# RAPPORT PROJET BPI

## JUSTESSE DU CODE :

- L'exécutable draw.py prend bien en argument la taille de l'image, le nombre de point et le nombre de chiffres après la virgule et renvoie 10 images au format ppm ainsi que le gif correspondant.
- Si l'utilisateur rentre un paramètre incorrect, le programme lui renvoie une erreur ainsi que la pile d'appels, par exemple `$ ./draw.py -800 100000 4`
- Les noms des images sont conformes à la consigne, le nom du gif est choisi sans importance (legif.gif).
- Le module subprocess a été importé pour utiliser le programme convert :

```
$ subprocess.call("convert -delay 50 " + les_images + " legif.gif", shell=True)
```

## RESPECT DES CONSIGNES :

Les exceptions sont bien lancées pour les différents arguments imposés à draw.py.
L'utilisation de f-string, spécifiquement de type f'bloblo{blabla:.{toto}f}tutu' a bien été prise en compte, pour créer le nom de chaque image.
Les images ont un nom correspondant au format img{i}_[0-9]-[0-9][0-9][0-9].ppm, et sont bien conformes à l'attente (vérifié grace à l'outil ImageMagick).
Les temps d'exécutions sont bons, passant tous les tests de performances.
L'affichage de l'approximation de pi se fait en accord avec le nombre de chiffre après la virgule choisie par l'utilisateur et sans module externe. Un afficheur 7 segments a été implémenté, centré au centre de l'image et s'adaptant à la taille de celle-ci. Les couleurs sont modifiables mais limitées (le fichier étant en nuance de gris).

## QUALITÉ DU CODE :

Après analyse de pylint, les deux programmes obtiennent des scores supérieurs à 9/10.
L'utilisation de variable globale aurait pu être envisagé pour améliorer la lisibilité du code ainsi que de diminuer le nombre de paramètres de certaines fonctions.

## ETUDE DE LA PERFORMANCE :

### OPTIMISATION :

- Comme le programme draw appellera plusieurs fois les modules d'approximation_pi, il est nécessaire de les optimiser au maximum. Ainsi pour la génération des points, la fonction random.uniform était tout d'abord utilisée, générant un flottant dans [-1,1].
  Cependant, on peut remarquer que cette fonction utilise elle-même random.random. Il était donc plus intéressant de directement cette dernière et de faire quelques ajustements pour avoir la génération d'un point dans le bon intervalle.
- A chaque itération de boucle, la liste_image et le dictionnaire_point ne sont pas réinitialisés. On garde ainsi sur la nouvelle image les points déjà tirés, la fonction approximation_liste ne devant alors générer qu'un dixième des points à chaque étape.
- L'utilisation d'un dictionnaire a été utilisée pour la fonction écrire_chiffre. Ce dictionnaire a pour clé i et renvoi la fonction chiffre_i. L'utilisation de ce dictionnaire remplace les 10 tests initialement prévus et coûteux en complexité.

- L'utilisation d'append a été limité au maximum. On a préféré créer des tableaux de taille connue et remplacer les valeurs au fur et à mesure. Cette optimisation a permis un gain de 20% en temps d'exécution.

## COMPLEXITÉ EN MÉMOIRE :

L'utilisation de yield aurait pu être envisagé pour diminuer grandement la complexité mémoire. Cependant cette option nécessitait la réécriture quasi-totale de l'architecture du programme, et n'a donc pas été implémentée.

## POIDS DES IMAGES :

Le fichier ppm a finalement été écrit en P2 nuance de gris au lieu du choix RGB initial. En effet, la couleur choisie est alors représentée par un seul caractère au lieu de 11, soit une réduction du nombre de caractère total par fichier d'environ 90%.

## PERFORMANCE DES FONCTIONS :

**Approximate_pi.py :**

- La complexité temporelle en fonction du nombre de points tirés est linéaire étant donné que la seule boucle for a pour taille ce paramètre.
- La complexité en mémoire est elle constante, car il n'y a pas de stockage de données autre que la liste crée, l'itérateur et un seul point créé puis sauvegardé à la fois. Le coût en mémoire est simplement la mémoire vive allouée pour la bonne exécution du programme

**Draw.py :**

- La fonction draw a une complexité linéaire en nombre de points et en nombre de chiffre après la virgule. Pour le paramètre taille, étant donné qu'on parcourt les pixels de l'image pour l'écriture du fichier, soit un nombre total taille*taille de pixels, la complexité est quadratique. En effet, plusieurs autres boucles sont présentes dans le programme mais aucune n'est imbriquée dans l'autre.
- La complexité mémoire de la fonction est constante avec le nombre de points de l'image mais augmente significativement avec la taille.

## CONCLUSION :

Le programme délivré est conforme aux attentes et permet d'approximer efficacement le nombre pi, avec un apport visuel sous forme de gif. Bien que le programme passe les différents tests imposés, des optimisations aussi bien temporelle que pour la mémoire aurait pu être envisagées, en particulier pour la génération d'image de taille importante.