On va travailler sur ce TP sur l'affichage de l'ensemble de Mandelbrot.

1 Classe `Complex`

Commencez par créer une classe pour stocker des complexes. Cette classe devra contenir les méthodes suivantes :

- un constructeur `__init__(self, real_part, imaginary_part)` qui initialise les parties réelles et imaginaire d'un complexe,
- une méthode `square(self)` qui renvoie le carré du complexe `self`,
- une méthode `modulus(self)` qui renvoie le module du complexe `self`,
- une méthode `add(self, other_complex)` qui renvoie un nouveau complexe étant la somme de `self` et `other_complex`.

2 Classe `RecursiveSequence`

Vous allez créer une classe permettant de représenter des suites définies par récurrence.

2.1 Définition de la classe

Cette classe contient deux attributs :

- `current_value` : la valeur courante de la suite qui sera au départ la valeur initiale de la suite,
- `recurrence_relation` : la fonction permettant de calculer le terme de rang $n+1$ à partir du terme de rang n.

Cela peut sembler étrange mais une fonction en Python est un objet comme un autre et peut donc être utilisé comme valeur d'une variable : on parle de *programmation fonctionnelle*.

Ajoutez une méthode `next_value(self)` à cette classe, qui renvoie la valeur courante de la suite tout en mettant à jour cette valeur en appelant `recurrence_relation`.

2.2 Lambda expression

Python offre une syntaxe intéressante qui permet de définir des mini-fonctions d'une ligne à la volée. La syntaxe utilise le mot-clé `lambda` suivi des arguments de la fonction suivis de `:` suivi de l'expression que doit retourner la fonction (qui peut alors utiliser les arguments). L'exemple ci-dessous illustre la définition de la même fonction de manière classique (`f`) ou bien à l'aide d'une `lambda` expression.

```
>>> def f(x):
      return x * 2
>>> f(3)
6
>>> g = lambda x: x * 2
>>> g(3)
6
```

Cette syntaxe permet donc de définir des fonctions très rapidement, ce qui sera très utile pour créer des suites qui demande une fonction.

2.3 Test de la fonction et affichage de suite

Vous allez tester la classe que vous venez de créer en affichant les 100 premiers termes la suite u_n définie par :

$$\begin{cases} u_0 = 0 \\ u_{n+1} = u_n + 3 \end{cases}$$

Pour cela vous aller devoir utiliser `pyplot` qui permet d'afficher des courbes. Il va vous falloir calculer deux listes `X` et `Y` contenant les valeurs des coordonnées des points que vous souhaitez afficher :

— `X` devra contenir les entiers de 0 à 99 inclus

— `Y` devra contenir les termes $u_0, u_1 \ldots, u_{99}$ de la suite.

Ensuite il vous suffit d'utiliser le code suivant pour afficher la suite :

```
from matplotlib import pyplot

pyplot.plot(X, Y)
pyplot.show()
```

3 Classe `Mandelbrot`

L'ensemble de Mandelbrot est une fractale définie comme l'ensemble des points c du plan complexe pour lesquels la suite de nombres complexes définie par récurrence ci-dessous est bornée :

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

Vous allez définir une classe `Mandelbrot` afin de représenter les différentes suites de Mandelbrot (dépendant du paramètre c). Cette classe devra contenir les éléments suivants :

— un constructeur `__init__(self, complex_value)` qui initialise un attribut `sequence` contenant la suite (instance de `RecursiveSequence`) de Mandelbrot définie par le complexe `complex_value`.

— une méthode `diverging_value` qui calcule les termes de la suite jusqu'à ce que le module d'un terme soit supérieur à 2 ou bien qu'on ait calculé 50 termes. Cette fonction renverra le rang plus un du premier terme dont le module est supérieur à 2 ou bien 0 si aucun terme calculé n'avait un module supérieur à 2. On utilise cette méthode pour avoir un test efficace (bien qu'approché a priori) du caractère divergent de la suite de Mandelbrot.

4 Affichage de l'ensemble de Mandelbrot

Vous allez maintenant afficher l'ensemble de Mandelbrot grâce à la classe que vous venez de définir. Pour cela, on va définir les coordonnées des points que l'on souhaite afficher. On va utiliser une résolution de 400×400, des valeurs de x comprises entre -2 et 0,5 et des valeurs de y comprises entre $-1,25$ et 1,25. On peut faire cela avec le code suivant (à l'aide du package `numpy`) :

```
from numpy import linspace

xmin, xmax = -2.0, 0.5
ymin, ymax = -1.25, 1.25
nx, ny = 400, 400
X = linspace(xmin, xmax, nx)
Y = linspace(ymin, ymax, ny)
```

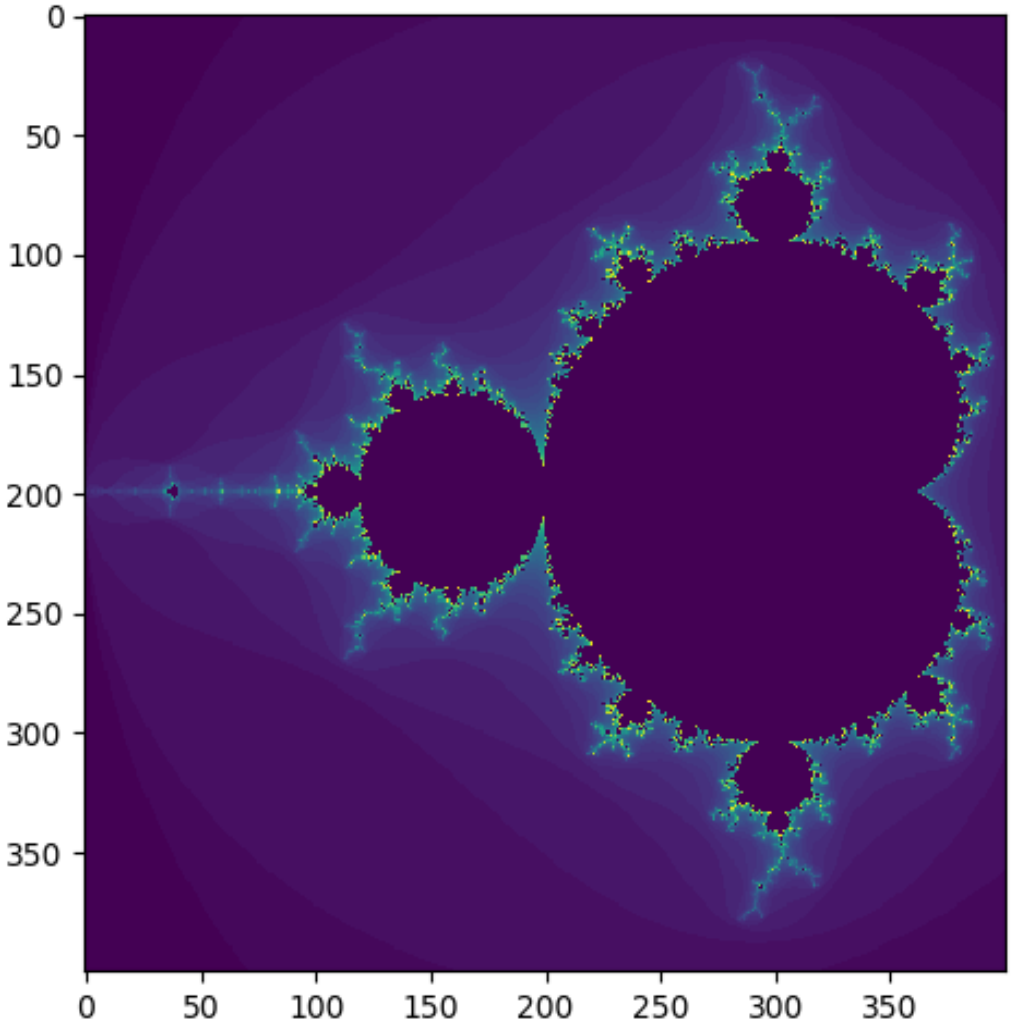
Maintenant, il vous suffit de calculer la valeur de `diverging_value` pour chacun des complexes correspondant aux points (x, y) avec $x \in X$ et $y \in Y$ et de stocker les valeurs dans une matrice. Pour calculer la matrice, nous vous conseillons d'écrire une fonction `compute_line(y)` qui calcule la ligne y et la renvoie, puis une fonction `compute_matrix()` qui calcule la matrice et la renvoie.

Vous pouvez ensuite afficher la matrice avec le code suivant :

```
from matplotlib import pyplot

pyplot.imshow(compute_matrix())
pyplot.show()
```

Vous devriez obtenir l'image suivante après le calcul :



Pour mesurer le temps de calcul, on peut utiliser la fonction `perf_counter()` du module `time`. Cette fonction retourne un flottant. La différence des valeurs retournées entre deux appels de la fonction est égale au temps écoulé entre les deux appels. Le code suivant permet donc de récupérer et d'afficher le temps écoulé par l'exécution d'un code :

```
import time
```

```
start = time.perf_counter()
# Code à exécuter
end = time.perf_counter()
print("Temps d'execution : " + str(end - start) + " secondes")
```

Utilisez cette technique pour mesurer le temps nécessaire pour dessiner la fractale de Mandelbrot.

5 Programmation parallèle

Essayons de diminuer ce temps total de calcul en dispersant le travail sur plusieurs processeurs en parallèle. Pour cela, on explore un peu plus les possibilités de la programmation fonctionnelle dans un premier temps : c'est l'un des atouts de ce paradigme, qui permet ensuite de paralléliser très aisément le travail.

5.1 Map

En python, il existe une fonction `map` ayant deux argument et qui permet d'appliquer une fonction (premier argument) sur chacun des élément d'un objet iterable (deuxième argument qui peut être une liste) et de créer un itérateur (qu'on peut transformer en liste avec `list()`) contenant chacun des retours de la fonction. Par exemple le code suivant, permet de calculer la liste Y des carrés des entiers de 0 à 99.

```
X = list(range(100))
Y = list(map(lambda x : x**2, X))
```

Réécrivez la fonction `compute_matrix()` pour utiliser un `map`de `compute_line(y)` sur la liste `Y`.

5.2 Multiprocessing

Il existe une variante de `map` dans le packaqe `multiprocessing` qui permet de paralléliser le calcul entre plusieurs processus et donc de profiter des multiples processeurs de votre ordinateur.

Le code suivant permet de distribuer le calcul du `map` précédant avec deux processus et donc d'avoir un calcul potentiellement plus rapide.

```
from multiprocessing import Pool

pool = Pool(2)  # nombre de processus en parallèle
X = list(range(100))
Y = list(pool.map(lambda x : x**2, X))
```

Changez le code de la fonction `compute_matrix()` pour utiliser un `pool` de 2 processus pour le `map`. Qu'observez-vous sur le temps de calcul ?

On peut obtenir l'identifiant du processus en cours utilisant le package `os` et la fonction `os.getpid()`. Changez le code de la fonction `compute_line` pour que la valeur dépende de l'identifiant du processus. On pourra utiliser par exemple `(os.getpid() % 11) + 2` lorsque la valeur de `diverging_value` est `0` et `1` sinon : cette formule un peu obscure permet de répartir les identifiants des processus, qui sont des entiers assez grands, sur une petite plage de valeurs, permettant d'obtenir une image plus esthétique.

Changez le code pour utiliser davantage de processus. Que remarquez-vous ? Vous pouvez obtenir le nombre de processeurs de votre ordinateur avec la méthode `cpu_count()` de la bibliothèque `multiprocessing`. Tracez puis commentez la courbe du temps de calcul de la fractale de Mandelbrot en fonction du nombre de processus utilisé.