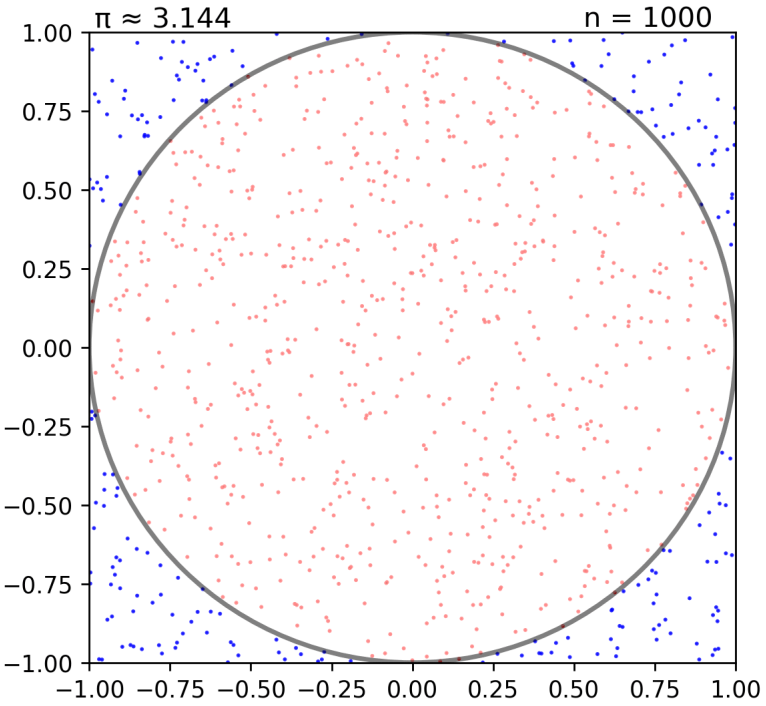
TP

Monte Carlo Simulation et intervalles de confiances



CHASSAGNOL Rémi
ZZ2 - F2 - promo24

Professeur: HILL Davide

7 octobre 2022

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:

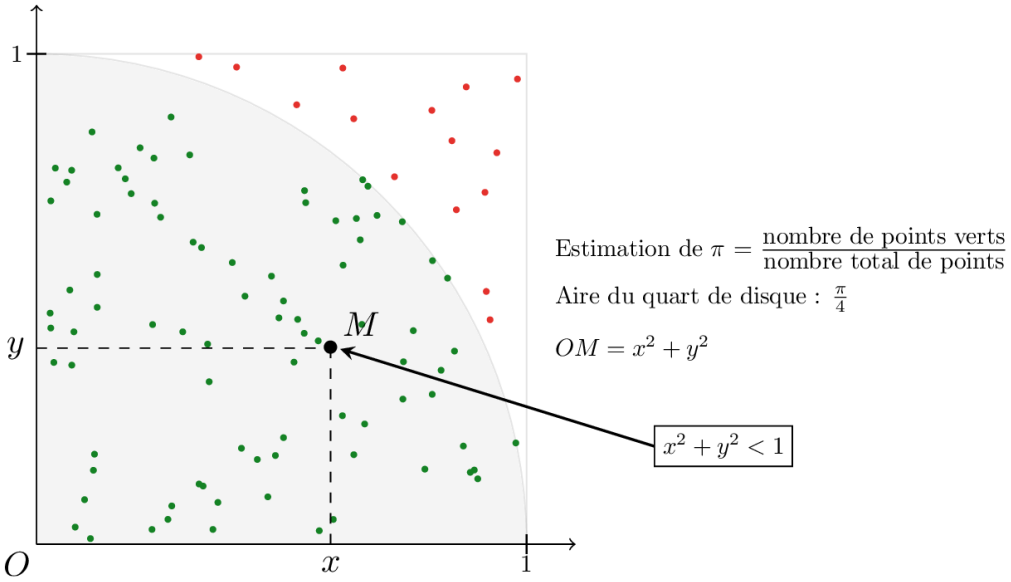


Figure 1: Simulation π Monte Carlo

[1]

On tire deux nombres aléatoires x et y, compris entre 0 et 1, à l'aide de **Mecene Twister**. Ces 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:

points générés	valeur de π trouvée
1000	3.124000
1000000	3.144720
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 relativement précise de π.

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

- `nbPoints`: le nombre de points à générer avec `simPi()`
- `nbSimulations`: 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`. A noter qu'on initialise bien **Mercene Twister** à chaque fois car on souhaite répéter la même expérience, on ne change que le nombre de points générés par `simPi()`.

```
void simPiErreur(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("%lf\t", mean);
  printf("%.10lf\n", fabs(mean - M_PI) / M_PI);
}
```

Voici les résultats pour des moyennes calculées sur 40 échantillons, on constate à nouveau que plus on génère de points plus la précision augmente.

points générés	moyenne	erreur
1000	3.142400	0.000257
1000000	3.141186	0.000129
1000000000	3.141586	0.000002

On calcule maintenant l'erreur pour **10, 11, ..., 39, 40** échantillons et **1000000000** points générés. Si on normalise les données obtenues (pour plus de lisibilité) et qu'on fait de la régression linéaire on obtient la droite suivante:

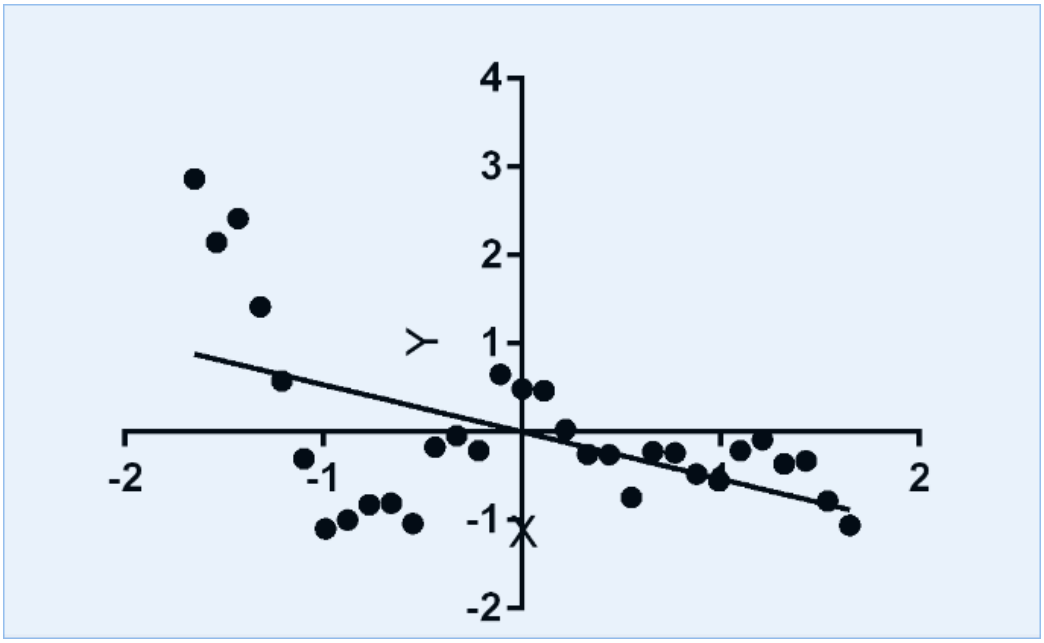


Figure 2: Erreur en fonction du nombre d'échantillons

On voit bien que la droite ci dessus a une pente de presque $-\frac{1}{2}$ ce qui montre bien que pour doubler la précision (diviser l'erreur par 2), il faut multiplier le nombre d'échantillons par 4.

3 Calcul d'intervalles de confiance à 95%

Dans cette dernier partie, on va écrire une fonction qui permet de calculer et d'afficher l'intervalle de confiance à 95% pour les moyennes simulées précédemment. Pour ce faire on commence par calculer la moyenne arithmétique $\overline{X}(n)$ sur un certain nombre de simulations de π. En même temps, on sauvegarde les valeurs de π trouvées dans un tableau car on en aura besoin par la suite. Ensuite on utilise la moyennes et le tableaux de valeurs de π pour calculer $S^2(n) = \frac{\sum_{i=1}^{n}(X_i - \overline{X}(n))^2}{n-1}$, qui est un estimateur sans biais de la variance. Enfin on utilise une loi de Student pour calculer le taux d'erreur $R = t_{n-1,1-\frac{\alpha}{2}} \times \sqrt{\frac{S^2(n)}{n}}$. En utilisant le taux d'erreurs trouvé, on peut calculer les bornes de l'intervalle de confiance à 95% $[\overline{X} - R, \overline{X} + R]$.

La fonction prend en entrée 2 paramètres, `nbPoints` (nombre de points à générer pour `simPi()`) et `n` (nombre d'expériences). On commence par générer `n` valeurs de π et on stocke les résultats dans un tableau. Dans le même temps on calcule la somme des valeurs générées pour le calcul de la moyenne. Après avoir calculer la moyenne, on peux calculer l'estimateur `s2n`. Enfin on calcule R, sachant que les 30 premiers coefficients de Student sont stockés dans le tableau globale `student`. À la fin on affiche l'intervalle.

```
void intervalleConfiance(unsigned int nbPoints, int n) {
  int     i;
  double pis[30] = {0};
  double mean;
  double sum = 0;
  double sumSn = 0, s2n;
  double tmp;
  double r;

  initMT();

  // calcul des pis
  for (i = 0; i < n; ++i) {
    pis[i] = simPi(nbPoints);
    sum += pis[i];
  }
  mean = sum / n; // calcul de la moyenne

  // Calcul de S (n)
  for (i = 0; i < n; ++i) {
    tmp = pis[i] - mean;
    sumSn += tmp * tmp;
  }
  s2n = sumSn / (n - 1);

  r = student[n - 1] * sqrt(s2n / n);

  printf("intervalle de confiance: [%lf; %lf].\n", mean - r, mean + r);
}
```

On trouve les résultats suivant:

- Pour **30** échantillons et **1000** points.
  - taux d'erreur: `0.016745`
  - intervalle de confiance: `[3.122988; 3.156479]`.
- Pour **30** échantillons et **1000000** points.
  - taux d'erreur: `0.000651`
  - intervalle de confiance: `[3.140491; 3.141794]`.
- Pour **30** échantillons et **1000000000** points.
  - taux d'erreur: `0.000019`
  - intervalle de confiance: `[3.141565; 3.141604]`.

Ces résultats sont cohérent par rapport au cours, on constate que plus on utilise de points pour générer π plus la variance est faible et donc plus l'intervalle de confiance est petit.