La courbe de Von Koch

Marc Lorenzi - 2 mai 2018

In [1]:
```
import matplotlib.pyplot as plt
%matplotlib inline
from cmath import *
import random
```

In [2]:
```
plt.rcParams['figure.figsize'] = (8, 8)
```

1. Similitudes du plan complexe

Une similitude (directe) est une application $f : \mathbb{C} \to \mathbb{C}$ définie par $f(z) = az + b$ où $a, b \in \mathbb{C}$ et $a \neq 0$. Certaines similitudes très particulières permettent, par composition, de fabriquer toutes les similitudes. Ce sont les **rotations**, les **homothéties** et les **translations**.

1.1 Les rotations

Une rotation f est caractérisée par son centre $\omega \in \mathbb{C}$ et son angle $\theta \in \mathbb{R}$ (on impose aussi parfois que $\theta \notin 2\pi\mathbb{Z}$). Elle est définie par

$$f(z) = \omega + e^{i\theta}(z - \omega)$$

In [3]:
```
def rotation(omega, theta, z):
    return omega + exp(1j * theta) * (z - omega)
```

In [4]:
```
rotation(1, pi / 4, 2)
```

Out[4]:
```
(1.7071067811865475+0.7071067811865475j)
```

1.2 Les homothéties

Une homothétie f est caractérisée par son centre $\omega \in \mathbb{C}$ et son rapport $\mu \in \mathbb{R}_+^*$. Elle est définie par

$$f(z) = \omega + \mu(z - \omega)$$

In [5]:
```
def homothetie(omega, mu, z):
    return omega + mu * (z - omega)
```

In [6]:
```
homothetie(1, 3, 1+1j)
```

Out[6]:
```
(1+3j)
```

1.3 Les translations

Une translation f est caractérisée par un nombre complexe t. Elle est définie par

$$f(z) = z + t$$

In [7]:
```
def translation(t, z): return z + t
```

In [8]:
```
translation(1+1j, 2+3j)
```

Out[8]:
```
(3+4j)
```

2. Dessiner une liste de nombres complexes

`matplotlib` n'est pas très amatrice de complexes. Écrivons donc une fonction qui nous permet de tracer une liste de points donnés par leurs affixes.

La fonction `vers_points` prend en paramètre une liste `zs` de complexes. Elle renvoie un couple `(xs, ys)` où `xs` et `ys` sont respectivement les listes des parties réelles et imaginaires des éléments de `zs` .

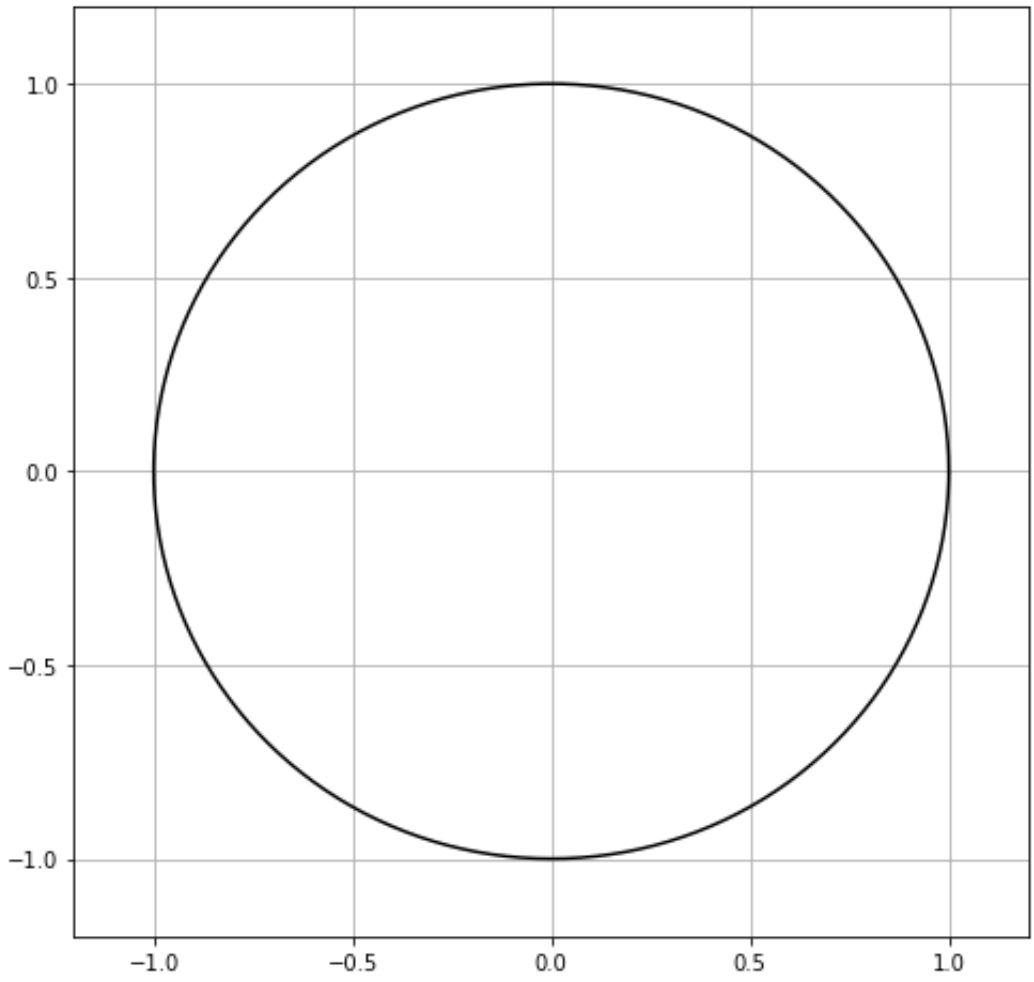
In [9]:
```
def vers_points(zs):
    xs = [z.real for z in zs]
    ys = [z.imag for z in zs]
    return xs, ys
```

La fonction `plotC` prend en paramètre une liste de nombres complexes et affiche les points reliés dans le plan complexe. Il y a également un paramètre `bornes` qui permet d'affiner la zone de tracé.

In [10]:
```
def plotC(zs, bornes):
    xs, ys = vers_points(zs)
    plt.plot(xs, ys, 'k')
    plt.axis(bornes)
    plt.grid()
    plt.show()
```

À titre d'exemple, traçons ... le cercle unité ? Pour être précis, on trace les racines millièmes de l'unité et on les relie :-).

In [11]:
```
zs = [exp(2j * k * pi / 1000) for k in range(1000)]
plotC(zs, [-1.2, 1.2, -1.2, 1.2])
```



Bon, ça fonctionne. Oubliés les x et les y, les sinus, les cosinus. Vive $\mathbb{C}$! Dorénavant, nous identifierons les points du plan et les nombres complexes. Quand je dirai "soit A un point", je penserai souvent "soit A un nombre complexe".

3. La courbe de Von Koch

3.1 Première étape

On part d'un segment $S = [A, B]$ où A et B sont deux points du plan. Notons P le point situé au tiers du segment et Q le point situé aux deux tiers du segment. On définit 4 nouveaux segments.

- $S_0 = [A, P]$.
- Soit U l'image de Q par la rotation de centre P et d'angle $\frac{\pi}{3}$. On pose $S_1 = [P, U]$ et $S_2 = [U, Q]$.
- Enfin, $S_3 = [Q, B]$.

On pose ensuite $S' = S_0 \bigcup S_1 \bigcup S_2 \bigcup S_3$. Nous appellerons S' le **transformé de Von Koch** de S.

La fonction `transforme` prend les points A et B en paramètres. Elle renvoie la liste $[A, P, U, Q, B]$.

In [12]:
```
def barycentre(A, B, t):
    return (1 - t) * A + t * B
```

In [13]:
```
def transforme(S):
    A, B = S[0], S[1]
    P = barycentre(A, B, 1/3)
    Q = barycentre(A, B, 2/3)
    U = rotation(P, pi/3, Q)
    return [A, P, U, Q, B]
```

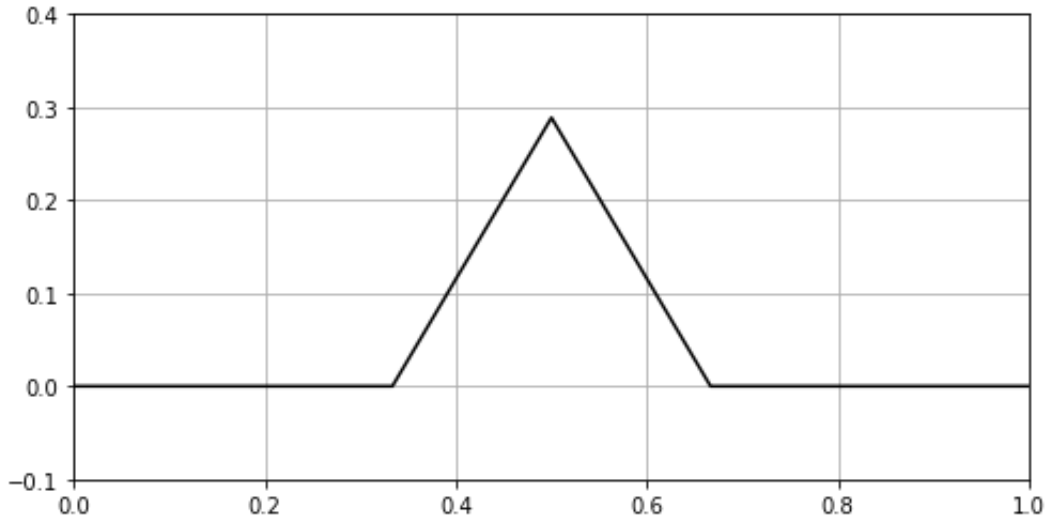
In [14]:
```
T = transforme([0, 1])
print(T)
```

```
[0, 0.3333333333333333, (0.5+0.28867513459481287j), 0.6666666666666666, 1]
```

Voici le dessin. Il vaut mieux que le discours ci-dessus.

In [15]:
```
plt.rcParams['figure.figsize'] = (8, 4)
```

In [16]:
```
plotC(T, [0, 1, -0.1, 0.4])
```



3.2 Deuxième étape

Définissons maintenant la transformée d'une réunion de segments comme la réunion des transformées des chacun des segments. La fonction ci-dessous prend en paramètre une liste de nombres complexes. Deux éléments successifs de la liste son censés représenter un segment. La fonction renvoie la transformée de Von Koch de la liste de segments.

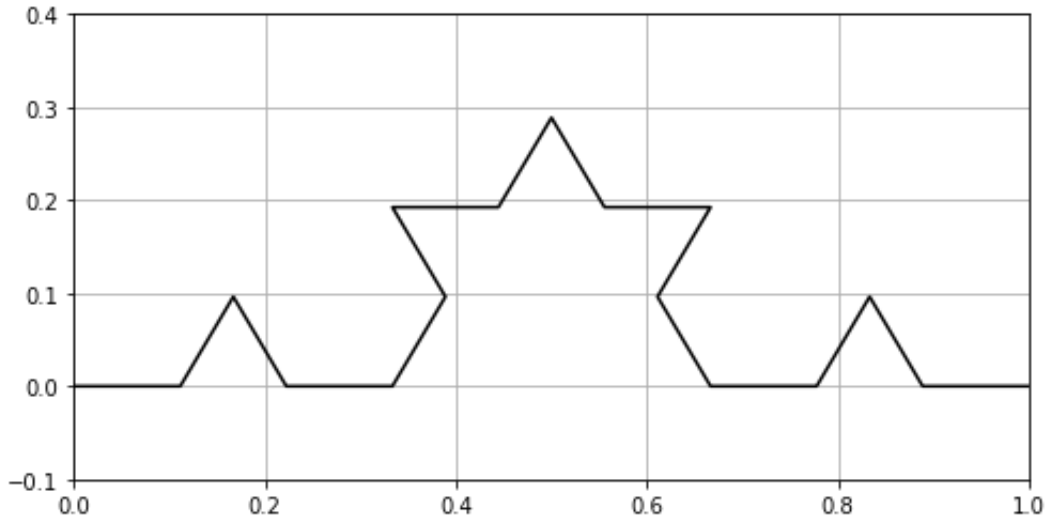
Remarque : les concaténations de listes ci-dessous ne sont pas très efficaces, mais elles seront suffisantes pour nos besoins.

In [17]:
```
def transforme2(zs):
    s = []
    n = len(zs)
    for k in range(n -1):
        s1 = transforme([zs[k], zs[k + 1]])
        s = s + s1[:-1]
    s.append(zs[n - 1])
    return s
```

In [18]:
```
T = transforme2(transforme([0, 1]))
print(T)
```

```
[0, 0.1111111111111111, (0.1666666666666669+0.09622504486493762j),
0.2222222222222222, 0.3333333333333333, (0.3888888888888889+0.096225
04486493762j), (0.33333333333333337+0.1924500897298752j), (0.4444444
444444444+0.19245008972987523j), (0.5+0.28867513459481287j), (0.5555
555555555556+0.19245008972987526j), (0.6666666666666667+0.1924500897
2987526j), (0.6111111111111112+0.09622504486493763j), 0.666666666666
6666, 0.7777777777777778, (0.8333333333333334+0.09622504486493756j),
0.8888888888888888, 1]
```

In [19]:
```
plotC(T, [0, 1, -0.1, 0.4])
```



On a compris ... on va recommencer.

3.3 On itère

Soient A et B deux points du plan. On pose $K_0 = [A, B]$ et, pour tout entier $n \in \mathbb{N}$, $K_{n+1} = \mathcal{T}(K_n)$, où $\mathcal{T}$ est la transformée de Von Koch.

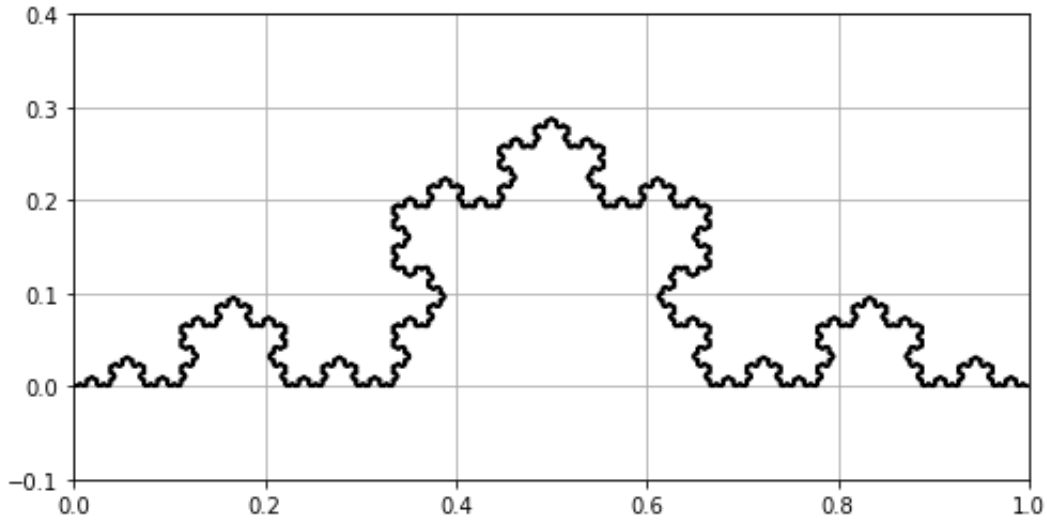
La fonction ci-dessous prend en paramètres un entier n et les deux points A et B. Elle renvoie la liste des extrémités des segments de l'ensemble K_n.

In [20]:
```
def vonkoch(n, A, B):
    S = [A, B]
    for k in range(n):
        S = transforme2(S)
    return S
```

In [21]:
```
T = vonkoch(2, 0, 1)
print(T)
```

```
[0, 0.1111111111111111, (0.1666666666666669+0.09622504486493762j),
0.2222222222222222, 0.3333333333333333, (0.3888888888888889+0.096225
04486493762j), (0.33333333333333337+0.1924500897298752j), (0.4444444
444444444+0.19245008972987523j), (0.5+0.28867513459481287j), (0.5555
555555555556+0.19245008972987526j), (0.6666666666666667+0.1924500897
2987526j), (0.6111111111111112+0.09622504486493763j), 0.666666666666
6666, 0.7777777777777778, (0.8333333333333334+0.09622504486493756j),
0.8888888888888888, 1]
```

In [22]:
```
plotC(vonkoch(6, 0, 1), [0, 1, -0.1, 0.4])
```



3.4 Que prendre comme valeur de n ?

Prenons comme unité de longueur de nos raisonnements le pixel. Supposons que la largeur de nos graphiques soit de 1000 pixels, ce qui est évidemment très optimiste. L'appel à `vonkoch(0, 0, 1)` trace donc un segment de 1000 pixels. À chaque itération de la transformation de Von Koch, la taille des segments est divisée par 3. Ainsi, la taille des segments lors du tracé de `vonkoch(n, 0, 1)` est $\frac{1000}{3^n}$ pixels. Pour $n = 7$, par exemple, qu'obtient-on ?

In [23]:
```
1000 / 3 ** 7
```

Out[23]:
```
0.45724737082761774
```

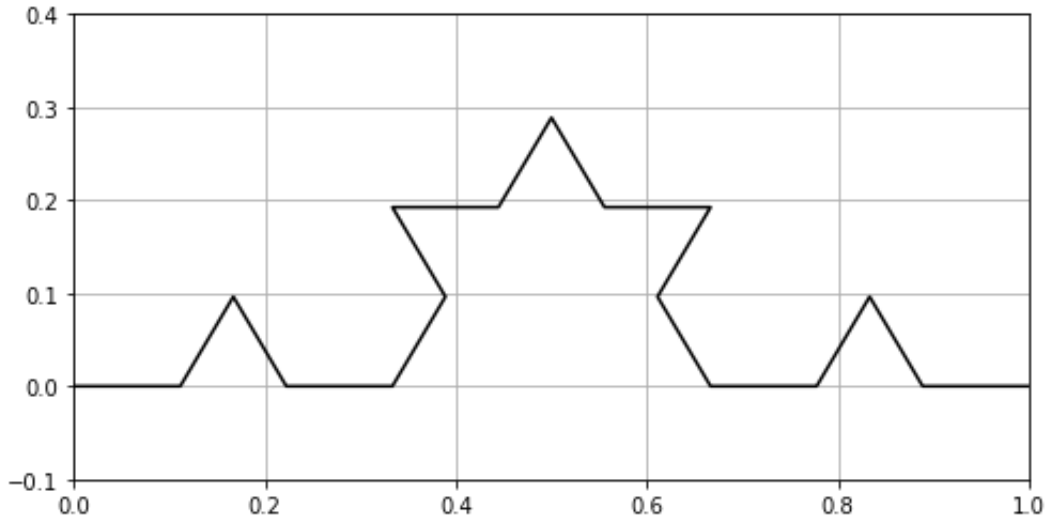
Un demi-pixel, c'est peu. Inutile de dépasser $n = 6$, cela ne se verra de toute façon pas sur le dessin.

4. Von Koch bis

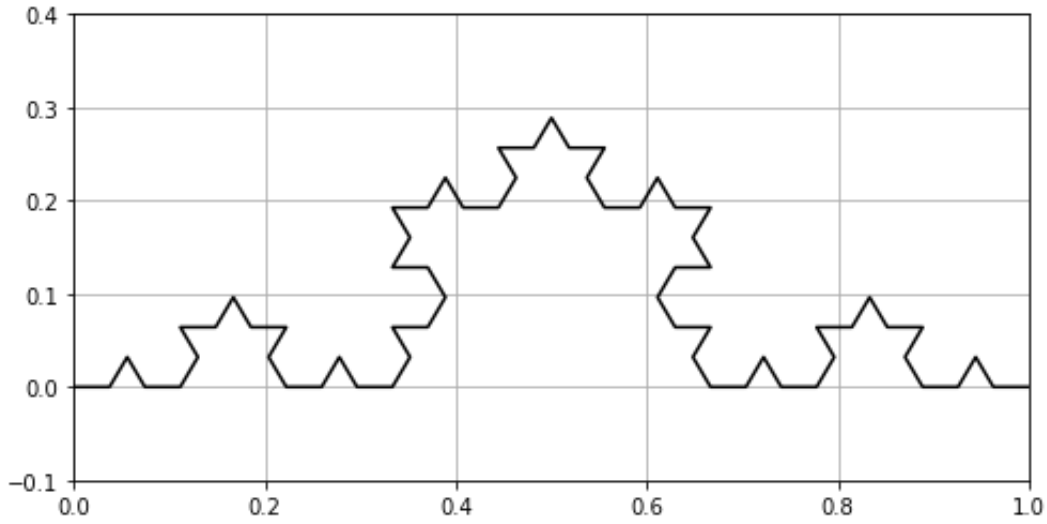
4.1 La "courbe" de Von Koch ? C'est quoi ?

Bon, tout cela est bien beau mais c'est quoi la "courbe" de Von Koch ? Nous allons réinterpréter la transformée de Von Koch. Regardons les deux figures ci-dessous.

In [24]:
```
plotC(vonkoch(2, 0, 1), [0, 1, -0.1, 0.4])
```



In [25]:
```
plotC(vonkoch(3, 0, 1), [0, 1, -0.1, 0.4])
```



On voit que K_3 est la réunion des images directes de K_2 par 4 similitudes que nous allons noter $s_0, \dots, s_3$.

La première (enfin la zéroième), qui nous donne la partie gauche "horizontale", est évidente. C'est l'homothétie de centre O et de rapport $\frac{1}{3}$: $s_0(z) = \frac{1}{3}z$.

Pour la partie droite "horizontale", il s'agit de l'homothétie de centre 1 et de rapport $\frac{1}{3}$: $s_3(z) = 1 + \frac{1}{3}(z-1) = \frac{2}{3} + \frac{1}{3}z$. Mais inutile de les calculer, Python fera cela pour nous :

In [26]:
```
s = 4 * [None]
s[0] = lambda z: homothetie(0, 1/3, z)
s[3] = lambda z: homothetie(1, 1/3, z)
```

Passons à la partie milieu-gauche en biais. On commence par faire une homothétie h de rapport $\frac{1}{3}$ et de centre $\frac{1}{2}$. Puis on effectue la rotation ρ de centre $\frac{1}{3}$ et d'angle $\frac{\pi}{3}$. Ainsi, $s_1 = \rho \circ h$.

Pour le morceau restant, à vous de deviner : $s_2 = \rho' \circ h$, où $\rho' = $? Cela dit la réponse est juste dessous.

In [27]:
```
s[1] = lambda z: rotation(1/3, pi/3, homothetie(1/2, 1/3, z))
s[2] = lambda z: rotation(2/3, -pi/3, homothetie(1/2, 1/3, z))
```

On a donc $K_3 = s_0(K_2) \bigcup s_1(K_2) \bigcup s_2(K_2) \bigcup s_3(K_2)$. On peut maintenant définir une transformation $\mathcal{T}$ plus générale que celle dont nous avons parlé plus haut. Pour **TOUTE** partie E du plan, posons

$$\mathcal{T}(E) = \bigcup_{k=0}^{3} s_k(E)$$

Nous disposons maintenant de la fonction $\mathcal{T} : \mathcal{P}(\mathbb{C}) \to \mathcal{P}(\mathbb{C})$. Et on a $\forall n \in \mathbb{N}, K_{n+1} = \mathcal{T}(K_n)$.

Considérons l'ensemble (que nous noterons $\mathcal{C}$) des parties *compactes* du plan complexe $\mathbb{C}$. *Compact* veut dire *fermé* et *borné*, je n'en dirai pas plus ici. On peut munir $\mathcal{C}$ d'une distance D : la **distance de Hausdorff**. On peut alors montrer qu'il existe un ensemble K tel que, lorsque $n \to \infty$, $D(K_n, K) \to 0$. En d'autres termes, $K_n \to K$ pour la distance de Hausdorff.

Définition : K est appellé la courbe de Von Koch.

Fait : Personne n'a jamais vu K.

En réalité, on a infiniment mieux que cela ...

Théorème :

1. Il existe un unique $K \in \mathcal{C}$ tel que $\mathcal{T}(K) = K$.
2. Soit $E \in \mathcal{C}$. Soit $(E_n)_{n \geq 0}$ la suite définie par récurrence par $E_0 = E$ et, pour tout $n \geq 0$, $E_{n+1} = \mathcal{T}(E_n)$. Alors $E_n \to K$ au sens de la distance de Hausdorff lorsque n tend vers l'infini.

Cela veut dire qu'au lieu de démarrer d'un segment K_0, comme nous l'avons fait, on peut démarrer de n'importe quoi. Par exemple de Bilbo le Hobbit, ou d'un yack tibétain.

Vous n'y croyez pas, c'est trop théorique, c'est du bluff ? Alors programmons, et regardons le yack se transformer en courbe de Koch.

4.2 Fonctions élémentaires

On va manipuler des matrices de taille $N \times N$ dont les éléments représentent des nombres complexes du carré $[0, 1] \times [0, 1]$. Quelques petites fonctions auxiliaires vont nous être utiles.

La fonction `pixel_vers_complexe` prend deux entiers i et j censés être entre 0 et $N - 1$. Elle renvoie le nombre complexe correspondant.

In [28]:
```
def pixel_vers_complexe(i, j, N):
    y = i / N
    x = j / N
    return x + y * 1j
```

In [29]:
```
pixel_vers_complexe(3, 8, 10)
```

Out[29]:
```
(0.8+0.3j)
```

Pour arrondir ... x est un réel *censé* être entre 0 et N. On renvoie l'entier correspondant, ou à peu près. Si x est trop petit ou trop grand on élague pour rester entre 0 et $N - 1$.

In [30]:
```
def round(x, N):
    i = int(x)
    if i <= 0: i = 0
    elif i >= N: i = N - 1
    return i
```

In [31]:
```
round(33.6, 67)
```

Out[31]:
```
33
```

La fonction réciproque de `pixel_vers_complexe` ...

In [32]:
```
def complexe_vers_pixel(z, N):
    x = z.real
    y = z.imag
    j = round(N * x + 0.5, N)
    i = round(N * y + 0.5, N)
    return (i, j)
```

In [33]:
```
complexe_vers_pixel(0.5+0.3j, 10)
```

Out[33]:
```
(3, 5)
```

Fabriquer une matrice $N \times N$ remplie de zéros ...

In [34]:
```
def matrice(N):
    A = N * [None]
    for i in range(N): A[i] = N * [0]
    return A
```

In [35]:
```
print(matrice(5))
```

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

4.3 La transformée de Von Koch

Nous y voilà. Que fait `transforme3` ? Pour chaque case de la matrice A égale à 1, elle calcule le nombre complexe z correspondant à la case. Elle applique les 4 similitudes dont nous avons parlé plus haut et met à 1 les cases correspondantes d'une matrice B initialement remplie de zéros. Enfin, la fonction renvoie B.

In [36]:
```
def transforme3(A):
    N = len(A)
    B = matrice(N)
    for i in range(N):
        for j in range(N):
            if A[i][j] == 1:
                z = pixel_vers_complexe(i, j, N)
                for simil in s:
                    u, v = complexe_vers_pixel(simil(z), N)
                    B[u][v] = 1
    return B
```

In [37]:
```
plt.rcParams['figure.figsize'] = (8, 8)
```

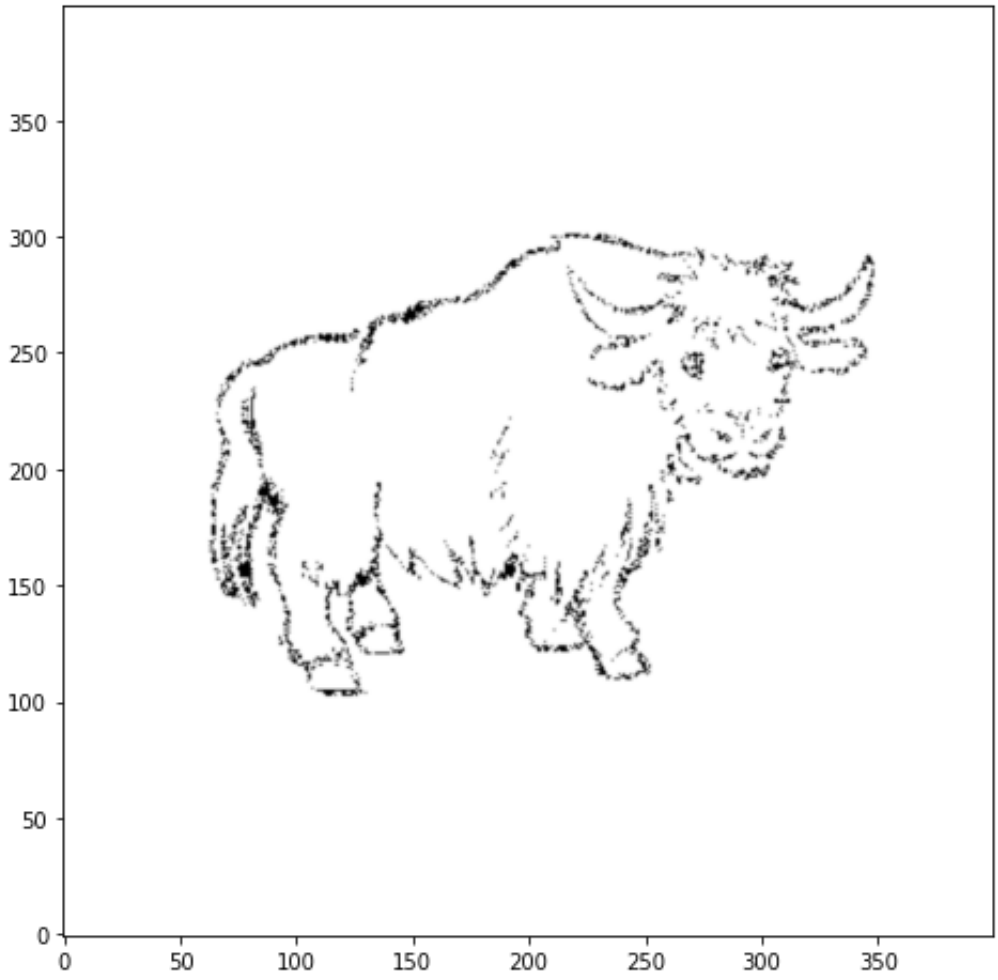
Petite fonction pour afficher le contenu d'une matrice ...

In [38]:
```
def plot_matrix(A):
    plt.imshow(A, origin='lower', interpolation='bicubic', cmap='Greys
```

Pour notre matrice d'exemple, nous allons prendre un yack. Mais un hobbit aurait tout aussi bien fait l'affaire.

In [39]:
```
def read_yack():
    A = matrice(400)
    f = open('yack-400x400.raw', 'rb')
    for i in range(400):
        for j in range(400):
            c = f.read(3)
            if int.from_bytes(c, byteorder='big',signed=False) ==0: A[
    f.close()
    return A
```

In [40]:
```
A = read_yack()
plot_matrix(A)
```



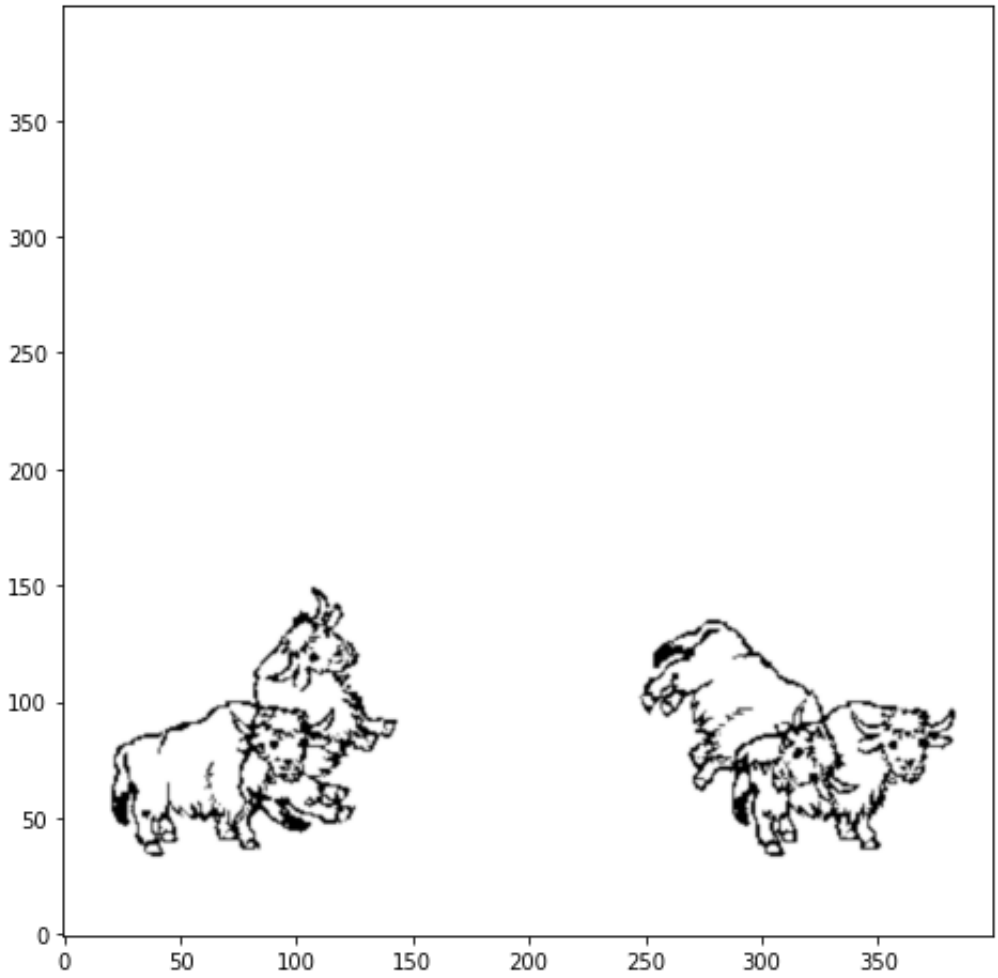
4.4 Itérations

Voici la fonction qui itère n fois $\mathcal{T}$ à partir d'un compact du plan représenté par une matrice A ...

In [41]:
```
def iter(n):
    A = read_yack()
    for k in range(n):
        A = transforme3(A)
    return A
```

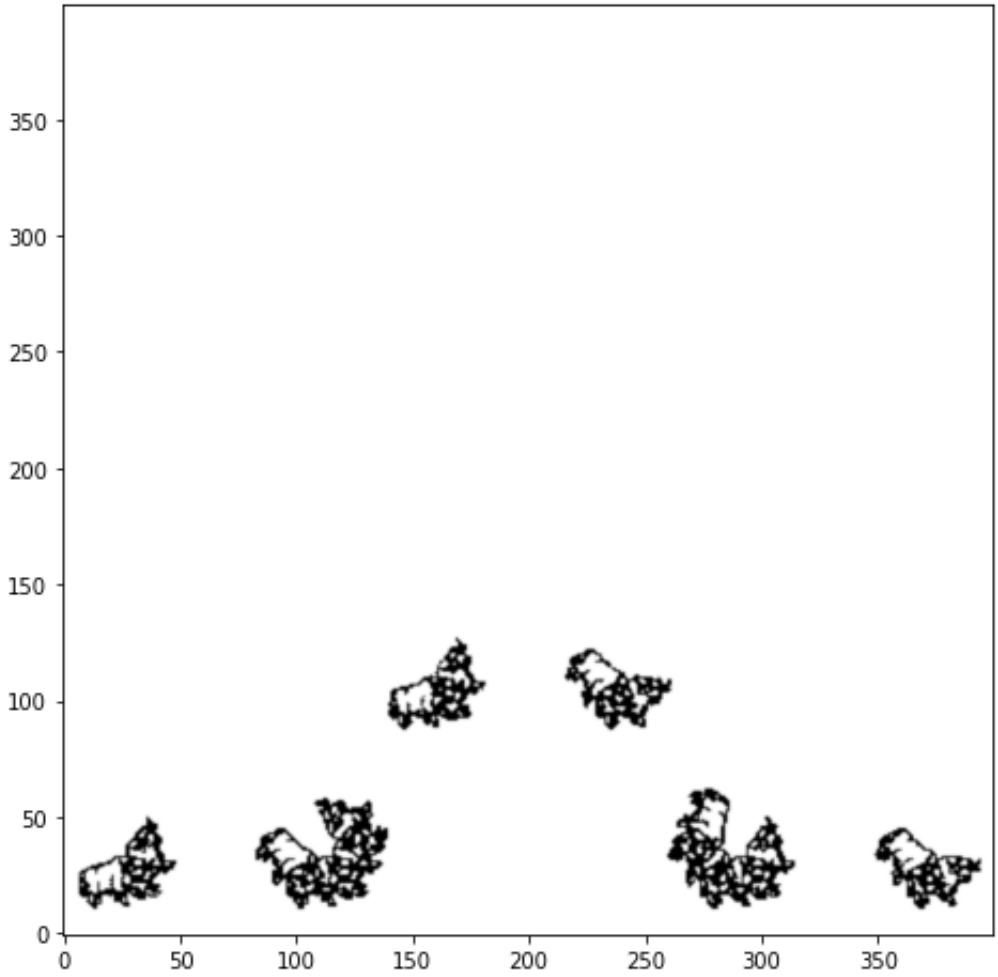
Lançons une itération.

In [42]:
```
plot_matrix(iter(1))
```



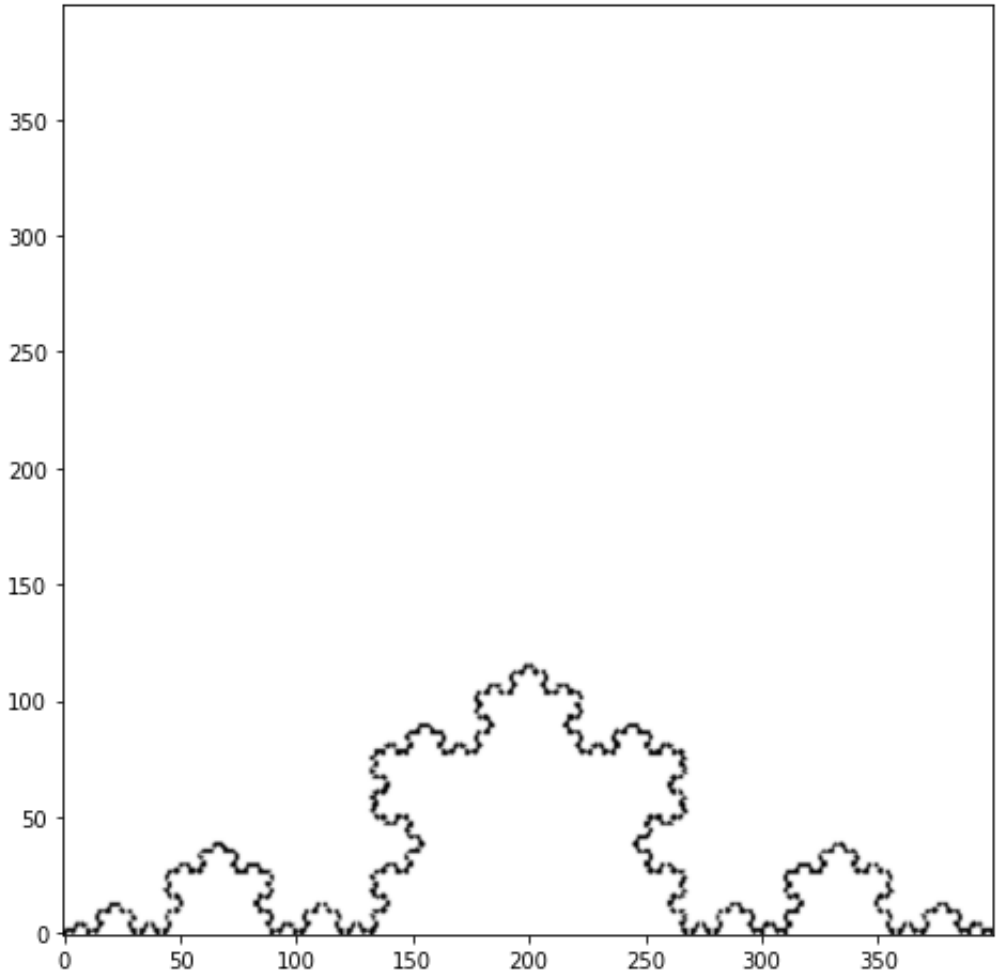
Oui, bon, on a 4 disques. Normal. Alors deux itérations ?

In [44]:
```
plot_matrix(iter(2))
```



Je vous laisse tester 3 et 4. Je lance 5 itérations

In [45]:
```
plot_matrix(iter(5))
```



Il fallait faire confiance aux maths. La courbe de Von Koch est bien la limite d'une itérée de yacks.

5. La dimension de K

Tout le monde le sait, une courbe c'est de dimension 1. Euh oui, les ensembles K_n sont des courbes (des unions de segments en fait), mais K c'est une autre paire de manches. Une limite de yacks tibétains est-elle une courbe ???

Rappelez-vous le **fait** : personne n'a jamais vu K.

5.1 Longueur de K_n

La longueur $L(K_n)$ de K_n ne pose aucun problème. K_0 est de longueur 1. Et à chaque étape on a 4 fois plus de segments, qui sont 3 fois plus petits. Donc $L(K_{n+1}) = \frac{4}{3}L(K_n)$ et ainsi $L(K_n) = (\frac{4}{3})^n$.

5.2 Longueur généralisée de K_n

Rigueur absente des deux prochains paragraphes ... tout ceci peut être mathématisé, mais c'est une longue histoire, trop longue pour un notebook.

Imaginons que je veuille mesurer la **longueur d'un segment** (de longueur π par exemple) avec une règle **non graduée**, mais de longueur connue. Je dispose de tout un attirail de règles de longueur 1, $\frac{1}{2}$, $\frac{1}{3}$, etc. J'applique 3 fois ma règle de longueur 1, trop peu. Je l'applique 4 fois, ça dépasse. J'en déduis que la longueur du segment est entre 3 et 4. Je recommence avec une règle de longueur $\frac{1}{10}$, j'applique ma règle une trentaine de fois, j'en déduis que la longueur du segment est entre 3.1 et 3.2, etc ...

Maintenant je veux mesurer la **surface d'un carré**. Cette fois-ci, je me procure des règles carrées de côtés divers, j'applique mes règles sur le carré à mesurer. J'en déduis des encadrements de la surface que je veux mesurer. Mais je n'oublie pas d'élever le côté de mes règles au carré, parce que ce sont des surfaces. Oui, j'imagine que vous êtes au courant, vous avez appris ça au CE1. Où veux-je en venir ?

Je veux mesurer, idée folle, la **longueur d'un carré**. Je reprends mes règles plates, je les applique à l'intérieur du carré en essayant de le recouvrir ... peine perdue, je n'arrive pas à remplir le carré. Sa longueur est **infinie**.

Allez, une dernière expérience. On n'en est plus à ça près, je veux mesurer la **surface d'un segment** de longueur 1. Je prend par exemple une "règle carrée" de côté $\frac{1}{n}$, je l'applique n fois et je recouvre le segment. Ben oui ça dépasse, difficile de faire autrement. J'en déduis que la surface S du segment vérifie $S \leq n \times (\frac{1}{n})^2 = \frac{1}{n}$. Pour tout n, bien entendu. Donc $S = 0$.

Je peux recommencer, essayer de calculer le **volume d'un carré**, la **surface d'un cube**, etc. Arrêtons de parler de longueur, surface, volume, et parlons de 1-mesure, 2-mesure, 3-mesure. En dimension d ce sera donc la d-mesure. L'idée c'est que pour chaque objet de dimension d, je dois choisir une règle adaptée pour obtenir une mesure qui ne soit ni nulle, ni infinie. Si j'utilise des δ-règles pour calculer la "mesure" d'un objet de dimension d, trois cas se présentent :

1. $\delta < d$: j'obtiens une δ-mesure infinie, comme quand je mesure la longueur d'un carré.
2. $\delta > d$: j'obtiens une δ-mesure nulle, comme quand je mesure la surface d'un segment.
3. $\delta = d$: c'est le bon choix, j'obtiens une δ-mesure ni nulle, ni infinie.

Prenons un réel $d > 0$ positif et d-mesurons K_n. On mesure 4^n segments, qui sont chacun de longueur $\frac{1}{3^n}$. Je prends une d-règle de longueur $\frac{1}{3^n}$, ce qui me donne une évaluation de la d-mesure du segment : $\frac{1}{3^{nd}}$. Une estimation de la d-mesure $\mu_d(K)$ de K est peut-être :

$$\mu_d(K) \simeq \mu_n = \frac{4^n}{3^{nd}}$$

5.3 La dimension de K.

Que se passe-t-il lorsque n tend vers l'infini ?

- Si $3^d < 4$, μ_n tend vers $+\infty$. On est dans le cas de celui qui veut mesurer un carré avec une règle. On a visé un peu léger.
- Si $3^d > 4$, μ_n tend vers 0. Un peu comme le type qui veut mesurer la surface d'un segment.

Mais si $3^d = 4$, alors $\mu_n = 1$ et tend donc vers une limite ni nulle ni infinie. On a trouvé la bonne valeur, LE d qui K comme il faut, la bonne règle. Passons aux logaritmes, il vient $d \ln 3 = \ln 4$.

Il se trouve que l'on peut vraiment définir mathématiquement la dimension d'un objet comme K. Il existe même beaucoup de définitions concurrentes (dimension de Hausdorff, dimension de boîte, etc.) et très différentes dans leur formulation. Mais pour l'ensemble K elles donnent toutes la même valeur :

Théorème : $\dim K = \frac{\ln 4}{\ln 3}$.

On est entre 1 et 2. K est plus gras qu'une courbe, mais plus maigre qu'une surface ...

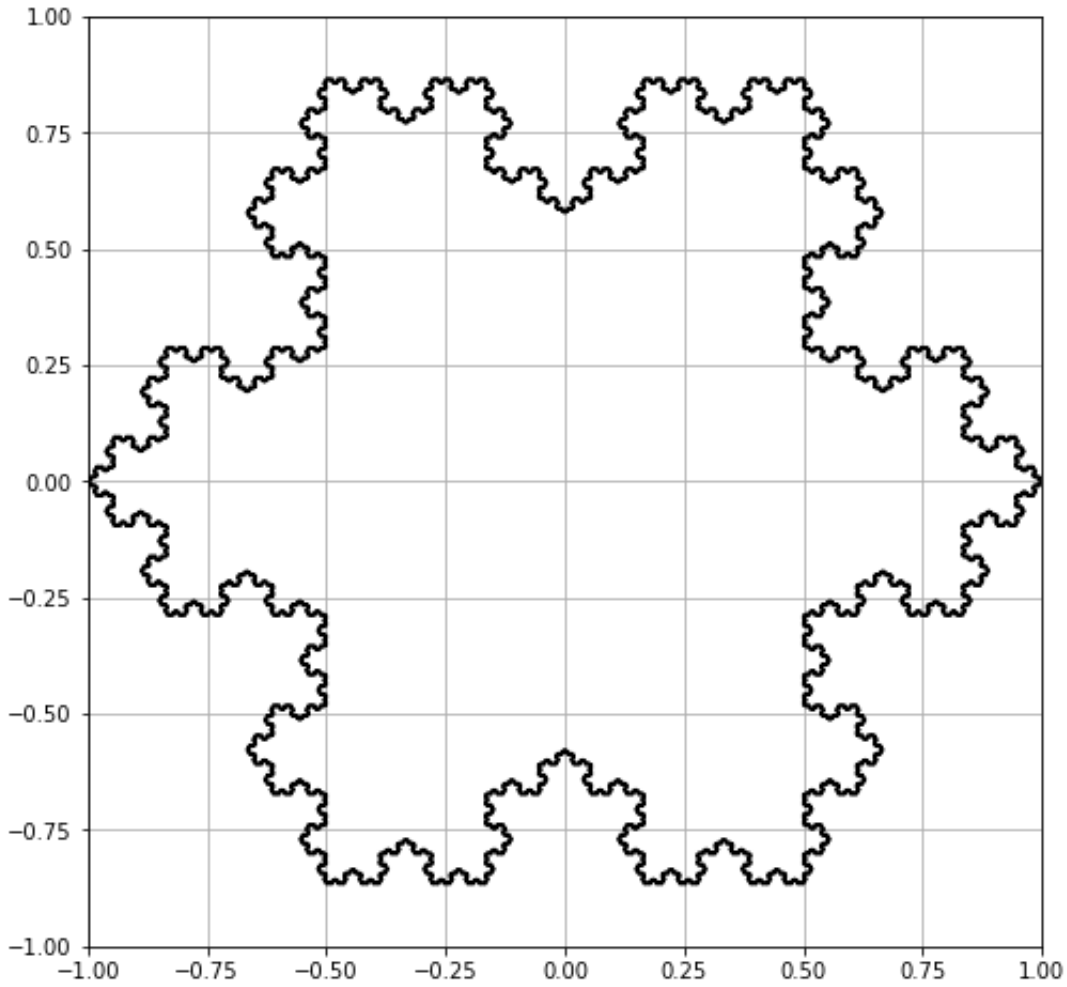
Annexe : le flocon de neige

Cette annnexe n'est là que pour faire joli. Le "flocon de neige" est le recollement de 3 courbes de Von Koch d'extrémités $1, j, j^2$ où j est la célèbre racine cubique de 1. Un dessin sera plus parlant.

In [46]:
```
plt.rcParams['figure.figsize'] = (8, 8)
```

In [47]:
```
def flocon(n):
    j = exp(2j * pi / 3)
    j2 = exp(-2j * pi / 3)
    v1 = vonkoch(n, 1, j2)
    v2 = vonkoch(n, j2, j)
    v3 = vonkoch(n, j, 1)
    return v1 + v2 + v3
```

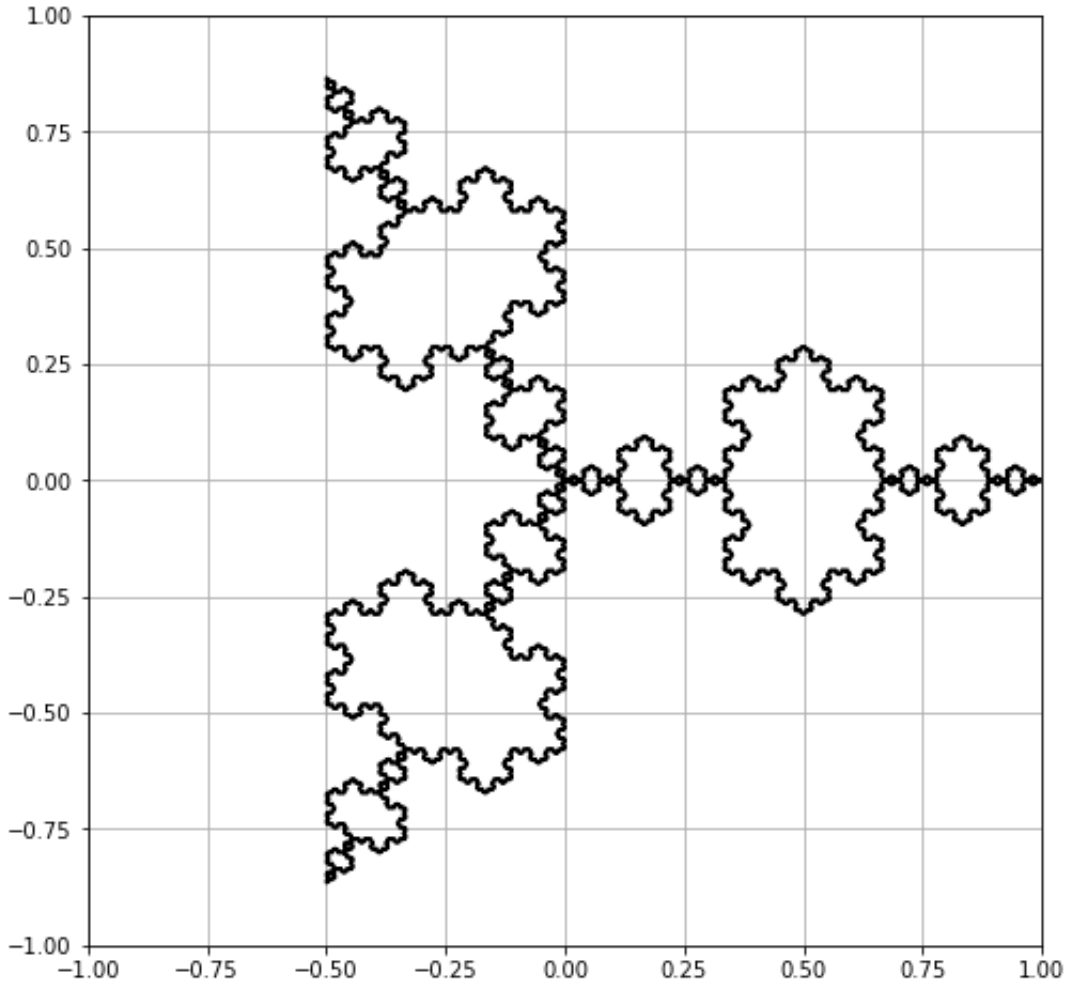
In [48]:
```
plotC(flocon(6), [-1, 1, -1, 1])
```



On l'appelle le flocon de neige parce que ça ressemble à un flocon de neige. Au fait, on a pris $1, j^2, j$ dans cet ordre ... et si on se trompe ?

In [49]:
```
def floconbis(n):
    j = exp(2j * pi / 3)
    j2 = exp(-2j * pi / 3)
    v1 = vonkoch(n, 1, j)
    v2 = vonkoch(n, j, j2)
    v3 = vonkoch(n, j2, 1)
    return v1 + v2 + v3
```

In [50]:
```
plotC(floconbis(6), [-1, 1, -1, 1])
```



C'est joli aussi, c'est un anti-flocon :-).

In [ ]:
```

```