An Investigation of the Thomas Attractor

Introduction

An **Attractor** is defined as a region of space to which points are pulled into as they evolve over time. The shape of that region of attraction of course varies depending on the parameters one uses to build the attractor. Naturally, with such a vague definition, there end up being quite a few different systems that end up falling under this definition. For example, orbital motion can be defined as one such attractor with the gravitational bodies pulling one another into a closed orbital region (that is, assuming that one of the orbital bodies does not break out of their orbit). An attractor can even be something as simple as the 2-d system of a ball trapped between two hills. Here, the hills can be considered to be the attractor with the region acting to "pull" the ball towards the lowest point on the hill.

One such subset of the class of equations are known as **Strange Attractors**. Developed in 1963 by Edward Lorenz, these systems are much like ordinary attractors except for the fact that they exhibit *fractal* behavior, that is to say that the system itself is self-similar, where the shape of the entire system is similar to smaller sections of the shape. The first and most well-known of these types of attractors are known as **Lorenz Attractors**. These systems, aptly named after their creator, were used as a means of generating a simplified model of atmospheric convection. Since then, a large number of different variants of strange attractors have been created, a large number of which have not yet been found to represent any sort of real physical phenomena, but have become somewhat popularized due to the unique shapes they form from the paths of particles within their field of influence.

The **Thomas Attractor** (or to be more accurate, *Thomas' Cyclically Symmetric Attractor*) is one such attractor. However, unlike most of the other strange attractors, this system actually does have some form of physical analogue. Proposed by Belgian scientist Rene Thomas, the region is a cylically symetric area meant to define the motion of a particle moving through a 3D lattice of forces as its motion becomes dampened by friction. Thus, we can break this motion down into two main components, both of which depend soley on the position of the particle in x, y and z:

1. Driving Force - defined by a sin wave, designed to make the particle move chaotically throughout the space
2. Frictional Force - Depends linearly on position. Is the main force that restricts the particle to a defined area

For the most part, this strange attractor, like most of the other strange attractors that have been found after the Lorenz Attractor, has proven to be quite useful in testing various different kinds of mathematical analysis methods due to how simple the system is.

However, outside the world of math, due to the fact that there is no explicit physical phenomena that this attractor is used to describe (or at least no physical phenomena that are not more accurately described by another model), these attractors are mostly valued for the particularly interesting patterns they form at certain values of the friction coefficient.

Numerical Solution

The equation for the numerical solution defining this region is actually quite simple and can be defined by numerically integrating the following set of equations:

$$\frac{dx}{dt} = \sin(y) - bx$$

$$\frac{dy}{dt} = \sin(z) - by$$

$$\frac{dz}{dt} = \sin(x) - bz$$

Where x, y and z represent the unitless spatial location of the particle.

The constant b defines how *dissipative* the system is (that is to say, how strong the friction is), by varying this value, we can significantly alter the state in which the attractor behaves.

To be more precise, there are 6 distinct behaviors that this attractor displays as we vary the value of b:

$b > 1$: There is a single attractor at the origin in a stable equilibrium

$b = 1$: The system goes through *pitchfork bifurcation* where the main attractor splits off to form two separate attractors opposite from one another

$b = 0.32899$: The point at which the system undergoes a *Hopf bifurcation*, after this point the system looses its stability

$b \approx 0.208186$: The system enters chaotic motion

$b = 0$: There isn't any kind of frictional force, which causes the particles within they attractor to move randomly (or as the Wikepidia says: *ergodically*)

$b < 0$: Everything explodes. Frictional coefficients shouldn't be negative

Analytical Solution

Due to the fact that this system is restricted to a 3-dimensional space, there is no general equation in terms of an arbitrary number of dimensions that is able to accurately represent this system.This means that it is impossible to generate a set of exact analytical soltions for this problem as attempting to solve each of the 3 given equations individually will result in a set of equations that still depend on the value of its position in another dimension (for example, trying to solve the equation for the x-dimension will result in an equation that depends on both the y position and t), which isn't a very usable in terms of being able to generate sets of solutions as they evolve over time.

Methodology/ Results

In this investigation, we will be relying on some of the standard python libraries along with the odeint() function from the scipy library.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
```

We also import some additional libraries to generate some sliders in some of our plots.

In [2]:
```
from IPython.display import display, clear_output
import time
```

Additionally, to help better visualize our 3d plots, we will be using the plotly graph objects library to make interactive 3d plots that are much easier to understand.

In [3]:
```
import plotly.graph_objects as go
```

Below, we show the equation we will be using for the investigation

In [4]:
```
def thomas_attractor(r, t, b):
    """
    Arguments:
        r: a list of the position values x, y and z represent the particl
        current position within the system

        t: represents the current time, only included so odeint() runs
        properly. Has no actual use inside the function

        b: the friction coeffitient, varying this value greatly alters ho
         the particle acts within the system. Can be set to any value gre
         than 0

    Returns:
        [xdot, ydot, zdot]: a list of the derivatives of x, y and z respe
    """
    x, y, z = r[0], r[1], r[2]

    xdot = np.sin(y) - b*x
    ydot = np.sin(z) - b*y
    zdot = np.sin(x) - b*z

    return [xdot, ydot, zdot]
```

Phase Plots

We will begin y first generatng a few phase plots of the system at different values of b. To do this we must first account for the fact that we are working in 3 spatial dimensions rather than two. There are 2 ways we can go about accounting for this:

1. Generate a 3D phase plot:

    - While very much possible and usefull via the plotly cone plot function, attempted use of this function has resulted in some very messy plots that do not reveal very much information about the phase of the system at all
2. Generate an animation of a 2D phaseplot of the xy-plane as we adjust the value along the z-axis
    - This method, while somewhat limited in terms of helping us understand the overall phase, these plots do provide very accurate information about the phase of each cross section at different values of z that, when combined with a slider to change the z values, helps us at least gain some understanding for at to how the different xy-plane phaseplots relate to one another, and by extension, how the overall phase might behave

For the sake of this investigation, we will only be generating animations for 4 different values of b: 1, 0.32899, 0.208186 and 0, as they generate the phaseplots with the most unique features.

In [5]:

```python
def animate_streamplot(z_max, b):
    """
    A function I made to animate streamplots with the thomas_attractor()
    function plots an animation of the streamplot in the xy-plane
    at different values of z

    Arguments:
        b: determines the strength of friction within the system

        z_max: value of the z-bounds of the system to which the animatio
        iterates towards

    Returns:
        an animation of the phase plot that iterates from -z_max to +z_m
    """
    # defining array of z values
    z_arr = np.linspace(-z_max,z_max,31)

    # defining arrays for x and y
    x = np.linspace(-4, 4, 200)
    y = np.linspace(-4, 4, 200)

    # turning them into meshgrid objects
    X,Y = np.meshgrid(x, y)

    fig = plt.figure()
    ax = plt.axes()

    # for loop to run animation
    for frame in range(len(z_arr)):
        u = thomas_attractor([X,Y,z_arr[frame]], 0, b)
        x_dot = u[0]
        y_dot = u[1]

        # generating streamplot for the frame
        ax.streamplot(X, Y, x_dot, y_dot)

        # setting axis labels/titles
        ax.set_xlabel("X", fontsize = 15)
        ax.set_ylabel("Y", rotation = 0, fontsize = 20)
        ax.set_title(rf"$z$ = {np.round(z_arr[frame], 2)}", fontsize = 2

        # prevents last frame from being deleted
        if z_arr[frame] == z_arr[-1]:
            break
        else:
            display(fig)
            time.sleep(0.00001)
            plt.cla()
            clear_output(wait=True)
```

In [6]:
```python
# streamplot animation at b = 1
z_max = 4
b = 1
animate_streamplot(z_max, b)
```

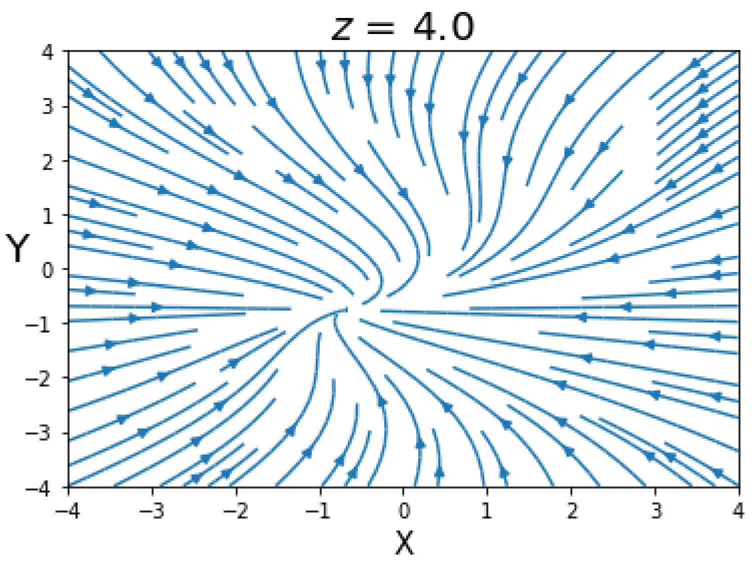

Fig 1. For the b value of 1, the lines of the plots essentially converge onto a single point around the origin that only slightly changes as we vary z, and as we increase the value of b beyond 1, the magnitude of this oscillation only decreases. So, for the sake of avoiding redundancy, these animations have not been included.

In [7]:
```python
# streamplot animation at b = 0.32899
z_max = 4
b = 0.32899
animate_streamplot(z_max, b)
```

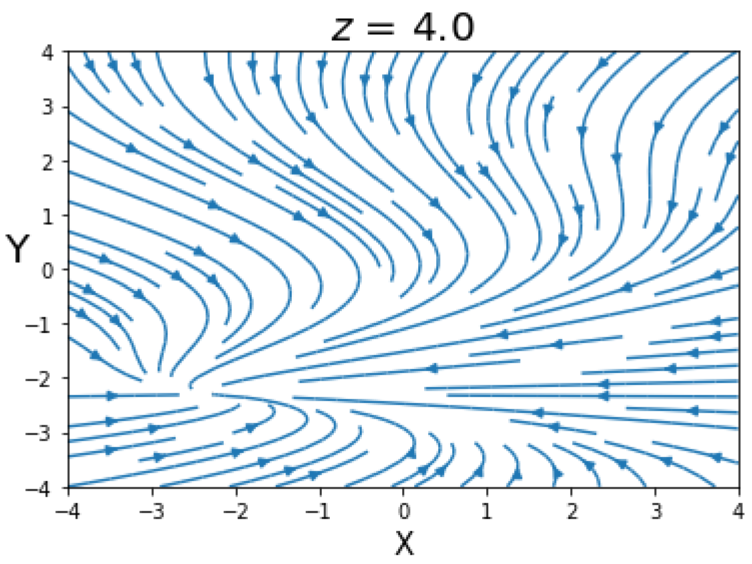

Fig 2. In this plot we immediately notice that the entirety of the x-y phase cross-section converges onto a singlular point. As we vary the z value, we see that this convergance point does not dissapear, but begins to move in a vaguely sinusoidal fashion. From $z = -4$, the point lies in the top right corner of the plot where it then moves to the bottom left by about $z = -1.33$, then back to the top left by $z = +1.33$ then back to the bottom left by $z = +4$.

In [8]:
```python
# streamplot animation at b = 0.208186
b = 0.208186
animate_streamplot(z_max, b)
```

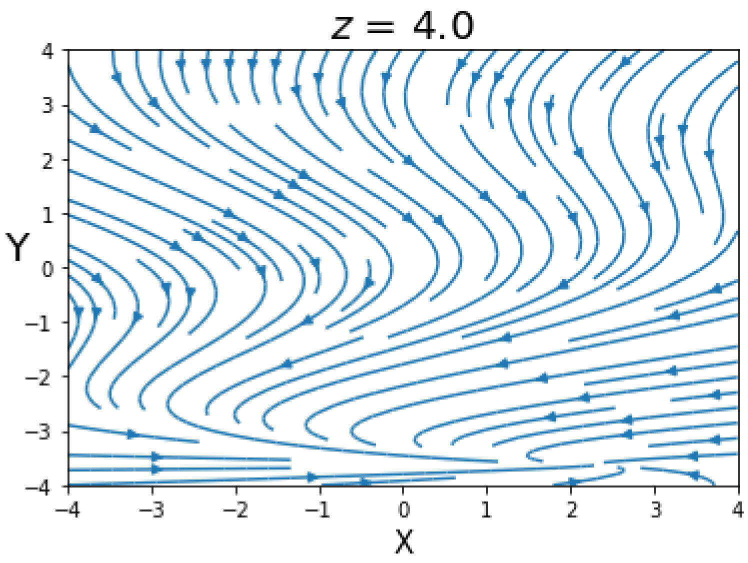

Fig 3. Much like in the figure above, we also see that the plot consistently converges onto a single point. However, in this case, we see that this point moves far more erratically throughout the space, as if the sinusoidal path that the **Fig 2** convergence point followed had its amplitude drastically increased. It should also be noted that instead of oscillating between the top right and bottom left corners, the convergence point now moves between the top left and bottom right corners of the sytem.

In [9]:
```python
# streamplot animation at b = 0
b = 0
animate_streamplot(z_max, b)
```

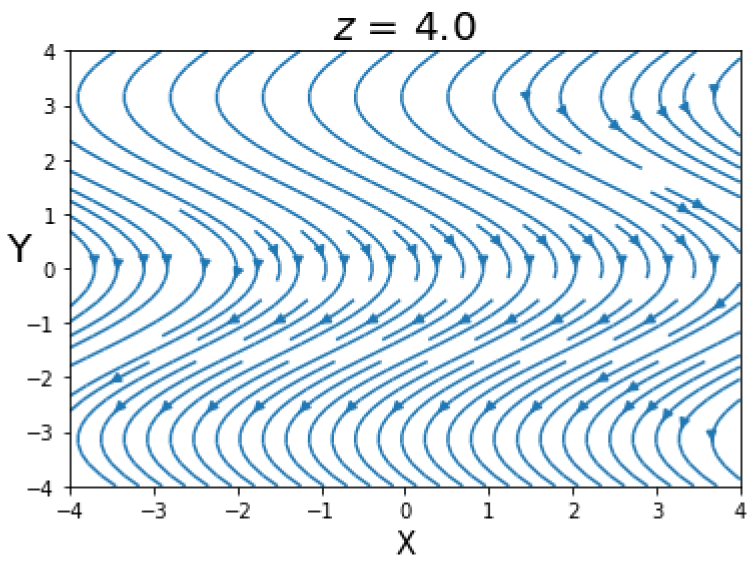

Fig 4. Far from the 2 figures above, this case with no frictional force whatsoever there is no longer any kind of central convergence in the x-y plane for any values of z. Instead, all the phase plot simply shows a series of sin waves for all values of x and y, reguardless of the z value. The only noticable change that we see as the z-value is varied is that the amplitude of the x-y sin oscillations tend to increase and decrease as we alter the value of z. Additionally, the phase plot cross sections also indicate that when z is negative, the sin waves propagate down the y-axis, but when z is positive, the plot flips and the sin waves face in the positive y-direction.

Full Plots of Numerical Solutions

Now we will generate full plots using the numerical solutions of our attractor. Fortunately, unlike with the phase plots that we created in the section above, we are now able to utilize the *plotly* library to generate interactive 3-dimensional plots of our solutions which will enable us to perform a much more in-depth analysis than if we had been simply utlizing a regular matplotlib 3-D plot.

By using *plotly* we are also provided the opportunity to make our plots much more concise by utilizing the slider functionality within the library (matplotlib has something like this, but it's really inefficient) to include plots for all values of b that we desire witin a single graph object. That being said, in the figure below, we have created a 3D plot of the numerical solutions of the Thomas Attractor for all of the different behaviors (indicated in the introduction) of the plot from $b = 0$ to $b = 1$, with some extra b values included in between these different b values to help build some intuition for how the figure evolves as we change the amount of friction within the system.

As for the initial conditions we choose, due to the fact that we are working with an attractor, having very precise initial conditions isn't too important as any point we choose will end up moving into the pattern which the attractor dictates. The only way that the initial conditions can affect the features of the plot is if there were multiple *attractive wells* to which the particle could end up being pulled towards.

Fortunately, by looking at the phase plots we generated above and at the existence of the moving *convergence point* in some of the x-y plane phase plots, we can tell that, at least for some values of b, there are multiple nodes. As for how many there are and the range of their respective fields of influence, its a little hard to tell. Thus, we had to rely by experimenting a little bit with the initial conditions and using the information given to us by the description of the attractor itself.

By using this information, we are able to determine that, for only some values of b, there are 2 wells, one in the 1st octant and another one in the exact opposite side in the 7th octant.

Note the first plot that comes out of here is kind of a mess, most likely just something weird that goes on with plotly. You might have to move the slider around once to get the plot in a readable state.

Also, if you're running this code on a potato of a computer like mine, it might take a bit before the actual plot appears

In [10]:

```python
# slider addition from https://plotly.com/python/sliders/

# generating list of 2 initial conditions (1 for each attractor)
r0_list = [[-1, -2, -3.5],[1, 2, 3.5]]

# creating figure objet
fig = go.Figure()

# creating list of desired friction coefficients
b_list = [0, 0.5, 0.1, 0.15, 0.208186, 0.25, 0.32899, 0.4, 0.6, 0.8, 1]

# loops through each b value, generates a graph object for each value of
for b in b_list:
    x, y, z = np.array([]), np.array([]), np.array([])
    for r0 in r0_list:
        t_arr = np.linspace(0,1000,10000)
        r = odeint(thomas_attractor, r0, t_arr, args = (b,))
        x = np.append(x, r[:,0])
        y = np.append(y, r[:,1])
        z = np.append(z, r[:,2])

    fig.add_trace(go.Scatter3d(x = x,
                               y = y,
                               z = z,
                               mode = "markers",
                               marker=dict(size=1)))

steps = []

# for each graph object, generates information about those plots
# stores information in a list of dictionaries
for i in range(len(fig.data)):
    step = dict(
        method="update",
        args=[{"visible": [False] * len(fig.data)},
              {"title": "Thomas Attractor at Variable Friction Coefficie
    )
    step["args"][0]["visible"][i] = True  # Toggle i'th trace to "visibl
    step["label"] = b_list[i]
    steps.append(step)

# creates slider object
sliders = [dict(
    active=10,
    currentvalue={"prefix": "b value: "},
    pad={"t": 50},
    steps=steps
)]

# applying the slider object to figure
fig.update_layout(
    sliders=sliders
)

fig.show()
```

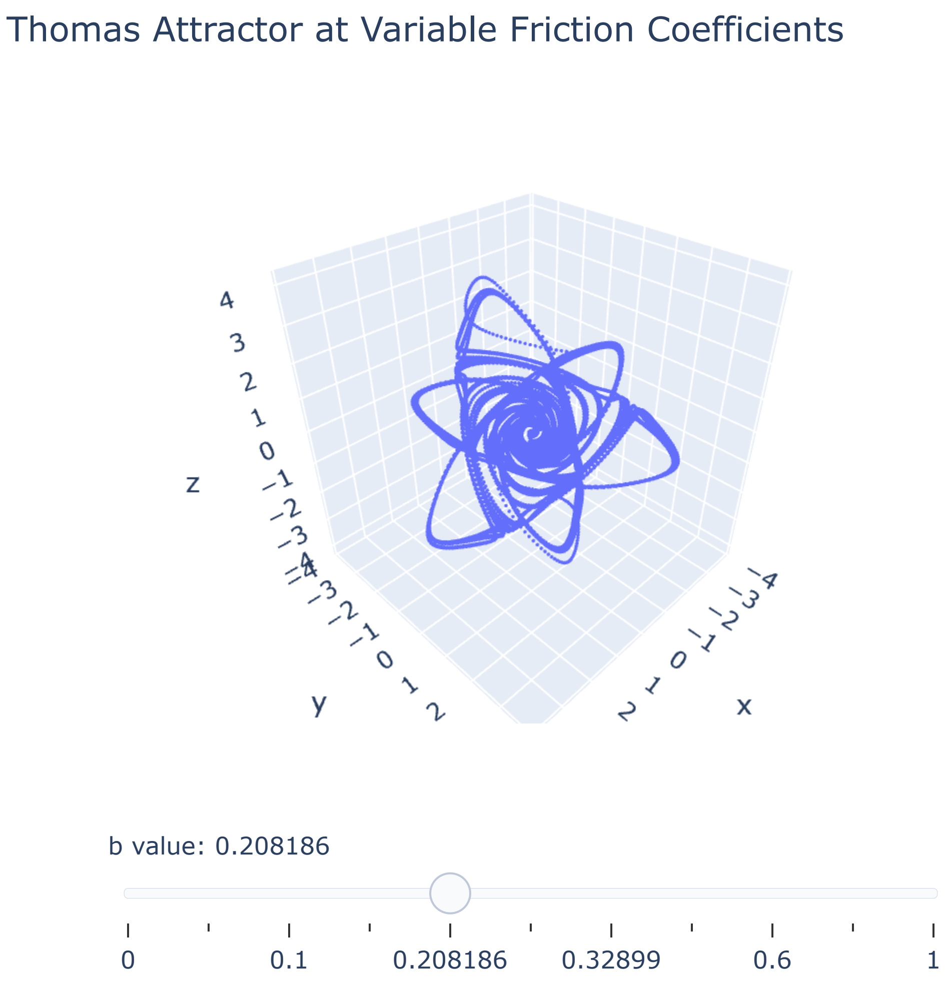

Fig 5. In this figure, we find a very large variety of different values for each b value:

$b = 1$: Is where the system is supposed to undergo *pitchfork bifurcation*, but at this point, the evidence of this bifurcation is almost completely unoticable and it still looks as though the points are very much still attracted to the origin. It isn't until about $b = 0.8$ until the bifurcation begins to become more apparent.

$b = 0.32899$: Where *Hopf bifurcation* occurs, after this point the system looses its stability. A little bit of instability is apparent here, but becomes very obvious at $b = 2.5$.

$b = 0.208186$: Here forms the most popular shape that the Thomas attractor is known for. After this point the plot begins to break down into more and more chaotic motion as we continue to decrease the value of b.

$b = 0$: As expected, without any kind of frictional force, the plot simply moves randomly in a sinusoidal pattern all about the system. We have reached the point where the Thomas Attractor is no long an attractor.

Discussion

Evaluating Accuracy

In this scenario, with no true analytical solutions, and no physical analogue to compare our results to, we will be quite limited in terms of how we can evaluate how effective our results are. As of now, we are only left with the option of making comparisons between the numerical solutions and how effectively they Of course, there is always the method of simply trying to understand how the numerical solutions with the phase plot animations that we created, such a task would be rather wordy and the connections that would need to be made would end up becoming needlessly complicated.

Instead we will simply create a few 3D quiver plots using matplotlib then plot the numerical solutions on top of the plot. With the combination of the information that we gathered from the previous plots and this plot, we will be able to obtain a much more complete picture of how well the numerical solution matches the phase plot.

For this purpose, we will generate the following function:

In [11]:
```python
def check_accuracy(b):
    """
    A function to check how well 3D numerical solutions align with thier
    phase plots

    Arguments:
        b: determines the strength of friction within the system

    Returns:
        a 3D quiver plot of the phase plot alongside a plot of the numeri
        of the equation
    """

    # generate the quiver plot first
    # lowered number of arrows to keep plot readable and
    x = np.linspace(-1, 4, 10)
    y = np.linspace(-1, 4, 10)
    z = np.linspace(-1, 4, 10)

    # turning them into meshgrid objects
    X, Y, Z = np.meshgrid(x, y, z)
    u = thomas_attractor([X,Y,Z], 0, b)

    x_dot = u[0]
    y_dot = u[1]
    z_dot = u[2]

    ax = plt.figure(figsize = [15,15]).add_subplot(projection='3d')

    # generate quiver plot
    ax.quiver(X, Y, Z, x_dot, y_dot, z_dot, length = 0.2, normalize = Tru

    # generating initial conditions
    r0 = [1, 2, 3.5]

    # plotting numerical solution
    t_arr = np.linspace(0,1000,10000)
    r = odeint(thomas_attractor, r0, t_arr, args = (b,))
    x = r[:,0]
    y = r[:,1]
    z = r[:,2]

    ax.plot(x,y,z, color = "red")

    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.set_zlabel("z")
    ax.set_title("3D Phase Plot Compared With Numerical Solutions")

    return
```

For the sake of simplicity, we will only be performing this analysis on 1 value of b at $b = 0.208186$. This value was chosen as it was oddly enough resulted in the plot that was the easiest to read in terms of finding how well they aligned with our numerical solutions.

It was also decided to only include the portion of the plot within the 1st octant to keep the system as simple as possible.

Note: This function also takes a bit to run (unless your computer isn't a potato)

In [12]:
```
b = 0.208186
check_accuracy(b)
```

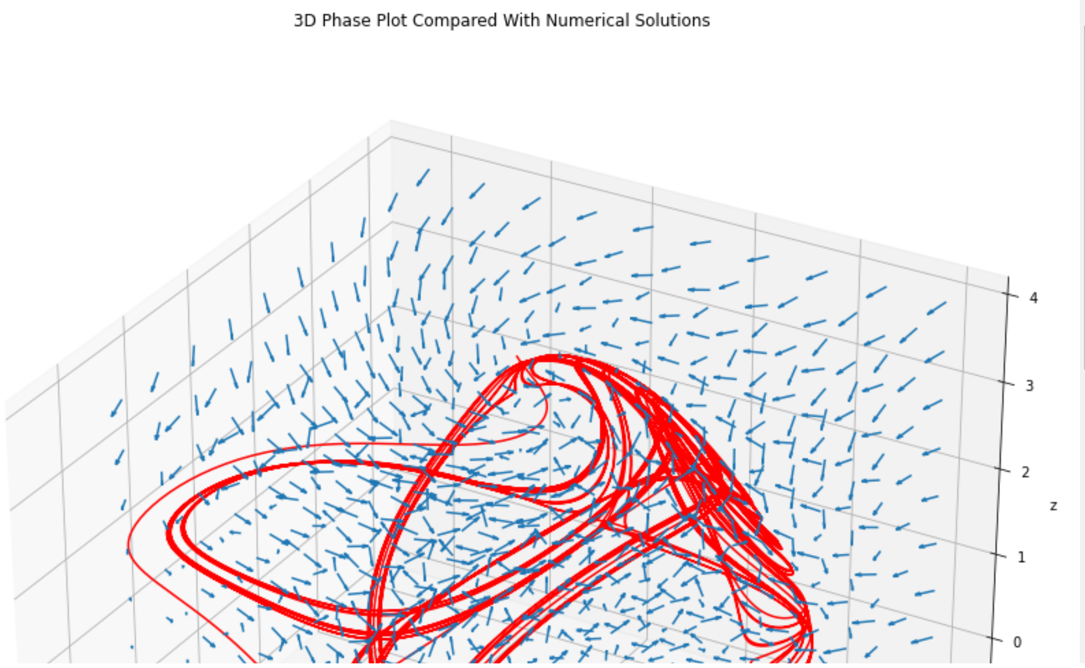

Fig 6. This shows the 3D plot of the Thomas Attractor's numerical solutions as well as it's phase plot in the 1st octant. Due to the inability to rotate the image and zoom in, this plot is quite cumbersome to perform a complete analysis of.

As expected, despite our best efforts, this comparison plot is quite difficult to interpret. However, upon closer inspection (especially near the outer rings of the plot), it is actually possible to see that there are quite a few points in which it is actually somewhat clear that the numerical solutions follow the phase plot quite well.

When attempting to combine this information with the information we gathered from the corresponding 2D streamplots animation (**Fig 3.**), it is nearly impossible to see how the two plots relate to one another, but at the very least it would be possible to understand that there is some kind of a relationship between the position of this *convergence point* that we find in each of the frames of the animation and how the particle moves in the plot.

All of the evidence here seems to indicate that yes, the data is accurate. However, this evidence is somewhat lacking and a much deeper analysis of the accuracy of the results is required before we can definitively say that these numerical results match their respective phase plots.

Limitations

While the overall analysis of this system went quite well, there ended up being quite a few roadblocks that we ran into that somewhat inhibited our ability to analyze this system to its fullest extent. Most of these limitations were software related, having to do with many of the libraries we used simply lacking the correct visualization tools. Matplotlib, while having the ability to create 3D quiver plots, lacked the ability to make it so that the system could be properly analyzed, while Plotly gave us the appropriate tools to inspect our data, but the quiver plot functionality that we required to generate interactive 3D phase plots was very awkward, and didn't allow for the creation of plots that showed any kind of useful information.

Additionally there's also the fact that we had no physical analogues or analytical solutions whatsoever. Had we even one of these, the analysis of the system would have gone much smoother, and we would end up having far more information on just how accurate this system was with respect to these other results.

Conclusion

Based on the analysis perfromed of the Thomas Attractor and the different behaviors that the system exhibits based on the amount of friction that exists within the system, we have succesfully completed a series of different methods of analysis such as generating phase plots, animations and 3D plots of the attractor. As for how accurate they are, due to the nature of the strange attractor (or at least this attractor) being a purely mathematical object, there do not exist any real-world phenomena that can be described by this system. Fortunately, the system does seem to resemble the numerical solutions generated by others, so it is unlikely that there are inaccuracies within the methodology, although a more throrough analysis of this will be necessary to fully confirm this idea.

Citations

Campuzano, J. C. P. (2018, July 1). Dynamic mathematics. Strange Attractors. Retrieved October 16, 2022, from https://www.dynamicmath.xyz/strange-attractors/#:~:text=The%20term%20'Strange%20Attractor'%20is,become%20exponentially%20sepa (https://www.dynamicmath.xyz/strange-attractors/#:~:text=The%20term%20'Strange%20Attractor'%20is,become%20exponentially%20sepa

Musanna, F., Dangwal, D., Kumar, S., & Malik, V. (2020). A chaos-based image encryption algorithm based on multiresolution singular value decomposition and a symmetric attractor. The Imaging Science Journal, 68(1), 24–40. https://doi.org/10.1080/13682199.2020.1732116 (https://doi.org/10.1080/13682199.2020.1732116)

Wikimedia Foundation. (2020, May 16). Thomas' cyclically symmetric attractor. Wikipedia. Retrieved October 16, 2022, from https://en.wikipedia.org/wiki/Thomas%27_cyclically_symmetric_attractor (https://en.wikipedia.org/wiki/Thomas%27_cyclically_symmetric_attractor)

Wikimedia Foundation. (2022, September 15). Lorenz System. Wikipedia. Retrieved October 16, 2022, from https://en.wikipedia.org/wiki/Lorenz_system (https://en.wikipedia.org/wiki/Lorenz_system)

Ávalos-Ruiz, L. F., Zúñiga-Aguilar, C. J., Gómez-Aguilar, J. F., Escobar-Jiménez, R. F., & Romero-Ugalde, H. M. (2018). FPGA implementation and control of chaotic systems involving the variable-order fractional operator with Mittag–Leffler law. Chaos, Solitons & Fractals, 115, 177–189. https://doi.org/10.1016/j.chaos.2018.08.021 (https://doi.org/10.1016/j.chaos.2018.08.021)

In [ ]:

```

```