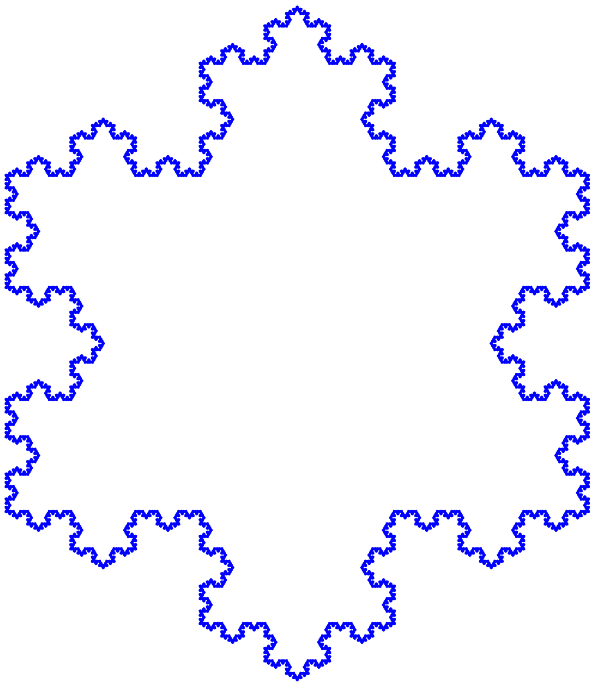
TP5

Paul Mercat

06/12/2016

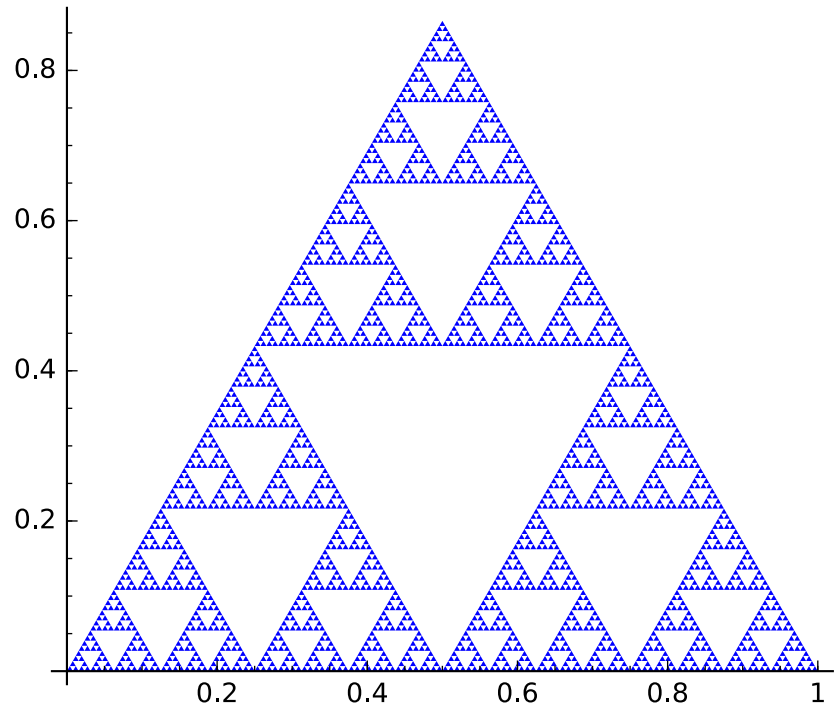
```
#Flocon de VonKoch
#On subdivise chaque segment (représenté par deux nombres complexes)\
    en deux segments n fois
#Le segment (z1, z2) est remplacé par les deux segments (z3, z1) et \
   (z2, z3) où z3 = z1 + (z2-z1)*z0 (le nombre complexe z0 sert à \
   multiplier le vecteur z2-z1 par 1/sqrt(3) et à le tourner de \
   arcsin(1/2))
z0 = CC(exp(I*asin(1/2))/sqrt(3))
def VonKoch(z1, z2, n):
    if n==0:
        return line([CC(z1), CC(z2)], aspect_ratio=1, axes=False) #\
   pour n=0, on dessine le segment donné en argument
    #sinon on subdivise en deux morceaux
    z3 = z1 + (z2-z1)*z0 #calcul du nouveau point
    return VonKoch(z3, z1, n-1)+VonKoch(z2, z3, n-1) #on dessine ré\
   cursivement pour chacun des deux segments

n = 10 #on choisit le nombre d'itérations
VonKoch(1, 0, n)+VonKoch(.5+sqrt(3)/2*I, 1, n)+VonKoch(0, .5+sqrt(3)\
   /2*I, n) #on dessine les itérations sur chacun des trois segments\
    d'un triangle équilatéral
```



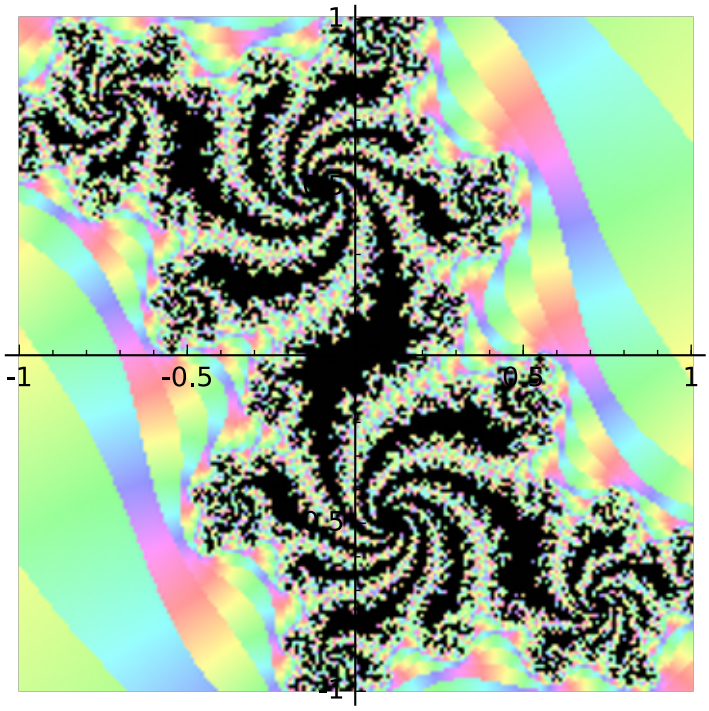
```
#Tapis de Sierpinski
#On subdivise le triangle en trois triangle deux fois plus petits
def Sierpinski (T, n):
    if n == 0:
        return polygon([(t.real(), t.imag()) for t in T]) #si n est \
   nul, on dessine le triangle passé en argument
    #sinon, on subdivise en trois morceaux, et on dessine ré\
   cursivement les triangles de Sierpinski pour chaque morceau
    g = Graphics()
    for i in range(3):
        g += Sierpinski((T[i], (T[i]+T[(i+1)%3])/2, (T[i]+T[(i+2)\
   %3])/2), n-1)
    return g

T = [0,1,.5+sqrt(3)*I/2] #on choisit un triangle de départ
Sierpinski(T, 7) #on dessine le tapis de Sierpinski avec 7 ité\
   rations
```



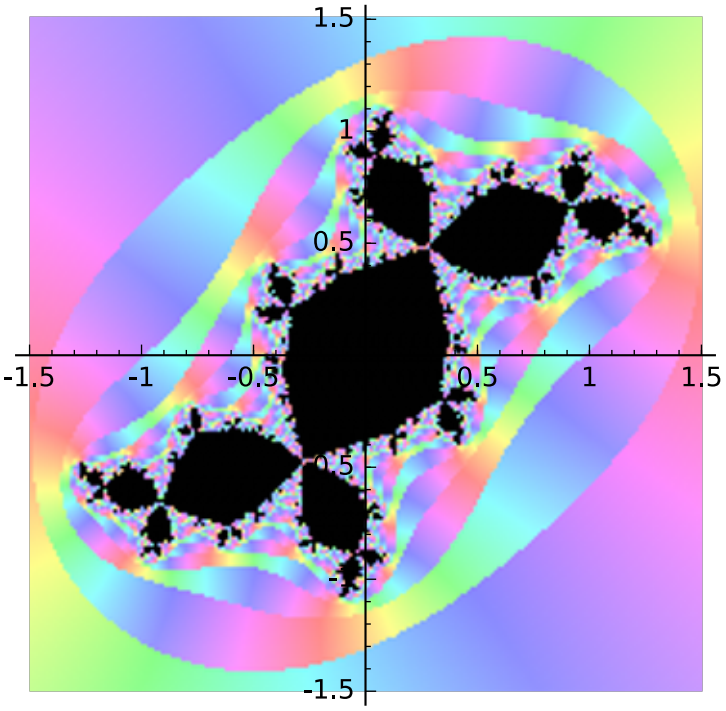

```
#Ensemble de Julia
def julia(z, c, n):
    for i in xrange(n):
        z = z*z + c
        if z.abs() > 2: #on arête la boucle dès que z est trop grand
            return (n+1)*z #on met n+1 plutôt que n pour que 0 \
   corresponde uniquement aux points de la fractale
    return 0

n = 150 #on choisit le nombre d'itérations
c = 0.134+0.6*I #on choisit le paramètre
complex_plot(lambda z:julia(z, c, n), (-1, 1), (-1,1), aspect_ratio\
   =1, plot_points=200) #long à calculer
```



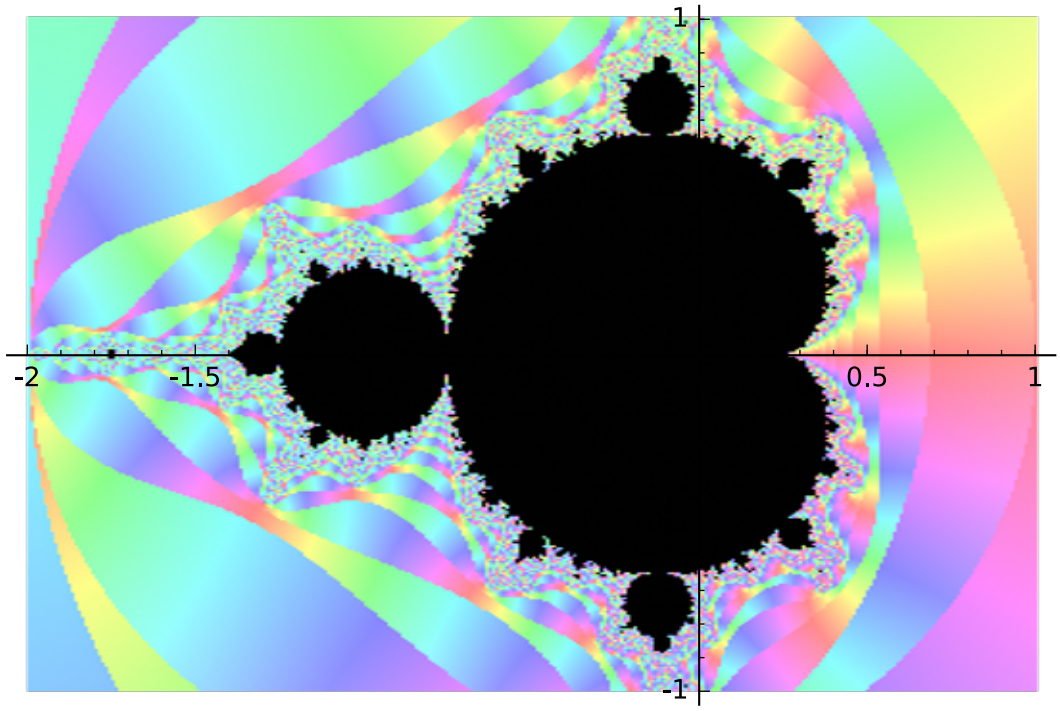

```
#Lapin de Douady
n = 80 #on choisit le nombre d'itérations
c = (x^3+2*x^2+x+1).roots(ring=CC)[1][0] #on choisit le paramètre de\
    la fractale de Julia pour avoir le lapin de Douady
print "c =",c
complex_plot(lambda z:julia(z, c, n), (-1.5, 1.5), (-1.5,1.5), \
   aspect_ratio=1, plot_points=200)
```

c = -0.122561166876654 - 0.744861766619744*I




```
#Ensemble de Mandelbrot
def f(z0, n):
    z = 0
    c = z0
    for i in xrange(n):
        z = z*z + c
        if z.abs() > 2:
            break
    if z.abs() > 2:
        return (n+1)*z
    return 0

n = 80
complex_plot(lambda z:f(z, n), (-2, 1), (-1,1), aspect_ratio=1, \
   plot_points=300)
```




```
#éponge de menger
colors = [(1,0,0),(0,1,0),(0,0,1),(1,1,0),(1,0,1),(0,1,1)] #on \
    choisit les couleurs des différentes faces des cubes

#subdivise un cube de position pos et taille size
def subdivise_cube (pos, size):
    res = []
    #parcours les 27 sous-cubes de taille 3 fois plus petite
    for i in range(3):
        for j in range(3):
            for k in range(3):
                if [i,j,k].count(1) < 2: #on ne garde que les cubes \
    qui ne sont pas "à l"intérieur"
                    size2 = size/3
                    res.append(((pos[0]+i*size2, pos[1]+j*size2, pos\
    [2]+k*size2), size2))
    return res

#calcule l'éponge de Menger avec n itérations
def Menger (n):
    l = [((0,0,0), 1)] #on part de ce cube
    for i in range(n): #on subdivise n fois
        l2 = []
        for pos,size in l:
            l2 += subdivise_cube(pos, size)
        l = l2
```

```
    return sum([cube(pos, size, color=colors, opacity=.5, \
    frame_thickness=1, frame_color="black") for pos,size in l])

Menger(2) #avec 3 c'est trop lent (8000 cubes à afficher !)
```