Compte Rendu TP3
Intervalle de confiance et Fibonacci

Jérémy ZANGLA

February 14, 2020

Contents

1 Préambule

Nous utilisons pour la génération des nombres aléatoires le code de Matsumoto reprennant le générateur de Mersenne Twister. Nous ne décrirons pas son code, et nous n'utiliseront que la fonction genrand_real1 qui nous renvoie un nombre pseudo-aléatoire appartenant à [0; 1].
Tout le code que nous avons développé sera décrit ici. De plus une documentation sera jointe à ce rapport. Enfin une série de résultats obtenus avec différents paramètres sera également jointe.

1.1 Contenu du programme

Le programme va tout d'abord initialisé le générateur pseudo-aléatoire. Ensuite, il affichera 3 estimations de *pi* en utilisant la méthode de Monte-Carlo avec respectivement 1 000, 1 000 000 et enfin 1 000 000 000 de points. Puis le programme fera une série d'estimations de *pi* avec la méthode de Monte-Carlo, en calculera la moyenne et l'intervalle de confiance. Enfin il calculera les 95 premières valeurs de la suite de Fibonacci.

```c
/**
 * @brief Point d'entrée du programme.
 *
 * Tout d'abord nous avons besoin d'initialiser le générateur aléatoire.
 *
 * Nous répondons ensuite aux questions du sujet dans l'ordre.
 *
 * @return int
 */
int main()
{
    double values[NB_VALUES];    // Valeurs de PI calculées par la méthode de Monte-Carlo
    unsigned long fib[100];      // Valeurs de la suite de Fibonacci
    int i;                       // Variable d'itération
    double mean;                 // Moyenne de values
    double variance;             // Variance de values
    double r;                    // Rayon de l'intervalle de confiance de values
    double minR, maxR;           // Intervalle de confiance de values

    // Initialisation du générateur de nombre pseudo-aléatoire
    unsigned long init[4]={0x123, 0x234, 0x345, 0x456}, length=4;
    init_by_array(init, length);

    // Affichage des paramètres utilisés pour obtenir le résultat suivant
    printf("Nombre de valeurs : %d\n Nombre de points : %d\n\n", NB_VALUES, NB_POINTS);

    printf("Question 1 : \n");
    // Calcule de Pi par la methode de Monte-Carlo avec 1 000 points
    printf("PI (%10d) : %.10lf\n",1000, monte_carlo(1000000));
    // Calcule de Pi par la methode de Monte-Carlo avec 1 000 000 points
    printf("PI (%10d) : %.10lf\n",1000000, monte_carlo(1000000));
    // Calcule de Pi par la methode de Monte-Carlo avec 1 000 000 000 points
    printf("PI (%10d) : %.10lf\n",1000000000, monte_carlo(1000000000));


    // A partir de maintenant chaque génération de Monte-Carlo se fera avec NB_POINTS points
    printf("Question 2 : \n");
    // Calcul de Pi comme étant la moyenne des valeurs
    // données par NB_VALUES génération de Monte-Carlo
    mean = replicates_monte_carlo(NB_VALUES, values);
    printf("Mean : %.10f\n", mean); // Affichage de la moyenne calculée

    printf("Question 3 : \n");
    // Calcul de la variance du jeu de valeur
    variance = calc_variance(NB_VALUES, values, mean);

    // Calcul du rayon de l'intervalle de confiance
    r = calc_radius(NB_VALUES, values, variance);
    // Calcul de l'intervalle de confiance
    minR = mean - r;
    maxR = mean + r;

```

```
53      // Affichage des valeurs calculées pour la question 3
54      printf("Var : %.10f\nR : %.10f\n[%.10f;%.10f]\n", variance, r, minR, maxR);
55
56      printf("Question 4 : \n");
57      fibo(95, fib); // Calcul des 96 premières valeurs de la suite de Fibonacci
58
59      for (i = 0; i < 10; ++i) // Affichage des 10 dernières valeurs calculées
60          printf("Fibo[%d] = %lu\n", 85 + i, fib[85 + i]);
61
62      return 0;
63  }
```

2 Méthode de Monte-Carlo

2.1 Description de la méthode

L'algorithme suivi ici est tout simple, on travaille dans le carré déterminé par les points (0; 0) et (1; 1). On utilise également le cercle de centre (0; 0) et de rayon 1, plus particulièrement le quart de cercle appartenant au carré précédent. Prenons maintenant un point P quelconque appartenant au carré. On sait que la probabilité qu'il appartienne au cercle est égale au rapport entre la surface du cercle et la surface du carré. On peut donc généré une multitude de ces points de manière aléatoire et compter le nombre de point appartenant au cercle. En divisant le nombre de point du cercle par le nombre de point total, on retombe bien sur notre probabilité d'appartenir au cercle. Donc, nous pouvons déduire *pi* de la manière suivante :

$$
\begin{aligned}
p &= \frac{surface_{cercle}}{surface_{carre}} \\
\frac{nb\_inner}{nb\_points} &= \frac{surface_{cercle}}{surface_{carre}} \\
\frac{nb\_inner}{nb\_points} &= \frac{\pi \times r^2}{c^2} \\
\frac{nb\_inner}{nb\_points} &= \frac{\pi}{4} \\
4 * \frac{nb\_inner}{nb\_points} &= \pi
\end{aligned}
$$

2.2 Implémentation de la méthode

```
1   double monte_carlo(int nb_points)
2   {
3       double x, y;        // Coordonnée du point généré
4       int i;              // Variable d'itération
5       int nb_inner = 0;   // Nombre de point à l'intérieur du quart de cercle trigonométrique
6
7       for (i = 0; i < nb_points; ++i)
8       {
9           x = genrand_real1(); // On génère un nouveau point
10          y = genrand_real1();
11
12          // Si il est dans le cercle, on incrémente le compteur
13          if (pow(x, 2) + pow(y, 2) < 1)
14              nb_inner++;
15      }
16
17      // On renvoie la valeur de PI (explications plus haut)
18      return 4.0 * nb_inner / nb_points;
19  }
```

Nous nous posons la question de la précision de ce résultat. Nous avons donc utilisé cette fonction avec 1 000, 1 000 000 et enfin 1 000 000 000 de points.

Nombre de points	1 000	1 000 000	1 000 000 000
Valeur de *pi* obtenue	3.124 00	3.144 72	3.141 54
Ordre de grandeur de la précision	10^{-1}	10^{-2}	10^{-3}

On peut conclure rapidement qu'il faut multiplier par 1 000 le nombre de points pour obtenir une précision seulement 10 fois meilleure. On remarque alors la limite de cette méthode de calcul, il faut générer beaucoup de nombres aléatoires pour obtenir une précision assez faible.

2.3 Etude approfondie des résultats

Nous allons maintenant étudié plus en détails les résultats précédents. Pour celà nous allons construire un tableau contenant 100, 1 000 et enfin 10 000 estimations de *pi*. Pour chacun de ses essaies nous allons aussi étudier l'importance du nombre de points, nous allons reprendre les paliers de la question précédente à savoir 1 000 et 1 000 000 de points par estimation de *pi*. Nous ne travailleront plus avec 1 000 000 000 de points par estimations, car le temps de calcul est beaucoup trop important.

2.3.1 Configuration

Pour changer le nombre de points utilisés par estimation de *pi* nous modifierons la constante NB_POINTS. Pour choisir le nombre d'estimations, il faudra changer la valeur de la constante NB_VALUES.

```
1  /**
2   * @def NB_VALUES
3   *
4   * @brief Nombre de valeurs de \pi utilisées pour faire les statistiques.
5   */
6  #define NB_VALUES 10000
```

```
1  /**
2   * @def NB_POINTS
3   *
4   * @brief Nombre de points utilisés pour le calcule de PI
5   * via la méthode de Monte-Carlo, lors des statistiques.
6   */
7  #define NB_POINTS 1000000
```

2.3.2 Implémentation

Nous avons décidé ici de faire une fonction qui stock les estimations de *pi* dans un tableau, tout en calculant la moyenne de ces valeurs.

```
1  /**
2   * @brief Réitère la méthode de Monte-Carlo pour en faire des statistiques.
3   *
4   * Chaque valeur de \pi utilisée pour les statistiques sera calculée
5   * à parti de 1 000 000 000 de points.
6   *
7   * @param nbReplicates Nombre de valeur que l'on veut utilisé pour les futures statistiques.
8   * @param values Tableau dans lequel on va mettre les valeurs de \pi calculées.
9   *
10  * @return double Moyenne de toutes les valeurs calculées.
11  */
12 double replicates_monte_carlo(int nb_replicates, double * values)
13 {
14     int i;          // Variable d'itération
15     double sum = 0; // Somme de chaque valeur de pi calculée
16
17     for (i = 0; i < nb_replicates; ++i)
18     {
19         // Avec 1 000 000 000 et 100 essais : 3.1415838135
20         values[i] = monte_carlo(NB_POINTS);
21         sum += values[i]; // Mise à jour de la somme
22     }
23
24     return sum / nb_replicates; // Calcul de la moyenne
25 }
```

2.3.3 Moyennes des résultats

Nous nous intéressons tout d'abord à la moyenne des valeurs obtenues. Nous faisons une troncataure à 7 chiffres significatifs.

Avec 1 000 points

Nombre d'estimations	Moyenne
100	3.135 120
1 000	3.140 084
10 000	3.142 187

Avec 1 000 000 de points

Nombre d'estimations	Moyenne
100	3.141 660
1 000	3.141 582
10 000	3.141 569

On remarque ici que sur seulement 100 estimations de PI avec 1 000 points la précision est très faible 10^{-1}. Cependant lorsque l'on augmente le nombre de valeurs à 1 000, la précision semble s'améliorer pour atteindre 10^{-2}, ce qui serait confirmer par la moyenne à 10 000 estimations qui n'a pas régressé en précision.

Avec 1 000 000 de points par estimation la conclusion est la même la précision avec seulement 100 estimations est moins précise que pour 1 000 et 10 000 (respectivement 10^{-3} et 10^{-4}).

La question est donc de savoir pourquoi. Faire 1 000 estimations à 1 000 points chacune, nous fait générer 1 000 000 de points et l'on retombe bien sur la précision de 10^{-2} mesurée précédemment. Idem pour 1 000 estimations de 1 000 000 de points, on retombe sur une estimation à 1 000 000 000 de points (précision de 10^{-4}).

Il faut faire attention en traitant ces résultats, en effet, ce sont des moyennes. Nous ne somme donc pas assuré que chacun des éléments du jeu de valeurs possède la même précision que la moyenne. Nous sommes même sûr du contraire, si chaque élément pris individuellement avait la même précision que la moyenne, le nombre d'estimations n'influencerait pas la précision de cette dernière.

2.3.4 Implémentation de l'intervalle de confiance

Pour calculer l'intervalle de confiance d'un jeu de valeurs, il faut tout d'abord calculer sa variance.

```
1  /**
2   * @brief Calcule de la variance du jeu de valeur.
3   *
4   * @param nb Nombre de valeur dans le jeu.
5   * @param vals Tableau contenant les valeurs.
6   * @param mean Moyenne des valeurs du tableau.
7   *
8   * @return double La variance du jeu.
9   */
10 double calc_variance(int nb, double * vals, double mean)
11 {
12     double s = 0;    // Somme des valeurs
13     int i;           // Variable d'itération
14
15     // Calcul de la somme des carrés des écarts entre chaque valeur et la moyenne
16     for (i = 0; i < nb; ++i)
17         s += pow(vals[i] - mean, 2);
18
19     // Division par le nombre de valeur - 1 pour finir de calculer la variance
20     return s / (nb - 1);
21 }
```

On a ensuite besoin du quantile à 95%, il est donné par la table suivante :

```
1  /**
2   * @var t_values
3   * @brief Tableau contenant les quantiles pour le calcul de l'intervalle de confiance.
4   */
5  double t_values[] = {
6      12.706, 4.303, 3.182, 2.776, 2.571, 2.447, 2.365, 2.308, 2.262, 2.228, 2.201, 2.179,
7      2.160, 2.145, 2.131, 2.120, 2.110, 2.101, 2.093, 2.086, 2.080, 2.074, 2.069, 2.064,
8      2.060, 2.056, 2.020, 2.048, 2.045, 2.042, 2.021, 2, 1.980, 1.960
9  };
```

Le quantile dépendant du nombre de valeurs dans le jeu, nous avons donc une fonction qui nous donne directement le bon quantile à partir du tableau et du nombre de valeurs.

```
/**
 * @brief Permet d'obtenir le quantile correspondant au nombre de valeurs que l'on utilisent
 *
 * Les 30 premières valeurs du tableau sont en accès direct. \n
 * Les suivantes correspondent à des intervalles : \n
 *  [30; 40[ : case 30 du tableau \n
 *  [40; 80[ : case 31 du tableau \n
 *  [80; 120[ : case 32 du tableau \n
 *  [120; +inf[ : case 33 du tableau
 *
 * @param nb Nombre de valeurs utilisées.
 *
 * @return double Valeur du quantile.
 */
double get_t(int nb)
{
    return t_values[nb <= 30 ? 30 :
                nb < 40 ? 30 :
                nb < 80 ? 31 :
                nb < 120 ? 32 :
                33 ]; // Accède directement à la bonne case dans le tableau de quantiles.
}
```

Il faut ensuite calculer le rayon de confiance, c'est à dire la distance entre la moyenne et chaque borne de l'intervalle de confiance.

```
/**
 * @brief Calcule le rayon de l'intervalle de confiance.
 *
 * @param nb Nombre de valeur dans le jeu.
 * @param vals Tableau contenant les valeurs du jeu.
 * @param variance Variance du jeu de valeurs.
 *
 * @return double Rayon de l'intervalle de confiance.
 */
double calc_radius(int nb, double * vals, double variance)
{
    return get_t(nb) * sqrt(variance / nb);
}
```

2.3.5 Intervalles de confiance

Nous allons reprendre ici les mêmes paramètres que pour la moyenne ainsi que la troncature à 10^{-6}.

Avec 1 000 points

Nombre d'estimations	Intervalle de confiance	Rayon de confiance	Distance à *pi*
100	[3.124 303; 3.145 936]	0.0108164344	0.017 289
1 000	[3.136 786; 3.143 381]	0.0032974956	0.004 806
10 000	[3.141 170; 3.143 204]	0.0010171427	0.001 611

Avec 1 000 000 de points

Nombre d'estimations	Intervalle de confiance	Rayon de confiance	Distance à *pi*
100	[3.141 314; 3.142 005]	0.0003458297	0.000 412
1 000	[3.141 477; 3.141 687]	0.0001049311	0.000 115
10 000	[3.141 537; 3.141 602]	0.0000324047	0.000 055

L'intervalle de confiance à 95% nous donne comme information que 95% des valeurs du jeu sont dans les bornes. La première chose à étudier en voyant ces intervalles est si *pi* y appartient bel et bien. Ici, c'est vérifié, ensuite on peut étudier la précision des valeurs. C'est à dire regarder la précision de 95% des estimations. Ici, on voit que 95% des valeurs du premiers jeu d'estimations (1 000 points, 100 valeurs) sont approchés à moins de 0.02 de *pi*. Ainsi on sait que pour une utilisation ne nécessitant que cette précision, on peut travailler avec ces valeurs et s'attendre à ce que 95% des valeurs soient fonctionnelles. C'est à dire avoir 5% de résultats qui seront erronés.

Enfin on peut comparer les intervalles entre eux, plus particulièrement les rayons de confiance. On remarque qu'ils se réduisent lorsque l'on augmente le nombre de points par estimations et/ou le nombre d'estimations.

2.4 Conclusion

Cette méthode de calcul de *pi*, n'est vraiment pas performante. Pour obtenir une précision très faible il faut faire beaucoup de calculs (et de générations de nombres aléatoires). On atteint donc plusieurs limitations, la première étant le temps de calcul. Supposons que ça n'est pas un problème, que l'on peut paralléliser suffisamment l'algorithme pour que le calcul soit fait instantanémment, la seconde étant la génération des nombres aléatoires. Nous utilisons ici un générateur pseudo aléatoire, pour pouvoir reproduire les résultats. Malheureusement, comme il est nécessaire d'avoir un nombre de points très grand pour augmenter la précision du résultat, on peut craindre le dépassement de la période du générateur aléatoire lorsque l'on voudrait obtenir une précision supérieure (ou bien même vouloir continuer la découverte des décimales de *pi*).

On peut donc conclure que cette méthode n'est pas efficace pour déterminer des décimales de *pi*.

3 Suite de Fibonacci

Cette partie simule de manière très basique l'accroissement d'une population de lapin (sans décés). L'itération x signifie qu'au bout de x mois, la population de lapins sera de fibo(x) couples, donc 2*fibo(x) lapins.

3.1 Implémentation

Nous implémentons ici la version itérative de la suite de Fibonacci.

```
1  /**
2   * @brief Calcul de la suite de fibonnaci de manière itérative.
3   *
4   * @param n Indice de l'élément que l'on veut calculer
5   * @param vals Liste des valeurs de la suite de fibonnaci jusqu'a l'élément n.
6   * @return unsigned long Valeur numéro n de la suite de fibonacci.
7   */
8  unsigned long fibo(int n, unsigned long * vals)
9  {
10     int i; // Variable d'itération
11     vals[0] = 1; // Initialisation des premières valeurs de la suite.
12     vals[1] = 1;
13
14     for (i = 2; i <= n; ++i) // Calcul de toutes les valeurs jusqu'à celle demandée.
15         vals[i] = vals[i - 1] + vals[i - 2];
16
17     return vals[n]; // Renvoie de la valeur demandée.
18 }
```

3.2 Résultats

On remarque que les résultats sont valables jusqu'à l'itération 92, la 93 n'étant pas la sommes de la 91 et la 92. Cette erreur est causée par la limite de la taille de la donnée (unsigned long) : on dépasse le maximum de ce type.

```
1  Fibo[85] = 420196140727489673
2  Fibo[86] = 679891637638612258
3  Fibo[87] = 1100087778366101931
4  Fibo[88] = 1779979416004714189
5  Fibo[89] = 2880067194370816120
6  Fibo[90] = 4660046610375530309
7  Fibo[91] = 7540113804746346429
8  Fibo[92] = 12200160415121876738
9  Fibo[93] = 1293530146158671551
10 Fibo[94] = 13493690561280548289
```