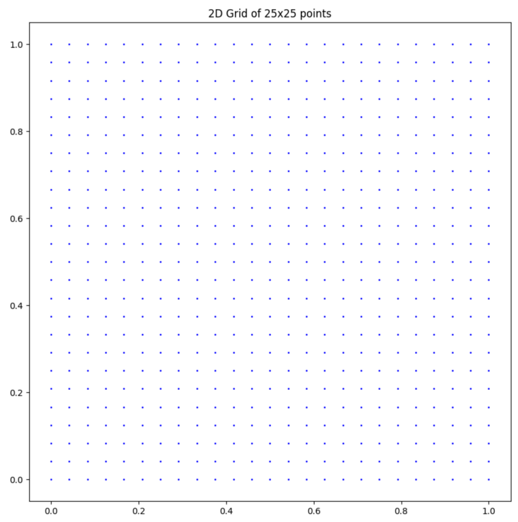
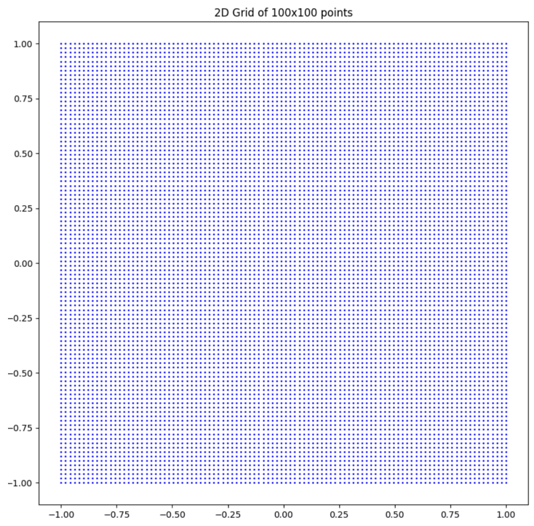
The first block of code defines a two-by-two array of a defined size, which will be carried through in later blocks of code.

```
# Set the size of the grid
size = 25  # You can adjust this to make the grid denser or sparser
```

The difference between sizes sets the number of dots/points equally in both axes. When size is equal to 25 versus size equal to 200:

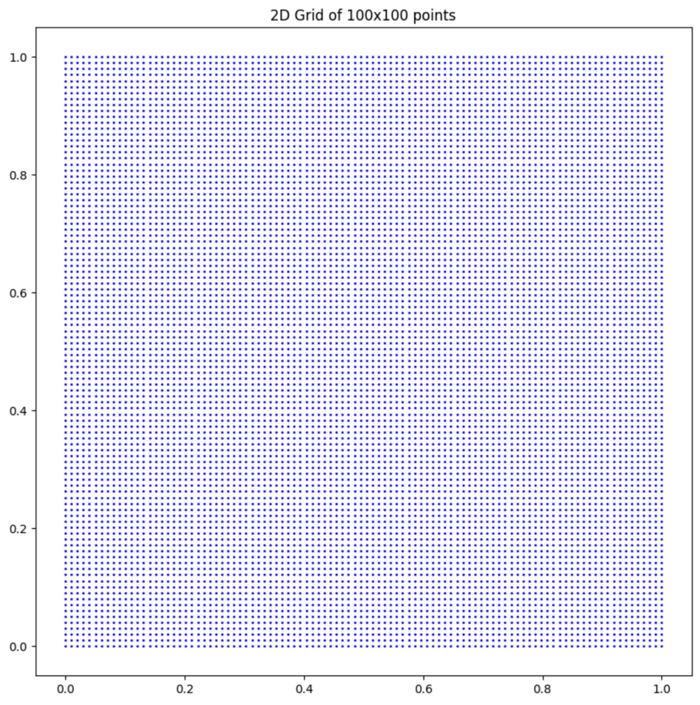




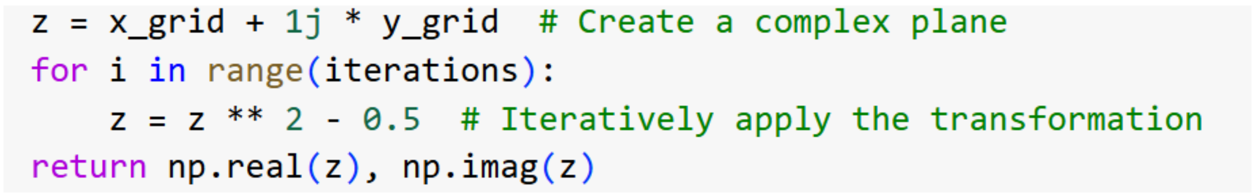
The grid itself is defined in the first function:

```
# Create a simple 2D grid of points using NumPy
def generate_grid(size):
    x = np.linspace(-1, 1, size)  # Generate 'size' number of points between -1 and 1
    y = np.linspace(-1, 1, size)
    grid = np.meshgrid(x, y)  # Create a 2D grid of points
    return grid
```

The parameters for x and y set the minimum and maximum of the coordinates along those axes. As seen in the prior examples, the range is from -1 to 1. Changing the values has no immediate impact on the creation of the planes.



The remaining three images are conversions of the standard coordinate plane into the complex plane with the real numbers on the x-axis and the imaginary numbers on the y-axis. To do this, first the transformation is defined, bringing in the grid value from the original block of code, which determines the number of points to be plotted, along with the number of iterations, with sequentially increasing ones used in the three following blocks of code.



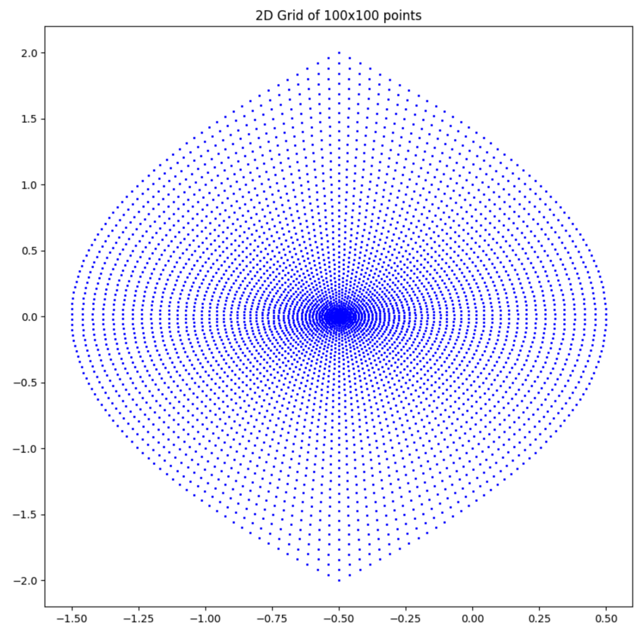
```
z = x_grid + 1j * y_grid  # Create a complex plane
for i in range(iterations):
    z = z ** 2 - 0.5  # Iteratively apply the transformation
return np.real(z), np.imag(z)
```

A large struggle I had was why you had z set to be x_grid + 1j * y_grid. You mention it creates the complex plane, but not HOW it happens. After looking it up, 1j represents the square root of -1 or i, which makes perfect sense. For example, given the point (1, .5) on the coordinate plane, it would initially be equal to 1+.5i for the complex plane. From there you take the complex number, square it, subtract .5 and return the real and imaginary pieces out to be graphed. Again, using the example above, the point (1, .5) would become

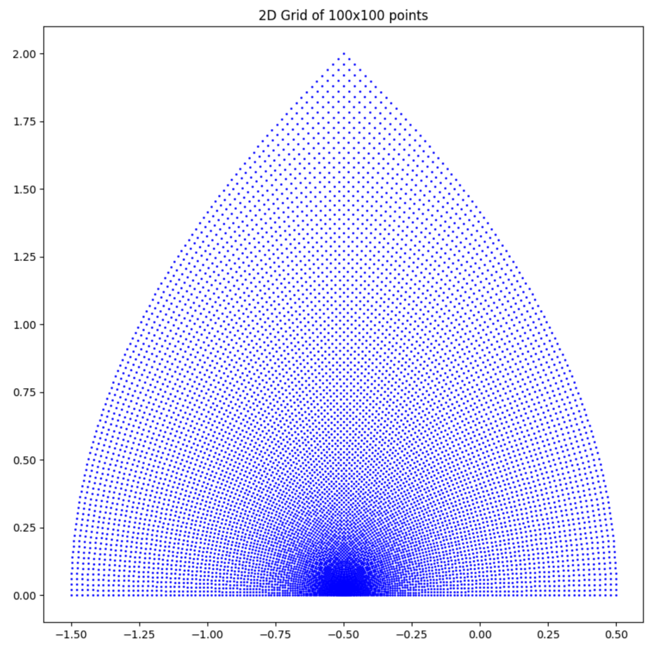
$$(1+.5i)^2 - .5 = 1 + i - .25 - .5 = .25 + i$$

which returns the value (.25, 1) to be graphed. For later iterations, the product is squared and subtracted the number of times listed for the iterations.

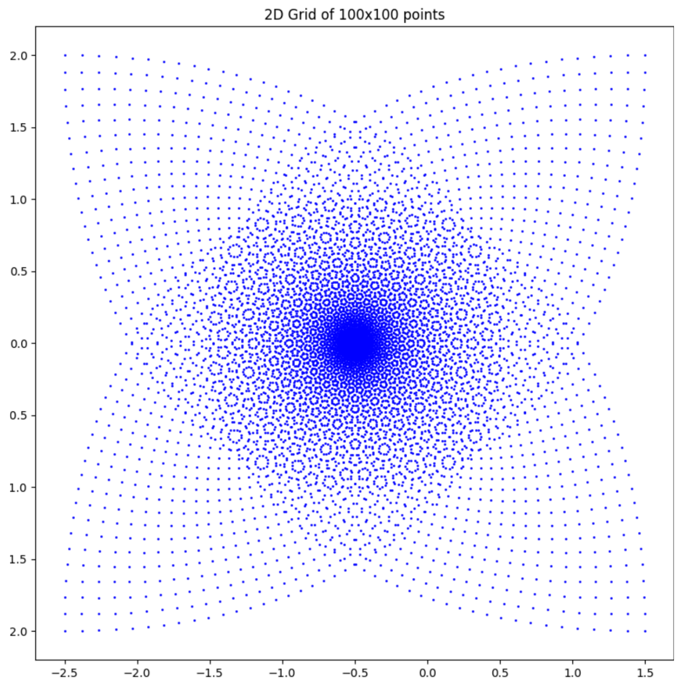
Playing with the code, I want to keep the grid size of 100 as it applies a nice level of density without turning into a solid block of dots.   On the original code, you get the following pattern:

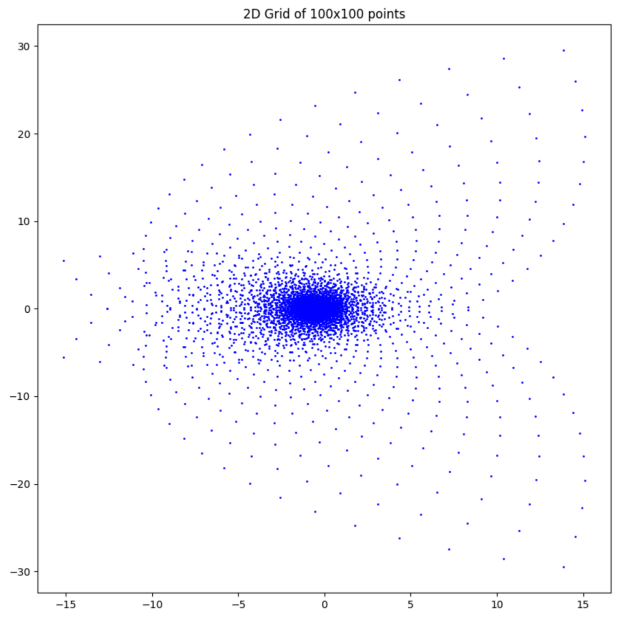


By recreating this, but focusing strictly on positive values, you unsurprisingly get the top half of the pattern.

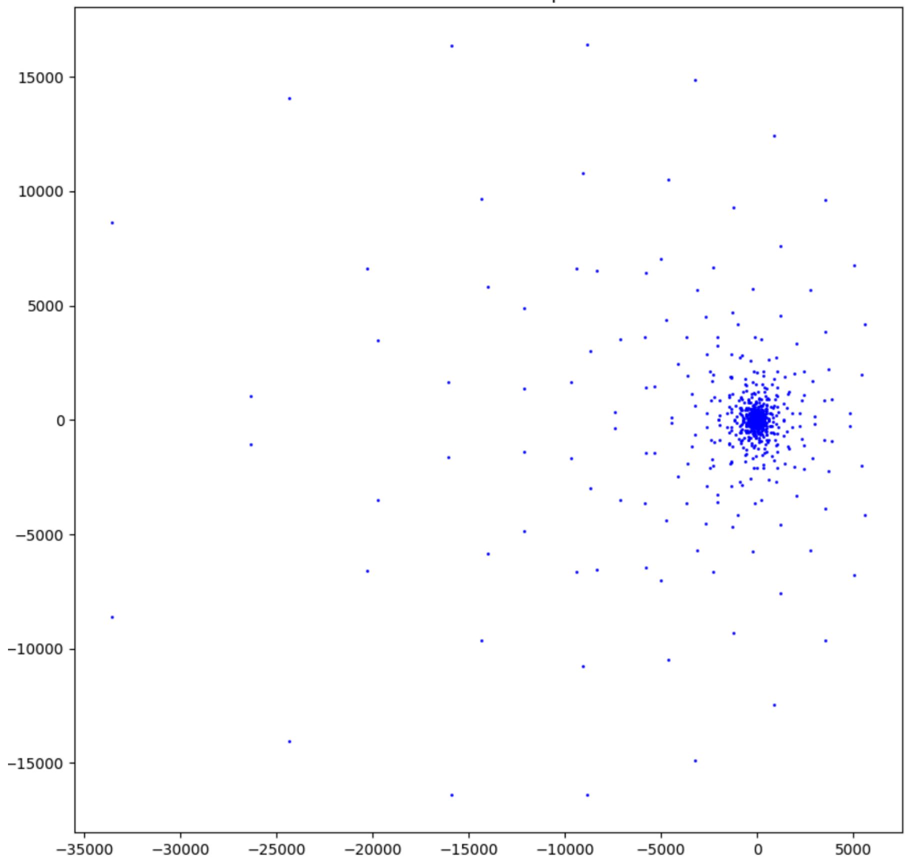


Removing the subtraction portion for the transformation makes almost no change. The points get tighter together, but not particularly noticeably so. What does make a large difference is cubing imaginary number rather than squaring it. The three iterations are as follows:





2D Grid of 100x100 points


15000
10000
5000
0
−5000
−10000
−15000
−35000
−30000
−25000
−20000
−15000
−10000
−5000
0
5000