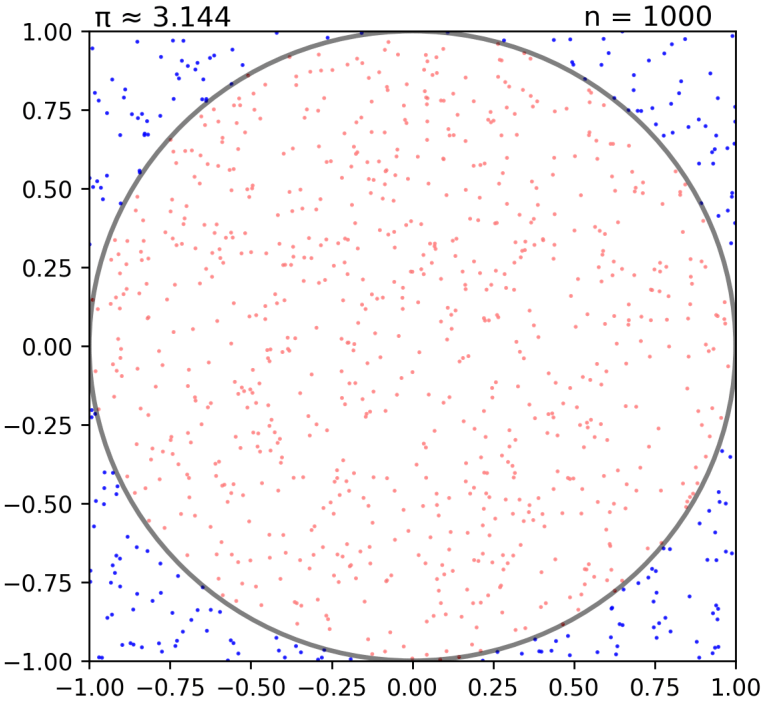
TP

Monte Carlo Simulation et intervalles de confiences



CHASSAGNOL Rémi
ZZ2 - F2 - promo24

Professeur: HILL Davide

TODO

Contents

1 Simulation de π

Le but de cette première partie est de déterminer une valeur approchée de π en utilisant une méthode de **Monte Carlo**. Le principe est le suivant:

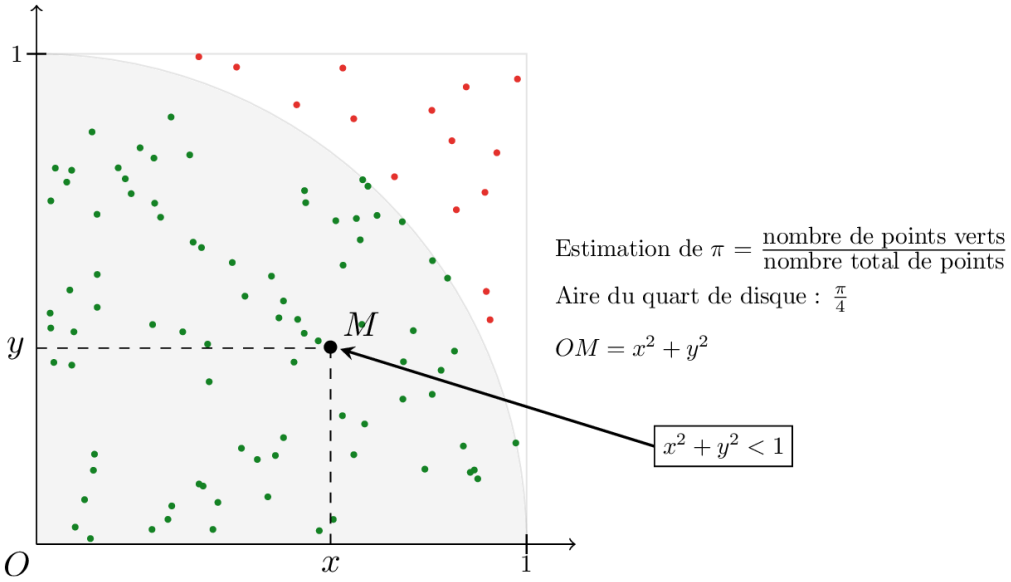


Figure 1: Simulation π Monte Carlo

[1]

On tire deux nombres aléatoires x et y, compris entre 0 et 1, à l'aide de **Mecene Twister**. C'est deux nombres sont les coordonnées d'un point appartenant un à espace représenté par un carré de coté 1.

Ensuite, on teste si $x^2 + y^2 < 1$, si c'est le cas, cela signifie que le point trouvé se trouve dans le quart de cercle de rayon 1 situé à l'intérieur de notre carré comme ci-dessus. À la fin on fait le rapport du nombre de points trouvés dans le quart de cercle sur le nombre de points total et on obtient une valeur approchée de $\frac{\pi}{4}$. Là il suffit de multiplier par 4 pour trouver π.

[1] Figure 1: `https://media.eduscol.education.fr/ftp_eduscol/2019/Ressources/Mathematiques/RA19_Lycee_G_1_MATH_Algorithmique_et_Programmation_activite_11.html`

Pour l'implémentation, on va utiliser `genrand_real1()` pour avoir des nombres aléatoire dans $[0;1]$ et on suit les étapes expliquées précédemment. La fonction prend en paramètre le nombre de points à générer. Comme Mercene Twister est un bon générateur qui explore bien l'espace, plus on génère de points, plus on couvre l'espace et ce de manière *"uniforme"*. De ce fait, plus on augmente le nombre de points générés, plus la précision augmente.

```
double simPi(unsigned int nbPoints) {
  unsigned int pointsInCircle = 0;
  unsigned int i;
  double       x, y;

  for (i = 0; i < nbPoints; ++i) {
    x = genrand_real1();
    y = genrand_real1();

    if (x * x + y * y < 1) {
      pointsInCircle++;
    }
  }

  return (double)(4 * pointsInCircle) / nbPoints;
}
```

Voici le résultat pour différents nombres de générations:

```
pi 1000: 3.124000
pi 1000000: 3.144720
pi 1000000000: 3.141541
```

On constate bien que plus on génère de points, plus on se rapproche de la valeur de π.

2 Calcul de précision

Dans cette deuxième partie, on va réutiliser la fonction `simPi()` implémentée précédemment, et on va s'intéresser à la précision. Le but va être de calculer une moyenne sur un certain nombre de simulation de π. Pour chacune des moyennes, la simulation se fera avec différents nombres de points à générer. Ensuite, on compare la valeur de π trouvée (en moyenne) avec `M_PI` qui est une macro de `math.h` qui contient une valeur précise de π.

Notre fonction `simPiPrecisison()` va donc prendre deux paramètre:

- le nombre de points à générer avec `simPi()`
- le nombre d'expériences à effectuer pour le calcule de la moyenne

La fonction affiche simplement le pourcentage de différence entre la valeur de π simulée et `M_PI`.

```
void simPiPrecisison(unsigned int nbPoints, int nbSimulations) {
  int    i;
  double mean;
  double sum = 0;

  initMT();

  // calcul des pis
  for (i = 0; i < nbSimulations; ++i) {
```

```
    sum += simPi(nbPoints);
  }
  mean = sum / nbSimulations;

  printf("mean: %lf\n", mean);
  printf("precision: %lf\n", fabs(mean - M_PI) / M_PI);
}
```

Voici les résultats pour des moyennes calculées sur 40 Simulations:

points générés	moyenne	précision
1000	3.142400	0.000257
1000000	3.141186	0.000129
1000000000		