

Compte Rendu R5A12

GOKCEN Bayram

27 novembre 2023



TD-TP 1

1.1 Générateur de Von Neumann

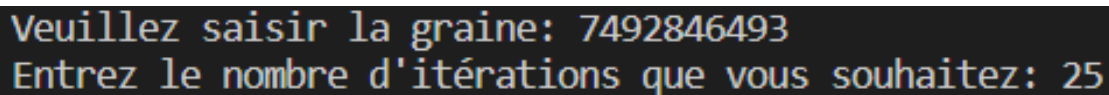
1.1.1 Programmation du RNG

Le générateur de Von Neumann est un algorithme inventé par John von Neumann qui permet de produire des nombres pseudo-aléatoires (compris entre 0 et 9999) à partir d'un nombre source qui se nomme "graine". Cette méthode est très simple, elle consiste à prendre un nombre, à l'élever au carré et à prendre les chiffres au milieu comme sortie et ce plusieurs fois. Cependant, cette méthode a des limitations et n'est pas souvent utilisé de nos jours en raison de problèmes potentiels et d'inefficacité par rapport à d'autres méthodes de génération de nombres aléatoires plus modernes.

1.1.2 Tests et limites du générateur

1.1.3 Cas général

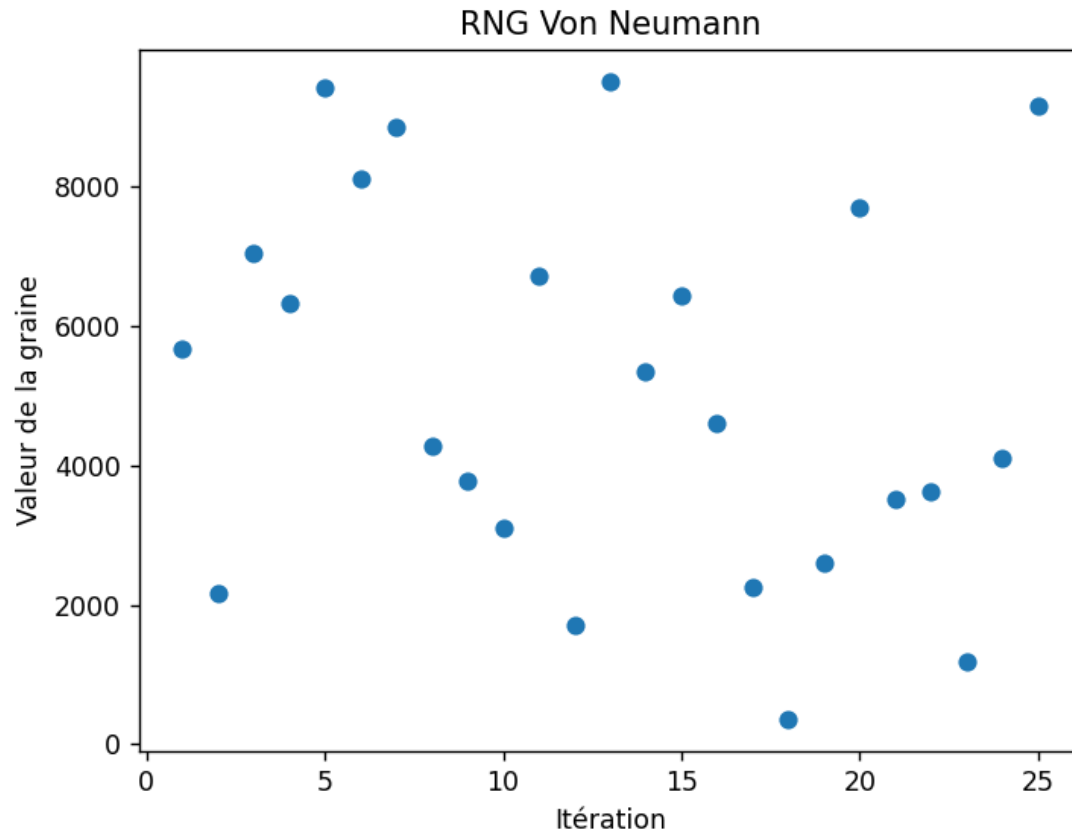
Nous avons choisi, dans un premier temps, de prendre une graine aléatoire et d'itérer la méthode de Von Neumann 25 fois.



```
Veuillez saisir la graine: 7492846493
Entrez le nombre d'itérations que vous souhaitez: 25
```

Avec cette méthode, le générateur nous génère une certaine quantité de nombres pseudo-aléatoires (25 dans notre cas).

Nous avons alors utilisé le plugin python matplotlib pour modéliser les tirages et mieux mettre en évidence l'aléatoire comme vous pouvez le voir dans le graphique ci-dessous :



Nous avons alors un graphique qui représente les 25 tirages dont la valeur est comprise entre 0 et 9999.

Ces valeurs sont donc pseudo-aléatoires, cela veut dire qu'on obtient des valeurs qui s'approchent de l'aléatoire mais qui sont en réalité générées par un algorithme déterministe. Cela nous permet d'obtenir forcément des petites, des moyennes et des grandes valeurs sans risquer d'obtenir, par exemple, que des petites valeurs (ce qui pourrait être le cas avec de l'aléatoire classique).

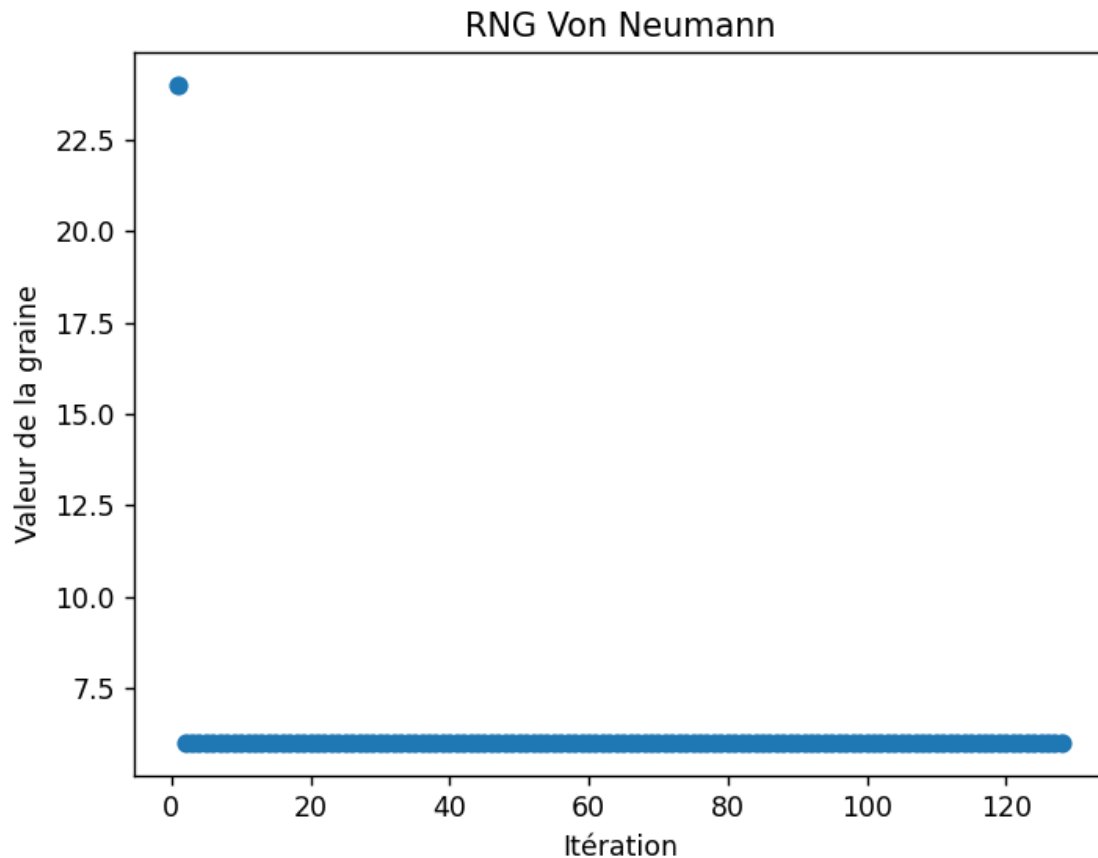
Cependant, nous allons voir que cette méthode possède de nombreux problèmes...

1.1.4 Problème du 0

Le problème du zéro avec la méthode de Von Neumann se produit lorsque la graine utilisée pour générer des nombres aléatoires contient un grand nombre de zéros consécutifs. Lorsque ces zéros consécutifs commencent à intervenir dans les valeurs, toutes les valeurs suivantes deviennent nulles.

```
o Veuillez saisir la graine: 12000001
  Entrez le nombre d'itérations que vous souhaitez: 128
```

Nous utilisons donc une graine qui contient plusieurs 0 consécutifs et nous voulons obtenir 128 valeurs.



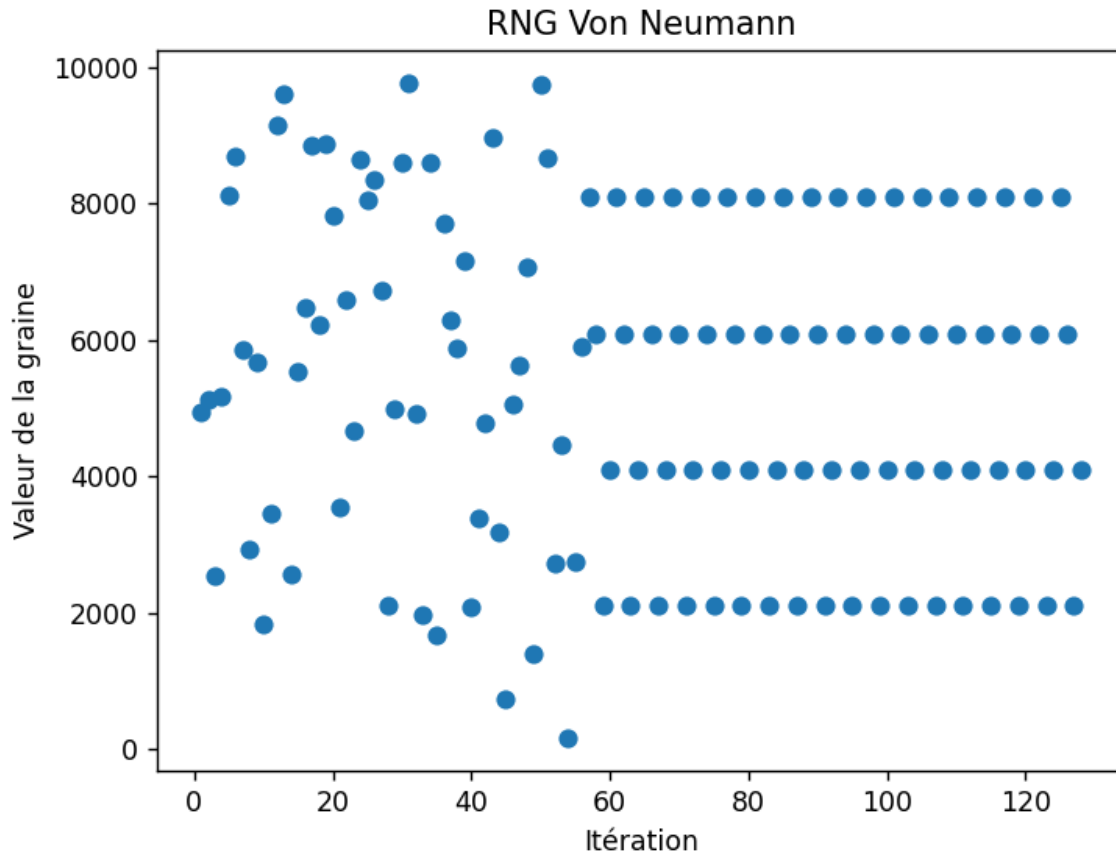
Nous pouvons alors voir que très rapidement les valeurs qu'on obtient valent 0 et restent bloquées à 0. Cette erreur est un réel problème, en effet, dès que la graine contient plusieurs 0, il n'y a plus d'aléatoires et l'algorithme reste bloqué à 0.

1.1.5 Problème de périodicité

Le problème de périodicité est une limitation importante associée à cette méthode de Von Neumann. Si la graine présente des caractéristiques de répétition ou de périodicité, les valeurs que l'on obtient vont également présenter une périodicité, c'est-à-dire qu'elles vont finir par se répéter et on aura tout le temps les quelques mêmes nombres.

```
o Veuillez saisir la graine: 12321321
  Entrez le nombre d'itérations que vous souhaitez: 128
```

On a donc choisi d'utiliser une graine périodique, avec des répétitions pour voir ce que l'on obtient.



On voit bien dans le graphique ci-dessus qu'à partir de la 58ème itération, il y a un cycle périodique qui se crée et les valeurs obtenues sont 2000, 4000, 6000 ou 8000.

En somme, l'algorithme ne parvient pas à briser cette périodicité. Cela signifie que les nombres prétendument aléatoires générés par la méthode de Von Neumann peuvent finir par entrer dans un cycle prévisible, ce qui compromet leur propriété d'aléatoire.

1.2 Générateur "Randu"

1.2.1 Programmation du Randu

RANDU est un générateur de nombres pseudo-aléatoires inventé par IBM dans les années 1960. Il est basé sur une relation de récurrence linéaire simple pour générer des séquences de nombres pseudo-aléatoires.

La formule de récurrence est la suivante : $X_{n+1} = (65539 * X_n) \bmod(2^{31})$

X_n représente le nombre pseudo-aléatoire à l'itération n.

Le nombre 65539 est la constante de récurrence.

Cet algorithme génère des séquences de nombres pseudo-aléatoires en utilisant la for-

mule de récurrence mentionnée ci-dessus. Il faut dans un premier temps choisir la graine (X_0). Ensuite, à chaque itération, l'algorithme utilise la formule de récurrence pour calculer le nombre suivant (X_{n+1}), puis ce nombre est utilisé pour calculer le nombre suivant, et ainsi de suite.

Ainsi on obtient la quantité de nombres pseudo-aléatoires que l'on souhaite.

Cependant, ces nombres sont compris entre 1 et $2^{31} - 1$. Il faut donc que l'on divise le résultat par 2^{31} pour obtenir un réel entre 0 et 1. (si l'on souhaite poser ces valeurs dans un carré de côté unité par exemple.

1.2.2 Problème à comprendre

On nous dit que $X_{n+2} = 6 * X_{n+1} - 9X_n$

On peut, dans un premier temps, regarder si cette égalité est vraie pour une valeur aléatoire :

Prenons une graine qui vaut $X_0 = 94852$.

Alors $X_1 = (65539 * 94852) \bmod(2^{31}) = 1921537932$

Finalement, $X_2 = (65539 * 1921537932) \bmod(2^{31}) = 790955684$

On va donc maintenant voir ce que cela donne avec l'équation que l'on doit vérifier :

$$X_{n+2} = (6 * X_{n+1} - 9X_n) \bmod(2^{31})$$

$$X_2 = (6 * X_1 - 9X_0) \bmod(2^{31})$$

$$X_2 = (6 * 1921537932 - 9 * 94852) \bmod(2^{31}) = 790955684$$

Les deux résultats sont bien les mêmes, cela fonctionne donc pour la valeur $X_0 = 94852$. On va donc maintenant résoudre le problème pour tous les nombres.

On cherche à prouver que :

$$X_{n+2} = (6 * X_{n+1} - 9X_n) \bmod(2^{31})$$

Sachant que :

$$A) X_{n+1} = (65539 * X_n) \bmod(2^{31}) \text{ soit } B) X_{n+1} = ((2^{16} + 3) * X_n) \bmod(2^{31})$$

(En effet, on sait que $65539 = 2^{16} + 3$)

On déduit donc que :

$$C) X_{n+2} = (65539 * X_{n+1}) \bmod (2^{31})$$

(On ajoute simplement 1 à X_n et à X_{n+1} de l'équation A)

$$\text{Soit : } X_{n+2} = ((2^{16} + 3) * (2^{16} + 3) * X_n) \bmod (2^{31})$$

$$\text{Ou encore : } X_{n+2} = ((2^{16} + 3)^2 * X_n) \bmod (2^{31})$$

(On remplace le X_{n+1} de l'équation C par B)

D'ici, on va simplifier $(2^{16} + 3)^2$ afin de soulager notre égalité.

$$\text{On a donc : } (2^{16} + 3)^2 = 2^{32} + 6 * 2^{16} + 9$$

Sachant que $2^{32} = 2 * 2^{31}$, $2^{32} \bmod(2^{31}) = 0$ et $6 * 2^{16} + 9 < 2^{31}$, on peut dire que $(2^{16} + 3)^2 = (6 * 2^{16} + 9) \bmod (2^{31})$.

Afin de se rapprocher du résultat attendu, on va modifier "arbitrairement" la valeur que l'on veut obtenir (cela nous permettra de vérifier l'égalité) :

$$6 * 2^{16} + 9 = 6 * 2^{16} + 9 + 9 - 9 = 6 * 2^{16} + 18 - 9 = 6 * 2^{16} + (6 * 3) - 9$$

Cela peut paraître inutile, voire encombrant mais grâce à cette méthode, on peut factoriser :

$$6 * (2^{16} + 3) - 9$$

Avec ce résultat, on peut retourner sur notre égalité de départ :

$$X_{n+2} = ((2^{16} + 3)^2 * X_n) = (6 * (2^{16} + 3) - 9) * X_n$$

Il ne nous reste plus qu'à développer :

$$(6 * (2^{16} + 3) - 9) * X_n = 6 * (2^{16} + 3) * X_n - 9 * X_n$$

$$\text{Or, on sait que } X_{n+1} = (2^{16} + 3) * X_n$$

$$\text{On a donc : } X_{n+2} = 6 * X_{n+1} - 9X_n$$

On peut donc dire qu'un problème de la méthode RANDU est le choix du multiplicateur 65 549. En effet, on retrouve la relation suivante : $X_{n+2} = 6X_{n+1} - 9X_n$

Les multiplicateurs 6 et 9 sont alors bien trop petits, de fait que si X_n ou X_{n+1} n'est modifié que légèrement, X_{n+2} sera également peu modifié. Idéalement, on voudrait qu'un changement moindre modifie du tout au tout le résultat pour obtenir des résultats complètement différents.

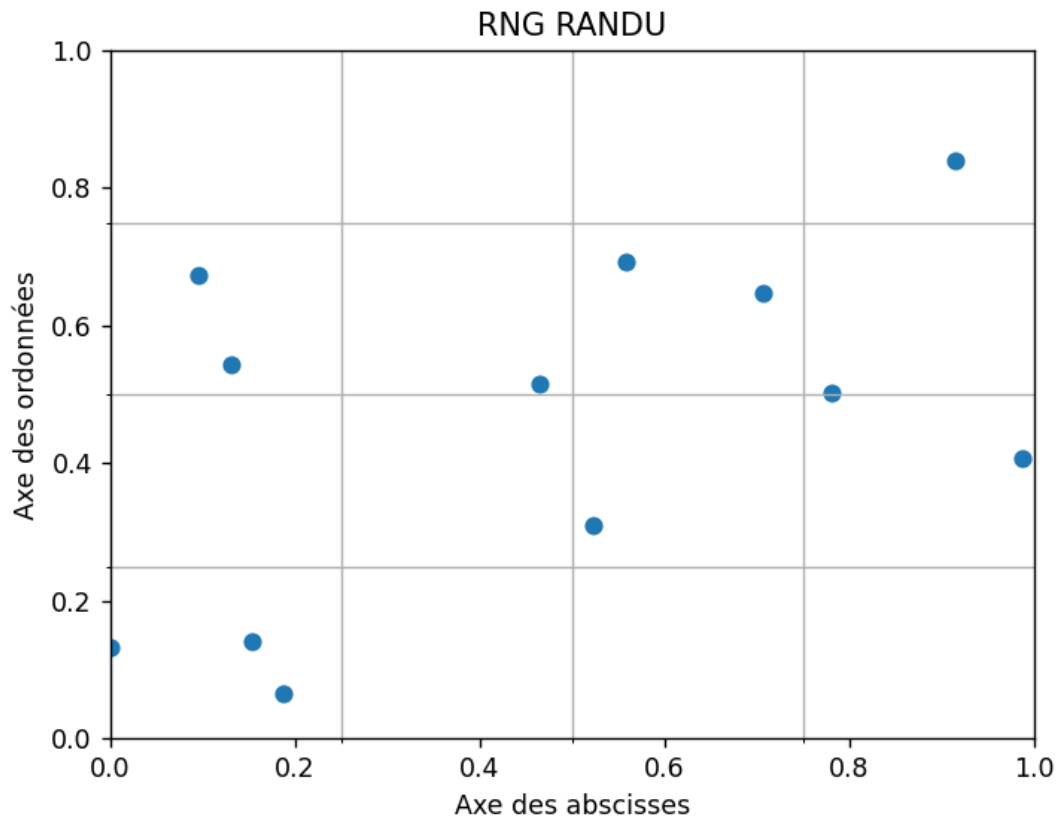
1.2.3 Représentation graphique

Nous allons alors utiliser cet algorithme de Randu pour obtenir différentes valeurs comprises entre 0 et $2^{31} - 1$ puis diviser cette valeur par 2^{31} pour obtenir des valeurs comprises entre 0 et 0,9999..

On réalisera alors des paires de valeurs pour obtenir des coordonnées x et y. Finalement, on mettra ce point de coordonnées x et y $\in [0, 1[$ dans un carré de côté unité (1 sur 1).

Prenons un premier exemple, avec une graine classique, 12 itérations (donc 24 valeurs en prenant x et y).

```
Veuillez saisir la graine: 1544322
Entrez le nombre d'itérations que vous souhaitez: 12
Entrez le nombre de sous carrés que vous souhaitez: 16
```



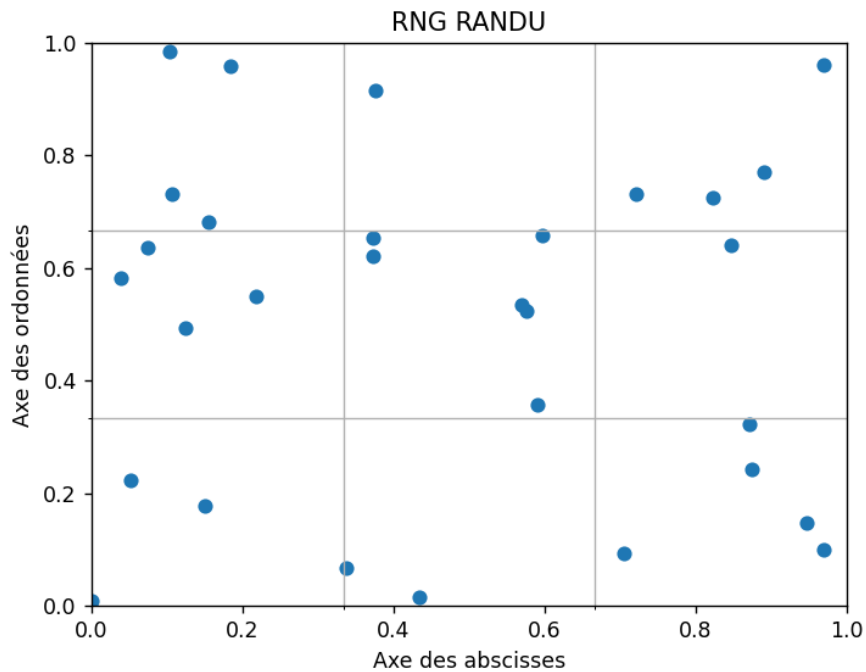
On obtient alors un carré unité avec les 12 points demandés !

1.2.4 Comptage des points par sous-carré

Comptage manuel

On peut d'abord compter manuellement le nombre d'échantillon présent par sous-carré.

Si l'on prend l'essai ci-dessous :



On peut alors voir qu'il y a :

4 échantillons dans le sous-carré 1,1

4 échantillons dans le sous-carré 1,2

2 échantillons dans le sous-carré 1,3

1 échantillon dans le sous-carré 2,1

6 échantillons dans le sous-carré 2,2

2 échantillons dans le sous-carré 2,3

4 échantillons dans le sous-carré 3,1

1 échantillon dans le sous-carré 3,2

5 échantillons dans le sous-carré 3,3

Les échantillons sont donc répartis sur tous les sous-carrés, il y fatalement des sous-carrés avec plus d'échantillons que d'autres.

Il s'agit d'un des points faibles de la méthode de Randu. Nous allons voir ce qu'il se passe quand nous utiliserons des grands nombres.

Comptage automatique

Nous avons alors compté le nombre de points par sous-carré de façon manuelle et avons remarqué très vite que les nombres pseudo-aléatoires donnés par la méthode de Randu ne sont pas réellement pseudo-aléatoires, ces valeurs se rapprochent plus de nombres juste aléatoires.

Si l'on utilise cette méthode avec une graine aléatoire, 100 itérations et 16 sous-carrés (avec des nombres légèrement plus grands), on obtient ceci :

```
Veuillez saisir la graine: 21315435
Entrez le nombre d'itérations que vous souhaitez: 100
Entrez le nombre de sous carrés que vous souhaitez: 16
```

```
Sous-carré (0, 0): 7 points
Sous-carré (0, 1): 8 points
Sous-carré (0, 2): 7 points
Sous-carré (0, 3): 4 points
Sous-carré (1, 0): 7 points
Sous-carré (1, 1): 3 points
Sous-carré (1, 2): 4 points
Sous-carré (1, 3): 5 points
Sous-carré (2, 0): 5 points
Sous-carré (2, 1): 5 points
Sous-carré (2, 2): 8 points
Sous-carré (2, 3): 6 points
Sous-carré (3, 0): 6 points
Sous-carré (3, 1): 9 points
Sous-carré (3, 2): 8 points
Sous-carré (3, 3): 8 points
```

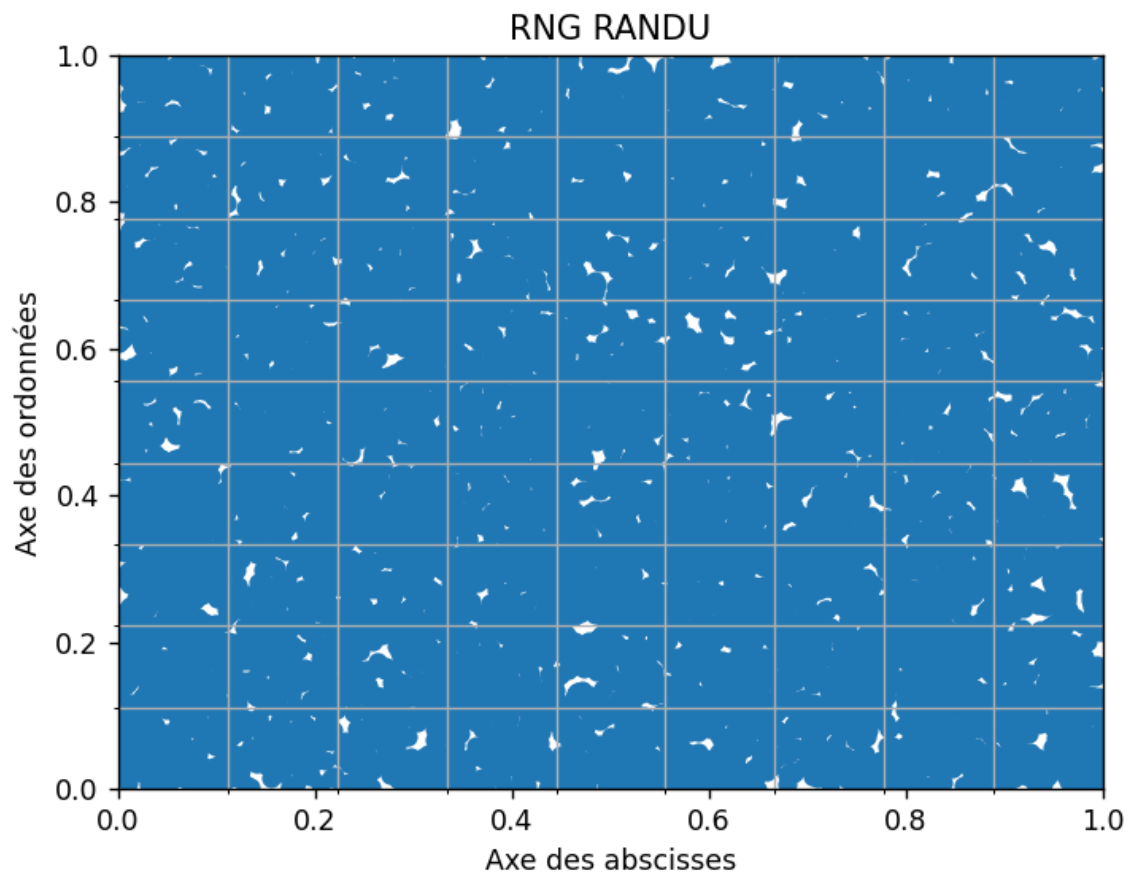
On obtient avec ces nombres pas très grands, une répartition aléatoire, en effet, on va de 3 points dans le sous-carré (1,1), jusqu'à 9 points dans le sous-carré (3,1). Cette différence peut paraître légère, mais cela veut dire que certains sous-carrés possèdent jusqu'à 3 fois plus de points que d'autres.

Finalement, nous allons essayer, cette méthode avec des nombres beaucoup plus grands :

```
Veuillez saisir la graine: 842753127
Entrez le nombre d'itérations que vous souhaitez: 10000
Entrez le nombre de sous carrés que vous souhaitez: 81
```

Prenons cet exemple où on demande 10000 itérations avec 81 sous-carrés.

On obtient alors ce graphique qui est bien plus rempli que les précédents :



Nous pouvons voir qu'il y a des zones et des sous-carrés avec énormément de points (bleus), où l'on ne voit même plus le fond blanc, mais aussi des zones plus ou moins grandes (dans plusieurs sous-carrés) avec aucun point.

Regardons maintenant en termes de chiffres, est-ce que l'aléatoire reste toujours très disparate ?

```
Sous-carré (0, 0): 120 points
Sous-carré (0, 1): 123 points
Sous-carré (0, 2): 133 points
Sous-carré (0, 3): 131 points
Sous-carré (0, 4): 118 points
Sous-carré (0, 5): 116 points
```

...

...

...

```
Sous-carré (8, 4): 97 points
Sous-carré (8, 5): 122 points
Sous-carré (8, 6): 122 points
Sous-carré (8, 7): 130 points
Sous-carré (8, 8): 124 points
```

Finalement, nous pouvons voir, en utilisant des nombres plus grands (mais qui restent quand même petit) que l'écart entre les différents sous-carrés diminue, pour autant, comme on l'a vu avec le graphique, il reste des écarts encore considérables.

En effet, le sous-carré avec le moins de points en compte 97 et celui avec le plus de points en compte 144. Le ratio passe donc de 3 fois plus de points (pour l'exemple précédent) à 1.5 fois plus de points.

Et plus les nombres deviennent grands, plus l'écart entre les sous-carrés diminue.

TD-TP 2

2.0.1 Méthode de Monte-Carlo

2.0.2 Evaluation de π

Nous avons vu dans le premier TD-TP comment générer des nombres pseudo-aléatoires.

L'objectif est alors, maintenant, d'utiliser ces valeurs aléatoires pour évaluer ou estimer d'autres valeurs. Il est possible, par exemple, d'estimer la valeur de π en utilisant des variables aléatoires et la méthode de Monte-Carlo. Voici une explication rapide de la méthode :

Imaginez un carré de côté 2 unités, centré à l'origine (0,0), de sorte que les coins du carré aient des coordonnées (-1, 1), (-1, -1), (1, -1) et (1, 1). Ce carré a une superficie de 4 unités carrées.

Maintenant, inscrivez un cercle de rayon 1 unité à l'intérieur de ce carré. Le cercle est centré à l'origine et est donc entièrement contenu dans le carré. La superficie du cercle est π unités carrées.

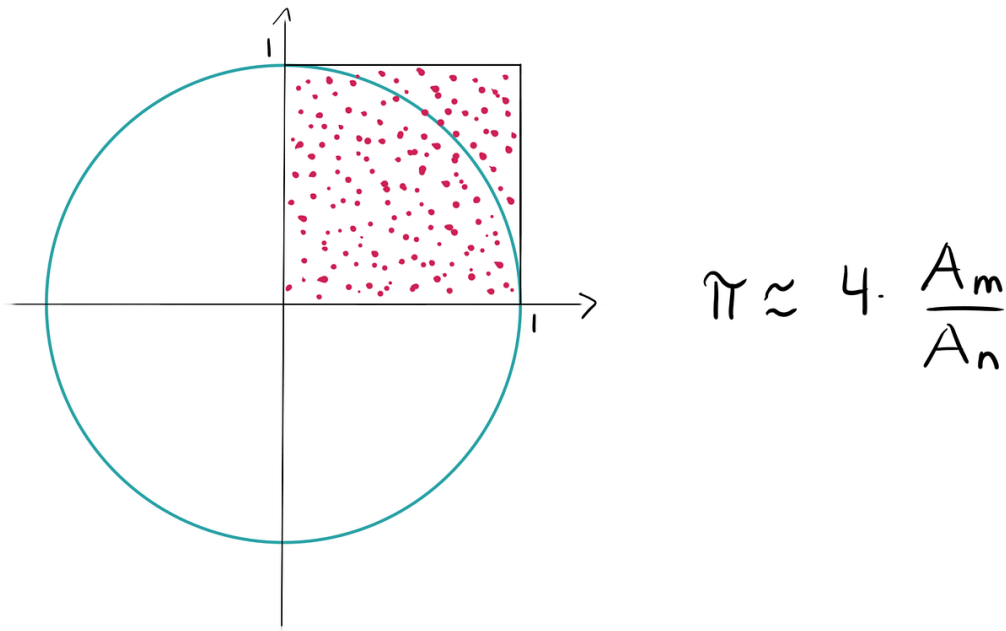
Générez un grand nombre de points aléatoires uniformément répartis dans le carré, en spécifiant des coordonnées x et y aléatoires entre -1 et 1.

Comptez combien de ces points se trouvent à l'intérieur du cercle (c'est-à-dire, à une distance de l'origine inférieure ou égale à 1 unité).

Estimez la valeur de π en utilisant la formule : $\pi = 4 * (\text{nombre de points dans le cercle}) / (\text{nombre total de points générés})$.

On obtient alors une estimations de la valeur de π . Plus on génère de points aléatoires, plus notre estimation de π sera précise.

Cette méthode exploite la relation entre l'aire du carré et du cercle pour estimer la valeur de π en comptant les points à l'intérieur du cercle par rapport au nombre total de points générés dans le carré.



On obtient alors la figure ci-dessus. Les points rouges sont toutes les valeurs aléatoires.

2.0.3 Tests avec différents RNG

Nous allons alors calculer π avec les différents générateur de RNG que l'on a étudié, dans l'exercice précédent.

RNG Python

Nous avons alors, dans un premier temps, utiliser le générateur d'aléatoire de la librairie random de Python.

En somme, c'est grâce à la fonction `random.random()`, que nous obtenons des termes aléatoires compris entre 0 et 1 pour x et y . Puis, nous évaluons π comme expliqué précédemment.

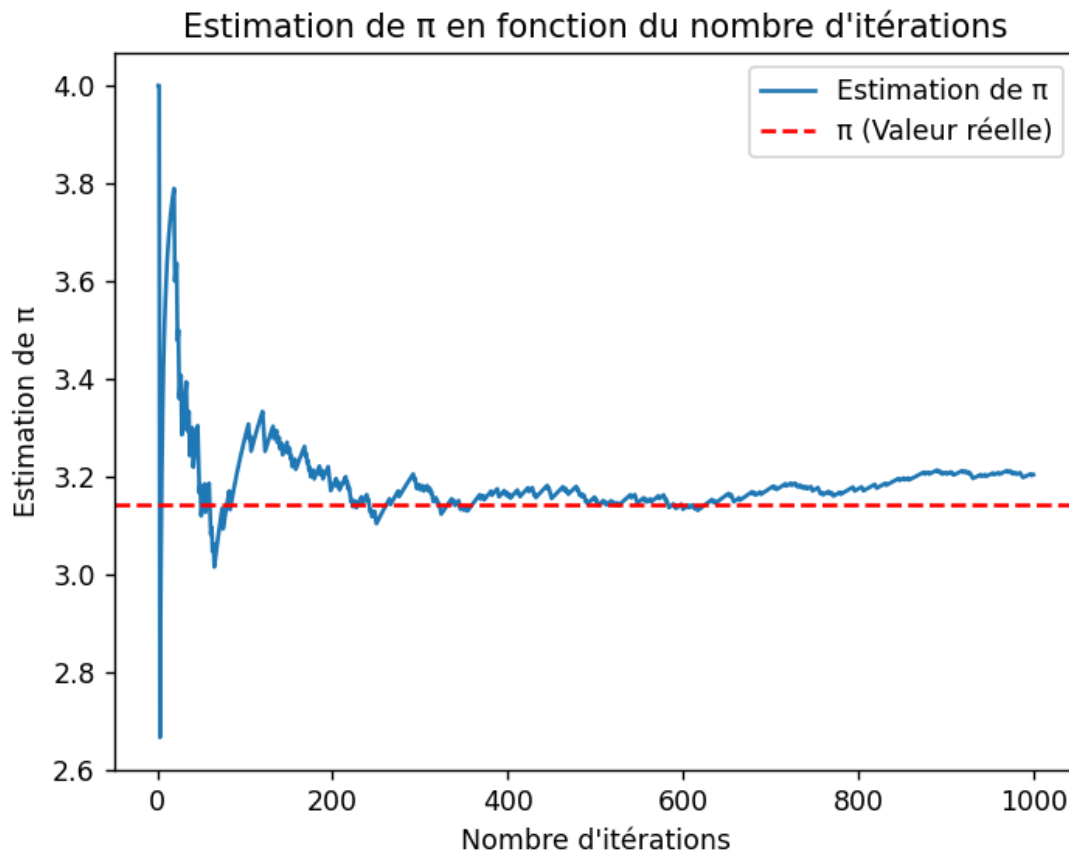
Pour résumer la méthode d'évaluation, il suffit, simplement, de calculer la somme du carré des coordonnées x et y ($x^2 + y^2$). Si cette valeur est inférieure ou égale à 1, alors le point se trouve à l'intérieur de la fonction représentative de π .

Finalement, c'est grâce au ratio de points à l'intérieur sur les points à l'extérieur que l'on obtient la valeur de π .

Avec 1000 itérations, la bibliothèque random de Python nous donne alors le résultat suivant :

```
● Entrez le nombre de tirage : 1000
  La valeur approximative de pi est 3.116
```

Par définition, plus on fait d'itérations, plus on obtient un résultat proche de π , c'est pourquoi nous avons réalisé un graphique qui montre l'évolution du résultat obtenu en fonction du nombre d'itérations :



Nous pouvons alors voir qu'avec 1000 itérations, la valeur théorique de π tend de plus en plus vers sa valeur réelle. Pour autant, il s'agit d'aléatoire, de ce fait, on peut voir que, par exemple, vers 600 itérations, la valeur théorique était légèrement plus proche de la valeur réelle.

RNG Neumann

Le deuxième RNG que nous avons utilisé pour calculer la valeur rapprochée de π est Neumann.

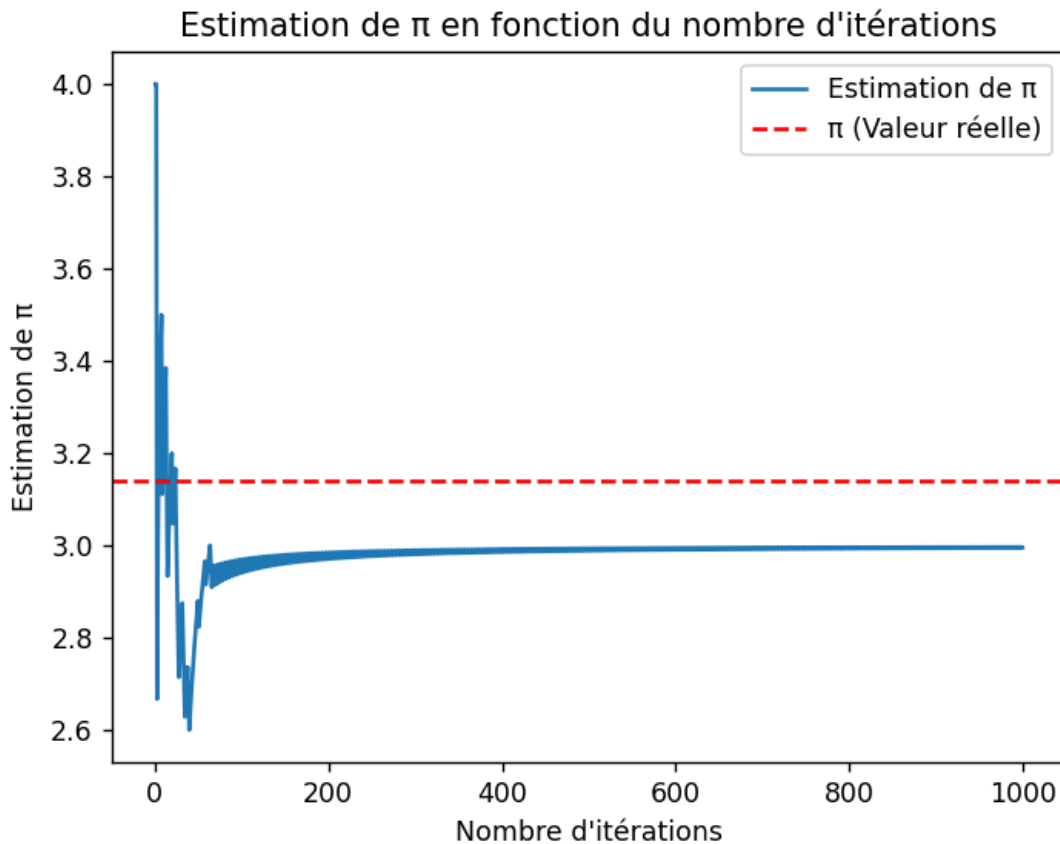
Alors, comme expliqué dans le TD-TP1, nous avons calculé des valeurs aléatoires grâce à une graine, puis comme pour la librairie random de Python, nous avons utilisé ces valeurs dans le théorème de Monte-Carlo.

Cependant, il faut bien diviser ces valeurs pseudo-aléatoires par 10000 pour obtenir des valeurs comprises entre 0 et 1 (car elles sont initialement comprises entre 0 et 10000).

On obtient alors l'approximation suivante avec 1000 itérations :

```
• Veuillez saisir la graine: 42156548
  Entrez le nombre d'itérations que vous souhaitez: 1000
  La valeur approximative de pi est 3.225806451612903
```

Si l'on réalise le même graphique qu'avec la bibliothèque random de Python, on obtient ceci :



Il s'agit d'un exemple où l'on a réalisé 1000 itérations avec une graine aléatoire. La première chose que l'on remarque est que la fonction représentative est très peu précise. En effet, comme nous l'avons expliqué au TD-TP1, la méthode de RNG de Neumann ne fonctionne pas très bien et souvent il y a des problèmes comme des boucles ou des valeurs qui restent bloquées à 0.

Dans ce cas, nous pouvons imaginer que les valeurs aléatoires données par Neumann ont commencé à boucler à partir d'une centaine d'itérations, ce qui donne cette courbe étrange qui ne tend pas réellement vers la valeur de π .

RNG Randu

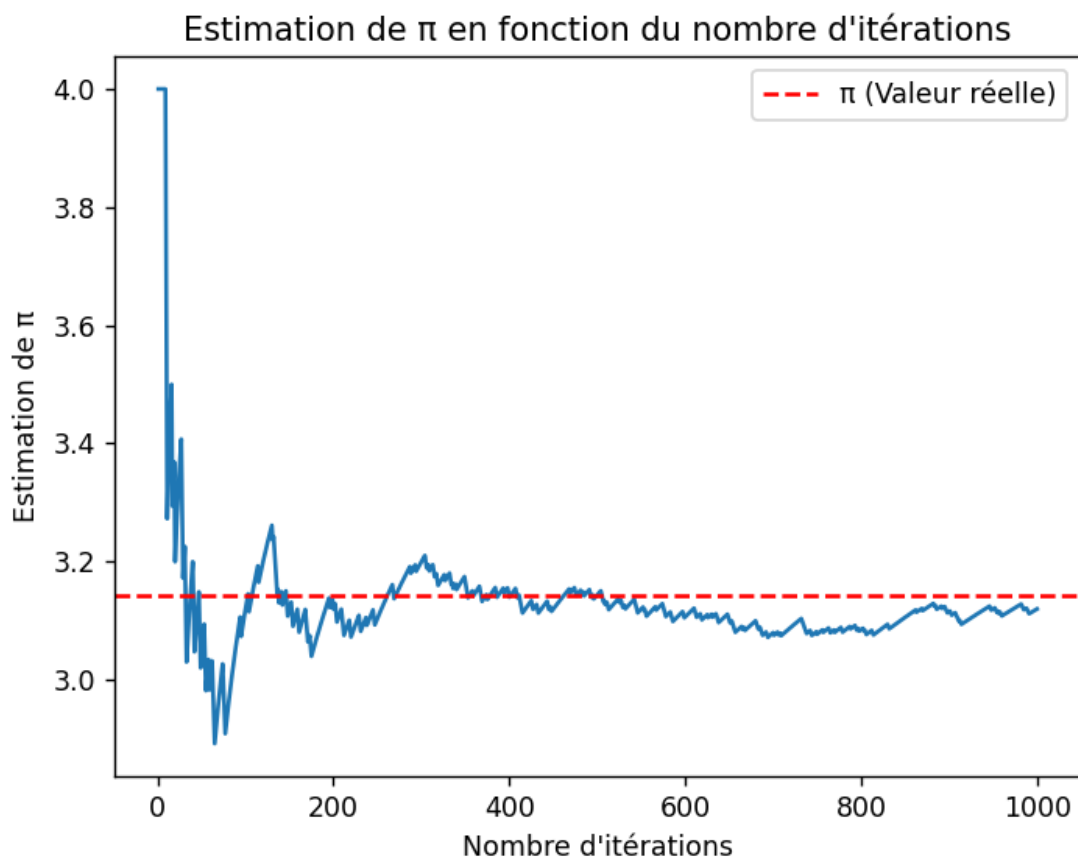
Finalement, nous avons utilisé la méthode de RNG de Randu pour calculer une valeur approximative de π

A l'instar des deux autres méthodes, nous avons utilisé l'RNG de Randu pour obtenir des valeurs pseudo-aléatoires. (Ces valeurs sont divisées par (2^31) pour obtenir des nombres compris entre 0 et 1).

On obtient alors l'approximation suivante avec 1000 itérations et une graine aléatoire :

```
Veuillez saisir la graine: 956281258
Entrez le nombre d'itérations que vous souhaitez: 1000
La valeur approximative de pi est 3.2
```

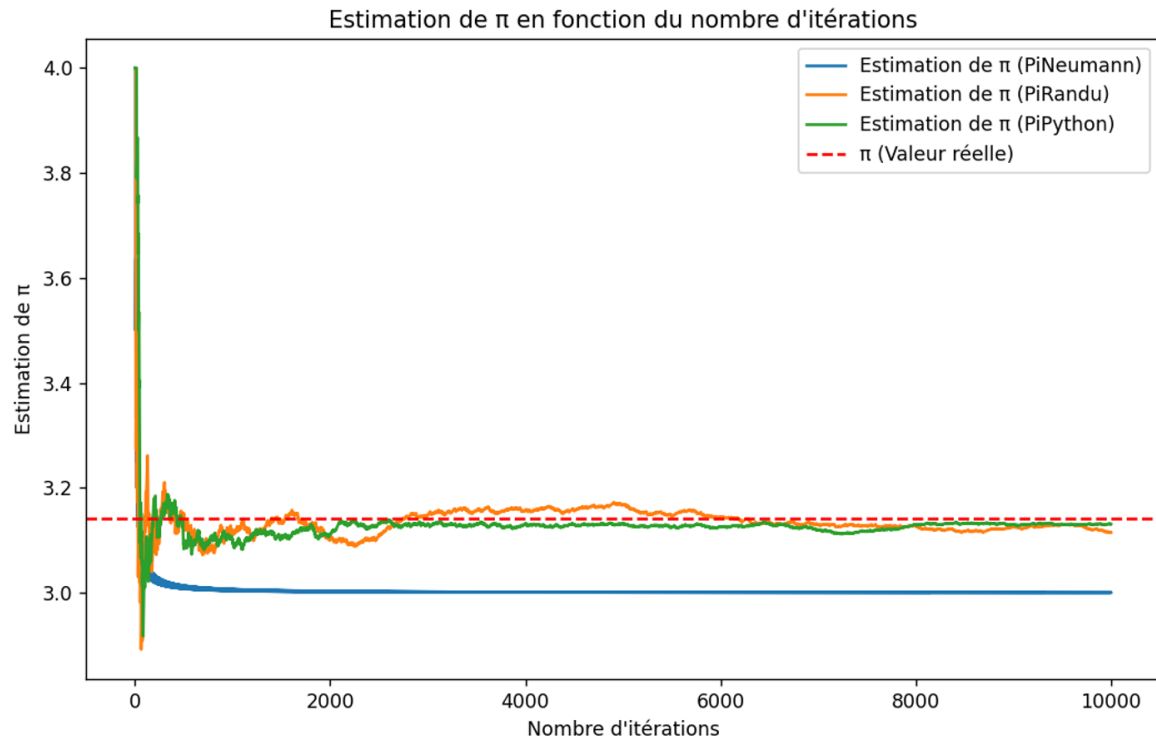
Si l'on réalise le graphique qui compare l'approximation en fonction du nombre d'itérations, on obtient ceci :



On peut voir que ce graphique ressemble de près à celui obtenu pour le RNG de la bibliothèque random. Alors, plus il y a d'itérations, plus la valeur approximative de π tend vers la valeur réelle.

Comparasion des méthodes

Nous avons alors réalisé un dernier graphique qui combine les 3 graphiques précédents, pour pouvoir comparer simplement les différentes méthodes de RNG (en utilisant la même graine pour Neumann et Randu).



Nous pouvons alors voir, comme nous l'avons remarqué précédemment que la méthode initiale de RNG de Python (bibliothèque random) et le RNG Randu sont très semblables, en effet ces deux méthodes donnent des résultats très proches de la valeur réelle de π et plus on fait d'itérations, plus cette valeur tend vers π . On peut voir qu'à partir de 300 ou 400 itérations la valeur donnée se situe entre 3,1 et 3,2 et à partir de 9000 à 10000 itérations elle vaut entre 3,13 et 3,15.

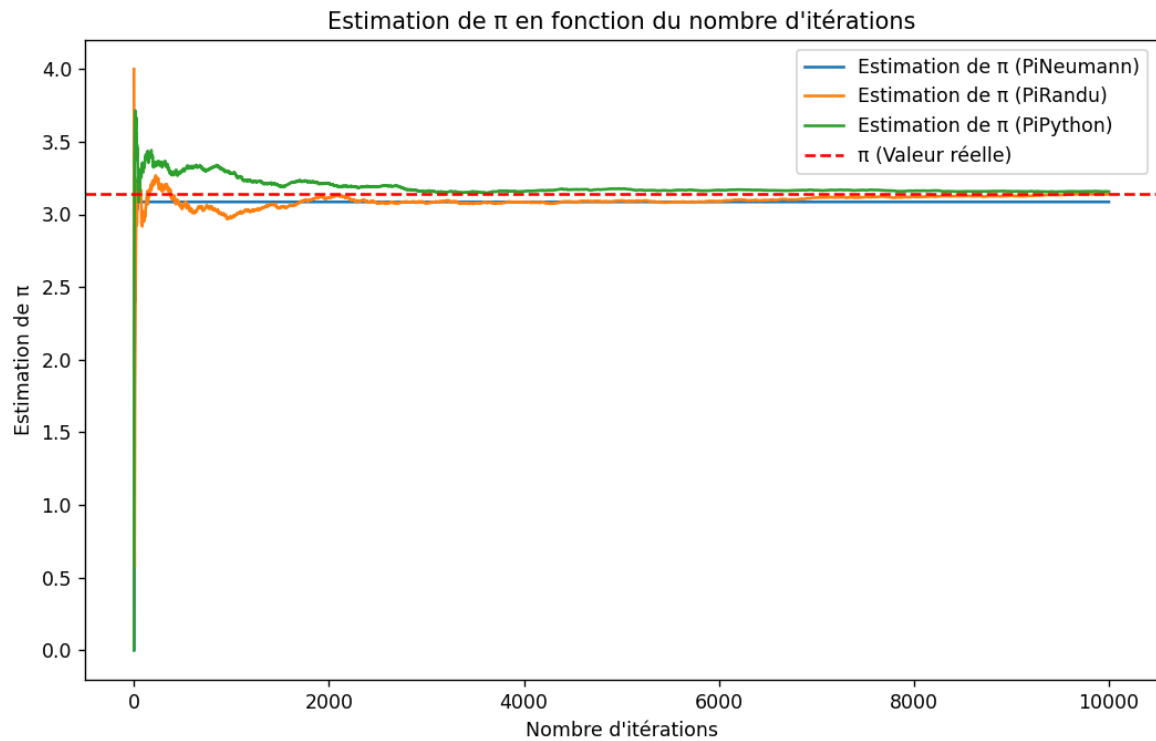
Pour autant, encore une fois, la méthode de Neumann donne des valeurs plus éloignées de la réelle valeur de π . En effet, on obtient très rapidement une valeur approximative de 3.0, puis cette valeur ne change presque plus, la méthode de Neumann a encore une fois rencontré un problème de cycle.

Si l'on choisit des graines différentes, les résultats restent similaires dans la plupart des cas, seul l'estimation de Neumann peut, si l'on choisit une graine sans aucun bug, être plus proche et tendre vers la valeur de π .

Voici un exemple où les 3 méthodes donnent des résultats similaires :

```

Veillez saisir la graine : 84265312953
Entrez le nombre d'itérations que vous souhaitez : 10000
  
```



En somme, on peut dire qu'il est possible d'obtenir une valeur approximative de π grâce à des nombres aléatoires et au théorème de Monte-Carlo. Pour autant, il faut faire attention à la véracité de l'aléatoire, en effet on a pu voir avec le RNG de Neumann ayant de nombreux problèmes, que la valeur de π en sortie était souvent éloignée de la réalité.

2.0.4 Volume d'une sphère

De la même manière, nous pouvons estimer par une méthode de Monte-Carlo, le volume d'une sphère.

Nous allons alors utiliser la librairie random de Python car nous avons vu, précédemment, que celui-ci fonctionne assez bien. Puis, nous allons générer des points avec 3 coordonnées x , y et z .

Ensuite, à l'instar de l'évaluation de π , nous allons faire un ratio des points à l'intérieur de la sphère sur les points à l'extérieur pour obtenir le volume d'une portion de la sphère.

Dans notre cas, étant donné que nous avons simulé une portion (1/8ème) de la sphère, il suffit de multiplier par 8 la valeur du ratio que l'on obtient, on obtient alors le volume d'une sphère de 1 de rayon.

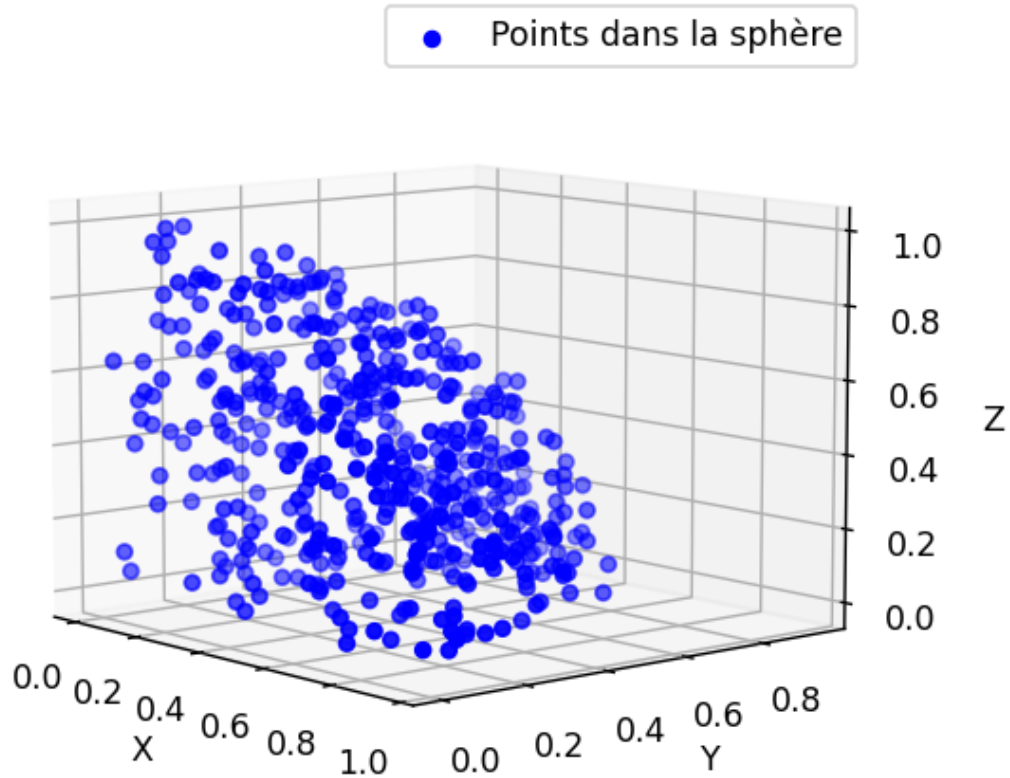
Pour obtenir le volume d'une sphère de rayon n , il suffit de multiplier le résultat par n^3 (en effet, il s'agit d'une figure en 3D).

On a alors calculé le volume d'une sphère de rayon 4 avec 1000 itérations :

```
Entrez le rayon de la sphère : 4
Le résultat obtenu (de façon approximative) est 257.024 et celui attendu est 268.082573106329
```

On obtient alors une valeur du volume de $(257 *^3)$ qui s'approche de la valeur réelle $(268 *^3)$.

Représentation 3D de la sphère



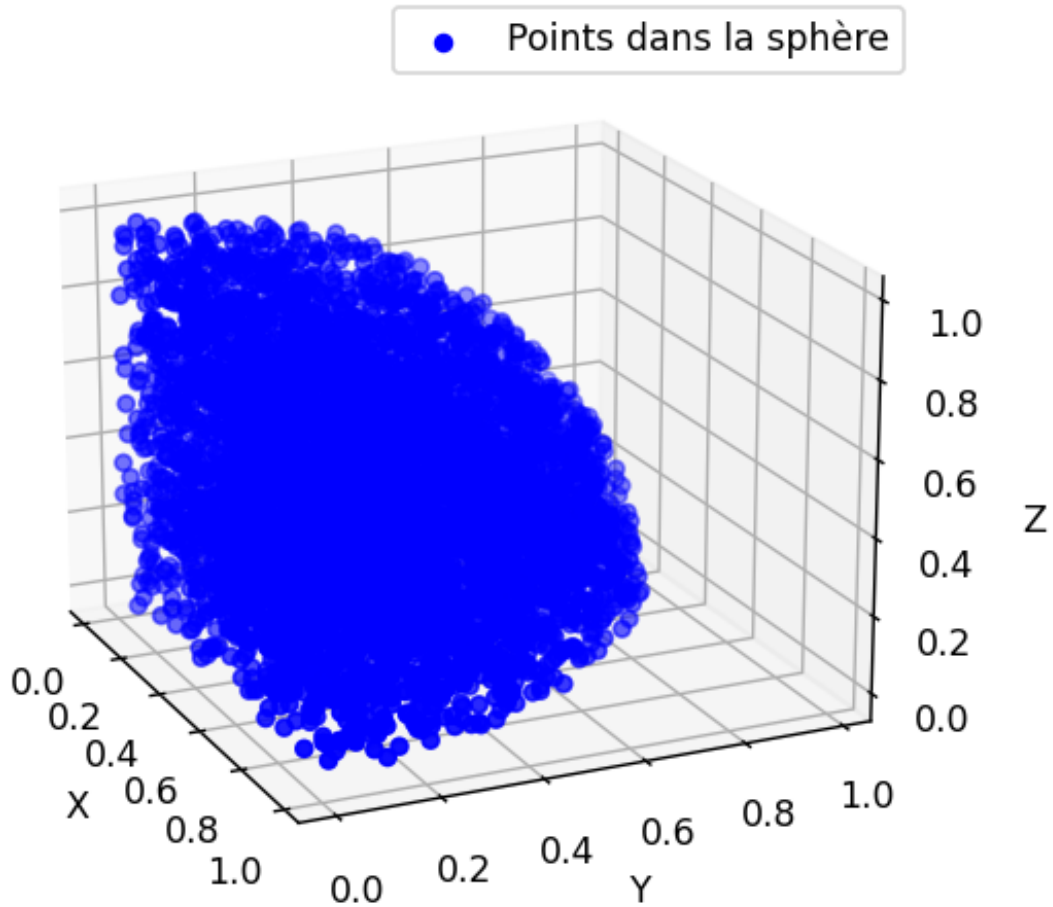
On a ci-dessus la représentation d'une portion (1/8ème) de la sphère. Dans notre cas, on a donc multiplié le volume de cette portion par 8 pour obtenir le volume total.

Si l'on refait la même sphère avec non pas 1000 mais 10000 itérations, on obtient :

```
Entrez le rayon de la sphère : 4
Le résultat obtenu (de façon approximative) est 270.8992 et celui attendu est 268.082573106329
```

On remarque alors que la valeur du volume $(270 *^3)$ est beaucoup plus proche du volume réel $(268 *^3)$, en effet, plus on fait d'itérations, plus cette valeur approximative tend vers la valeur réelle.

Représentation 3D de la sphère



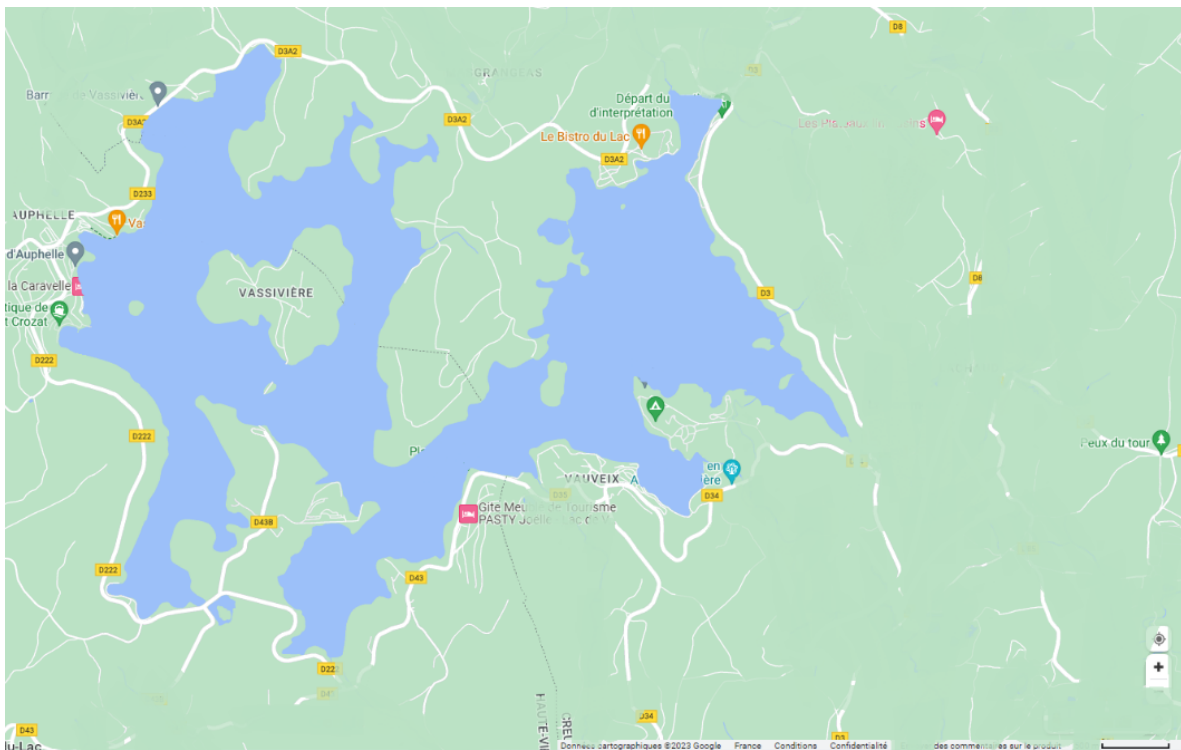
On obtient alors cette sphère qui est logiquement, plus rempli et mieux défini.

2.0.5 Superficie du lac de Vassivière

Pour mesurer la superficie du lac de Vassivière, nous avons décidé d'utiliser une image Google Maps où l'on voit tout le lac et en fonction des couleurs des pixels on détermine s'il s'agit d'un pixel du lac ou non.

Nous avons alors, dans un premier temps, récupérer une image Google Maps que nous avons aménagé pour pas qu'il y ait d'écriture (blanche ou d'une autre couleur) sur le lac. En effet, il faut que tous les pixels qui constituent le lac soit de la même couleur de bleu.

Voici alors notre image :



On a bien enlevé du lac chaque impureté (que ce soit des noms de places ou des points).

On remarque aussi un détail très important, il y a une échelle sur la carte, c'est cette échelle qui va nous permettre de passer de mesures en pixels à des mesures en mètres ou kilomètres.

En effet, on a une image qui fait une taille de 1335*843 pixels et une échelle de 73 pixels qui représente 500 mètres.

On peut alors calculer la surface de l'image en km^2 :

En effet, pour calculer, par exemple, la hauteur de l'image en mètre il suffit de diviser la hauteur de l'image en pixels (843) par la taille de l'échelle en pixels (73) puis de multiplier le quotient par la distance en mètre que représente cette même échelle (500).

On obtient alors la distance en mètre qui va (dans la réalité) du point en haut de l'image jusqu'en bas. On peut faire de même pour la longueur (1335 pixels) puis multiplier ces deux valeurs pour obtenir la surface de l'image (toujours dans la réalité) en m^2 . On divise finalement cette surface en m^2 par 1000000 pour obtenir des km^2 .

Puis, on vérifie pour un certain nombre d'itérations si le pixel choisi possède ou non la couleur du lac. En effet, on regarde si le pixel possède le RGB de la couleur du lac : (156,192,249). Si c'est le cas, ce pixel est un pixel du lac, sinon il en fait pas partie.

Il suffit finalement, à l'instar de l'évaluation de π , de multiplier la surface totale de

l'image en km^2 (que l'on a calculé précédemment) par le quotient du nombre de pixel du lac (donc qui possèdent les bonnes couleurs) sur le nombre de pixels en tout que l'on a testé.

On obtient alors la surface en km^2 du lac seulement.

Si l'on calcule une approximation de la surface avec 10000 itérations, on obtient ceci :

```
Entrez le nombre d'itérations: 10000
La surface du lac de Vassivière est de  9.50332613998874  Km²
```

D'après le programme, la surface du lac est d'environ 9.5 km^2 . Si l'on compare à la valeur réelle, on peut voir sur internet que la surface réelle est de 9.76 km^2 .

Lac de Vassivière / Area
9.76 km^2

Pour 10000 itérations, les résultats sont donc très similaires.

Si nous faisons 100000 itérations, nous obtenons alors des résultats beaucoup plus précis.

```
Entrez le nombre d'itérations: 100000
La surface du lac de Vassivière est de  9.710815427378495  Km²
```

Nous pouvons voir ci-dessus que le résultat estimé est de 9.71 km^2 pour la surface du lac (ce qui s'approche vraiment des 9.76 km^2 de surface).

De plus, quand on dessine graphiquement toutes les points qui sont dans le lac, on obtient bien une forme qui s'approche du plan initial du lac de Vassivière.

