Numerical Python Plots for Mandelbrot Set & Lorenz' Equations

Nicole Jiang

Abstract—This writeup presents visualizations of the Mandelbrot set and Lorentz' equations. These results were computed and plotted using numerical Python programming, specifically through simple iteration and libraries like NumPy, Matplotlib, and SciPy. The purpose is to implement and analyze numerical methods in Python to visualize the fractal geometry of the Mandelbrot set and the chaotic dynamics of the Lorenz system, thereby deepening our practical understanding of nonlinear dynamical systems.

1. Introduction

The Mandelbrot set and Lorenz' equations are classic examples of nonlinear dynamical systems that have made significant contributions to science, extending beyond their initial discoveries. Both concepts provide valuable practice and learning opportunities for numerical Python and scientific programming. The Mandelbrot set, a certain collection of complex numbers, is plotted by iterating a complex quadratic map over a dense grid in the complex plane. Lorenz' equations, a system of three ordinary differential equations, model atmospheric convection. The instability of the equations' solutions demonstrate the sensitivity of the system and how small changes in initial conditions or numerical parameters can lead to drastically different trajectories.

2. Methods and Results - The Mandelbrot Set (Q1)

2.1. Q1 Plot 1

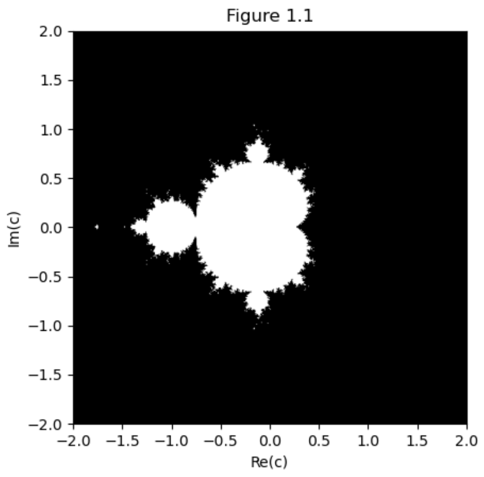
Given $c = x + iy, -2 < x < 2, -2 < y < 2, z_0 = 0$, and $z_{i+1} = z_i^2 + c$, the first task was to iterate z_{i+1} by designing a function to call upon before plotting. The function, recorded in a separate .py file later imported to the main program, was defined to return a Boolean array indicating which points of the graph escaped, running off to infinity and diverging. The points were determined to diverge if $|z| > 2$. This was found through the following steps:

Let R represent escape radius. By triangle inequality,

$z_{n+1} = z_n^2 + c \rightarrow |z_{n+1}| \geq |z_n^2| - |c| \geq |z_n|^2 - R.$

It follows that $|z_n|^2 - R \geq R \rightarrow |z_n|^2 \geq 2R$. For $|z_n| = R$, $R^2 \geq 2R \rightarrow R \geq 2$.

Therefore, the escape radius must be 2 since any more than that and the point would diverge, not coming back. In regards to the plot, the colour was set to reverse grayscale so the points that did not escape would be False and white while the diverging points become black. See **Figure 1.1** for reference. The resulting plot is a fractal with infinitely intricate boundaries.



2.2. Q1 Plot 2

A second plot regarding the Mandelbrot set included a colour legend so that the colour of a point indicated the iteration number at which the given point diverged. See **Figure 1.2** for reference, and **Code 1** for the function body of the iteration program that made it work. It is clear that the iteration at divergence is higher around the borders of the fractal.

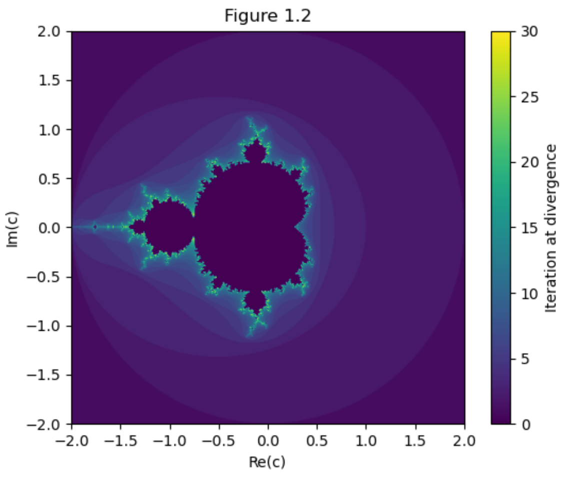


Figure 1. Enter Caption

```
    # Create the complex grid
    x = np.linspace(-2, 2, N) # 1D arrays
    y = np.linspace(-2, 2, N)
    x, y = np.meshgrid(x, y) # 2D NxN grid of x and y
    c = x + 1j * y  # 2D NxN array c of complex numbers

    # Iterate through z and mark escaped points
    diverged = np.zeros(c.shape, bool)  # Set up
    ↪ boolean array, same size as c, to mark points
    ↪ that diverge. Initialize as filled with False

    # Iterate through z, this time recording escape
    ↪ iteration
    count = np.zeros(c.shape, int)  # Hold iteration
    ↪ count
    for i in range(1, max_iter + 1):  # Start at 1 for
    ↪ first update z0 -> z1
        with np.errstate(over="ignore",
    ↪ invalid="ignore"):  # Ignore error statement
    ↪ that shows up because values go to inf
            z = z**2 + c
        just_div = (np.abs(z) > 2) & (~diverged)  #
    ↪ Just diverged array will hold points that
    ↪ diverge for the first time
        count[just_div] = i  # Record iteration count
    ↪ when point escapes
        diverged[just_div] = True  # Record that point
    ↪ diverged

    return count
```

Code 1. Q1 Plot 2 Iteration Function Body

3. Methods and Results - Lorenz Equations (Q2)

Using Lorenz' equations 25-27, a function was defined to return an array representing a vector quantity with three entries: the rate of change of x, y, and z each with respect to dimensionless time. See **Code 2** for the function body.

```
# Unpack W
x, y, z = W

# Lorentz' equations for derivatives of x, y, z
 ↪ with respect to time
xprime = -1 * sigma * (x - y)
yprime = r * x - y - x * z
zprime = -1 * b * z + x * y

# Return vector as an array
return np.array([xprime, yprime, zprime])
```

Code 2. Lorenz' Equations

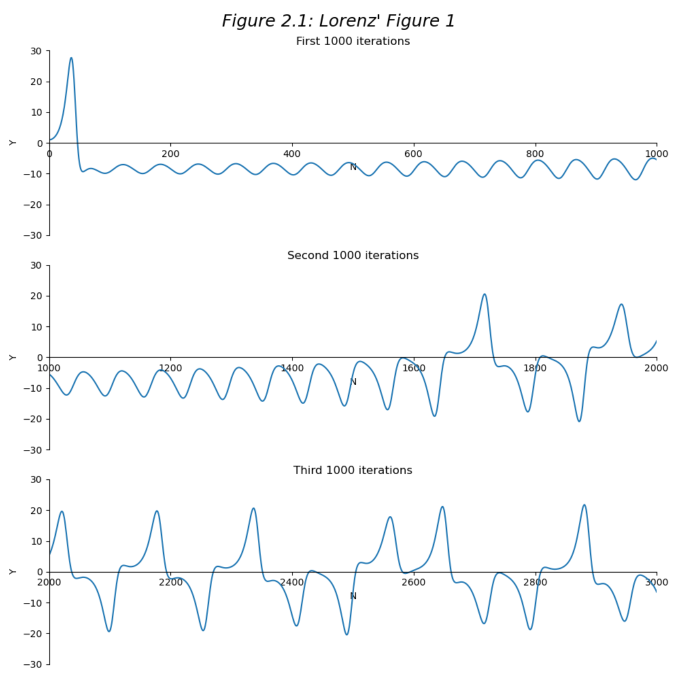
Furthermore, an ODE solver was programmed and prepared to be implemented into future plots. Knowing that the interval of time given spanned 60.0s seconds, and that $dt = 0.01$, it was possible to calculate how many evenly spaced values of time the ODE solver needed to compute.

$$N = t/dt = 60.0/0.01 = 6000.$$

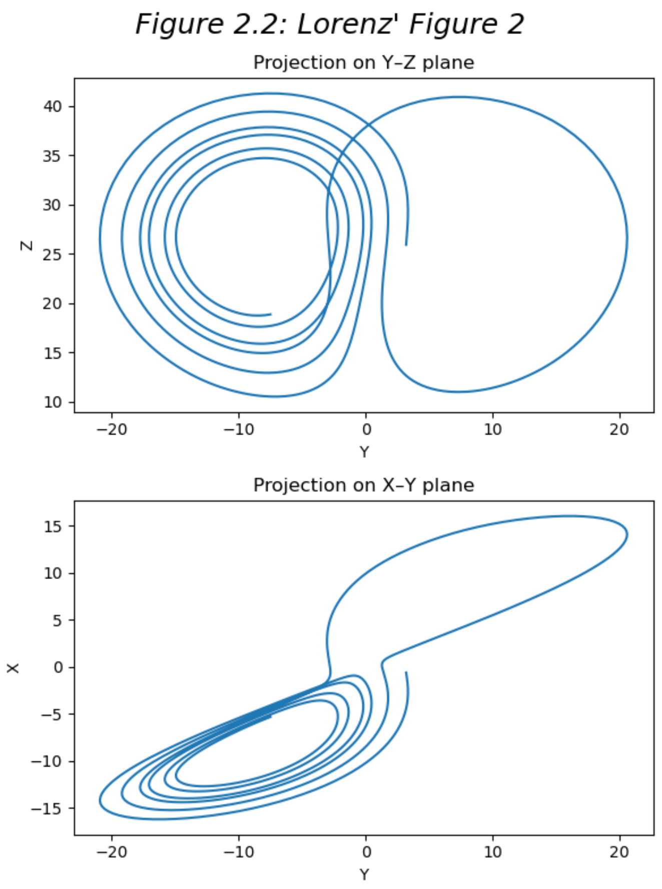
It follows that the argument for the evaluation of t takes 6001 to have 6000 iterations.

3.1. Q2 Plot 1

The first figure to reproduce was Lorenz' Figure 1 from his paper: 3 graphs, each of Y as a function of time for 1000 iterations. See **Figure 2.1** for reference.



It is evident that even though the overall trend and the shape of the graphs are close, there are small discrepancies when compared to Lorenz' original figure, such as an extra spike in some places or maybe a missing one in another. This is because of the instability of the solutions and their susceptibility to be thrown off by minuscule differences. Even though the same values were given, there are a lot of small but impactful discrepancies that sets one hand apart from another. As per the methodology, **Code 3** is how values of time were converted to iterations. **Code 4** is a for loop that made labeling the three different graphs swiftly.



3.2. Q2 Plot 2

The next plot is called figure 2 in Lorenz' paper. It shows the phase-space projection on the x-y and y-z plane. Similarly to the last, these plots are extremely sensitive and have clear differences when compared to Lorenz' figure. However, the overall shape suggests an accurate execution of calculations, since the phase-space system associated with Lorenz' equations are characterized to be similarly swirly shaped as in **Figure 2.2**.

```
# Get values for axes
y = sol.y[1]
dt = 0.01  # timestep
N = sol.t / dt
interval = 1000
```

Code 3. Lorenz' Equations

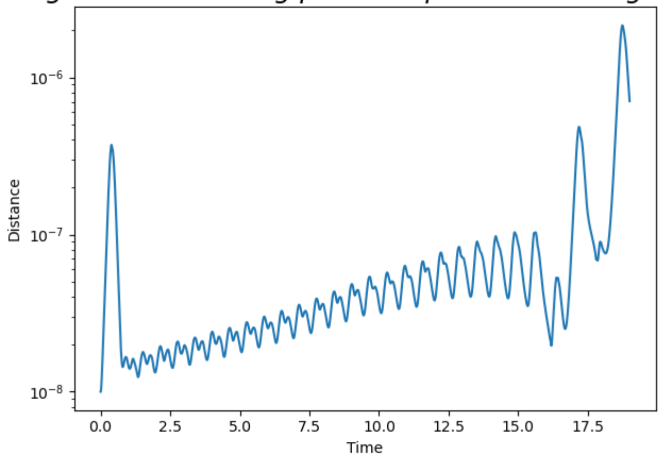
```
for i in range(3):
ax[i].set_xlabel('N')
ax[i].set_ylabel('Y')
ax[i].set_ylim(-30, 30)  # Making y-axis symmetrical
ax[i].spines['bottom'].set_position(('data', 0))  #
 ↪ Setting x-axis on y = 0
ax[i].spines['top'].set_visible(False)  # Hiding
 ↪ bars for cleaner look
ax[i].spines['right'].set_visible(False)
```

Code 4. Mass Labeling

3.3. Q2 Plot 3

The last plot, **Figure 2.3**, demonstrated how different a graph behaves with different initial values. This graph does not look related to the others, yet only a few things in the code are different.

Figure 2.3 - Semilog plot of exponential divergen



4. Conclusion

Numerical experiments were successfully implemented in Python to visualize two nonlinear systems: the Mandelbrot set and the Lorenz equations. By iterating the complex quadratic map over a fine grid, the fractal boundary of the Mandelbrot set was produced. It was observed how escape-time coloring reveals its intricate structure. Integrating the Lorenz system using a fixed time step to illustrate the system's extreme sensitivity to initial conditions. Future work could explore different time stepping for more efficient Lorenz simulations, or extend the fractal investigation.