Chaos and Legendre Polynomials Visualization in MATLAB and Paraview

Instructors

Dr. Jens Lorenz, Dr. Deborah Sulsky

Funding and Support

The Department of Mathematics and Statistics

Written by

Jeffrey R. Gordon

Abstract

This paper is to summarize research done on the visualization of Ordinary Differential Equations, “ODE'(s)”, and Partial Differential Equations, “PDE'(s)”. It is separated into three sections: the first section will discuss chaotic systems such as the Lorenz attractor and how to analyze them, the second section will discuss the Legendre Polynomials with regards to how they are derived and their symmetry with the sphere, and the third section will explain how to use Paraview for plotting these kinds of functions.

Table of Contents

1 Chaos Theory and Chaotic Plots

1.1 The Definition of Chaos

A chaotic system in mathematics can roughly be defined as a system of ODE's that shows extreme sensitivity to initial conditions. A more precise definition is for a system to be chaotic is must show sensitivity to initial conditions, it must be topologically mixing, and its orbits must be dense. These definitions can be cumbersome and do require some explanation.

Several mathematical consequences arise from this reliance on initial data. One of the most notable is how quickly solutions will diverge from one another no matter how small the change is. For small time, the solutions will be nearly identical to one another. As time grows, the trajectories of the chaotic systems will suddenly have no correlation with the other. This raises several questions about the models for systems such as the weather. Weather is an extremely chaotic system and the tools used to measure the initial conditions given to these models are finitely accurate. This means that as time evolves the predictions made about the weather patterns will shortly be inaccurate.

The idea of topological mixing is often omitted from texts on Chaos Theory, although sensitivity to initial conditions alone is far from sufficient to account for chaotic behavior. Having this mixing implies that the system will evolve over time such that every open set of its phase space will eventually intersect with every other open region. To provide a counter example, consider a dynamical system that is produced through doubling of the initial conditions. This system shows extreme sensitivity to initial conditions since points will begin to quickly separate. It is not chaotic though, so this clearly shows that for a system to be in chaos it requires more than sensitivity.

The density of the orbits is also of importance to prove that a system is acting chaotically. The dynamical system must be arbitrarily close in all phase space to some periodic orbit. For those who have studied ODE's this will seem counter-intuitive, because what we generally think of as an orbit is one that repeats itself. This is not the case in a chaotic system. Its orbitals may never come close to anything resembling repeating. Combine this idea with topological mixing and the true nature of chaos is revealed. For any small perturbation in the system this arbitrary distance between the dynamical system's solution and the periodic orbit will change and then something entirely unpredictable will occur. This is why mathematicians, even young ones such as myself, find this incredible.

1.2 The Lorenz Attractor

Perhaps the most famous example of a chaotic system is the Lorenz Attractor. Named for Edward N. Lorenz who defined the equations. These are a set of ODE's that define the chaotic behavior of the Lorenz Oscillator. They are:

dx/dt = sigma*(y-x)

dy/dt = x*(rho-z)-y

dz/dt = x*y-beta*z

where beta, sigma, and rho are parameters which can vary. This system shows several types of behavior depending on the parameters. For some values the ODE is very well behaved and shows no signs of chaos. For values such as:

beta = 8/3

sigma = 10

rho = 28

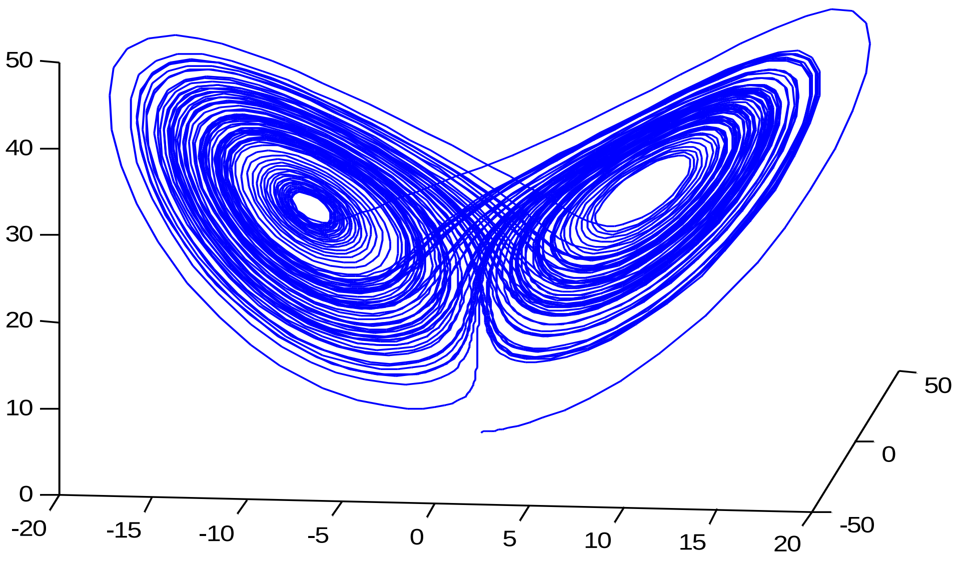
the Lorenz functions become strange attractors. What does this chaotic behavior imply? One way of observing the implications is in the form of what the solutions look like plotted. The following script in MATLAB will solve for any desired value of the parameters and provide a three dimensional plot over a specified time, and below are two examples of these plots.

```
function [T, X, Y, Z] = myLorenzSolver(tSpan, S0, sigma, rho, beta)
[T,S] = ode45(@myLorenz,tSpan,S0,[],sigma,rho,beta);
X = S(:,1);
Y = S(:,2);
Z = S(:,3);
end

function [dS] = myLorenz(t,S,sigma,rho,beta)
%x = S(1);
%y = S(2);
%z = S(3);

x = sigma*(S(2) - S(1));
y = S(1)*(rho-S(3)) - S(2);
z = S(1)*S(2) - beta*S(3);
dS = [x;y;z];
end

>>plot3(X,Y,Z,'b')
```

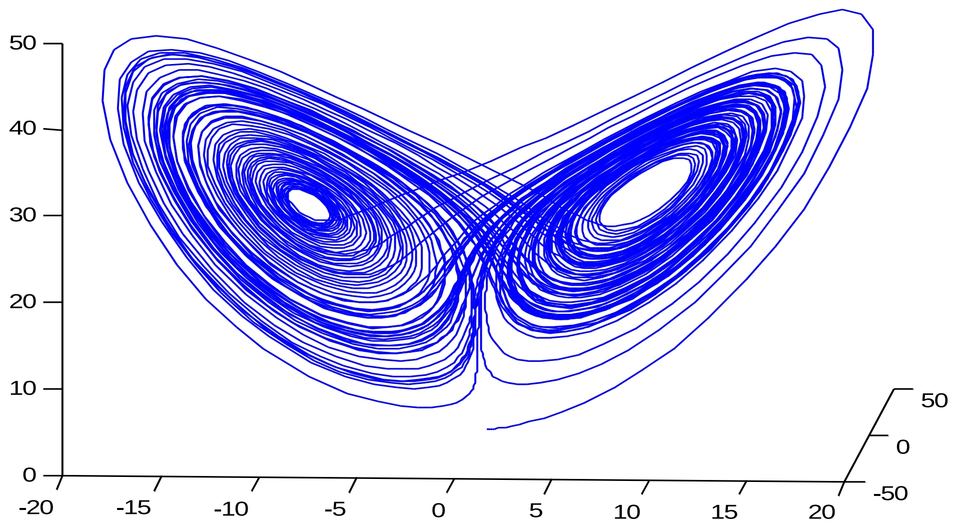


(Lorenz Attractor 1 - MATLAB)

The oscillatory trajectories of these plots illustrates the idea of orbital density, although they do not give light to how the initial conditions will drastically change the path the solution follows. To illustrate this point, the initial conditions used to plot this trajectory were:

$$x = .1$$
$$y = .1$$
$$z = .1$$

these will all be increased to .2, and plotted on the same time interval. The path is now quite different.

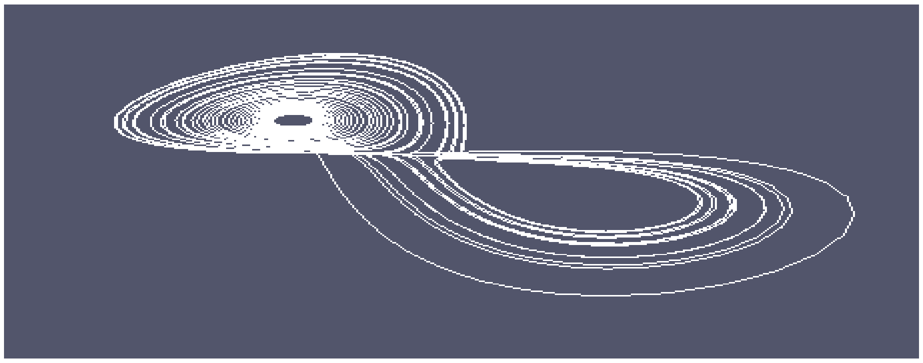


(Lorenz Attractor 2 – MATLAB)

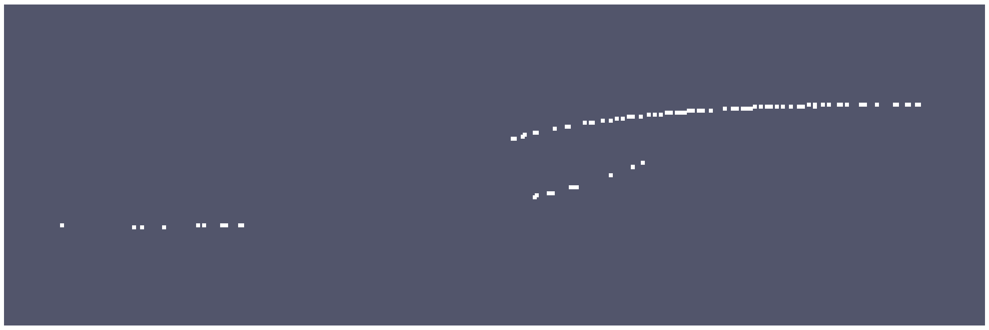
Although MATLAB does an excellent job of showing the overall appearance of the Lorenz Attractor, a more three dimensional view can help gain an appreciation for not only the complexity of this dynamical system but also its beauty. This will be discussed in further detail in section five.

1.3 Poincaré Sections

To further study these attractors the Poincaré section is often used. The section itself is simply a plane cutting into the trajectories of the attractor. When the plane cuts through the attractor it plots the points where they intersect on the plane’s surface. In most cases these maps cannot be defined explicitly so the use of a computer is required to create them. For our purposes, Paraview is the easiest tool to use in generating the section. Taking the Lorenz attractor and plotting it in Paraview results in the following graph.



Using Paraview to take a slice from this plot the Poincaré section can be extracted. The following image is the resulting section.


(Poincaré Section – Paraview)

This section has revealed a few different trends in the attractor for the time period it was plotted over. The density of the upper most orbital is much greater than the lower orbital, which implies that for these initial conditions the oscillator will spend more time in the upper orbital than it will in the lower. The lower orbital also seems to have a wider range of values it occupies than the upper orbital. The lower orbital also has a much wider gap than the upper orbital. This illustrates that quite a bit of information about how a particular dynamical system is behaving can be derived from these sections.

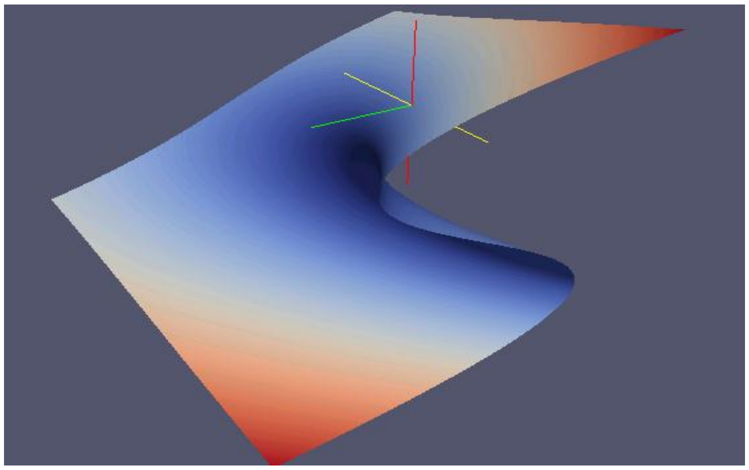
1.4 Bifurcation Surfaces

Those who are familiar with chaos theory are acquainted with the idea of a bifurcation and the logistic map. These parameterized ODE’s can go from having one stable point under certain parameters and then change to have two or more stable points. Most of the systems that a student is presented have one parameter. Things get a little more interesting when there are two or more parameters that can be varied. Instead of having bifurcation lines there are now bifurcation surfaces which can show incredibly interesting behavior, such as in the case of the cusp bifurcation catastrophe.

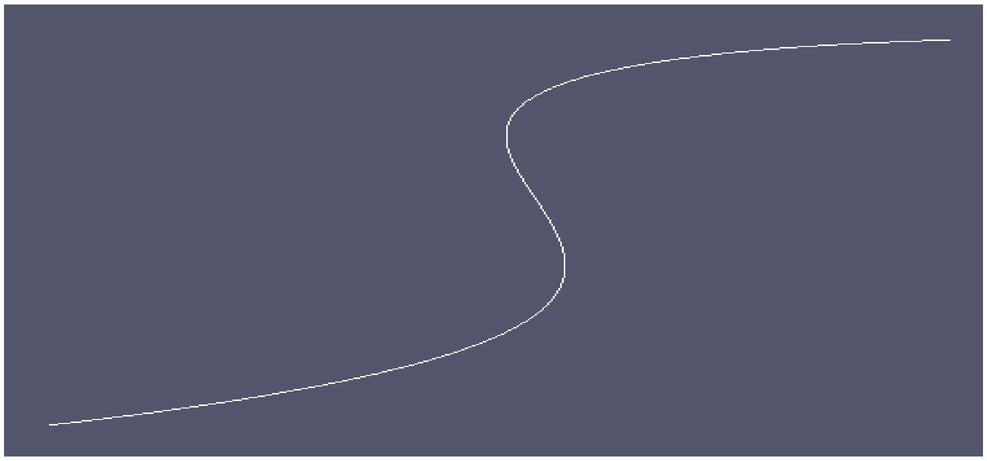
One of the most common examples is the function:

$$f(x) = f(x;a,b) = x^3 + a*x + b = 0$$

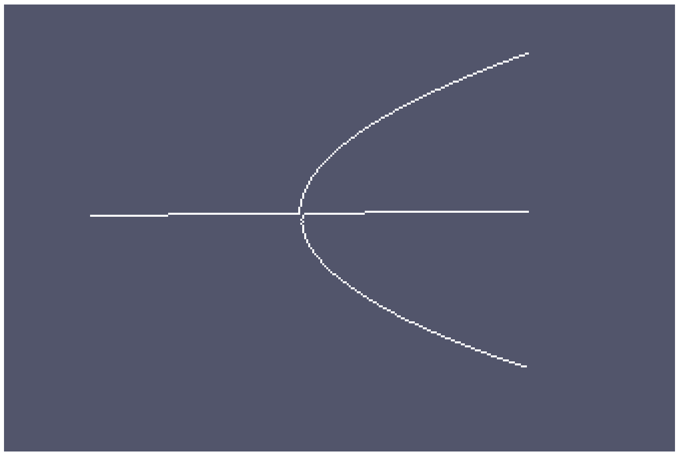
Called the elementary cusp catastrophe this particular example has the following graph.


(Cusp Bifurcation – Paraview)

If we allow for one of the parameters to be held constant the reasons for this odd behavior from the surface can be observed. When the parameter a is held constant and b is allowed to vary results in the following graphs.


(Cusp Bifurcation Cubic – Paraview)

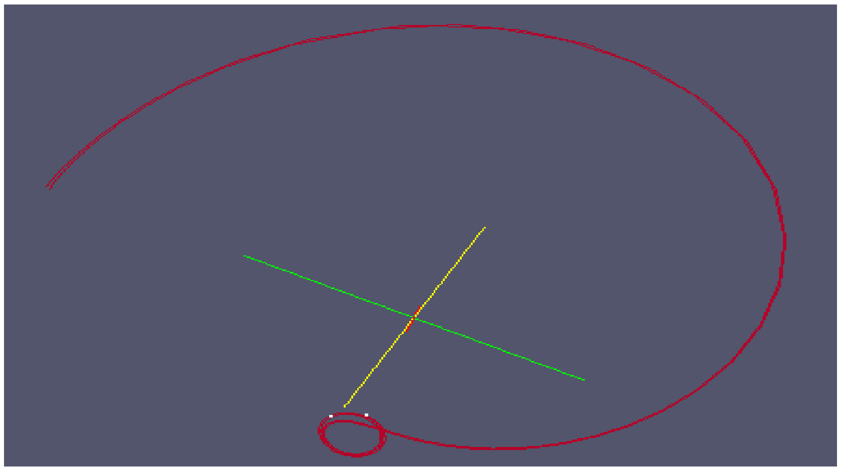
This portion of the surface is quite obviously a cubic, which helps explains the origin on the cusp, although to fully understand the evolution of the bifurcation surface b must now be held constant and a allowed to vary. When this happens the following graph occurs.


(Cusp Bifurcation Pitchfork – Paraview)

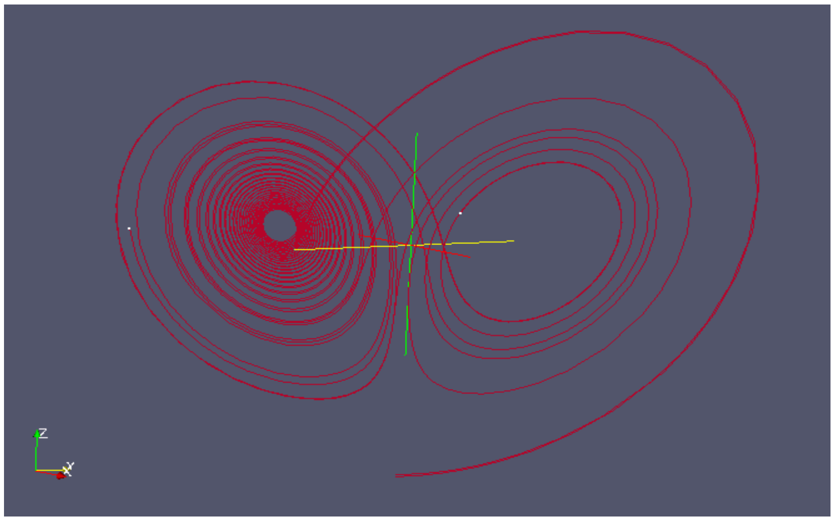
This portion of the surface shows the evolution of one stable point into what resembles a pitchfork bifurcation, except splitting into three possible stable points. Now with this is in mind the surface and the reason it appears as it does makes quite a bit of sense.

1.5 Evolution of the Trajectories

Going back to the Lorenz Attractor, another method of analyzing these systems is to watch the trajectories evolve over time. This will allow for comparison between different initial conditions in a way that Poincaré sections or visual analysis of the plots cannot supply. Watching two different set of initial conditions which are close to one another start out with the same trajectory then suddenly diverge from one another is fascinating experience. Using Paraview, it is possible to watch this happen between two different attractors in real time.


(Trajectory 1 – Paraview)

This plot shows the current difference between two trajectories with similar initial conditions. As demonstrated they are following very similar paths, although as time continues to evolve these attractors will begin their own separate paths.



(Trajectory 2 – Paraview)

At this point the two paths have diverged from one another, and the behavior of either path cannot be predicted from the other.

2 The Legendre Polynomials

2.1 Definitions and Derivation

In mathematics there are some problems which have vast complexity, especially in Partial Differential Equations. The diffusivity equation, the wave equation, Laplace's equation, and Schrodinger's equation are classic examples of PDE's that are difficult to solve analytically. These problems can be further complicated by trying to evaluate them with some inherent symmetry. A prevalent example of this symmetry would be the sphere in the cases of an electron orbiting a Hydrogen atom, vibrations in a ball, and electric potential in two hemispheres. Although there may not be an analytical solution to these problems, as is seen in their 1-dimensional cases one can construct an answer from the Fourier series.

$$\frac{a_0}{2} + \sum_{n=1}^{\infty} [a_n \cos(nx) + b_n \sin(nx)]$$

Some of the properties of the Fourier Series:

Property 1)
The coefficients a_n and b_n of the series are computed from 2*pi periodic functions.

Property 2)
If f is a 2*pi periodic odd function, a_n = 0 for all n

Property 3)
If f is a 2*pi periodic even function, b_n = 0 for all n

The Fourier series is a vector space constructed of sines and cosines which maps a function from L2 functional space to L2 functional space and is automorphic, since the series also maps a function from itself onto to itself. The summation and its coefficients can be thought of as a statement of how much of each sine and cosine wave is present in the function. It can also be shown that the series converges quickly so long as there are no discontinuities in the function. This means that there is no real need to have an analytic solution to compute these functions.
In the case of a spherical coordinate system the same principle can be applied to solve any of these PDE's. The primary difference between the 1-D case and the spherical case is the presence of the Legendre Polynomials in the sum.

The Legendre polynomials are solutions to the Legendre differential equation, which at first glance looks like a rather docile ODE:

$$\frac{d}{dx}\left[(1-x^2)\frac{d}{dx}P_n(x)\right] + n(n+1)P_n(x) = 0.$$

Looks can be deceiving though, because from the solutions to this ODE arise an orthogonal basis for the sphere. The solutions take the form of the recursive function:

$$(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x).$$

2.2 Symbolic Legendre Polynomials

From the solution to the Legendre differential equation come the Legendre polynomials. These polynomials are what form the basis for the sphere and the following are the first ten Legendre polynomials.

$$1$$

$$x$$

$$\tfrac{1}{2}(3x^2 - 1)$$

$$\tfrac{1}{2}(5x^3 - 3x)$$

$$\tfrac{1}{8}(35x^4 - 30x^2 + 3)$$

$$\tfrac{1}{8}(63x^5 - 70x^3 + 15x)$$

$$\tfrac{1}{16}(231x^6 - 315x^4 + 105x^2 - 5)$$

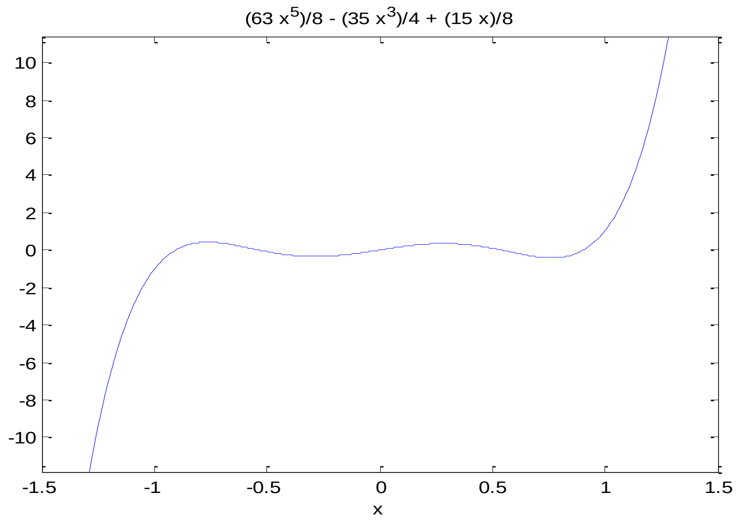
$$\tfrac{1}{16}(429x^7 - 693x^5 + 315x^3 - 35x)$$

$$\tfrac{1}{128}(6435x^8 - 12012x^6 + 6930x^4 - 1260x^2 + 35)$$

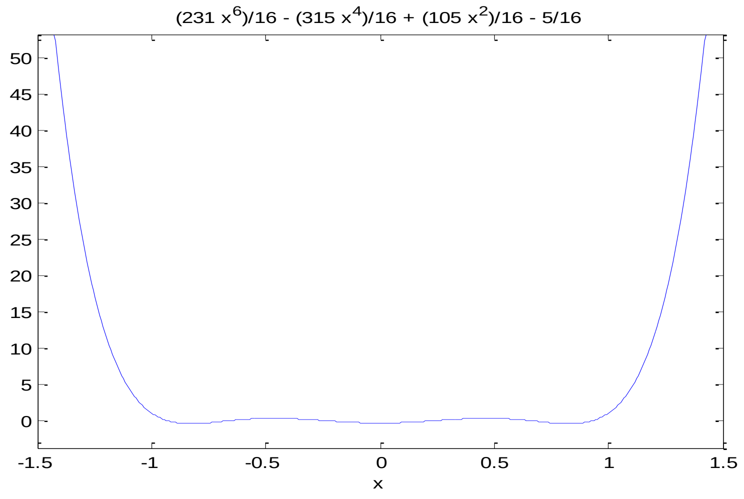
$$\tfrac{1}{128}(12155x^9 - 25740x^7 + 18018x^5 - 4620x^3 + 315x)$$

$$\tfrac{1}{256}(46189x^{10} - 109395x^8 + 90090x^6 - 30030x^4 + 3465x^2 - 63)$$

The polynomials have a unique symmetry on the interval [-1,1] as the following graphs show.



(P_5(x) Legendre - MATLAB)

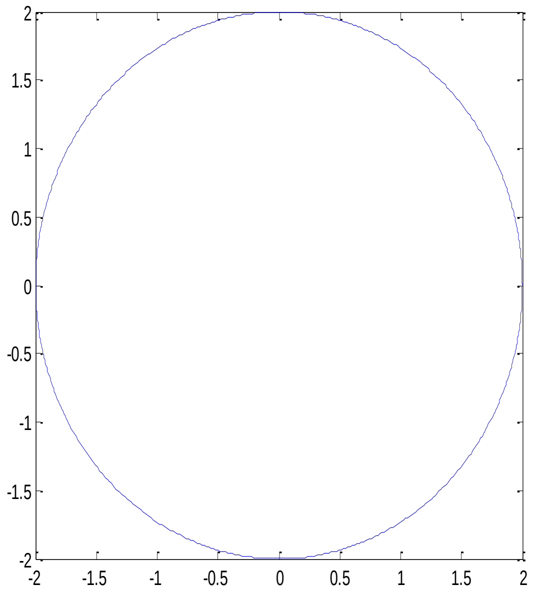


(P_6(x) Legendre - MATLAB)

These symmetries are part of what give the Legendre polynomials their spherical properties. As will be discussed in the further sections, the higher the order of the polynomial the more transformations of the surface of the sphere that will occur. These manipulations are what allow these polynomials to function like the sines and cosines in the Fourier series and construct solutions to functions that exist on the sphere.

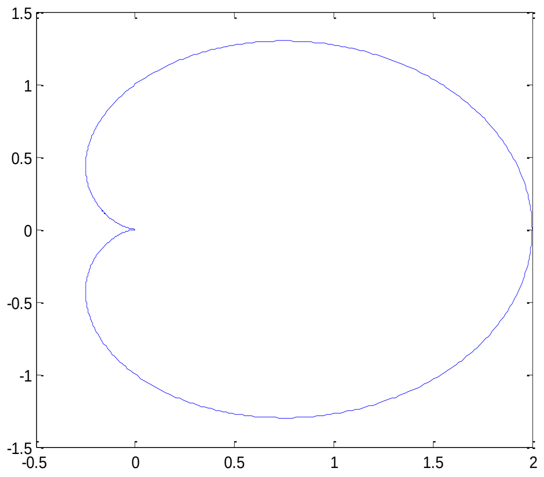
2.3 Legendre Polynomials on the Circle

To demonstrate this symmetry, MATLAB will be used to plot the Legendre polynomials on the circle. There will be various plots using higher order polynomials to show case how the polynomials manipulate the edges of the circle. The first case is for the zeroth order polynomial.

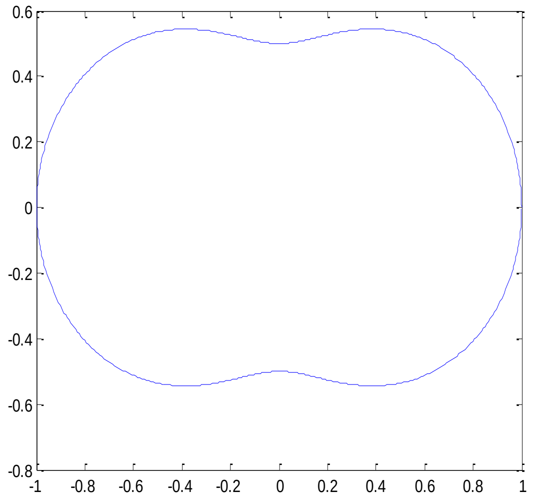


(Legendre Circle Zero Order - MATLAB)

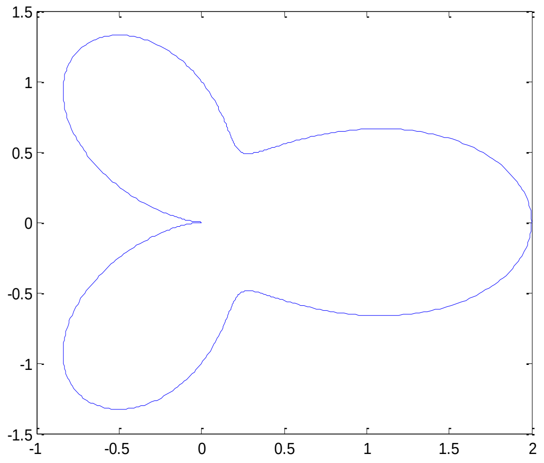
This is obviously a circle, although to observe the more interesting transformations the next plots will be the first, second, and third order Legendre polynomials on the circle.



(Legendre Circle First Order - MATLAB)



(Legendre Circle Second Order - MATLAB)



(Legendre Circle Third Order - MATLAB)

These transformation of the circle allow the Legendre polynomials to function like the sines and cosines in the Fourier sum. Any function that has circular symmetry can have its solution constructed by a similar sum of these circles.

2.4 Legendre Polynomials on the Sphere

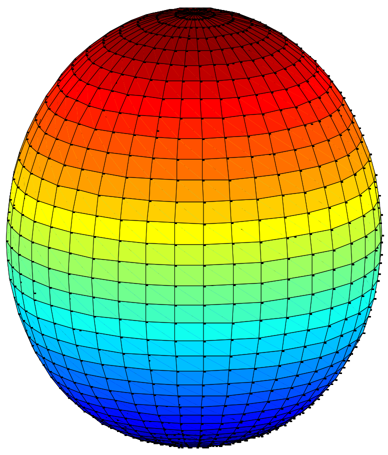
This symmetry can now be expanded from the circle to the sphere in what are called the spherical harmonics. These harmonics have countless applications from video game design, solutions to the wave equation, to art. As has been discussed earlier these harmonics are constructed from a summation similar to the Fourier series taking the following form.

$$f(\theta,\varphi) = \sum_{\ell=0}^{\infty} \sum_{m=-\ell}^{\ell} f_\ell^m \, Y_\ell^m(\theta,\varphi).$$

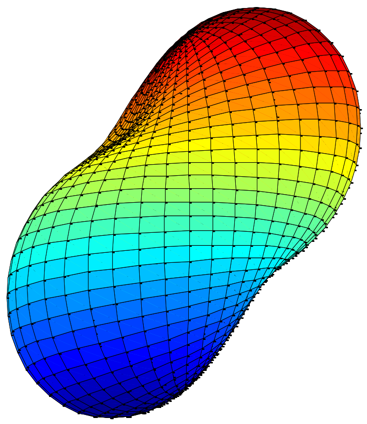
Where,

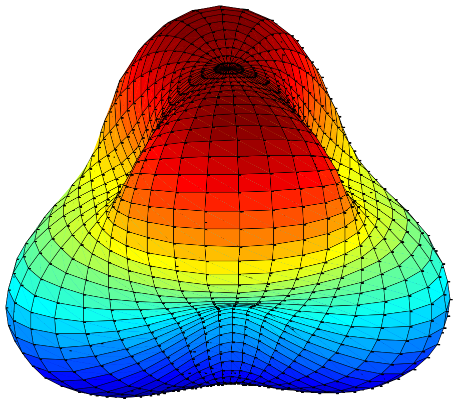
$$Y_\ell^m(\theta,\varphi) = \sqrt{\frac{(2\ell+1)}{4\pi}\frac{(\ell-m)!}{(\ell+m)!}}\, P_\ell^m(\cos\theta)\, e^{im\varphi}$$

Putting the summation aside, the spherical harmonics have the same effect on the sphere that the Legendre polynomials had on the circle. They manipulate the sphere and allow them to sum together constructing solutions to functions with spherical symmetries. Several plots of these functions are below.



(Spherical Harmonic Y^1_0 - MATLAB)


(Spherical Harmonic Y^2_1 - MATLAB)


(Spherical Harmonic Y^3_2 - MATLAB)

Similar to how the coefficients of the Fourier series adjust the sines and cosines to satisfy a function, the Legendre polynomials adjust the sphere to satisfy a spherical function.

3 Paraview

3.1 What is Paraview

Paraview is a visualization software kit that allows scientists, mathematicians, and engineers to demonstrate their system models in a three dimensional environment. It was developed in part by Sandia National Laboratories and is used extensively by research communities around the world. The mathematical explanations that have been given in this paper have meant to prepare the reader to use Paraview effectively for these chaotic functions, surfaces, and differential equations.

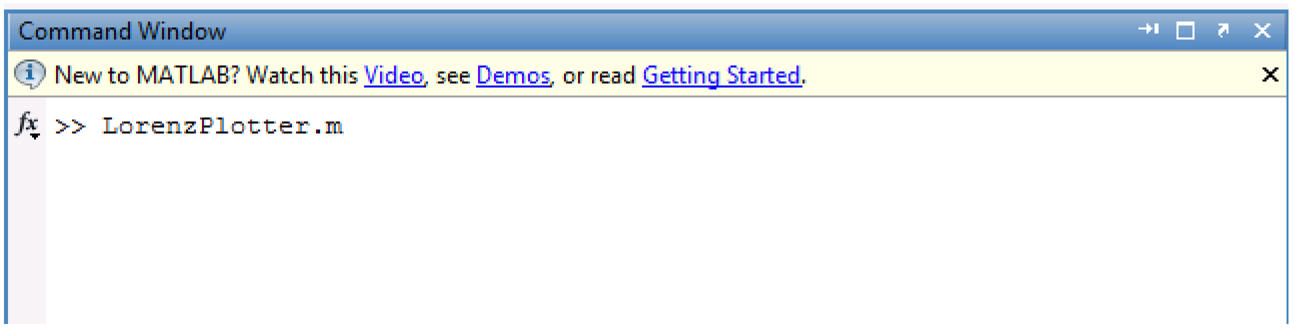
The software is extremely versatile and can be used to map data over a period of time, solve contours, map particle paths, visualize collisions, and a myriad of other applications. Understanding how to use and apply this piece of software may be extremely beneficial to anyone wanting to demonstrate their theories visually. The following sections are meant to be a mixture of an instruction manual and a theoretical guide to how Paraview functions. This paper will discuss a few different file types that the program accepts and how to manipulate that data in a meaningful fashion.

3.2 How Paraview Understands Data

Paraview is capable of accepting and understanding a wide range of data files, which include CSV (Comma Separated Value), VTK polydata, mathematical functions, point sources, contoured equations, structured data sets, and a hundreds of others. This paper will focus on legacy VTK polydata, mathematical functions, contoured equations, and CSV files.

For example, if one wants to plot the time series of a Lorenz Attractor as it evolves over time it is quite efficient to use a series of CSV files to input into Paraview. In Appendix A, the reader will find the scripts LorenzPlotter.m, LorenzPlotter.fig, and MyLorenzSolver.m. These scripts will allow the reader to build a set of CSV files ready for Paraview to read.

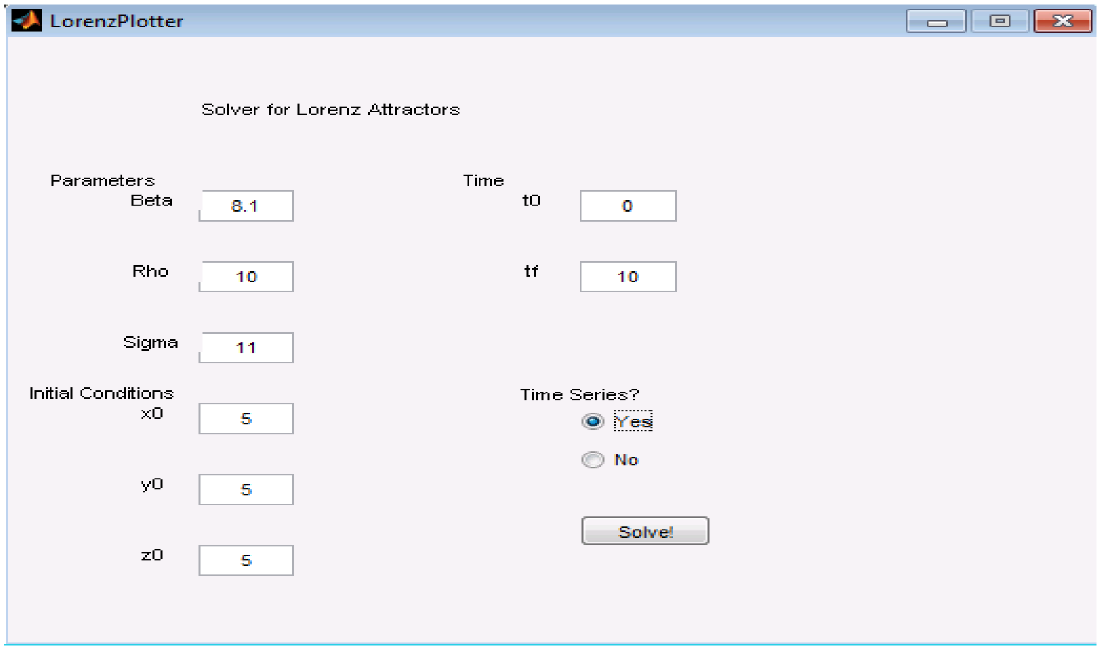
The first step is to open MATLAB and input the following,


Command Window
New to MATLAB? Watch this Video, see Demos, or read Getting Started.

```
>> LorenzPlotter.m
```

(MATLAB Command Window)

After running the program an input window will appear asking for the various values of rho, beta, and sigma. It will also ask for several initial conditions for each variable and the time period that is to be plotted over.



Beta	8.1	t0	0
Rho	10	tf	10
Sigma	11		
x0	5		
y0	5		
z0	5		

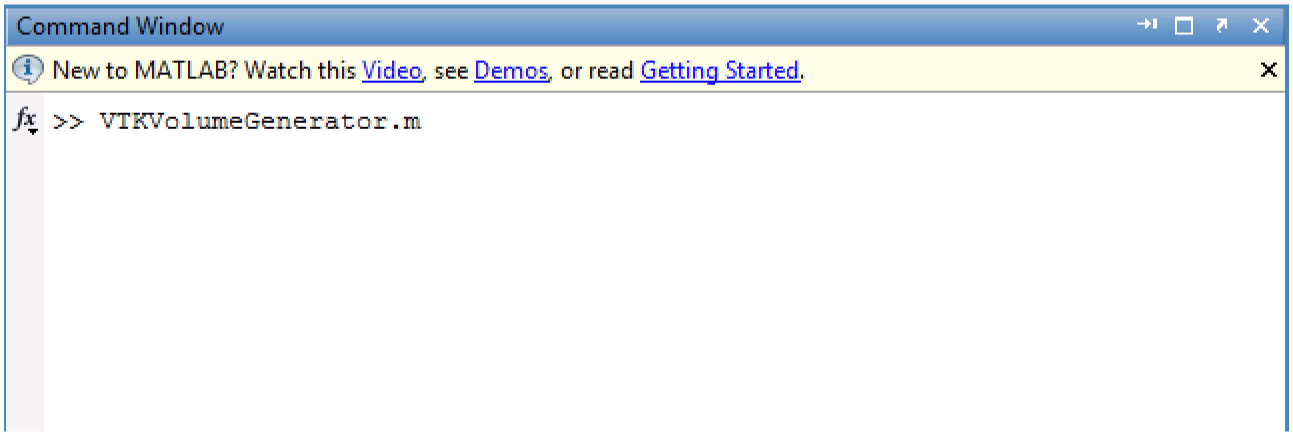
Time Series? [✓ Yes] No

(LorenzPlotter GUI - MATLAB)

After accepting the values and running the script, the code will either generate one large CVS file containing all of the values of the Lorenz attractor or create a series of files each containing a single point of the attractor. The main difference is how Paraview will interpret the data. If the reader wishes to create a series of files then the user will pick, yes, under the time series toggle. If the user wants to make a large CSV file of each point then the reader will pick, no, under the time series toggle.

Paraview can read these CSV files in two different ways, as either a static set of points or as a time series of points. If a large CSV table is uploaded into Paraview it has no way of understanding when these points came into existence, so it will plot them simultaneously. If a series of CSV files are uploaded, then Paraview can break them down into a time series which will allow for several different manipulations of the data. Namely in the Lorenz attractor example, the attractor's trajectory as a function of time can be plotted in real time resulting in the graphs seen in section 1.5.

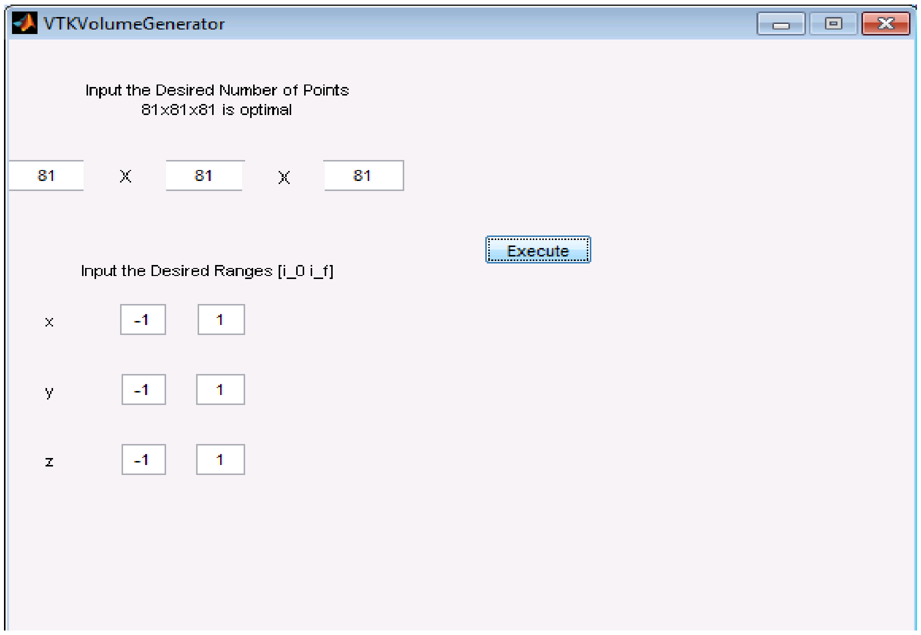
The next major file type accepted in Paraview are VTK polydata files. Once again, in Appendix A the reader will find a set of files titled vect2str.m, VTKVolumeGenerator.m, and VTKVolumeGenerator.fig. The files are capable of building a VTK cube, which the reader will be able to perform a variety of operations on. The way Paraview understands these structures is as a volume of interconnected points that can be evaluated by functions and then contoured to find solutions against. To create one of these files using the provided scripts run the following in MATLAB.


Command Window
New to MATLAB? Watch this Video, see Demos, or read Getting Started.

```
>> VTKVolumeGenerator.m
```

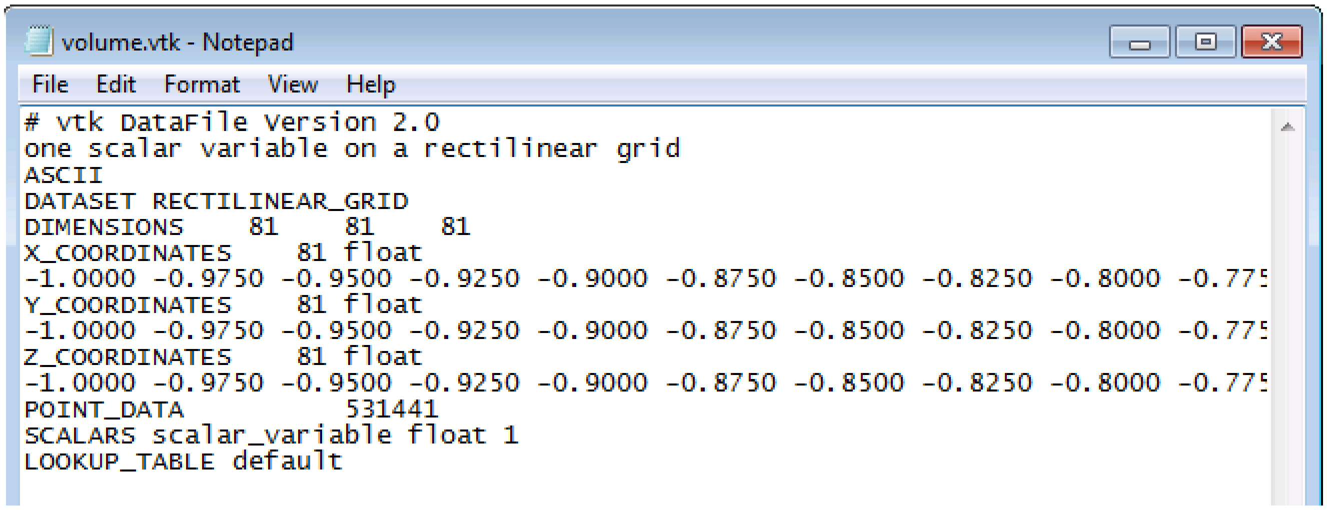
(VTKVolumeGenerator Command Window)

This will once again bring up a form allowing the user to select from a variety of values. In this case, the desired input values are the overall dimensions of the cube which will determine how many data points exist inside the polydata and the range for it to plot in all three dimensions.


VTKVolumeGenerator
Input the Desired Number of Points
81x81x81 is optimal
81
X
81
X
81
Execute
Input the Desired Ranges [i_0 i_f]
x
-1
1
y
-1
1
z
-1
1

(VTKVolumeGenerator Window)

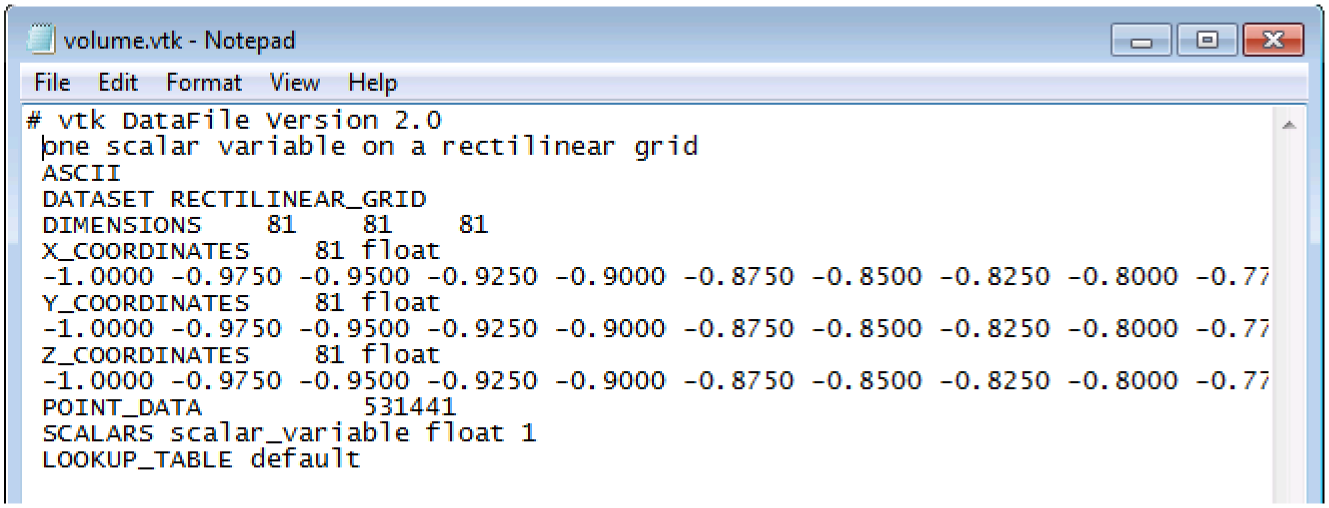
After running the script a legacy VTK polydata file will be automatically created in the same folder as the script. This file can be uploaded directly into Paraview. If there does happen to be an error when uploading it into Paraview, open the file with a text editor and ensure that there are no spaces at the beginning of each line of the code. This will cause Paraview to be unable to read the data. When opened with a text editor the file should look as follows.



```
# vtk DataFile Version 2.0
one scalar variable on a rectilinear grid
ASCII
DATASET RECTILINEAR_GRID
DIMENSIONS     81     81     81
X_COORDINATES     81 float
-1.0000 -0.9750 -0.9500 -0.9250 -0.9000 -0.8750 -0.8500 -0.8250 -0.8000 -0.775
Y_COORDINATES     81 float
-1.0000 -0.9750 -0.9500 -0.9250 -0.9000 -0.8750 -0.8500 -0.8250 -0.8000 -0.775
Z_COORDINATES     81 float
-1.0000 -0.9750 -0.9500 -0.9250 -0.9000 -0.8750 -0.8500 -0.8250 -0.8000 -0.775
POINT_DATA          531441
SCALARS scalar_variable float 1
LOOKUP_TABLE default
```

(VTK Polydata Example)

If there are spaces at the beginning of the data, then Paraview will try to read those spaces as commands. It won't be able to realize that the data is being presented as an ASCII file constructing a rectilinear grid with an associated set of x,y, and z data. So, if the file is opened and looks as follows



```
# vtk DataFile Version 2.0
 one scalar variable on a rectilinear grid
 ASCII
 DATASET RECTILINEAR_GRID
 DIMENSIONS    81    81    81
 X_COORDINATES    81 float
 -1.0000 -0.9750 -0.9500 -0.9250 -0.9000 -0.8750 -0.8500 -0.8250 -0.8000 -0.77
 Y_COORDINATES    81 float
 -1.0000 -0.9750 -0.9500 -0.9250 -0.9000 -0.8750 -0.8500 -0.8250 -0.8000 -0.77
 Z_COORDINATES    81 float
 -1.0000 -0.9750 -0.9500 -0.9250 -0.9000 -0.8750 -0.8500 -0.8250 -0.8000 -0.77
 POINT_DATA          531441
 SCALARS scalar_variable float 1
 LOOKUP_TABLE default
```

(VTK Polydata Faulty Example)

Then the user needs to remove the spaces from in front of each line and the file will load properly. Unfortunately, this issue seems to exist intermittently depending on the machine that the VTKVolumeGenerator is running on. This solution has been tested on dozens of machines that have the error and it has always resolved the issue.

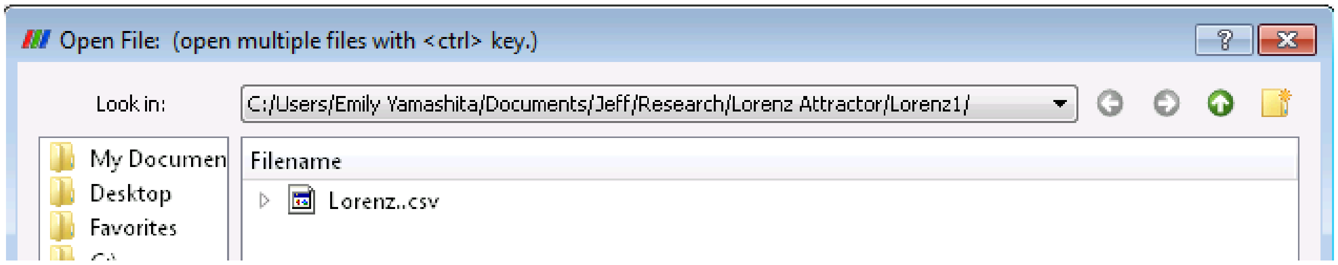
Once the file is successfully loaded into Paraview, the user will be able to apply several mathematical operations to the volume. Namely, the reader can input a function into Paraview to evaluate the volume against. Then that function can be used to make a contour map of the function. This creates images similar to the one seen in the Bifurcation surface seen earlier in the paper.

This concludes the major file types and how Paraview will interpret them. The next step is to demonstrate how to actually make Paraview run them necessary steps to effectively plot these various kinds of data.

3.3 How to use Paraview

To begin using Paraview, the user must go to http://www.paraview.org/ and download the program. It is free to use and open source, so anyone may download and begin using it. Once the program is installed, the user should open the program and prepare some information to upload.

The first example provided is a CSV table. In this case, the file that will upload is a time series CSV table. The table will represent the Lorenz Attractor as it evolves over time. The first step is to open the CSV time series.



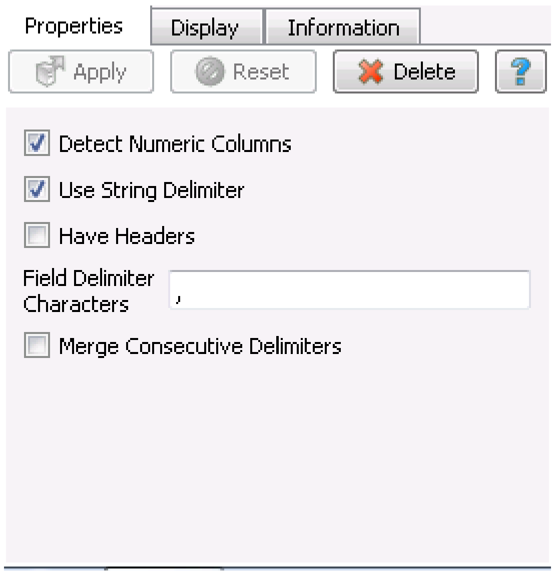
Look in: C:/Users/Emily Yamashita/Documents/Jeff/Research/Lorenz Attractor/Lorenz1/

Lorenz..csv

(Uploading a CSV Time Series)

The reader may notice that while the folder containing the CSV files may have thousands of separate CSV files, but Paraview has recognized them as a time series can compressed them into a set of files. This is normal and nothing to worry about.

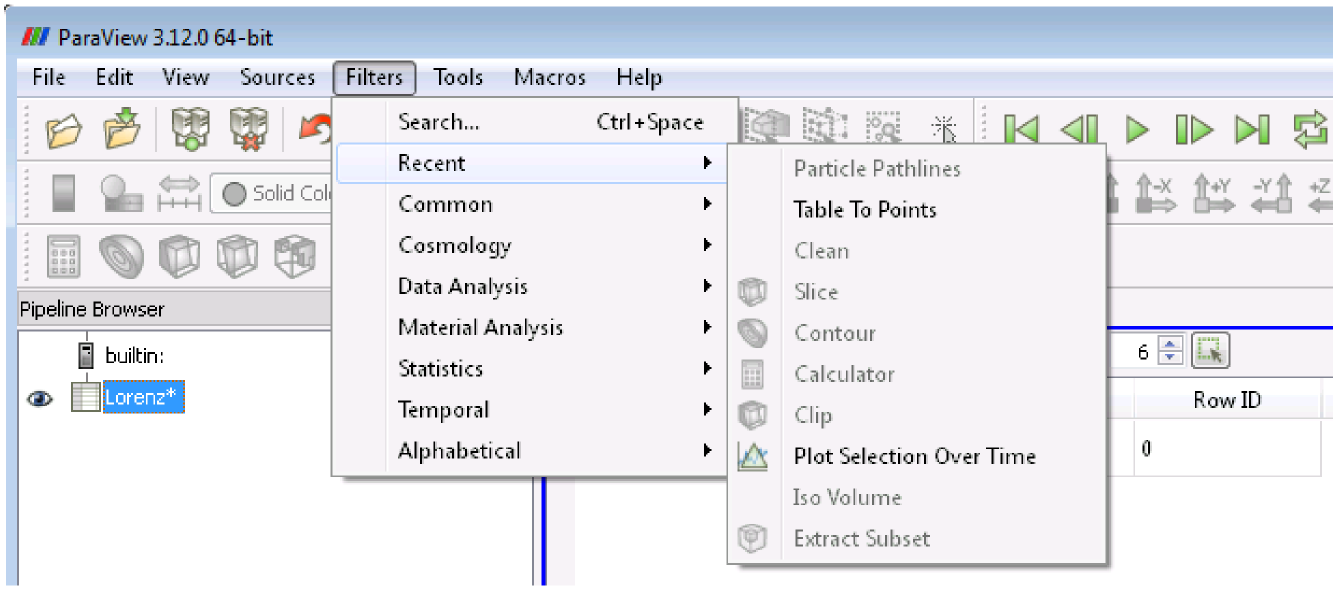
Once the files are uploaded, the user must apply the data. Before doing that, please ensure that the check box for, "Have Headers" is unchecked.



[✓] Detect Numeric Columns
[✓] Use String Delimiter
[ ] Have Headers
Field Delimiter Characters: ,
[ ] Merge Consecutive Delimiters

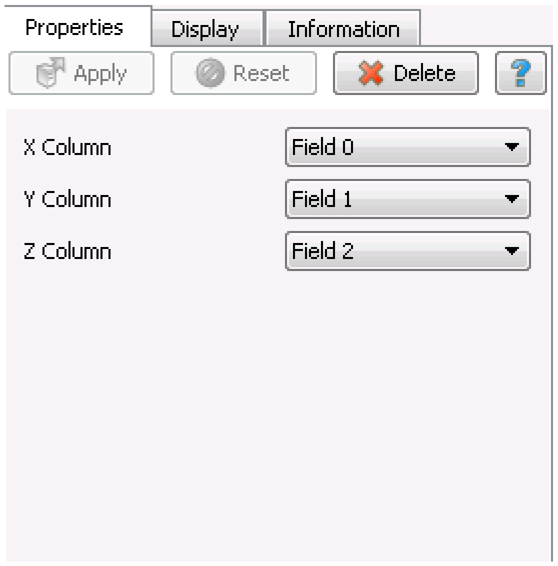
(Object Inspector)

After the time series is applied then the user must run what is called a "Filter" on the data. This will allow Paraview to understand more about the type of data is being upload. The first filter that is to be applied is, "Table to Points."



(Paraview Filter – Table to Points)

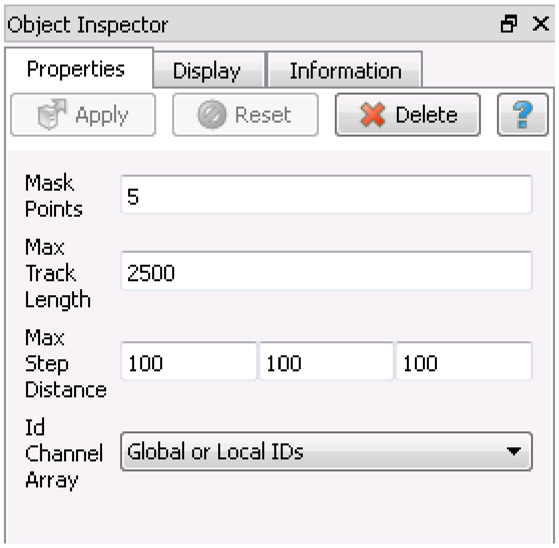
Once the filter is selected it will open a new window in the object inspector. This will allow the user to configure how the data will be interpreted. In this case, the X-Column, Y-Column, and Z-Column should be set to Field 0, Field 1, and Field 2 respectively. This just simply tells Paraview which column contains the respective x, y, and z data to plot.



X Column: Field 0
Y Column: Field 1
Z Column: Field 2

(Object Inspector – Table to Points)

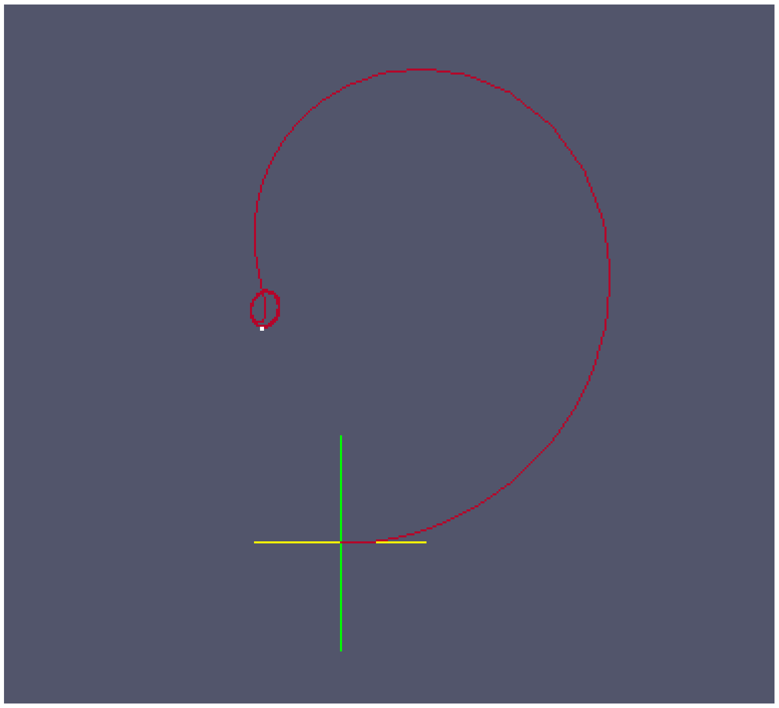
After applying this filter, the next filer to apply is Particle Pathlines, which will allow Paraview to map the trajectory of the attractor. Once the filter is loaded, it will also open a new window in the object inspector. The following settings are ideal for running this program.



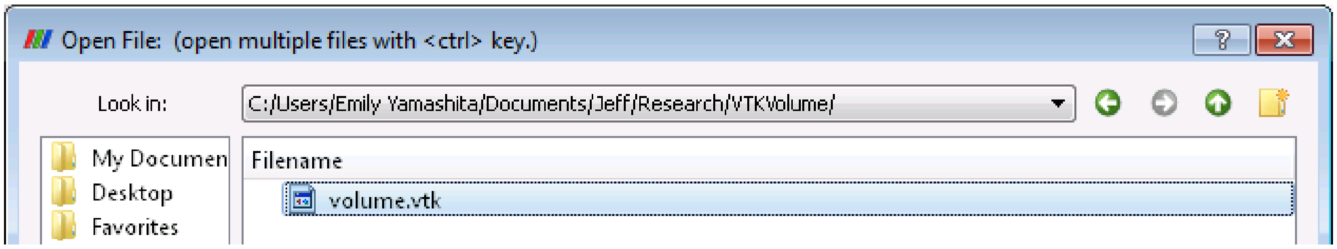
Mask Points: 5
Max Track Length: 2500
Max Step Distance: 100 100 100
Id Channel Array: Global or Local IDs

(Object Inspector – Particle Pathlines)

After applying this filter, the program is ready to run. Press the 3D view button in the display window, then press the start button on the tool bar. You should get a plot that looks similar to this.


(Example Plot – Lorenz Attractor)

The next thing to learn how to load into Paraview is the VTK polydata. After using the MATLAB script to generate the polydata file upload it into Paraview.



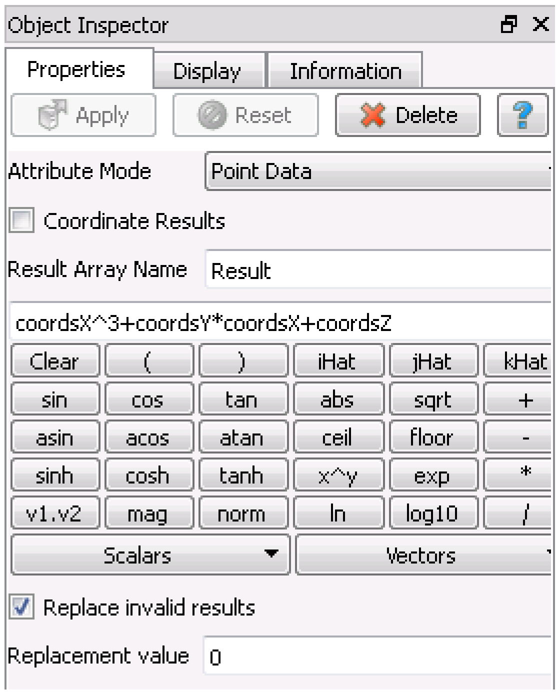
Look in: C:/Users/Emily Yamashita/Documents/Jeff/Research/VTKVolume/

Filename
volume.vtk

(Upload Example – VTK polydata)

This time, the object inspector only requires the user to apply the changes from the VTK polydata.

In this example, there will be a function applied to the polydata and then a contour applied to it. This will create a surface to observe. The first step is to apply the "calculator" filter to the polydata. The object inspector will now ask for the user to plug in an equation for it to evaluate. Please note, that the x, y, and z variables must be called coordsX, coordsY, and coordsZ respectively. The example will be the cusp bifurcation catastrophe.



Attribute Mode: Point Data

☐ Coordinate Results

Result Array Name: Result

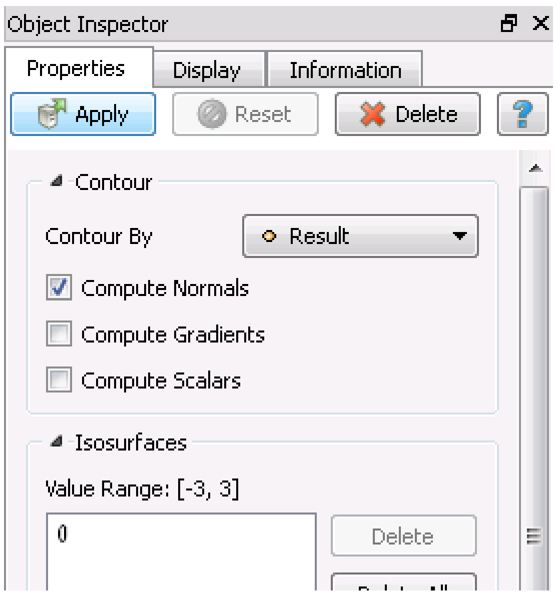
coordsX^3+coordsY*coordsX+coordsZ

☑ Replace invalid results

Replacement value: 0

(Calculator Example)

After applying the changes from the calculator the next filter to load is the "contour" filter. Once the filter is loaded the object inspector will ask for what values you wish to find, in this case we will ask the filter to find all the zeroes in the range.



Contour By: Result

☑ Compute Normals

☐ Compute Gradients

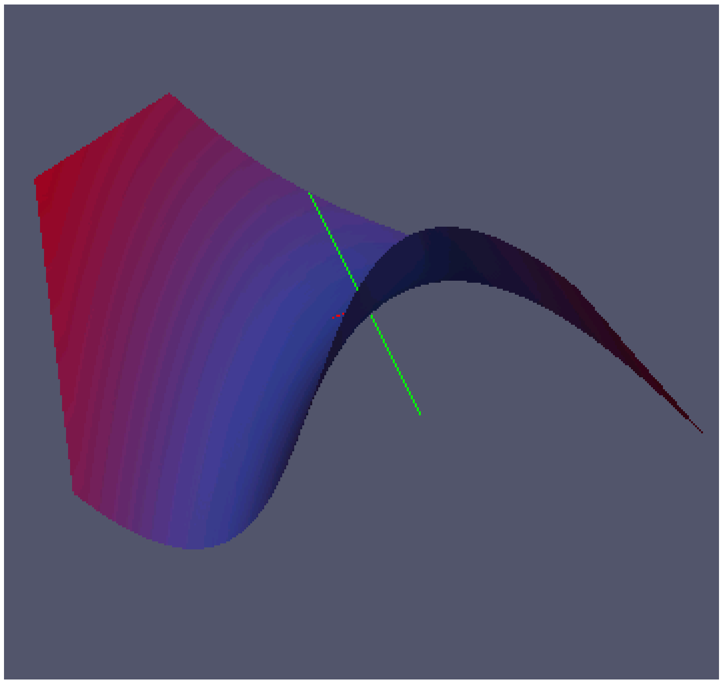
☐ Compute Scalars

Value Range: [-3, 3]

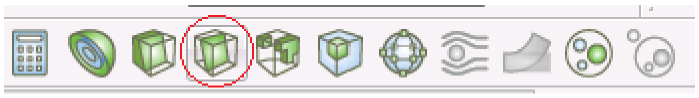
0

(Contour Example)

After inputting the value to contour, check the box that says "Compute Gradients." This will make the plot more colorful and easier to see. After applying the contour you should get a graph that looks like this,


(Contour Example – Cusp)

At this point, the last major tool to use is the slice tool which will allow the reader to make Poincaré sections and take slices of these surfaces. To use the slice tool simply click this button on the tool bar.


(Tool bar – Slice)

Pressing this will create a plane that you can move around with the mouse. Once the plane is in the desired position, simply apply it and Paraview will display the section taken.

Everything past here in using Paraview is just modifications of what has been presented in this paper. As the reader has seen, this program is extremely versatile and if a little practice can make all the difference when trying to effectively present information.

4 Appendixes

A.1 MATLAB codes

The following are the various scripts that have been used with the production of this paper. They are free to use to anyone who wishes to apply them.

<u>LorenzPlotter</u>

```
function varargout = LorenzPlotter(varargin)
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @LorenzPlotter_OpeningFcn, ...
                   'gui_OutputFcn',  @LorenzPlotter_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end

% --- Executes just before LorenzPlotter is made visible.
function LorenzPlotter_OpeningFcn(hObject, eventdata, handles, varargin)
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% --- Outputs from this function are returned to the command line.
function varargout = LorenzPlotter_OutputFcn(hObject, eventdata, handles)

varargout{1} = handles.output;



function Beta_Callback(hObject, eventdata, handles)

input = str2num(get(hObject,'String'));

% --- Executes during object creation, after setting all properties.
function Beta_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function Rho_Callback(hObject, eventdata, handles)

input = str2num(get(hObject,'String'));

% --- Executes during object creation, after setting all properties.
function Rho_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function Sigma_Callback(hObject, eventdata, handles)
```

```
input = str2num(get(hObject,'String'));

% --- Executes during object creation, after setting all properties.
function Sigma_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function x0_Callback(hObject, eventdata, handles)

input = str2num(get(hObject,'String'));

% --- Executes during object creation, after setting all properties.
function x0_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function y0_Callback(hObject, eventdata, handles)

input = str2num(get(hObject,'String'));

% --- Executes during object creation, after setting all properties.
function y0_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function z0_Callback(hObject, eventdata, handles)

input = str2num(get(hObject,'String'));

% --- Executes during object creation, after setting all properties.
function z0_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function t0_Callback(hObject, eventdata, handles)
```

```
input = str2num(get(hObject,'String'));

% --- Executes during object creation, after setting all properties.
function t0_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function tf_Callback(hObject, eventdata, handles)

input = str2num(get(hObject,'String'));

% --- Executes during object creation, after setting all properties.
function tf_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in TimeSeries1.
function TimeSeries1_Callback(hObject, eventdata, handles)

input = str2num(get(hObject,'Value'));

% --- Executes on button press in TimeSeries2.
function TimeSeries2_Callback(hObject, eventdata, handles)

input = str2num(get(hObject,'Value'));


% --- Executes on button press in pushbutton2.
function pushbutton2_Callback(hObject, eventdata, handles)

% --- Initializing variables for the ODE solver --- %

t0    = get(handles.t0,'String');
tf    = get(handles.tf,'String');
x0    = get(handles.x0,'String');
y0    = get(handles.y0,'String');
z0    = get(handles.z0,'String');
Sigma = get(handles.Sigma,'String');
Rho   = get(handles.Rho,'String');
Beta  = get(handles.Beta,'String');
yes   = get(handles.TimeSeries1,'Value');
no    = get(handles.TimeSeries2,'Value');

tSpan = [str2num(t0) str2num(tf)];
S0    = [str2num(x0) str2num(y0) str2num(z0)];
sigma = [str2num(Sigma)];
```

```
rho    = [str2num(Rho)];
beta   = [str2num(Beta)];
Yes    = [yes];
No     = [no];

% --- Initialzing myLorenz Solver --- %

[T, X, Y, Z] = myLorenzSolver(tSpan, S0, sigma, rho, beta);

% --- Generating the CSV file(s) --- %

CSVConstruct = [X Y Z];

if Yes == 0 & No == 0
    sprintf('Please declare if this a Time Series or not\n')
end

if Yes == 1 & No == 1
    sprintf('Please declare if this a Time Series or not\n')
end

if Yes == 0 & No == 1
    csvwrite('lorenz_no_time.csv',CSVConstruct);
end

if Yes == 1 & No == 0
    [m,n]=size(CSVConstruct);
    for i = 1:m
        j = CSVConstruct(i,:);
        csvwrite(sprintf('Lorenz%.0f.csv',i),j);
    end
end
```

MyLorenzSolver:

```
function [T, X, Y, Z] = myLorenzSolver(tSpan, S0, sigma, rho, beta)
[T,S] = ode45(@myLorenz,tSpan,S0,[],sigma,rho,beta);
X = S(:,1);
Y = S(:,2);
Z = S(:,3);
end

function [dS] = myLorenz(t,S,sigma,rho,beta)
%x = S(1);
%y = S(2);
%z = S(3);

x = sigma*(S(2) - S(1));
y = S(1)*(rho-S(3)) - S(2);
z = S(1)*S(2) - beta*S(3);
dS = [x;y;z];
end
```

CSV Writer:

```
function s = vect2str(v,varargin)
```

```
ip = inputParser;
ip.FunctionName = mfilename;
ip.CaseSensitive = false;
ip.addRequired('v', @isvector);
ip.addParamValue('formatString', '%f', @ischar);
ip.addParamValue('openingDelimiter', '(', @ischar);
ip.addParamValue('closingDelimiter', ')', @ischar);
ip.addParamValue('separator', ', ', @ischar);
ip.parse(v, varargin{:});
s = ip.Results.openingDelimiter;

for k = 1 : numel(v)
    s = [s sprintf([ip.Results.formatString],v(k))];
    if k<numel(v)
        s = [s ip.Results.separator];
    end
end
s = [s ip.Results.closingDelimiter];

end
```

VTK Volume Generator:

```
function varargout = VTKVolumeGenerator(varargin)

gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @VTKVolumeGenerator_OpeningFcn, ...
                   'gui_OutputFcn',  @VTKVolumeGenerator_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT


% --- Executes just before VTKVolumeGenerator is made visible.
function VTKVolumeGenerator_OpeningFcn(hObject, eventdata, handles, varargin)

handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% --- Outputs from this function are returned to the command line.
function varargout = VTKVolumeGenerator_OutputFcn(hObject, eventdata, handles)

varargout{1} = handles.output;
```

```
function x_points_Callback(hObject, eventdata, handles)

input = str2num(get(hObject,'String'));


% --- Executes during object creation, after setting all properties.
function x_points_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end




function y_points_Callback(hObject, eventdata, handles)

input = str2num(get(hObject,'String'));


% --- Executes during object creation, after setting all properties.
function y_points_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end




function z_points_Callback(hObject, eventdata, handles)

input = str2num(get(hObject,'String'));


% --- Executes during object creation, after setting all properties.
function z_points_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end




function x_range1_Callback(hObject, eventdata, handles)

input = str2num(get(hObject,'String'));

% --- Executes during object creation, after setting all properties.
function x_range1_CreateFcn(hObject, eventdata, handles)
```

```
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function y_range1_Callback(hObject, eventdata, handles)

input = str2num(get(hObject,'String'));


% --- Executes during object creation, after setting all properties.
function y_range1_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function z_range1_Callback(hObject, eventdata, handles)

input = str2num(get(hObject,'String'));


% --- Executes during object creation, after setting all properties.
function z_range1_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function x_range2_Callback(hObject, eventdata, handles)

input = str2num(get(hObject,'String'));

% --- Executes during object creation, after setting all properties.
function x_range2_CreateFcn(hObject, ~, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function y_range2_Callback(hObject, eventdata, handles)

input = str2num(get(hObject,'String'));
```

```
% --- Executes during object creation, after setting all properties.
function y_range2_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function z_range2_Callback(hObject, eventdata, handles)

input = str2num(get(hObject,'String'));

% --- Executes during object creation, after setting all properties.
function z_range2_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% --- Executes on button press in pushbutton1.
function pushbutton1_Callback(hObject, eventdata, handles)

% --- Finding the number of points per dimension. --- %

l = get(handles.x_points,'String');
h = get(handles.y_points,'String');
w = get(handles.z_points,'String');

v = str2num(l)*str2num(h)*str2num(w);
d = [str2num(l) str2num(h) str2num(w)]

dx = str2num(l)
dy = str2num(h)
dz = str2num(w)

% --- Finding the range values per dimension --- %

x0 = get(handles.x_range1,'String');
xf = get(handles.x_range2,'String');

y0 = get(handles.y_range1,'String');
yf = get(handles.y_range2,'String');

z0 = get(handles.z_range1,'String');
zf = get(handles.z_range2,'String');

% --- Calculating the points in each dimension --- %

l2 = linspace(str2num(x0),str2num(xf),str2num(l));
h2 = linspace(str2num(y0),str2num(yf),str2num(h));
w2 = linspace(str2num(z0),str2num(zf),str2num(w));
```

```
% --- Constructing the VTK Polydata --- %

dimension = vect2str(d, 'formatstring', '%5.f', 'openingDelimiter', '', ...
    'closingDelimiter', '', 'separator', ' ' );
dimenx= vect2str(dx, 'formatstring', '%5.f', 'openingDelimiter', '', ...
    'closingDelimiter', '', 'separator', ' ' );
dimeny= vect2str(dy, 'formatstring', '%5.f', 'openingDelimiter', '', ...
    'closingDelimiter', '', 'separator', ' ' );
dimenz= vect2str(dz, 'formatstring', '%5.f', 'openingDelimiter', '', ...
    'closingDelimiter', '', 'separator', ' ' );
volume = vect2str(v, 'formatstring', '%15.f', 'openingDelimiter', '', ...
    'closingDelimiter', '', 'separator', ' ' );
length = vect2str(l2, 'formatstring', '%5.4f', 'openingDelimiter', '', ...
    'closingDelimiter', '', 'separator', ' ' );
height = vect2str(h2, 'formatstring', '%5.4f', 'openingDelimiter', '', ...
    'closingDelimiter', '', 'separator', ' ' );
width  = vect2str(w2, 'formatstring', '%5.4f', 'openingDelimiter', '', ...
    'closingDelimiter', '', 'separator', ' ' );

% --- Outputting the VTK Polydata to volume.vtk --- %

fid = fopen('volume.vtk','wt');
fprintf(fid,'# vtk DataFile Version 2.0\n one scalar variable on a rectilinear
grid\n ASCII\n DATASET RECTILINEAR_GRID\n DIMENSIONS %s\n X_COORDINATES %s float\n
%s\n Y_COORDINATES %s float\n %s\n Z_COORDINATES %s float\n %s\n POINT_DATA %s\n
SCALARS scalar_variable float 1\n LOOKUP_TABLE default',dimension, dimenx, length,
dimeny, height, dimenz, width, volume);
fclose(fid);
```

SphericalHarmonics

```
function [ output_args ] = SphericalHarmonics( n,m )

% Define constants.
degree = n;
order = m;

% Create the grid
delta = pi/40;
theta = 0 : delta : pi; % altitude
phi = 0 : 2*delta : 2*pi; % azimuth
[phi,theta] = meshgrid(phi,theta);

% Calculate the harmonic
Ymn = legendre(degree,cos(theta(:,1)));
Ymn = Ymn(order+1,:)';
yy = Ymn;
for kk = 2: size(theta,1)
    yy = [yy Ymn];
end;
yy = yy.*cos(order*phi);

order = max(max(abs(yy)));
rho = 5 + 2*yy/order;

% Apply spherical coordinate equations
r = rho.*sin(theta);
```

```
x = r.*cos(phi);    % spherical coordinate equations
y = r.*sin(phi);
z = rho.*cos(theta);

% Plot the surface
clf
surf(x,y,z)
light
lighting phong
axis tight equal off
view(40,30)
camzoom(1.5)
```