TP1 - Visulisation de fractales : Ensemble de Mandelbrot

Consignes

Cadre

Les Tps se font par binôme (ou un monôme en cas de nombre d'étudiants impairs) afin de palier au manque de machines en salle. Cela veut dire qu'il y aura seulement un rendu par groupe.

Rendu

Le rendu se compose des deux éléments suivants:

- Le code complété à partir du répertoire initial qui accompagne ce sujet.
- Un rapport écrit au format word ou pdf selon le modèle disponible dans le répertoire de travail. Ce rapport doit rendre compte du travail que vous avez effectué en présentant pour chaque question demandé les problèmes à résoudre et la solution proposée avec une explication avec vos propres mots et enfin le résultat obtenu (par exemple un affichage du résultat obtenu sur plusieurs exemples dans le cas d'une fonction).

Les deux éléments sont **essentiels**. En effet, avoir un code fonctionnel et répondant à l'ensemble des questions. ne suffit pas pour avoir une bonne évaluation. Le rapport permet de rendre compte que vous avez compris ce que vous faisiez.

> *Le dépôt du rendu (code + rapport) se fait sur la plateforme **Github Classroom** dont le lien est disponible sur la page moodle du cours au niveau relatif au TP2. **De plus, il faut aussi déposer uniquement le rapport** sur la zone dépot du TP2 sur la page moodle.*

Test unitaires

Des tests unitaires sont présents dans le dossier **tests** afin de permettre une évaluation rapide de votre progression dans le TP et également de vous permettre de savoir si la solution que vous proposer répond bien à ce qui est demandé en termes de spécifications. Lorsque vous réaliser une fonction, méthode ou classe démandés par une question, il faut exécuter la série de tests unitaires (en éxécutant le fichier de tests) comme présenté dans le dernier TD sur les tests unitaires.

> *Dans le cadre du TP1, deux fichiers concentrent les tests :*
>
> - ***test_nombrecomplexe.py** qui concentre des tests sur une classe **Image** à réaliser.*
> - ***test_reconnaissance.py** qui concentre des tests sur une fonction de reconnaissance de chiffres à écrire.*

Évaluation

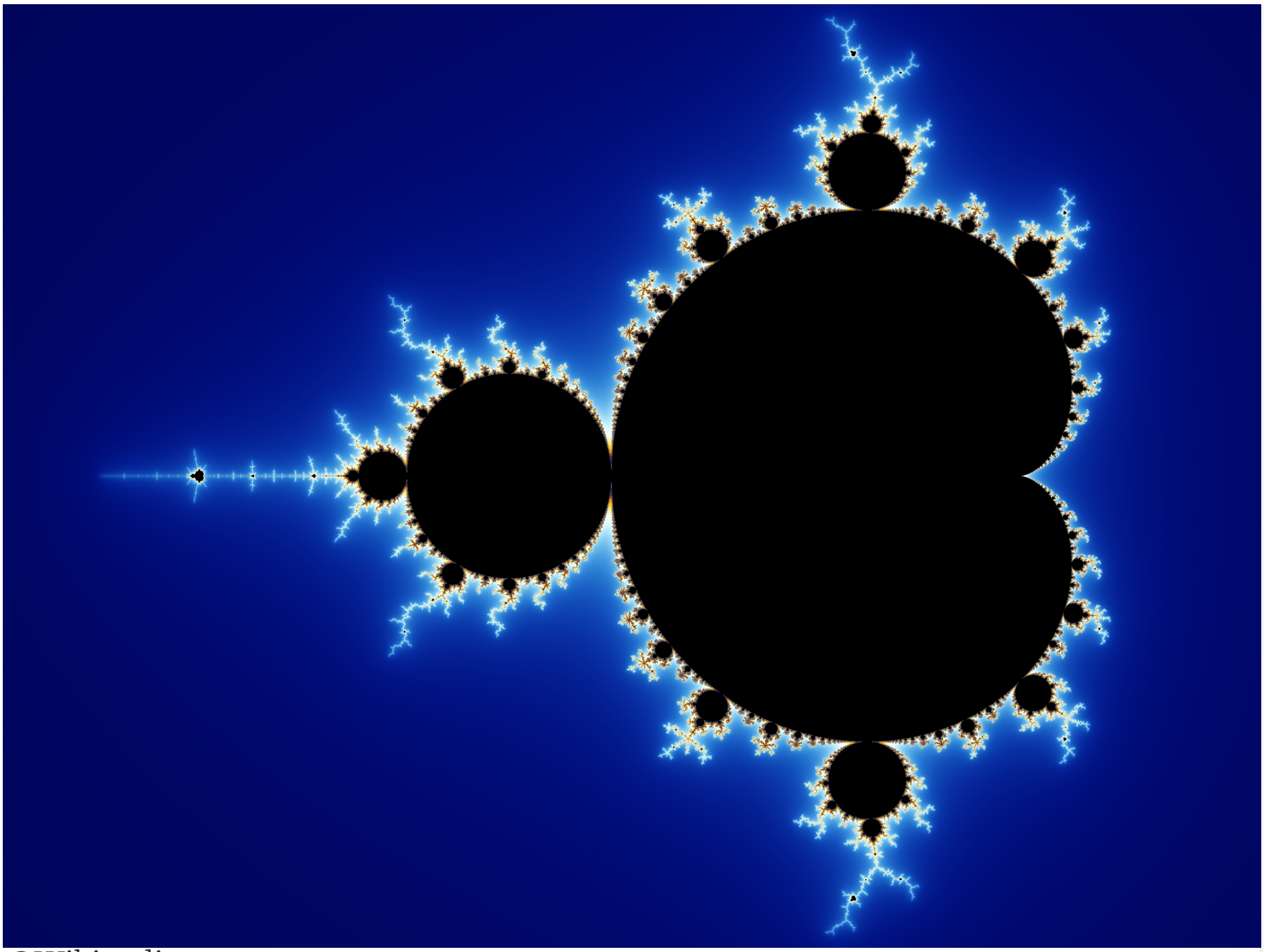
L'évaluation prend en compte d'une part le résultat des tests unitaires dans une moindre mesure mais surtout en grande partie la qualité des explications et du rapport. **Ce TP n'est pas noté** afin de vous permettre de vous familiariser avec l'approche par tests unitaires et également sur le procédure de rendu. **Il est toutefois nécéssaire de faire un rendu sans quoi votre note de TP sera pénalisée.**

Délai

Afin de vous laisser le temps de faire le rapport si besoin, la date limite du rendu est donnée au **soir du jour où le TP a été programmé (avant minuit)**.

I - Présentation du TP

Le but de ce TP est de d'obtenir une visualisation de l'ensemble de Mandelbrot qui est une fractale bien connue:



@Wikipedia

La fractale est définie comme l'ensemble des points c du plan complexe pour lesquels la suite de nombres complexes définie par récurrence par :

$$\begin{cases} z_0 & = & 0 \\ z_{n+1} & = & z_n^2 + c \end{cases}$$

est bornée.

Elle peut être tracée plus simplement à l'aide du résultat suivant:

> *Si la suite des modules des z_n est strictement supérieure à 2 pour un certain indice alors, cette suite est croissante à partir de cet indice, et elle tend vers l'infini.*

Consignes

> *Il est nécéssaire de les tester manuellement à chaque fois les fonctions que vous implémentez en faisant des exemples dans le main du programme.*

II - Préliminaires : les nombres complexes

Dans un premier temps, nous avons besoin d'avoir une représentation des nobmre complexes afin de pouvoir effectuer les calculs des itérations de la suite de Mandelbrot. Pour ce faire, nous allons définir une classe.

1. Définir une classe **NombreComplexe** qui permet de représenter un nombre complexe à l'aide de deux attributs **real** et **imag** correspondant à la partie réelle et imaginaire avec le prototype suivant:

```python
class NombreComplexe:
    """Classe représentant un nombre complexe."""
    def __init__(self, real, imag):
        # A remplir
```

2. Ajouter à la classe la méthode **module** qui renvoie le module du nombre complexe représenté par la classe. On aura besoin de la fonction **sqrt** de la librairie math qui s'importe comme suit en préambule du code:

```python
from math import sqrt
```

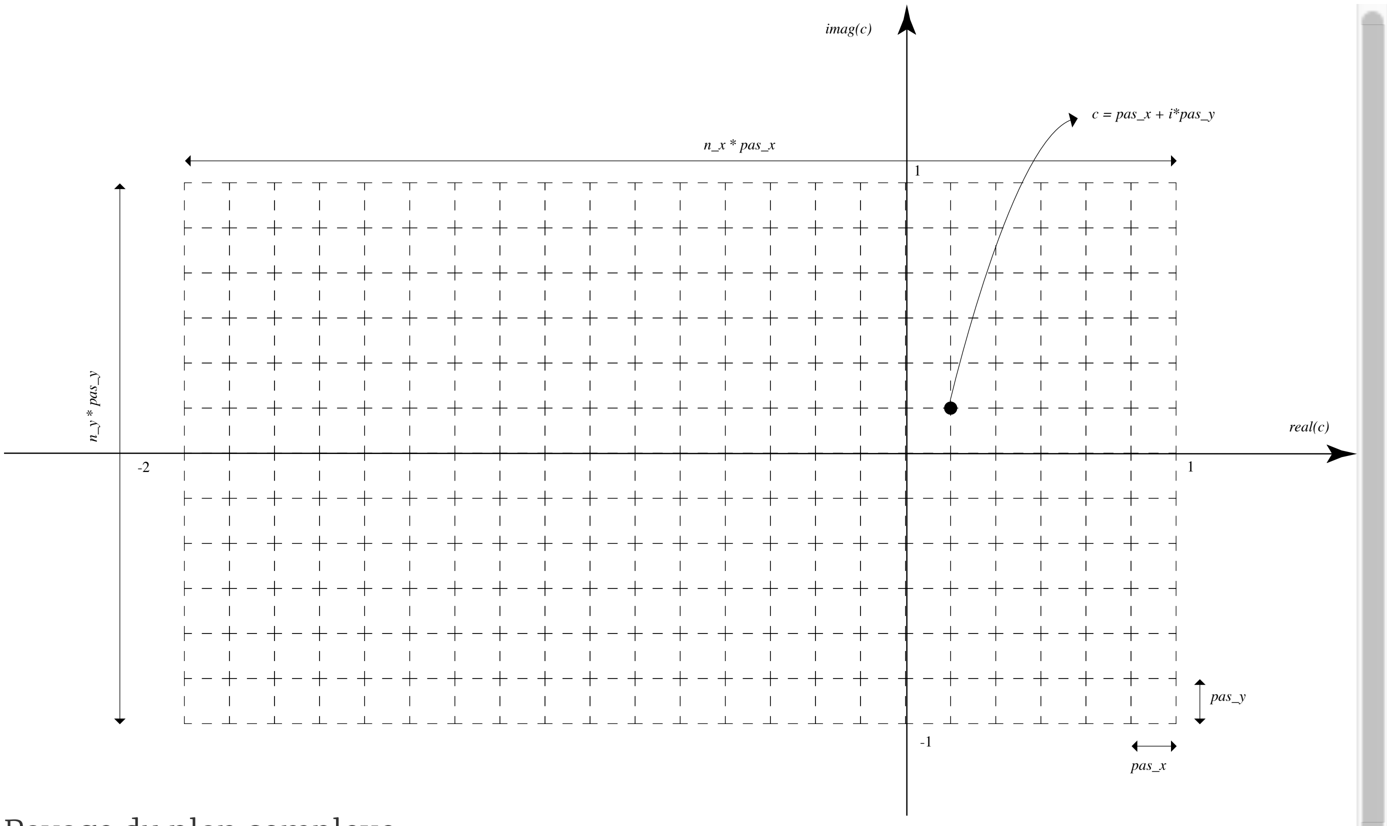
3. Surcharger la méthode **str** afin de pouvoir afficher le nombre complexe à l'aide de la fonction **print**. On doit gérer les cas où la partie imaginaire est positive, négative ou nulle. Exemple:
    - NombreComplexe(1,10) -> "1 + 10i"
    - NombreComplexe(5,-10) -> "5 - 10i"
    - NombreComplexe(-3, 0) -> "-3"
4. Surcharger les méthodes pertinentes pour pouvoir additioner et multiplier des nombres complexes à l'aide des symboles +, - et *.
5. Surcharger la méthode pertinente pour pouvoir utiliser le symbole ** afin de réaliser la puissance d'un nombre complexe.

III - Le plan complexe comme une image

L'ensemble de Mandelbrot concerne essentiellemnt les éléments du plan complexe respectant les conditions suivantes:

$$\begin{cases} \mathrm{Re}(z) \in [-2, 1] \\ \mathrm{Im}(z) \in [-1, 1] \end{cases}.$$

Pour pouvoir afficher le résultat de la fractale, nous devons pouvoir représenter les nombres complexes de cet ensemble mais en discrétisant l'espace. Nous n'allons pas pouvoir en effet traiter tous les points (une infinité !) et on se propose de construire une grille de la manière suivante:



Pavage du plan complexe

Afin d'avoir une représentation arbitrairement fine, on choisit deux paramètres n_y et n_x qui permettent de donner le nombres de nombres réprésentés. Cette grille sera représentée par une liste de liste (à la manière des matrices vues en TD).

Étant donné une grille de taille $n_y \times n_x$, et une résolution donnée par deux pas pas_x et pas_y, un nombre complexe représenté par le pixel à la ligne k, et colonne l est le suivant:

$$c_{kl} = l \times pas_x + i(k \times pas_y) - z_d,$$

avec $z_d \in \mathbb{C}$, $(pas_x, pas_y) \in \mathbb{R}^2$ étant nombres bien choisis afin que $c_{00} = -2 + i$ et $c_{(n_y-1)(n_x-1)} = 1 - i$.

1. Quels sont les valeurs de pas_x, pas_y et z_d ?
2. Implémenter la fonction **nombre_complexe** qui renvoie le nombre complexe à partir de l'indice du pixel. Le prototype est le suivant suivant:

```
def nombre_complexe(k, l, n_y, n_x):

    return ...
```

3. Faire une fonction **grille_complexe** qui prend en entrée les paramètres n_y, n_x, et qui renvoie un tableau bidimensionnel (liste de liste) correspondant à la grille de nombres complexes. Le prototype sera le suivant:

```
def grille_complexe(n_y, n_x):

    return ...
```

IV - Visualisation à l'aide de la librairie matplotlib

Pour pouvoir visualiser la fractale, nous nous aidons de la librairie **matplotlib** qui permet de tracer facilement des courbes et graphiques et afficher des images. Pour ce faire il faut ajouter la ligne d'importation suivante au préambule du code:

```
import matplotlib.pyplot as plt
```

1. Expliquer ce que fait cette ligne.
2. À partir d'une grille contruite par la fonction **grille_complexe**, construire un tableau dans une variable nommée **tableau_module** contenant le module de chaque nombre complexe.
3. Afficher le résultat comme une image (un point = un pixel) à l'aide des commandes suivantes:

```
plt.figure()
plt.imshow(tableau_module, aspect='auto')
plt.colorbar()
plt.show()
```

V - Algorithme de calcul de la fractale

On va maintenant s'intéresser au problème principal à savoir trouver pour chaque nombre complexe de la grille, trouver si la suite à l'équation (1) converge. Pour cela on va réaliser un algorithme itératif qui va calculer les termes de la suite jusqu'à ce que la valeur du module est supérieure à 2 (suite diverge) ou jusqu'à un certain nombre N défini par l'utilisateur (suite ne diverge pas).

1. Écrire une fonction **est_divergente** qui prend en paramètre un nombre complexe c, un nombre N et qui renvoie $True$ si la suite diverge ou $False$ sinon. Le prototype est le suivant:

```
def est_divergente(c, N):

    return ...
```

2. Écrire une fonction **image_mandelbrot** qui calcule pour une image de taille et résolution données, une image avec comme valeur de pixel 0 si la suite converge et 255 sinon. Le prototype de la fonction est le suivant:

```
def image_mandelbrot(n_y, n_x, N):

    return ...
```

3. Créer une image à l'aide de la fonction **image_mandelbrot** et la visualise à l'aide de **matplotlib**. Choissisez pour valeurs $n_x = 100$, $n_y = 200$ et $N = 20$.
4. Essayer avec différentes valeurs de n_x, n_y et N. Observer les différences. Sur quoi chaque paramètre a une influence ?

> *Il est possible d'avoir un affichage visuel de l'avancement d'une boucle for à l'aide de la libraire tqdm. Pour ce faire, il faut ajouter au préambule du fichier (à condition que la librairie soit instalée):*
>
> *from tqdm import trange*
>
> *Ensuite remplacer par exemple: **for x in range(5):** par **for x in trange(5):**.*
>
> *A titre d'exemple, on aura l'affichage suivant:*
>
> 
>

5. (Bonus, à faire que si en avance) Ajoutons de la couleur ! Pour cela au lieu de mettre 0 au pixel lorsque la suite diverge, mettre la valeur de l'itération à partir de laquelle le module dépasse 2. Le prototype de la fonction est le suivant:

```python
def image_mandelbrot_couleur(n_y, n_x, N):

    return ...
```

On pourra s'aider d'une fonction **nombre_iterations(c, N)** qui renvoie le nombre d'itérations effectuées la suite au point c.

VI - Utilisation de librairies standard

Il se trouve que python gère nativement les données complexe et il n'y a pas besoin de refaire une classe pour cela (à par pour des fins pédagogiques bien sur !). Pour instancier un nombre complexe, il suffit de d'écrire par exemple:

```python
z = 5 - 3j
```

Les opérations usuelles (+, *, - et **) sont déjà implémentées. D'où l'utilité de faire une recherche dans la documentation pour savoir si ce que l'on cherche à faire n'existe pas déjà !

On se propose ici d'utiliser dans cet esprit une librairie qui s'appelle **numpy** (https://numpy.org). Celle-ci permet de gérer efficacement des objets de type tableaux (tels que les grilles où les images que l'on a pu rencontrer plus tôt). La différence est qu'un certain nombre de fonctions existent déjà et ont été codés dans un language pré-compilé tel que le C. il est ainsi en général plus rapide en terme de temps de calcul. Pour pouvoir utiliser la librairie il faut ajouter au préambule du fichier la ligne suivante:

```python
import numpy as np
```

Dans un premier temps, il est possible de de créer un tableau de type **numpy** à partir d'une liste **tab** (ou liste de liste existante) en utilisant la commande suivante:

```python
tab_numpy = np.array(tab)
```

il est tout à fait possible de partir d'une liste de nombre complexes (natifs et non pas de la classe **NombreComplexe** malheureusement 😕, à moins de faire une fonction qui convertit la classe NombreComplexe en objet natif...).

1. Redéfinir une fonction **nombre_complexe_numpy(k, l, n_y, n_x)** qui renvoie cette fois un nombre complex natif.
2. Redéfinir une fonction **grille_complexe_numpy(n_y, n_x)** qui crée une liste de liste pour la grille à partir de nombres complexes natifs et créer un équivalent en objet numpy à renvoyer.

Considérons maintenant les opérations sur les tableaux, Pour ce faire, il faut comprendre que lorsque l'on additionne, soustrait ou multiplie des tableaux entre eux, les opérations se font point par point (tous les éléments s'additionnent, se soustraient ou se multiplient) comme on a pu le voir pour la classe matrice en TD. (Plus de détails à voir sur https://numpy.org/doc/stable/user/quickstart.html).

3. Essayer de faire des exemples de tableaux simples de taile artbitraire faites des opérations d'adition, soustration et multiplication. Observez le résultat.
4. À partir de la fonction **grille_complexe_numpy(n_y, n_x)**, créer une grille complexe et construire un nouveau tableau numpy correspondant au module pour chaque élément du tableau (en faisant des opérations sur tableaux). Comparer avec le module de la partie précédente.
5. Afin de calculer l'image de Mandelbrot, on donne la fonction suivante:

```python
def image_mandelbrot_numpy_couleur(n_y, n_x, N):
    """Crée une image de Mandelbrot couleur de taille définie par les entrées et
    paramétrée par un nombre d'itérations maximum. Version avec numpy.

    Parameters
    ----------
    n_y : int
```

```
        nombre de points en ligne.
    n_x : int
        nombre de points en colonne.
    N : int
        le nombre d'itérations maximum à partir du quel on considère
        que la suite converge.

    Returns
    -------
    array, de taille (n_y, n_x)
        la grille sous forme d'un array numpy.
    """
    c = grille_complexe_numpy(n_y, n_x)
    z = np.zeros((n_y, n_x), dtype=complex)
    masque_non_divergent = np.full((n_y, n_x), True, dtype=bool)
    image = np.zeros((n_y, n_x))
    for n in trange(N):
        z[masque_non_divergent] = z[masque_non_divergent]**2 +\
                                  c[masque_non_divergent]
        masque_nouveau_divergent = np.logical_and(
            masque_non_divergent, np.abs(z) > 2
        )
        image[masque_nouveau_divergent] = n
        masque_non_divergent = (np.abs(z) <= 2)
    return image
```

Tester la fonction en l'ajoutant à votre code. Comparer le résultat avec la fonction faite maison plus tôt et comparer également vitesse d'exécution. Que peut-on conclure ?