Assignment 3

Arianna Lasinski

2023-05-08

Abstract

This paper explores two different mathematical models. The first model involves iterating the equation $z_{i+1} = z_i^2 + c$ for complex values of c in a specific range. The iteration produces two sets of values - one that remains bounded in absolute value and the other that diverges to infinity. Visualizations of these two sets of values using different color schemes are created. The second model explores Lorenz equations that were developed to study the behavior of Earth's atmosphere. The Lorenz equations are implemented into a function and then solved using an ODE solver. Two plots from the original paper are reproduced. The first plot shows the evolution of the system over time, while the second plot shows the behavior of the three variables over that range. Finally, the sensitivity of the solutions to initial conditions is investigated, highlighting the rapid growth of errors in initial conditions, and how it impacts future behavior predictions.

1 Question 1

Fractals are fascinating mathematical objects that exhibit self-similarity and complexity at every level of magnification. They have captured the imaginations of mathematicians, scientists, and artists alike, and have applications in fields ranging from computer graphics to physics and biology. One of the most famous fractals is the Mandelbrot set, which is created by iterating a simple equation for each point in the complex plane. In this paper, we will explore the behavior of the Mandelbrot set using a similar iterative process.

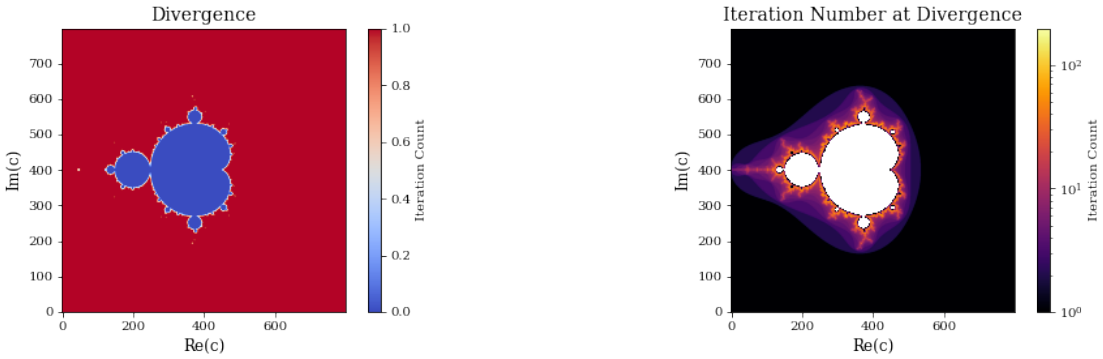
For each point c in the complex plane, we will set $z_0 = 0$ and iterate the equation $z_{i+1} = z_i^2 + c$. By examining the behavior of the resulting z_i values, we can determine whether the point c is part of the Mandelbrot set or not. Specifically, if the absolute value of z_i remains bounded (i.e., $|z|^2 = \Re(z)^2 + \Im(z)^2$ does not exceed a certain threshold for all iterations, then c is considered part of the Mandelbrot set. However, if the absolute value of z_i grows without bound, then c is not part of the set. To visualize this behavior, we will create two images. In the first image, we will color the points that diverge with one color and the points that stay bounded with another color. In the second image, we will use a color scale to indicate the iteration number at which the given point diverged. By exploring the Mandelbrot set in this way, we can gain a deeper understanding of its fascinating structure and properties.

To do so, we define a function `complex_plane` that takes one argument, `max_iter`, which specifies the maximum number of iterations to be performed for each point (See Appendix A). The code then imports the `NumPy` library and creates two arrays, 'x' and 'y', using the `linspace` function. These arrays represent the range of values for the real and imaginary parts of the complex numbers in the plane. Next, the `meshgrid` function is used to create two 2D arrays, 'X' and 'Y', which represent the coordinates of each point in the complex plane. The 'c' array is then created by combining 'X' and 'Y' into complex numbers. The 'z' array is initialized as a copy of 'c'. The 'divergent' array is created as a Boolean array of the same shape as 'c', initialized with 'False' values. The 'iterations' array is also created as an integer array of the same shape as 'c', initialized with '0' values.

The code then enters a 'for' loop that iterates `max_iter` times. In each iteration, the 'z' array is updated using the equation for the Mandelbrot set: $z_{i+1} = z_i^2 + c$. The 'divergent' array is updated based on the condition that the absolute value of 'z' exceeds 2. The 'iterations' array is updated with the iteration number

for each point that has just diverged. Finally, the function returns the 'divergent' array, which contains a Boolean value for each point in the complex plane indicating whether that point diverged or not, and the 'iterations' array, which contains the iteration number at which each divergent point was identified.

The result is a simple and efficient way to generate the Mandelbrot set and determine which points are part of the set and which are not. By varying the value of `max_iter`, we can increase the resolution of the image and explore more intricate details of the fractal. Figures 1 and 2 below depict the results of this process.



(a) (b)

Figure 1: (a) A complex plane plot of the function $z = z^2 + c$, where c is a complex number and z is iteratively updated until it diverges or a maximum number of iterations is reached. The color of each point represents the number of iterations required for the function to diverge at that point. The plot has a colorbar on the right indicating the range of iteration counts. The x-axis and y-axis correspond to the real and imaginary parts of c, respectively. (b) The number of iterations taken for complex numbers c to diverge from the sequence $z_{n+1} = z_n^2 + c$, plotted in the complex plane. The colormap indicates the iteration count on a logarithmic scale, with warmer colors indicating a higher number of iterations. The real and imaginary components of c range from -2 to 2.

2 Question 2

In this section, we explore the famous Lorenz system, which is a set of three coupled ordinary differential equations that describe the behavior of a simplified model of atmospheric convection. The Lorenz system was first introduced by Edward Lorenz, a meteorologist, in his seminal 1963 paper "Deterministic Nonperiodic Flow," which is one of the earliest examples of the phenomenon of chaos. The equations that govern the Lorenz system are highly nonlinear, which means that even small variations in the initial conditions can lead to drastically different outcomes. This sensitivity to initial conditions is known as the butterfly effect, and it is a hallmark of chaotic systems. Despite its simplicity, the Lorenz system has been the subject of extensive research and has found applications in many areas of science, including meteorology, physics, and engineering.

We begin by defining a function that implements the Lorenz equations in Python, using a function definition with proper docstrings. We then use an ODE solver, `solve_ivp`, to integrate the equations for a given period of time using Lorenz's initial conditions and parameter values. We reproduce two of Lorenz's figures to visualize the behavior of the system and demonstrate its chaotic nature. Finally, we investigate the sensitivity of the Lorenz system to initial conditions by slightly perturbing the initial conditions and calculating the distance between the perturbed solution and the original solution over time. We plot the result on a semilog plot, which will show us the exponential growth of the difference between the two solutions, further emphasizing the chaotic nature of the Lorenz system.

We first define a function, `lorenz`, that represents the Lorenz system of differential equations (See Appendix B). The function takes in two parameters, t representing time and W representing an array of three variables x, y and z. The function also uses three other parameters, σ, r, and b, which are constants that determine the behavior of the system. The function calculates the derivatives of the variables x, y and z using the given input variables and returns them in an array-like object named `dW_dt`. The function can be used as input to an ODE solver to integrate the Lorenz system and study its behavior over time. The `solve_ivp` function from the `scipy.integrate` module is used to solve the Lorenz system of equations specified by the function, `lorenz`, which takes as input the time t and an array of the variables W representing the x, y and z variables of the Lorenz system (See Appendix C).

The `solve_ivp` function integrates the Lorenz system from time t=0 to t=60 with the initial conditions W0 and parameters as follows:

- Prandtl number $\sigma = 10$
- Rayleigh number $r = 28$
- Dimensionless length scale $b = 8/3$
- Initial values of the x, y, and z variables $W_0 = [0, 1, 0]$

The `dense_output=True` argument enables the solution to be interpolated at any time between the initial and final times, allowing for more flexibility in the use of the solution. The output of `solve_ivp` is stored in the `sol` variable, which is an object of the `scipy.integrate.OdeSolution` class. The `sol` object contains information about the solution, such as the time points and the corresponding values of the variables x, y and z.

We then recreate Lorenz' plots of the numerical solutions of the convection equation with y as a function of time for the first 1000 iterations, then for the second 1000 iterations, and finally for the third 1000 iterations. We also plot the projection on the X-Y plane and the Y-Z plane in phase space of the segment of the trajectory. These are depicted in figure 2 below.

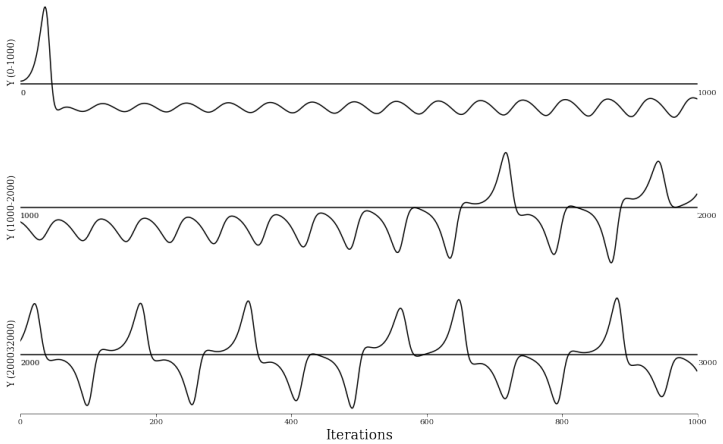


Figure 2: Evolution of y variable in the Lorenz system over time for subsequent intervals of 1000 iterations. Initial conditions and parameters are: $\sigma = 10$, $r = 28$, $b = 8/3$, $W_0 = [0, 1, 0]$. The system is solved using the `solve_ivp` function from `scipy.integrate`. The plot shows y as a function of the number of iterations, with each panel corresponding to a different time interval. The black horizontal line represents $y = 0$.

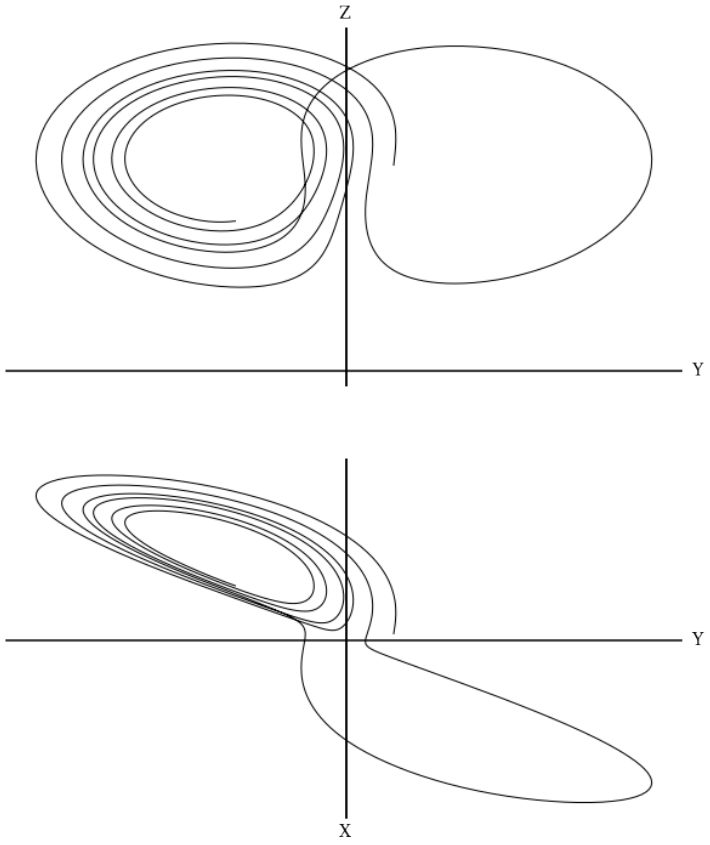


Figure 3: Plots illustrating the behavior of the Lorenz system with initial conditions and parameters $\sigma = 10$, $r = 28$, $b = 8/3$, and $W_0 = [0, 1, 0]$. In the top plot, the Y variable trajectories are shown over 2000 iterations, with three plots corresponding to iterations 0-1000, 1000-2000, and 2000-3000. The Y variable is plotted on the y-axis, and the x-axis represents the iteration number. In the second plot, the phase portrait for Y and Z variables (left) and Y and X variables (right) is shown, with the Y variable on the y-axis and the Z or X variable on the x-axis. The phase portrait illustrates the attractor of the Lorenz system.

Finally, we solve the Lorenz system of equations with a different set of initial conditions (See Appendix D). First, the initial conditions for the Lorenz system are defined as they are originally. Then a new set of initial conditions `W0_prime` is generated by adding a small perturbation to each element of the initial conditions list `W0`. The perturbation is given by the list `[0., 1.e-8, 0.]` and is added to the corresponding element of `W0` using a list comprehension and the `zip()` function. This results in `W0_prime` being set to $[0., 1.00000001, 0.]$. Next, a time span for the integration is defined as `t_span =` $[0, 60]$. Using the `solve_ivp` function from `scipy.integrate`, the Lorenz system is solved for both initial conditions, `W0` and `W0_prime`, over the time span `t_span`. The resulting solutions are stored in sol and `sol_prime`, respectively. Then, the Euclidean distance between the solutions at each time step is calculated). Finally, the distance is plotted as a function of time on a semilog plot using `matplotlib`. The results are shown in Figure 4. If the distance increases exponentially with time, this indicates that small differences in the initial conditions can lead to large differences in the solutions over time, which is known as the butterfly effect.

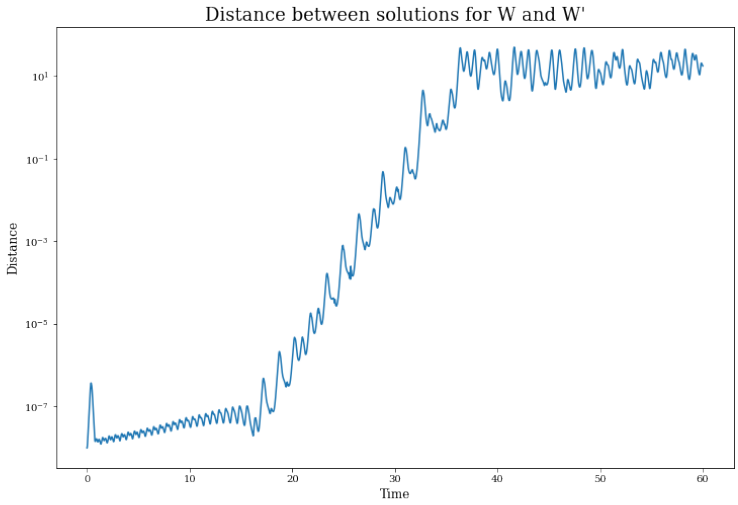


Figure 4: Distance between solutions for W and W' as a function of time. The Lorenz system of equations was solved for both initial conditions, where $W_0 = [0, 1, 0]$ and $W_0' = [0, 1 + 10^{-8}, 0]$, and the Euclidean distance between the solutions at each time step was calculated. The plot shows the distance on a semilog scale, with time on the x-axis and distance on the y-axis.

Appendix A Complex Plane

We define a function `complex_plane` that generates the Mandelbrot set and determines which points are part of the set and which are not.

```
def complex_plane(max_iter):
    x = np.linspace(-2, 2, 800)
    y = np.linspace(-2, 2, 800)
    X, Y = np.meshgrid(x, y)
    c = X + Y * 1j
    z = c.copy()
    divergent = np.full_like(c, False, dtype=bool)
    iterations = np.zeros_like(c, dtype=int)

    for i in range(max_iter):
        z[divergent] = 0
        z[~divergent] = z[~divergent] ** 2 + c[~divergent]
        divergent[np.logical_and(~divergent, np.abs(z) > 2)] = True
        iterations[np.logical_and(divergent, iterations == 0)] = i

    return divergent, iterations
```

Appendix B Lorenz Equations

We define a function, `lorenz`, that represents the Lorenz system of differential equations.

```
def lorenz(t,W):
    """
    Lorenz system of equations:
    dx/dt = -sigma*(x-y)
    dy/dt = r*x - y - x*z
    dz/dt = -b*z + x*y

    Parameters:
    -----------
    t: float
        time
    W: array_like, shape(3,)
        x, y and z variables of Lorenz system
    sigma: float
        Prandtl number
    r: float
        Rayleigh number
    b: float
        Dimensionless length scale

    Returns:
    --------
    dW_dt: array_like, shape(3,)
        Derivatives of x, y and z variables
    """
    dX = -sigma*(W[0]-W[1])
    dY = r*W[0] - W[1] - W[0]*W[2]
```

```
    dZ = -b*W[2] + W[0]*W[1]
    return [dX,dY,dZ]
```

Appendix C Solving Lorenz System of Equations

The `solve_ivp` function from the `scipy.integrate` module is used to solve the Lorenz system of equations specified by the function.

```
sol = solve_ivp(lorenz, [0,60], W0, dense_output=True)
```

Appendix D Differing Initial Conditions

We solve the Lorenz system of equations with a different set of initial conditions.

```
# Define initial conditions
W0 = [0, 1, 0]
W0_prime = [w + wp for w, wp in zip(W0, [0., 1.e-8, 0.])]

# Define time span for integration
t_span = [0, 60]

# Solve Lorenz system of equations for both initial conditions
sol = solve_ivp(lorenz, t_span, W0, t_eval=np.linspace(t_span[0], t_span[1], 10000))
sol_prime = solve_ivp(lorenz, t_span, W0_prime, t_eval=np.linspace(t_span[0], t_span[1], 10000))

# Calculate Euclidean distance between solutions at each time step
distance = np.sqrt(np.sum((sol.y - sol_prime.y)**2, axis=0))

# Plot distance as a function of time on a semilog plot
fig, ax = plt.subplots(figsize=(12, 8))
ax.semilogy(sol.t, distance)
ax.set_xlabel('Time')
ax.set_ylabel('Distance')
ax.set_title('Distance between solutions for W and W\'')
plt.show()
```