Mandelbrot

March 17, 2019

1 L'ensemble de Mandelbrot

Marc Lorenzi
16 mars 2019

```
In [1]: import matplotlib.pyplot as plt
        import math
        import cmath
        import random
```

```
In [2]: plt.rcParams['figure.figsize'] = (10, 10)
```

Avertissement : Avant de lire ce notebook, regardez d'abord celui consacré aux ensembles de Julia.

L'ensemble de Mandelbrot est une partie de $\mathbb{C}$ étudiée et popularisée par le mathématicien Benoît Mandelbrot. L'arrivée d'ordinateurs permettant à peu de coût de représenter graphiquement des objets complexes a montré que cet objet, que nous noterons $\mathcal{M}$ en l'honneur de Benoît M., était d'une incroyable complexité.

Dans ce notebook nous allons nous contenter d'effleurer, quelques problèmes mathématiques élémentaires ainsi que quelques algorithmes liés à $\mathcal{M}$.

1.1 1. Première approche

1.1.1 1.1 C'est quoi l'ensemble de Mandelbrot ?

Soit $c \in \mathbb{C}$. Soit $f_c : \mathbb{C} \to \mathbb{C}$ la fonction définie par $f_c(z) = z^2 + c$. Considérons la suite $(z_n)_{n\geq 0}$ définie par $z_0 = c$ et pour tout $n \in \mathbb{N}$,

$$z_{n+1} = z_n^2 + c$$

La suite (z_n) est-elle convergente ? Si elle diverge, est-elle bornée ? Notons $\mathcal{M}$ l'ensemble des nombres complexes c tels que la suite (z_n) soit bornée. $\mathcal{M}$ est appelé **l'ensemble de Mandelbrot**. Cet ensemble est une partie très compliquée du plan complexe, et nous allons examiner dans ce qui suit quelques moyens de nous en faire une idée concrète.

Si vous avez lu le notebook sur les ensembles de Julia, alors vous avez compris que

$$\mathcal{M} = \{c \in \mathbb{C}, c \in K_c\}$$

où K_c est l'ensemble de Julia rempli associé à c.

Remarque : On peut aussi définir la suite (z_n) en posant $z_0 = 0$, puisqu'on a alors $z_1 = c$. Ainsi, on a aussi

$$\mathcal{M} = \{c \in \mathbb{C}, 0 \in K_c\}$$

Remarque : La suite (z_n) dépend de la valeur de c. Ne le perdez pas de vue dans tout le notebook, nous devrions systématiquement écrire $z_n(c)$, ce que je ne ferai pas pour alléger les notations.

1.1.2 1.2, Bornée, pas bornée ?

Avant de nous lancer sans réfléchir dans du code Python, voici quelques questions apparemment liées :

1. Comment montrer qu'une suite (z_n) est bornée ?
2. Comment montrer qu'une suite (z_n) n'est pas bornée ?
3. Comment montrer qu'une suite (z_n) est majorée en module par le nombre 2 (ou n'importe quel autre nombre) ?

Pour l'informaticien la situation n'est pas réjouissante :

- Il est impossible de répondre **algorithmiquement** "oui" aux questions 1 et 2.
- Il est impossible de répondre **algorithmiquement** "non" aux questions 1 et 2.
- Il est impossible de répondre **algorithmiquement** "oui" à la question 3.

Par contre :

- Il est possible de répondre **algorithmiquement** "non" à la question 3.

Prenons par exemple la fonction ci-dessous. Elle renvoie `False` si la suite $(|z_n|)$ n'est pas majorée par le réel A. En revanche, elle ne termine pas si la suite est majorée en module par A.

```
In [3]: def bornee(c, A):
            z = c
            while abs(z) <= A:
                z = z * z + c
            return False

In [4]: bornee(0.26, 2)

Out[4]: False
```

Pour $c = 0.26$, la suite (z_n) n'est donc pas majorée en module par 2.

Mais si nous essayons `bornee(0.25, 2)` nous risquons d'attendre longtemps ... l'éternité en gros, comme nous le verrons plus loin :-). Bref, pour décider si un nombre $c \in \mathbb{C}$ est dans l'ensemble de Mandelbrot, il va nous falloir réfléchir un peu à la façon de procéder. Commençons par un résultat qui va nous aider.

1.1.3 1.3 Bornée, mais par quoi ?

Théorème : Soit $c \in \mathbb{C}$. Alors $c \in \mathcal{M}$ si et seulement si pour tout entier $n \geq 0$, $|z_n| \leq 2$.

Démonstration : Dans un sens c'est évident. Si pour tout $n \geq 2$, $|z_n| \leq 2$, alors la suite (z_n) est bornée et donc, par définition de $\mathcal{M}$, on a $c \in \mathcal{M}$. Nous allons maintenant prouver la réciproque. Cela se fera en deux temps.

1.3.1 Dans un sens ... **Lemme 1** : Soit $c \in \mathbb{C}$ tel que $|c| > 2$. Alors $|z_n(c)| \to +\infty$ lorsque n tend vers l'infini et, donc, $c \notin \mathcal{M}$.

Démonstration : Posons $|c| = 2 + h$, où $h > 0$. On a $|z_0| = |c| = 2 + h$. Soit $n \in \mathbb{N}$. Supposons l'existence d'un réel $t_n > 0$ tel que $|z_n| \geq 2 + t_n h$. On a alors

$$|z_{n+1}| = |z_n^2 + c| \geq |z_n|^2 - |c| \geq (2 + t_n h)^2 - (2 + h)$$

Mais

$$(2 + t_n h)^2 = 4 + 4t_n h + t_n^2 h^2 \geq 4 + 4t_n h$$

De là,

$$|z_{n+1}| \geq 2 + 3t_n h$$

On en déduit facilement que pour tout $n \geq 0$,

$$|z_n| \geq 2 + 3^n h$$

et donc que $|z_n|$ tend vers $+\infty$ lorsque n tend vers l'infini.

Corollaire : $\mathcal{M} \subset \mathcal{D}(O, 2)$ où

$$\mathcal{D}(O, 2) = \{z \in \mathbb{C}, |z| \leq 2\}$$

est le disque fermé de centre 0 et de rayon 2.

1.3.2 Et puis dans l'autre **Proposition** : Soit $c \in \mathbb{C}$ tel que $|c| \leq 2$. Supposons que pour un certain entier n on ait $|z_n| > 2$. La suite $(|z_n|)$ tend alors vers $+\infty$, et donc $c \notin \mathcal{M}$.

Démonstration : Posons $|z_n| = 2 + h$ où $h > 0$. On a

$$|z_{n+1}| = |z_n^2 + c| \geq |z_n|^2 - |c| \geq (2+h)^2 - 2 \geq 2 + 4h$$

Par une récurrence facile on montre que pour tout $p \in \mathbb{N}$,

$$|z_{n+p}| \geq 2 + 4^p h$$

Ainsi, la suite $(|z_n|)$ tend vers $+\infty$.

Conséquence : Si nous réunissons les résultats des deux paragraphes précédents, nous obtenons

$$\exists n, |z_n| > 2 \Rightarrow c \notin \mathcal{M}$$

Le théorème est démontré. On a même montré un peu mieux :

- Si $c \in \mathcal{M}$ alors $|c| \leq 2$ et pour tout entier n, $|z_n| \leq 2$.
- Si $c \notin \mathcal{M}$ alors $|z_n| \to +\infty$ lorsque n tend vers l'infini.

1.1.4 1.4 Maintenant, itérons

Voici LA fonction.

```
In [5]: def f(z, c):
            return z * z + c
```

Soit $c \in \mathbb{C}$. Nous savons que si, pour un certain entier n, on a $|z_n(c)| > 2$ alors $c \notin \mathcal{M}$. La question est : quel est cet entier n ? Est-on capable d'en avoir une estimation ? Pour l'instant, non. La fonction `iterer` ci-dessous prend en paramètres deux flottants x et y qui sont la partie réelle et imaginaire de c. Elle prend également un troisième paramètre `niter`. Une boucle est effectuée avec un indice k prenant les valeurs $0, 1, \ldots$.

- Si $k <$ `niter` en sortie de boucle, on est alors **certains** que $c \notin \mathcal{M}$.
- Si $k =$ `niter` la boucle termine mais le doute subsiste quant à l'appartenance de c à $\mathcal{M}$.

Disons que plus le paramètre `niter` est choisi grand, plus notre fonction nous fournira des renseignements précieux sur c.

```
In [6]: def iterer(c, niter):
            z = c
            k = 0
            while abs(z) <= 2 and k < niter:
                z = f(z, c)
                k = k + 1
            return k
```

```
In [7]: iterer(0.26, 128)
```

```
Out[7]: 29
```

On a donc $0.26 \notin \mathcal{M}$ puisque $29 < 128$.

```
In [8]: iterer(0.25, 1024)
```

```
Out[8]: 1024
```

Et donc ... rien du tout. $0.25 \in \mathcal{M}$, ou pas. Allons, un petit effort !
Proposition : $\frac{1}{4} \in \mathcal{M}$.
Démonstration : Posons, pour tout $n \in \mathbb{N}$, $z_n = \frac{1}{2} - w_n$. On a $w_0 = \frac{1}{4}$ et, pour tout entier n,

$$\frac{1}{2} - w_{n+1} = (\frac{1}{2} - w_n)^2 + \frac{1}{4}$$

ce qui donne

$$w_{n+1} = w_n - w_n^2$$

Exercice : La suite (w_n) converge vers 0. Indication : elle est décroissante, minorée par 0.
Ainsi, z_n tend vers $\frac{1}{2}$. Comme une suite convergente est bornée, nous avons $\frac{1}{4} \in \mathcal{M}$.

```
In [9]: iterer(0.25 + 1e-9, 100000)
```

```
Out[9]: 99343
```

Et donc $\frac{1}{4} + 10^{-9} \notin \mathcal{M}$ puisque $99943 < 100000$. Le point $\frac{1}{4}$ est donc sans doute remarquable. Est-il au "bord" de $\mathcal{M}$?

Exercice : Pour quelques valeurs très particulières de c il est possible de démontrer aisément que $c \in \mathcal{M}$. Montrer que $0 \in \mathcal{M}$ et $-1 \in \mathcal{M}$. Montrer aussi que $i \in \mathcal{M}$.

1.2 2. Premiers tracés

Oui, je sais, vous voulez tracer $\mathcal{M}$. Mais histoire de ne pas réécrire cinquante fois la même chose, mettons en place quelques outils.

1.2.1 2.1 Paramètres

Nous allons bientôt faire nos premiers tracés de $\mathcal{M}$. Enfin, nous devrions plutôt dire **approximations** de $\mathcal{M}$. Nos fonctions de tracé auront besoin d'un certains nombre de paramètres, alors plutôt que de passer des tas de paramètres à nos fonctions, définissons une classe `Parametre`. Le constructeur de cette classe prend un certain nombre de paramètres :

- `xc` et `yc` sont les coordonnées du centre de la figure que nous voulons tracer.
- `zoom` est un facteur de zoom. Quand `zoom` vaut 1, la figure est un carré de côté 3.
- `n` est le nombre de points sur chaque ligne et chaque colonne de la figure. Nos figures seront des *carrés* représentés par des matrices.
- `niter` est le nombre maximal d'itérations que nous ferons lors des appels à la fonction `iterer`.

Le constructeur calcule à partir de ces données les réels `xmin`, `xmax`, `ymin` et `ymax` qui sont les coordonnées minimales et maximales des points de la figure. C'est tout l'intérêt d'un constructeur : il automatise les tâches d'initialisation.

```
In [10]: class Parametre:

             def __init__(self, xc, yc, n, zoom, niter):
                 self.xc = xc
                 self.yc = yc
                 self.zoom = zoom
                 self.n = n
                 self.niter = niter
                 d = 3 / (2 * zoom)
                 self.xmin = xc - d
                 self.xmax = xc + d
                 self.ymin = yc - d
                 self.ymax = yc + d
```

Pour `zoom=1`, la fenêtre sera un carré de côté 3. Pourquoi 3 ? Parce que l'ensemble $\mathcal{M}$ tient dans le carré $[-2,1] \times [-\frac{3}{2},\frac{3}{2}]$ (preuve ?) qui est précisément de côté 3. Le centre de ce carré est $(-\frac{1}{2},0)$.

```
In [11]: param = Parametre(xc=-0.5, yc=0, n=300, zoom=1, niter=64)
         print(param.xmin, param.xmax)
         print(param.ymin, param.ymax)
```

```
-2.0 1.0
-1.5 1.5
```

1.2.2 2.2 Conversion de coordonnées

Une *image* de $\mathcal{M}$ (ce que nous voulons afficher à l'écran) est consituée de pixels à coordonnées entières. Les points de $\mathcal{M}$, eux, sont des nombres complexes dont les parties réelle et imaginaire sont des nombres réels. La fonction `point` prend en paramètre un couple (i,j), coordonnées d'un pixel d'une image, et renvoie le nombre complexe $x+iy$ correspondant dans le monde réel $\mathbb{C}$. Bien entendu, le second paramètre `param` qui est un objet de la classe `Parametre` permet de faire la conversion.

Notre image sera bentôt stockée dans une matrice et i et j seront les numéros respectifs de ligne et de colonne du pixel dans la matrice. Ainsi, i est une *ordonnée*, qui va "du haut vers le bas" et j est une abscisse qui va "de gauche à droite". Comprenez bien la fonction `point`, elle est la clé d'un grand nombre d'algorithmes graphiques !

```
In [12]: def point(p, param):
             i, j = p
             x = param.xmin + j * (param.xmax - param.xmin) / param.n
             y = param.ymax + i * (param.ymin - param.ymax) / param.n
             return x + 1j * y
```

Voici le point au centre de l'image pour la valeur de `param` définie plus haut.

```
In [13]: point((150, 150), param)
```

```
Out[13]: (-0.5+0j)
```

Exercice important : Écrivez une fonction `pixel`, réciproque de la fonction `point`.

1.2.3 2.3 Calcul de l'image

"Calculons" $\mathcal{M}$. Lisez la fonction `mandel` ci-dessous, elle ne présente aucune difficulté : elle remplit une matrice carrée à l'aide de deux boucles imbriquées puis renvoie cette matrice. Comme nous avons écrit proprement la fonction `point`, il n'y a plus aucun souci.

```
In [14]: def matrice(n):
             m = n * [None]
             for i in range(n):
                 m[i] = n * [0]
             return m

In [15]: def mandel(param):
             m = matrice(param.n)
             for i in range(param.n):
                 for j in range(param.n):
                     c = point((i, j), param)
                     k = iterer(c, param.niter)
                     if k == param.niter: m[i][j] = 2
                     else: m[i][j] = k % 2
             return m
```

Affichons $\mathcal{M}$ dans sa totalité.

```
In [16]: param = Parametre(xc=-0.5, yc=0, n=20, zoom=1, niter=64)
         print(mandel(param))
```

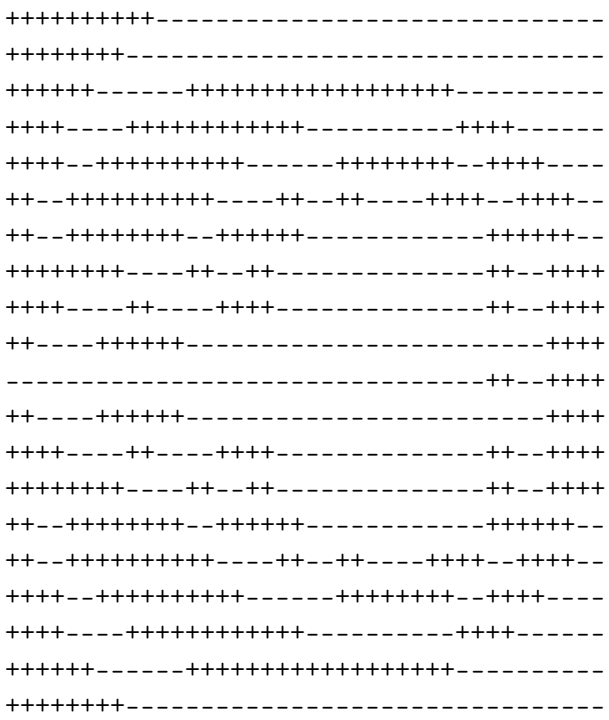
```
[[0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1,
```

Euh, oui, bof. Allez, nous y sommes presque. Mais d'abord une blague d'informaticien. Eh oui, il fut un temps où les écrans ne pouvaient afficher que du texte, alors plongeons-nous un court instant à la fin des années 70 :-).

1.2.4 2.4 Juste pour rire

```
In [17]: def mandel_geek(m, param):
             s = ""
             n = len(m)
             for i in range(n):
                 for j in range(n):
                     if m[i][j] == 0: s += '++'
                     else: s += '--'
                 s += '\n'
             return s

In [18]: param = Parametre(xc=-0.5, yc=0, n=20, zoom=1, niter=32)
         m = mandel(param)
         print(mandel_geek(m, param))
```



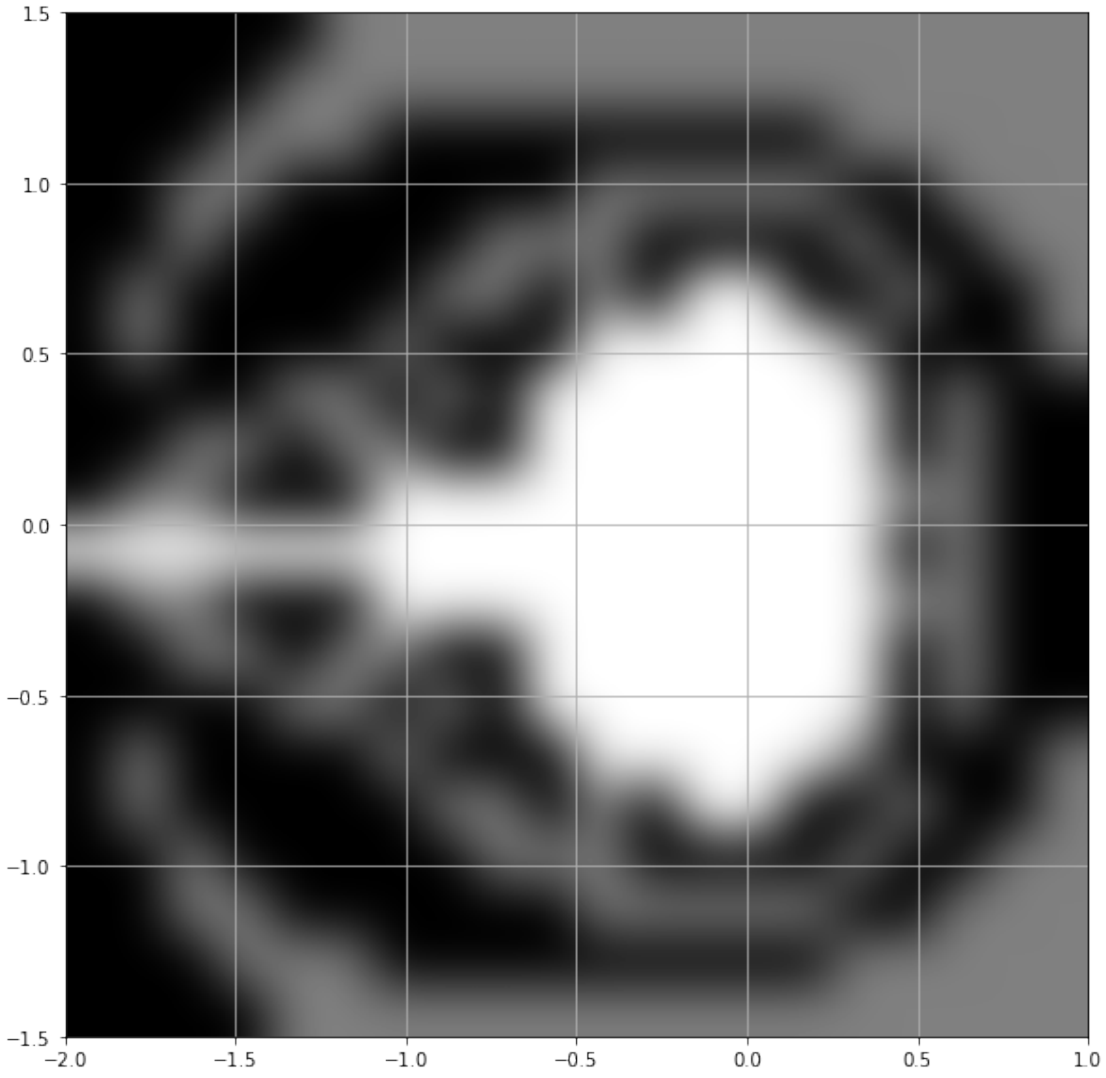
Retour au XXIème siècle ?

1.2.5 2.5 Tracé

```
In [19]: def show_mandel(m, param):
             plt.imshow(m, interpolation='bicubic', cmap='gray', extent=(param.xmin, param.xmax,
             plt.grid()

In [20]: def run_mandel(param):
             m = mandel(param)
             show_mandel(m, param)

In [21]: param = Parametre(-0.5, 0, 20, 1, 32)
         run_mandel(param)
```

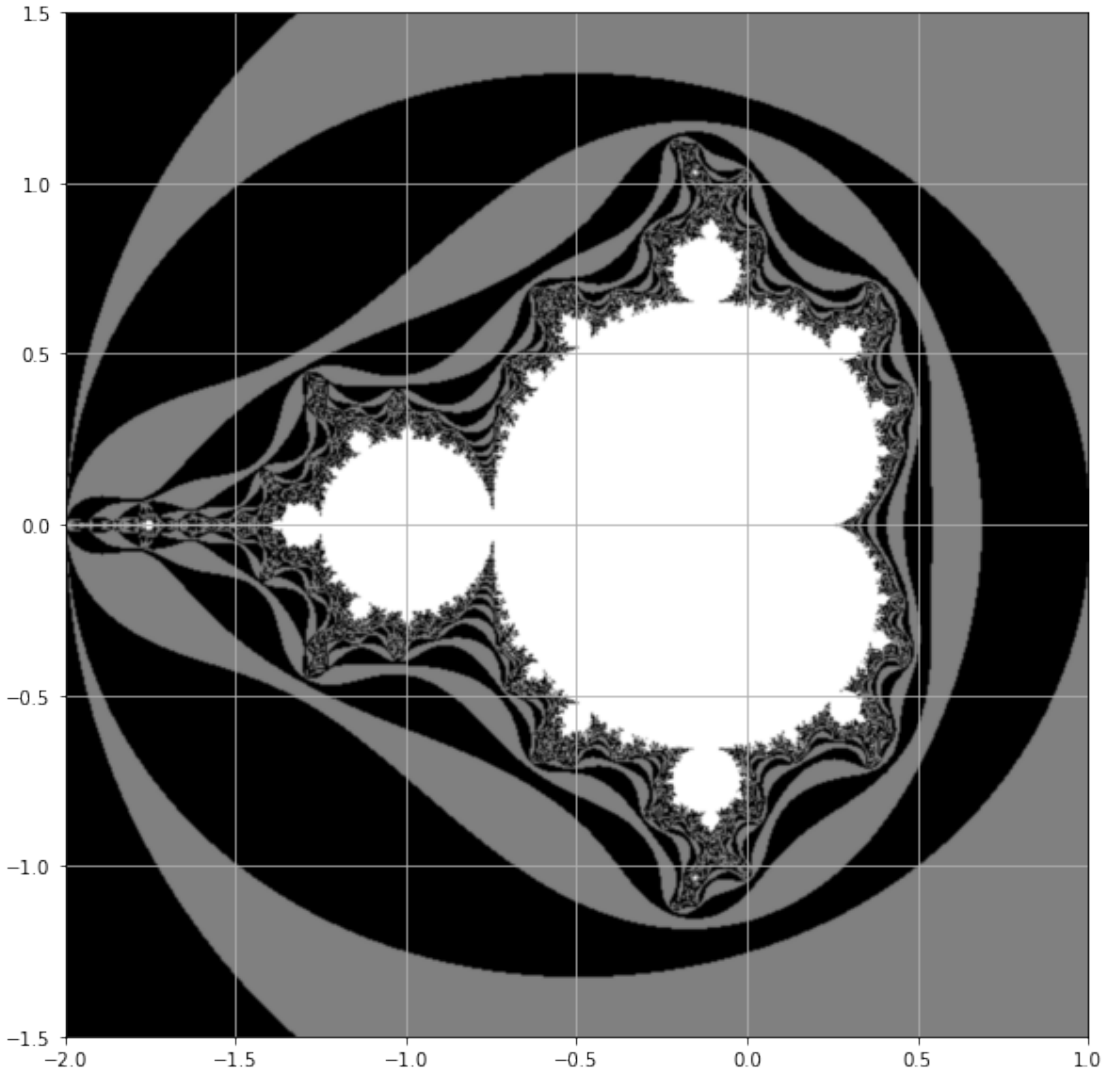


Une image de taille 20×20 c'est très peu question résolution. On devine la forme de $\mathcal{M}$ et encore, c'est grâce aux algorithmes magiques d'interpolation de `pyplot.imshow`. Augmentons donc la taille de l'image.

Avertissement : Si nous traçons $\mathcal{M}$ dans une image de taille $n \times n$ il nous faut calculer n^2 pixels, chaque pixel nécessitant une boucle pouvant faire jusqu'à `niter` itérations. Par exemple, si vous prenez $n =$ `niter` $= 1000$ il y aura des milliards d'opérations effectuées. Python n'est pas fort en boucles, alors un conseil : commencez par de petites valeurs de n et `niter` juste pour voir que tout va bien. Puis mettez n à une valeur plus grande, `niter` aussi si nécessaire.

Lançons avec $n = 600$: 5 secondes sur ma machine.

```
In [22]: param = Parametre(-0.5, 0, 600, 1, 64)
         run_mandel(param)
```



1.2.6 2.6 Que voyons-nous ?

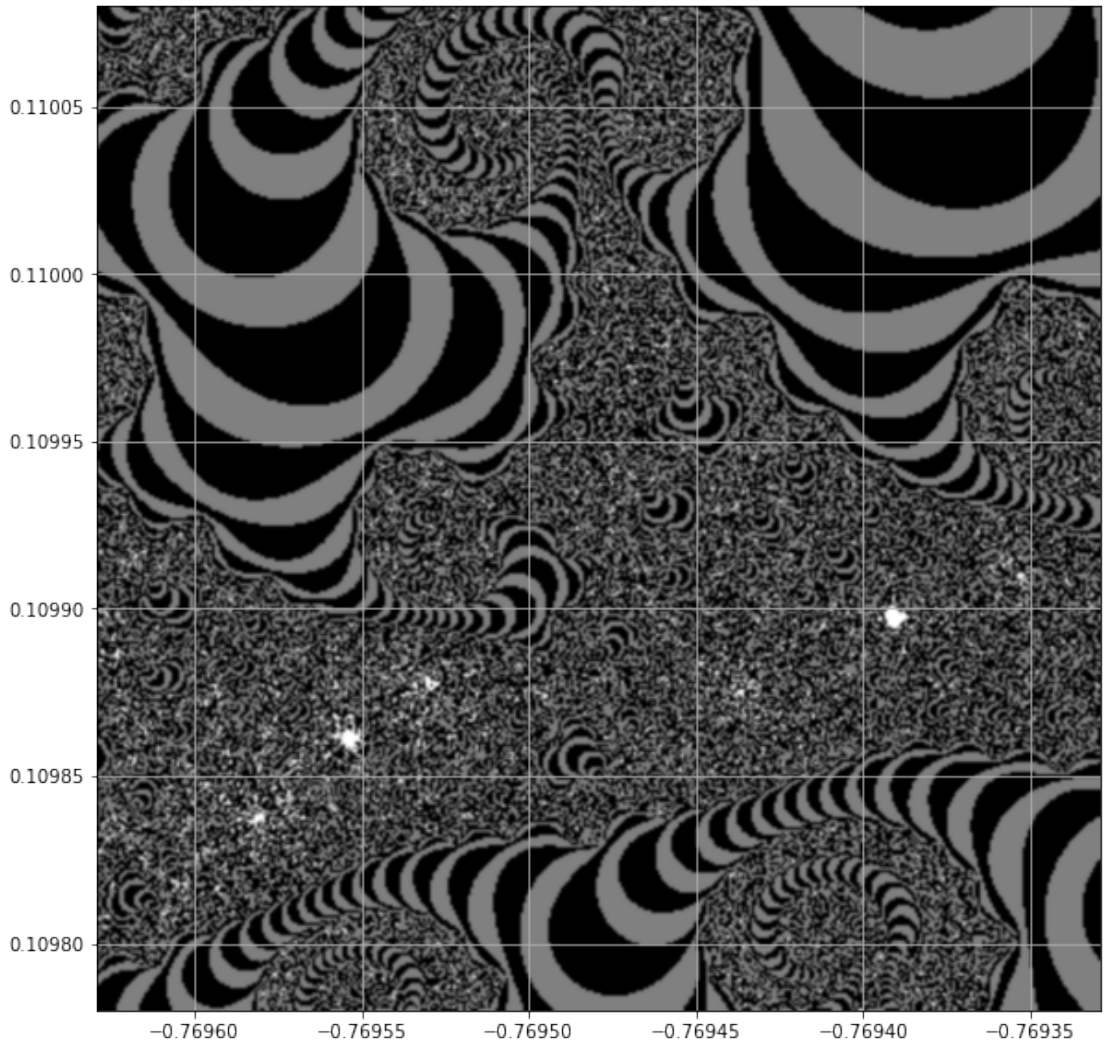
Ceci est une question importante. Nous voyons des points colorés, certains sont blancs, d'autres sont noirs ou gris. Que garantit notre algorithme exactement ? Il nous assure que les points qui ne sont pas blancs ne sont pas dans $\mathcal{M}$. Quant aux points blancs, ils sont dans $\mathcal{M}$ ... ou pas.

Que penser de l'apparent désordre graphique près de la zone blanche ? Mathématiquement, rien pour l'instant. Mais notre bon sens nous dit que si nous augmentons la résolution de l'image (la valeur de `param.n`), des points blancs pour l'instant invisibles apparaîtront pas très loin.

Alors disons que ce que nous voyons est une approximation artistique de $\mathcal{M}$. Question marketing c'est vendeur, mais il va nous falloir faire mieux que cela :-).

Allons, faisons un zoom.

```
In [23]: param = Parametre(-0.769479, 0.10993, 400, 10000, 512)
         run_mandel(param)
```



Que voit-on ?

- Une espèce de voie lactée vers le bas de la figure avec quelques petites galaxies blanches. Il y en a une par exemple pas loin des coordonnées $(-0.76938, 0.10990)$.
- Des spirales grises.

Nous sommes dans une zone où l'ensemble $\mathcal{M}$ doit être terriblement fin. Conséquence, on ne voit pas grand chose. Il va falloir faire infiniment mieux que cela. Patience, ce sera le cas ...

Exercice : Zoomez autour de la galaxie dont je viens de parler avec un facteur de zoom de 10^5 et un nombre d'itérations de 1024. Vous resonnaîtrez sans peine la forme de celle-ci et serez d'accord avec mon bilan :

Bilan : L'ensemble $\mathcal{M}$ est un ensemble complexe, quoique bien réel :-).

Partant de l'image de $\mathcal{M}$ tout entier, libre à vous maintenant de faire des zooms pour explorer différentes zones de l'ensemble. Quelles sont les zones intéressantes ? Aaaah, mais là est toute la question ... $\mathcal{M}$ est la carte d'un monde avec une infinité de lieux différents à visiter.

1.3 3. La cardioïde

1.3.1 3.1 C'est quoi ?

Regardez le gros machin en forme de coeur qui occupe la grande partie centrale de l'ensemble de Mandelbrot. C'est une **cardioïde**. Précisément c'est l'ensemble des $z \in \mathbb{C}$ de la forme

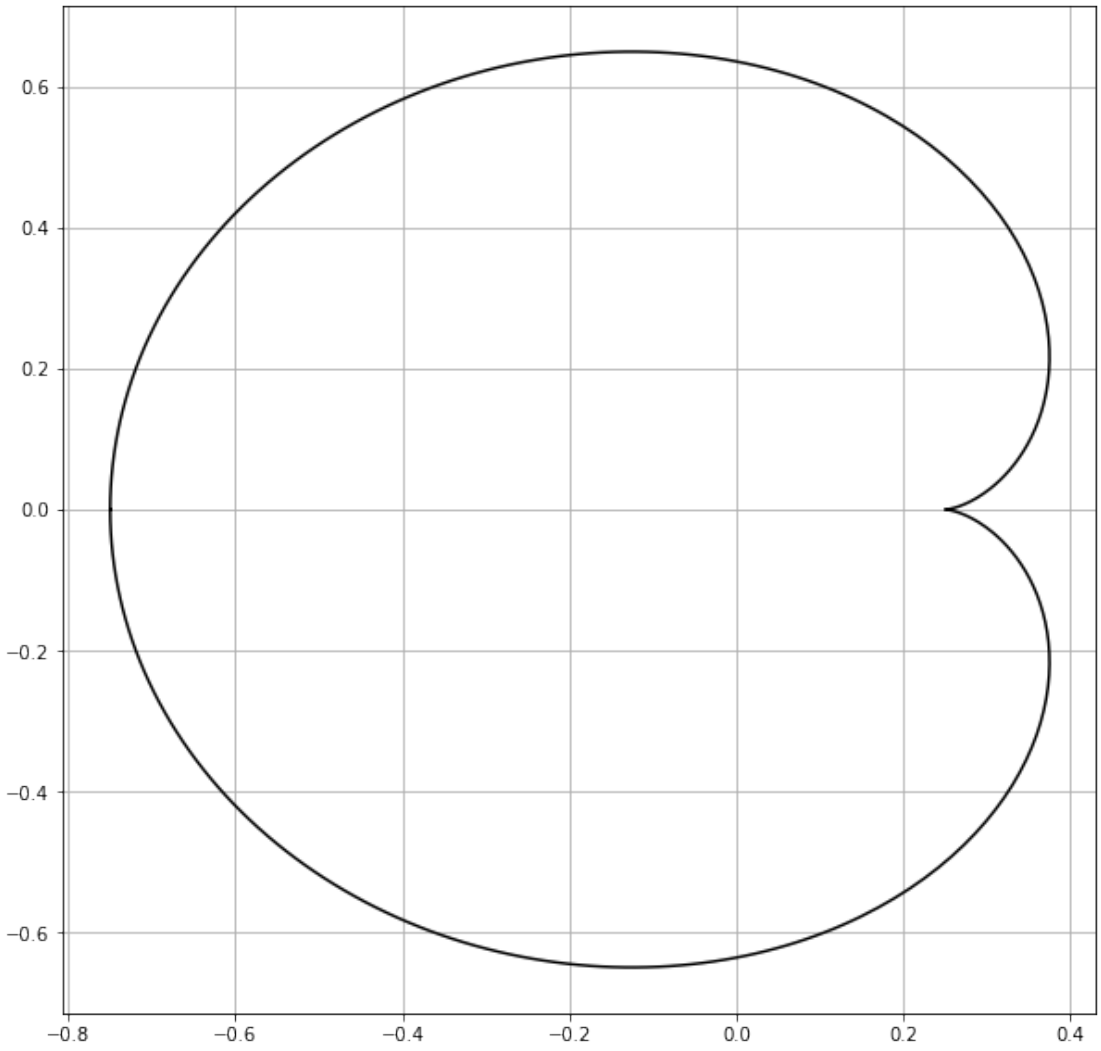
$$z = \frac{1}{4} + r(1 - \cos\theta)e^{i\theta}$$

où $0 \leq r \leq 1$ et $-\pi \leq \theta \leq \pi$.

Pour $r = 1$ on obtient le bord de la cardioïde : c'est l'endroit dans $\mathcal{M}$ où s'accrochent tout un tas de sortes de ronds ... Avant de vous dire en quoi cette cardioïde est importante, traçons-la.

```
In [24]: def plot_cardioid():
             ts = [k * math.pi / 200 for k in range(-200, 201)]
             xs = [1/4 + 1 / 2 * (1 - math.cos(t)) * math.cos(t) for t in ts]
             ys = [1 / 2 * (1 - math.cos(t)) * math.sin(t) for t in ts]
             plt.plot(xs, ys, 'k')

In [25]: plot_cardioid()
         plt.grid()
```



Ah oui, c'est très ressemblant.

1.3.2 3.2 Orbite des points dans la cardioïde

L'orbite du point c est la suite $(c, f(c), f(f(c)), \ldots)$, bref, la suite (z_n). La fonction `plot_orbit` prend en paramètres deux flottants x et y correspondant à $c = x + iy \in \mathbb{C}$ et un entier n. Elle trace ensuite les points $z_0, z_1, \ldots, z_{n-1}$.

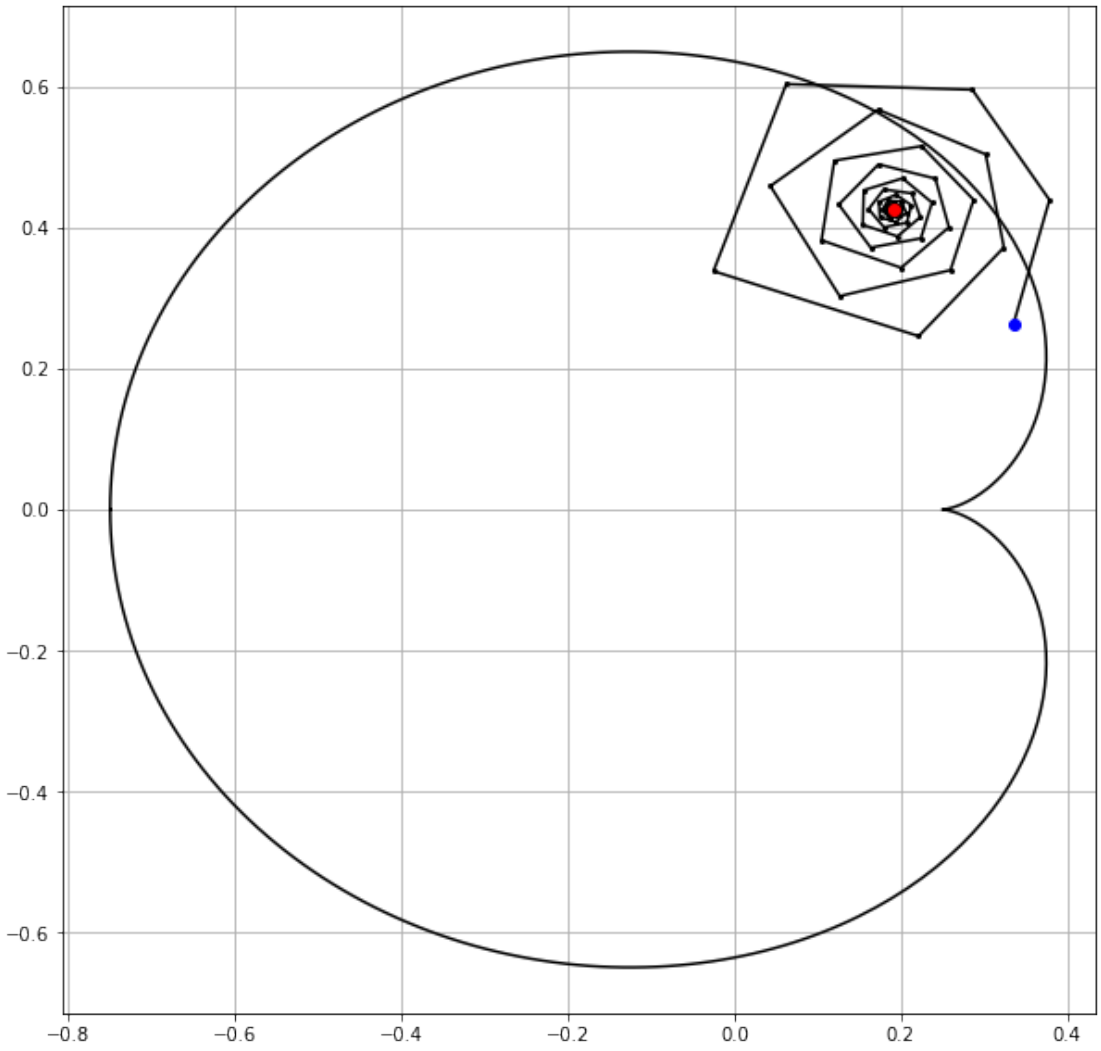
La fonction marque en bleu le point c et en rouge le dernier point calculé. Si la suite (z_n) converge, le point rouge est censé être proche de la limite.

```
In [26]: def plot_orbit(c, n):
             xs = []
             ys = []
             z = 0
             k = 0
             while k <= n and abs(z) <= 2:
```

```
            z = z * z + c
            k = k + 1
            xs.append(z.real)
            ys.append(z.imag)
        plt.plot(xs, ys, 'ok', markersize=2)
        plt.plot(xs, ys, 'k')
        plt.plot([x], [y], 'ob')
        plt.plot([xs[-1]], [ys[-1]], 'or')
```

Prenons un point dans la cardioïde et traçons son orbite. Évaluez la cellule ci-dessous autant que vous le voudrez, en changeant les valeurs de $r \in [0, 1[$ et $t \in [-\pi, \pi]$. Essayez des points dans un peu toutes les régions possibles de la cardioïde pour voir ce que devient l'orbite du point.

In [27]:
```
t = 0.4 * math.pi
r = 0.8
x = 1/4 + 1/2 * r * (1 - math.cos(t)) * math.cos(t)
y = 1/2 * r * (1 - math.cos(t)) * math.sin(t)
plot_orbit(x + 1j * y, 100)
plot_cardioid()
plt.grid()
```



1.3.3 3.3 Convergence

Voyez-vous où nous voulons en venir ?

Proposition : Soit $c \in \mathbb{C}$. - Si c est à l'intérieur de la cardioïde (bord exclus) la suite (z_n) est convergente. - Si c est à l'extérieur de la cardioïde, la suite est divergente.

Je ne montrerai pas ce résultat. Nous allons simplement essayer de nous convaincre. Si la suite (z_n) converge vers $\ell \in \mathbb{C}$, alors $\ell = \ell^2 + c$, ce qui nous donne en résolvant l'équation $2l = 1 \pm \delta$ où $\delta^2 = 1 - 4c$.

Commençons par regarder le cas où $c \notin]\frac{1}{4}, +\infty[$, (on a donc $\theta \neq 0$).

Nous pouvons alors $c = \frac{1}{4} + \frac{1}{2}r(1 - \cos\theta)e^{i\theta}$. Si $0 \leq r < 1$, on est à l'intérieur de la cardioïde. Si $r > 1$ on est à l'extérieur. Et si $r = 1$ on est sur le bord. On a

$$1 - 4c = -2r(1 - \cos\theta)e^{i\theta} = -4r\sin^2\frac{\theta}{2}e^{i\theta}$$

Une racine carrée de $1 - 4c$ est donc

$$\delta = 2i\sqrt{r}\sin\frac{\theta}{2}e^{i\theta/2}$$

et

$$2\ell = 1 \pm 2i\sqrt{r}\sin\frac{\theta}{2}e^{i\theta/2}$$

Calculons $|2\ell|^2$.

$$|2\ell|^2 = 4\ell\overline{\ell} = (1 + 2i\varepsilon\sqrt{r}\sin\frac{\theta}{2}e^{i\theta/2})(1 - 2i\varepsilon\sqrt{r}\sin\frac{\theta}{2}e^{-i\theta/2})$$

En développant, on obtient

$$|2\ell|^2 = 1 - 4\varepsilon\sqrt{r}\sin^2\frac{\theta}{2} + 4r\sin^2\frac{\theta}{2}$$

d'où

$$|2\ell|^2 - 1 = 4\sqrt{r}\sin^2\frac{\theta}{2}(\sqrt{r} - \varepsilon)$$

Rappelons-nous que $f_c(z) = z^2 + c$. Conséquence du calcul précédent :

- Si $r > 1$, $|f_c'(\ell)| > 1$.
- Si $r = 1$, $|f_c'(\ell)| = 1$ pour $\varepsilon = 1$ et $|f_c'(\ell)| > 1$ pour $\varepsilon = -1$.
- Si $0 \leq r < 1$, $|f_c'(\ell)| < 1$ pour $\varepsilon = 1$ et $|f_c'(\ell)| > 1$ pour $\varepsilon = -1$.

Proposition :

- Un point ℓ tel que $|f_c'(\ell)| > 1$ est **répulsif** : la suite ne peut pas converger vers ℓ, sauf si l'un de ses termes est exactement égal à ℓ.
- Un point ℓ tel que $|f_c'(\ell)| < 1$ est **attractif** : la suite converge vers ℓ, à condition que l'un de ses termes soit "suffisamment proche" de ℓ.
- Pour un point ℓ tel que $|f_c'(\ell)| = 1$ on ne peut rien dire sans une étude plus approfondie.

Je n'en dirai pas plus mais nous avons de bonnes raisons de penser que le théorème énoncé plus haut est plausible. Sa preuve complète est hors de la portée de ce notebook.

Exercice : Traitez le cas où $c \in]\frac{1}{4}, +\infty[$. Montrez qu'alors $|f_c'(\ell)| > 1$. Si vous êtes motivés, montrez que la suite (z_n) (qui est alors une suite réelle) tend vers $+\infty$, et donc que $c \notin \mathcal{M}$.

1.3.4 3.4 Points périodiques

Les points de $\mathcal{M}$ sont les nombres complexes c tels que la suite (z_n) associée soit bornée. Une suite peut être bornée de plusieurs façons :

- Elle peut converger. Nous venons de voir que les points c pour lesquels la suite converge sont ceux à l'intérieur de la cardioïde. Un point c pour lequel se produit ce phénomène, nous l'appellerons un point de période 1.
- Elle peut "converger de 2 en 2", c'est à dire que les suites (z_{2n}) et (z_{2n+1}) convergent, mais vers des limites différentes. Un point c pour lequel se produit ce phénomène, nous l'appellerons un point "ultimement de période 2".

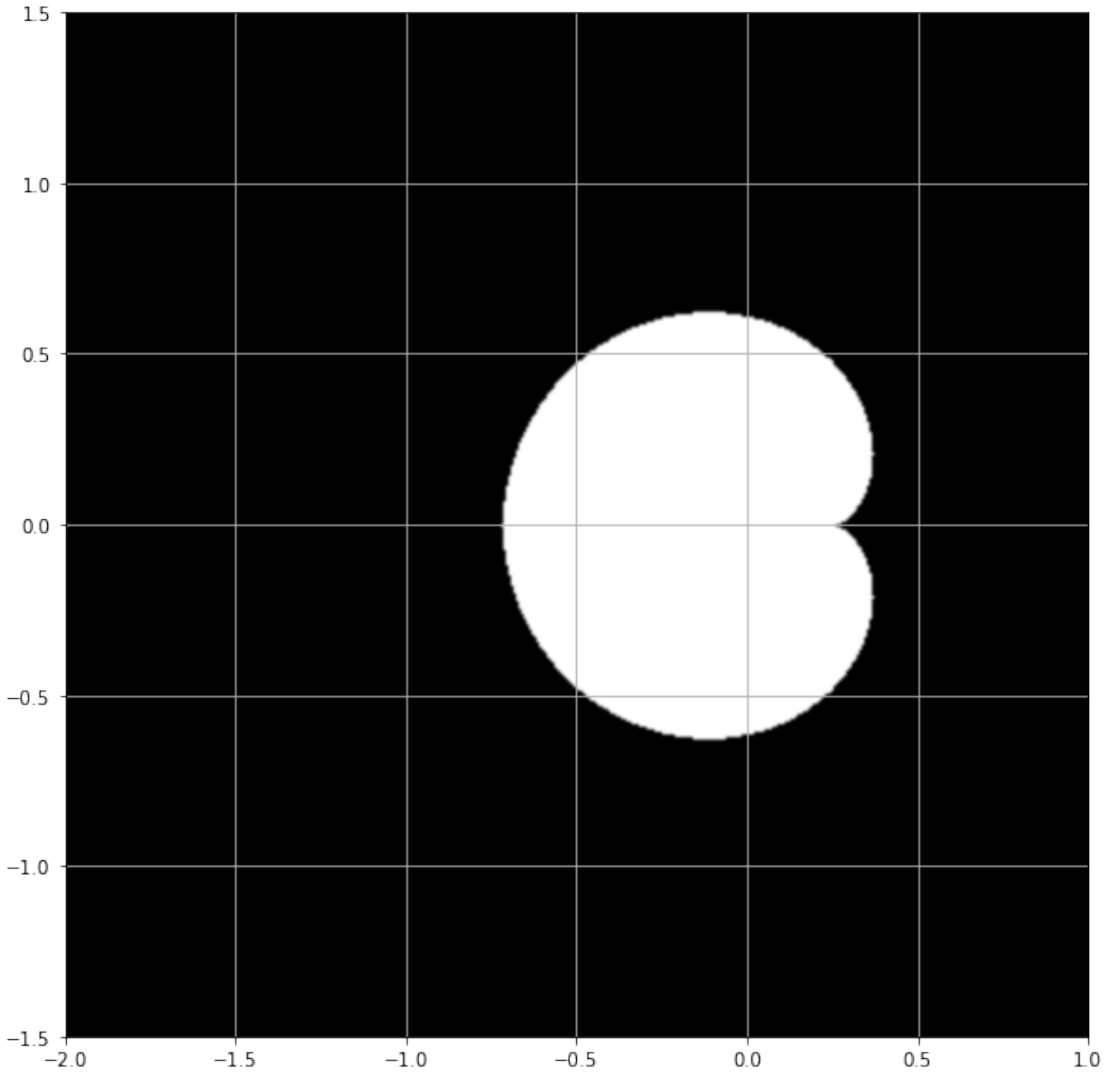
- Elle peut "converger de 3 en 3", c'est à dire que les suites (z_{3n}), (z_{3n+1}) et (z_{3n+2}) convergent, mais vers des limites différentes. Un point c pour lequel se produit ce phénomène, nous l'appellerons un point "ultimement de période 3".
- Etc.
- Elle peut faire aussi des choses très compliquées, dont nous ne parlerons pas ici.

La fonction `show_period` prend en paramètre un entier $T > 0$ et affiche les points de $\mathcal{M}$ dont la période **divise** T. Cette fonction n'est pas très "solide" : pour $c \in \mathbb{C}$ donné, on calcule z_n pour un certain n, puis z_{n+T}. Si $z_n \simeq z_{n+T}$ on allume le pixel correspondant.

```
In [28]: def show_period(T, param):
             tol = 1e-2
             m = matrice(param.n)
             for i in range(param.n):
                 for j in range(param.n):
                     c = point((i, j), param)
                     z = c
                     for k in range(param.niter): z = z * z + c
                     w = z
                     for k in range(T): w = w * w + c
                     if abs(w - z) < tol: m[i][j] = 1
             show_mandel(m, param)
```

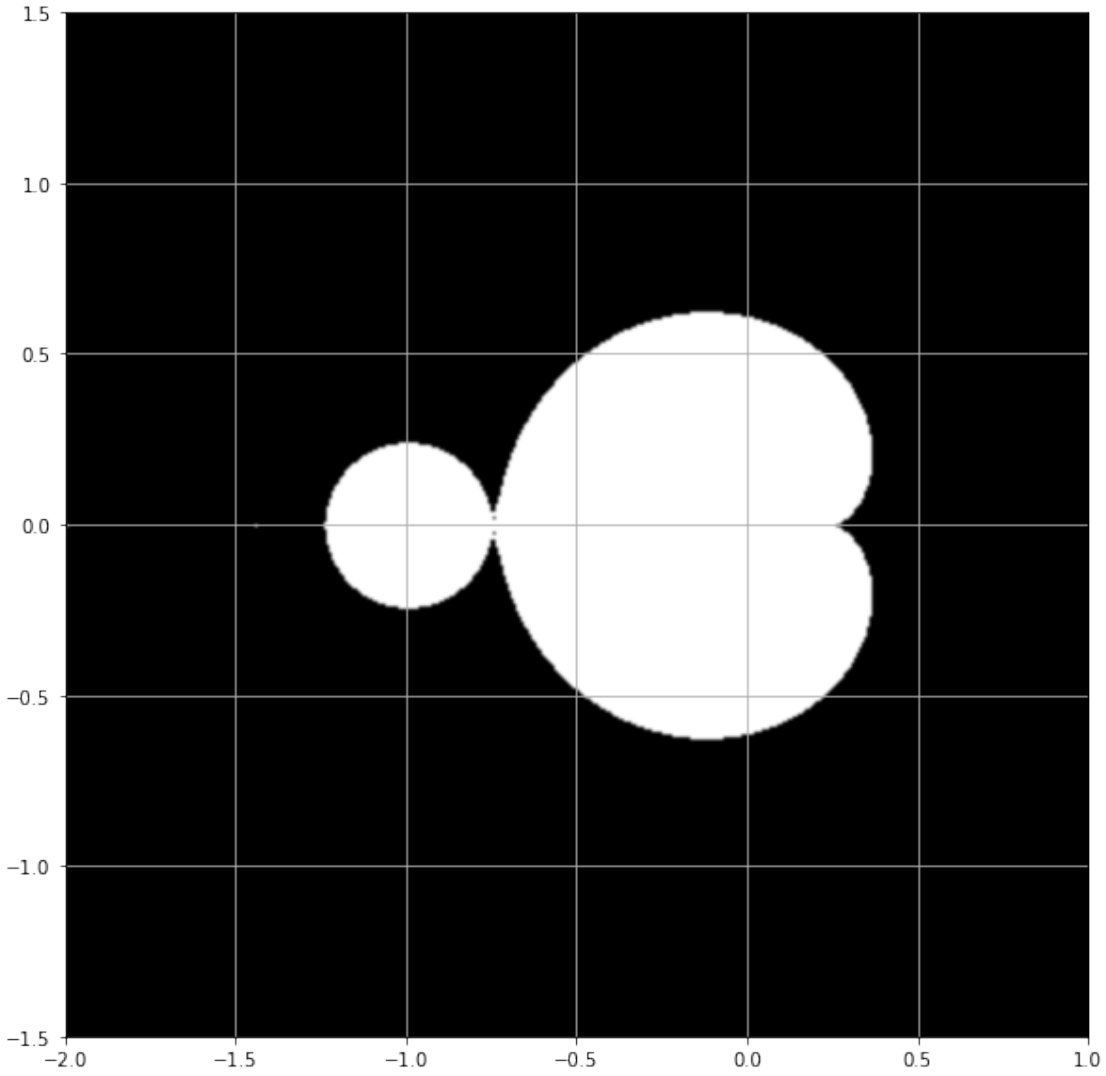
Avec le paramètre $T = 1$ nous devrions obtenir la cardioïde ...

```
In [29]: param = Parametre(-0.5, 0, 400, 1, 128)
         show_period(1, param)
```



Et pour $T = 2$?

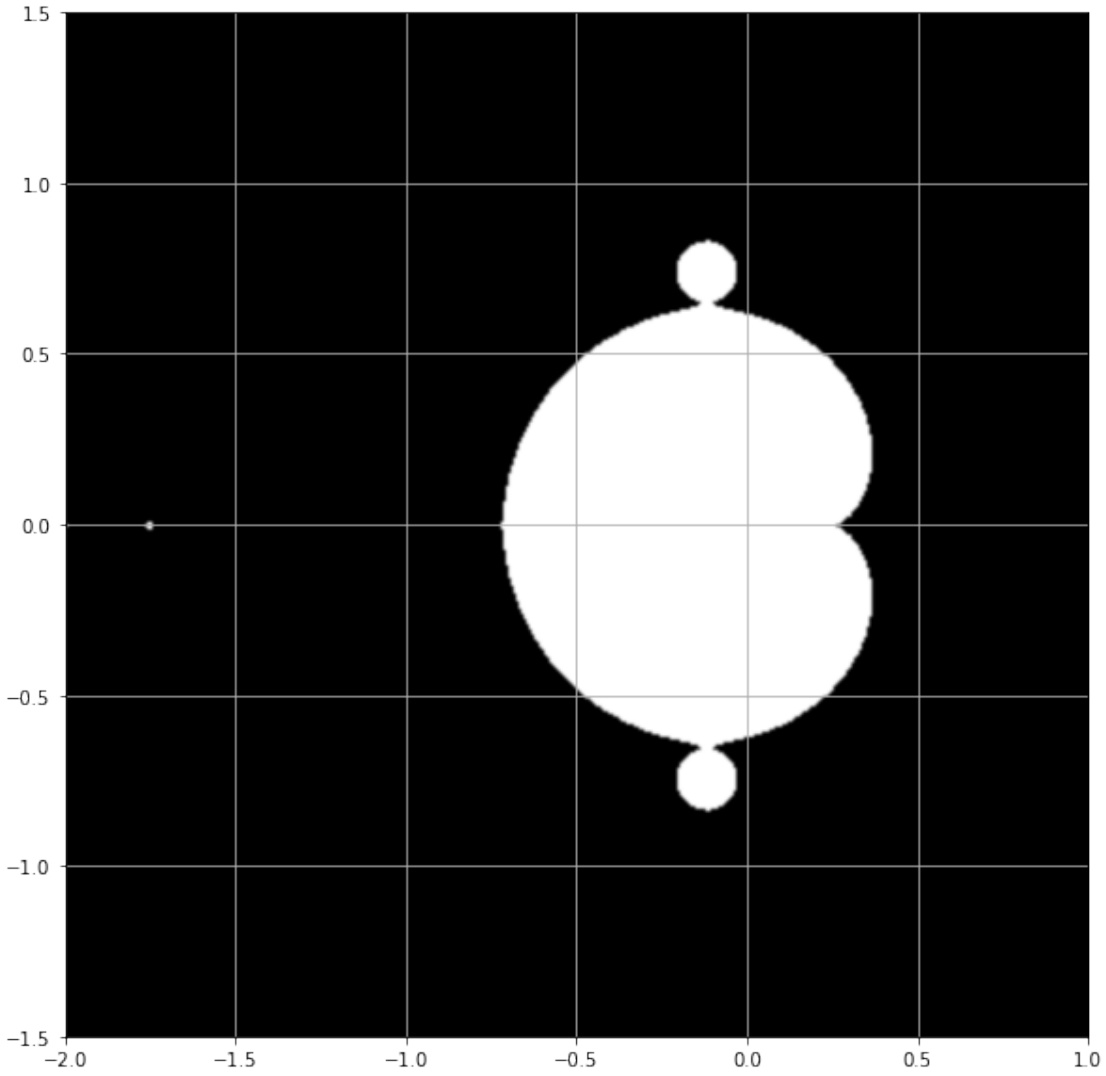
```
In [30]: param = Parametre(-0.5, 0, 400, 1, 128)
         show_period(2, param)
```



On retrouve évidemment la cardioïde puisque 1 divise 2. Et également une espèce de rond. En fait **c'est** un disque : le disque de centre -1 et de rayon $\frac{1}{4}$. Je ne le montrerai pas ici.

Et si $T = 3$?

```
In [31]: param = Parametre(-0.5, 0, 400, 1, 128)
         show_period(3, param)
```



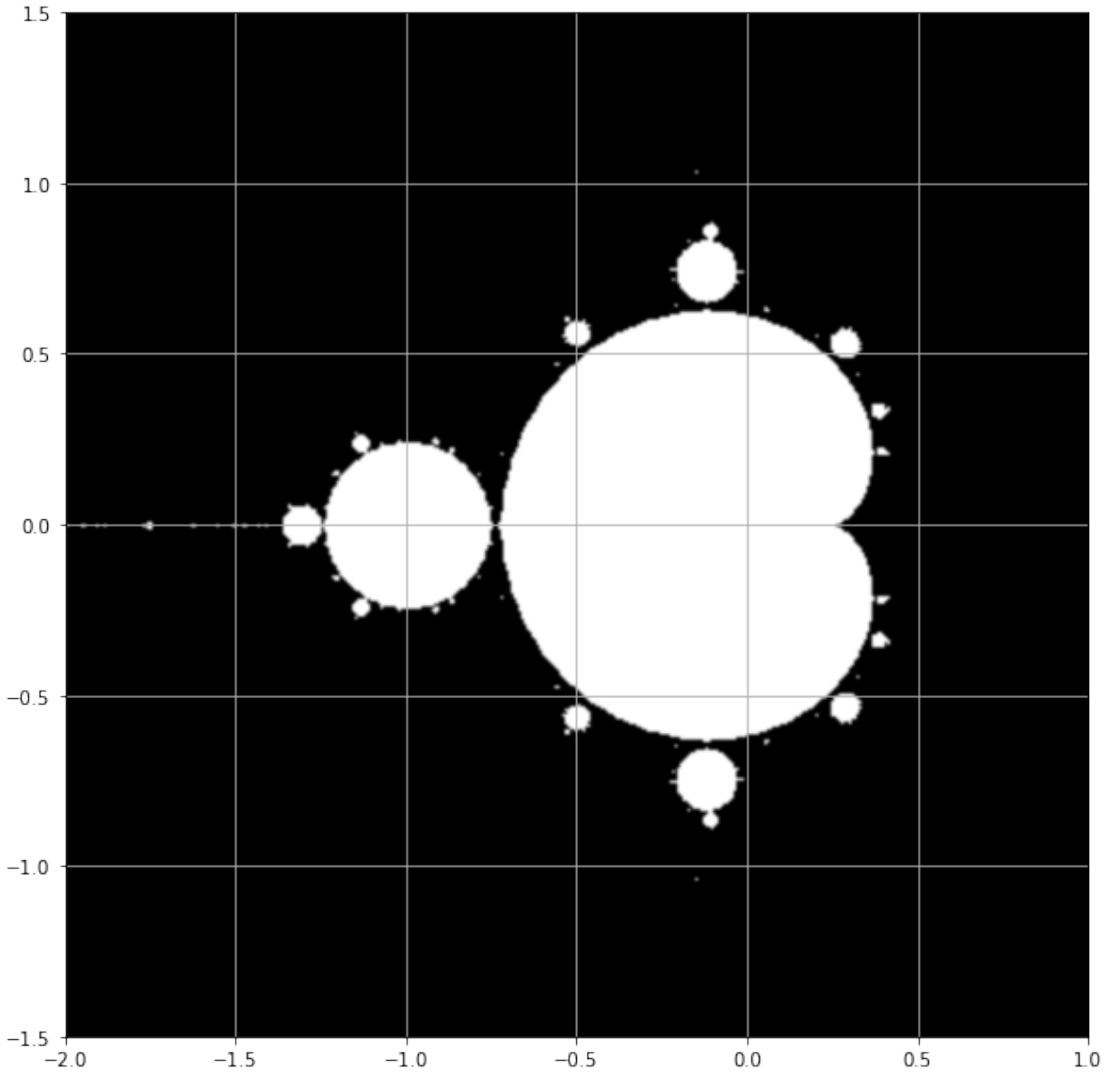
Deux nouveaux “ronds” apparaissent. Remarquez aussi, sur la gauche du graphique, le point allumé . . . la situation se complique.

Je vous laisse expérimenter avec quelques autres valeurs de T. Exemples intéressants :

- $T = 5$, $T = 7$, etc . . .
- Les puissances de 2.

Allez, un dernier exemple, $T = 60$. Nous devrions voir les points de périodes 1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30 et 60 (ai-je oublié des diviseurs de 60 ?).

```
In [32]: param = Parametre(-0.5, 0, 400, 1, 128)
         show_period(60, param)
```



1.4 5. Un estimateur de distance

1.4.1 5.1 Distance à une partie de $\mathbb{C}$

Soit $A \subset \mathbb{C}$ un ensemble non vide. Soit $c \in \mathbb{C}$. Considérons l'ensemble

$$E = \{|z - c|, z \in A\}$$

E est une partie de $\mathbb{R}$ non vide et minorée (par 0). L'ensemble E possède donc une borne inférieure, que l'on appelle la distance de c à A. Notons-la $d(c, A)$.

Si nous voulons dessiner un ensemble A très compliqué, avec des détails extrêmement fins, comme par exemple $\mathcal{M}$, une idée consiste à

- Se donner un réel $\delta > 0$.
- Considérer l'ensemble $A_\delta = \{c \in \mathbb{C}, d(c, A) \leq \delta\}$.

L'ensemble A_δ est appelé le δ-voisinage fermé de A. C'est en gros l'ensemble A "épaissi" de δ. Il est possible qu'un dessin de A_δ fournisse des détails de A qui nous étaient cachés. Le problème, bien entendu, est de pouvoir calculer la distance de c à A. Il s'avère que pour l'ensemble de Mandelbrot il existe un algorithme qui permet de le faire.

1.4.2 5.2 Le potentiel de Hubbard-Douady

Rappelons nous notre suite préférée : $z_0 = c$ et pour tout $n \geq 0$, $z_{n+1} = z_n^2 + c$. Pour tout $n \in \mathbb{N}$, $z_n = z_n(c)$ est un polynôme en c. Quelle est la dérivée z'_n de z_n par rapport à c ?

- $z'_0 = 1$
- Pour tout $n \geq 0$, $z'_{n+1} = 2z_n z'_n + 1$.

Imaginez un cylindre vertical infini dont la base est l'ensemble $\mathcal{M}$. Supposez que ce cylindre est chargé électriquement. Les mathématiciens Adrien Douady et John Hubbard ont montré que le champ électrique engendré par ce cylindre dérive d'un potentiel G. Nous avons assez longuement discuté de ce potentiel dans le notebook dédié aux ensembles de Julia, équipotentielles, lignes de champ, aussi n'y reviendrai-je pas ici. Mais ce que nous avions fait alors peut être refait ici. C'est d'ailleurs un excellent exercice de copier-coller :-).

Quelle est la valeur du **potentiel de Hubbard-Douady** en un point $c \in \mathbb{C}$? Cette valeur est donnée par

$$G(c) = \lim_{n\to\infty} \frac{1}{2^n} \lg |z_n|$$

où lg désigne le logarithme en base 2. En fait, le choix de la base du logarithme n'a pas d'importance car un potentiel est défini à constante additive près.

Pour tout $c \in \mathbb{C}$ extérieur à $\mathcal{M}$, une estimation de la distance de c à $\mathcal{M}$ est

$$d(c, \mathcal{M}) \simeq \frac{G(c)}{|G'(c)|}$$

La dérivée de G par rapport à c est, en admettant l'interversion possible de limtes et de dérivées :

$$G'(c) = \lim_{n\to\infty} \frac{1}{2^n} \frac{|z'_n|}{|z_n|}$$

Ainsi,

$$d(c, \mathcal{M}) \simeq \lim_{n\to\infty} \frac{|z_n|}{|z'_n|} \lg |z_n|$$

1.4.3 5.3 L'algorithme

L'idée est d'utiliser la même boucle que pour la fonction `iterer`. Mais à chaque itération on calcule non seulement z_k mais aussi z'_k. Il peut arriver au cours des itérations que $|z'_k|$ devienne "très grand". Cela signifie (admettons-le) que $d(c, \mathcal{M})$ est très petit, et donc que c est très près de $\mathcal{M}$. On décrète alors (à tort peut-être) que $d(c, \mathcal{M}) = 0$.

Autre changement important : au lieu de sortir de la boucle lorsque $|z_k| > 2$, on attend qe $|z_k|$ devienne "très grand" (un million ci-dessous), dans l'espoir que $\frac{\lg |z_k|}{2^k}$ soit très proche de $G(c)$.

```
In [33]: def distance(c, niter):
             z = c
             z1 = 1
             k = 0
             while k < niter and abs(z) <= 1e6 and abs(z1) <= 1e50:
                 z, z1 = z * z + c, 2 * z * z1 + 1
                 k = k + 1
             if k == niter or abs(z1) > 1e50: return 0
             else:
                 m1 = abs(z)
                 m2 = abs(z1)
                 return m1 * math.log(m1, 2) / m2
```

1.4.4 5.4 Mandelbrot, le retour

Voici une nouvelle classe `Parametre2`. Un objet de cette classe a les mêmes champs qu'un objet de la classe `Parametre`, plus un champ `delta`. une valeur par défaut de `delta` est calculée. Elle donne souvent de bons résultats, mais pas toujours ...

```
In [34]: class Parametre2(Parametre):

             def __init__(self, xc, yc, n, zoom, niter, delta=None):
                 Parametre.__init__(self, xc, yc, n, zoom, niter)
                 if delta != None: self.delta = delta
                 else:
                     self.delta = (self.xmax - self.xmin) / (15 * self.n)
```

Voici notre nouvelle fonction `iterer`. Nous l'appellerons `iterer2`. Cette fonction prend en paramètres un nombre complexe c et un objet `param` de la classe `Parametre2`. Notre fonction renvoie 1 si $d(c, \mathcal{M}) \leq \delta$, et 0 sinon, où $\delta =$ `param.delta`.

```
In [35]: def iterer2(c, param):
             if distance(c, param.niter) <= param.delta: return 0
             else: return 1
```

Les fonctions qui suivent sont quasiment identiques à celles que nous avions déjà créées plus haut.

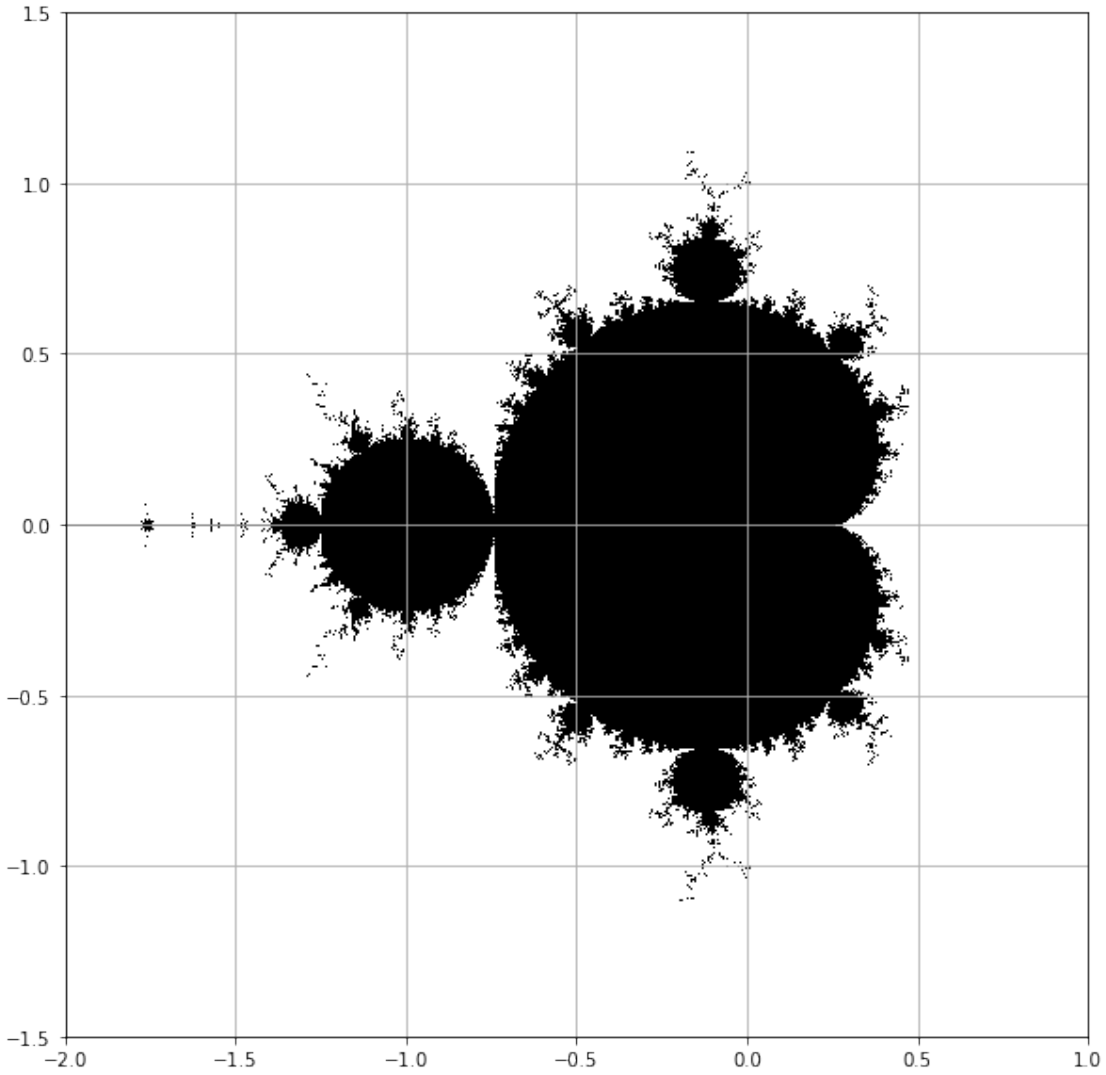
```
In [36]: def mandel2(param):
             m = matrice(param.n)
             for i in range(param.n):
                 for j in range(param.n):
                     c = point((i, j), param)
                     k = iterer2(c, param)
                     m[i][j] = k
             return m
```

```
In [37]: def show_mandel2(m, param):
             plt.imshow(m, cmap='gray', interpolation='none', extent=(param.xmin, param.xmax, pa
             plt.grid()
```

```
In [38]: def run_mandel2(param):
             m = mandel2(param)
             show_mandel2(m, param)
```

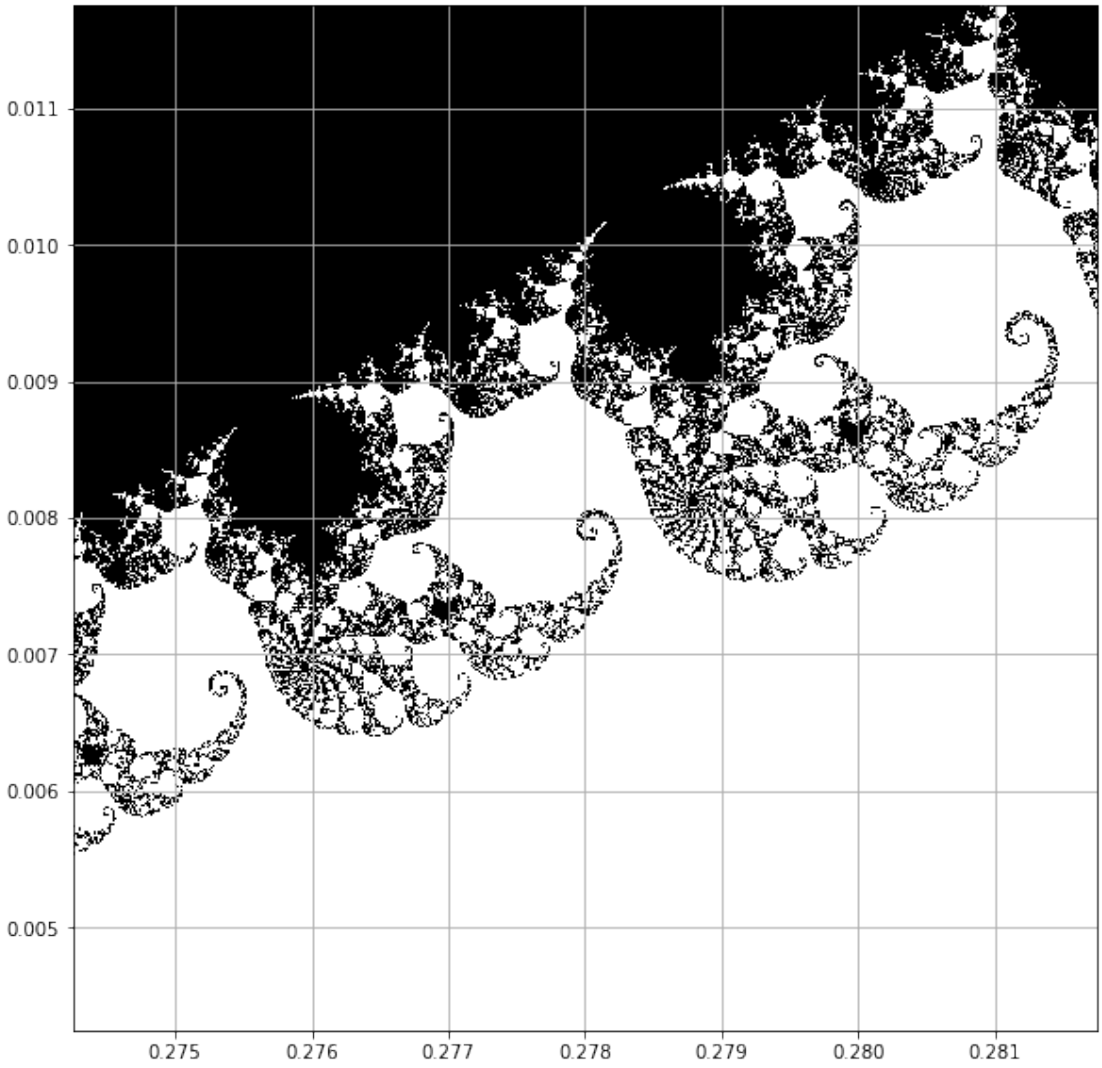
Voici tout d'abord l'ensemble $\mathcal{M}$ tout entier.

```
In [39]: param = Parametre2(xc=-0.5, yc=0, n=600, zoom=1, niter=256)
         run_mandel2(param)
```



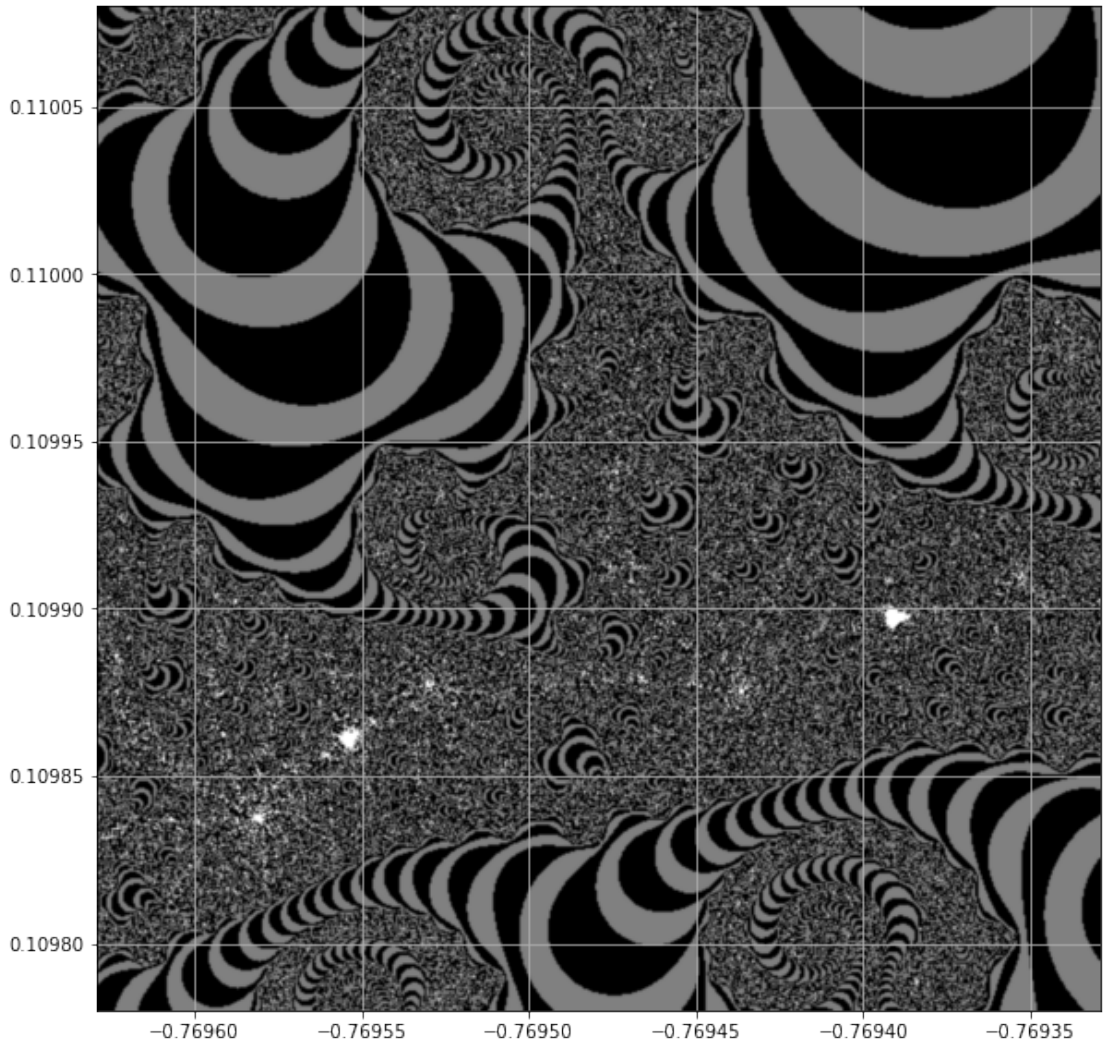
Et un zoom pas loin de $c = \frac{1}{4}$.

```
In [40]: param = Parametre2(xc=0.278, yc=0.008, n=600, zoom=400, niter=256)
         run_mandel2(param)
```



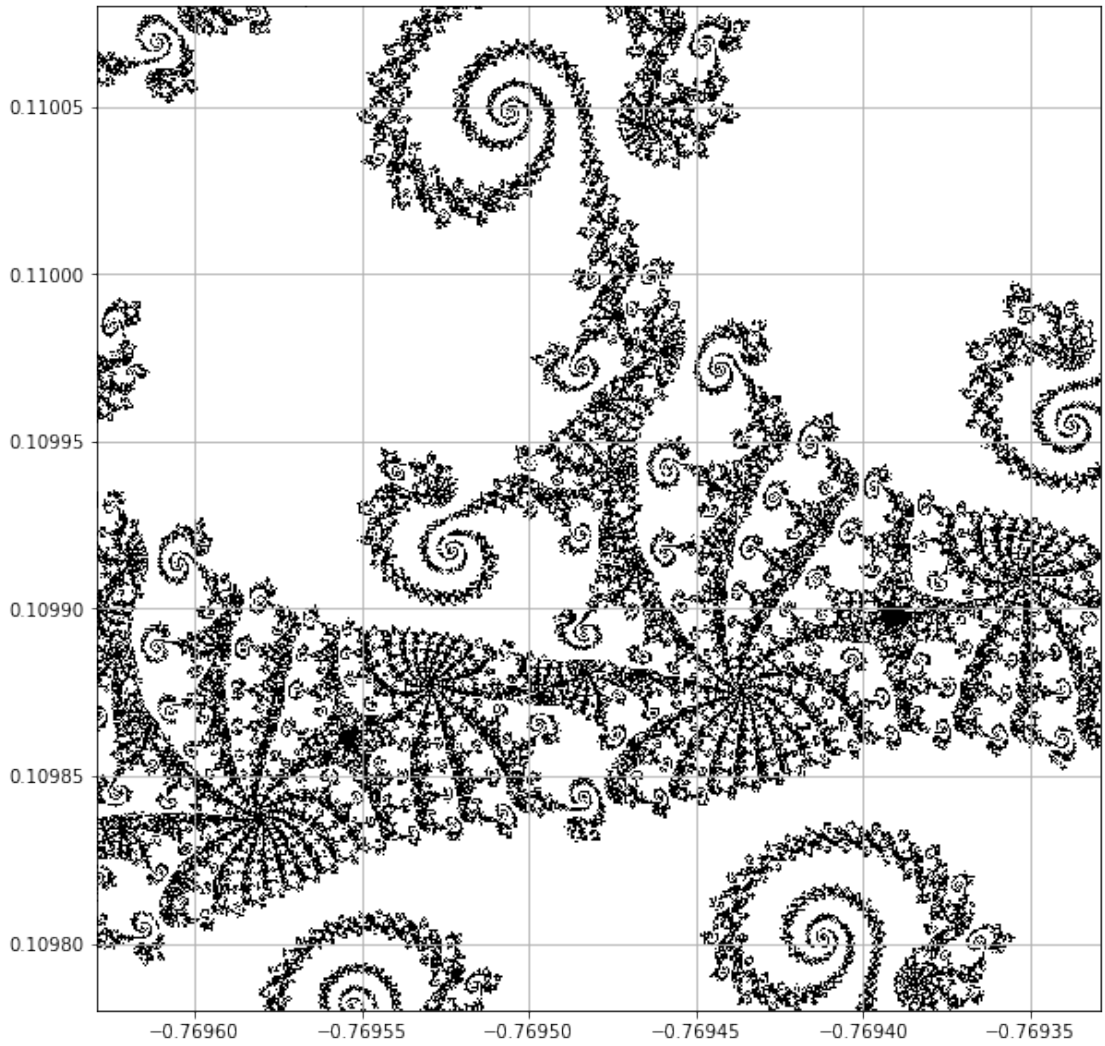
Voici l'amélioration promise à la fin du paragraphe 2. Soyez patients, le calcul prend un certain temps mais nous mesurerons les progrès accomplis par nos algorithmes en ce qui concerne la **résolution** des images. Rappelez-vous tout d'abord l'image renvoyée par notre premier algorithme :

```
In [41]: param = Parametre(-0.769479, 0.10993, 600, 10000, 512)
         run_mandel(param)
```



Et maintenant, voici ce que l'on obtient avec l'estimateur de distance. Soyez patients, environ une minute sur ma machine.

```
In [42]: param = Parametre2(-0.769479, 0.10993, 600, 10000, 512)
         run_mandel2(param)
```



On peut faire encore mieux mais nous pouvons déjà être très fiers :-).

1.5 6. Et après ?

Nous n'avons fait ici qu'effleurer un vaste sujet. Les mathématiques de l'ensemble $\mathcal{M}$ sont très sophistiquées, elles font appel à des résultats non triviaux de la théorie des fonctions de variable complexe. Je joins à ce notebook un programme Java permettant de faire quelques expériences sur $\mathcal{M}$. Ce programme utilise l'estimateur de distance dont nous avons parlé pour afficher l'ensemble de Mandelbrot.

Il est possible de faire des zooms avant et arrière sur l'ensemble à l'aide de la souris : maintenez le bouton de la souris appuyé et déplacez la souris pour créer un cadre pour un zoom avant. Clic droit pour un zoom arrière. La fenêtre `stat` affiche le nombre d'opréations effectuées depuis le lancement du programme.

Sont également affichés :

- L'ensemble de Julia K_c où c est le point correspondant au pointeur de la souris. Un appui sur

la touche Shift permet de déplacer le pointeur vers l'ensemble de Julia et de faire des zooms sur celui-ci.

- L'orbite du point c.

Si vous êtes béni de Thoth, le dieu des scribes et de la sagesse, un double clic sur l'icone `mandelbrot.jar` lance le programme. Si cela ne fonctionne pas, ouvrez un terminal et tapez, en vous plaçant dans le bon dossier, `java -jar mandelbrot.jar`. Si cela ne fonctionne toujours pas, tant pis pour vous :-). Amusez-vous bien !

In [ ]: