La méthode de Monte Carlo

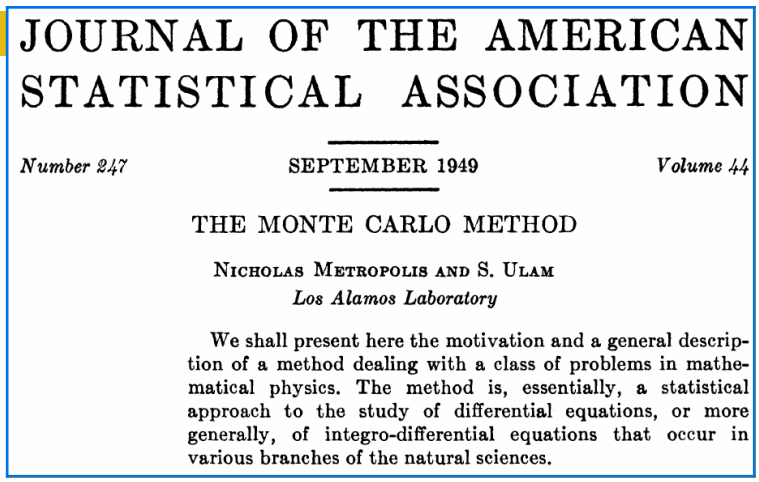
Présentation et objectifs

Dans le programme (spécialité Terminale)

Probabilités	**Calcul intégral / Exemples d'algorithme**
Espérances. Loi des grands nombres.	Méthode de Monte-Carlo.

Une histoire atomique

La méthode dite de Monte-Carlo est une méthode visant à approcher une solution d'une équation mathématique, voire toute valeur numérique, en utilisant des procédés aléatoires, c'est-à-dire des techniques probabilistes. Le nom de ces méthodes, qui fait allusion aux jeux de hasard pratiqués à Monte-Carlo, a été inventé en 1947 par Nicholas Metropolis, et publié en 1949 dans un article coécrit avec Stanislaw Ulam lors du développement de l'arme nucléaire[1].



JOURNAL OF THE AMERICAN STATISTICAL ASSOCIATION

Number 247 SEPTEMBER 1949 *Volume 44*

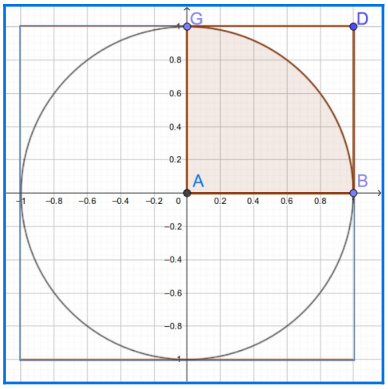
THE MONTE CARLO METHOD

NICHOLAS METROPOLIS AND S. ULAM
Los Alamos Laboratory

We shall present here the motivation and a general description of a method dealing with a class of problems in mathematical physics. The method is, essentially, a statistical approach to the study of differential equations, or more generally, of integro-differential equations that occur in various branches of the natural sciences.

Situation déclenchante

Comme exemple de recherche de valeur numérique, nous allons voir comment trouver des valeurs approchées du nombre π en traitant le problème d'un point de vue probabiliste. La précision du résultat sera directement liée au nombre de répétitions qui seront réalisées. De ce fait, la précision du résultat sera liée à la durée de la simulation.



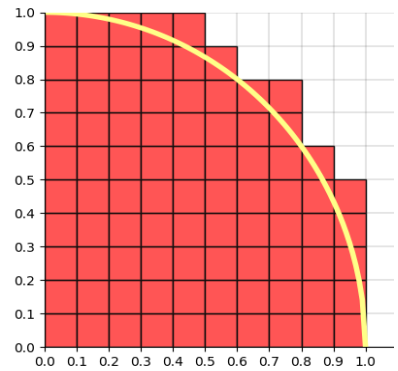
Le nombre π, considéré comme la surface délimitée par un cercle de rayon 1 unité, vaut approximativement 3,14159 unités. Comment la déterminer par simulation de Monte-Carlo ? Il s'agit de représenter cette valeur comme une proportion qui sera considérée comme l'espérance d'une variable aléatoire.

Pour ce faire, nous traçons un disque de rayon 1 unité et l'englobons dans un carré de côté 2 unités, donc d'aire 4 unités d'aire, tangent au cercle en 4 points, comme sur la figure ci-contre.

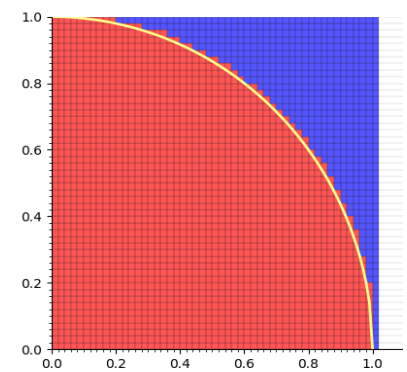
Nous considérons alors l'aire du quart de disque grisé divisée par l'aire du carré ABDG, valant π/4 unités d'aire. Si nous disposons une grille très fine sur ce carré, formée par M lignes horizontales et M lignes verticales (M étant un très grand entier), nous avons M^2 petits carrés qui se répartissent en deux sortes : les carrés « rouges » dont le coin en bas à gauche se trouve à l'intérieur du quart de disque, et les carrés « bleus » qui ne sont pas de ce type. On admet

1 Plus précisément, il s'agissait de modéliser la trajectoire moyenne des neutrons dans un réacteur ou pendant une explosion nucléaire.

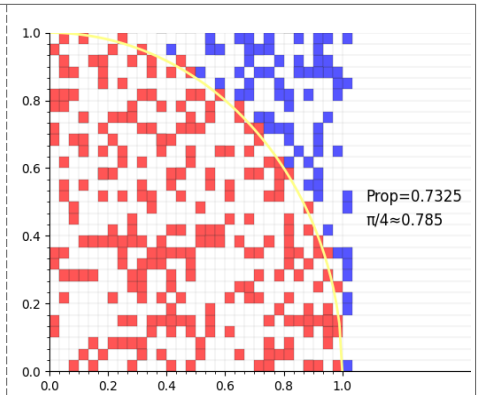
aisément que le total des aires des carrés rouges est proche (et supérieur) à l'aire du quart de disque grisé (et d'autant plus proche que M est grand), de sorte que la proportion des carrés rouges parmi l'ensemble des petits carrés (notons-la p) est une approximation de $\pi/4$ (d'autant meilleure que M est grand).



Couverture (du carré et du quart de disque) par 100 petits carrés



Même couverture, par 2500 = 50×50 très petits carrés



Exemple de tirage (400 répétitions), proportion de « succès » = 73,25%

Or, approcher des proportions au moyen d'expériences aléatoires est exactement l'objectif de la Statistique. Nous procédons donc à un tirage aléatoire de petits carrés au sein du carré ABDG (avec une loi uniforme sur les M^2 petits carrés) ; soit la variable aléatoire Z valant 1 si le petit carré tiré est rouge et 0 sinon. La probabilité d'avoir Z=1 est exactement p, et c'est aussi l'espérance de Z. Pour estimer p, on considère un grand nombre n de tirages indépendants, suivant la même loi que Z, de sorte que quand n tend vers l'infini, la proportion de carrés rouges obtenus converge vers l'espérance de Z, soit p. Le nombre π peut donc être approché par 4 fois la proportion de carrés rouges.

Pour tirer un petit carré au hasard, il suffit de tirer l'abscisse et l'ordonnée de son coin en bas à gauche ; pour cela, on trouve dans le module `random` une fonction `random` qui réalise ce dont nous avons besoin.

En effet, cette fonction tire une valeur au hasard parmi $M=2^{53}$ valeurs régulièrement réparties entre 0 et 1 : c'est bien un très grand nombre (valant environ 10^{16}).

Objectifs

1. Écrire un script permettant de générer une approximation du nombre $\pi/4$ à l'aide de l'approche probabiliste décrite ci-dessus.
2. Examiner si les approximations semblent converger lorsque n tend vers l'infini : tester l'écart entre $\pi/4$ et l'approximation trouvée pour un nombre de tirages variant de 1000 en 1000, ou croissant plus rapidement encore.
3. Modifier le script pour approcher l'aire comprise entre la courbe représentative de la fonction carré, l'axe des abscisses, les droites d'équation $x=0$ et $x=1$.

L. DIDIER & R. CABANE

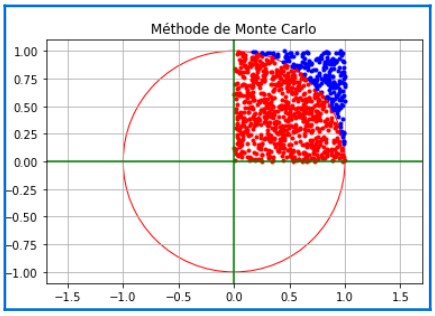
Fiche méthode

Objectif 1 : proposition de résolution

Stratégie

On crée deux fonctions dans ce script :

- Une « fonction de décision » `d` qui prend comme arguments deux réels `x` et `y` et qui renvoie le nombre réel correspondant au carré de la distance OM entre O(0,0) et M(x,y) diminué de 1 (l'usage de la racine carrée n'est ici pas utile et consomme du temps de calcul) ; on teste si `d(a,b)` est négatif pour savoir si on est dans la zone souhaitée.
- Une fonction `mc` qui prend comme argument un entier naturel *n* et renvoie un nombre réel correspondant à la fréquence (ou proportion observée) des carrés générés aléatoirement se situant dans le quart de cercle rouge sur le dessin ci-contre.




```
EDITEUR : MONTECAR
LIGNE DU SCRIPT 0012
from random import random
def d(x,y):
  return x*x+y*y-1
def mc(n):
  c=0
  for i in range(n):
    a=random()
    b=random()
    if d(a,b)<=0:
      c=c+1
  return c/n
Fns… | a A # | Outils | Exéc | Script
```

Étapes de résolution

On commence par assurer la nécessaire importation de la bibliothèque `random`, avant de définir les fonctions `d` et `mc`.

On peut afficher la valeur approchée de $\pi/4$ pour comparer avec la simulation et ainsi observer la précision du résultat de la simulation.

On notera que l'approximation n'est pas bonne pour des petites valeurs de n (il faut prendre au moins $n=10000$ pour une approximation au centième près). Par ailleurs, le temps de calcul peut être très important, déjà 35 secondes pour `mc(80000)` sur une TI-83 Premium CE Edition Python, moins de 3 secondes sur une TI-Nspire™ CX II-T. C'est la caractéristique des méthodes de Monte Carlo : pour atteindre une précision correcte il faut un nombre d'itérations très élevé et accepter des temps de calcul énormes (ou disposer de matériel très performant)[2].

```
from math import *
from random import *
from ti_system import *
def d(x,y):
  return x*x+y*y-1
def mc(n):
  c=0
  for i in range(n):
    a=random()
    b=random()
    if d(a,b)<=0:
      c=c+1
  return c/n
def arithm():
  X1=[k*1000  for k in range(2,80)]
  store_list("X1",X1)
  L1=[mc(n) for n in X1]
  store_list("L1",L1)
```

Objectif 2

Nous pouvons créer un script Python qui va appeler de manière répétée la fonction `mc` et tester l'écart avec la valeur cible. Pour se rendre compte de la convergence le mieux est de prendre des valeurs de *n* assez rapidement croissantes, par exemple de 1000 en 1000 ou suivant une suite quadratique (basée sur les carrés des entiers).

2 Le matériel utilisé par Metropolis et Ulam (ENIAC) était électromécanique, car les circuits intégrés n'existaient pas encore, pas plus que les transistors ; d'où des temps de calcul énormes (des heures ou des jours), imposant d'éviter absolument les erreurs logicielles (ou « bugs »). L'usage du mot « bug » provenait des insectes qui entraient dans les relais et provoquaient des faux contacts …

L. DIDIER & R. CABANE

Sur une calculatrice TI-Nspire™ CX II-T, on peut aisément explorer les valeurs de `mc(n)` pour n allant jusqu'à 80000 (ou plus).

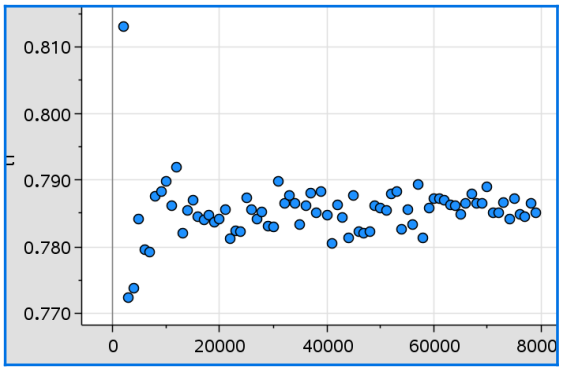
Une représentation graphique permet de dégager une idée intuitive du phénomène. Le plus commode pour cela est de créer une liste et de la traiter dans l'environnement « natif » de la calculatrice grâce à la fonction `store_list` prévue à cet effet dans la bibliothèque `ti_system`.

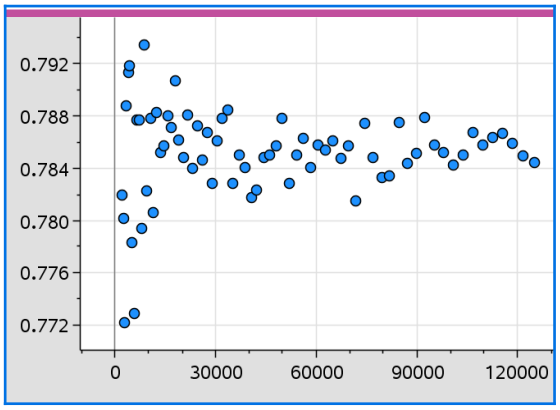
La représentation graphique est alors facile à faire à l'aide des outils fournis par la machine.

Nspire CX L'instruction `store_list("L1",L)` crée une liste nommée L1 dans le système, liste qui peut s'afficher dans l'environnement « données et statistiques » ; on crée similairement une liste X pour les abscisses.

On constate que les approximations ne convergent que très lentement, et assez irrégulièrement (côté aléatoire du processus).

Si on veut explorer plus loin, il est intéressant de donner à n des valeurs croissant plus rapidement, en suite quadratique plutôt qu'arithmétique (ici, $n=20k^2$, k entier). Le temps de calcul peut alors devenir important ; exemple ci-dessous.





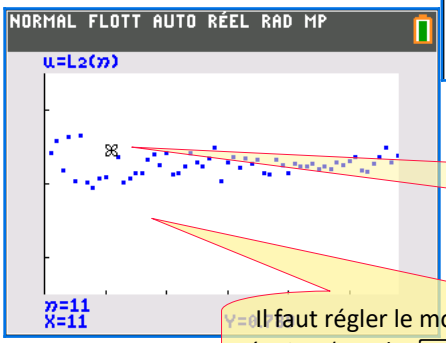
On emploie ici une « liste en compréhension » (voir Appendice 1 pour plus de détails).

*MonteCar3.py

```
def quadrat():
    X2=[20*k*k  for k in range(10,80)]
    store_list("X2",X2)
    L2=[mc(n) for n in X2]
    store_list("L2",L2)
```

TI-83 On procède de manière semblable. Ici, l'instruction `store_list("1",L)` permet de copier le contenu de la liste L dans la variable système L_1.

La représentation graphique est alors facile à faire à l'aide des outils fournis par la machine avant de tracer.




```
L1=[i*1000 for i in range(2,60)]

store_list("1",L1)
L2=[mc(n) for n in L1]
store_list("2",L2)
```

Le mode de tracé est interactif et permet de « suivre » la suite en bougeant le curseur.

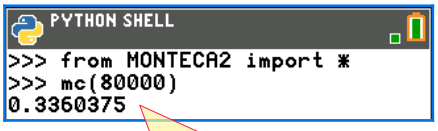
Il faut régler le mode graphique en « suite » et « point épais » (touche mode), puis définir la suite (touche f(x)) et définir la suite $u(n)=L_2(n)$ et enfin la fenêtre.

La lenteur de cette calculatrice ne permet pas d'explorer autant qu'avec une TI-Nspire™ CX II, à moins d'une grande patience !

Objectif 3

On va modifier le script précédent pour approcher l'aire comprise entre la courbe représentative de la fonction carré, l'axe des abscisses, les droites d'équation $x=0$ et $x=1$, c'est-à-dire l'intégrale $\int_0^1 x^2\,\mathrm{d}x=\frac{1}{3}$.

Il suffit en fait de modifier la « fonction de décision » d en y codant l'équation de la parabole ; le reste du programme est inchangé.



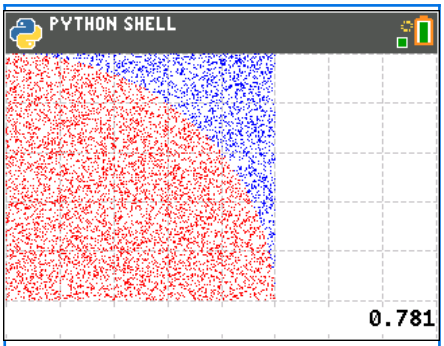
Comme annoncé, la précision est mauvaise à moins de faire un très grand nombre de calculs …

Pour aller plus loin

Approfondissement (TI 83 ou Nspire CX II)

TI-83 **Nspire CX-II** En approfondissement, on peut travailler sur la représentation graphique de la simulation. Il faudra donc importer la bibliothèque `ti_plotlib` de manière à pouvoir effectuer la représentation graphique. On rajoutera donc au code précédent uniquement des lignes qui vont gérer les paramètres de la représentation graphique (voir l'Appendice 2) :

- en préliminaire le cadrage de la fenêtre,
- puis à chaque itération le choix de la couleur selon le positionnement,
- puis l'affichage d'un pixel correspondant au tirage au hasard,
- et à la fin l'affichage du graphique.



8192 tirages de points au hasard

```
EDITEUR : MONTECA3
LIGNE DU SCRIPT 0002
from random import random
import ti_plotlib as plt
def d(x,y):
  return x*x+y*y-1
def mc(n):
  plt.cls()
  plt.window(0,1.6,-.15,1)
  plt.axes("off")
  plt.grid(.2,.2,"dash")
  c=0
  for i in range(n):
    a=random()
    b=random()
    if d(a,b)<=0:
      c=c+1
      plt.color(255,0,0)
    else:
      plt.color(0,0,255)
    plt.plot(a,b,".")
  plt.color(0,0,0)
  s=str(round(c/n,3))
  plt.text_at(12,s,"right")
  plt.show_plot()
  return c/n
```

Remarque : dans l'instruction :

`plt.text_at(12,str(c/n),"right")`

on met l'instruction `str` devant `c/n` de manière à le convertir en chaîne de caractères car c'est le type attendu dans l'instruction `plt.text_at`.