Chap08_lab

November 12, 2020

1 Capítulo 8

1.1 Números Complejos

El uso de números complejos puede no ser familiar para muchos de Uds. en Geociencias, pero en el tratamiento de datos y series de tiempo, se usa para el análisis de Fourier, y en geofísica también es muy utilizado.

En algunos lenguajes, el uso de números complejos no es sencillo. En Fortran por ejemplo, un número complejo es estandar, mientras que en C/C++ no lo es y se requiere cargar librerías para poder trabajar con ellos.

En Python se puede definir un número complejo usando

```
z = complex(2,3)
```

o también

```
z = 2+3j
```

donde `j` se usa en vez de `i` en Matlab.

Un programa básico con algunos ejemplos básicos del uso de números complejos en Python se muestra:

[3]:
```
# basic_complex.py
# Some basic complex number concepts

import cmath

z0 = 1j
z1 = 1j * 1j

print(z0)
print(z1)
```

```
1j
(-1+0j)
```

[4]:
```
z0 =  complex(2,3)
z  = 2+3j
print(z, z0)
```

```
(2+3j) (2+3j)
```

[5]:
```
print(z.real)
print(z.imag)
print(z.conjugate())

z = complex(3,4)
print(z, abs(z))
print(pow(z, 2))
```

```
2.0
3.0
(2-3j)
(3+4j) 5.0
(-7+24j)
```

[6]:
```
# Some function on complex
print("Functions on complex numbers")
z = complex(2,3)
zsin = cmath.sin(z)
zcos = cmath.cos(z)

print('For z = ', z)
print('cos(z), sin(z)', zcos, zsin)
```

```
Functions on complex numbers
For z =  (2+3j)
cos(z), sin(z) (-4.189625690968807-9.109227893755337j)
(9.15449914691143-4.168906959966565j)
```

1.1.1 Hágalo Ud mismo

Algunos ejemplos de números complejos Escriba un programa con interacción del usuario para multiplicar dos números complejos. Imprima los resultados, el valor absoluto y su raíz cuadrada.

[7]:
```
# testcomplex.py
import cmath

zreal,zimag = input("Enter first  complex number ").split()
a = complex(float(zreal),float(zimag))

b = complex(-0.5,0.1)

c = a*b

print('# complejo a    = ', a)
print('# complejo b    = ', b)
```

```
print('c = a x b       = ', c)
print('|c|             = ', abs(c))
print('sqrt(c)         = ', cmath.sqrt(c))
```

```
Enter first  complex number 3 5
# complejo a    =  (3+5j)
# complejo b    =  (-0.5+0.1j)
c = a x b       =  (-2-2.2j)
|c|             =  2.973213749463701
sqrt(c)         =  (0.6975721286948401-1.5768978643944733j)
```

Los números complejos en Python se representan como dos pares de números en parentesis, el segundo con una `j`. Por esto, no es fácil pedir el `input` del usuario. Se solicita dos números y el programa los lee y los pone dentro de un número complejo. Cualquiera de los siguientes casos

```
1, 1
(1, 1j)
1+1j
```

producirían errores en nuestro programa

Python proporciona las funciones para unir dos números dentro de un complejo, o extraer la parte real e imaginarias de un número complejo.

[8]:
```
# testcomplex2.py
import cmath
import numpy as np

zreal = np.random.rand()
zimag = np.random.rand()
a = complex(zreal,zimag)

print ('z      = (%.2f, %.2f) ' %(a.real, a.imag))

b = cmath.exp(a)
print ('exp(a) = (%.2f, %.2f)' %(b.real, b.imag))

print ('Real and Imaginary parts %.3f, %.3f' %(b.real, b.imag))
```

```
z      = (0.66, 0.26)
exp(a) = (1.86, 0.50)
Real and Imaginary parts 1.864, 0.501
```

1.1.2 Arreglos de números complejos

`NumPy` tiene la capacidad de trabajar con arreglos de números complejos, lo cual puede ser de gran utilidad. Incluso las operaciones en `numpy` trabajan sin problema (o por lo menos la parte que he mirado) con arreglos complejos y sin necesidad de cargar `cmath`.

Abajo un ejemplo simple de creación de un arreglo complejo, y el uso de multiplicación matricial `matmul()` con arreglos complejos.

[9]:
```
# array_complex.py
import numpy as np

a = np.array([1+2j, 3+4j, 5+6j])

print('Complex array ', a.shape)
print(a)

print ('Imag part ', a.imag)
a.imag = np.array([8, 10, 12])

print('New array ', a)
```

```
Complex array  (3,)
[1.+2.j 3.+4.j 5.+6.j]
Imag part  [2. 4. 6.]
New array  [1. +8.j 3.+10.j 5.+12.j]
```

[10]:
```
a = np.array([1+2j, 3+4j, 5+6j])
a = a[:, np.newaxis]
b = a.T

print('Array a, (3,1) ', a)
print('Array b, (1,3) ', b)
```

```
Array a, (3,1)  [[1.+2.j]
 [3.+4.j]
 [5.+6.j]]
Array b, (1,3)  [[1.+2.j 3.+4.j 5.+6.j]]
```

[11]:
```
c = np.matmul(a,b)

print ('matmul(a*a.T)')
print(c)

c = np.matmul(a,np.conjugate(b))

print ('matmul(a*conj(a.T))')
print(c)

c = np.matmul(a,np.conjugate(b))

print ('sqrt(c)')
print(np.sqrt(c))
```

```
matmul(a*a.T)
[[ -3. +4.j  -5.+10.j  -7.+16.j]
 [ -5.+10.j  -7.+24.j  -9.+38.j]
```

```
 [ -7.+16.j  -9.+38.j -11.+60.j]]
matmul(a*conj(a.T))
[[ 5.+0.j 11.+2.j 17.+4.j]
 [11.-2.j 25.+0.j 39.+2.j]
 [17.-4.j 39.-2.j 61.+0.j]]
sqrt(c)
[[2.23606798+0.j         3.33019068+0.30028311j 4.15115943+0.48179311j]
 [3.33019068-0.30028311j 5.        +0.j         6.24704924+0.16007558j]
 [4.15115943-0.48179311j 6.24704924-0.16007558j 7.81024968+0.j        ]]
```

1.2 Fractales

Si no lo han visto, en internet se encuentran figuras muy llamativas de fractales. Uno de los sets mas famosos son el *Mandelbrot Set*, el cual es relativamente fácil de generar en un programa de computador. Para hacerlo, se requiere poder manejar números complejos en Python.

La idea básica detrás del cálculo de fractales es que hay ciertas funciones complejas que, cuando se calculan repetidamente, pueden divergir o estar limitadas. El que divergan o no, es muy sensible a pequeños cambios en el valor del número complejo que inicia el cálculo. Esto se observa muy cerca de ciertas regiones en el plano complejo.

Una de las imagenes más famosas de fractales es el *Mandelbrot Set*. Para generar este *set*, empezamos considerando un número complejo `c`, al cual se le aplica el siguiente algoritmo:

```
comience con z=0
calcule repetidamente z = z*z + c
verifique si |z| > 2 y cuantas repeticiones
requirió para sobrepasar ese límite.
```

Por ejemplo, si $c = 0.3 + 0.3i$, entonces:

```
1ra rep: z =  0.30 + 0.30i  |z| = 0.42
2da rep: z =  0.30 + 0.48i  |z| = 0.57
3ra rep: z =  0.16 + 0.59i  |z| = 0.61
4ta rep: z = -0.02 + 0.49i  |z| = 0.49
```

En este caso, `z` permanecerá limitado aún hasta después de miles de iteraciones. Sin embargo, para $c = 0.5 + 1.0i$:

```
1ra iter: z =  0.50 + 1.00i  |z| = 1.10
2da iter: z = -0.25 + 2.00i  |z| = 2.00
3ra iter: z = -3.44 + 0.00i  |z| = 3.40
4ta iter: z = 12.32 + 1.00i  |z| = 12.4
5ta iter: z = 151.1 +25.63i  |z| = 153.4
```

y el valor de `z` rápidamente explota a valores infinitos. En la iteración 10 el valor seguramente ya excederá la capacidad para que un computador pueda guardar el número en memoria. Sin embargo, podemos evitar hacer estos cálculos de `z` una vez su valor absoluto exceda 2.0, ya que se puede demostrar que una vez ese valor es alcanzado, `z` tiende a diverger.

El cálculo se realiza para una serie de valores de `c` y el resultado se grafica como función de la posición de `c` en el plano complejo (la parte real en el eje `x` la parte imaginaria en el eje `y` y el

número de iteraciones como valor de amplitud). El *Mandelbrot set*, es el set de números complejos `c` para los cuales el tamaño de z^2+c es finito aun después de un número infinito de iteraciones. Una buena aproximación, es por ejemplo realizar esta operación hasta un número grande de iteraciones (1000 puede ser un buen ejemplo), y asumir que si $|z| > 2$, el valor va a diverger.

1.2.1 Hágalo Ud mismo

Escriba un código para calcular el *Mandelbrot set*, donde se define de manera automática el número de puntos para el eje `x` y el eje `y` (eje de los reales e imaginarios).

Por ejemplo, siempre haga 100 puntos en `x` y 100 en `y`, para un total de 10.000 puntos (`x,y`).

También permita que el usuario defina la región (`Xmin`, `Xmax`, `Ymin`,`Ymax`) donde quiere hacer el cálculo (es como permitir hacer zoom).

Para cada punto

```
c = complex(x,y)
```

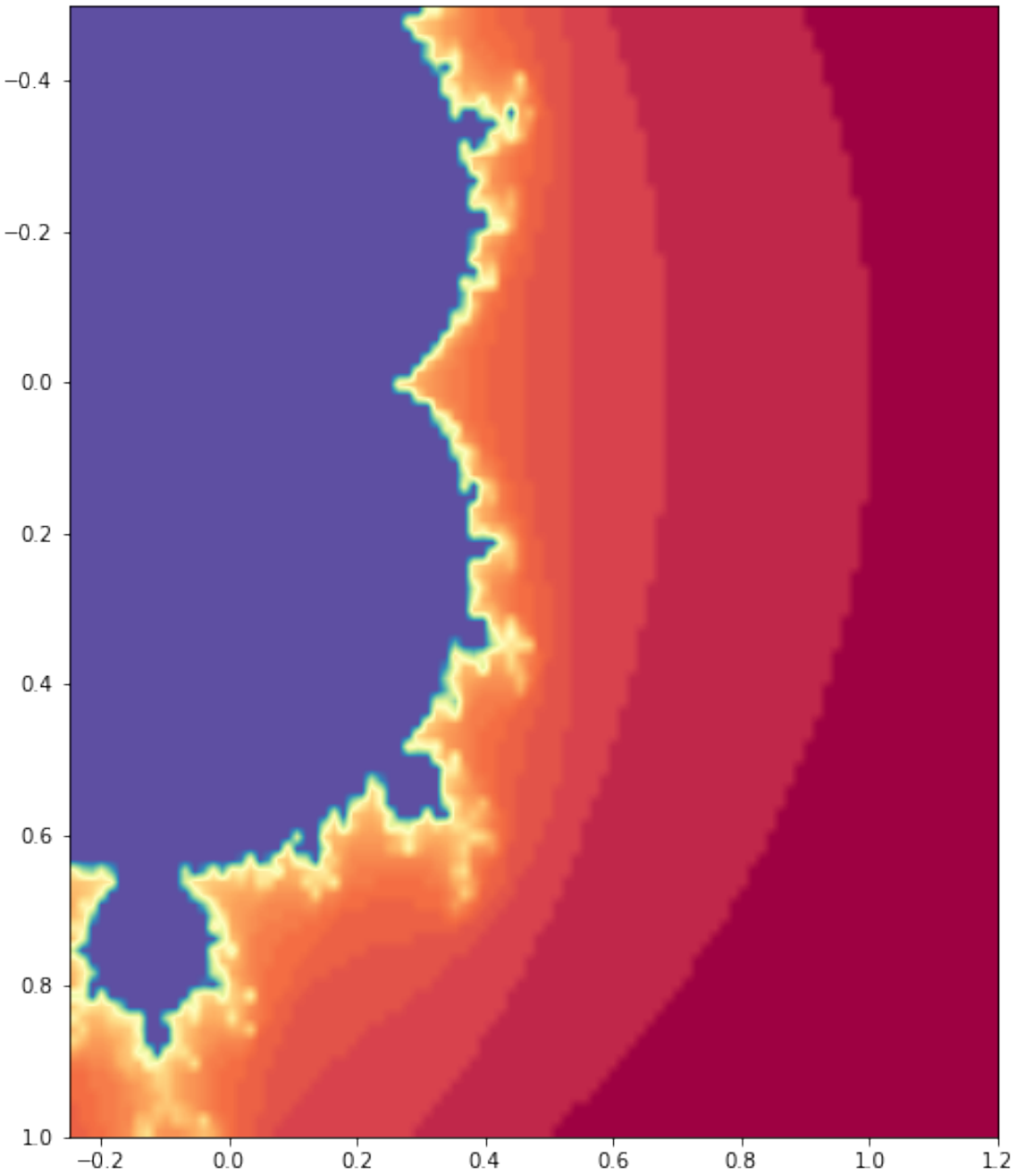
cuente el número de repeticiones que necesita hacer para que $|z| > 2$. Si ha hecho más de 1000 repeticiones, pare y continue con el siguiente par de números (`x,y`).

Haga la figura a color de `x`, `y` y el número de repeticiones, que sería una matriz de 100,100 (por ejemplo con `imshow`).

[12]:
```
# mandel1.py
# Plot the mandelbrot set for a number of points
#

import numpy as np
import matplotlib.pyplot as plt

# Defina los límites X,Y
x1,x2,y1,y2 = input("Enter x1,x2,y1,y2 ").split()
x1 = float(x1)
x2 = float(x2)
y1 = float(y1)
y2 = float(y2)

# Define grid size and start matrix
nx = 100
ny = 100
dx = (x2-x1)/float(nx)
dy = (y2-y1)/float(ny)
dat = np.zeros((nx,ny))

# Main loop for each value
for ix in range(nx):
   for iy in range(ny):
      cr = x1 + dx/2. + dx*float(ix)
      ci = y1 + dy/2. + dy*float(iy)
```

```
      # Create complex number
      c = complex(cr,ci)
      z = complex(0.0, 0.0)
      for it in range(1000):
         z = c + z*z
         if (abs(z) > 2):
            break
      dat[ix,iy] = it+1

# Rotate matrix and plot log10 scale
dat = np.transpose(dat)
zdat = np.log10(dat)

# Plot result
fig = plt.figure(figsize=(8,10))
ax  = fig.add_subplot()
ax.imshow(zdat,interpolation='bilinear',
          extent=(x1,x2,y2,y1),cmap='Spectral')
ax.axis('equal')
ax.axis('tight')
plt.show()
```

```
Enter x1,x2,y1,y2 -0.25 1.2 -0.5 1.0
```


−0.4
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2