FRACTALES

Fichiers de données

L'objectif du prochain exercice est de visualiser des fractales de la famille des ensembles de Julia.
Les données pour tester ce programme sont stockées dans un fichier. Le travail préalable demandé ici consiste à lire ce fichier pour récupérer les données utilisées dans le programme.

Les données à lire sont les valeurs de *xmin*, *xmax*, ymin, ymax, *nx*, *ny*, *niter* et une succession de paires de lignes contenant alternativement un coefficient *p* et un complexe *cpx*. Vous découvrirez le sens de ses valeurs dans la suite de l'énoncé.

Le fichier contenant les données est structuré de la façon suivante :

```
# xmin xmax
xmin xmax
# ymin ymax
ymin ymax
# nx ny
nx ny
# niter
niter

# nombre de données à tester
n
# suite de 2 lignes avec p sur la 1ère ligne et preelle et pimag sur la seconde
p
pripim
p
pripim
p
pripim
…
```

Tout ce qui suit un caractère # est un commentaire et les lignes blanches doivent être ignorées.
Les complexes sont stockés dans le fichier sous la forme *aib*

On vous demande d'écrire une fonction de lecture des données contenues dans ce fichier et de récupérer les valeurs de *xmin*, *xmax*, ymin, ymax, *nx*, *ny*, *niter* puis de construire une liste avec des couples (*p, Complexe(preelle, pimage))* lus sur les lignes suivantes.

- On reprend la clase *Complexe* définie préalablement
- Une fonction de lecture *lire_ligne* permettant d'ignorer les lignes de commentaires et les lignes blanches est fournie sur Arche. Elle peut être utilisée en lieu et place de la fonction *readline* pour lire les 1ères lignes du fichier.

Exemple de résultat sur le fichier ci-dessous :

xmin = -1 xmax = 1

ymin = -1 ymax = 1

nx = 100 ny = 100

niter = 256

n = 2

lparametres = [(2, Complexe(-1.2, 1.2)), (4, Complexe(0.484, 0))]

```
# xmin xmax
-1 1
# ymin ymax
-1 1
# nx ny
100 100
# niter
256

# nombre de données
2
# suite des autres paramètres
2
-1.2i1.2
4
0.484i0
```

Ensembles fractals de julia et de mandelbrot

1. Introduction

L'objectif de cet exercice est de visualiser des fractales de la famille des ensembles de Julia et de Multibrot.

1.1 Ensembles fractals

L'ensemble de Julia est défini à partir du comportement d'une suite de nombres complexes et est caractérisé par le fait qu'une petite perturbation au départ se répercute en un changement radical de cette suite de nombres complexes (chaos). L'ensemble de Julia offre de nombreux exemples de fractales.

Pour simplifier la définition de cet ensemble, on le contraindra à des fonctions polynomiales particulières et on le définit à partir de la divergence de cette suite de nombre complexes.

1.1.1 Ensemble de Julia

1.1.1.1. Appartenance à l'ensemble de Julia

Soient z et c deux nombres complexes et p un nombre entier positif. On considère la suite u_n de nombres complexes, n étant un nombre entier positif, définie par la récurrence :

$$\begin{cases} u_0 = z \\ u_{n+1} = u_n^p + c \end{cases}$$

On considérera que la valeur initiale z appartient à l'ensemble de Julia si et seulement si la suite u_n reste bornée (elle ne diverge pas). Il existe un critère simple permettant de savoir en un temps fini si la suite diverge :

Théorème : la suite u_n diverge si et seulement si un des u_n a une norme supérieure à 2

En pratique il se peut que la divergence de la suite u_n n'apparaisse pas dans les premiers termes de la suite, on limite donc le nombre d'itérations maximal autorisé pour effectuer cette recherche. Le nombre maximal d'itérations autorisé correspond approximativement au temps alloué pour le calcul. Il faut donc choisir une valeur assez faible. Néanmoins, plus ce nombre est faible, moins les détails de l'ensemble apparaissent. Un bon compromis est en général 128, mais tout dépend de la partie du plan complexe que vous voulez représenter.

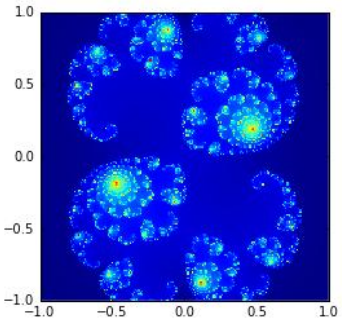
https://fr.wikipedia.org/wiki/Ensemble_de_Julia

1.1.1.2 Représentation des ensembles de Julia

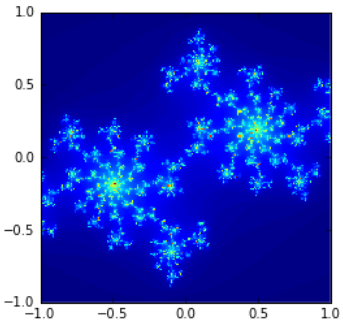
L'une des façons les plus simples de visualiser cet ensemble remarquablement élaboré consiste à supposer que chaque point dans le plan complexe représente une valeur *z (a +ib)*. On associe à chaque point z dans le plan complexe, le nombre d'itérations nécessaires pour qu'il diverge par rapport au nombre d'itérations maximal autorisé. Si le point ne diverge pas (il appartient alors à l'ensemble de Julia), la valeur 1 lui sera associée.

https://www.youtube.com/watch?v=Y4ICbYtBGzA

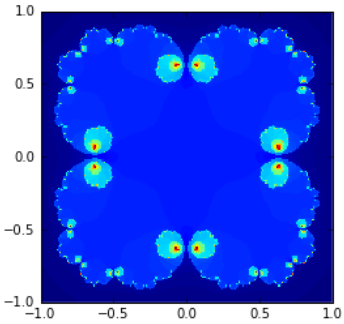
Exemples de représentation d'ensembles de Julia :



p = 2, c = 0.285 + 0.01i



p = 2, c = -0.70176 -0.3842i



p = 4, c = 0.484

1.1.2 Ensemble de multibrot et Mandelbrot

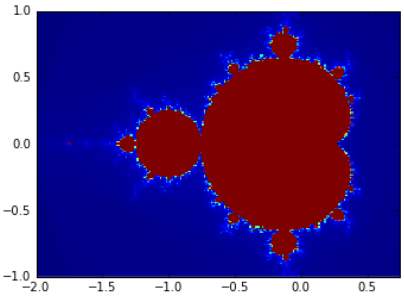
Un ensemble de multibrot peut être défini par la suite complexe suivante :

$$\begin{cases} u_0 = z \\ u_{n+1} = u_n^p + z \end{cases}$$

On considérera que z appartient à cet ensemble si et seulement si la suite u_n reste bornée (elle ne diverge pas). L'ensemble de Mandelbrot correspond au cas particulier où p = 2.

https://fr.wikipedia.org/wiki/Ensemble_de_Mandelbrot

Exemple d'ensemble de Mandelbrot :



2. Démarche

Pour atteindre notre objectif nous vous proposons la démarche décrite ci-dessous.

2.1 La classe Complexe

Reprendre la classe Complexe définie précédemment et y ajouter la surcharge de l'opérateur * (méthode *__mul__*) ainsi que de l'opérateur ** (méthode *__pow__*).

Je veux vérifier mon programme, je vais à la section 0

2.2 La classe SuiteComplexe

La classe *SuiteComplexe* permet la définition d'une suite de type $u_{n+1} = u_n^p + c$. Elle est caractérisée par les attributs *p_* (nombre entier), la constante *c_* (nombre complexe), le nombre d'itérations maximal pour tester la convergence de cette suite *niter_* (par défaut 128), ainsi que par sa valeur initiale *u0_* (par défaut Complexe(0, 0)).

- Définir le constructeur *__init__* de la classe *SuiteComplexe*. Les valeurs de tous les attributs sont fournis en paramètres du constructeur.
- Définir les mutateurs de la valeur initiale u_0 (set_u0) et de la constante *c* (*set_c*)
- Définir l'accesseur du nombre d'itérations maximal (*get_nbiter*)
- Définir la méthode *suivant* qui calcule u_{n+1} à partir de u_n. Cette méthode reçoit en paramètre un terme u_n et retourne le terme suivant correspondant à u_{n+1}
- Définir la méthode *diverge* qui renvoie le nombre d'itérations à partir duquel la suite diverge. Si la condition de convergence reste vérifiée, cette méthode renvoie le nombre d'itérations maximal.
  Rappel de la section 1.1.1.1 :

  Théorème : la suite u_n diverge si et seulement si un des u_n a une norme supérieure à 2

  Aide :

  1. *Je me débrouille seul, super je continue.*
  2. *J'ai besoin d'aide, je peux aller regarder l'algorithme à mettre en œuvre à la section 0*

2.3 La classe EnsembleJulia

Définir une classe *EnsembleJulia* qui correspond à la représentation graphique d'un ensemble de Julia. Une instance de cette classe est caractérisée par l'attribut *suite_* correspondant à la suite complexe qui lui est associée. Les paramètres *p* et *c* de la suite, ainsi que le nombre d'itérations maximal pour tester la convergence de cette suite (par défaut 128) sont fournis en paramètres du constructeur.

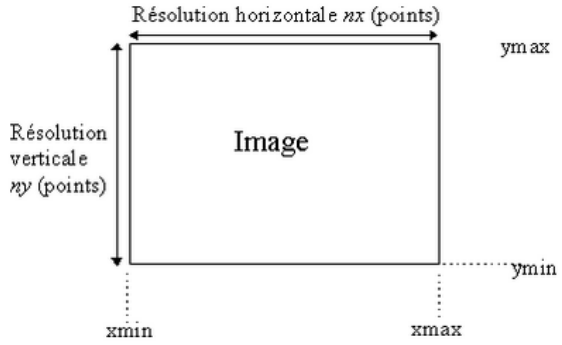
Par ailleurs, de façon à pouvoir représenter un ensemble de Julia, les instances de cette classe disposent des attributs suivants :

- les coordonnées *xmin_, xmax_, ymin_, ymax_* définissant la partie du plan complexe que l'on veut représenter, les valeurs des complexes z prises en compte dans l'ensemble étant comprises entre :

  $x_{min\_} \leq \text{Re } z \leq x_{max\_}$

  $y_{min\_} \leq \text{Im } z \leq y_{max\_}$

- La résolution de la grille qu'on veut afficher graphiquement : *nx_* et *ny_*



***Aide** :*

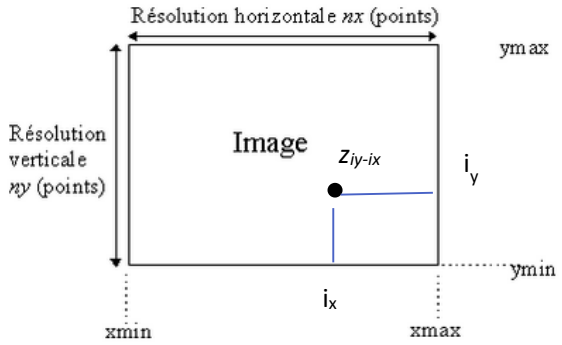
1. *Je me débrouille seul, super je continue.*
2. *J'ai besoin d'aide, je peux aller regarder quelle forme peut avoir l'entête du constructeur à la section 0*

- Définir la méthode *calculePoint* qui associe à un point *z* du plan complexe, le nombre d'itérations nécessaires pour que la suite associée de valeur initiale z diverge divisé par le nombre d'itérations maximal autorisé.

  Aide :

  1. *Je me débrouille seul, super je continue.*
  2. *J'ai besoin d'aide, je peux aller regarder l'algorithme à mettre en œuvre à la section 0*

- Définir la méthode *creeImage* qui calcule un nouvel attribut *image_*, matrice *numpy* à deux dimensions (*nx, ny*) telle que :
  - Les valeurs contenues dans les cases de la matrice *image_* décrivent le dessin devant apparaître dans la fenêtre graphique
  - A chaque coordonnées (*iy, ix*) de l'*image_* est associé un $z_{iy\text{-}ix}$ du plan complexe calculé à partir de *xmin, xmax, ymin, ymax et nx et ny*.
  - La case *image_[iy, ix]* correspondra à la valeur calculée par la méthode *calculePoint*.



Aide :

1. *Je me débrouille seul, super je continue.*
2. *J'ai besoin d'aide, je peux aller regarder l'algorithme à mettre en œuvre à la section 0*

- La méthode *dessineImage* qui permet de dessiner un ensemble de Julia est fournie sur Arche.

2.4 Application

On pourra tester le programme sur les valeurs stockées dans le fichier fourni sur arche qu'on pourra lire à l'aide de la fonction écrite au début de l'énoncé. Les valeurs stockées à la fin du fichier correspondent aux valeurs de p et c des suites à tester.

Je veux vérifier mon programme, je vais à la section 0

2.5 Les sous-classes Multibrot et Mandelbrot

Définir la classe *Multibrot* qui est une **sous-classe** de la classe *EnsembleJulia* (voir section 1.1.2). On pensera à modifier la méthode *calculePoint*.

Définir la sous-classe *Mandelbrot,* sous-classe de la classe *Multibrot*.

2.6 Application

On pourra tester le programme sur les valeurs suivantes :

	xmin	xmax	ymin	ymax	nx	ny	niter	p
Multibrot	-1	1	-1.2	1.2	100	100	128	3
Multibrot	-1	1	-1.2	1.2	100	100	128	4
Mandelbrot	-1	1	-1.2	1.2	100	100	128	
Mandelbrot	-2.1	0.6	-1.2	1.2	100	100	256	
Mandelbrot	-0.19050	-0.13824	1.01480	1.06707	200	200	128	

Aides

3.1 Question 2.1

```
cpx1 = Complexe(2, 7)
cpx2 = Complexe(6, 3)
cpxmul = cpx1 * cpx2
print(cpxmul)
```

Donne le résultat :

```
-9 + 48 i
```

```
    cpx1 = Complexe(-5, 10)
    cpx2 = Complexe(4, 8)
    cpxmul = cpx1 * cpx2
    print(cpxmul)
```

Donne le résultat :

```
-100 + 0 i
```

```
    cpx1 = Complexe(-5, 10)
    print(cpx1**4)
```

Donne le résultat :

```
-4375 + 15000 i
```

3.2 Algorithme de la méthode diverge question 2.2

```
def diverge:
     # en entrée : self une suite de complexe
     # en résultat : le nombre d'itération correspondant au dépassement du critère de
                     convergence pour la norme.
                     S'il n'y a pas convergence, renvoie niter

     un = valeur initiale de la suite

     pour i de 0 à nombre d'itérations maximal faire :

            si la norme de un > 2 :
            # on atteint la condition de divergence
            alors on retourne i
```

```
            sinon un ← valeur suivante de la suite

      retourner la valeur maximale atteinte
```

```
    c = Complexe(0.285, 0.01)
    zn = Complexe(-0.98, -1.0)

    suite = SuiteComplexe(2, c, 256, zn)
    print(suite.suivant(zn))
    print(suite.diverge())

    z = Complexe(-0.09, -0.44)
    suite = SuiteComplexe(3, c, 128, z)
    print(suite.diverge())
```

Donne les résultats :

```
0.2453999999999999 + 1.97 i
2
128
```

3.3 Constructeur de la classe EnsembleJulia

```
class EnsembleJulia:
    """La classe ensemble de Julia représente un ensemble de points du plan complexe
    divergeant pour une suite donnée."""

    def __init__(self, xmin, xmax, ymin, ymax, nx=100, ny=100, p=2, z0=None, c=None,
niter=128):
```

3.4 Algorithme de la méthode calculepoint

```
def calculePoint :
      # en entrée : une instance de l'ensemble de Julia (self)
                    et un point z du plan complexe
      # en résultat : nombre d'itérations pour que la suite associée à self diverge
                      divisé par le nombre d'itérations maximal possible

    affectation du premier terme de la suite à z
    resultat = nombre d'itérations à partir de laquelle la suite diverge / nombre max
d'itérations autorisé
```

3.5 Algorithme de la méthode creeimage

Premier algorithme, pouvant être optimisé :

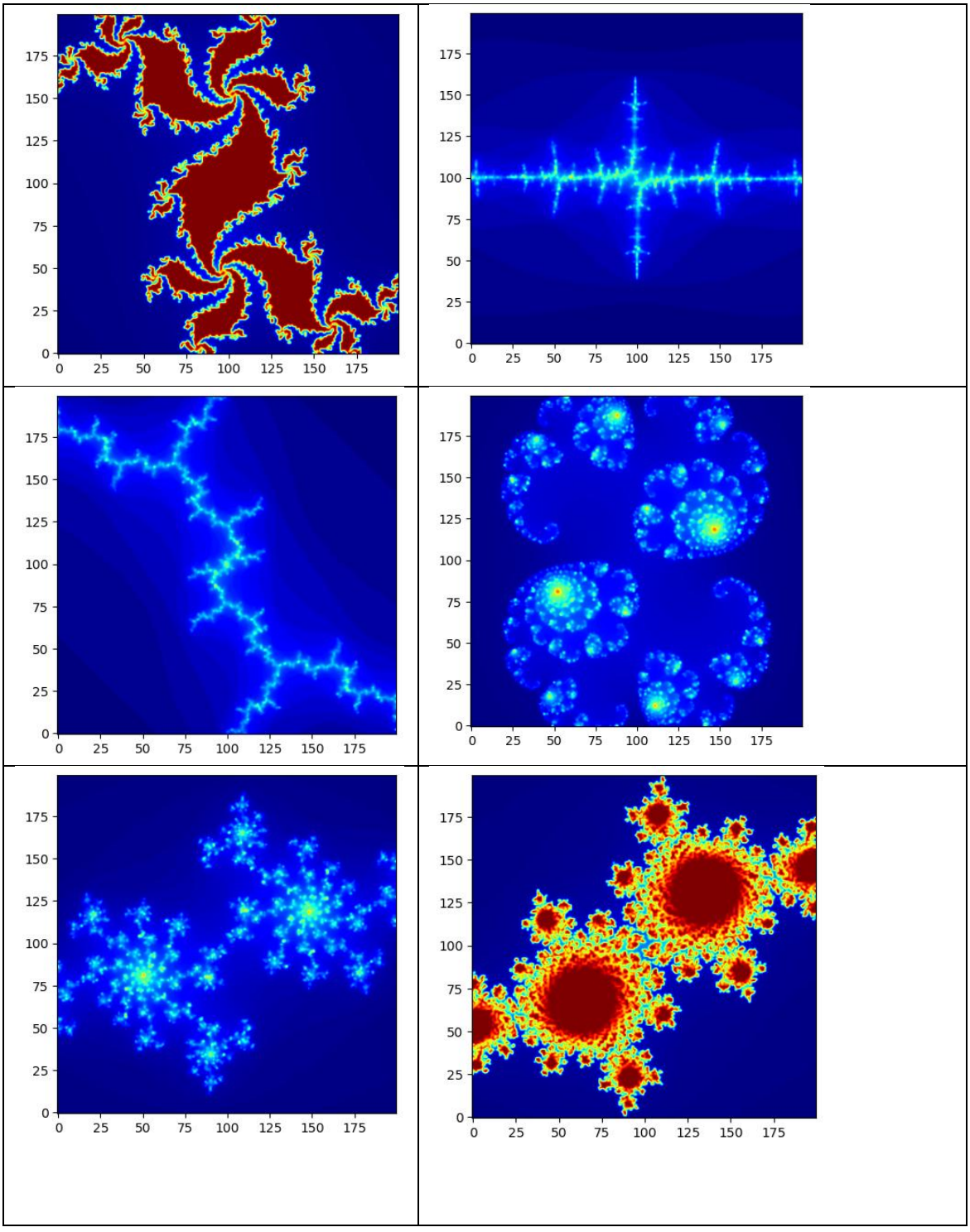
```
def creeImage:
      # en entrée : une instance de l'ensemble de Julia self
      # en résultat : une matrice de type numpy permettant de représenter l'ensemble de Julia
      # dans cette matrice, la valeur du point ziy-ix est calculé par la méthode CalculePoint

    initialiser la matrice numpy avec des 0
    # Calculer dx et dy distance entre chaque abscisse et chaque ordonnée
    dx ← (xmax – xmin) / (nx-1)
    dy ← (ymax - ymin) / (ny-1)

     Pour chaque iy sur l'axe des y:
      # les y correspondent aux lignes dans la matrice
      Pour chaque ix sur l'axe des x:
             # les x correspondent aux colonnes dans la matrice
             calculer preelle et pimag les parties réelle et imaginaire du complexe z(ix, iy)
             Créer le complexe z
            image[iy, ix] = résultat de calculePoint(z)

    retourner image
```

3.7 Résultats des tests sur les ensembles de Julia



3.8 Résultats des tests sur multibrot et Mandelbrot