Rapport projet BPI

1. Justesse du code

Le programme réalise bien ce qui est demandé et emploie la commande *convert* pour la génération du .gif, la commande *ppmlabel* pour écrire sur les images ppm (choisi car il est contenu dans le même paquet que *convert*, qui est *imagemagick*), et le module *subprocess* de Python pour lancer des processus.

2. Qualité du code

Structure du projet

Le dossier rendu contient 3 scripts Python :

- Simulator.py : contient deux fonctions liées à l'algorithme de Monte-Carlo, l'une utilisée directement par le script (retournant directement une approximation sur la sortie standard), l'autre étant appelée par approximate_pi.py ;
- Approximate_pi.py : contenant les fonctions nécessaire à la génération des images et du gif animé ;
- Debug.py : détaillé au paragraphe « Débogage ».

Débogage

A des fins de débogage et de profilage, un module supplémentaire *debug.py* a été développé et rendu avec le projet. Il contient une unique fonction *printd* permettant d'afficher du texte sur la sortie standard lors qu'une certaine constante (*DEBUG*, contenue dans *debug.py*) contient le booléen *True*.

Pour ne pas encombrer la sortie standard lors de l'évaluation, j'ai donc assigné *False* à cette constante *DEBUG* (mais rien ne vous empêche d'essayer le déboguage en assignant *True* à cette constante).

Mesure de qualité

Les trois scripts rendus obtiennent une note de 10/10 de Pylint avec la configuration .pylintrc donnée.

3. Performance du programme

Une grande attention a été attribuée à la performance du programme, et notamment au temps d'exécution de ce dernier, ce qui a demandé la mise en place de solutions relativement complexes pour certaines fonctionnalités.

Afin d'en comprendre la démarche, voici une évolution des solutions employées pour différentes fonctionnalités.

Ecriture de l'approximation sur l'image

Comme cela a été justifié plus tôt, j'ai choisi *ppmlabel* pour écrire sur mes images.

1. Une première solution était donc d'écrire les images générées sur le disque puis d'ensuite écrire l'approximation. Le programme supprimait alors les images sans approximation avant la fin d'exécution du programme (comme elles n'étaient pas demandées).

2. Il s'avère que *ppmlabel* accepte également une image passée via un « pipe ». J'ai donc implémenté une communication directe avec l'entrée standard (après avoir appelé *ppmlabel* via *subprocess*) qui permet de « simuler » un pipe comme on le ferait dans un invité de commande classique [1]. Est à noter le fait que ce n'est pas visible sur la sortie standard de l'utilisateur. Cela a rendu l'exécution plus rapide.

Format de l'image .ppm

1. Le programme avait d'abord été conçu pour le format P3, produisant des images de taille de 4 Mo à peu près mais rendant les fichiers lisibles et donc plus facile à déboguer pour comprendre comment écrire correctement ces fichiers.

2. Comme il l'était subtilement suggéré, il a alors fallu ensuite coder les images en P6 afin de les rendre plus petite (1,8 Mo) et accélérant également leur écriture au prix de la lisibilité des fichiers.

Points de l'image en RAM

La solution qui s'est avérée être la plus rapide et la moins couteuse en stockage en mémoire était de stocker une liste de liste (formant donc une matrice 2D) donc les index de la première et de la seconde liste étant respectivement utilisés comme positions x et y sur l'image.

Chacune des « cases » de la liste de liste contenait alors non pas la couleur entière (ce qui alourdirait la taille de la variable en mémoire) mais un identifiant de couleur :

- 2 pour un point non généré (toutes les cases sont initialisées à 2) correspondant à du blanc ;
- 1 pour un point situé dans le cercle correspondant à du rouge ;
- 0 pour un point hors cercle correspondant à du bleu.

Au moment de l'écriture de tous les points de l'image, on écrit donc la couleur[2] correspondant à l'identifiant stocké dans chaque case de la matrice.

Performance mesurées

Tests effectués avec la commande *./approximate_pi.py 800 1000000 6.*
Taille maximale occupée en RAM (mesuré avec /usr/bin/*time – v ./approximate_pi.py 800 1000000*) : **203 Mo**.

Temps d'exécution moyen[3] (*./approximate_pi.py 800 1000000 6)* :

	Ecriture des images avec approximation sur disque	Conversion en .gif	Total
Machine perso. (WSL)	3.07 s	1.9 s	5.05 s
Machine Ensimag (ensipc504)	4.40 s	0.7 s	5.09 s

[1] Le code correspondant à cette partie est visible dans le fichier *approximate_pi.py* entre les lignes 46 et 58.
[2] Codée au préalable en binaire pour le format P6 dans la variable constante *COLORS_BIN*, contenu dans le fichier *approximate_pi.py*
[3] Mesures reproductibles en assignant *False* à la variable *DEBUG* dans le fichier *debug.py*.