Visualizing Dynamical Systems

Matthew Jee

March 2014

Abstract

This paper presents an application that facilitates the exploration of the relationship between the parameter space and state space of a dynamical system. The application can integrate and display three-dimensional, first-order differential equations, with an interface to 'scrub' through the parameter space in realtime.

1 Introduction

1.1 Motivation

A dynamical system is a mathematical model for describing the evolution of a system by a fixed rule, usually over time. More formally, a system consists of a state space, and a function mapping a given point in the state space and a particular time to a new position in the state space [1]. In addition to time and the current position, the evolution function of a dynamical system is usually dependent on set of parameters. It can be difficult to discern the effects of a system's parameters on its evolution by analyzing its rule function alone. Also, qualitative features of a dynamical system, such as critical points and limit cycles, can be much easier to discover 'at a glance' when the system is plotted visually. The aim of this project is to create an visualization tool that facilitates the exploration of the relationship between the parameter and state space of dynamical systems.

1.2 Original Goals

This application will render the dynamical system by first computing the evolution of a set of points in the state space using numerical integration. These points will be mapped to spatial dimensions and rendered as a path. Consequently, the application will be limited to computing and rendering systems with a state space of three or fewer dimensions. The application will also attempt to detect critical points and limit cycles within each evolution and mark them in the rendering. Methods for visualizing the stability of parameter configurations will be explored.

The application will be able to render paths for both discrete and continuous systems. However, continuous systems will be limited to first-order differential equations because it is more computationally expensive to integrate systems of arbitrary degree. If there is time, a more general integration scheme will be implemented.

The application will present controls for modifying the parameters of the system. Whenever a parameter is modified, the system evolution will be re-computed and re-rendered. If this turns out to be too computationally expensive to do in real time, methods for progressively computing the evolution will be explored. This kind of interactivity will allow the user to gain a more intuitive sense of the relation between parameters and the state space.

Functions for the Lorenz System, the Rossler System, and the Logistic Map, will be built in to the application. If time permits, a system that allows the user specified evolution functions will be implemented.

1.3 Related Works

There exist many programs that can render dynamical systems. Mathematica has the capability to plot a massive variety of systems and equations, dynamical systems included [11]. Matlab similarly provides ways to visualize dynamical systems. Both are probably more flexible than this project in terms of what can be integrated, but have more cumbersome interaction.

There is another program called Chaoscope, which renders very nice looking pictures of chaotic systems[8]. This aim of this project is similar to that of Chaoscope, but with less of a focus on aesthetic value and more on exploration of system properties.

2 Implementation

2.1 Visualization Methods

The application can display an arbitrary number of windows, each containing a visualization of a single dynamical system. These systems can either be chosen from one of the defaults (Lorenz or Rossler), or loaded from a Javascript file (see 2.2 for details).

Each window consists of three primary components:

- The Parameter-Space View
- The State-Space View
- The 'Sidebar' manipulation controls

The parameter-space view is used to 'scrub' the parameter space of the dynamical system. Within the view are two 'handles' that can be moved around the space with the mouse. These two handles are the start and end points of a linear path through the parameter space, so a gradient-colored path is drawn between them. This path forms the set of parameter configurations that are rendered in the state-space view. Since the path is continuous, it is first discretized into a finite number of configurations before being used to evaluate evolutions. The number of parameter configurations to evaluate can be set by a slider labeled 'evolutions'.

The state-space view contains a number of seeds that are used to specify the starting position for evolutions. For each of these starting positions, one path is evaluated and rendered for each parameter configuration given by the parameter-space view. Each path in the state-space view is given a color that corresponding to a color along the path in the parameter-space view. By doing this, it is possible to see the qualitative effects of transitioning linearly from one parameter configuration to another. In particular, this multi-evolution visualization makes it easy to see bifurcations.

This system works very well for certain dynamical systems, but there are some catches. One limitation is that the parameter-space view is only capable of displaying three dimensions. This is fine for systems with three or less parameters (Lorenz, Rossler, Chen-Lee), but systems with more parameters (Lorenz-84) cannot have their entire parameter space mapped to the view. To alleviate this concern, there are three menus which users can use to choose which parameters they wish to map onto the axes. Even with this solution, there still exists a need to manipulate parameters that are not mapped to the parameter-space view. To fix this issue, a table of all the parameters and their current start and end values is displayed beside the spatial views. Through this table, the user can enter specific parameters by typing and both views will update automatically with the new configuration.

There is also the issue of integration error. At some points in the state-space, the integration becomes very jumpy and inaccurate. This was addressed by providing a slider to change the integration step size. A better way to address this would be to use adaptive Runge-Kutta methods. I was not able to implement this in time.

During development, I considered implementing the ability to specify arbitrary splines within the parameter space, rather than just a simple line with two handles. This would allow more flexible analysis of transitions between configurations. However, it was determined that this would require a large amount of work only tangentially related to the project (a spline renderer and manipulation interface), so it was not implemented.

2.2 Technical Details

The application code is more or less split into four components:

- Dynamical Systems Simulator (C++)
- Renderer (C++)
- Custom-System Interface (C)
- User Interface (C, Objective-C)

Dynamical Systems Simulator The Dynamical Systems Simulator is written in C++ to take advantage of the STL, but still maintain some portability, which is not so great with Objective-C. In retrospect, this was not a good decision (see 2.2). The simulator is broken into three classes:

- Integrator

  This class performs numerical integration, given an Integrable, a position in state-space, and a set of parameters. An Integrable is defined to be a function pointer. An RK4 and Euler Integrator were written. A 'discrete' integrator, evaluating rules of the form $x_{n+1} = f(x_n, p)$, was planned, but was not created.

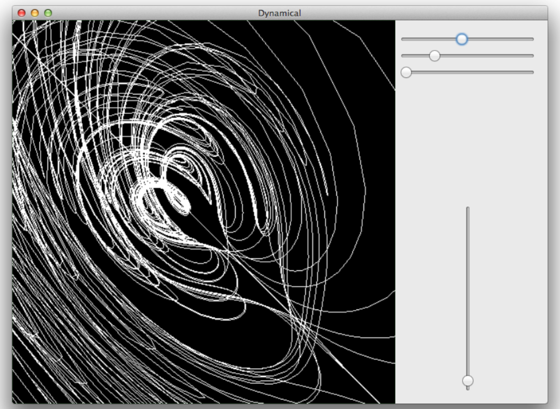


Figure 1: Early Integrator Test

- Parameter

  Parameter provides an abstraction over parameter values. Each Parameter contains a min/start, max/end, and current value.

- DynamicalSystem

  This class encapsulates and integrator, and integrable, and a set of parameters.

Renderer The renderer is a simple library that I wrote in 2013 that provides convenient abstractions over OpenGL [5]. It is slightly modified to interoperate well with the Dynamical Systems Simulator. The Vector3 class, which is shared with the simulator, was switched from floats to doubles for improved precision.

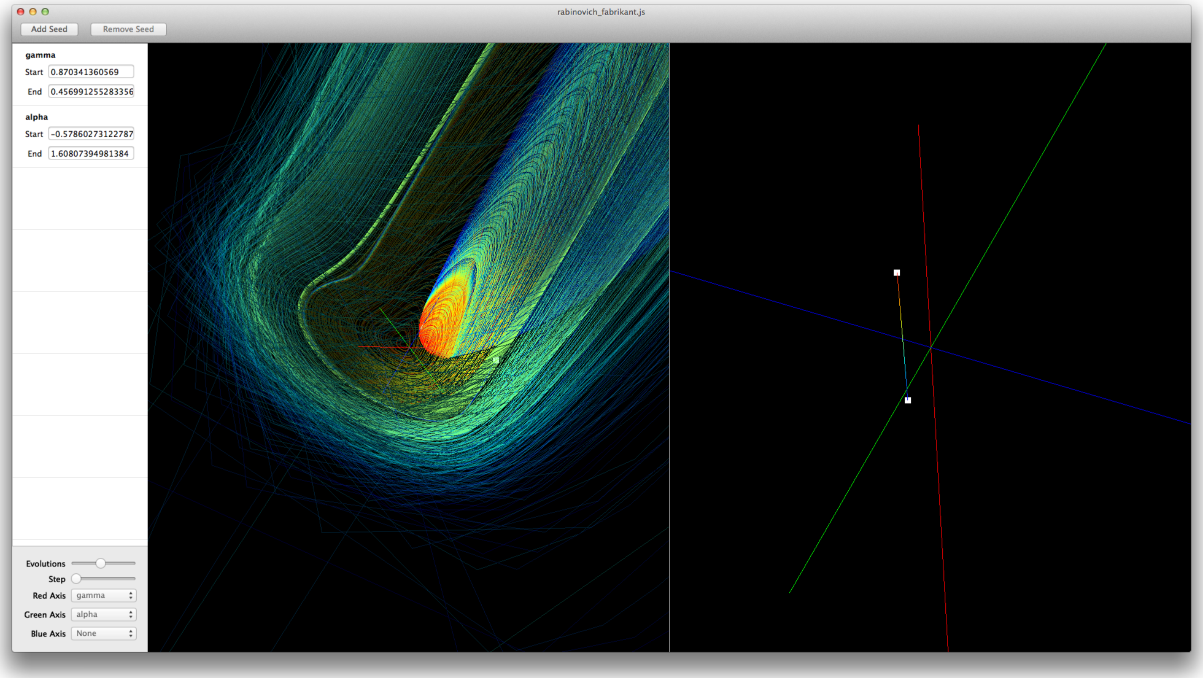
User Interface The Dynamical Systems Simulator was prototyped with GLFW [6] providing the window and OpenGL context. The final application uses Cocoa [7] for the user interface and windowing system. Cocoa was chosen because there are graphical tools to do GUI layout, which is a pain to do in code. All of the application viewports share a single OpenGL 3.2 context, all running on the same thread. Each view is updated lazily to save CPU time.

Custom System Interface The Custom System Interface is the code that maps user-specified dynamical systems written in a scripting language to the Dynamical Systems Simulator. Originally, the plan was to use Python as the scripting language. However, Javascript was chosen instead for several reasons. Firstly, the embedded Python API does not allow for multiple Python contexts/interpreters, which is almost necessary given the way that the Dynamical Systems Simulator was implemented (see below for details). Secondly, the speed of modern Javascript runtimes is surprisingly fast compared to the current Python runtime. The Benchmarks Game [4] shows Google's V8 Javascript engine performing much faster than Python 3. This project does not use V8, but instead uses Webkit's JavascriptCore, which is somewhat slower than V8 but still quicker than Python.

Implementing user-specified systems was tricky because the evolution functions are represented by function pointers, and it is not possible to add new functions at runtime. To work around this, each user function is given it's own Javascript context. Each context contains a function named 'evolution' which evaluates the user function. The engine maintains a single 'current' context which is used when evaluating evolution functions. Thus, all of the custom user functions can be routed through a single C function, as long as the current context is switched appropriately. This explanation may not be so clear, so a diagram has been provided.

The Javascript evolution function engine is not reentrant and not suitable for multithreading, which is problematic if the application were to be scaled to multiple threads for performance. However, this problem really lies with the way the dynamical system library depends on function pointers. C++ functors and Objective-C blocks were considered as alternatives to function pointers, but this results in a strange mix of C, Objective-C, and C++ (see 2.2).

An Aside on Languages This project uses a mix of C, C++, and Objective-C. This is not good. The resulting codebase is a very messy mix of C++ style object-orientation, Objective-C style object-orientation, and C procedurality. Most of the renderer and simulator could be rewritten in straight C, which would do wonders for the code complexity. At the start of the project it seemed like a good idea to take advantage of an existing C++ codebase and mix it with an Objective-C UI to speed up development. Alas, this only served to make development more difficult in the later stages.

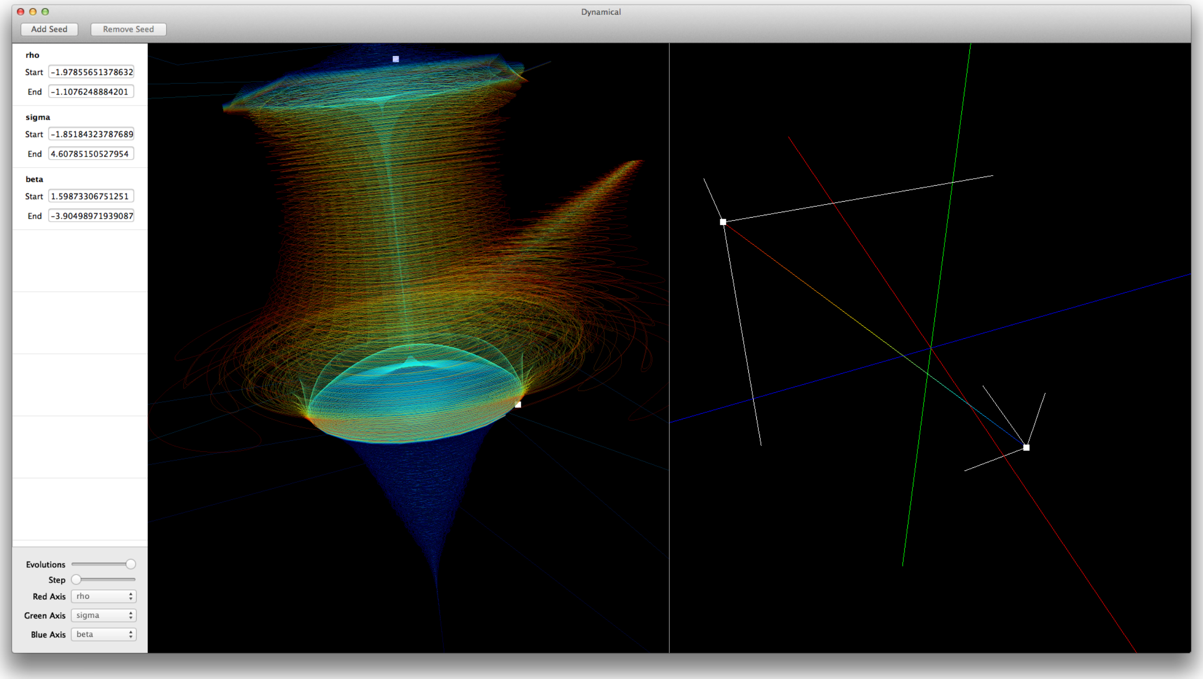


gamma
Start 0.870341360569
End 0.456991255283356

alpha
Start -0.57860273122787
End 1.60807394981384

Red Axis gamma
Green Axis alpha
Blue Axis None

Figure 2: Rabinovich-Fabrikant Equations, $\gamma = 0.87, \alpha = -0.58 \rightarrow \gamma = 0.46, \alpha = 1.61$



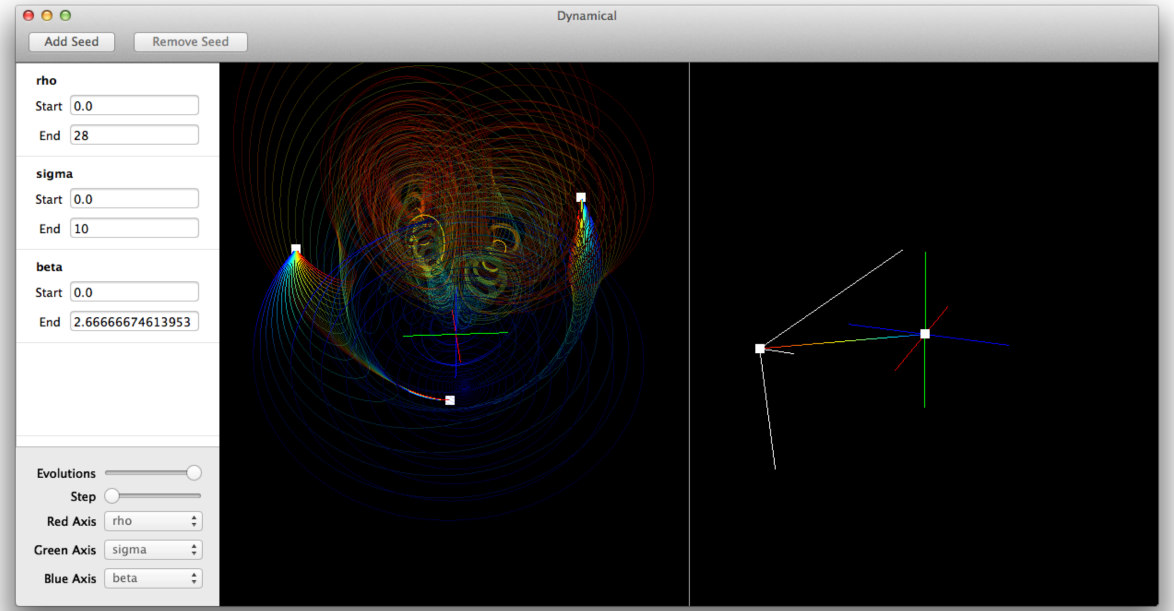
rho
Start -1.97855651378632
End -1.1076248884201

sigma
Start -1.85184323787689
End 4.60785150527954

beta
Start 1.59873306751251
End -3.90498971939087

Red Axis rho
Green Axis sigma
Blue Axis beta

Figure 3: Lorenz System, $\rho = -1.98, \sigma = -1.85, \beta = 1.60 \rightarrow \rho = -1.11, \sigma = 4.61, \beta = -3.90$



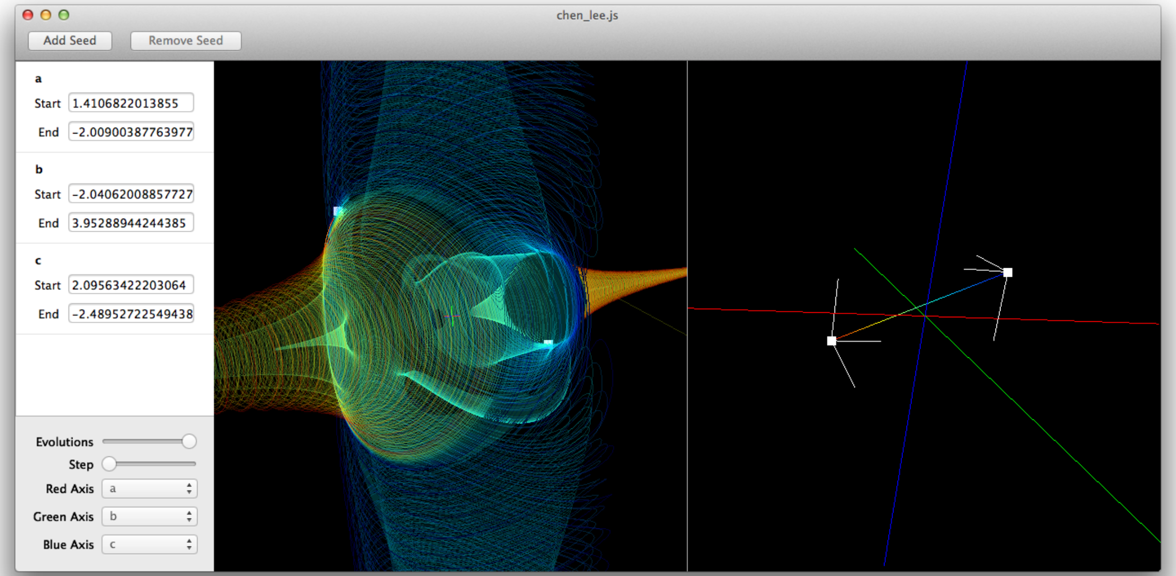
rho
Start 0.0
End 28

sigma
Start 0.0
End 10

beta
Start 0.0
End 2.66666674613953

Red Axis rho
Green Axis sigma
Blue Axis beta

Figure 4: Transitioning from $\rho = 0, \sigma = 0, \beta = 0$ to $\rho = 28, \sigma = 10, \beta = \frac{8}{3}$ in the Lorenz System



a
Start 1.4106822013855
End -2.00900387763977

b
Start -2.04062008857727
End 3.95288944244385

c
Start 2.09563422203064
End -2.48952722549438

Red Axis a
Green Axis b
Blue Axis c

Figure 5: Chen-Lee System, $a = 1.41, b = -2.04, c = 2.10 \rightarrow a = -1.01, b = 3.95, c = -2.49$

3 Results

What follows are several qualitative analyses of renderings generated by the application. I am not very well versed in the study of dynamical systems, so the analyses are crude and may not make much sense; they are just meant to show what kind of patterns the visualization can reveal.

3.1 Rabinovich-Fabrikant Equations

Figure 2 shows the application rendering several evolutions of the Rabinovich-Fabrikant equations [12] for some arbitrary parameter values. This set of equations only has two parameters, so only the Red and Green axes are mapped to parameters. The evolutions close to the start parameter configuration are displayed in 'cooler' (blue, green) colors, while the evolutions closer to the end configuration are 'warmer' (orange, red). The rendering shows that this transition in parameters results in shorter cycles that are closer to the origin.

The rendering was generated with the following custom system definition:

```
parameters = ["gamma", "alpha"];

evolution = function(param, x, y, z) {
    return [
        y*(x-1+x*x)+param[0]*x,
        x*(3*z+1-x*x)+param[0]*y,
        -2*z*(param[1]+x*y)
    ]
}
```

3.2 Lorenz System

Similar to the Rabinovich-Fabrikant example, figure 3 shows a set of evolutions for an arbitrarily chosen configuration of parameters, but for the Lorenz System. It is clear from the rendering that evolutions with a configuration close to $\rho = -1.98, \sigma = -1.85, \beta = 1.60$ have a tendency to 'explode' towards infinity. As the parameter configuration tends towards $\rho = -1.11, \sigma = 4.61, \beta = -3.90$, the evolution paths cling more tightly to the 'vertical' axis, which is shown by the blue-green cylindrical paths within the wider, reddish cylindrical paths. Then very close to the end configuration, the system evolves towards infinity again, although in a different direction (the dark blue paths).

Figure 4 shows the transition from the origin of the parameter space, to the classic Lorenz Attractor values of $\rho = 28$, $\sigma = 10$, and $\beta = \frac{8}{3}$. The attractor is rendered in red.

3.3 Chen-Lee System

Figure 5 displays a rendering of the Chen-Lee system [13]. The rendering shows reddish paths progressing towards infinity along the z-axis, while the blue paths tend toward infinity along the y-axis. The light blue-green paths seem to form limit cycles around the center of the system. The light blue-green paths represent an intermediate step between the red and dark blue parameter configurations. Through the visualization it is possible to see how that parameter transition affects the outcome of the system.

The rendering was generated with the following custom system definition:

```
parameters = ["a", "b", "c"];

evolution = function(param, x, y, z) {
    return [
        -y*z+param[0]*x,
        x*z-param[1]*y,
        (1/3)*x*y-param[2]*z
    ]
}
```

4 Conclusion

Through parameter-space 'scrubbing', the presented application allows the user to easily visualize how linear changes in the parameter space effect the state space of a system. However, this was only one of my goals for the project.

One of the original goals was to have the application detect and display limit cycles and critical points. This feature was scrapped because detecting limit cycles proved to be much more difficult than I had originally thought. A naive implementation would be $O(n^2)$, since it would necessitate checking each previous point along the current evolution for each new point. I tried to research better approaches, but I was not able to find any methods that I could implement in time.

Another goal that I was not able to meet was adding a feature that would characterize and visualize the stability of configurations in the parameter space. The idea was to calculate some metric representing the stability of the system for each point in the parameter space, and render a colored glyph at that point. The problem is that system stability is sensitive to initial position as well as parameter configuration. this means that the visualization would require rendering points in $n + m$ dimensions, where n is the dimension of the state space and m is the number of parameters. Most of the the systems that I used for testing have $m = 3$ parameters, and $n = 3$ state dimensions, so the visualization would require six dimensions. I was not able to implement any of the various high-dimensional data mapping techniques in time to make this visualization.

Further work on this project will probably involve developing these two missing features.

References

[1] Dynamical Systems - Scholarpedia `http://www.scholarpedia.org/article/Dynamical_systems`

[2] Adaptive Step-size `http://en.wikipedia.org/wiki/Adaptive_stepsize`

[3] On the Detection of Limit Cycles by the Variational Velocity Method `http://link.springer.com/article/10.1023%2FA%3A1006158306350`

[4] Benchmarks Game `http://benchmarksgame.alioth.debian.org/u64/benchmark.php?test=all&lang=v8&lang2=python3&data=u64`

[5] Github - marmphco - fluidsim `https://github.com/marmphco/fluidsim/tree/master/src`

[6] GLFW - An OpenGL Library `http://www.glfw.org`

[7] Cocoa - OS X Technology Overview - Apple Developer `https://developer.apple.com/technologies/mac/cocoa.html`

[8] Attractors in Chaoscope `http://www.chaoscope.org/doc/attractors.htm#polynomial_a`

[9] Bifurcation - Scholarpedia `http://www.scholarpedia.org/article/Bifurcation`

[10] List of chaotic Maps - Wikipedia `http://en.wikipedia.org/wiki/List_of_chaotic_maps`

[11] Wolfram Demonstrations Project: Lorenz Attractor `http://demonstrations.wolfram.com/LorenzAttractor/`

[12] Stochastic self-modulation of waves in nonequilibrium media, Soviet Journal of Experimental and Theoretical Physics, Rabinovich, M.I. and Fabrikant, A.L.

[13] Generation of hyperchaos from the ChenLee system via sinusoidal perturbation `http://www.sciencedirect.com/science/article/pii/S0960077907000410`