# Les ensembles de Julia

Marc Lorenzi

14 mars 2019

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

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

Dans son **Mémoire sur l'itération des fonctions rationnelles** (1918), le mathématicien Gaston Julia étudie le comportement de certaines suites récurrentes de nombres complexes. Il associe à ces suites des ensembles dont le nom adopté par les mathématiciens lui rend hommage : les **ensembles de Julia**.

Le masque en cuir que porte Gaston Julia sur la photo ci-dessus est dû à de graves blessures qu'il a subies au visage en 1915 pendant la première guerre mondiale.

Nous allons faire un petit voyage dans le monde merveilleux des ensembles de Julia. Quelques propriétés "faciles" de ces ensembles seront prouvées, d'autres, beaucoup plus difficiles à montrer, seront simplement énoncées. Et certaines, encore plus difficiles, ne seront même pas abordées.

La théorie des ensembles de Julia est extrêmement vaste et j'ai dû faire des choix. Comme dans nombre de mes notebooks je m'arrête quand Jupyter me le demande : souffrance au chargement du notebook, ralentissements ? C'est le signe certain que la limite est atteinte :-).

## 1. Introduction

Désolé pour la théorie qui va suivre, c'est indispensable avant de pouvoir coder en Python.

## 1.1 C'est quoi ?

Soit $f : \mathbb{C} \to \mathbb{C}$. Pour tout $z_0 \in \mathbb{C}$, considérons la suite $(z_n)_{n\geq 0}$ définie par la relation de récurrence

$$z_{n+1} = f(z_n)$$

- L'ensemble de Julia "rempli" $K(f)$ est l'ensemble des $z_0 \in \mathbb{C}$ tels que la suite $(z_n)$ soit bornée.
- L'ensemble de Julia $J(f)$ est la **frontière** de $K(f)$. En d'autres termes, c'est l'ensemble des $z \in \mathbb{C}$ tels que, pour tout disque $D$ de centre $z$ et de rayon $> 0$, on ait $D \cap K(f) \neq \varnothing$ et $D \cap (\mathbb{C} \setminus K(f)) \neq \varnothing$. De façon très intuitive, $J_c$ est le "bord" de $K_c$. Mais attention à l'intuition, elle peut nous jouer de mauvais tours :-).

Dans ce notebook nous nous intéresserons uniquement à des fonctions $f : z \mapsto z^2 + c$, où $c \in \mathbb{C}$. Une telle fonction sera notée $f_c$ , et on notera plus simplement $K_c = K(f_c)$ et $J_c = J(f_c)$.

## 1.2 Un exemple

Prenons l'exemple le plus facile à étudier, $c = 0$. La suite $(z_n)$ est alors définie par la récurrence $z_{n+1} = z_n^2$. Il est immédiat que pour tout entier $n$,

$$z_n = z_0^{2^n}$$

- Si $|z_0| < 1$, alors $z_n$ tend vers 0 et, donc, $z_0 \in K_0$.
- Si $|z_0| > 1$, alors $|z_n|$ tend vers $+\infty$ et, donc, $z_0 \notin K_0$.
- Si, enfin, $|z_0| = 1$, alors pour tout entier $n$ $|z_n| = 1$, donc $z_0 \in K_0$.

Ainsi,

$$K_0 = \{z \in \mathbb{C}, |z| \leq 1\}$$

et la frontière de $K_0$ est

$$J_0 = \{z \in \mathbb{C}, |z| = 1\}$$

Ainsi, $K_0$ est le disque de centre $O$ et de rayon 1 et $J_0$ est le cercle de centre $O$ et de rayon 1.

## 1.3 Suite bornée ... mais par combien ?

**Quelques notations** : Soient $z \in \mathbb{C}$ et $r > 0$. Nous noterons $D(z, c)$ le *disque ouvert* de centre $z$ et de rayon $r$ :

$$D(z, r) = \{w \in \mathbb{C}, |w - z| < r\}$$

Nous noterons également

$$D(\infty, r) = \{w \in \mathbb{C}, |w| > r\}$$

Je laisse planer le mystère sur la notation $D(\infty, r)$ : un disque de centre à l'infini ? Bigre ...

**Proposition** : Soit $c \in \mathbb{C}$. Soit $z_0 \in \mathbb{C}$. On a $z_0 \in K_c$ si et seulement si

$$\forall n \geq 0, |z_n| \leq r(c)$$

où

$$r(c) = \max(|c|, 2)$$

**Démonstration** : Dans un sens c'est évident. Si la suite $(z_n)$ est majorée en module par $r(c)$ alors elle est bornée, donc $z_0 \in K_c$. Inversement, supposons que pour un certain entier $n$, $|z_n| > r(c)$. Posons $|z_n| = r(c) + h$, où $h > 0$. On a

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

**Exercice** : Montrer que $r(c)^2 - |c| \geq r(c)$. On distinguera 2 cas.

Ainsi,

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

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

$$|z_{n+p}| \geq r(c) + 4^p h$$

Ainsi, la suite $(z_n)$ n'est pas bornée et $z_0 \notin K_c$.

**Proposition** : Soit $c \in \mathbb{C}$. Alors

- $K_c$ est un ensemble compact.
- $J_c \subset K_c$.
- $J_c$ est un ensemble compact.

**Démonstration (délicate, peut être sautée)** : Nous avons déjà montré que $K_c$ est borné. Précisément, pour tout $r > r(x)$, on a

$$K_c \subset D(0, r)$$

Pourquoi $K_c$ est-il fermé ? Soit $z_0 \notin K_c$. Il existe donc un entier $n$ tel que $w = z_n \in D(\infty, r(c))$. Soit $\varepsilon > 0$ tel que pour tout $z \in D(w, \varepsilon)$ on ait $z \in D(\infty, r(c))$. Remarquons que $w = f_c^n(z_0)$. La fonction $f_c^n$, polynomiale, est continue en $z_0$. Il existe donc un réel $\alpha > 0$ tel que

$$f_c^n(D(z_0, \alpha)) \subset D(w, \varepsilon)$$

Conséquence : pour tout $z_0' \in D(z_0, \alpha)$, on a $|z_n'| > r(c)$ et, donc, $z_0' \notin K_c$. Nous venons donc de montrer que le complémentaire de $K_c$ est ouvert, c'est à dire que $K_c$ est fermé.

Les deux derniers points sont évidents : la frontière d'un ensemble fermé est incluse dans celui-ci, et la frontière d'un compact est compacte.

**Proposition** : $J_c$ est un ensemble d'intérieur vide.

**Démonstration** : Supposons que $J_c$ a un point intérieur $z$. Il existe alors un disque $D$ centré en $z$ de rayon $> 0$ et inclus dans $J_c$. Mais alors ce disque est inclus dans $K_c$ puisque $J_c \subset K_c$. Cela contredit le fait que $J_c$ est la frontière de $K_c$.

Disons pour faire vite que $J_c$ est un ensemble "tout fin". Quant à $K_c$, eh bien cela dépend de $c$. Nous verrons que pour certaines valeurs de $c$, $K_c$ est "tout tout fin" ... tandis que pour d'autres, il est "tout tout épais".

## 1.4 Itérations

Nous voici en possession d'une méthode pour estimer si un nombre complexe $z_0$ appartient à $J_c$. On calcule $z_0, z_1, \ldots$. Si l'un des $z_k$ est, en module, supérieur à $r(c)$, alors $z_0 \notin J_c$. Évidemment, on ne peut pas itérer indéfiniment. La fonction `iterer` ci-dessous prend en paramètres deux nombres complexes $z$ et $c$, un entier `niter` et un réel $r > 0$. Elle renvoie le plus petit entier $k \leq$ `niter` tel que $|z_k| > r$.

Entrée [3]:
```
def iterer(z, c, niter, r):
    k = 0
    w = z
    while k < niter and abs(w) <= r:
        w = w * w + c
        k += 1
    return k
```

L'évaluation ci-dessous nous prouve que $1.00001 \notin K_0$, ce que nous savions déjà :-).

Entrée [4]:
```
iterer(1.00001, 0, 256, 2)
```

Out[4]: 17

## 1.5 Paramètres, pixels, points

Tracer $K_c$ est maintenant facile. Pour "tous" les nombres complexes $z$ on appelle `iterer`. Si la fonction renvoie `niter` on décrète que $z \in K_c$. Sinon, on est certains que $z \notin K_c$. Évidemment, on ne peut pas regarder **tous** les nombres complexes. Nous tracerons $K_c \cap D$ où $D$ est un carré. Un tel carré peut être donné en fixant son centre, $z_c = x_c + iy_c$, et un facteur de zoom `zoom`. Décidons par exemple que lorsque `zoom=1` le carré est de côté 4. La classe `Parametre` ci-dessous permet de créer des objets contenant tous les paramètres nécessaires au tracé de $K_c$. Elle contient juste un constructeur qui prend en paramètres :

- Un nombre complexe $c$.
- Les coordonnées $xc$ et $yc$ du centre du carré.
- Un entier $n$ dont nous allons parler.
- Le facteur de zoom.
- Le nombre maximal d'itérations souhaitées.

Tous ces paramètres (mis à part $c$) ont des valeurs par défaut.

Le constructeur calcule également à partir de ces données les coordonnées minimales et maximales du carré qui nous intéresse.

Entrée [5]:
```
class Parametre:

    def __init__(self, c, xc=0, yc=0, n=300, zoom=1, niter=64, r=None
        self.c = c
        if r == None: self.r = max(2, abs(c))
        else: self.r = r
        self.xc = xc
        self.yc = yc
        self.zoom = zoom
        self.n = n
        self.niter = niter
        d = 2 / zoom
        self.xmin = xc - d
        self.xmax = xc + d
        self.ymin = yc - d
        self.ymax = yc + d
```

Entrée [6]:
```
param = Parametre(c=0, xc=0, yc=0)
print(param.xmin, param.xmax)
print(param.ymin, param.ymax)
print('----------------')
param = Parametre(c=0, xc=0, yc=0, zoom=10)
print(param.xmin, param.xmax)
print(param.ymin, param.ymax)
```

```
-2.0 2.0
-2.0 2.0
----------------
-0.2 0.2
-0.2 0.2
```

On ne peut pas évidemment non plus tester l'appartenance à $K_c$ de **tous** les points d'un carré ! Concrètement, une *image* sur notre écran de $K_c$ est formée de pixels et possède un certain nombre de lignes et un certain nombre de colonnes, disons $n$. Ce $n$ est passé en paramètre au constructeur de la classe `Parametre` .

Chaque *pixel* de l'image correspond à un *nombre complexe*. Comment passer des coordonnées $(i, j)$ d'un pixel à un nombre complexe ? La fonction `point` ci-dessous fait le travail.

Remarquez quelques subtilités dans les formules : $i$ et $j$ représentent le numéro de ligne et le numéro de colonne d'une matrice. Si vous "dessinez" cette matrice, vous allez vous rendre compte que $i$ est une **ordonnée** qui "augmente lorsqu'on descend", et $j$ est une **abscisse** qui "augmente quand on va vers la droite". D'où les subtiles interversions, $x$ est copain de $j$, $y$ est anti-copain de $i$ ...

Entrée [7]:
```
def point(pixel, param):
    i, j = pixel
    x = param.xmin + j * (param.xmax - param.xmin) / param.n
    y = param.ymax + i * (param.ymin - param.ymax) / param.n
    return x + 1j * y
```

Entrée [8]:
```
param = Parametre(0)
print(point((0, 0), param))
print(point((300, 0), param))
```

```
(-2+2j)
(-2-2j)
```

## 1.6 Dessiner les ensembles de Julia

Oui, bon, quand est-ce qu'on dessine les ensembles de Julia ? Si vous voulez des dessins vite fait sans rien comprendre, allez sur Google. Ce n'est absolument pas le but recherché ici.

D'abord une fonction qui prend un entier $n$ en paramètre et renvoie une matrice $n \times n$ initialisée à 0.

Entrée [9]:
```
def matrice(n):
    m = n * [None]
    for i in range(n):
        m[i] = n * [0]
    return m
```

Entrée [10]:
```
matrice(3)
```

Out[10]:
```
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

Voici la fonction `julia`. Elle prend un objet de la classe Parametre en ... paramètre. Elle remplit une matrice des résultats renvoyés par la fonction `iterer`.

Entrée [11]:
```
def julia(param):
    m = matrice(param.n)
    for i in range(param.n):
        for j in range(param.n):
            z = point((i, j), param)
            k = iterer(z, param.c, param.niter, param.r)
            m[i][j] = k
    return m
```

`show_julia` affiche une image d'une matrice

Entrée [12]:
```
def show_julia(m, param):
    plt.imshow(m, interpolation='bicubic', cmap='hot', extent=(param
    plt.grid()
```

Et `run_julia` combine `julia` et `show_julia`.

Entrée [13]:
```
def run_julia(param):
    m = julia(param)
    show_julia(m, param)
```

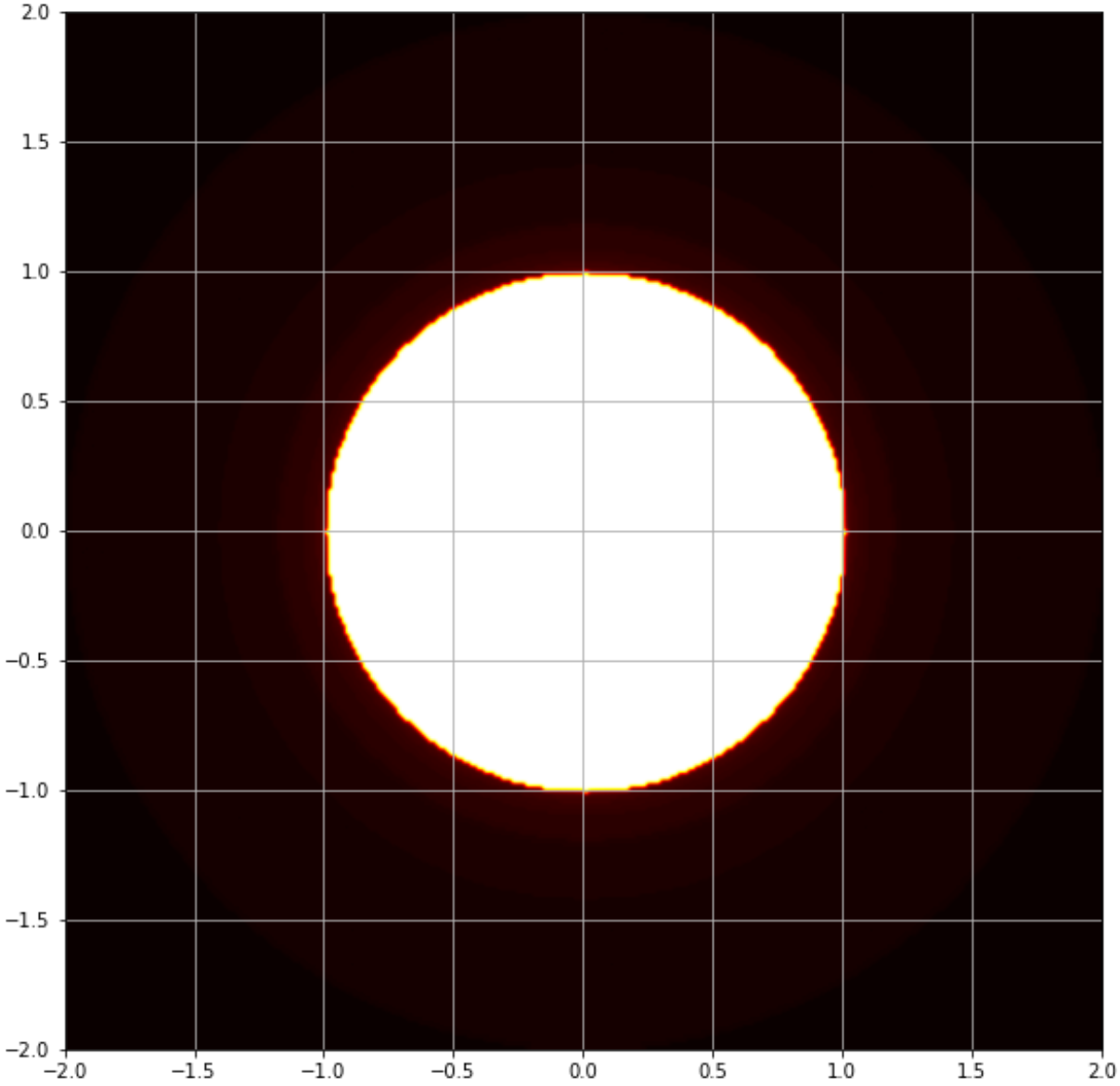
## 1.7 Quelques exemples

Nous y sommes. Dans ce paragraphe nous allons lancer `run_julia` sur quelques valeurs typiques de $c$.

### 1.7.1 $c = 0$

Tout d'abord le cas où $c = 0$. Nous savons qu'un disque doit apparaître.

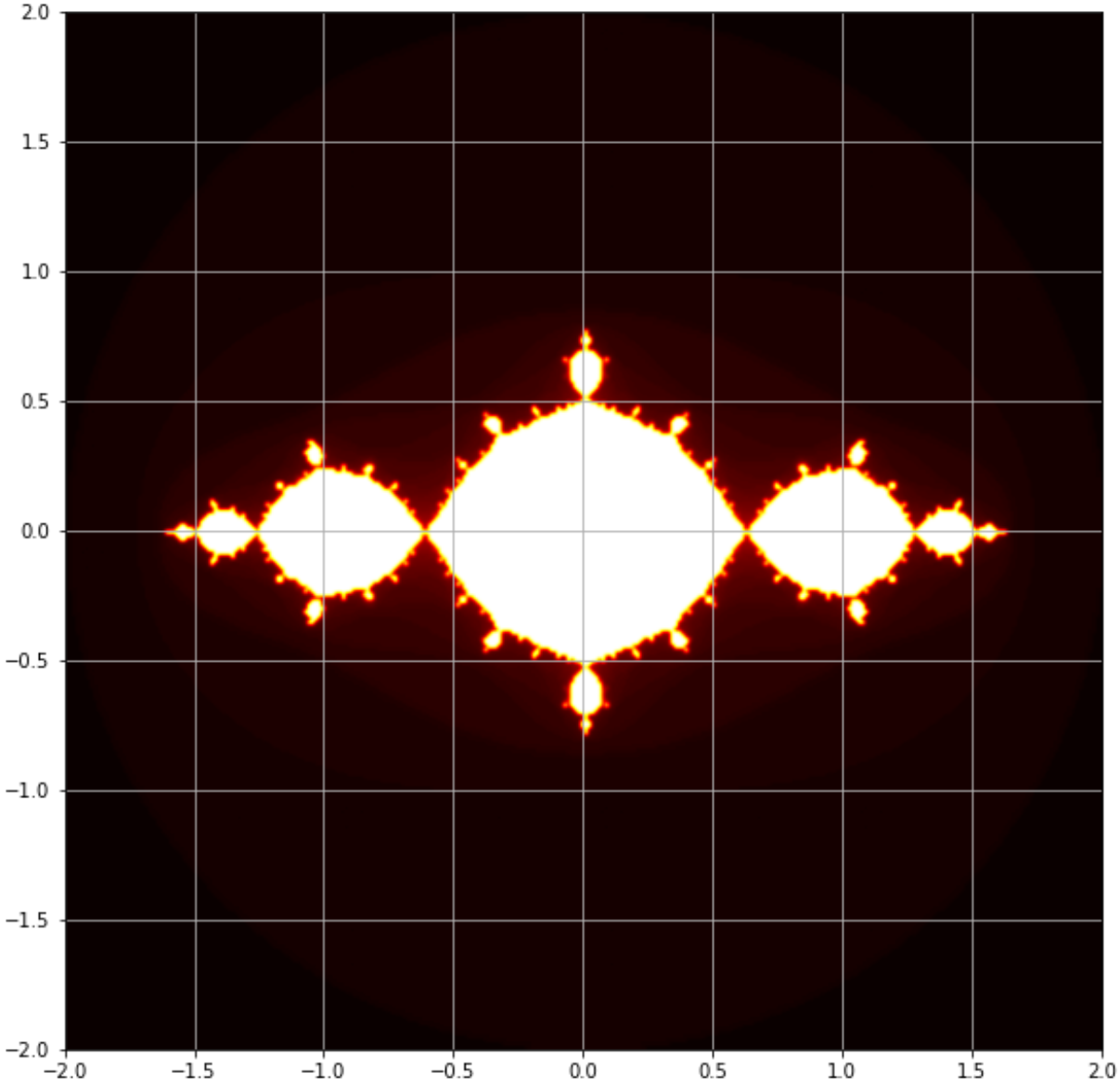
Entrée [14]:
```
param = Parametre(0)
run_julia(param)
```




### 1.7.2 $c = -1$

Le cas $c = -1$ est aussi fort intéressant.

Entrée [15]:
```
param = Parametre(-1)
run_julia(param)
```

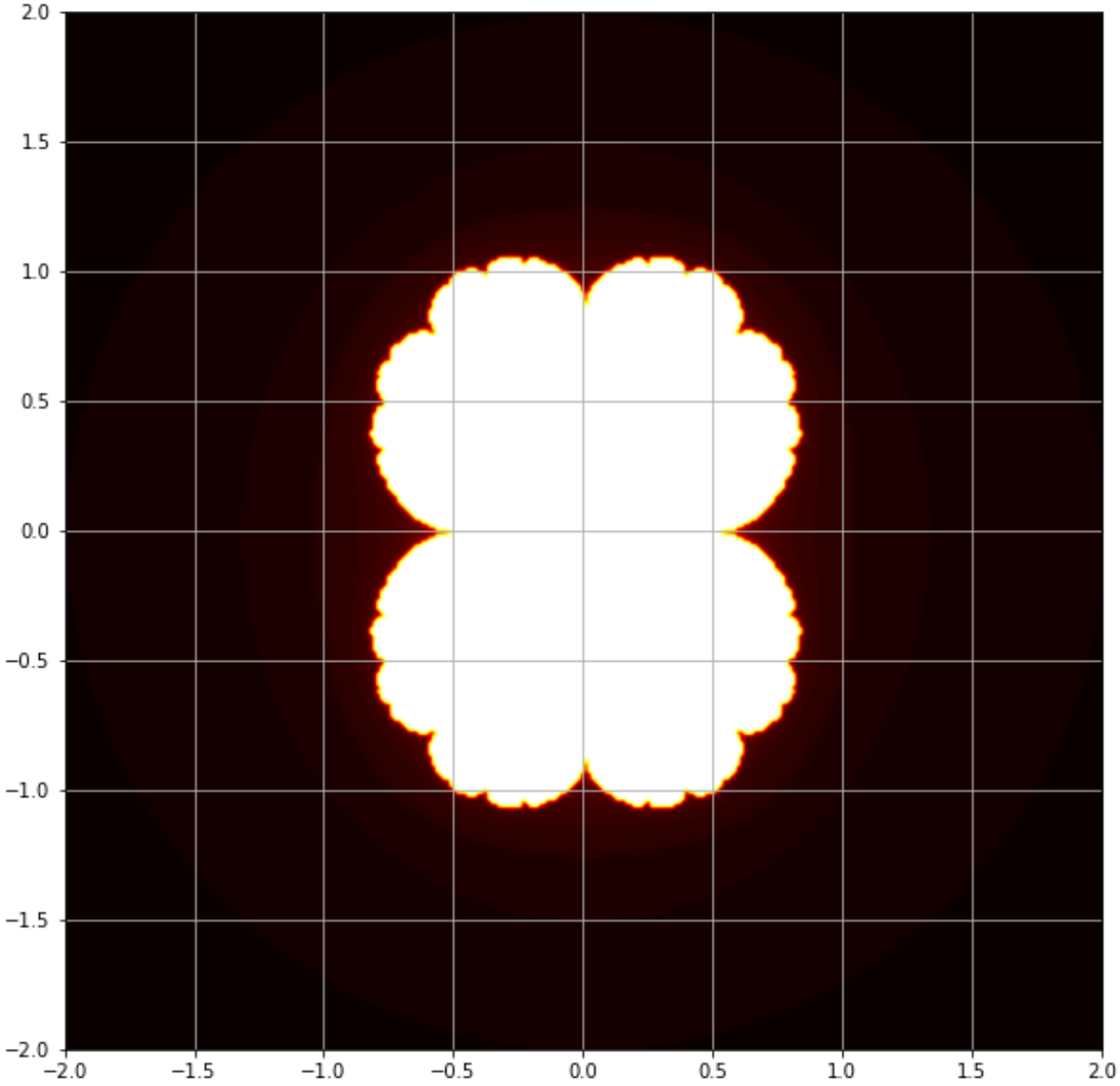



**Exercice** : Le nombre d'or se cache quelque part dans le dessin ci-dessus. Saurez-vous le trouver ?

### 1.7.3 $c = \frac{1}{4}$

Le cas où $c = \frac{1}{4}$ est aussi fort instructif. Je n'y reviendrai pas par manque de place dans ce notebook.

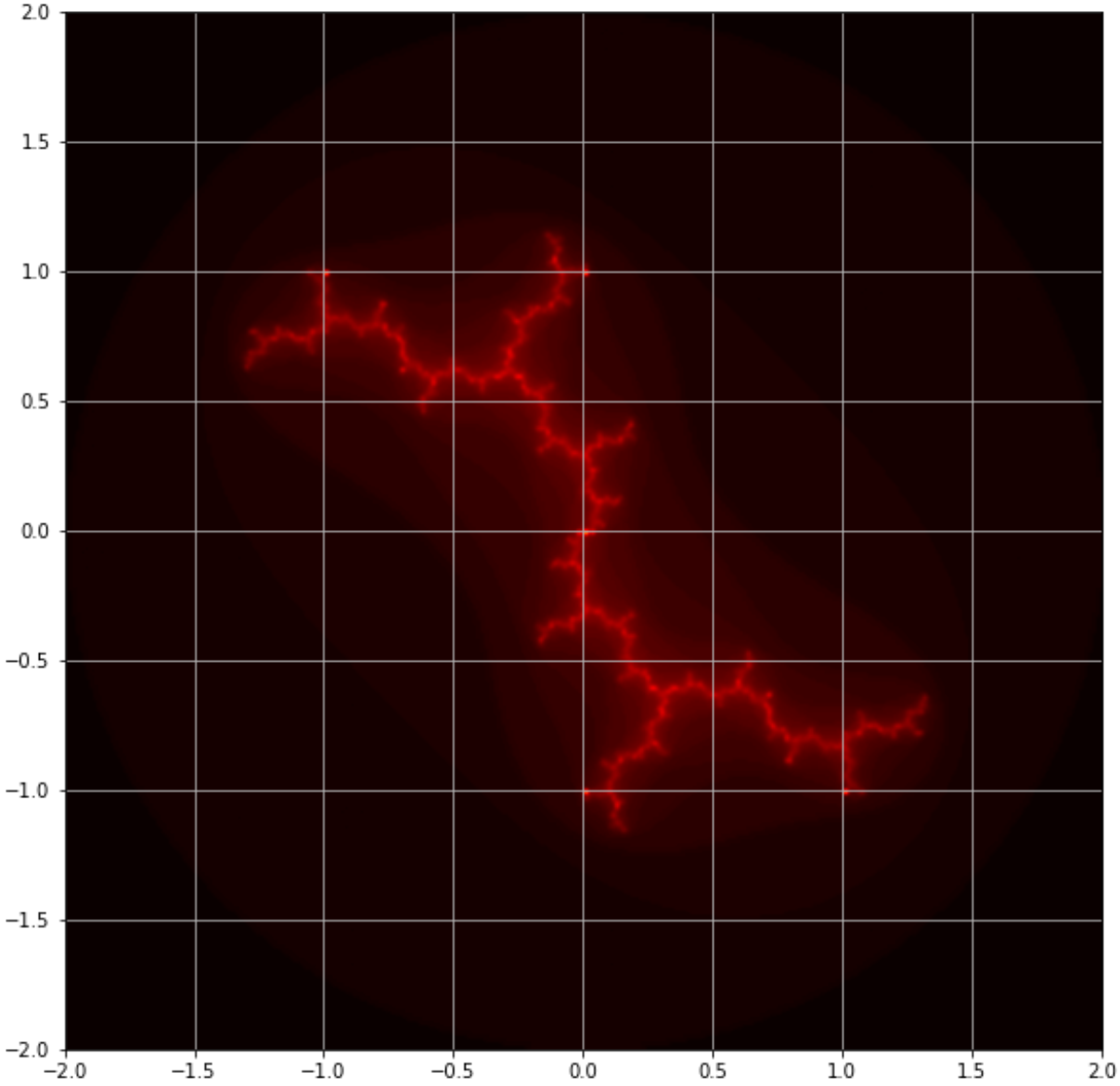
Entrée [16]:
```
param = Parametre(1 / 4)
run_julia(param)
```




### 1.7.4 Une dendrite

Tentons $c = i$.

Entrée [17]:
```
param = Parametre(1j)
run_julia(param)
```

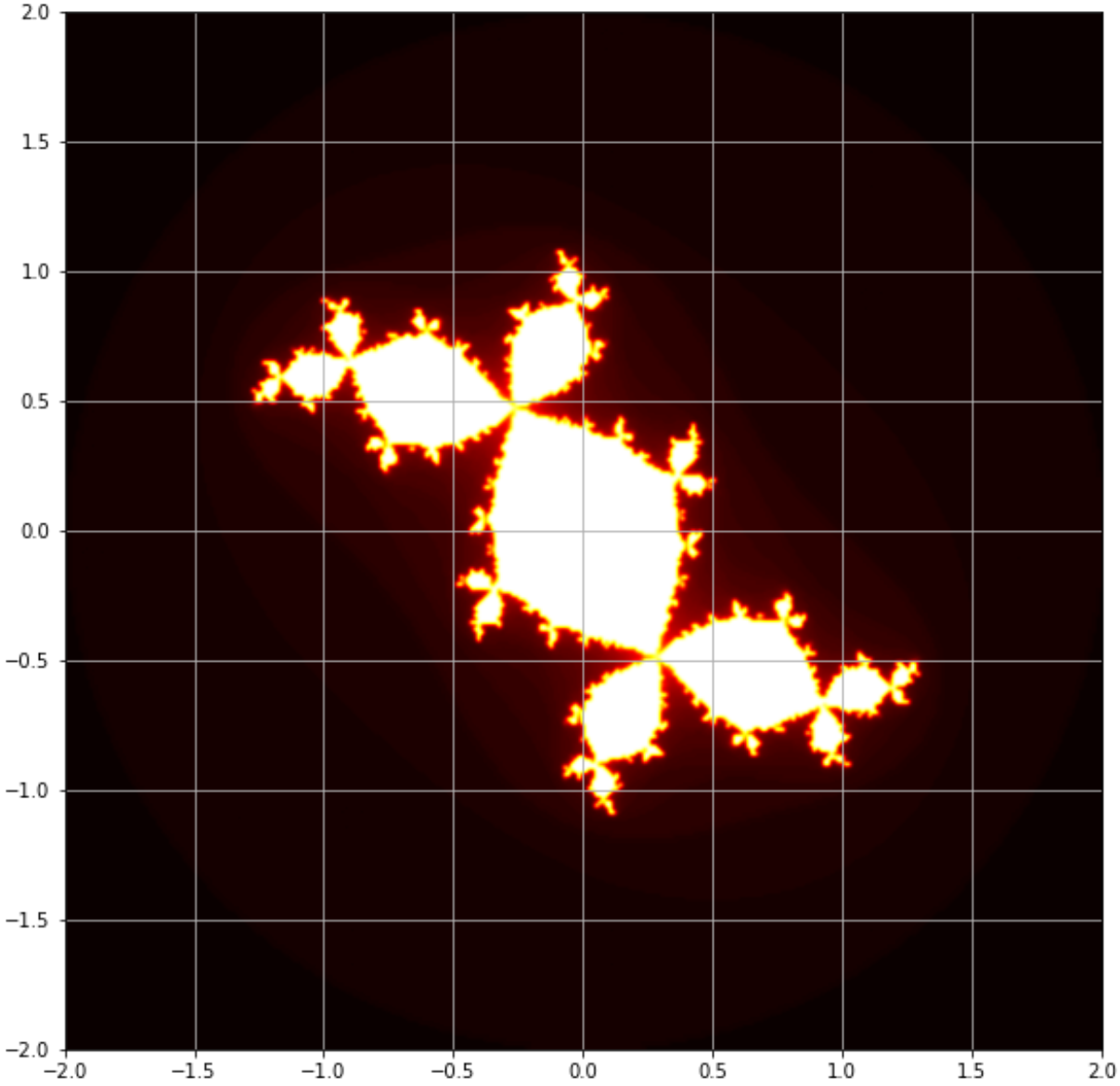



On obtient un ensemble formé de sortes de filaments. Les spécialistes appelent ce type d'ensemble de Julia une *dendrite*.

### 1.7.5 Le lapin de Douady

Le cas $c \simeq -0.12 + 0.74i$ a été étudié par le mathématicien Adrien Douady dont nous reparlerons plus loin. Il a appelé $J_c$ et les ensembles de la même famille des "lapins".

Entrée [18]:
```
param = Parametre(-0.12 + 0.74 * 1j)
run_julia(param)
```




# 2. Le potentiel de Hubbard-Douady

## 2.1 Introduction

Ce dont je vais parler ici est le résultat des travaux de deux mathématiciens, John Hubbard et Adrien Douady.

Imaginons un cyclindre vertical infini dont la section horizontale (la base, si vous préférez) est l'ensemble $K_c$. Supposons ce cylindre chargé électriquement. Le cylindre crée un champ électrique dans l'espace extérieur au cylindre : peut-on préciser l'expression de ce champ ?

Le cas $c = 0$ nous donne un "vrai" cylindre, un fil épais dirons-nous. Le champ est alors bien connu, il ne dépend pas de l'altitude du point considéré, et l'intensité de celui-ci en un point $M \in \mathbb{R}^3$ d'abscisse $x$ et d'ordonnée $y$ a une amplitude qui décroît comme $\frac{1}{r}$ où $r = \sqrt{x^2 + y^2}$. Ce champ dérive du potentiel

$$G(M) = \lg r$$

où $\lg$ est le logarithme en base 2.

Oublions dorénavant la troisième dimension, et posons-nous dans le plan complexe. Le potentiel en un point $z_0 \in \mathbb{C}$ dans le cas $c = 0$ est donc $G(z_0) = \lg |z_0|$.

Qu'arrive-t-il maintenant pour $c$ quelconque ? Adrien Douady et John Hubbard ont montré que si l'ensemble $J_c$ est **connexe** alors le champ dérive d'un potentiel. Et non, ce n'est pas du tout évident ! Peut-on avoir une expression du potentiel $G(z_0)$ en un point $z_0$ quelconque ? Eh bien

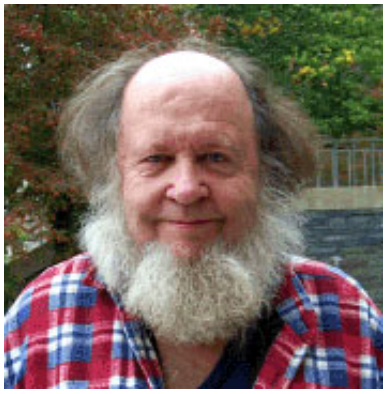
$$G(z) = \lim_{n \to \infty} \frac{\lg |z_n|}{2^n}$$

Remarquez que pour $c = 0$ nous avions trouvé au début de ce notebook que $z_n = z_0^{2^n}$. Ainsi,

$$\lg |z_n| = 2^n \lg |z_0|$$

et dans ce cas trivialissime nous retrouvons bien la valeur du potentiel de Douady-Hubbard. Il n'y a même pas besoin de passer à la limite :-).

**Exercice** : Qui est John Hubbard ? Et qui est Adrien Douady ? Indication : passez la souris sur les images :-).



## 2.2 Valeur approchée du potentiel

Pour évaluer le potentiel en un point $z_0$ il suffit donc de calculer $\frac{\lg |z_n|}{2^n}$ pour un entier $n$ "assez grand". Grand comment ? Là nous allons rester vagues : suffisamment grand pour que $|z_n|$ soit grand :-). La fonction `potentiel` ressemble comme deux gouttes d'eau à la fonction `iterer` sauf que l'on sort de la boucle lorsque $|z_n| \geq 10^6$ ou bien lorsque le compteur $k$ dépasse une certaine valeur `niter` .

- Si, à la sortie de la boucle, $k =$ `niter` , on décrète que $z \in K_c$. Le potentiel n'est alors pas défini en $z$ (enfin, si, mais je n'en parlerai pas ici). On renvoie arbitrairement 0. Cette valeur est d'ailleurs assez logique puisque, dans le cas où $z_0 \in K_c$, la suite $(z_n)$ est bornée et, donc, $\frac{\lg |z_n|}{2^n}$ tend vers 0 lorsque $n$ tend vers l'infini.
- Dans le cas contraire, on renvoie $\frac{\lg |w|}{2^k}$ où $w = z_k$. La valeur renvoyée, est, on l'espère, une bonne approximation de $G(z)$.

Entrée [19]:
```
def potentiel(z, c, niter):
    k = 0
    w = z
    while k < niter and abs(w) <= 1e6:
        w = w * w + c
        k += 1
    if k == niter: return 0
    else:
        return math.log(abs(w), 2) / 2 ** k
```

Entrée [20]:
```
potentiel(8, -1, 256)
```

Out[20]:
```
2.988549066143139
```

Entrée [21]:
```
potentiel(8, 0, 256)
```

Out[21]:
```
3.0
```

Les deux valeurs trouvées ci-dessus ont l'air très proches. Comment se fait-il ? Eh bien vu de très loin, notre ensemble de Julia ne diffère pas beaucoup d'un point. Les valeurs du potentiel pour $z$ de grand module devraient toutes ressembler à celles que l'on obtient pour $c = 0$.

Tentons par exemple la valeur du potentiel pour $z = 1024 = 2^{10}$ et $c =$, par exemple, $3 + 2i$. Cela devrait donner à peu près 10 ...

Entrée [22]:
```
print(potentiel(1024, 3 + 2 * 1j, 256))
```

```
10.00000206379017
```

## 2.3 Les équipotentielles

Les équipotentielles, comme leur nom l'indique, sont les lignes de niveau du potentiel, c'est à dire les courbes d'équations $G(z) = c^{\text{te}}$. Comment les visualiser ? La fonction `julia2` est une quasi-recopie de la fonction `julia`. En "chaque" point $z \in \mathbb{C}$ extérieur à $J_c$, elle calcule $G(z)$. Puis elle renvoie l'unique entier $j$ tel que $2^j \leq G(z) < 2^{j+1}$. Plus précisément, $j = \lfloor \lg G(z) \rfloor$ et la fonction renvoie $j \bmod 2$. Le dessin que nous allons obtenir est donc en noir et blanc.

Entrée [23]:
```
def julia2(param):
    m = matrice(param.n)
    for i in range(param.n):
        for j in range(param.n):
            z = point((i, j), param)
            G = potentiel(z, param.c, param.niter)
            if G == 0: m[i][j] = 0
            else: m[i][j] = math.floor(math.log(G, 2) ) % 2
    return m
```

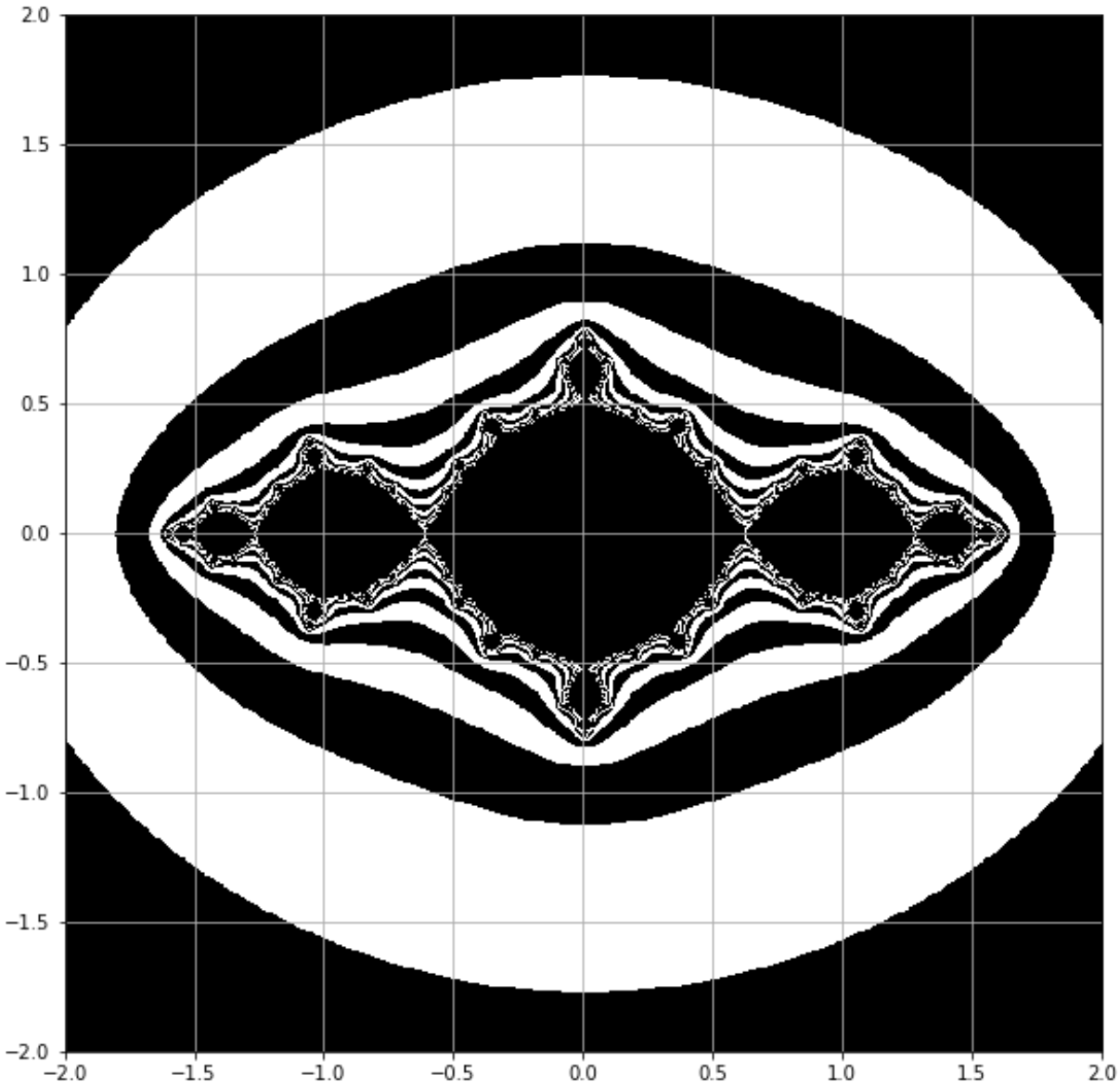
Les fonctions `show_julia2` et `run_julia2` ne nécessitent pas d'explications.

Entrée [24]:
```
def show_julia2(m, param):
    plt.imshow(m, interpolation='none', cmap='gray', extent=(param.x
    plt.grid()
```

Entrée [25]:
```
def run_julia2(param):
    m = julia2(param)
    show_julia2(m, param)
```

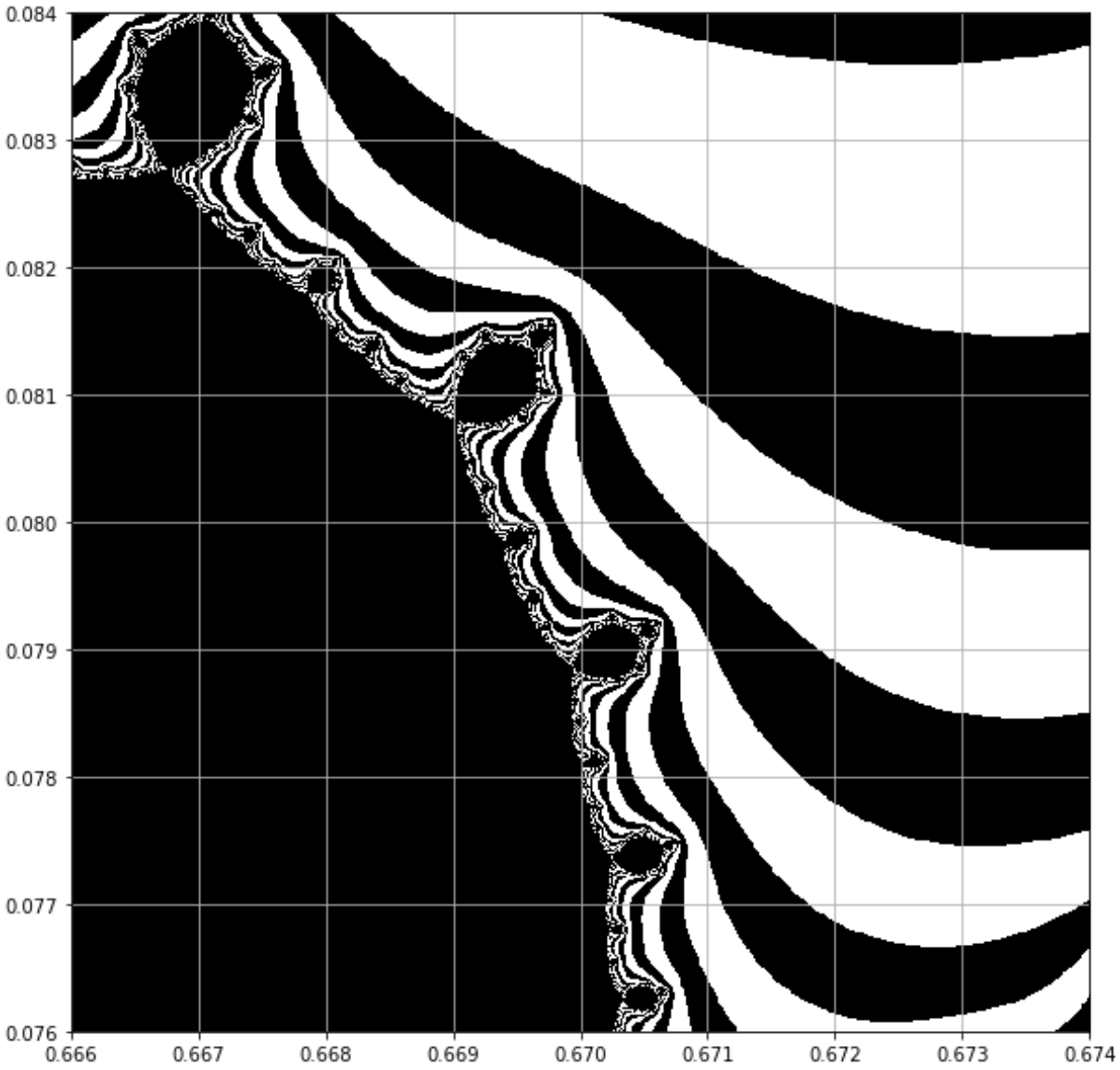
Voici $J_{-1}$, nouvelle version.

Entrée [26]:
```
param = Parametre(-1, zoom=1, n=600, niter=256)
run_julia2(param)
```



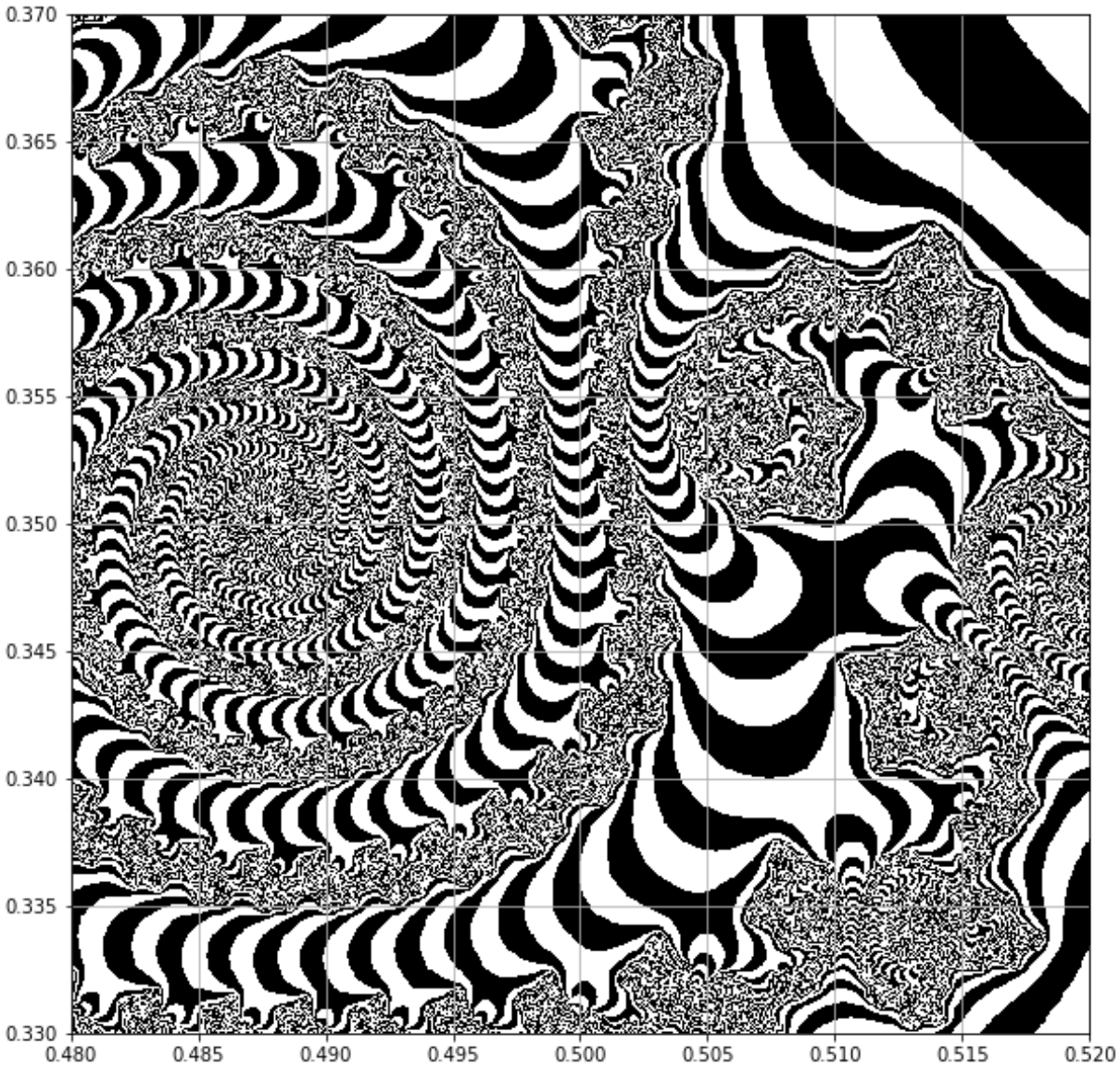

Un zoom sur $J_{-1}$ ?

Entrée [27]:
```
param = Parametre(-1, xc=0.67, yc=0.08, zoom=500, n=600, niter=256)
run_julia2(param)
```




Voici un dernier exemple pour montrer toute la complexité des ensembles de Julia.

Entrée [28]:
```
param = Parametre(-0.174893+0.656878*1j, xc=0.5, yc=0.35, n=600, zoo
run_julia2(param)
```




Houlala ! Vous pouvez maintenant tester avec d'autres valeurs de $c$, des zooms appropriés, etc. Amusez-vous bien.

# 3. Lignes de champ

## 3.1 C'est quoi ?

Supposons donné un champ électrique $\overrightarrow{E}$. Les lignes de champ de $\overrightarrow{E}$ sont les courbes dont la tangente est parallèle au champ. Ces courbes sont orthogonales aux équipotentielles. Dans cette section nous allons tracer les lignes du champ associé à un ensemble de Julia.

Soit $G$ le potentiel que nous avons défini dans la section précédente. Le champ est donné par

$$\overrightarrow{E} = -\overrightarrow{\nabla} G$$

où $\overrightarrow{\nabla} G$ est le **gradient** de $G$, donné en termes complexes par

$$\overrightarrow{\nabla} G = \frac{\partial G}{\partial x} + i\frac{\partial G}{\partial y}$$

Ce que nous cherchons à tracer, ce sont les **courbes paramétrées** $f : I \to \mathbb{C}$ telles que pour tout $t \in I$, $f'(t)$ soit colinéaire à $\overrightarrow{E}(f(t))$. Sans entrer dans des détails sur la théorie sous-jacente, nous allons résoudre (de façon approchée) l'équation différentielle

$$f' = \lambda \overrightarrow{E} \circ f$$

où $\lambda$ est une constante dont nous ne nous préoccuperons pas. Dans la suite nous prendrons très négligemment $\lambda = 1$.

## 3.2 Ligne de champ passant par un point donné

Comment approcher la ligne de champ passant par un point $z_0$ donné ? Supposons le gradient connu, et ses valeurs stockées dans une matrice $G$. Soit $(i, j)$ le pixel associé à $z_0$. Nous allons approximer $\frac{\partial G}{\partial x}(z_0)$ par

$$\frac{\partial G}{\partial x}(z_0) \simeq \frac{G[i][j+1] - G[i][j]}{\delta}$$

où $\delta$ est la distance entre deux complexes correspondant à des pixels adjacents. De même

$$\frac{\partial G}{\partial y}(z_0) \simeq \frac{G[i-1][j] - G[i][j]}{\delta}$$

Soit $h > 0$ un réel fixé. Mettons que $z_0 = f(t_0)$ pour un certain $t_0$. On a alors

$$f(t_0 + h) \simeq f(t_0) + hf'(t_0) = f(t_0) + h\overrightarrow{E}(f(t_0)) = z_0 - \overrightarrow{\nabla} G(z_0)$$

Vous avez sans doute reconnu la méthode d'Euler de résolution approchée d'une équation différentielle ?

Posons $z_1 = f(t_0 + h)$, puis recommençons avec $z_1$ ... etc.

La fonction `ligne_champ` affiche une ligne de champ. Elle prend en paramètres :

- Une matrice $G$ contenant les valeurs du potentiel associé au champ.
- Un nombre complexe $z$ par lequel passe la ligne que nous voulons tracer.
- Un réel $h > 0$ supposé "petit".
- Un réel $\varepsilon \geq 0$ supposé aussi "petit".
- Et le sempiternel `param`, objet de la classe `Parametre` qui contient toutes les informations nécessaires pour passer d'un pixel à un nombre complexe et vice versa.

La fonction effectue une boucle `while`. Cette boucle est équipée d'un certain nombre de garde-fous dont je vous laisse méditer l'utilité.

Entrée [29]:
```
def ligne_champ(G, z, h, eps, param):
    i, j = pixel(z, param)
    zs = [z]
    count = 0
    delta = (param.xmax - param.xmin) / param.n
    while count < 10000 and i > 0 and i < param.n and j >= 0 and j <
        u = (G[i][j + 1] - G[i][j]) / delta
        v = (G[i - 1][j] - G[i][j]) / delta
        z = z - (u + 1j * v) * h
        zs.append(z)
        count += 1
        i, j = pixel(z, param)
    xs = [z.real for z in zs]
    ys = [z.imag for z in zs]
    plt.plot(xs, ys, 'r')
```

Et la ligne `i, j = pixel(point, param)`, c'est quoi ? Nous avions écrit il y a bien longtemps une fonction `point` convertissant un pixel en point du plan. Voici sa réciproque, la fonction `pixel`.

Entrée [30]:
```
def pixel(z, param):
    i = int(param.n * (z.imag - param.ymax) / (param.ymin - param.ym
    j = int(param.n * (z.real - param.xmin) / (param.xmax - param.xm
    return (i, j)
```

## 3.3 Tracés

La fonction `calcul_potentiels` renvoie une matrice contenant les valeurs du potentiel.

Entrée [31]:
```
def calcul_potentiels(param):
    m = matrice(param.n)
    for i in range(param.n):
        for j in range(param.n):
            z = point((i, j), param)
            G = potentiel(z, param.c, param.niter)
            m[i][j] = G
    return m
```

La fonction `julia3` est quasiment identique à `julia2`, sauf qu'elle renvoie aussi la valeur du potentiel en "tout" point.

Entrée [32]:
```
def julia3(param):
    G = calcul_potentiels(param)
    m = matrice(param.n)
    for i in range(param.n):
        for j in range(param.n):
            z = point((i, j), param)
            if G[i][j] == 0: m[i][j] = 0
            else: m[i][j] = math.floor(math.log(G[i][j], 2) ) % 2
    return (G, m)
```

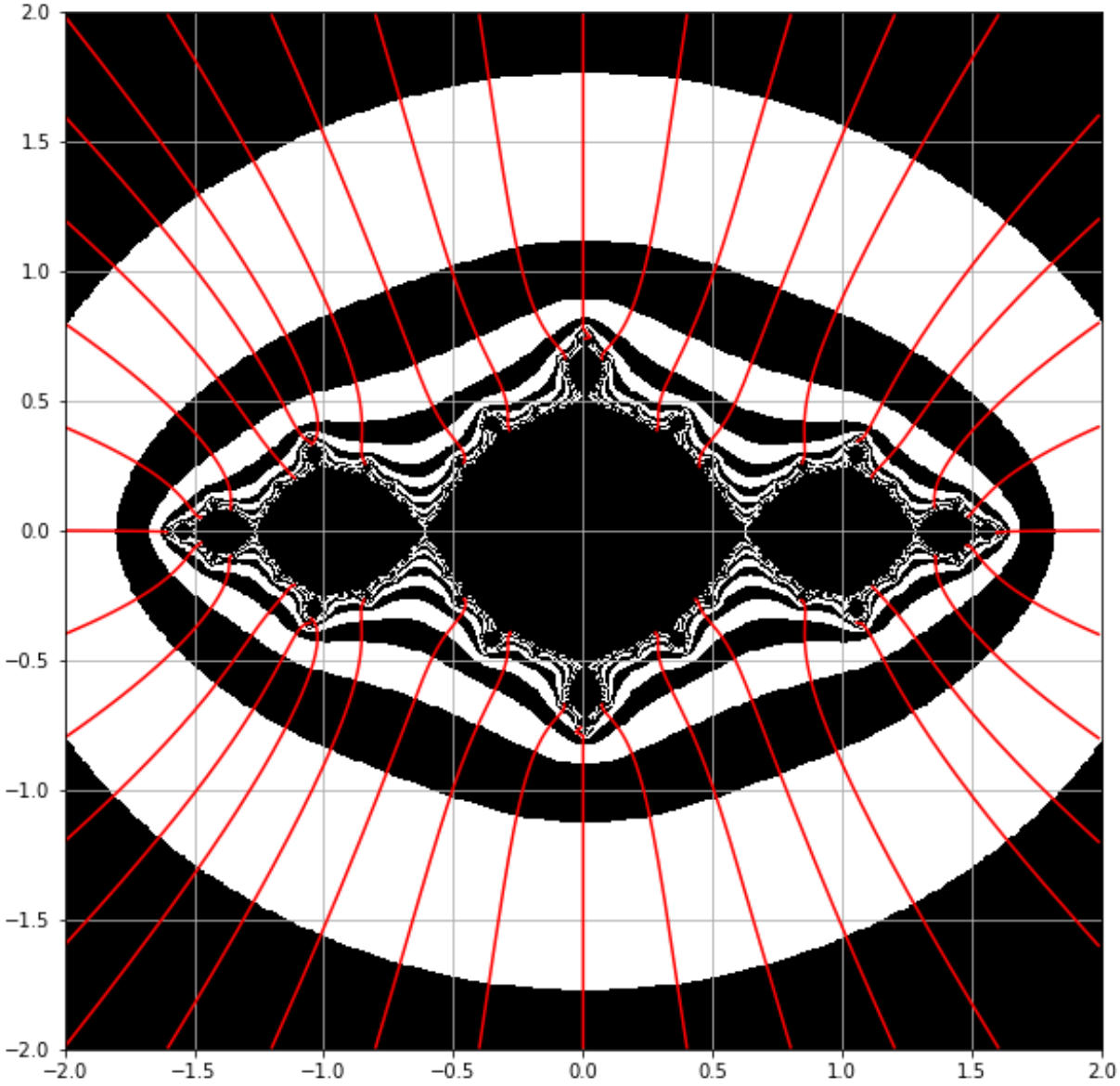
Enfin, `run_julia3` appelle `julia3` puis trace quelques lignes de champ. Plus précisément, elle trace les lignes passant par des points régulièrement espacés sur le bord du carré défini par `param`. Le paramètre $N$ définit le nombre de lignes pour chaque côté du carré.

J'ai mis des valeurs par défaut pour les paramètres $h$, `eps` et $N$. Le résultat est très sensible à la valeur de $h$. Si vous obtenez des courbes qui "font n'importe quoi", diminuez $h$. Si, au contraire, les lignes de champ n'atteignent pas l'ensemble de Julia, c'est que $h$ est trop petit. Augmentez sa valeur.

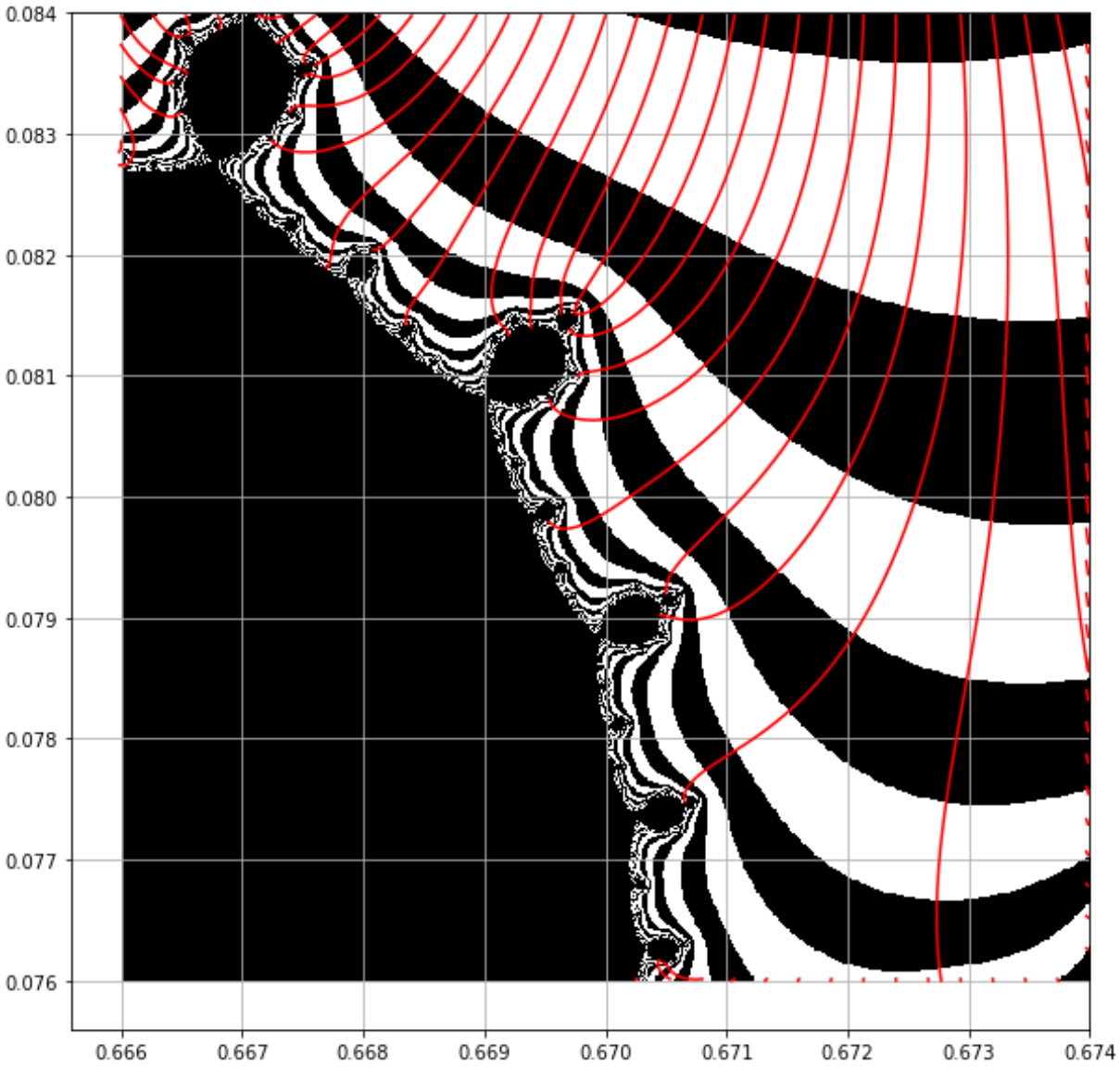
Entrée [33]:
```
def run_julia3(param, h=1e-2, eps=0, N=10):
    (G, m) = julia3(param)
    plt.imshow(m, interpolation='none', cmap='gray', extent=(param.x
    for i in range(N + 1):
        i1 = i * param.n / N
        z = point((i1, 0), param)
        ligne_champ(G, z, h, eps, param)
        z = point((i1, param.n - 2), param)
        ligne_champ(G, z, h, eps, param)
        z = point((2, i1), param)
        ligne_champ(G, z, h, eps, param)
        z = point((param.n - 1, i1), param)
        ligne_champ(G, z, h, eps, param)
    plt.grid()
```

Entrée [34]:

```
param = Parametre(-1, n=600)
run_julia3(param)
```



Entrée [39]:
```
param = Parametre(-1, xc=0.67, yc=0.08, zoom=500, n=600, niter=256)
run_julia3(param, h=1e-3, N=30)
```



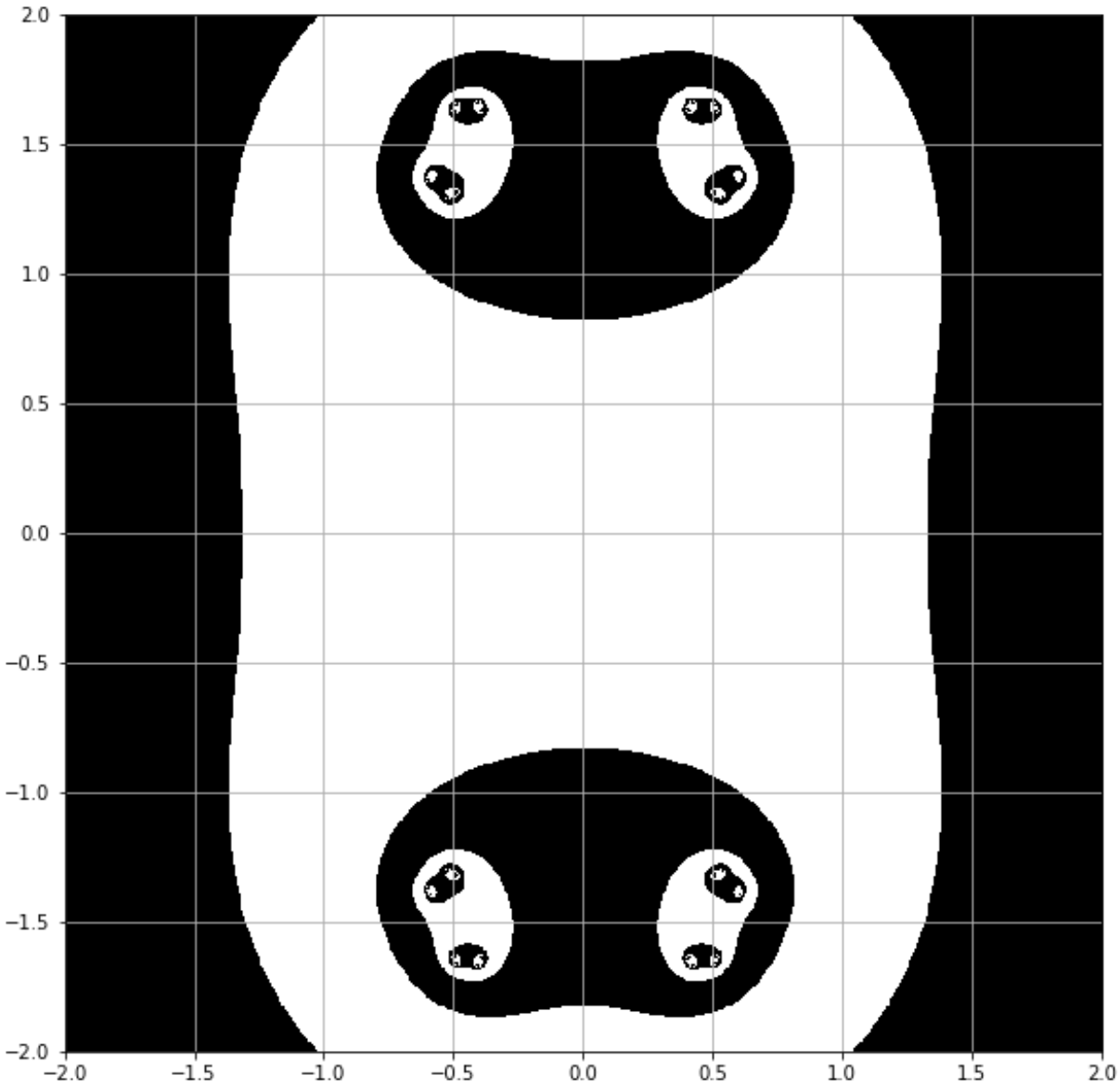

**Exercice** : Essayez de tracer $K_{-1}$ avec des valeurs de $\varepsilon$ égales à 1, $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, etc. Explication ???

**Exercice (durée : 1 heure)** : Prenez d'autres valeurs de $c$, par exemple $0$, $i$, $\frac{1}{4}$, $-1.6$, le $c$ du lapin de Douady, etc. Faites des zooms. Bref, obtenez par l'expérience une **compréhension** des équipotentielles et des lignes de champ.

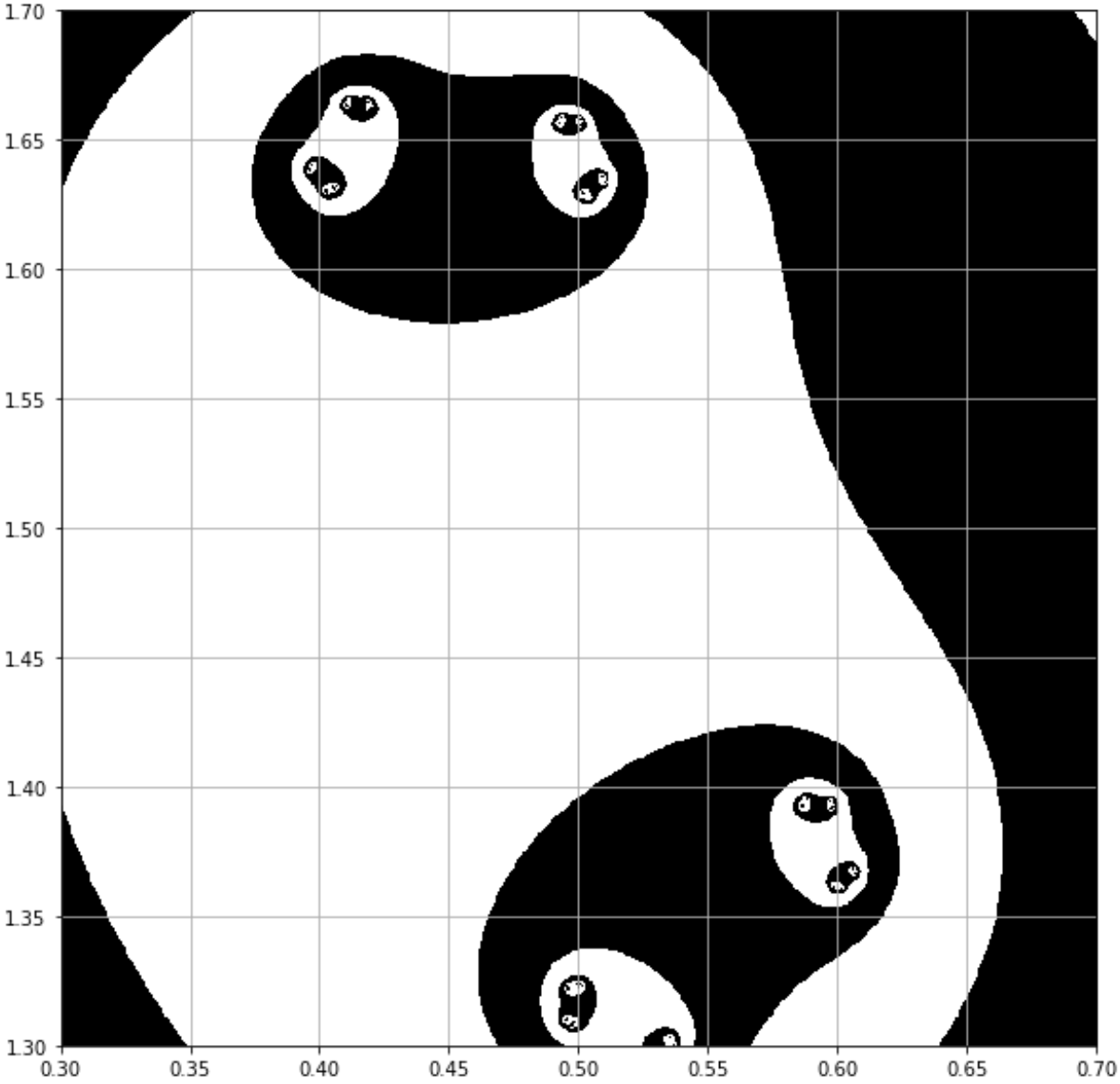
## 4. Poussières de Cantor

Dans cette section nous allons examiner ce qui se passe lorsque $|c| \geq 2$. Ci-dessous, l'ensemble de Julia $J_2$. Je garderai $c = 2$ dans tous les exemples mais rien ne vous empêche d'essayer d'autres valeurs !

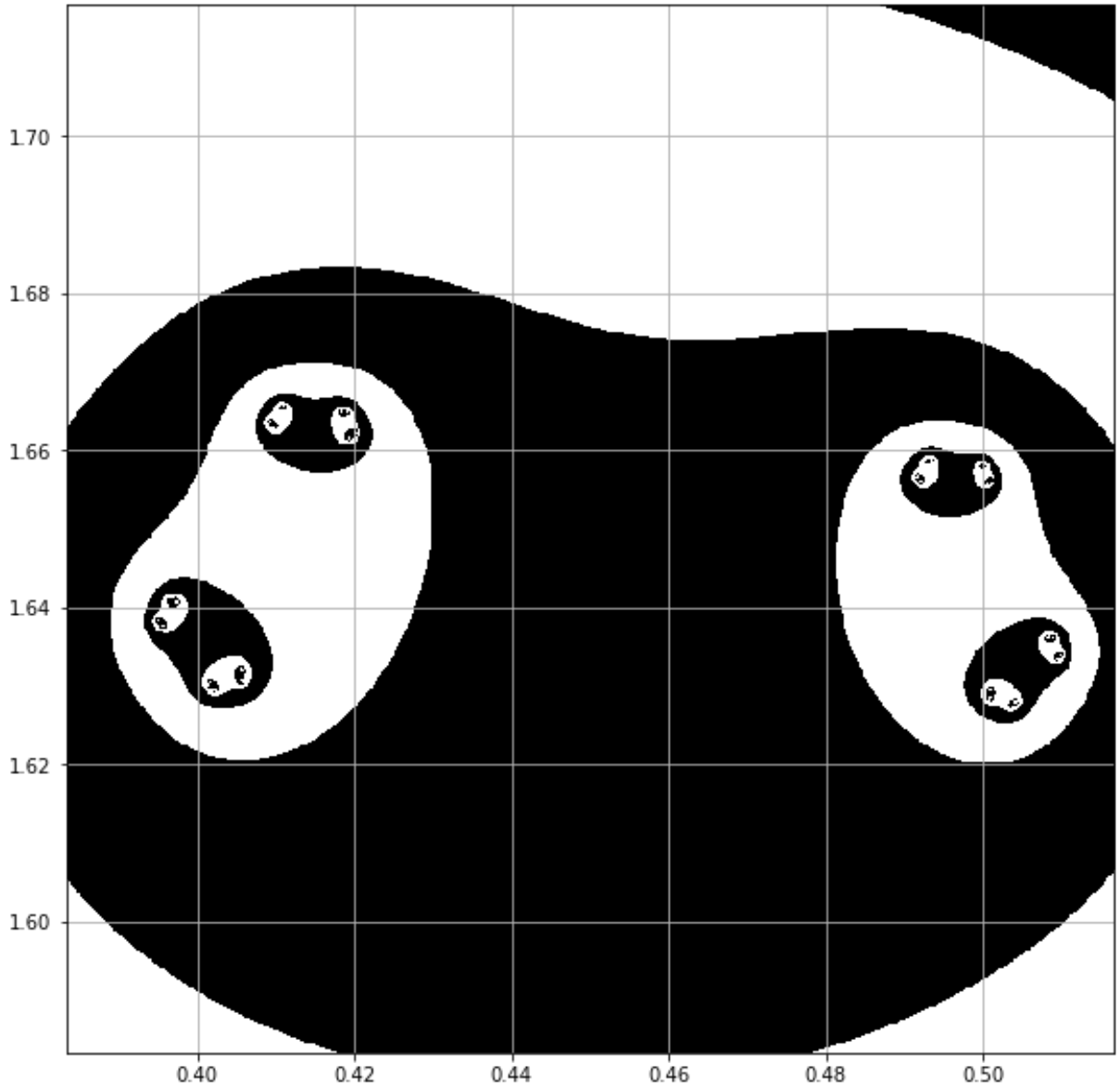
Entrée [40]:
```
param = Parametre(2, n=600)
run_julia2(param)
```



Entrée [41]:
```
param = Parametre(2, n=600, xc=0.5, yc=1.5, zoom=10)
run_julia2(param)
```



Entrée [42]:
```
param = Parametre(2, n=600, xc=0.45, yc=1.65, zoom=30)
run_julia2(param)
```




Que se passe-t-il ici ? C'est ce que nous allons essayer de comprendre ...

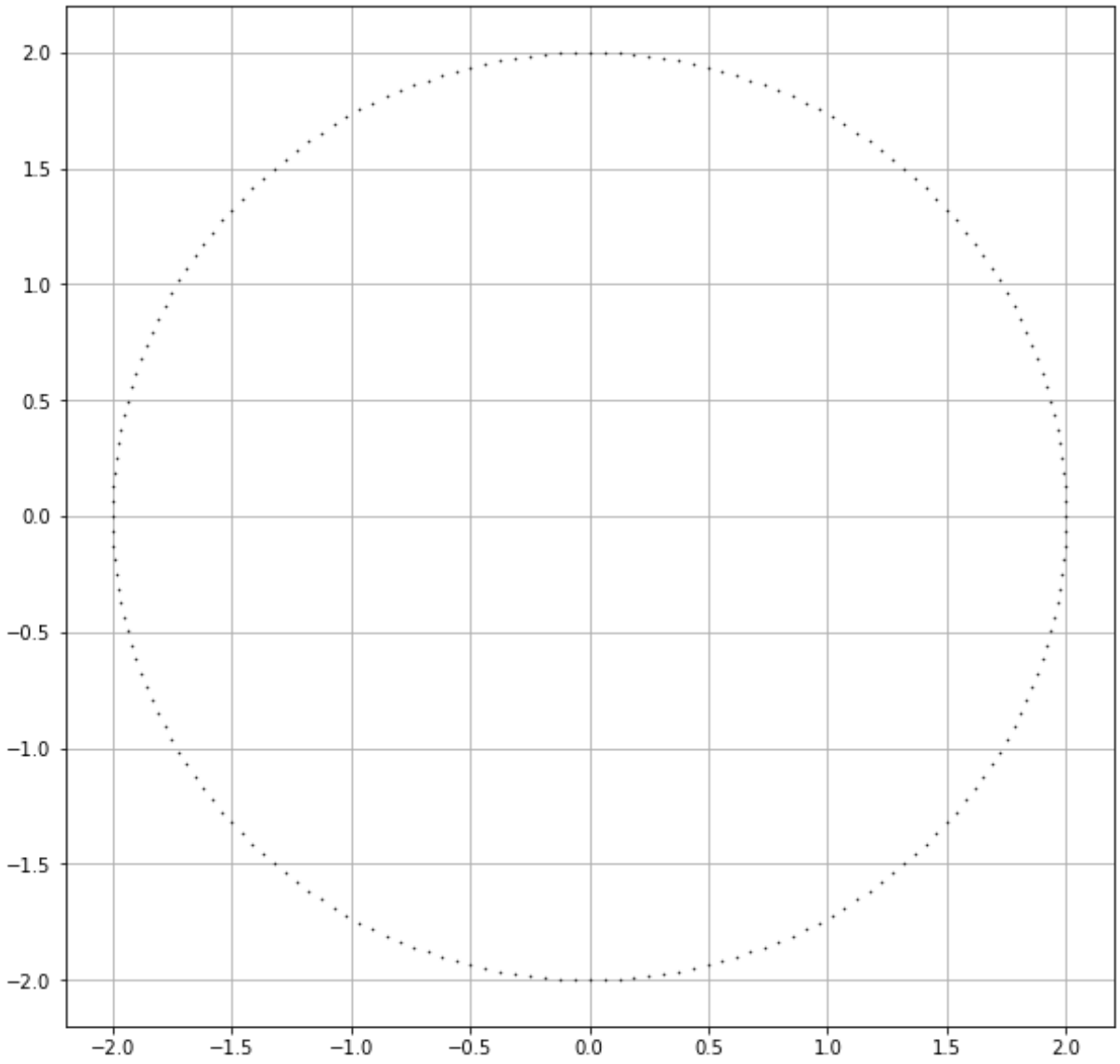
## 4.1 Tracer une liste de nombres complexes

La fonction `plot_complex` prend en paramètre une liste de nombre complexes et affiche les points.

Entrée [43]:
```
def plot_complex(zs):
    xs = [z.real for z in zs]
    ys = [z.imag for z in zs]
    plt.plot(xs, ys, '.k', markersize=1)
```

Voici par exemple le cercle de centre $O$ et de rayon $|c|$.

Entrée [44]:
```
c = 2
N = 200
thetas = [2 * k * math.pi / N for k in range(N)]
zs = [abs(c) * cmath.exp(1j * theta) for theta in thetas]
plot_complex(zs)
plt.grid()
```




## 4.2 Préimages

Soit $z_0 \in K_c$, où $|c| \geq 2$. Nous avons vu il y a de cela bien longtemps que $|z_0| \leq |c|$. L'ensemble $K_c$ est donc inclus dans le disque limité par le cercle que nous venons de dessiner. Mais nous avons aussi $|z_1| \leq |c|$. Plus mathématqiuement, si nous notons $\mathcal{D} = \{z \in \mathbb{C}, |z| \leq |c|\}$ alors

$$\forall z \in K_c, f_c(z) \in \mathcal{D}$$

ou encore

$$K_c \subset f_c^{-1}(\mathcal{D})$$

Quel est l'ensemble $f_c^{-1}(\mathcal{D})$ ? Pour des raisons d'efficacité, dessinons l'ensemble des nombres complexes $z_0$ tels que $|z_1| = |c|$, c'est à dire $f_c^{-1}(\mathcal{C})$ où $\mathcal{C}$ est le cercle de centre $O$ et de rayon $|c|$, c'est à dire le **bord** de $\mathcal{D}$. Nous avons

$$z_1 = f_c(z_0) = z_0^2 + c$$

et donc

$$z_0 = \pm\sqrt{z_1 - c}$$

où $\sqrt{z}$ désigne *une* racine carrée du nombre complexe $z$.

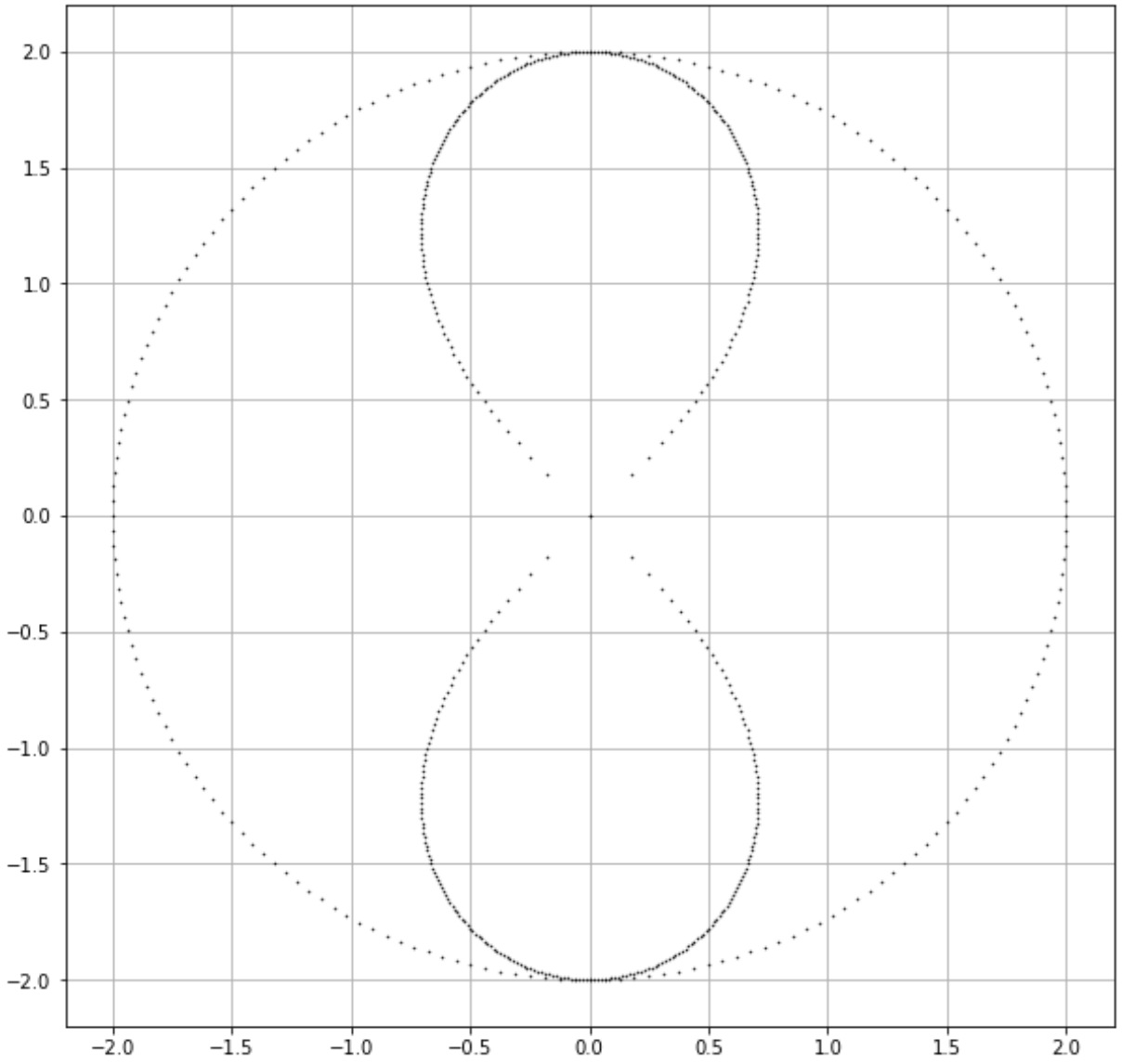
La fonction `preimage` prend en paramètre une liste `zs` de nombres complexes et un nombre complexe $c$ et renvoie la liste des $\pm\sqrt{z - c}$ pour $z \in zs$.

Entrée [45]:
```
def preimage(zs, c):
    ws = [cmath.sqrt(z - c) for z in zs]
    return ws + [-w for w in ws]
```

Calculons la préimage de $\mathcal{C}$ et affichons-la.

Entrée [46]:

```
ws = preimage(zs, c)
plot_complex(ws)
plot_complex(zs)
plt.grid()
```



On obtient une courbe en forme de "huit".

Petite remarque : soit $z \in \mathbb{C}$. Supposons que $|z| > c$. Alors

$$|z^2 + c| \geq |z|^2 - |c| > |c|^2 - |c| = |c|(|c| - 1) \geq |c|$$

La dernière inégalité provenant du fait que $|c| - 1 \geq 1$. Ainsi,

$$|z| > |c| \Rightarrow |f_c(z)| > |c|$$

Si l'on contrapose, on obtient

$$|f_c(z)| \leq |c| \Rightarrow |z| \leq |c|$$

Admettons que l'intérieur (resp. l'extérieur) de $\mathcal{C}$ est l'image de l'intérieur (resp. l'extérieur) du 8. C'est humainement évident, mais c'est **très** loin de l'être mathématiquement. Nos inégalités prouvent que le huit est inclus dans le disque. Et l'ensemble $K_c$ est inclus dans le huit.

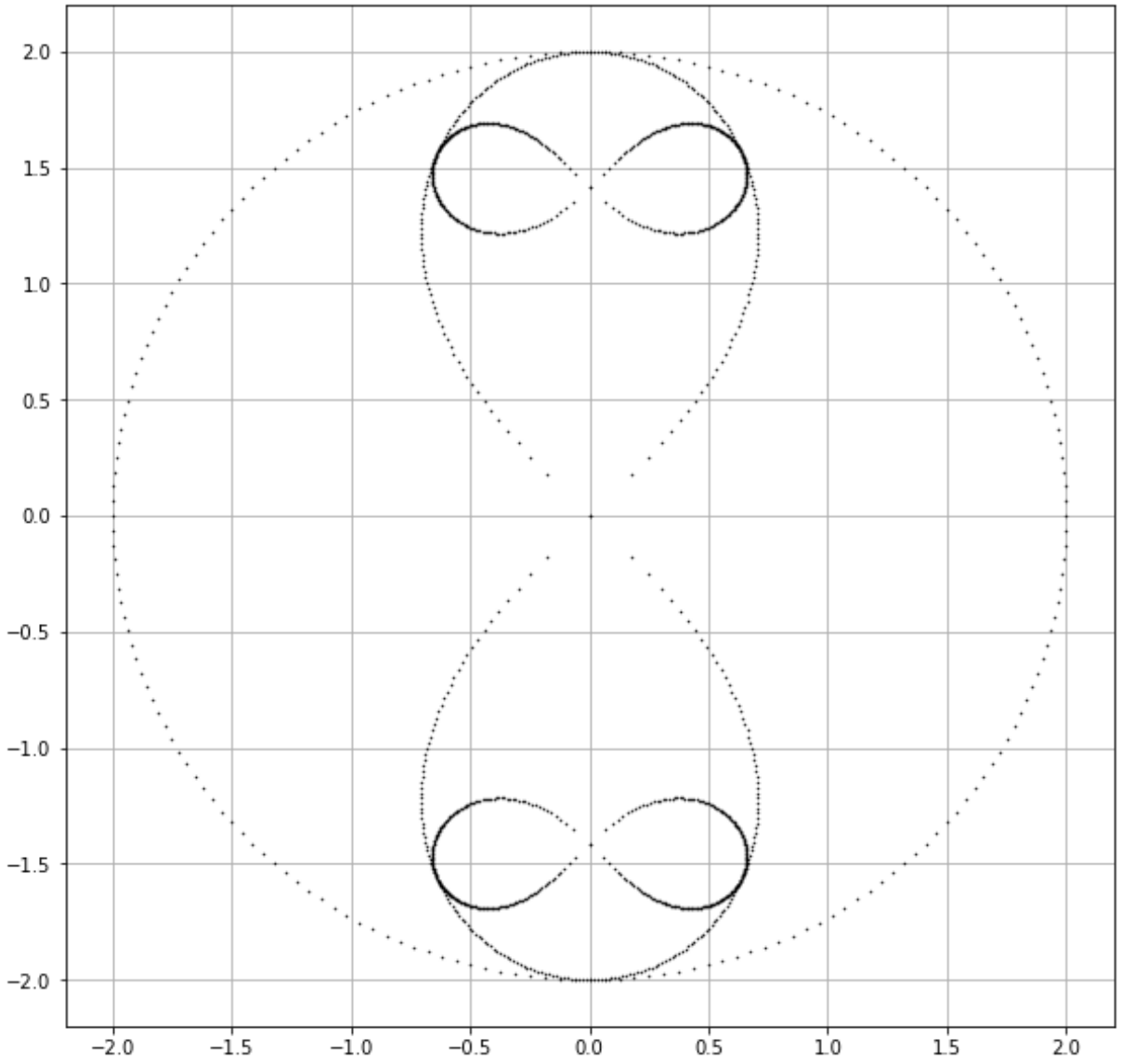
## 4.3 Itérons

Euh, oui, mais ce que nous venons de raconter pour $z_1$ est aussi valable pour $z_2$ !

$$K_c \subset f_c^{-2}(\mathcal{D})$$

Si nous calculons la préimage du 8 nous obtenons quoi ? Vite, un dessin.

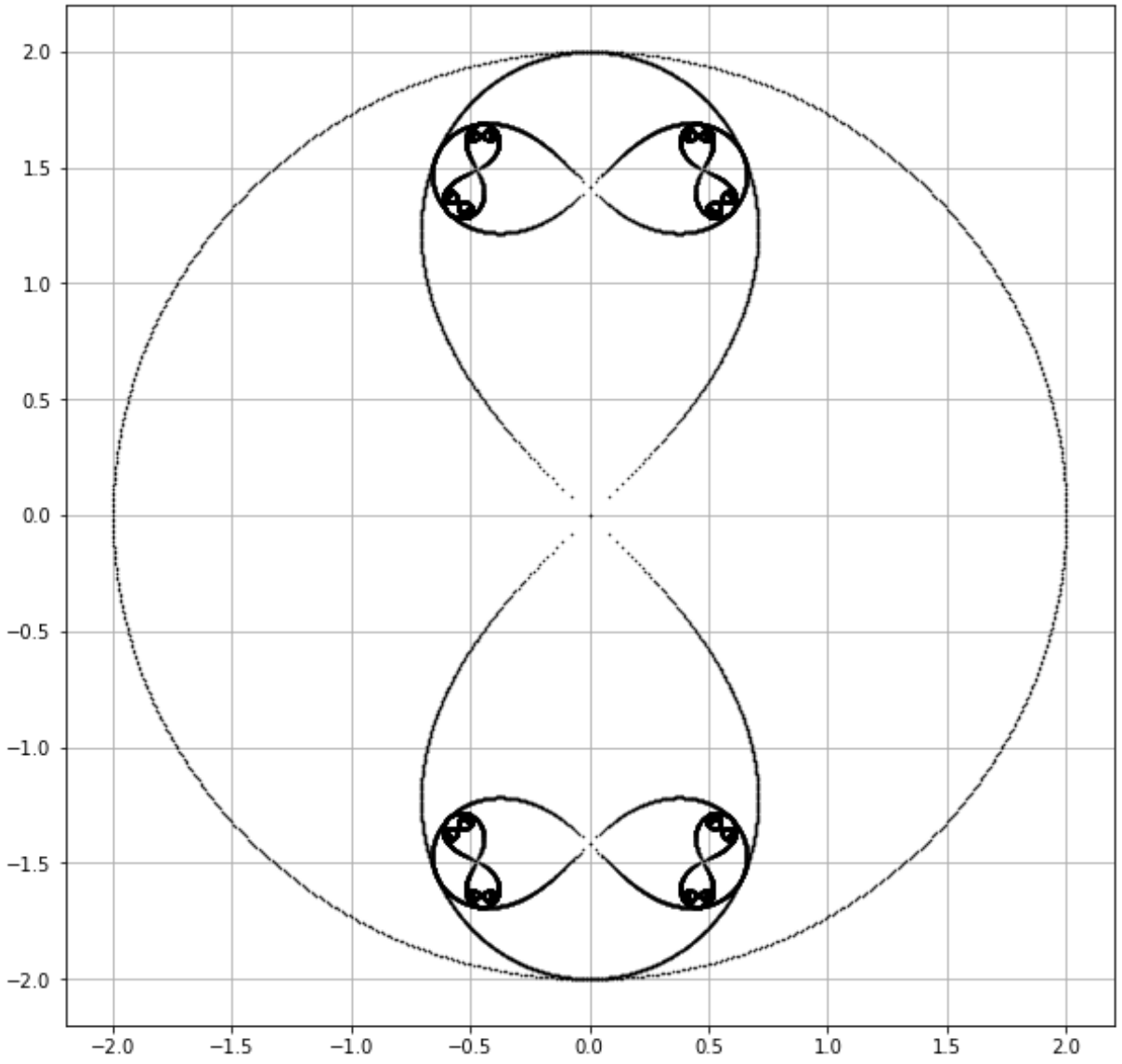
Entrée [47]:
```
ws1 = preimage(ws, c)
plot_complex(ws1)
plot_complex(ws)
plot_complex(zs)
plt.grid()
```




Nous obtenons deux "huits". Et, toujours en admettant que l'intérieur est envoyé sur l'intérieur, $K_c$ est inclus dans la réunion des 4 petites boucles. Remarquez que nos nouveaux "huits" sont plus petits que le premier, et inclus dans celui-ci (même calcul que ci-dessus). Essayez avec d'autres valeurs de $c$, et vous verrez que si $|c| \geq 2$ cette propriété se vérifie à chaque fois.

Et si on regarde $z_3$ ? on obtient 4 "huits". Et pour $z_4$ on obtient 8 "huits". Et commme toujours $K_c$ est inclus dans les boucles de ces huits.

Entrée [48]:
```
c = 2
N = 1000
Niter = 5
thetas = [2 * k * math.pi / N for k in range(N)]
zs = [max(2, abs(c)) * cmath.exp(1j * theta) for theta in thetas]
plot_complex(zs)
for k in range(Niter):
    zs = preimage(zs, c)
    plot_complex(zs)
plt.grid()
```




**Exercice** : Réévaluez la cellule ci-dessus avec d'autres valeurs de $c$, pas forcément de module supérieur à 2. Valeurs conseillées : $2.5$, $-2$, et bien sûr $0$, $-1$ et $i$. Augmentez la valeur de `Niter` à 10 (attention, à chaque itération il y deux fois plus de points à tracer !).

1. Qu'observez-vous ?
2. Pouvez-vous **montrer** certaines des choses que vous observez ?

## 4.4 Un peu de théorie

Soit $c \in \mathbb{C}$ tel que $|c| \geq 2$. Définissons par récurrence la suite $(\mathcal{D}_k)$ de parties de $\mathbb{C}$ par :

- $\mathcal{D}_0 = \mathcal{D}$, le disque de centre $O$ et de rayon $|c|$.
- Pour tout entier $k \geq 0$, $\mathcal{D}_{k+1} = f_c^{-1}(\mathcal{D}_k)$.

**Proposition** : $K_c = \bigcap_{k=0}^{\infty} D_k$.

**Démonstration** : Rappelons-nous que nous avons montré que $z_0 \in J_c$ si et seulement si $\forall k \geq 0, |z_k| \leq |c|$. On montre par une récurrence facile que pour tout entier $k$, $z_0 \in \mathcal{D}_k$ si et seulement si $f_c^k(z_0) \in \mathcal{D}$, c'est à dire $z_k \in \mathcal{D}$. Ainsi, $z_0$ est dans l'intersection des $\mathcal{D}_k$ si et seulement si pour tout $k \in \mathbb{N}$, $z_k \in \mathcal{D}$, c'est à dire $|z_k| \leq |c|$. Mais ceci équivaut à $z_0 \in K_c$.

Pour les adeptes de topologie, $K_c$ est une intersection décroissante d'ensembles compacts et non vides. **Donc** $K_c$ est lui-même un ensemble compact **et non vide**. Cela dit, on a vraiment l'impression qu'il ne reste pas grand chose dans $K_c$. Je n'entrerai pas dans les détails ici, mais (sauf dans un cas dont nous parlons au paragraphe suivant), $K_c$ est un ensemble **totalement déconnecté** : ses composantes connexes sont des singletons. Nous avons là ce que l'on appelle un **ensemble de Cantor**, et cet ensemble est très loin d'être vide. En fait il est infini et non dénombrable.

## 4.5 Le cas $c = -2$

Bien que l'on ait $|-2| = 2$, le cas $c = -2$ est un peu à part. Quel est donc l'ensemble $K_{-2}$ ? Soit $z_0 \in \mathbb{C}$. La récurrence qui nous intéresse est donc, pour tout entier $n$, donnée par $z_{n+1} = z_n^2 - 2$. Posons $w_n = 2 - 4z_n$. On a donc

$$2 - 4w_{n+1} = (2 - 4w_n)^2 - 2 = 2 - 16w_n + 16w_n^2$$

ou encore

$$w_{n+1} = 4w_n(1 - w_n)$$

Posons, pour tout $z \in \mathbb{C}$,

$$\sin z = \frac{e^{iz} - e^{-iz}}{2i} \quad \text{et} \quad \cos z = \frac{e^{iz} + e^{-iz}}{2}$$

On définit ainsi des prolongements à $\mathbb{C}$ des fonctions usuelles "sinus" et "cosinus". On vérifie facilement que toutes les formules de trigonométrie standard restent valides.

**Exercice** : Que valent $\sin(ix)$ et $\cos(ix)$ lorsque $x$ est un réel ? réponse : cela a à voir avec les fonctions sinus et cosinus **hyperboliques**.

**Exercice** : Montrer que $\sin : \mathbb{C} \to \mathbb{C}$ est surjective.

Posons $w_0 = \sin^2 \alpha$, où $\alpha \in \mathbb{C}$. Si vous avez fait l'exercice, vous savez que c'est possible : il suffit de prendre pour $\alpha$ un antécédent par $\sin$ d'une racine carrée de $w_0$. On a alors

$$w_1 = 4w_0(1 - w_0) = 4\sin^2\alpha(1 - \sin^2\alpha) = 4\sin^2\alpha\cos^2\alpha = \sin^2(2\alpha)$$

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

$$w_n = \sin^2(2^n \alpha)$$

Posons $\alpha = p + iq$, où $p, q \in \mathbb{R}$. On a

$$\sin(2^n \alpha) = \sin(2^n (p + iq)) = \sin(2^n p)\cos(i2^n q) + \cos(2^n p)\sin(i2^n q) = \sin(2^n p)\cosh(2$$

De là,

$$|w_n| = |\sin 2^n \alpha|^2 = \sin^2(2^n p)\cosh^2(2^n q) + \cos^2(2^n p)\sinh^2(2^n q)$$

En remplaçant $\cosh^2(2^n q)$ par $1 + \sinh^2(2^n q)$ on obtient finalement

$$|w_n| = \sin^2(2^n p) + \sinh^2(2^n q)$$

Clairement, la suite $(w_n)$ est bornée si et seulement si $q = 0$, c'est à dire $\alpha \in \mathbb{R}$. Cela équivaut à $w_0 \in \mathbb{R}$ et $0 \leq w_0 \leq 1$. Comme, bien entendu $(w_n)$ est bornée si et seulement si $(z_n)$ est bornée, on en déduit que $z_0 \in K_{-2}$ si et seulement si $z_0 \in \mathbb{R}$ et $0 \leq 2 - 4z_0 \leq 1$, ou encore $-2 \leq z_0 \leq 2$. Finalement :

$$K_{-2} = [-2, 2]$$

Et la frontière de $K_{-2}$ est

$$J_{-2} = K_{-2} = [-2, 2]$$

### 4.5 Et si $|c| < 2$ ?

Quand, précisément, $K_c$ est-il un ensemble de Cantor ? Lorsque $|c| \geq 2$ et $c \neq -2$, c'est le cas, nous venons de nous en faire une petite idée pour $c = 2$. Et sinon ? On peut montrer le résultat suivant :

**Proposition** :

- Si $0 \notin K_c$ alors $K_c$ est un ensemble de Cantor.
- Si $0 \in K_c$ alors $K_c$ n'est pas un ensemble de Cantor. Mieux, $K_c$ et $J_c$ sont alors **connexes**.

Quels sont les nombres complexes $c$ tels que $0 \in K_c$ ? Eh bien ces nombres forment un ensemble fort compliqué qui est appelé **l'ensemble de Mandelbrot**. Mais ceci est une autre histoire, voir le notebook à ce sujet.

## 5. Le jeu du chaos

### 5.1 Rétro-itérations de $f_c$

Pour terminer, jouons à un petit jeu. Soit $z \in \mathbb{C}$. Quels sont les $w \in \mathbb{C}$ tels que $f_c(w) = z$ ? Facile, puisque $w^2 + c = z$ si et seulement si

$$w = \varepsilon\sqrt{z - c}$$

où le symbole "racine carrée" désigne **une** racine carrée du nombre complexe et $\varepsilon = \pm 1$.

La fonction `retro_iter` renvoie un antécédent au hasard de $z$ par la fonction $f_c$.

Entrée [49]:
```
def retro_iter(z, c):
    eps = 2 * random.randint(0, 1) - 1
    return eps * cmath.sqrt(z - c)
```

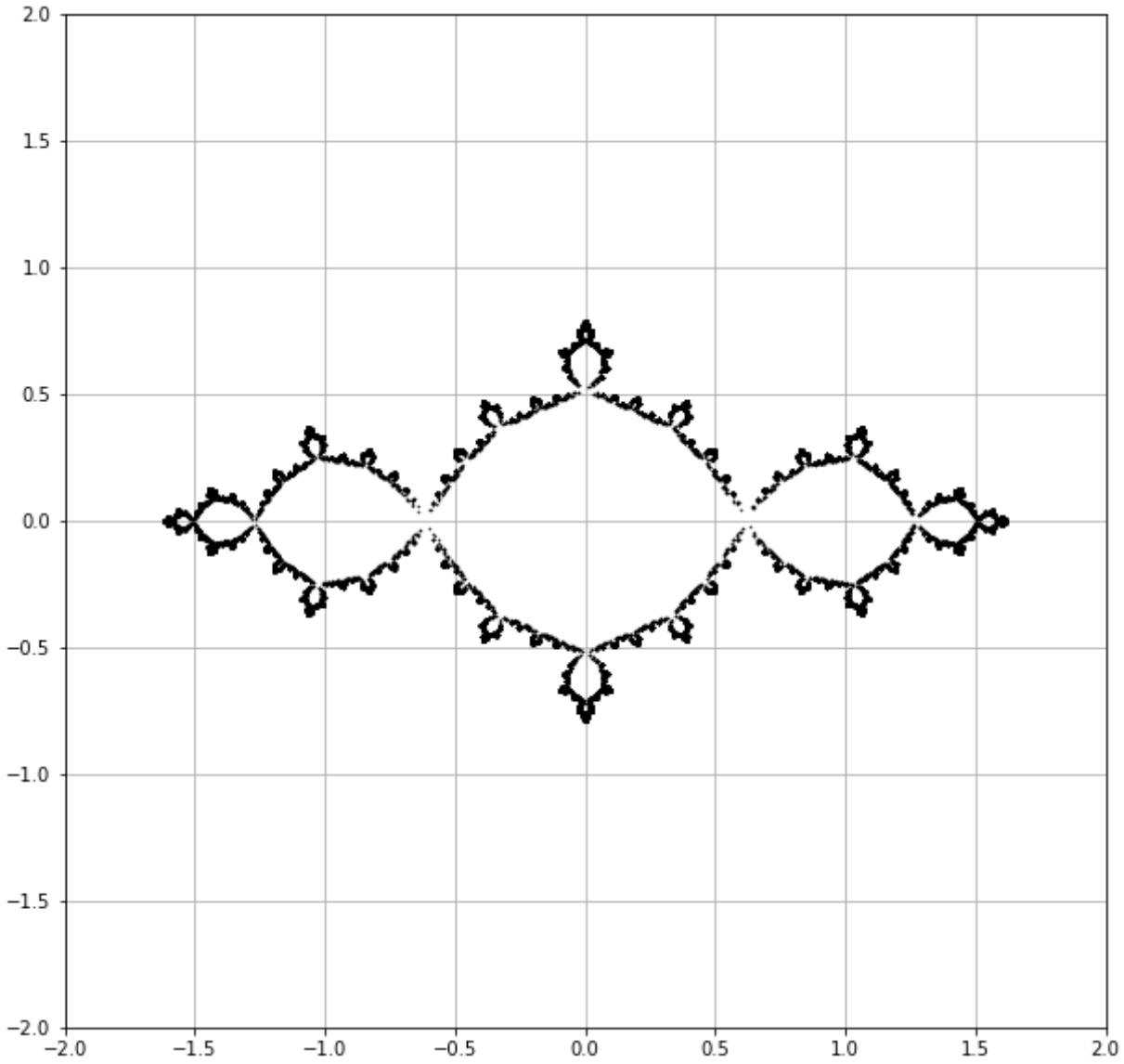
Bon, et le jeu dans tout ça ? Prenons $z$ au hasard et itérons la fonction `retro_iter`. Puis dessinons les points obtenus. Le bon sens suggère que l'on obtiendra n'importe quoi vu qu'on tire les $\varepsilon$ au hasard ...

La fonction `julia_retro` fait le travail. Elle tire un nombre complexe "au hasard", effectue des itérations grâce à `retro_iter` et stocke les valeurs obtenues (les 100 premières valeurs sont laissées de côté). Puis elle affiche les points stockés.

Entrée [50]:
```
def julia_retro(c, niter):
    xs = []
    ys = []
    z = random.uniform(0, 1) + 1j * random.uniform(0, 1)
    for k in range(100): z = retro_iter(z, c)
    for k in range(niter):
        z = retro_iter(z, c)
        xs.append(z.real)
        ys.append(z.imag)
    plt.xlim(-2, 2)
    plt.ylim(-2, 2)
    plt.plot(xs, ys, '.k', markersize=1)
    plt.grid()
```

On essaie ?

Entrée [51]:
```
julia_retro(-1, 100000)
```




Bigre !!! Comment est-ce possible ???

## 5.2 Quelques explications

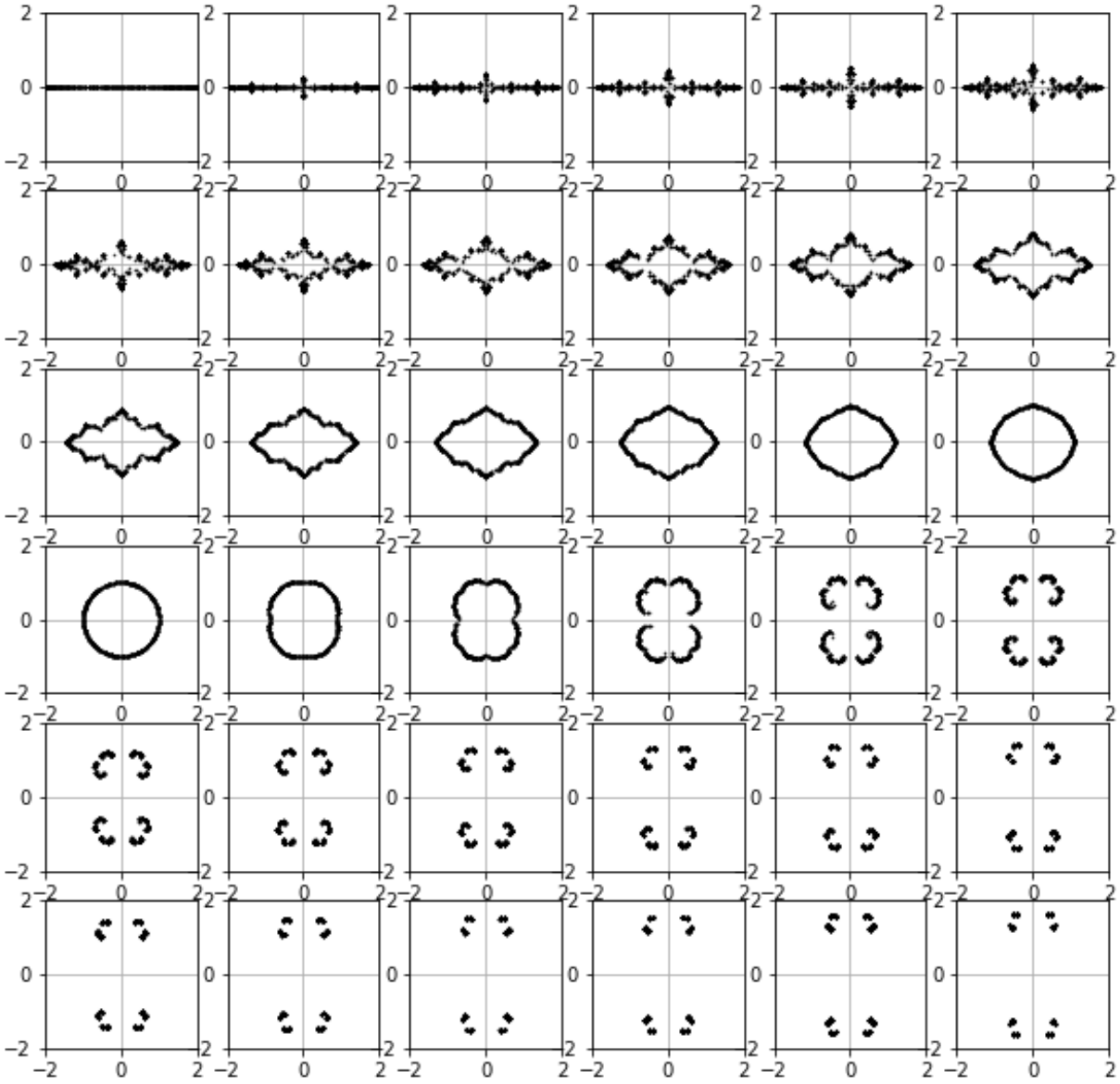
L'ensemble $J_c$ est un **attracteur** de la suite $(z_n)$. J'imagine que tous mes lecteurs ont entendu parler d'une suite convergente. Ou d'une suite telle que $(z_{2n})$ tend vers une limite et $(z_{2n+1})$ tend vers une autre limite. De façon générale, un nombre complexe $w$ est un **point d'accumulation** de la suite $(z_n)$ s'il existe une suite extraite de $(z_n)$ qui converge vers $w$. Dit très vite et très vaguement, l'attracteur de la suite est l'ensemble de ses points d'accumulation. Ici, le $\varepsilon$ choisi au hasard complique la théorie, mais disons que l'attracteur de la suite $z_{n+1} = \texttt{retro\_iter}\ (z_n, c)$ est l'ensemble de Julia $J_c$.

*Comme j'imagine qu'il n'y a plus personne pour me lire, je n'entre pas dans les détails :-).*

## 5.3 Pour terminer

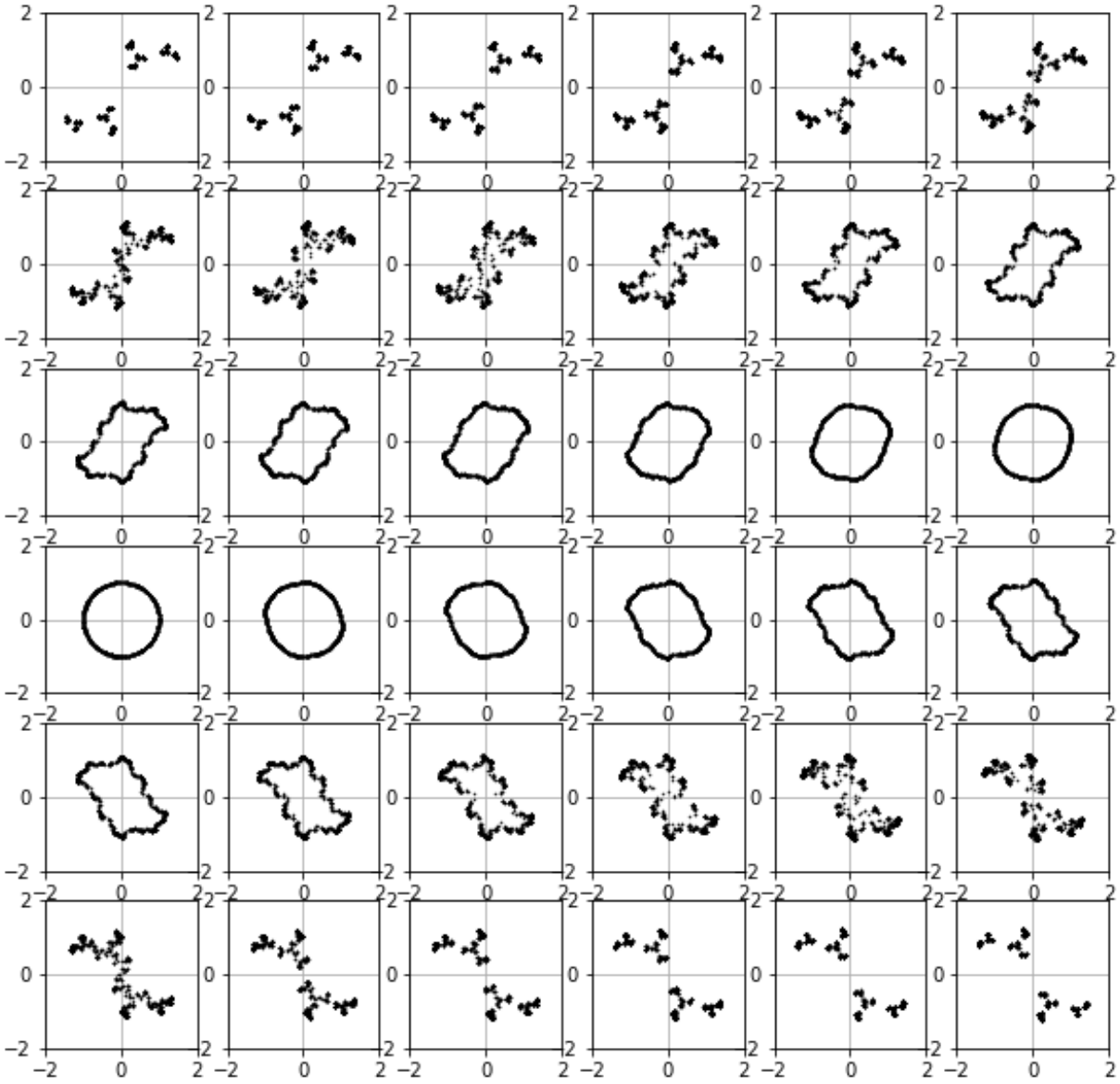
L'avantage de cette méthode est de pouvoir visualiser rapidement des ensembles de Julia. Voici pour terminer quelques tableaux de graphiques. Commençons par les ensembles $J_c$ pour $c$ réel entre $-2$ et 2.

Entrée [52]:
```
plt.figure(1)
for k in range(36):
    plt.subplot(6, 6, 1 + k)
    julia_retro(-2 + 4 * k / 36, 1000)
```



Même expérience, mais $c$ est imaginaire pur, et va de $-\frac{3}{2}i$ à $\frac{3}{2}i$.

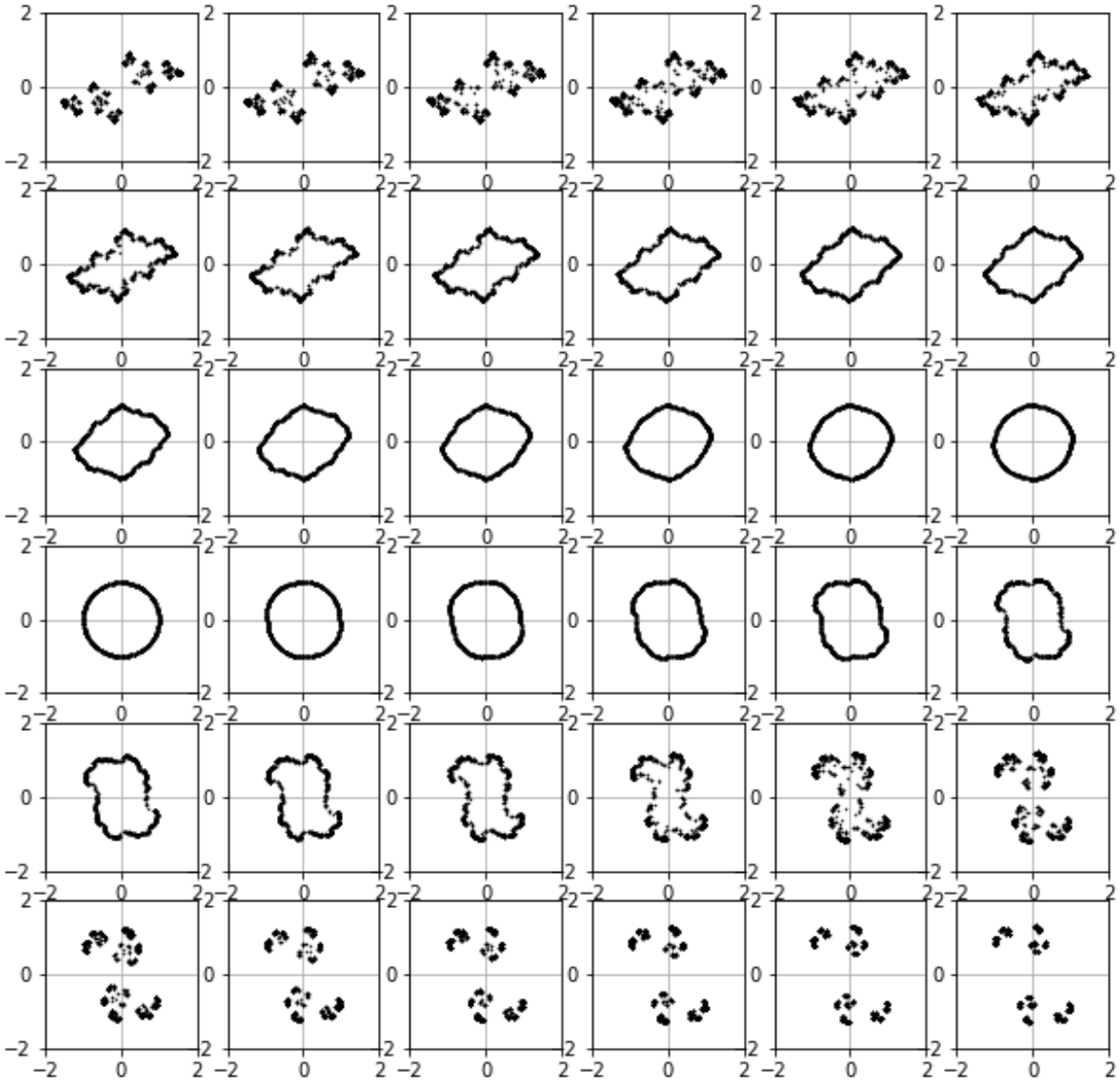
Entrée [53]:
```
plt.figure(1)
for k in range(36):
    plt.subplot(6, 6, 1 + k)
    julia_retro((-1.5 + 3 * k / 36) * 1j, 1000)
```



Et pour finir, faisons varier $c$ "en diagonale" ...

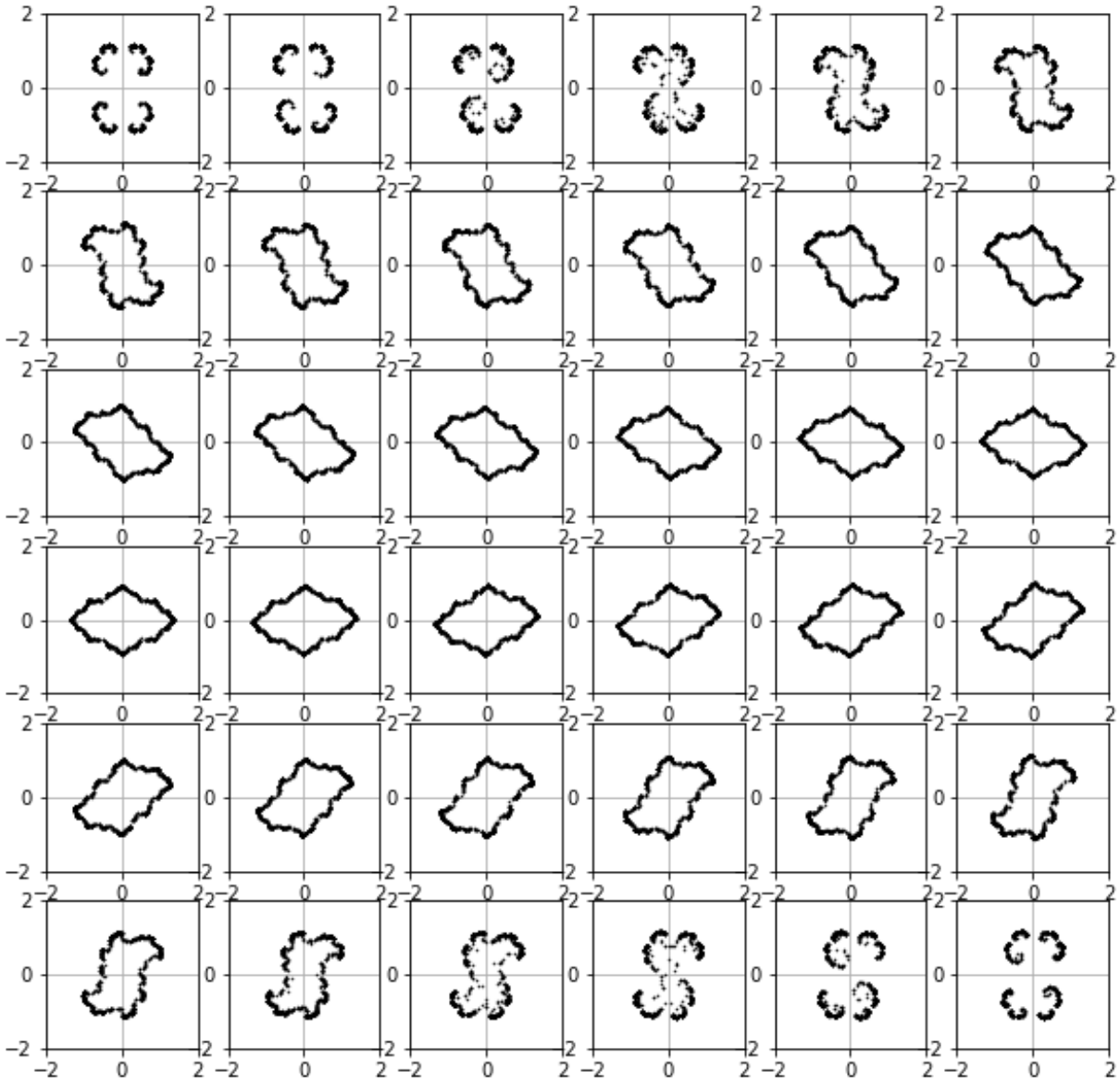
Entrée [54]:

```
plt.figure(1)
for k in range(36):
    plt.subplot(6, 6, 1 + k)
    julia_retro((-1.5 + 3 * k / 36) * (0.5 + 0.5 * 1j), 1000)
```



Et sur le cercle de centre $O$ et de rayon $\frac{1}{2}$ !

Entrée [55]:
```
plt.figure(1)
for k in range(36):
    plt.subplot(6, 6, 1 + k)
    c = 0.5 * cmath.exp(2 * 1j * k * math.pi / 36)
    julia_retro(c, 1000)
```



Tentez d'autres expériences ...

## 6. Et ensuite ?

Il y a un nombre incalculable de choses dont nous n'avons pas parlé ici. Pour n'en citer que trois :

- La **dynamique** des ensembles de Julia et ce que l'on appelle la **décomposition binaire** de ces ensembles.
- L'utilisation du potentiel de Douady pour estimer la distance d'un point à $J_c$.
- Le lien entre les ensembles de Julia et l'ensemble de Mandelbrot.

Mais ce notebook est déjà terriblement long, alors arrêtons-nous là :-).

Entrée [ ]: