RAPPORT PROJET

BPI SEMESTRE 1

1A

Information :

J'ai utilisé Python 3.8.6.
Les mesures de temps ont été faites avec le module time, les mesures de RAM avec HTOP.
Objectif :

Générer un GIF fait de 10 images montrant au centre l'évolution d'une simulation de π par la méthode de Monte-Carlo.

Méthode utilisée :

1) Rôles des modules *simulator* et *approximate_pi*

Simulator : Génère aléatoirement le nombre de points voulu qui seront affichés, les utilise pour la méthode de Monte-Carlo. Si utilisé en module principale, nécessite 1 argument : le nombre de points à tirer. Utilise numpy, random et sys.

Approximate_pi : Génère les 10 images PPM et le GIF fait des 10 images. Le GIF montre l'évolution de la simulation de Monte-Carlo : On voit apparaître à chaque image un nouveau dixième de points du nombre de points total à tirer en plus. Utilise sys, subprocess, et *simulator*.

2) Fonctionnement des 2 modules

Simulator :

Sa fonction principale est approximation_pi. Cette fonction prend comme argument ntpat (nombre total de point à tirer) et img_sze (la taille de l'image). Elle retourne : une liste de 10 approximations et une liste de 10 tableaux. L'approximation i prend en compte i-dixièmes du nombre total de points demandé par l'utilisateur. Donc la dernière approximation est celle prenant en compte la totalité des points et c'est celle qui est affichée si c'est le module principal. Le tableau i est un array numpy ; soit une matrice carrée de taille img_sze qui représente analytiquement la i-ème image, donc i-dixième du nombre total de points a été tiré. Un coefficient vaut :

-> 0 si le point n'est pas tiré
-> 1 si le point est tiré mais n'est pas dans le cercle
-> 2 si le point est tiré et est dans le cercle
Pour savoir ça on utilise la fonction in_circle.

Remarque : J'effectue une **homothétie**. En effet, je tire un flottant dans [-1,1], puis je lui rajoute 1 pour qu'il soit dans [0,2], puis je le multiplie par (taille image // 2) pour qu'il soit dans [0, img_sze], et enfin je prends la partie entière. Cela permet de limiter les calculs pour la fonction in_circle et d'accélérer le tirage aléatoire de chaque flottant.

J'ai utilisé np.equal pour gagner des points sur pylint. ^^

L'avantage des listes est que tout est stocké, le module *approximate_py* a juste besoin de lire les 2 listes. La somme de 2 tableaux est en O(img_sze) ce qui est correct. La chose qui prend le plus de temps est au final de tirer un grand nombre de points, ce qui est inévitable.

Approximate_pi :

La fonction principale de ce module est generate_gif_file. Elle prend ntpat (nombre total de points à tirer), img_sze (taille de l'image), et cav (chiffre après la virgule). Elle va d'abord récupérer la liste d'approximation et la liste de tableaux générées par la fonction approximation_pi du module *simulator*. Et va appeler 10 fois la fonction generate_ppm_gile qui génère l'image PPM avec sa propre valeur d'approximation de PI et son propre nombre de dixième de points. Puis va utiliser convert sur les 10 images pour générer le GIF.

Generate_ppm_file parcourt le tableau donné en argument et ajoute à la chaine de caractère les valeurs des pixels en conséquence. La simulation de pi tronqué avec un certain nombre de chiffre après la virgule et le nom de l'image sont générés par la fonction genere_nom_approx.

Remarque : au lieu de coder un affichage 7 segments, j'ai préféré utiliser **convert –annotate** qui écrit la chaine de caractère de la simulation de pi. L'avantage c'est que le coût/temps est limité car on peut indiquer directement où on veut marquer la chaine et que le programme ne parcourt pas le fichier en entier. J'ai obtenu 0.12 sec pour une image de 800x800 et 0.17s pour une image de 1000x1000. Avec -pointsize, je peux modifier la taille des écritures en fonction de la taille de mon image. On peut également modifier la police et la couleur. C'est également plus rapide à coder (nécessite une ligne de commande avec subprocess).

Remarque : J'écris **une seule et unique fois** dans le fichier ppm car c'est plus rapide par rapport à écrire pixel par pixel. J'ai également remarqué que cela diminuait également la taille du fichier PPM.

3) Efficacité temporelle, mémorielle et améliorations possibles

Chaque image a une taille de 1.9 Mo ce qui est cohérent avec les indications du sujet.

Je reste sous les 200Mo de RAM utilisée lors de l'exécution d'*approximate_pi*.

Sur mon PC j'ai eu un temps de 10.2sec pour une taille de 800 et 1000000 de points. Chaque image PPM est créée en entre 0.5s et 0.6 sec. Mais une image de taille 1600*1600 est créée en environ 2.5sec. (La création ne dépend pas du nombre de points tirés). Ce n'est pas exactement linéaire (4x plus de temps pour 4x plus de pixels) mais la différence reste tout de même limitée.

Cependant quand je tire 10 millions de points, je passe à environ 35sec d'exécution. Ce qui confirme que la chose qui prend le plus de temps dans l'exécution du programme est le tirage de point. En effet, que je stocke ces points dans la liste de tableau (1ère version de mon code) ou non, le tirage de points + et la simulation de pi me prends toujours plus de 30sec. A titre informatif, 1 million de points me prend environ 3.5sec, ce qui correspond à un dixième du temps pour 10 millions de points.

Une des solutions pour gagner en efficacité aurait été d'appeler le module *simulator* 10 fois ; cela aurait permis de diminuer en mémoire, et de gagner un peu de temps car il n'y aurait pas de somme de tableau à faire (un seul et unique tableau que l'on devrait rafraîchir à chaque appel), mais aurait augmenté la longueur du module *approximate_pi*. Cela aurait cependant permis de passer sous la barre symbolique des 10sec.

J'ai obtenu 10 au pylint de *simulator*, mais j'obtiens 9.80 à *approximate_pi*, le « problème » étant que j'utilise in range(len()) au lieu d'enumerate, ce qui complexifierait le code.