estime", on a en général aucune idée sur l'erreur que l'on commet ; lorsqu'on "approxime", oui. Ce calcul d'erreur est un aspect fondamental des méthodes que je vous présentent, mais demande une analyse mathématique trop poussée pour être développée ici.

Le résultat pour 100 rectangles est de 19.05, et pour 1000 rectangles, il est de 18.77. Cette fonction étant relativement simple, il existe des moyens de calculer exactement son intégrale (cf. cours de Terminale) ; ce calcul nous donne 18.75. On voit donc que notre approximation n'est pas trop mauvaise, mais avoir une telle erreur avec 1000 rectangles est une bien piètre performance, faites moi confiance. Il existe d'autres méthodes que celle-ci (qui est appelée tout simplement "méthode des rectangles"), comme la méthode des trapèzes, de Simpson (rien à voir avec Bart ou Homer ) ou de Newton-Cotes. Ces méthodes sont (beaucoup) plus efficaces mais reposent sur le même principe de base : découper l'intervalle en plusieurs parties sur lesquelles on approche f par quelque chose de plus simple.

Vous allez me dire : "C'est bien beau tout ça, mais c'est quoi le rapport avec Monte Carlo ?". Pour l'instant, aucun. Mais nous allons voir dans le chapitre qui vient comment Monte Carlo peut nous aider à calculer cette intégrale d'une manière complètement

différente que celle que nous avons vue.

Le calcul par Monte Carlo

L'idée de l'algorithme

L'idée de l'algorithme de Monte Carlo est très simple : on tire un point x_1 au hasard dans l'intervalle *[a,b]* sur lequel on veut intégrer, et on considère que la fonction vaut $f(x_1)$ sur tout l'intervalle. Son aire est donc : *"longueur de l'intervalle"*f(x1) = (b-a)*f(x1)*. Notons cette aire A_0.

Évidement, si on s'arrête ici, le résultat a toutes les chances d'être très faux. On va donc recommencer avec d'autres nombres x_2, x_3, x_4, ..., x_n. Et de la même façon, on aura $A_2=(b-a)*f(x_2)$, $A_3=(b-a)*f(x_3)$, ..., $A_n=(b-a)*f(x_n)$.

Mais ça revient au même ! Toutes ces aires sont aussi fausses que la première !

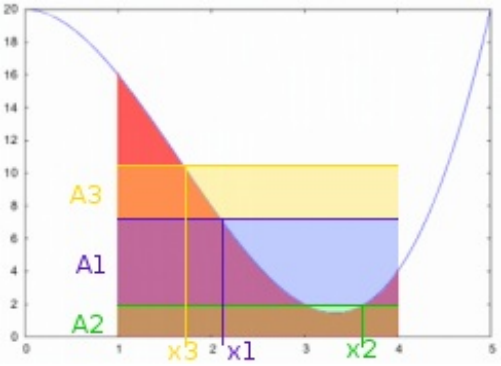
Très juste ! Mais maintenant, on en a beaucoup (*n* pour être précis ). Et lorsqu'on a un grand nombre de données aléatoires mesurant un paramètre, ce paramètre est - à peu près - égal à la moyenne de ces données.

Vous n'avez rien compris à cette phrase ? C'est pourtant un principe que vous appliquez tout le temps ; par exemple à l'école : **une** note à **un** contrôle ne veut pas dire grand chose (il se peut que vous ayez cramé votre carte mère, le drame affreux qui vous empêche de vous concentrer). Donc pour évaluer votre niveau, vos profs vont prendre plusieurs notes (qui, prises séparément n'ont pas de sens) et en faire cette fameuse moyenne (qui elle a un sens).

En théorie, cette moyenne reflète votre niveau et vous permettra (ou pas) d'accéder aux écoles de vos rêves (je dis bien en théorie car je ne suis pas convaincu qu'une note puisse refléter un quelconque niveau ).

Avec les aires que nous avons calculées, c'est la même chose. L'estimation de l'intégrale s'écrit donc :

$$\int_1^4 f(x)dx \simeq \frac{A_1 + A_2 + ... + A_n}{n}$$



Le calcul concret

Je ne vous ferai pas l'affront de faire le calcul pour un *n* petit, vous pouvez le faire par vous-mêmes ; d'autant que si le nombre de tirages n'est pas très important, de par la nature de la méthode, le résultat sera entaché d'une erreur. Passons donc directement à son implémentation.

Il nous manque une petite fonctionnalité pour écrire cet algo. : tirer un nombre au hasard compris entre deux autres. On peut faire ça en réutilisant la fonction rand_0_1 (je vous renvoie encore une fois au tutoriel de Natim sur le sujet) :

Code : C

```
double rand_a_b(double a, double b)
{
   return rand_0_1()*(b-a) + a;
}
```

Je ne donne que le "cœur" du programme, le reste changeant très peu ; il suffit donc de remplacer la boucle `for` du programme précédent par celle-ci (et le texte demandant le nombre de rectangles par un texte demandant le nombre de points à tirer) :

Code : C

```
int i;
for( i = 0 ; i<n ; i++)
{
    // On tire un point au hasard :
    x = rand_a_b(1,4);
    // Calcul de l'aire du rectangle que l'on ajoute à la surface
totale
    surface += f(x)*3; // (3=4-1 est la longueur de l'intervalle
[1,4])
}
// On divise par N pour avoir la moyenne
surface = surface / n;
```

Ici, nous n'avons aucun moyen de connaître l'erreur, donc nous n'avons pas approximé, mais estimé la valeur de l'intégrale.

J'ai fait tourner l'algo. et c'est complètement pourri ! Avec la méthode des rectangles, on avait beaucoup plus de précision et le code n'était pas plus compliqué. Monte Carlo ça sert à quoi finalement ?

Effectivement, sur cet exemple, l'intérêt de la méthode n'est pas encore évident. Je voudrais quand même vous faire remarquer que si on tire un nombre **infini** de points, cette méthode est équivalente à la méthode des rectangles. L'infini est un bien beau concept, mais malheureusement ce n'est qu'un concept, non approchable dans la pratique ; donc en pratique, la méthode des rectangles sera bien plus efficace que Monte Carlo pour calculer cette intégrale.

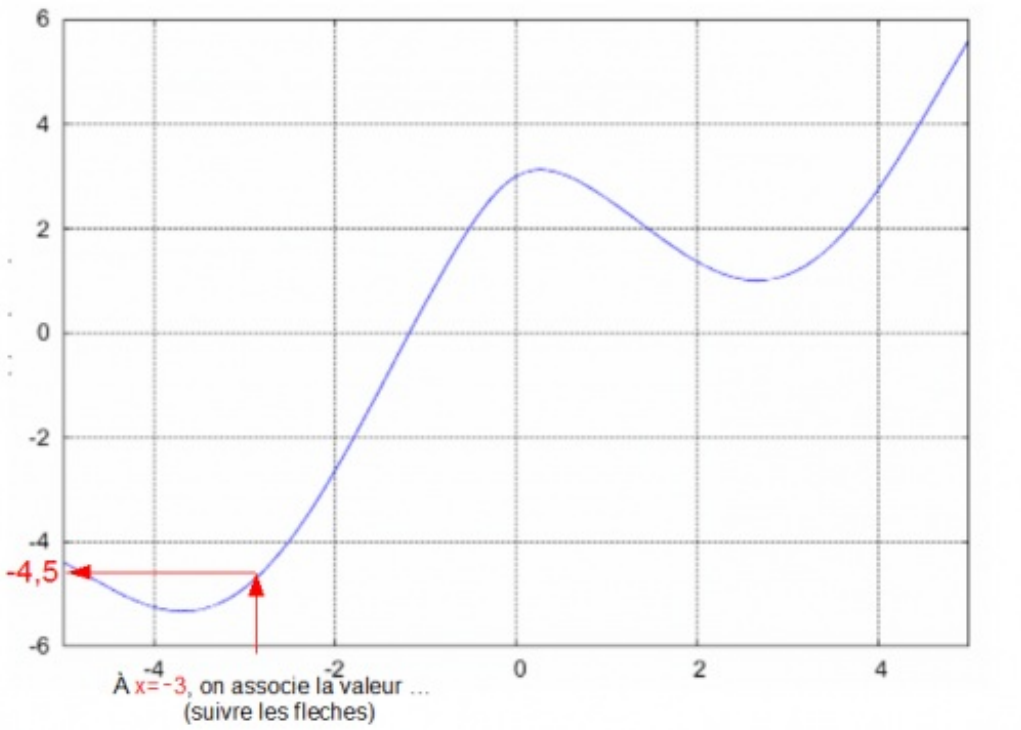
Pour commencer à voir où Monte Carlo sera efficace, il faut voir ce qu'il en est du calcul intégral d'une fonction... à deux variables ! Accrochez-vous bien, le chapitre suivant va un peu décoiffer, mais le résultat vaudra le coup.

Passez dans les dimensions supérieures

Nous allons parler ici des fonctions à deux variables. Pourquoi s'arrêter à 2 ? Tout simplement car la difficulté se situe au niveau de passage de la première à la deuxième dimension. Après, une fois qu'on a bien compris comment ça marche, on a compris pour trois, quatre, voire n dimensions (après pour les dimensions infinies, c'est une autre histoire).

Les fonctions à deux variables, ou comment avoir de jolies courbes en 3D

Si vous avez survécu jusqu'ici, vous connaissez parfaitement les fonctions à une variable : $f(x) = 4x+5$, par exemple. Cette fonction associe à chaque valeur de $\mathbb{R}$ (l'ensemble de tous les nombres), une autre valeur dans $\mathbb{R}$. On représente souvent ces fonctions à l'aide d'un graphe en 2 dimensions :

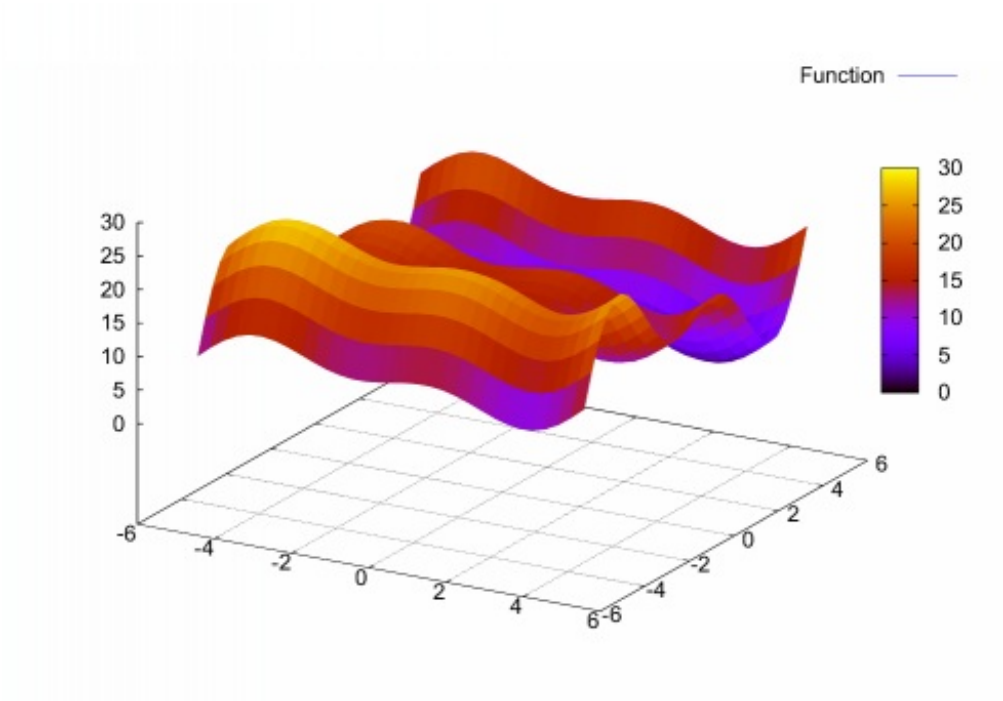


Je pense que je ne vous apprends pas grand chose. Maintenant, au lieu d'associer un point de $\mathbb{R}$ à un autre, nous allons prendre un point $\mathbb{R} \times \mathbb{R} = \mathbb{R}^2$ auquel nous associons un point de $\mathbb{R}$.

Euh... $\mathbb{R}^2$?

Vous connaissez déjà cet espace sans le savoir. $\mathbb{R}$ peut se représenter comme une droite infinie. Eh bien $\mathbb{R}^2$ se représente comme un plan infini (une surface toute plate qui s'étend aussi loin que vous voulez). En général, un "point" de $\mathbb{R}^2$ est noté (x,y).

Une fonction à deux variables est donc quelque chose de la forme : $f(x,y)=x*cos(x)+3y*cos(y)+15$. Maintenant, si on veut un moyen assez efficace de la représenter, on ne peut plus utiliser un graphe en deux dimensions. On a alors plusieurs possibilités : associer une couleur à un nombre, tracer des lignes de niveaux, ou, si notre carte graphique ne date pas de la préhistoire, utiliser une représentation 3D. C'est ce que nous allons utiliser ici. Par exemple, la fonction que j'ai donnée ci-dessus est représentée par le graphe suivant (j'ai aussi utilisé les couleurs, pour une meilleure lisibilité) :



C'est joli, mais comment ça se lit ?

On peut voir ce graphe comme une carte de relief d'une région vallonnée. Par exemple, vous souhaitez savoir quelle est l'altitude en *(2,4)* : on se place grâce à la grille en bas (c'est cette grille qui représente $\mathbb{R}^2$) sur l'intersection de la ligne 2 et de la colonne 4 ; on "remonte" ensuite pour lire l'altitude ; grâce à la couleur, on voit que c'est autour de 5.

Il y avait bien sûr un moyen plus précis de connaître cette altitude : calculer directement $f(2,4) = 2\cos(2) + 3*4\cos(4) + 15 = 6.32398$.

J'anticipe déjà vos questions : *"Mais comment tu fais pour faire des jolis graphes comme ça ?"*. Vous avez deux solutions : soit utiliser Gnuplot directement, soit utiliser un logiciel comme Maxima qui utilise lui-même Gnuplot. Maxima est plus qu'une solution d'affichage, c'est un logiciel complet de calcul formel qui peut résoudre des équations, calculer des intégrales et pleins d'autres choses. Je vous conseille d'utiliser l'interface wxMaxima qui est très conviviale et permet de créer rapidement des courbes sans connaître aucune commande (il existe une autre interface graphique, xMaxima, qui est beaucoup moins pratique à mon goût).

Maintenant que nous avons vu ce qu'était une fonction à deux variables, intéressons-nous à ce que signifie une intégrale de ces fonctions.

Doublez les intégrales

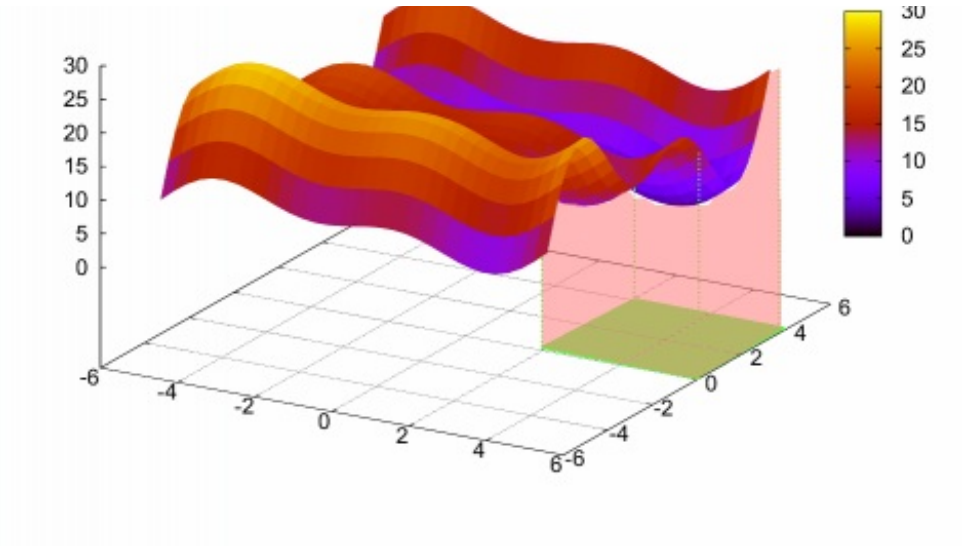
Juste avant, je vous ai parlé d'intégrales d'une fonction à une variable sur un intervalle, ou autrement dit, un segment ; ce sont des intégrales **simples**, dans le sens qu'il n'y a qu'une seule variable (et non parce que c'est facile). Voyons ce que sont les intégrales **doubles**.

Accrochez-vous bien et prenez votre temps ; cette partie est la plus délicate du tutoriel. Si vous n'êtes pas parfaitement réveillé, faites une sieste et revenez après !

Une histoire de volume

Avec des fonctions à deux variables, on n'intègre plus sur un segment, mais sur une surface ; cette intégrale ne représente plus une aire, mais le

volume contenu sous la courbe représentant la fonction. Pour l'instant (pour faire simple), nous intègrerons sur le carré *[0,4]x[2,6]*, c'est à dire l'ensemble des points *(x,y)* tels que :



$0 \leq x \leq 4 \quad et \quad 2 \leq y \leq 6$. Regardez le schéma à droite pour bien comprendre ce qu'il se passe : en vert, j'ai dessiné le carré *[0,4]x[2,6]*, et en rose transparent, j'ai représenté l'intégrale de la fonction sur ce carré (cette intégrale est donc un volume).

On note ça : $\iint_{[0,4]\times[2,6]} f(x,y)dx\ dy$.

Notez bien les différences avec la notation précédente :

- pour indiquer que l'on intègre sur une surface, on met deux symboles $\int$ au lieu d'un seul ;
- il y a deux variations *infinitésimales* : *dx* et *dy*, ces symboles indiquent par rapport à quelles variables on intègre.

Le retour des rectangles

Évidemment, ce qui nous intéresse ici est de calculer ces intégrales. Nous allons commencer par une méthode "classique", qui est l'analogue de la méthode des rectangles en une dimension (celle que nous avons vue juste avant). On va voir que le principe de la méthode est d'utiliser ce qu'on sait déjà faire, c'est à dire calculer des intégrales simples. C'est un procédé assez fréquent en maths : se ramener à des trucs plus simples qu'on sait déjà faire.

Dans la plupart des cas (du moment que vous ne vous frottez pas à l'infini, c'est bon), on a l'égalité suivante :

$$\iint_{[0,4]\times[2,6]} f(x,y)dx\,dy = \int_{[0,4]} \left(\int_{[2,6]} f(x,y)dy \right) dx = \int_{[2,6]} \left(\int_{[0,4]} f(x,y)dx \right) dy$$

Oui, bien sûr... Et ça veut dire quoi ?

Concrètement, que l'on peut calculer cette intégrale soit en intégrant d'abord par rapport à *x*, puis, par rapport à *y* soit dans l'ordre inverse, cela ne change rien (ça paraît "évident", mais malheureusement, cela ne l'est pas). Bon, je sens que c'est un peu abstrait pour vous, voici comment on pourrait calculer cette intégrale avec la méthode des rectangles :

Prenons déjà dx=dy=1 (le calcul aura une grosse erreur, mais c'est juste pour bien comprendre sans s'encombrer de constantes sans intérêt). Je choisis d'intégrer d'abord par rapport à *y*, autrement dit, je vais calculer :

$$I = \int_{[0,4]} \left(\int_{[2,6]} f(x,y)dy \right) dx$$

C'est ici que l'astuce du russe intervient : on pose

$$g(x) = \left(\int_{[2,6]} f(x,y)dy \right)$$

Et ça, on sait très bien le calculer, il suffit d'appliquer la méthode des rectangles vue en première partie. Donc avec notre *dy=1*, on a :

$$g(x) = f(x,2) * 1 + f(x,3) * 1 + f(x,4) * 1 + f(x,5) * 1$$

(je vous laisse, coder ça en exercice, cela revient pratiquement au même que le code de la première partie).

Et maintenant, on n'a plus qu'à calculer :

$$I = \int_{[0,4]} g(x)dx$$

ce qui est exactement pareil que ce que l'on a fait avant (remplacez le *g* par le *f*).

Au final, on a :

$$\begin{aligned} I &= g(0) * 1 + g(1) * 1 + g(2) * 1 + g(3) * 1 \\ &= f(0,2) + f(0,3) + f(0,4) + f(0,5) \\ &+ f(1,2) + f(1,3) + f(1,4) + f(1,5) \\ &+ f(2,2) + f(2,3) + f(2,4) + f(2,5) \\ &+ f(3,2) + f(3,3) + f(3,4) + f(3,5) \end{aligned}$$

Je vous invite très fortement à implémenter cet algo. Toutes les difficultés ont été aplanies, il suffit de manipuler un peu le code donné dans la partie précédente.

Bon, qu'est qu'on a fait en réalité ? Prenez un peu de recul, vous vous rendrez compte que l'on a simplement "quadrillé" notre petit carré ! Pour les intégrales simples, à une dimension, on découpait l'intervalle en petits segments ; ici, c'est le même principe sauf qu'on découpe le grand carré en carrés plus petits.

J'aurais très bien pu commencer à intégrer par rapport à x, on aurait eu :

$$g(y) = \left(\int_{[2,6]} f(x,y)dx \right)$$

Et en refaisant les mêmes calculs, on serait retombés sur la même chose à la fin.

Commentaires sur la méthode

On se rend compte assez aisément que pour calculer cette intégrale double, on a besoin de deux boucles imbriquées (même si l'imbrication de boucles est cachée par l'utilisation de fonctions intermédiaires). Si on avait une fonction à 3 variables, le calcul de son intégrale (qui n'est donc ni une surface, ni un volume mais un objet de dimension 4) demanderait trois boucles imbriquées.

Et ainsi de suite, pour une fonction à *n* variables, le calcul de son intégrale demanderait alors *n* boucles imbriquées. Je ne vous raconte pas la galère pour coder ça (on peut le faire en utilisant une fonction récursive, mais ça reste assez lourd) !

On voit donc ici une certaine limite de la méthode.

Et avec Monte Carlo, c'est mieux ?

On commence ici vraiment à toucher l'intérêt de cette méthode. Et cette fois, pas besoin de formules compliquées comme avec la méthode des rectangles, on utilise directement :

$$\iint_{[0,4]\times[2,6]} f(x,y)dx\ dy$$

Comment ça marche ?

Comme en dimension 1, on va tirer un point (x_1,y_1) dans l'espace sur lequel on veut intégrer (dans notre cas, le carré $[0,4]x[2,6]$).

Puis on calcule le volume du parallélépipède de base carré $[0,4]x[2,6]$ (les dimensions de ce carré sont :4x4, donc son aire est de 16) de hauteur $f(x_1,y_1)$.

On nomme le volume ainsi calculé V_1.

Il faut bien voir qu'on multiplie $f(x_1,y_1)$ par l'aire de la surface sur laquelle on intègre pour avoir ce volume. C'est important pour la suite d'avoir cette idée en tête.

On calcule de même V_2, V_3, ..., V_n ; puis on fait la moyenne de ce petit monde pour estimer l'intégrale, comme en une dimension.

À vous de jouer !

Je vous propose donc de coder un programme qui calcule l'intégrale de la fonction f(x,y)=x*cos(x)+3y*cos(y)+15 sur le carré $[0,4]x[2,6]$ en utilisant la méthode de Monte Carlo. Vous aurez besoin de la fonction *rand_a_b* que l'on a définie au chapitre précédent, et de vous inspirer du programme faisant le calcul à une dimension.

Secret (cliquez pour afficher)

Code : C

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// La fonction a integrer
double f(double x, double y){return x*cos(x)+3*y*cos(y)+15; }

double rand_0_1(){   return rand()/(double)RAND_MAX; }
double rand_a_b(double a, double b){ return rand_0_1()*(b-a) +a; }

int main(int argc, char *argv[])
{
 double  volume  = 0;
 int  n       = 0; //le nombre de points a tirer
 int  i;   //le compteur de boucles
 double x,y;      // (x,y) sera le point que l'on tire au hasard

 printf("Combien de points voulez-vous tirer pour calculer
l'integrale ? ");
 scanf("%d",&n);

 for( i=0; i < n ; i++)
 {
  // On tire un point au hasard dans le carre [0,4]x[2,6]
  x = rand_a_b(0,4);
  y = rand_a_b(2,6);
  volume += f(x,y) * 16; // 16 est l'aire du carre
 }
 volume = volume/n;

 printf("L'intégrale de la fonction sur le carre [0,4]x[2,6] est
estimee a %lf",volume);

 return 0;
}
```

Vous pouvez maintenant tester cet algo. et comparer le résultat avec la valeur exacte : 195.85... Si vous faites le comparatif avec la méthode des rectangles, vous verrez que Monte Carlo ne s'en sort pas trop mal à partir de 1000 tirages.

En réalité, plus la dimension est importante, mieux Monte Carlo s'en sortira par rapport aux méthodes classiques. C'est un résultat qui découle directement du Théorème de la limite centrale. C'est un théorème assez dur que je peux difficilement vous expliquer (cela me prendrait un tutoriel entier), mais pour simplifier très grossièrement, voyez ça comme une application de la loi des grands nombres : si vous lancez 10 fois une pièce, il est très probable que 70% des lancers tombent sur face ; par contre, si vous faites 1000 lancers, il est quasi improbable d'avoir une telle proportion.

Retenez que lorsque la dimension augmente, il devient plus intéressant (moins couteux et plus précis) d'utiliser Monte Carlo. Donc premier avantage !

Les patatoïdes arrivent !

Et nous avons aussi un autre avantage clair en faveur de Monte Carlo : la simplicité du code. Souvenez vous, pour n variables, il fallait n boucles imbriquées. Ici, on voit bien que quel que soit le nombre de variables, on n'a besoin que de la boucle de répétition du tirage (plus éventuellement une boucle pour itérer sur les variables).

L'avantage est déjà flagrant lorsqu'on intègre sur des rectangles, parallélépipèdes ou objets équivalents en dimensions supérieures, mais cet avantage est encore plus marqué lorsqu'on intègre sur des formes un peu plus exotiques. En effet, en prenant un peu de recul, on constate que la méthode des rectangles consiste à quadriller l'espace sur lequel on veut intégrer ; Monte Carlo nous épargne ce travail de quadrillage !

Par exemple, admettons que nous ayons l'idée, humaine, certes, mais bizarre, d'intégrer sur un "patatoïde". Comment ferions nous ?

C'est quoi un patatozormachin ? Encore un mot pour nous embrouiller !

Non ce n'est pas un nom scientifique, juste une vue de l'esprit ; un patatoïde, comme son nom l'indique, désigne tout objet se rapprochant de près ou de (très) loin à une patate (vous noterez la précision de la définition). En fait, je l'emploie pour désigner des formes un peu biscornues et irrégulières, comme ce que vous avez à droite (n'y voyez là aucune tentative de création artistique abstraite, juste un dessin fait avec Gnu Paint en 12 secondes).

Comment appliquer la méthode des rectangles à ce genre de formes ? C'est possible (tout est possible), mais c'est assez dur de faire ça efficacement. Avec Monte Carlo, c'est un jeu d'enfant... Comme lancer des fléchettes sur une cible (attention, Titi revient) !

Mais assez disserté pour ce chapitre ! Vous verez tout ça dans le chapitre qui vient, qui est un TP d'application **concret** de cette méthode.
Bravo à vous ! Vous venez de terminer la partie vraiment dure du tuto ; maintenant que vous avez compris, ça va être de la rigolade et vous allez vraiment prendre du plaisir. Félicitations si vous avez tenu jusque là ; honnêtement de mon côté j'ai eu du mal , j'espère avoir trouvé le juste milieu entre "dire trop de choses pas toutes utiles" et "ne pas donner assez d'explications" ! N'hésitez pas à commenter les parties de ce chapitre (et du précédent) qui vous laissent perplexes, je tenterai de corriger au mieux.

Je conclurais ce chapitre en revenant à ce que nous avons fait au premier chapitre. Pour calculer pi, nous avons en réalité appliqué la méthode de Monte Carlo à une fonction à 2 variables assez simple qui vaut 1 sur le quart de cercle, et 0 partout ailleurs !
Comme quoi, comme M. Jourdain, vous faisiez du Monte Carlo sans le savoir !

Allez, un petit café et on se retrouve au dernier chapitre pour un TP final (à venir prochainement).

Je remercie particulièrement OujA, alexzero, Catsoulet, Zmatt, hugo125, Layus, souls killer, yoch et mon papa (qui a le mérite de n'être ni matheux ni *geek* et d'avoir lu mon tutoriel en entier sans broncher) pour leurs relectures attentives de cette partie.