Chaos in Physics Project Write-up

Ali Chaudhary
67694450

The main goal of this project quite simply was to adequately demonstrate chaos in Physics through examples using C-programing and numerical analysis. There were two mains ways I intended to demonstrate this. One was to compare the results of a simulation of the single pendulum to a simulation of the double pendulum. The double pendulum in theory will output an unpredictable motion that will should the patently chaotic and sensitive nature of this model. The second and much simpler method in which I was able to demonstrate chaos in physics was through the Lorenz system. Through mapping a Lorenz system in C and prompting the user to give two different inputs I would show the extreme, chaotic differences in the two outputs.

As far as how I was able to actually do this and accomplish the goals of my project, I needed to implement the Runge-Kutta numerical method. As basic physics tells us, the basic differential equation for a single pendulum, assuming no driving function, is: $\frac{d^2\theta}{d^2} + \frac{g}{l}\sin\theta = 0$. This however is not something I could simply plug into a C program and get an output, nor could I even use this in the Runge-Kutta method as the equation is a *second order* differential equation, incompatible with the method. So instead I had to cast the differential equation into *two first order* differential equations that could simulate the pendulum. This very simply got me these two new equations: $\frac{d\theta}{dt} = \omega$ and $\frac{d\omega}{dt} = -\frac{g}{l} * \sin\theta$ where ω represents the angular velocity, g represents the gravity force and l represents the length of the string or rod of the pendulum. Then I wrote a series of print statements to tell the user what the program did, what a pendulum was, and to prompt the user to give inputs. The inputs I needed for the single pendulum was an initial angle for the pendulum, a length for the pendulum, and a mass for the pendulum. The mass is actually unimportant for simulating the pendulum, but will be important once we simulate the double pendulum. Once I had these equations and user inputs I subsequently implemented them by creating two functions in my C program that returned the value given by one of the equations as a double. Then, in order to implement the Runge-Kutta, method I created a function that returned nothing and instead changed two double pointers which represented the value of the function at a given point in time. This function was called in the main function up to the max time through a time interval of 0.1 seconds. As far as the max time was concerned, I did not want to use some arbitrary time like 10 seconds, in case the user picked a length that made it impossible to see the motion. Instead I used the equation for the period of a pendulum, $T_0 = 2\pi\sqrt{\frac{l}{g}}$, plugged in the user input and multiplied by five to make sure that the user would see at least 5 periods of the pendulum. When the Runge-Kutta 4 function was called, I had to manipulate the data to display the pendulum motion in gnuplot. This is because as it stands the Runge-Kutta method would only return the angle with respect to time, so it was necessary to use the appropriate trigonometric function to convert the angle to an x-coordinate and a y-coordinate. Once I had the program outputting all of this data, instead of printing out the results straight to the console, I used file pointers to print the data to a ".dat" file, which will be used later in a gnuplot macro.

As it pertains to the double pendulum, I simulated it in the same program as the single pendulum. This quite simply was done in order to provide simplicity to the user so as to not force the user to run multiple programs to see the chaos effect. The methodology of doing the double pendulum was, as intuition would suggest, just like doing two pendulums, except with two new equations. After casting the first derivative equation for the first pendulum was the derivative of theta with respect to time: $\frac{d\theta}{dt} = \omega_1$. The second equation was the derivative of the angular velocity: $\frac{d\omega_1}{dt} = \frac{m_2 l_1 \omega_1{}^2 \sin(\theta_2-\theta_1)\cos(\theta_2-\theta_1)+m_2\sin(\theta_2)\cos(\theta_2-\theta_1)+m_2 l_2 \omega_2{}^2 \sin(\theta_2-\theta_1)-(m_1+m_2)g sin(\theta_1)}{(m_1+m_2)l_1-m_2 l_1(\cos(\theta_2-\theta_1))^2}$ where the variables represent the parameters of the pendulums and g represents gravity. The equations for the second pendulum were similarly $\frac{d\theta}{dt} = \omega_2$ for the first equation and $\frac{d\omega_1}{dt} = \frac{-m_2 l_2 \omega_2{}^2 \sin(\theta_2-\theta_1)\cos(\theta_2-\theta_1)-(m_1+m_2)(g sin(\theta_1)\cos(\theta_2-\theta_1)-l_1\omega_1{}^2\sin(\theta_2-\theta_1)-g sin(\theta_2))}{(m_1+m_2)l_1-m_2 l_1(\cos(\theta_2-\theta_1))^2}$ for the second equation. Here the equations were painstakingly scribed into the c program, and then I created another Runge-Kutta 4 function which updated the values received from all four of these equations through the use of pointers. This was significantly more difficult than implementing the Runge-Kutta 4 method for the single pendulum. This is simply because the incredible amount of moving parts to the double pendulum equation made it hard to track where which variable was to be inputted. Once I had the values once again I had to convert the values to Cartesian coordinates but, this time I also had to shift the second set of coordinates corresponding to the bottom pendulum. This is because the angle received from the equation is from the base that is the first pendulum. And once again the data was stored into a ".dat" file for the gnuplot macro to deal with.

Once I had all the data and used gnuplot it was easy to see the Chaotic motion of the double pendulum. As viewable by the "png" files, just a slight change in the initial conditions created a vastly different outcome. This outcome was not only completely different to any other outcome, but it also was entirely unpredictable. There was absolutely no formulaic method from which I was able to predict what outcomes I would receive from entering different masses or lengths or initial angles for the pendulums. One other fun thing that I was able to do with gnuplot is create a "gif" from the data points using a gnuplot macro with a counter, an "if statement", and the replot function. For each time the counter iterated I plotted five more points from the ".dat" file that I had created earlier. This allowed me to create the illusion of motion and save each iteration into the "gif" file. I frustratingly had to do this simply because gnuplot 4.2 does not have loops which were added in gnuplot 4.6, and the linux system that the High Power Computer Cluster at UCI uses is equipped with gnuplot 4.2.

The next itinerary on the to-do list of this project was mapping the Lorenz Attractor in C. The output for the program was expected to be the typical solutions of the Lorenz Attractor, an unpredictable "butterfly" shaped plot that would chaotically vary based on small variations of the inputs. The Lorenz Attractor is actually a system of three diferential equations. The first being, $\frac{dx}{dt} = \sigma(y - x)$ for the x coordinate, the second being $\frac{dy}{dt} = x(\rho - z) - y$ for the y coordinate, and the final being $\frac{dz}{dt} = xy - \beta z$ for the z coordinate. The inputs that we will vary here are sigma, rho, and beta. These parameters are all that are needed to be changed to create the chaotic effect; the simplicity of these equations is the main reason that early on the Lorenz Attractor was the "cookie cutter" example of chaos in Physics. Comparing to the double pendulum, the Lorenz Attractor is a far simpler and quicker demonstration of chaos. As far as the C code itself is concerned, firstly, as in the pendulum and in the double pendulum examples, the inputs were taken from the user, in this case sigma, rho, beta, and "Nmax", the max number of points to plot. In this case I gave the user the option to choose the points, while suggesting to the user a decent number, simply because there wasn't a nice way to decide how long to run the simulation for like there was in the periodic pendulum. Once I got the data and checked to make sure the user inputted valid numbers I moved on to using the Runge-Kutta method. It was unnecessary to use the method in which lists are utilized, even though this system had greater than two equations. This is simply due to the simplicity of the equations as they allowed for the mapping of the equations to be done very simply. Once the equations were implemented all that was left to do was to use the "fprint" function and file pointers to print the data to a ".dat" file for gnuplot to use later. This is because unlike the pendulum example, these equations simply give out the x, y, and z coordinates, so there are no additional calculations needed to make the numbers graph ready. Once in gnuplot I was able to use "splot" to get a 3d interactive plot of the data. Again using macros I was able to easily do multiple graphs and not have to go through numerous series of command lines over and over again. Plotting the Lorenz Attractor for multiple initial values allowed me to clearly visualize the distinct differences in the outcomes. Although it was somewhat predictable in that all the functions ultimately produced the "butterfly" effect, it was completely unpredictable how they would fill that effect, and how the final "butterfly" would actually look like. As far as getting a ".gif" file working, I tried the same methodology that got me the desired result with the double pendulum, the only problem in this case is that the large number of data points, and a flagrant absence of any sort of compression software built in gnuplot, led to huge data files that were not realistic to work with so had to be discarded. Finding a way to compress gif files in gnuplot is thus a future project.

All in all this project's main goals of demonstrating chaos in physics is adequately accomplished through the attached images, gifs, and code. The main takeaway from the project was how powerful some numerical methods can be, to the point where they are necessary to solve equations that can’t be solved analytically.