Applied Mathematics

José Jaén Delgado

Introduction - In this paper algorithms related to Numerical Methods are explained with the corresponding mathematical derivations and code.

1 Chaos Theory: Logistic Map

Let us perform a Linear Stability Analysis on a polynomial mapping of degree two. We study the stability of fixed points of an archetypal non-linear dynamical equation: the Logistic Map. A fixed point is mathematically defined as:

$$x^* = f(x^*) \tag{1}$$

So we seek to analyze whether values close to x^* converge or diverge from said fixed point. Analytically, we can determine the stability of x^* by selecting a value $x^{(t)}$ distanced from x^* by a negligible margin $\epsilon^{(t)}$ and calculating $x^{(t+1)}$:

$$\begin{gathered} x^{(t)} = x^* + \epsilon^{(t)} \\ x^{(t+1)} = x^* + \epsilon^{(t+1)} \end{gathered}$$

So:

$$f(x^* + \epsilon^{(t)}) = x^* + \epsilon^{(t+1)} \tag{2}$$

Resorting to Taylor Series Expansion to find a final expression for $f(x^* + \epsilon^{(t)})$ and focusing solely on linear terms:

$$\begin{aligned} f(x^* + \epsilon^{(t)}) &= f(x^*) + \epsilon^{(t)} f'(x^*) + \dots \\ &= f(x^*) + \epsilon^{(t)} f'(x^*) \;\; \text{(since } \epsilon^{(t)} \text{ is close to machine epsilon)} \end{aligned}$$

Consequently, the magnitude relation between iterations results in:

$$\left|\frac{\epsilon^{(t+1)}}{\epsilon^{(t)}}\right| = |f'(x^*)| \tag{3}$$

Thus, it can be concluded that

$$x^* = \begin{cases} \text{Stable if} & |f'(x^*)| < 1 \\ \text{Unstable if} & |f'(x^*)| > 1 \end{cases}$$

Having defined the scope of this section and the key mathematical insights, we proceed to particularize the aforementioned Linear Stability Analysis framework for the Logistic Map. Note that the `Python` and `MATLAB` code corresponding to the following algorithms can be found on my **GitHub** repository.

The Logistic Map is a quadratic, one-dimensional iterative map popularized by Robert May (1976) which is used as a discrete-time demographic model. It can be expressed as:

$$x^{(t+1)} = \mu x^{(t)}(1 - x^{(t)}) \tag{4}$$

Where μ is the reproduction rate and $x^{(t)}$ represents the ratio of existing population to the maximum possible population at the t-th iteration. It is then easy to see that:

$$f'(x^{(t)}) = \mu - 2\mu x^{(t)} \tag{5}$$

Furthermore, fixed points can be calculated by solving $x^* = \mu x^*(1 - x^*)$. The simplest one sets $x^* = 0$ whereas the second fixed point requires dividing by x^*

$$1 = \mu\frac{1 - x^*}{x^*} \implies x^* = 1 - \frac{1}{\mu}$$

Since we aime to study the behavior of $x^{(t+1)}$ as μ varies, let us restrict this last paremeter so that $\mu \in (2.4, 4)$.

As the nature of Logistic Map is intrinsically iterative, we need to define an algorithm that will allow us to compare the different values of the reproduction rate against $x^{(t)}$. Likewise, note that some transient iterations are needed so that new values of x^* branch out.

We expound the Logistic Map bifurcation algorithm in the next page.

Algorithm 1: Bifurcation of Logistic Map

```
Input: $\mu$ region $\Theta = (2.4, 4)$, Transient Iterations $m$, Data Iterations $n$
Output: $x^{(t+1)}$ population ratio values
# Randomly Initialize Population Ratio
$x^{(0)} \sim N(0.5, 0.29)$
for $\mu \in \Theta$ do
    # Loop over transient
    for $i = 1, \ldots, m$ do
        $x^{(i+1)} = \mu^{(i)} x^{(i)} (1 - x^{(i)})$
    end
    # Update final values
    for $t = 1, \ldots, n$ do
        $x^{(t+1)} = \mu^{(t)} x^{(t)} (1 - x^{(t)})$
    end
end
```

It is much more intuitive to graphically analyze x when μ varies to identify divergence. Note how for some μ, the population ratio starts branching out to different values. Hence, Chaos Theory was developed to study such behavior.

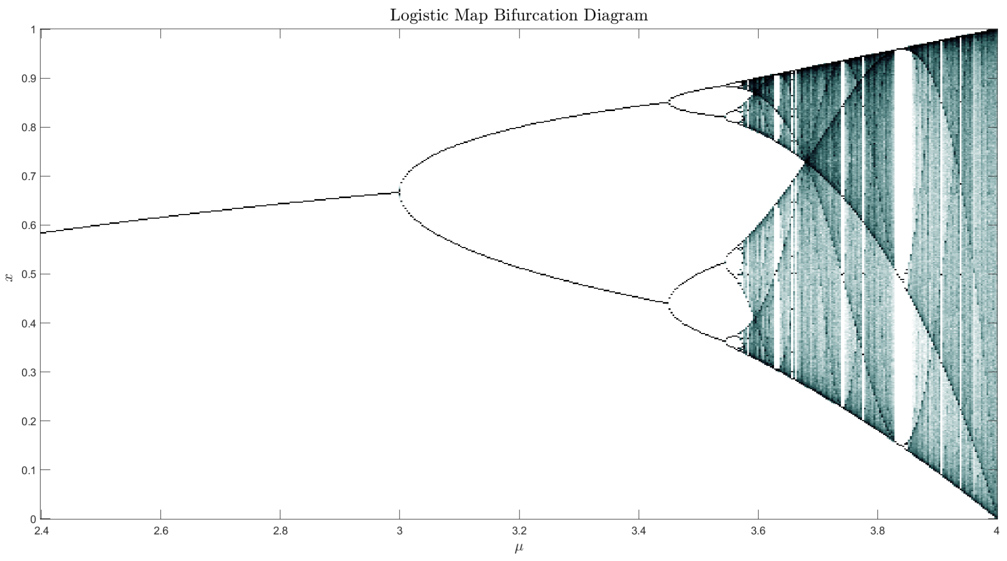

Figure 1: Bifurcation Diagram

Our `Python` implementation initializes $x^{(t)}$ around 0.5 to facilitate convergence, although directly setting it to 0.5 procures almost identical results (as carried out in `MATLAB`). Thus, a normal distribution with mean 0.5 and low standard deviation is chosen for retrieving such convenient initial value. It is also recommended to loop over a certain range of transient iterations before actually starting to collect data.

The time complexity of this Algorithm is $O(nmp)$, where p is equal to the length of Θ. After trying different combinations of iteration values (n, m, p), it was found that 2000 transient iterations and 1000 data iterations seem enough to smoothly plot the bifurcation diagram for a total of 500 values of μ. `Python` was noticeably slower than `MATLAB`, which is why higher iteration values were tried out in the latter.

2 Chaos Theory: Feigenbaum Delta

Following our results in the preceding section, it can be noticed how for some values of μ, the population ratio spreads out in two paths. Thus, after some transient iterations for which $x^{(t)}$ is constant, its value changes into two ones. Feigenbaum (1978) studied period-doubling bifurcations in the context of one-dimensional maps with a single quadratic maximum. He derived a constant that characterizes the geometric approach of the bifurcation parameter to its limiting value. A cycle is composed of different $x^{(t)}$ values for the same $\mu^{(t)}$. Formally, Feigenbaum Delta is expressed as:

$$\delta = \lim_{n \to \infty} \frac{\mu_{n-1} - \mu_{n-2}}{\mu_n - \mu_{n-1}} \tag{6}$$

Rather than iterating over transient, we focus on superstable cycles, which are those that contains 0.5 in their orbit. An orbit is a sequence $\{x_0, x_1, \ldots, x_{N-1}\}$ where $x_0 = x_N$. Consequently, we aim to estimate Feigenbaum Delta by finding superstable values. The reason for choosing $x_0 = 0.5$ is that convergence to the cycle is attained fast. Practically, we are just looping over short iterations to get pass the transient.

Estimation is feasible when μ_i is replaced by m_i in the Feigenbaum Delta expression, where m_i is the period-N cycle ($N = 2^N$) with $x_0 = 0.5$ in the orbit. Analytically, it is easy to find the first pair (m_0, m_1) when x_0 is initialized to 0.5:

$$m_0 : x_0 = \mu x_0(1 - x_0) \implies 0.5 = 0.5\mu(0.5)$$

Clearly, $(m_0, \mu) = (2, 2)$.

For the second period (m_1) a system of equations is derived for (x_0, x_1):

$$\begin{cases} x_1 = \mu x_0(1 - x_0) \\ x_0 = \mu x_1(1 - x_1) \end{cases}$$

Substituting, a third degree polynomial equation is derived:

$$\mu^3 - 4\mu^2 + 8 = 0$$
$$m_1 : \mu = 1 + \sqrt{5}$$

The solution vector to the equation above is $(2, 1-\sqrt{5}, 1+\sqrt{5})$. Since we already knew 2 was a solution and $m_i \geq 0 \; \forall i$ under our restrictions, then $m_1 = 1 + \sqrt{5}$.

Following an iterative process we can obtain $\{m_0, m_1, m_2, \ldots, m_{11}\}$ by solving:

$$\delta = \lim_{n \to \infty} \frac{m_{n-1} - m_{n-2}}{m_n - m_{n-1}} \tag{7}$$

Note how we computed (m_0, m_1) from a set of equations. Thus, we can define a root-finding problem to retrieve the rest of $\{m_j\}_{j=2}^{N}$. Newton–Raphson Method or Newton's Method is an efficient Algorithm of order two that produces successively better approximations to the roots of a real-valued function.

Parting from an initial value x_0, the t-th iteration value $x^{(t)}$ yields a value $f(x_t)$ for which we can draw a tangent line that intersects the x-axis such that $f'(x^{(t)}) = 0$. Then, from the tangent line equation:

$$y^{(t+1)} - y^{(t)} = f'(x^{(t)})(x^{(t+1)} - x^{(t)})$$
$$-f(x^{(t)}) = f'(x^{(t)})(x^{(t+1)} - x^{(t)})$$

Consequently:

$$x^{(t+1)} = x^{(t)} - \frac{f(x^{(t)})}{f'(x^{(t)})} \tag{8}$$

This holds since $y^{(t+1)} = f(x^{(t+1)}) = 0$ as we stated that the y-axis mapping of $x^{(t+1)}$ should be zero, and clearly $y^{(t)} = f(x^{(t)})$. Below, a graphical representation of our derivation is presented.

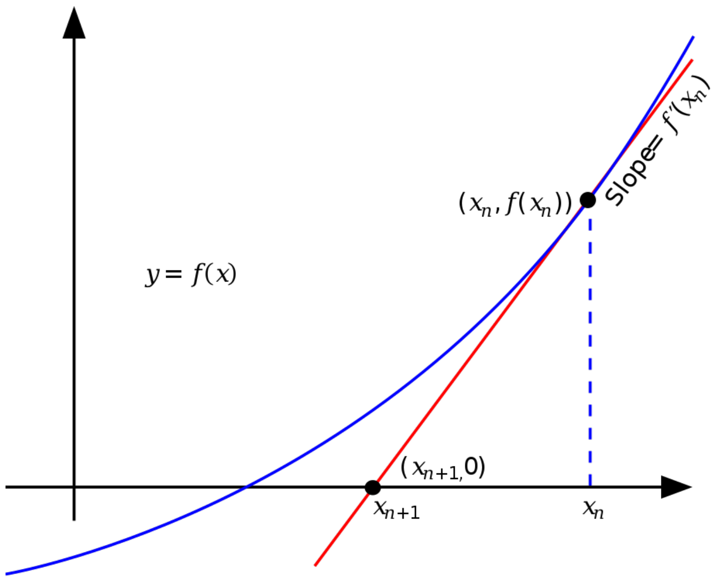

Figure 2: Newton-Raphson Algorithm

We can also show that it is indeed a computationally efficient Algorithm in comparison to others such as the Midpoint or Secant Methods.

Let us define the error as following:

$$\begin{aligned}
\epsilon^{(t)} &= r - x^{(t)} \\
r - x^{(t+1)} &= r - x^{(t)} + \frac{f(x^{(t)})}{f'(x^{(t)})} \quad \text{(error of next iteration)} \\
\epsilon^{(t+1)} &= \epsilon^{(t)} + \frac{f(r-\epsilon^{(t)})}{f'(r-\epsilon^{(t)})} \\
&= \epsilon^{(t)} + \frac{f(r) - \epsilon^{(t)} f'(r) + \frac{\epsilon^{2(t)}}{2!} f''(r) + \dots}{f'(r) - \epsilon^{(t)} f''(r) + \dots} \quad \text{(Taylor Series Expansion)} \\
&= \epsilon^{(t)} + \frac{-\epsilon^{(t)} + \frac{\epsilon^{2(t)}}{2!} f''(r) + \dots}{1 - \epsilon^{(t)} f''(r) + \dots} \qquad \text{since } (f(r), f'(r)) = (0,1) \\
&= \epsilon^{(t)} + \left(-\epsilon^{(t)} + \frac{\epsilon^{2(t)}}{2!} \frac{f''(r)}{f'(r)} + \dots\right)\left(1 + \epsilon^{(t)} \frac{f''(r)}{f'(r)} + \dots\right) \\
&= \epsilon^{(t)} + \left(-\epsilon^{(t)} + \frac{\epsilon^{2(t)}}{2!} \frac{f''(r)}{f'(r)} - \epsilon^{2(t)} \frac{f''(r)}{f'(r)} + \dots\right)
\end{aligned}$$

Lastly:

$$\epsilon^{(t+1)} = -\frac{f''(r)}{2f'(r)}\epsilon^{2(t)} \tag{9}$$

Note that the ratio of derivatives is but a constant, thus:

$$\left|\epsilon^{(t+1)}\right| = k\left|\epsilon^{(t)}\right|^2 \tag{10}$$

Which shows that time complexity for Newton's Method is approximately $O(n^2)$. Another way of reaching the same conclusion is by focusing on Matrix Algebra: the time complexity of updating the Jacobian Matrix for m steps is $O(mn^2 + (m/t)n^3)$, so the time taken per step is $O(n^2 + n^3/t)$, when $t \geq n$ then it is $O(n^2)$.

We can implement this method to our problem by turning it into a root-finding one. Let $x_N \coloneqq \mu x^{(i)}(1 - x^{(i)})$:

$$\begin{cases} g(\mu) = x_N - 1/2 \\ \text{s.t} \;\; x_0 = 1/2 \end{cases} \tag{11}$$

Then we can define a coupled differential equations that are computed together at each iteration:

$$\begin{cases} g'(\mu) = x^{(i)}(1 - x^{(i)}) + \mu x'^{(i)}(1 - 2x^{(i)}) \\ g(\mu) = \mu x^{(i)}(1 - x^{(i)}) - 1/2 \end{cases} \tag{12}$$

Whose initial conditions can be set to $(x_0, x'_0) = (0.5,\ 0)$. Then $\mu^{(t+1)}$ can be calculated as following:

$$\mu^{(t+1)} = \mu^{(t)} - \frac{x_N - 1/2}{x'_N} \tag{13}$$

Said parameter requires an initial guess $\mu^{(0)} = m^{(t)}$, which we supply by using an approximate value for $\delta^{(t)}$ by isolating m_n from the Feigenbaum Delta presented above. Ditching out the limiting behavior of said constant, we get a rough estimate for m_n as our initial guess for $\mu^{(0)}$. The fact that δ_{n-1} is present forces us to set an initial value for δ_1.

$$\delta = \frac{m_{n-1} - m_{n-2}}{m_n - m_{n-1}} \implies m_n \approx m_{n-1} + \frac{m_{n-1} - m_{n-2}}{\delta_{n-1}}$$

Wrapping everything up, the Algorithm is as follows:

```
Algorithm 2: Feigenbaum Delta
Input: Estimate Iterations $n$, Newton Iterations $p$
Output: $\hat{\delta}$ Feigenbaum Delta Estimate
# Initialize Early Iteration Parameters
$m_0 = 2, \quad m_1 = 1 + \sqrt{5}, \quad \delta_1 = 5$
# Loop over number of estimates
for $t = 2, \ldots, n$ do
    $\mu^{(0)} = m^{(t-1)} + \frac{m^{(t-1)} - m^{(t-2)}}{\delta^{(t-1)}}$
    # Loop until convergence in Newtons' Method
    for $i = 1, \ldots, n$ do
        $x^{(0)} = 1/2, \quad x'^{(0)} = 0$
        # Loop over each period-N
        for $j = 1, \ldots, 2^t$ do
            ${x'_N}^{(t+1)} = x^{(t)}(1 - x^{(t)}) + \mu^{(0)} x'^{(t)}(1 - 2x^{(t)})$
            $x_N^{(t+1)} = \mu^{(0)} x^{(t)}(1 - x^{(t)})$
        end
        $g'(\mu^{(t+1)}) = \frac{x_N^{(t+1)} - 1/2}{{x'_N}^{(t+1)}}$
        $\mu^{(t+1)} = \mu^{(0)} - g'(\mu^{(t+1)})$
    end
    # Update final values
    $m^{(t)} = \mu^{(t+1)}$
    $\delta^{(t)} = \frac{m^{(t-1)} - m^{(t-2)}}{m^{(t)} - m^{(t-1)}}$
end
```

Our `MATLAB` implementation uses variable-precision floating-point arithmetic (VPA) to increase computational precision. A similar package could be used in `Python`, namely *decimal*, however it does not support numpy arrays data types. A workaround was found by resorting to numpy's *format_float_positional* function, yielding the same results as in `MATLAB`. The final estimates are:

Table 1: Feigenbaum Delta

Iteration t	$m^{(t)}$	$\delta^{(t)}$
0	2	5
1	$1+\sqrt{5}$	4.708943013540505
2	3.4985616993277016	4.680770998010699
3	3.554640862768825	4.6629596111141085
4	3.5666673798562685	4.668403925917214
5	3.569243531637111	4.668953740968109
6	3.5697952937499453	4.6691571813815305
7	3.569913465422348	4.669191002146906
8	3.5699387742333064	4.669199473291195
9	3.569944194608063	4.669201107217582
10	3.569945355486473	4.669201677505845

3 Chaos Theory: Lorenz Equations

Having analyzed a one-dimensional root-finding problem with Newton-Raphson Algorithm, it is but natural to increase the dimensional space. Let us do it by studying the Lorenz Equations.

Lorenz (1953) proposed a simplified mathematical model for atmospheric convection, in which properties of a two-dimensional fluid layer uniformly warmed from below and cooled from above were examined. He set up a system of ordinary differential equations and noticed a set of chaotic solutions given some parameter choices and initial conditions. We now propose a Numerical Method to obtain fractals from Lorenz Equations by computing fixed points.

Firstly, we present the general framework for a three-dimensional root-finding problem. Then, we particularize said model to the system of ordinary differential equations that make up the atmospheric convection model developed by Lorenz.

Lorenz Equations are as following:

$$\begin{cases} \dfrac{\partial f(x,y,z)}{\partial t} = \sigma(y-x) \\ \dfrac{\partial g(x,y,z)}{\partial t} = x(\rho - z) - y \\ \dfrac{\partial h(x,y,z)}{\partial t} = xy - \beta z \end{cases} \tag{14}$$

So we seek to find $\dot{x} = \dot{y} = \dot{z} = 0$, where these letters represent the first derivative of the functions above with respect to t. More generally, such problem can be solved by a system of Taylor Series Expansion after t iterations:

$$f(x^{(t+1)}, y^{(t+1)}, z^{(t+1)}) = f^{(t)} + \Delta x^{(t)} \frac{\partial f^{(t)}()}{\partial x^{(t)}} + \Delta y^{(t)} \frac{\partial f^{(t)}()}{\partial y^{(t)}} + \Delta z^{(t)} \frac{\partial f^{(t)}()}{\partial z^{(t)}}$$
$$g(x^{(t+1)}, y^{(t+1)}, z^{(t+1)}) = g^{(t)}() + \Delta x^{(t)} \frac{\partial g^{(t)}()}{\partial x^{(t)}} + \Delta y^{(t)} \frac{\partial g^{(t)}()}{\partial y^{(t)}} + \Delta z^{(t)} \frac{\partial g^{(t)}()}{\partial z^{(t)}}$$
$$h(x^{(t+1)}, y^{(t+1)}, z^{(t+1)}) = h^{(t)}() + \Delta x^{(t)} \frac{\partial h^{(t)}()}{\partial x^{(t)}} + \Delta y^{(t)} \frac{\partial h^{(t)}()}{\partial y^{(t)}} + \Delta z^{(t)} \frac{\partial h^{(t)}()}{\partial z^{(t)}}$$

Where all functions $f^{(t)}(), g^{(t)}(), h^{(t)}()$ are evaluated at $x^{(t)}, y^{(t)}, z^{(t)}$ and the new vector is defined as $(\Delta x^{(t)}, \Delta y^{(t)}, \Delta z^{(t)}) = (x^{(t+1)} - x^{(t)}, y^{(t+1)} - y^{(t)}, z^{(t+1)} - z^{(t)})$.

Then, leveraging Matrix Algebra we can rewrite the problem as:

$$\begin{bmatrix} \dfrac{\partial f^{(t)}(x,y,z)}{\partial x} & \dfrac{\partial f^{(t)}(x,y,z)}{\partial y} & \dfrac{\partial f^{(t)}(x,y,z)}{\partial z} \\ \dfrac{\partial g^{(t)}(x,y,z)}{\partial x} & \dfrac{\partial g^{(t)}(x,y,z)}{\partial y} & \dfrac{\partial g^{(t)}(x,y,z)}{\partial z} \\ \dfrac{\partial h^{(t)}(x,y,z)}{\partial x} & \dfrac{\partial h^{(t)}(x,y,z)}{\partial y} & \dfrac{\partial h^{(t)}(x,y,z)}{\partial z} \end{bmatrix} \begin{bmatrix} \Delta x^{(t)} \\ \Delta y^{(t)} \\ \Delta z^{(t)} \end{bmatrix} = - \begin{bmatrix} f(x^{(t)}, y^{(t)}, z^{(t)}) \\ g(x^{(t)}, y^{(t)}, z^{(t)}) \\ h(x^{(t)}, y^{(t)}, z^{(t)}) \end{bmatrix}$$

Note that every function evaluated at the $t+1$-th iteration, namely, the left-hand-side of the Taylor Series Expansion, is set to zero as we seek to obtain fixed points. Computing the delta vector $(\Delta x^{(t)}, \Delta y^{(t)}, \Delta z^{(t)})$ allows to solve for $x^{(t+1)}, y^{(t+1)}, z^{(t+1)}$. Thus, the Jacobian matrix (the partial derivatives matrix) has to be inverted, for which it is required to be nonsingular. In other words, the Jacobian cannot contain linear combination of rows/columns, so as to guarantee that it is of full-rank.

Particularizing for our problem we need to solve:

$$\begin{bmatrix} -\sigma & \sigma & 0 \\ \rho - z^{(t)} & -1 & -x^{(t)} \\ y^{(t)} & x^{(t)} & -\beta \end{bmatrix} \begin{bmatrix} \Delta x^{(t)} \\ \Delta y^{(t)} \\ \Delta z^{(t)} \end{bmatrix} = - \begin{bmatrix} \sigma(y^{(t)} - x^{(t)}) \\ x^{(t)}(\rho - z^{(t)}) \\ x^{(t)}y^{(t)} - \beta z^{(t)} \end{bmatrix} \tag{15}$$

In other words:

$$\begin{bmatrix} x^{(t+1)} \\ y^{(t+1)} \\ z^{(t+1)} \end{bmatrix} = - \begin{bmatrix} -\sigma & \sigma & 0 \\ \rho - z^{(t)} & -1 & -x^{(t)} \\ y^{(t)} & x^{(t)} & -\beta \end{bmatrix}^{-1} \begin{bmatrix} \sigma(y^{(t)} - x^{(t)}) \\ x^{(t)}(\rho - z^{(t)}) \\ x^{(t)}y^{(t)} - \beta z^{(t)} \end{bmatrix} \tag{16}$$

For such task we resort to Newton's Method, setting initial values to a grid such that $(x_0, z_0) \in (-40, 40)$ while $y_0 = 3\sqrt{2}$. We follow a classical procedure by letting the original choice for $(\sigma, \rho, \beta) = (10, 28, 8/3)$ made by Lorenz (1963) be our parameter vector.

Solving the system of ordinary differential equations (ODE) and plotting the results for (x, y, z) yields the following graph:

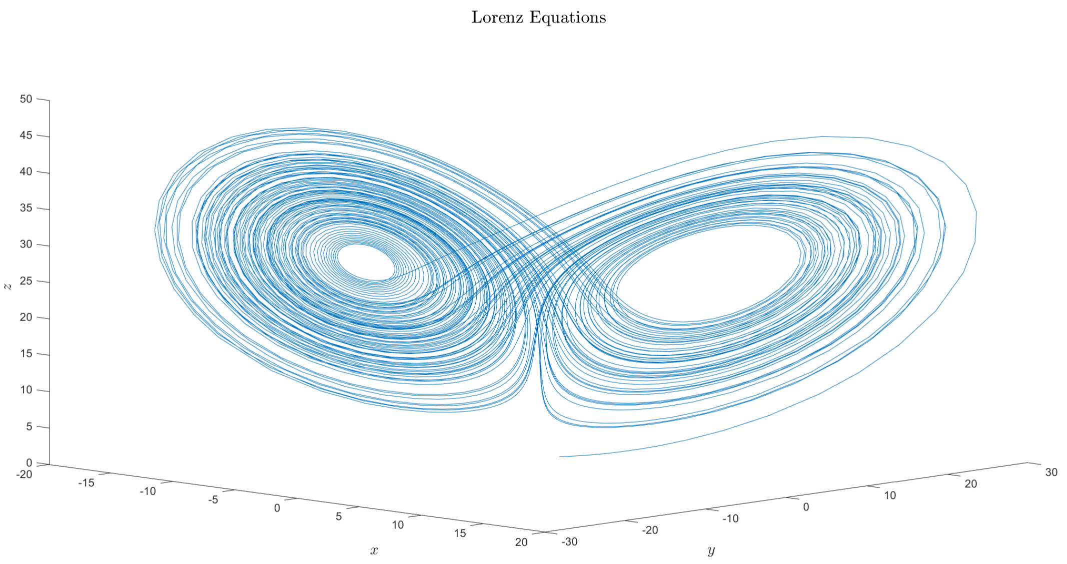

Figure 3: Lorenz Equations

It can be easily noticed the chaotic behavior of said system. We know specifically analyze the fractal values for the x-axis and z-axis.

This time so as to maximize efficiency, instead of setting a fixed number of iterations for Newton's Method, we iterate until convergence. Apart from inverting a matrix, there is a nested for loop within our implementation in both, `Python` and `MATLAB`, which increases the time complexity of our Algorithm to $O(nm + m^3)$. Since values close to fractals are more computationally expensive, in the sense that they require more Newton's Method iterations, we ought to save time for the fast converging values. This is attained by choosing a tolerance parameter that indicates whether or not the Algorithm has converged.

Algorithm 3: Fractals from the Lorenz Equations

```
Input: Parameters $\sigma, \rho, \beta$, Data matrices $\mathbf{X}$, $\mathbf{Z}$ of size $n \,\mathrm{x}\, m$
Output: Fractals $x^{(t+1)}, z^{(t+1)}$
# Initialize Parameters
$\sigma = 10, \quad \rho = 28, \quad \beta = 8/3$
# Define matrix of initial values
$\mathbf{x}_0 = [-40, \dots, 40], \qquad \mathbf{z}_0 = [-40, \dots, 40]$
$\mathbf{X}, \mathbf{Z} = \mathrm{meshgrid}(\mathbf{x}_0, \mathbf{z}_0)$
for $i = 1, \dots, n$ do
    for $j = 1, \dots, m$ do
        # Initialize $y_0$
        $y_0 = 3\sqrt{2}$
        while $|x^{(t+1)} - x^*| > \epsilon$ or $|y^{(t+1)} - y^*| > \epsilon$ or $|z^{(t+1)} - z^*| > \epsilon$ do
            # Newton's Method
            $$\begin{bmatrix} x^{(t+1)} \\ y^{(t+1)} \\ z^{(t+1)} \end{bmatrix} = - \begin{bmatrix} -\sigma & \sigma & 0 \\ \rho - z^{(t)} & -1 & -x^{(t)} \\ y^{(t)} & x^{(t)} & -\beta \end{bmatrix}^{-1} \begin{bmatrix} \sigma(y^{(t)} - x^{(t)}) \\ x^{(t)}(\rho - z^{(t)}) \\ x^{(t)}y^{(t)} - \beta z^{(t)} \end{bmatrix}$$
        end
    end
end
```

When represented, the fractals from Lorenz Equations justify why said system of differential equations is viewed as one of the typical Chaos Theory examples. The almost stochastic-like behavior of the regions yield interesting results.

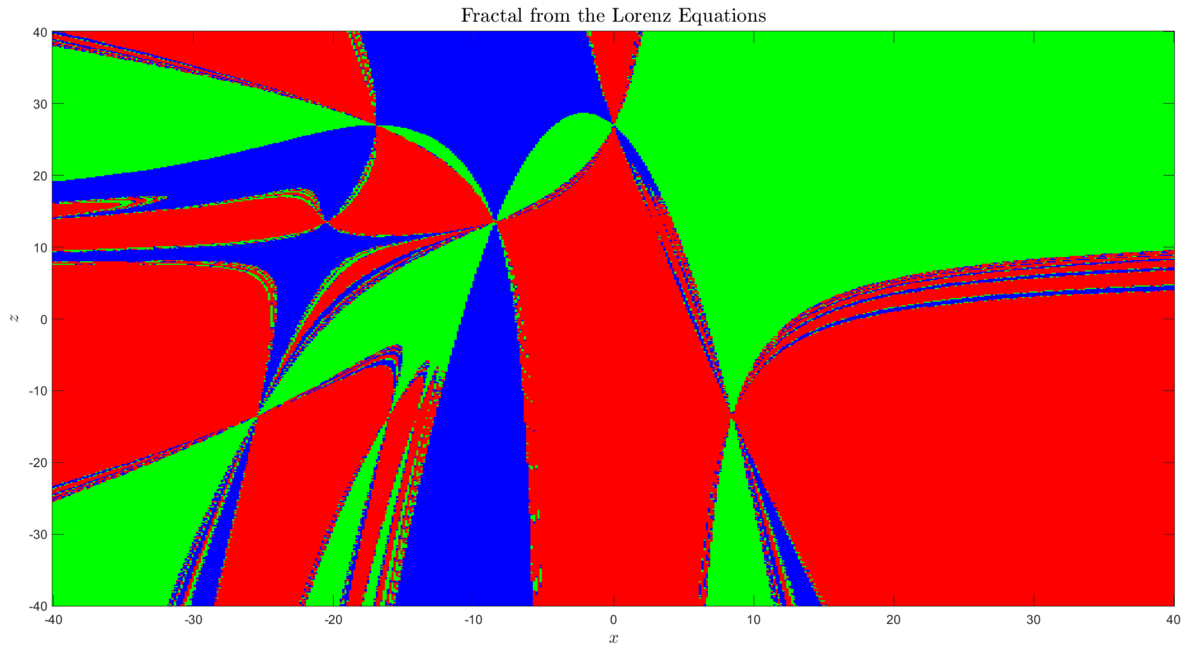

Figure 4: Fractals from Lorenz Equations

4 Numerical Integration: Bessel Functions

We now focus on numerical integration by analyzing a generalization of the sine function: the Bessel function. Concretely, we derive graphical representations and mathematical expressions for Bessel functions of the first kind. These can be interpreted as vibration of a string with variable thickness, variable tension or both. Nevertheless, its applications also extends to fields such as Statistics, since Bessel functions can be thought of as the probability density function of the product of two normally distributed random variables.

Bessel functions are canonical solutions of Bessel's differential equation:

$$x^2\frac{\partial^2 y}{\partial x^2}+\frac{\partial y}{\partial x}+(x^2-\alpha^2)y \tag{17}$$

Which can be expressed as

$$J_\alpha(x) = \sum_{n=0}^{\infty} \frac{-1^n}{n!(n+\alpha+1)} \left(\frac{x}{2}\right)^{(2n+\alpha)} \tag{18}$$

If $n \in \mathbb{Z}$, then it is possible to represent $J_n(x)$ as an integral from the Hansen-Bessel Formula:

$$J_n(x) = \frac{1}{2\pi}\int_{-\pi}^{\pi} e^{ix\cos(\theta)} e^{in(\theta-\pi/2)}\,\partial\theta = \frac{i^{-n}}{\pi}\int_0^{\pi} e^{ix\cos(\theta)}\cos(n\theta)\,\partial\theta$$

For the sake of convenience, we represent the result above in terms of cosine and sine functions:

$$J_n(x) = \frac{1}{\pi}\int_0^{\pi} \cos(x\sin(\theta) - n\theta)\,\partial\theta \tag{19}$$

In order to integrate said function we resort to Adaptive Quadrature, a Numerical Method that approximates the integral of $J_n(x)$ using static quadrature rules on adaptively refined subintervals of the region of integration.

Let $f(x)$ be a generic continuous function and $[a,b]$ the integration region such that $c \in [a,b]$. We obtain intagral approximations for the first and second level using the trapezoidal rule:

$$\begin{aligned} I_1 &= \int_a^b f(x)\,\partial x = \frac{h}{2}\left(f(a)+f(b)\right) - \frac{h^3}{12}f''(\xi) \\ I_2 &= \int_a^b f(x)\,\partial x = \frac{h}{4}\left(f(a)+2f(c)+f(b)\right) - \frac{h^3}{2^3 * 12}f''(\xi_-) - \frac{h^3}{2^3 * 12}f''(\xi_+) \end{aligned}$$

Splitting approximations and errors:

$$\begin{aligned} S_1 &= \frac{h}{2}\left(f(a)+f(b)\right) & E_1 &= -\frac{h^3}{12}f''(\xi) \\ S_2 &= \frac{h}{4}\left(f(a)+2f(c)+f(b)\right) & E_2 &= -\frac{h^3}{2^3 * 12}f''(\xi_-) - \frac{h^3}{2^3 * 12}f''(\xi_+) \end{aligned}$$

Suppose all second derivatives are equal, then $E_1 = 4E_2$. Consequently:

$$S_1 + E_1 = S_2 + E_2$$
$$|E_2| = \frac{|S_2 - S_1|}{3}$$

This last expression is the approximation error for the second level. While it is larger than some tolerance parameter ϵ, the Adaptive Quadrature Algorithm will keep computing deeper levels so as to attain a precise approximation.

Having explained the algorithmic mechanism behind numerical integration, we know show graphical representation for the first five Bessel functions of the first kind:

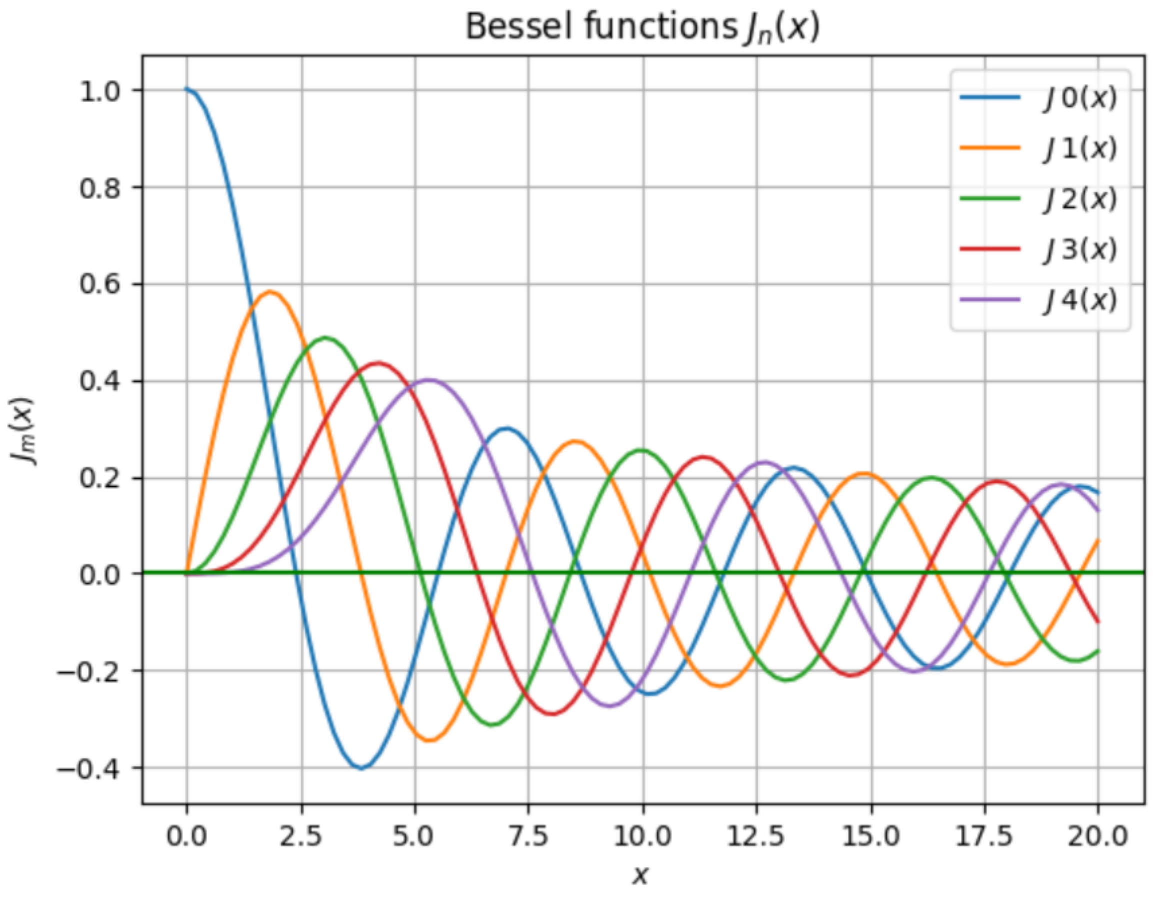

Figure 5: Fractals from Lorenz Equations

It is now time to obtain the precise values for which $J_n(x) = 0$, for which we resort to our previously developed root-finding Algorithm.

```
Algorithm 4: Zeros of Bessel Functions
Input: Initial guess $n \text{x} m$ matrix $\mathbf{X}$, Number of roots $m$, Number of functions $n$
Output: Solutions to $J_n(x) = 0$
# Compute roots for each $J_n(x)$
for $i = 1, \ldots, m$ do
    for $j = 0, \ldots, n-1$ do
        # Define $n$-th function
        $J_n(x) = \frac{1}{\pi}\int_0^{\pi} \cos(x\sin(\theta) - n\theta)\,\partial\theta$
        # Set initial values and solve
        $J_n(x_{ij}) = 0$
    end
end
```

Our `MATLAB` implementation exactly follows the procedure presented above. For `Python`, we make use of the *scipy* package, which offers a function that already computes the roots for Bessel Functions.

Table 2: Zeros of Bessel Functions

n	$J_0(x)$	$J_1(x)$	$J_2(x)$	$J_3(x)$	$J_4(x)$	$J_5(x)$
1	2.4048	3.8317	5.1356	6.3801	7.58834	8.7714
2	5.5200	7.0155	8.4172	9.7610	11.0647	12.3386
3	8.6537	10.1734	11.6198	13.0152	14.3725	15.7001
4	11.7915	13.3236	14.7959	16.2234	17.6159	18.9801
5	14.93091	16.4706	17.9598	19.4094	20.8269	22.2177

Comparing the output table above and the fifth figure, one can clearly see that our approximations are close to the roots. This time our `Python` implementation was almost as fast as `MATLAB` since much of *scipy* is implemented as `C` extension modules. Moreover, Adaptive Quadrature did not need to go deep into too many levels, since absolute tolerance was lowered at early stages of the Algorithm, resulting in fast and precise integral approximations.