Rapport du sujet de BPI

N° 161

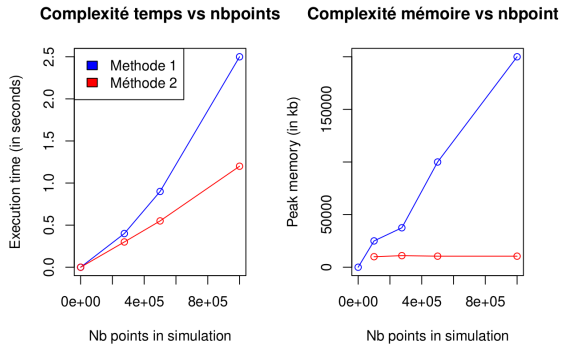
Ce document a pour but de décrire le déroulement du projet de BPI, portant sur l'illustration de la simulation de Monte-Carlo pour le calcul de π.
J'expliquerai globalement les principaux choix de conception que j'ai réalisés tout en montrant les difficultés techniques que j'ai rencontrées.

Choix des objets et fonctions à utiliser

Pour les deux modules, le critère de choix des types d'objets à utiliser s'est porté essentiellement sur leur coût en temps et en mémoire.

Module `approximate_pi`

La contrainte la plus importante pour ce module, est de pouvoir fournir les points utilisés pour le calcul de pi au module `draw`. Pour ce faire, j'ai testé deux méthodes.
La première consiste à créer un dictionnaire de points tirés, avec pour clé les coordonnées du point et pour valeur, le booléen indiquant si le point est dans le cercle unité. Pour avoir les valeurs courantes de π, le module `approximate_pi` doit alors fournir également une liste des 10 valeurs transitoires.
La deuxième méthode consiste à créer un générateur de points, à l'aide du mot clé `yield`. Les points ne sont alors plus stockés, ce qui réduit grandement l'espace mémoire utilisé.



Compte tenu de ces résultats, j'ai choisi d'utiliser un générateur de points. Mais, avec cette méthode, il devient impossible d'utiliser les mêmes points deux fois, à moins de les sauvegarder. Or, on a besoin des points une première fois pour le calcul de pi et une deuxième fois pour déterminer les couleurs des pixels. Pour remédier à ce problème, j'ai modifié mon générateur pour qu'à chaque fois qu'un dizième des points ont été tirés, il indique la valeur courante de π.
De plus, avec l'utilisation d'un générateur, pour une largeur d'image donnée, la complexité en mémoire en fonction du nombre de points du module `draw` est pratiquement constante.

Module `draw`

Pour ce module, la difficulté majeure consiste en la manière de retenir la couleur des pixels correspondant aux points déjà tirés. Là encore, j'ai eu à choisir entre deux méthodes: soit utiliser un dictionnaire de pixels,

ou alors utiliser une table (liste de listes). Mais les objects de type `dictionnaire` occupent environ 4 fois plus d'espace mémoire que les objets de type `list` comme l'illustre le code suivant:

```
from sys import getsizeof
s = {i for i in range(1000)}
l = [i for i in range(1000)]
print(f"Taille de s: {getsizeof(s)} bytes")
print(f"Taille de l: {getsizeof(l)} bytes")
```

```
## Taille de s: 32984 bytes
## Taille de l: 9016 bytes
```

Mon choix s'est donc porté sur une table de pixels, moins coûteux en mémoire. Bien que la complexité en calcul en fonction de la largeur de l'image soit alors en $\mathcal{O}(n^2)$, la variation du temps de calcul en fonction de la largeur de l'image reste néanmoins acceptable: pour 1000 points, si la taille de l'image est de 1000, le programmme met 3.6 secondes pour s'exécuter.

Ecriture de la valeur de pi dans les images

L'écriture de la valeur de pi dans les images ppm a constitué, pour moi, la partie la plus difficile à traiter. J'ai d'abord utilisé des `list` python pour retenir les pixels qui doivent être mis en noir pour écrire la valeur courante de pi pour chaque image. Mais étant donné qu'un test d'appartenance à une liste requiert un nombre de calculs de l'ordre de la taille de la liste, cette solution coûte énormément en temps d'exécution.
C'est pourquoi, j'ai choisi d'utiliser pour cette partie, des objets python de type `set`, qui ont l'avantage d'avoir un test d'appartenance avec une complexité en $\mathcal{O}(1)$.

Choix du format PPM à utiliser: `P3` ou `P6`

J'ai d'abord opté au départ pour le format `P3`, avec une intensité maximale de 255 pour avoir une plus grande palette de couleurs possibles. Mais, avec ce format, chaque triplet RGB nécessite 3 à 9 octets, et si on inclut l'en-tête ainsi que les espacements nécessaires entre chaque intensité, le poids de l'image est alors de 7.6 Mo. Si on ramène, l'intensité maximale à 1, l'espace nécessaire pour chaque intensité est alors réduit, et les images n'ont plus qu'un poids de 3.8 Mo.
Une manière d'améliorer encore plus le poids des images génerées est d'utiliser le format `P6`. En effet, avec ce format, les intensités sont codés en binaire, chaque triplet RGB nécessite alors exactement 3 octets. J'ai utilisé la fonction `bytes` qui convertit un objet de type `string` en bytes. Les images obtenus ont alors un poids de 1.9 Mo.

Test de performance

Pour finir, je présente ici un tableau récapitulant les complexités obtenues avec la version finale de mon projet. Les choix que j'ai effectués me permettent d'obtenir des programmes avec un bon compromis entre la complexité en temps et celle en mémoire.

Approximate_pi	
Complexité en temps en fonction du nombre de points	Linéaire
Complexité en mémoire en fonction du nombre de points	Constante

Draw	
Complexité en temps en fonction du nombre de points	Linéaire
Complexité en temps en fonction de la largeur de l'image	Quadratique
Complexité en mémoire en fonction de la largeur de l'image	Logarithmique
Complexité en mémoire en fonction du nombre de points	Constante