La librairie python numpy

Son but

Numpy permet d'effectuer des calculs matriciels.

Python permet déjà de créer des tableaux multi-dimensionnels, en imbriquant des listes. Mais aucune contrainte n'est imposée. Ces listes imbriquées peuvent ne pas être "rectangulaires" (chaque "ligne" peut avoir un nombre différents d'éléments), et il est possible de mettre n'importe quel type de variable dans chaque case.

```
[
    [1, 5.432, "cette ligne a 3 éléments"],
    ["il", "y a", 5, "éléments dans cette ligne", ("le dernier",
"étant", "un tuple")],
]
```

Cette structure offre une grande liberté, elle permet de manipuler les données plus facilement et d'avoir un code plus concis et plus clair. Mais les performances ne sont pas optimales. Si vous devez multiplier des milliers de nombres entre eux, l'interpréteur python va être obligé de vérifier le type des éléments un par un, pour s'assurer que l'opération de multiplication existe bien à chaque fois.

Numpy impose des contraintes sur ses tableaux. Chaque dimension a le même nombre d'éléments, et tous les éléments sont de même type. Cela permet d'augmenter énormément les performances par une implémentation en C, et de tirer profit des instructions de processeur effectuant une même opération sur plusieurs données.

Sans forcément aller jusqu'au fond du code source de numpy, on peut voir que la fonction permettant de créer une matrice (`numpy.array`) fait appel à une fonction plus interne `numpy.ndarray` , dont voici la documentation. Son code source est en C, il est défini quelque part par ici.

L'opérateur python `__getitem__` (les crochets `[]`), conventionnellement utilisé pour accéder à des sous-éléments d'une variable composée, a été implémenté sur les array numpy, avec énormément de possibilités. Il est possible de l'utiliser avec :

- un tuple ayant autant de valeurs que de dimensions dans la matrice, pour accéder à un élément unique.
- un tuple avec moins de valeurs, ou bien un tuple comportant des slices `a:b:c` , pour accéder à une sous-matrice.
- une matrice de booléens, pour indiquer plus précisément à quels éléments ou quelles sous-matrices on veut accéder.
- une ou plusieurs matrices de nombres entiers, pour réordonner les index d'éléments et de sous-éléments.

La plupart des opérateurs mathématiques sont également disponibles. Ceux-ci correspondent à chaque fois à une application de l'opérateur sur chaque élément de la

matrice.

Création de matrices, accès aux éléments

In [1]:
```
import numpy as np
# Création d'une matrice 2*3. Certains éléments sont des entiers, d'autres de
# numpy va choisir le type de données englobant les deux. On aura donc une ma
mat_a = np.array([[0, 1.1, 2.2],[0, -1, -2]])
print(mat_a)
```

```
[[ 0.   1.1  2.2]
 [ 0.  -1.  -2. ]]
```

In [2]:
```
# Une autre matrice 2*3. On met des éléments divers,
# mais on indique explicitement le type de données par le paramètre "dtype".
# Numpy va contraindre tous les éléments pour les convertir vers le type indi
mat_b = np.array([[0, 1, 2],[0, "10", 20.9]], dtype=int)
print(mat_b)
```

```
[[ 0  1  2]
 [ 0 10 20]]
```

In [3]:
```
# Une ligne de la matrice A
print(mat_a[1])
```

```
[ 0. -1. -2.]
```

In [4]:
```
# Une colonne de la matrice B. Le caractère ":" est un slice indiquant que l'
# Il fonctionne également sur certains types de bases du python, par exemple
# On est obligé d'indiquer ce slice qui prend tout, pour pouvoir indiquer exp
# après concerne la deuxième dimension et non la première.

# Notons que cette sous-matrice s'affiche sous forme d'une ligne.
# Le fait de n'avoir pris qu'une seule colonne permet de supprimer une dimens
# Le résultat est donc une matrice de 2 éléments (1 dimension), et non pas 2*
print(mat_b[:, 1])
```

```
[ 1 10]
```

In [5]:
```
# On peut créer des matrices à autant de dimensions que l'on veut.
# Ici, une matrice 2*2*3*3 :
hypercube = [
    [
        [
            ["cub1 devant haut gauche", "cub1 devant haut mil", "cub1 devant
            ["cub1 devant milieu gauche", "cub1 devant milieu mil", "cub1 dev
            ["cub1 devant bas gauche", "cub1 devant bas mil", "cub1 devant ba
        ],
        [
            ["cub1 derrière haut gauche", "cub1 derrière haut mil", "cub1 der
            ["cub1 derrière milieu gauche", "cub1 derrière milieu mil", "cub1
            ["cub1 derrière bas gauche", "cub1 derrière bas mil", "cub1 derri
        ],
    ],
    [
        [
            ["cub2 devant haut gauche", "cub2 devant haut mil", "cub2 devant
            ["cub2 devant milieu gauche", "cub2 devant milieu mil", "cub2 dev
            ["cub2 devant bas gauche", "cub2 devant bas mil", "cub2 devant ba
```

```
        ],
        [
            ["cub2 derrière haut gauche", "cub2 derrière haut mil", "cub2 der
            ["cub2 derrière milieu gauche", "cub2 derrière milieu mil", "cub2
            ["cub2 derrière bas gauche", "cub2 derrière bas mil", "cub2 derri
        ],
    ]
]
mat_hyper = np.array(hypercube)
# L'attribut shape indique la forme de la matrice,
# c'est à dire la taille de chacune de ses dimensions.
print(mat_hyper.shape)
```

```
(2, 2, 3, 3)
```

Opérations mathématiques simples

In [6]:

```
# Additions et produits de matrices.
# Attention, ce n'est pas le produit comme en mathématique.
# Ici, chaque élément de la première matrice est simplement multiplié
# avec l'élément correspondant de la deuxième.
print(mat_a + 1000*mat_b)
```

```
[[    0.   1001.1  2002.2]
 [    0.   9999.  19998. ]]
```

In [7]:

```
# Pour faire une multiplication matricielle, telle que définie en mathématiqu
# il faut utiliser la fonction matmul.
# L'une des deux matrices doit être transposée, pour avoir des dimensions 2*3
print(np.matmul(mat_a, mat_b.transpose()))
```

```
[[  5.5  55. ]
 [ -5.  -50. ]]
```

In [8]:

```
# Ou bien, des dimensions 3*2 et 2*3
print(np.matmul(mat_b.transpose(), mat_a))
```

```
[[  0.    0.    0. ]
 [  0.   -8.9 -17.8]
 [  0.  -17.8 -35.6]]
```

Sélection et modification via des matrice booléennes

In [9]:

```
# Récupération de certains éléments en les spécifiant via une matrice booléen
# Tous les éléments récupérés se retrouvent dans une matrice d'une seule dime
# car on les pioche un peu partout. Ça n'aurait pas de sens de garder des lig
mat_bool = np.array([[True, False, True], [True, False, False]])
print(mat_b[mat_bool])
```

```
[0 2 0]
```

In [10]:

```
# Ce n'est pas ce qui est le plus intéressant avec les matrices booléennes.
# En revanche, c'est très utile lorsqu'on veut modifier certains éléments et
mat_b[mat_bool] = 42
print(mat_b)
```

```
[[42  1 42]
 [42 10 20]]
```

In [11]:
```
# Lorsqu'on applique un opérateur de comparaison sur une matrice,
# cela génère une matrice booléenne ayant les mêmes dimensions.
# Le booléen est True ou False selon que la comparaison est vérifiée ou pas.
print(mat_b < 20)
```

```
[[False  True False]
 [False  True False]]
```

In [12]:
```
# On peut ensuite utiliser cette matrice booléenne générée "à la volée",
# pour ne modifier que certains de ses éléments.
# Par exemple : on multiplie par 51 tous les nombres inférieur à 20.
mat_b[mat_b < 20] *= 51
print(mat_b)
```

```
[[ 42  51  42]
 [ 42 510  20]]
```

Un peu de pratique, avec Mandelbrot

Maintenant que nous connaissons un peu mieux numpy, nous allons en profiter pour créer la fractale de MandelBrot.

Algorithme

On part du rectangle situé aux coordonnées (x = -2, y = 1) et (x = +1, y = +1). Pour chaque point C de ce rectangle, on considère le nombre complexe $c = a + i * b$.

Rappelons que le langage python sait manipuler nativement les nombres complexes. Le nombre imaginaire est noté "j", il faut obligatoirement mettre une valeur numérique devant, pour ne pas le confondre avec un nom de variable. Si vous avez une console python sous la main, vous pouvez tout de suite tester l'opération `1j * 1j`. Pas besoin de numpy.

Pour chaque nombre c, on calcule la suite :

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

C'est à dire qu'on calcule :

c

$c^2 + c$

$(c^2 + c)^2 + c$

$((c^2 + c)^2 + c)^2 + c$

$\dots$

Si cette suite tend vers l'infini, le point C ne fait pas partie de la fractale, sinon, il en fait partie. Intuitivement, on voit que si le point est éloigné de l'origine, la suite va augmenter assez vite. Alors que s'il est proche, cela peut rester dans des valeurs proches de 0. Mais il est difficile de visualiser où se situerait la limite.

Initialisation du plan complexe

Commençons par créer notre matrice de points C. On prend 16 points en largeur et 11 points en hauteur. C'est une très grosse approximation, mais cela permet d'avoir des matrices affichables. Il sera toujours possible de prendre plus de points après avoir validé le fonctionnement.

(Le code de cette partie du notebook concernant Mandelbrot est inspiré par cet article : https://tomroelandts.com/articles/how-to-compute-the-mandelbrot-set-using-numpy-array-operations).

In [13]:

```
m = 16
n = 11

# La fonction linspace génère une matrice de une dimension, contenant des nom
# répartis de manière linéaire. C'est un peu la fonction "range", mais pour n
# Ici, les nombres vont de -2 à +1 (inclus), et il y en a 16.
# C'est à dire : [-2.0, -1.8, -1.6, ... 0.6, 0.8, 1.0]
mat_x = np.linspace(-2, 1, num=m)
# On recopie cette ligne 11 fois, les unes en dessous des autres,
# pour avoir une matrice de deux dimensions 16*11
mat_xx = np.tile(mat_x, (n, 1))

# On utilise à nouveau linspace, pour générer 11 nombres à virgule, de -1 à +
# La fonction "reshape" permet de réorganiser la matrice sous une autre forme
# Ici, on crée une matrice "colonne". C'est à dire qu'elle a 2 dimensions.
# La première dimension compte 11 éléments,
# chacun d'eux contient un seul sous-élément : un nombre.
mat_y = np.linspace(-1, 1, num=n).reshape((n, 1))

# On recopie cette colonne 16 fois, pour avoir une autre matrice 16*11.
# Notez bien la différence des paramètres entre les deux fonctions tile.
# Pour la première, on indique (n, 1) et pour la seconde (1, m).
# C'est ce qui permet de faire une recopie "verticale" ou "horizontale".
mat_yy = np.tile(mat_y, (1, m))

# Puisque python gère les nombres complexes, on peut se permettre
# de gérer des matrices de nombres complexes.
# On additionne nos deux précédentes matrices, en utilisant mat_xx pour la pa
# et mat_yy pour la partie imaginaire.
# Nous avons maintenant notre rectangle de point C
mat_c = mat_xx + 1j * mat_yy
# On n'affiche que les deux premières lignes et la dernière,
# pour éviter d'afficher trop de données.
print(mat_c[(0, 1, -1), :])
```

```
[[-2. -1.j  -1.8-1.j  -1.6-1.j  -1.4-1.j  -1.2-1.j  -1. -1.j  -0.8-1.j
  -0.6-1.j  -0.4-1.j  -0.2-1.j   0. -1.j   0.2-1.j   0.4-1.j   0.6-1.j
   0.8-1.j   1. -1.j ]
 [-2. -0.8j -1.8-0.8j -1.6-0.8j -1.4-0.8j -1.2-0.8j -1. -0.8j -0.8-0.8j
  -0.6-0.8j -0.4-0.8j -0.2-0.8j  0. -0.8j  0.2-0.8j  0.4-0.8j  0.6-0.8j
   0.8-0.8j  1. -0.8j]
 [-2. +1.j  -1.8+1.j  -1.6+1.j  -1.4+1.j  -1.2+1.j  -1. +1.j  -0.8+1.j
  -0.6+1.j  -0.4+1.j  -0.2+1.j   0. +1.j   0.2+1.j   0.4+1.j   0.6+1.j
   0.8+1.j   1. +1.j ]]
```

Calcul avec une fonction parallélisée

Pour le calcul de la suite des nombres z_n, on adoptera également quelques approximations, car il n'est pas nécessaire de prouver mathématiquement qu'elle tend vers l'infini pour chaque point C. On effectuera l'algorithme suivant :

Sur 100 itérations :

- calculer le nombre suivant de la suite $z_n = z_{n-1}^2 + c$
- Si ce nombre complexe z_n a un module supérieur à 2, on considère que la suite tendra vers l'infini. Le point C n'appartient pas à l'ensemble de Mandelbrot.

Si on est arrivé au bout des 100 itérations sans dépasser, le point C appartient à l'ensemble de Mandelbrot.

On peut retirer une autre information de cette algorithme : le numéro d'itération à partir duquel le point atteint un module de 2. Plus on se rapproche de points appartenant à l'ensemble de Mandelbrot, plus ce nombre augmente. Il peut être utilisé pour afficher la couleur dans une représentation en image de la fractale.

On peut maintenant créer une fonction renvoyant "l'index d'itération Mandelbrot" à partir d'un nombre complexe c passé en paramètre.

In [14]:
```
def mandelbrot_index(c):
    z = 0j
    for i in range(100):
        z = z**2 + c
        if abs(z) > 2:
            # Ce point a quitté l'ensemble de Mandelbrot.
            # On renvoie le numéro d'itération jusqu'où on est allé
            # (En considérant que la première itération est indexé à 1, et no
            return i + 1
    # Après 100 itération, le module de z est toujours inférieur à 2.
    # On considère que ce point est dans l'ensemble de Mandelbrot.
    # Par convention, on décide de renvoyer 0.
    return 0
```

Il ne reste plus qu'à appliquer cette fonction sur chaque case de la matrice créée précédemment.

On pourrait le faire via deux boucles imbriquées : `for x in range(16): for y in range(11): etc.`. Mais on ne profiterait pas de toute la puissance de numpy.

Comme le calcul pour un point ne dépend pas des autres points de la matrice, on pourrait le paralléliser, ce que permet Numpy.

La fonction `np.vectorize` permet de créer un traitement parallélisé, applicable sur des matrices, à partir d'une fonction.

Il est possible de créer des traitements prenant plusieurs matrices en entrée. Il faut pour cela que la fonction originelle ait autant de paramètres que de matrices d'entrée.

Dans notre cas, c'est assez simple, il n'y a qu'un seul paramètre : le nombre c.

In [15]:
```
# On crée un traitement parallélisable à partir de la fonction définie ci-des
mandelbrotify_matrix = np.vectorize(mandelbrot_index)
# Ce traitement nécessite une matrice en entrée,
# et génère une nouvelle matrice contenant des entiers.
mat_mandelbrot = mandelbrotify_matrix(mat_c)
print(mat_mandelbrot)
```

```
[[ 1  1  2  3  3  3  3  4  4  7  0  4  3  2  2  2]
 [ 1  2  3  3  3  3  4  4  6  0 18  5  4  3  2  2]
 [ 1  3  3  3  3  4  5 12 26  0  0 12 15  3  3  2]
 [ 1  3  3  5  7  7  7  0  0  0  0  0  9  4  3  2]
 [ 1  4  5  6 18  0 15  0  0  0  0  0 31  4  3  2]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  7  4  3  3]
 [ 1  4  5  6 18  0 15  0  0  0  0  0 31  4  3  2]
 [ 1  3  3  5  7  7  7  0  0  0  0  0  9  4  3  2]
 [ 1  3  3  3  3  4  5 12 26  0  0 12 15  3  3  2]
 [ 1  2  3  3  3  3  4  4  6  0 18  5  4  3  2  2]
 [ 1  1  2  3  3  3  3  4  4  7  0  4  3  2  2  2]]
```

On reconnait (très approximativement) le dessin de la fractale.

Optimisation en utilisant les fonctions matricielles de base

Cependant, l'implémentation de l'algorithme n'est pas optimisée. `np.vectorize` offre une grande liberté, car la fonction originelle peut être plus ou moins compliquée. La parallélisation de cette fonction aide à aller un peu plus vite, mais si on avait de très grandes matrices, cela resterait très lent.

Si on pouvait effectuer notre algorithme en utilisant uniquement des opérations mathématiques de base directement sur les matrices, cela irait beaucoup plus vite. Les microprocesseurs ont des instructions spécialement prévues pour multiplier/additionner/etc. de grandes quantités de nombres.

La librairie numpy contient beaucoup de fonctions mathématiques, optimisées au maximum : https://numpy.org/doc/stable/reference/routines.math.html

Notre fonction `mandelbrot_index` n'est pas si compliquée que ça, on devrait pouvoir la convertir en fonctions de base. Au lieu d'avoir une approche "élément par élément", il faut avoir une approche "itération par itération".

- À chaque itération, on calcule la valeur suivante de la suite $z_n = z_{n-1}^2 + c$, pour toutes les cases de la matrice.
- Pour toutes les cases qui ont dépassé (module > 2), on enregistre l'index d'itération, et on ne fera plus de calcul avec. Pour les autres, on continue.
- À la fin des itérations, on laisse à zéro les cases qui n'ont jamais dépassé.

La difficulté dans cet algorithme est la partie "pour toutes les cases qui ont dépassé". Cela veut dire qu'on doit agir sur certaines cases d'une matrice, mais pas toutes. Ceci est possible grâce aux matrices booléennes.

On reprend l'algo, et tant qu'à faire, on met tout dans une fonction, car on s'en resservira plus tard.

In [16]:

```python
def make_mandelbrot_matrix(m, n):
    """
    m : largeur de la matrice.
    n : hauteur de la matrice.
    """

    # Même code que précédemment
    mat_x = np.linspace(-2, 1, num=m)
    mat_xx = np.tile(mat_x, (n, 1))
```

```
    mat_y = np.linspace(-1, 1, num=n).reshape((n, 1))
    mat_yy = np.tile(mat_y, (1, m))
    mat_c = mat_xx + 1j * mat_yy

    # Cette matrice contient tous les nombres z de chaque case, que l'on fera
    # Pour l'instant, elle est initialisée avec que des zéros.
    mat_z = np.zeros((n, m), dtype=complex)
    # Une matrice contenant uniquement des booléens True.
    # Certains deviendront False au fur et à mesure que la valeur z dépasse l
    mat_bool = np.full((n, m), True, dtype=bool)
    # La matrice contenant les "index d'itération Mandelbrot".
    # On l'initialise avec uniquement des zéros. On remplira avec les bonnes
    mat_mandelbrot = np.zeros((n, m), dtype=int)

    for i in range(100):
        # Calcul du terme suivant de la suite, uniquement pour les cases ayan
        # (Au départ, on le fait pour toutes les cases).
        mat_z[mat_bool] = mat_z[mat_bool]**2 + mat_c[mat_bool]
        # Mise à jour de mat_bool. On met à False les cases pour lesquelles
        # le nombre dans mat_z a un module supérieur à 2.
        # Nous avons ici un exemple de matrice booléenne générée "à la volée"
        # par l'instruction : np.abs(mat_z) > 2
        mat_bool[np.abs(mat_z) > 2] = False
        # Mise à jour de l'index d'itération pour les cases qui viennent de s
        # La génération de la matrice booléenne est ici un peu plus compliqué
        # On veut tous les éléments pour lesquels mat_bool est à False et mat_
        # C'est à dire les éléments qui sont sortis de l'ensemble, mais qui n
        # pas sortis à l'itération d'avant.
        # Pour tous ces éléments, on définit l'index d'itération, ce qui les
        # sortir définitivement de l'ensemble de Mandelbrot.
        mat_mandelbrot[np.logical_and(np.invert(mat_bool), mat_mandelbrot ==

    return mat_mandelbrot

mat_mandelbrot = make_mandelbrot_matrix(16, 11)
print(mat_mandelbrot)
```

```
[[ 1  1  2  3  3  3  3  4  4  7  0  4  3  2  2  2]
 [ 1  2  3  3  3  3  4  4  6  0 18  5  4  3  2  2]
 [ 1  3  3  3  3  4  5 12 26  0  0 12 15  3  3  2]
 [ 1  3  3  5  7  7  7  0  0  0  0  0  9  4  3  2]
 [ 1  4  5  6 18  0 15  0  0  0  0  0 31  4  3  2]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  7  4  3  3]
 [ 1  4  5  6 18  0 15  0  0  0  0  0 31  4  3  2]
 [ 1  3  3  5  7  7  7  0  0  0  0  0  9  4  3  2]
 [ 1  3  3  3  3  4  5 12 26  0  0 12 15  3  3  2]
 [ 1  2  3  3  3  3  4  4  6  0 18  5  4  3  2  2]
 [ 1  1  2  3  3  3  3  4  4  7  0  4  3  2  2  2]]
```

Pour finir ce chapitre, une représentation textuelle de la fractale. (On fera beaucoup plus joli dans le chapitre juste après).

In [17]:

```
# Conversion des cases de la matrice en string de 1 caractère chacune.
# Pour le coup, on utilise une fonction parallélisée.
# À ma connaissance, il n'y a pas de meilleure moyen.
# Numpy est fait pour manipuler des nombres, pas des strings.
def str_from_mandel_index(mandel_index):
    if mandel_index == 0:
        return ' '
    if mandel_index < 5:
        return '.'
    if mandel_index < 15:
        return 'o'
```

```
    else:
        return 'O'

stringify_mandel = np.vectorize(str_from_mandel_index)
str_mandel = stringify_mandel(mat_mandelbrot)
# On est obligé de faire une boucle pour rassembler tous les caractères.
# S'il existe une solution plus élégante, je suis preneur.
print(
    '\n'.join(
        ' '.join(cell for cell in line)
        for line
        in str_mandel
    )
)
```

```
. . . . . o   . . . . .
. . . . . o   O o . . . .
. . . . . o o O   o O . . .
. . . o o o o       o . . .
. . o o O   O       O . . .
                    o . . .
. . o o O   O       O . . .
. . . o o o o       o . . .
. . . . . o o O   o O . . .
. . . . . . o   O o . . . .
. . . . . . . o   . . . . .
```

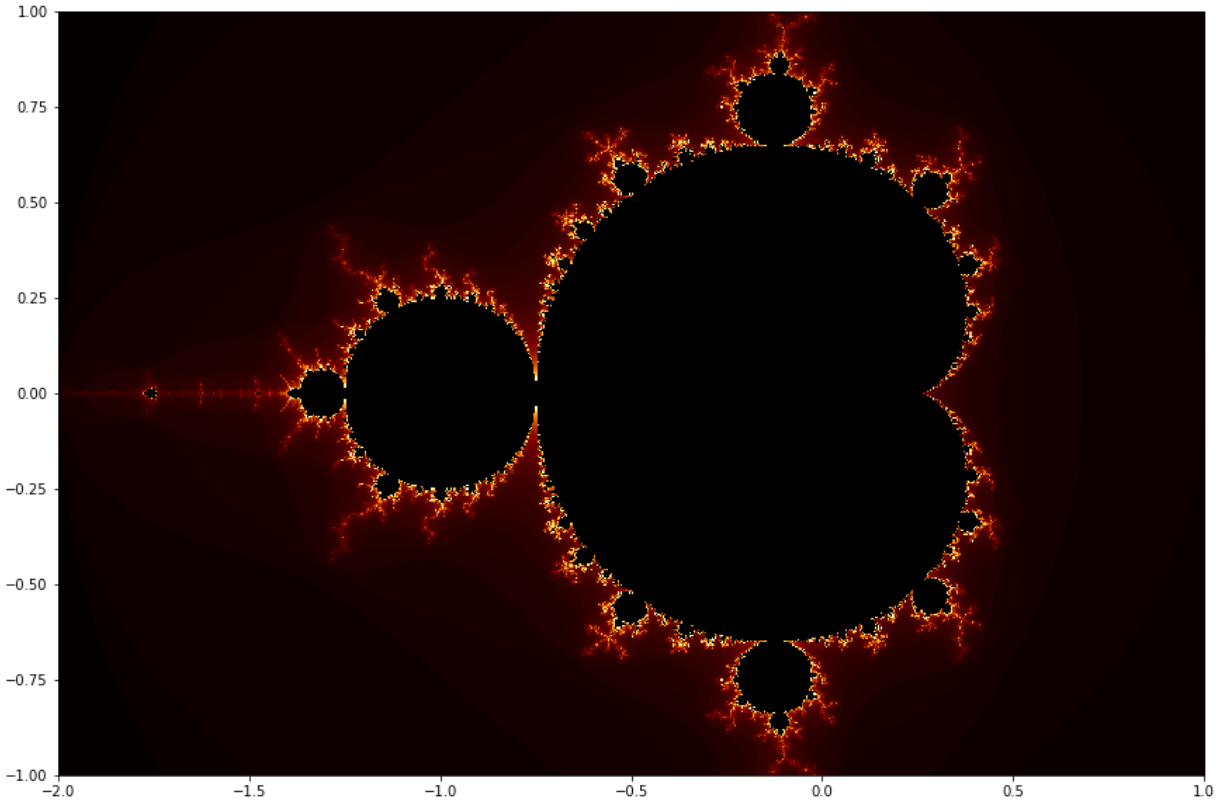
Représentation graphique

Nous utilisons la librairie python matplotlib pour générer une image à partir d'une matrice contenant la fractale. Plus "l'index d'itération Mandelbrot" est haut, plus le pixel correspondant est clair. Les points correspondants à l'ensemble de Mandelbrot en lui-même sont en noir.

In [18]:

```
import matplotlib.pyplot as plt

# Création de la fractale dans une matrice de dimension 750*500.
# On peut se le permettre, puisque notre fonction est super-optimisée.
mat_mandelbrot = make_mandelbrot_matrix(750, 500)
# Configuration de la taille de l'image que générera matplotlib.
# Les deux valeurs correspondent à la largeur et la hauteur de l'image, en in
# Les américains et le système métrique...
plt.figure(figsize = (15, 10))
# Génération de l'image. Le paramètre 'cmap' définit le jeu de couleurs.
# Voir : https://matplotlib.org/3.1.0/tutorials/colors/colormaps.html
# Le paramètre 'extent' permet de définir l'échelle des deux axes.
plt.imshow(mat_mandelbrot, cmap='afmhot', extent=[-2, 1, -1.0, 1.0], interpol
# Affichage de l'image générée
plt.show()
```



Maintenant qu'on a créé un bel algorithme, ce serait dommage de pas en profiter. Vous avez lu jusqu'ici, vous avez bien mérité une autre jolie image.

(Vous n'avez pas scrollé jusqu'en bas de ce notebook sans le lire, juste pour voir ce qu'il y avait à la fin, n'est-ce pas ?)

In [19]:

```
def make_zoomed_mandelbrot_matrix(m, n, start_x=-2, end_x=1, start_y=-1, end_
    """
    m : largeur de la matrice.
    n : hauteur de la matrice.

    start_x, end_x, start_y,, end_y : coordonnées (x, y) des coins supérieur
    et inférieur droit du rectange sur lequel on veut zoomer pour afficher la

    nb_iter : nombre d'itération à effectuer avant de décider si un point app
    l'ensemble de Mandelbrot ou pas. Plus on zoom, plus il faut mettre d'itér
    sinon il y a trop d'approximation et on ne voit que du noir.
    """

    # Même code que précédemment, mais avec les paramètres
    # start_x, end_x, start_y, end_y.
    # Pour zoomer sur une partie de la fractale
    mat_x = np.linspace(start_x, end_x, num=m)
    mat_xx = np.tile(mat_x, (n, 1))
    mat_y = np.linspace(start_y, end_y, num=n).reshape((n, 1))
    mat_yy = np.tile(mat_y, (1, m))
    mat_c = mat_xx + 1j * mat_yy

    # Même code que précédemment, à part le nombre d'itération.
    mat_z = np.zeros((n, m), dtype=complex)
    mat_bool = np.full((n, m), True, dtype=bool)
    mat_mandelbrot = np.zeros((n, m), dtype=int)

    for i in range(nb_iter):
        mat_z[mat_bool] = mat_z[mat_bool]**2 + mat_c[mat_bool]
        mat_bool[np.abs(mat_z) > 2] = False
```

```
        mat_mandelbrot[np.logical_and(np.invert(mat_bool), mat_mandelbrot ==

    return mat_mandelbrot

# Affichage de la fractale, zoomé sur le rectange (x = -0.75, y = 0.098), (x
mat_mandelbrot = make_zoomed_mandelbrot_matrix(800, 800, -0.75, -0.742, 0.098
plt.figure(figsize = (15, 10))
# Pour le paramètre extent, on met les mêmes valeurs que le rectangle de zoom
# On change le paramètre cmap pour avoir des couleurs différentes, c'est plus
plt.imshow(mat_mandelbrot, cmap='nipy_spectral', extent=[-0.75, -0.742, 0.098
plt.show()
```