Summer 2019

The effects of finite precision on the simulation of the double pendulum

Rebecca Wild

The Effects of Finite Precision on the Simulation of the Double Pendulum

An Honors College Project Presented to

the Faculty of the Undergraduate

College of Integrated Science and Engineering

James Madison University

by Rebecca Lynn Wild

April 2019

Accepted by the faculty of the Department of Computer Science, James Madison University, in partial fulfillment of the requirements for the Honors College.

PUBLIC PRESENTATION

This work is accepted for presentation, in part or in full, at the Computer Science Department Research Seminar on April 19, 2019.

Contents

List of Figures

Abstract

We use mathematics to study physical problems because abstracting the information allows us to better analyze what could happen given any range and combination of parameters. The problem is that for complicated systems mathematical analysis becomes extremely cumbersome. The only effective and reasonable way to study the behavior of such systems is to simulate the event on a computer. However, the fact that the set of floating-point numbers is finite and the fact that they are unevenly distributed over the real number line raises a number of concerns when trying to simulate systems with chaotic behavior. In this research we seek to gain a better understanding of the effects finite precision has on the solution to a chaotic dynamical system, specifically the double pendulum.

1 Introduction

We can define a system as a set of interacting components. For example, the solar system. We can consider the sun and the planets interacting components since the pull of the sun's gravity causes the planets to stay in orbit. A major aspect of studying systems is determining how they change and evolve over time. This area of study is known as dynamics and systems that evolve over time are called dynamical systems.

Systems can be composed of several components. For example, we can study the solar system with respect to just the sun and the planets or we can study it with respect to the sun, planets, moons, and other celestial bodies. The more variables we add the more complicated and mathematically cumbersome the system becomes. This is why in many cases the only effective and reasonable way to study the evolution of such systems over time is to abstract and simulate the event on a computer. However, because computers cannot represent every real number exactly, they become a source of error. Thus, any solution obtained from a simulation is an approximation.

For many systems, this limiting feature of computers has little effect and the approximated solutions are within some tolerable error bound. However, other systems can exhibit extreme sensitivity to any error induced by the computer. These systems are classified as chaotic because of extreme sensitivity they express to any perturbation in the data used to calculate the solution.

In this thesis, we seek to gain a better understanding of the effects finite precision has on the solution to a chaotic system and specifically the double pendulum. In particular, given some number of bits, we want to find out how long we can run a simulation before the trajectory diverges. We also seek to determine which variable of the system is most significantly impacted by the change in precision. By doing this analysis we hope to gain insight into how dependable computer simulations of chaos truly are.

2 Background

When approximating solutions numerically, it is important to question the validity of those solutions. This is especially important when modeling non-linear chaotic dynamical systems such as the double pendulum. This section is divided into four parts. First we will discuss the double pendulum and present the equations of motion. Second, we will define chaos and how it is measured. Third, we will discuss the sources of error present in computer simulations and finally, we will discuss the effects the error will have on our simulation.

2.1 The Double Pendulum

A single pendulum is a mass hanging from a fixed point that can swing back and forth. The double pendulum, depicted in Fig. 1, is a single pendulum with another pendulum attached to its mass. In this project we choose to study this system because it is known to exhibit chaotic behavior [2]. However, as an additional motivational point, the behavior of various mechanical systems can be understood in terms of the double pendulum. For example, cranes, robot arms, and even human movements such as the golf swing can be modeled as the double pendulum [2].

There are many variations of this system that could be studied. For example, we could vary the length and mass of each pendulum to obtain different systems with varying behavior. In [3] it was observed that for a particular initial condition the chaotic tendency of the double pendulum increases as the length or mass of the lower pendulum increases. We could also choose to permit motion in three dimensions or use pendulums with point or distributed masses. In this project however, we restrict motion to the two dimensional plane and use point masses.

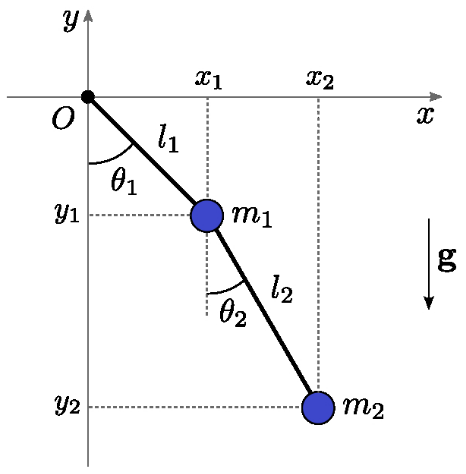


Figure 1: A free-body diagram of a double pendulum. We denote the angle, length, and mass of the first pendulum as θ_1, l_1, and m_1, respectively and denote the angle, length, and mass of the second pendulum as θ_2, l_2, and m_2, respectively.

With respect to the notation used in Fig. 1 we can write the position equations for the point mass of each pendulum as follows

$$\begin{aligned}
x_1 &= l_1 \sin\theta_1 \\
y_1 &= -l_1 \cos\theta_1 \\
x_2 &= l_1 \sin\theta_1 + l_2 \sin\theta_2 \\
y_2 &= -l_1 \cos\theta_1 - l_2 \cos\theta_2.
\end{aligned}$$

By calculating the velocity (the first derivative) and the acceleration (the second derivative) of the position equations and performing some algebraic manipulations we can obtain the following equations of motion

$$\begin{aligned}
\dot{\theta}_1 &= w_1 \\
\dot{w}_1 &= \frac{m_2 l_1 w_1^2 \sin\Delta\cos\Delta + m_2 g \sin\theta_2 \cos\Delta + m_2 l_2 w_2^2 \sin\Delta - (m_1+m_2) g \sin\theta_1}{(m_1+m_2) l_1 - m_2 l_1 \cos^2\Delta} \\
\dot{\theta}_2 &= w_2 \\
\dot{w}_2 &= \frac{-m_2 l_2 w_2^2 \sin\Delta\cos\Delta + (m_1+m_2)(g\sin\theta_1\cos\Delta - l_1 w_1^2 \sin\Delta - g\sin\theta_2)}{(m_1+m_2) l_2 - m_2 l_2 \cos^2\Delta}
\end{aligned}$$

where w_1 and w_2 denote the angular velocity of the first and second pendula, respectively and $\Delta = \theta_2 - \theta_1$. A detailed description of the derivation of these equations can be found in [4]. Given the size of the angular velocity equations and the number of variables to be considered, it would be nearly impossible to conduct mathematical analysis to determine the behavior of this system. The double pendulum is a clear example of a system that must be analyzed numerically.

2.2 Chaos

For certain values of energy, the double pendulum is known to exhibit chaotic behavior [2]. In [1], Strogatz defines chaos as aperiodic, long-term behavior in a deterministic system that exhibits *sensitive dependence* on initial conditions; two trajectories starting infinitesimally close to each other will diverge exponentially over time. To illustrate, refer to Fig. 2 and suppose $x(t)$ is some point on a trajectory at time t. Consider the other nearby point $x(t) + \delta(t)$ where $\delta(t)$ is the distance between the two points. Let δ_n be the separation at the n^{th} iteration. If the system we are modeling is chaotic, we observe that $|\delta_n| \sim |\delta| e^{n\lambda}$ where λ is known as the Liapunov Exponent. This tells us that δ_n increases exponentially at each iteration. The exponent, λ, is a statistic measuring the rate of divergence between trajectories. When it is positive, we know the trajectories are diverging away from each other at an exponential rate, which means it is exhibiting chaotic behavior.

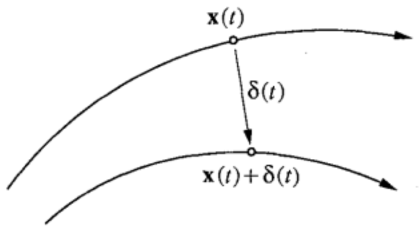


Figure 2: Divergence of two trajectories with extremely close initial conditions [1].

The ability to measure the chaos exhibited by a system for a particular initial condition is important when conducting analysis if its behavior over time. As stated, the double pendulum exhibits chaotic behavior for high energy values. Thus, not all initial conditions will necessarily result in chaotic behavior. For example, the initial condition ($\theta_1 = 0, w_1 = 0, \theta_2 = 45, w_2 = 0$) would not have high enough potential energy at the start to instigate chaotic motion. In fact, it would behave similarly to the single pendulum, and eventually reach equilibrium. Since many studies have been conducted on the pendulum there are many initial conditions known to lead to chaos. In [2] a positive Liapunov Exponent was calculated for initial conditions where $\theta_1 = \theta_2 \geq 90°$ and $w_1 = w_2 = 0$. In our research, we will choose to examine initial conditions in this range.

2.3 Sources of Error

While the double pendulum is the perfect example of a system that is best analyzed numerically, the high sensitivity it exhibits is cause for concern. Consider if we were to model the double pendulum physically; it would exhibit sensitive dependence to the friction at the fixed point and air resistance. When modeling the double pendulum numerically, we are able to control friction and air resistance in our calculations by neglecting these variables in the mathematical model. However, when we simulate it using a computer, we subject it to small perturbations in the data induced by the way computers represent and perform arithmetic operations on numbers. These forms of error cannot be controlled easily.

The three main sources of error that arise in numerical computation are data uncertainty, discretization, and rounding [5]. Data uncertainty occurs when there are errors in the measurements or estimates given as initial input to the system. This is primarily an issue when initial values are coming from something measured such as a population size, which could be inaccurate. Data uncertainty is not a source of error in this project since we are able to input the exact conditions we want at the start of the simulation. Discretization error is the result of evaluating the solution numerically for a finite number of discrete time values. This is opposed to evaluating the solution over a continuous interval of time, as is done in mathematical analysis. The significance of this error depends on the chosen step size, the distance between each time value at which the solution is evaluated. While there is a wealth of literature on the subject [5], in this project we will not be concerned with discretization error and the effects of data uncertainty. Instead, we will focus on the effects of rounding error which is consequent of floating-point operations [6] and limited word size. The word size of a number (the number of bits used to store the number digitally) and the result of floating-point operations on that number, such as the rounding of bits, determine how accurate the output is going to be.

Representing numbers in floating-point notation is similar in concept to representing numbers in scientific notation. The representation consists of a sign bit, s, a signed exponent, e, and a significand, M that is normalized, meaning the first bit is 1. The value of the floating-point number is then given by $(-1)^s + M \times 2^e$. This restricts us to at most $2^n - 1$ distinct values given n bits, which discretizes the real number line. Since we have a limited number of bits to represent computed numbers, rounding must occur. This has a significant impact on the approximation of chaotic solutions.

2.4 Effects of Error on Chaotic Systems

The fact that the set of floating-point numbers generated by n bits is finite and unevenly distributed over the real number line poses a number of research questions. Recall that one of Strogatz's requirements for chaos was aperiodic behavior. However, since we are evaluating solutions to the system using a finite set of values one would expect that it would eventually repeat itself. In [7] it was observed that since the set of floating-point numbers is finite, all computed trajectories of a system are ultimately periodic. That is, they repeat after some fixed period of time. While this periodic behavior may take a long time to manifest (depending on the precision used in the computation) this implies that chaos cannot be sustained forever in computer simulations.

While in our work we will focus on the effects of rounding, it is important to note that discretization error can have a similar influence on the accuracy of calculated solutions. Research has shown that both discretization and rounding errors have the ability to suppress or induce chaos during a simulation. Results in [8] show how certain numerical methods can actually suppress real chaos in a system due to the effects of discretization. They show that for large enough step sizes the implicit Euler method can artificially stabilize unstable fixed points and completely destroy known chaotic attractors. An unstable fixed point is an equilibrium point that nearby solutions diverge away from. In [8] it is also shown that floating-point simulations of a discrete map with a globally attracting fixed point at the origin can appear chaotic for extremely long times in single (32-bit), double (64-bit), or extended precision (80-bit). The artificial chaos in this example is induced by the rounding effects of finite precision.

The fact that chaos was obtained in single, double, or extended precision in the results of [8] may seem counter-intuitive. It is important to note that higher accuracy is not necessarily always obtained by higher precision. While this is true for algorithms with error bounds proportional to the unit round off (which is the distance between 1 and the next closest floating-point number) it may not always be the case for chaotic systems as shown in [8]. Another important thing to note is that inaccurate solutions are not always due to the accumulation of many rounding errors over time. As shown by example in [5] it is possible for a single rounding error to be the cause of poor results. This is especially plausible when modeling chaotic systems since in theory the perturbation of an initial condition at one step can drastically change future trajectories, regardless of the rounding error accumulated at later steps.

The ways in which the system can be influenced by discretization and rounding error will be important to consider when conducting our analysis. We want to be sure that we can correctly classify the type of error that influenced our results.

Through modeling the double pendulum system and conducting analysis on the rounding error associated with limited word size and floating-point operations, we can better understand how accurate simulations of chaos truly are. We can determine how much of an effect increasing the number of bits used to represent the variables has on the outcome of the simulation. By computing the solution over a fixed period of time for using 16-bit, 32-bit, 64-bit, 80-bit, and 128-bit precision we can gain insight regarding how much of an influence finite precision has on the solution of a chaotic system.

3 Methods

The goal of this project is to see how rounding error caused by limited word size can affect the simulation of the double pendulum. Given a particular initial condition we seek to obtain multiple long term solutions to the system where each solution is calculated using a different precision. Thus, the only independent variable in these calculations will be the number of bits used to store the significand of each parameter. In particular, we want to calculate multiple solutions using 11-bits, 24-bits, 53-bits, 64-bits, and 113-bits to store the significand of each variable used in the calculation. These values are chosen so the smallest significand value we use, 11, corresponds to IEEE half and the maximum, 113 corresponds to IEEE quad. These are the smallest and largest IEEE floating-point types, respectively. The standard floating point types and their significand size are listed in Table 1.

Name	Bits	Exp	Sig	Exp Range
IEEE half	16	5	10+1	-14-+15
IEEE single	32	8	23+1	-126-+127
IEEE double	64	11	52+1	-1022-+1023
C99 long double	80	15	64	-16382-+16383
IEEE quad	128	15	112+1	-16382-+16383

Table 1: The standard floating-point data types where Bits is the total number of bits used to store the value, Exp is the number of bits used to store the exponent, Sig is the number of bits to store the significand, and Exp Range is the range of exponents.

We use the C programming language to code a numerical solution to the double pendulum in order to obtain our data. However, since the floating-point types `single` and `double` are the only types supported in C, we will utilize the GNU MPFR (Multiple Precision Floating-Point Reliable) Library so that we may arbitrarily define the size of the significand of each variable in our code to be whatever we want. In this section we will discuss the methods used to 1) solve the double pendulum numerically, 2) write code where the precision of the double pendulum's parameters can be arbitrarily defined, and 3) analyze the data and visualize the results.

3.1 Solving the Double Pendulum Numerically

In order to solve the system of equations, $\dot{\theta_1}, \dot{w}_1, \dot{\theta_2}, \dot{w}_2$, computationally we will need to use a numerical method for integration. In this project we will use the fourth-order Runge-Kutta method, commonly known as RK4, for integration. This method was chosen because it is simple to implement and appears frequently in numerical analysis texts. It is not as time-efficient as other methods but runtime is not a concern of our research in this project. Another reason why we chose RK4 is because it can be used as a starting point for other numerical methods such as the fourth-order Adams-Bashforth method and the fourth-order Adams-Moulton method [2]. Future work may include comparing results generated using RK4 with results obtained from other methods.

As an example of how the Runge-Kutta method is implemented, consider the following initial value problem

$$\dot{y} = f(t, y); \quad y(t_0) = y_0$$

where y is a function of time t and the initial value of y at the initial time t_0 is defined to be y_0. In the case of the double pendulum, $\dot{y} = f(t, \theta_1, w_1, \theta_2, w_2)$. Given a step size h, we can approximate the solution to t_1 using the formula

$$\begin{aligned} y(t_1) = y_1 &= y_0 + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \\ t_1 &= t_0 + h. \end{aligned}$$

Thus, in general the solution at t_{n+1} is given by

$$\begin{aligned} y(t_{n+1}) = y_{n+1} &= y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \\ t_{n+1} &= t_n + h \end{aligned}$$

where

$$\begin{aligned} k_1 &= hf\Big(t_n, y_n\Big) \\ k_2 &= hf\Big(t_n + \frac{h}{2}, y_n + \frac{k_1}{2}\Big) \\ k_3 &= hf\Big(t_n + \frac{h}{2}, y_n + \frac{k_2}{2}\Big) \\ k_4 &= hf\Big(t_n + h, y_n + \frac{k_1}{3}\Big). \end{aligned}$$

Each k_i is a slope estimate. Referring to Fig. 3 for a depiction of these calculations we can see that k_1 is an approximation of the slope at the beginning of the time step, k_2 is an approximation of the slope at the midpoint using k_1, k_3 is an approximation of the slope at the midpoint using k_2, and k_4 is is an approximation of the slope at the end of the interval. When approximating y_{n+1} we take an average of these k_i values where greater weight is given to the increments at the midpoints.

The RK4 method has a local truncation error proportional to $(h)^5$ where h is the step size. The local truncation error is the amount of truncation error that occurs in one step of the numerical approximation. This is why the RK4 is a fourth-order method. The global truncation error, the cumulative error produced over many iterations, is proportional to $(h)^4$. The choice of the step size, h, is important to achieving

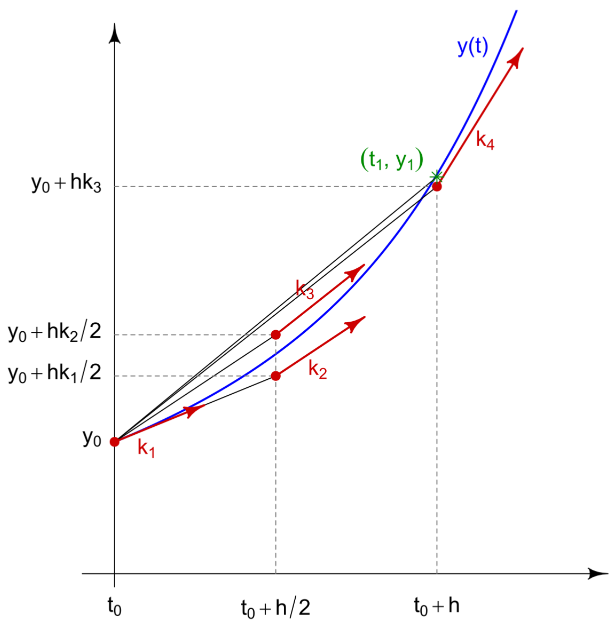


Figure 3: Depiction of the slopes used to approximate y(t) with RK4. [1].

accurate results. Due to truncation and rounding errors, it is not always the case that a smaller value for h will lead to more accurate results. Examples of this, as it relates to the double pendulum, are given in [2]. Using the Runge-Kutta formula to approximate the double pendulum they tested different values of h and found that the choicest time step is $h = 10^{-4}$. Their experiments show that smaller values result in less precision and a longer execution time. Based off of these results I chose to use $h = 10^{-4}$ as the time step in our implementation of the RK4 method for the double pendulum.

3.2 Writing Arbitrary Precision Code

We need to initialize the significand size of each variable used in the Runge-Kutta approximation to a particular value listed in Table 1. As mentioned previously, in C there are only two standard floating-point data types we can use. In order to represent all needed types we coded the Runge-Kutta approximation using the GNU MPFR library.

The GNU MPFR library [9] is a C library for multiple-precision floating-point computations. This library allows us to define explicitly the exponent range and significand size for any variable. It also supports various rounding modes that can be specified for each computation. In this project we used this library to code a portable, arbitrary-precision solution to the double pendulum.

To use MPFR to code our solution, instead of declaring all variables as one of the standard IEEE floating-point types such as `single` or `double`, we declare them all as `mpfr_t` objects. We are then able to use the function `mpfr_init2(mpfr_t x, mpfr_prec_t prec)` to set the significand of the `mpfr_t` object to be any length. The `mpfr_t` objects are treated as floating-point numbers internally. The

MPFR library uses the IEEE 754-2008 [6] standard for double-precision floating-point arithmetic. Thus, for the four arithmetic operations and the square root, given an `mpfr_t` object with a precision of 53 bits, MPFR is able to exactly reproduce all computations made with double-precision IEEE floating-point numbers, except that the default exponent range is much wider [9]. In particular, the following two declarations of x are equivalent

```
/* Declaration of x using MPFR library. */
mpfr_t x;
mpfr_init2(x, 53);

/* Declaration of x using IEEE double type. */
double x;
```

While MPFR does provide a means to set the exponent range it only allows you to set it globally. While setting the global exponent range may be of interest in future research, in this project I chose to use MPFR's default exponent range which will in general be much larger than the standard IEEE exponent ranges.

MPFR also allows us to specify the rounding mode we want to use for each computation. In this project I chose to use the library's `MPFR_RNDN` mode for rounding. This mode is equivalent to the IEEE 754-2008 `roundTiesToEven` mode which is the default rounding mode for floating-point operations. In this mode the floating-point value is rounded up or down to the nearest digit. In the case where the value to be rounded is exactly halfway between the two nearest numbers, the value is rounded to the nearest even digit [10].

Since MPFR allows us to specify the precision of our variables and the rounding standard used during computations, our code is portable across most modern machines. Anyone should be able to run this code and reproduce the results found this project as long as GNU and MPFR are installed on their machine. The arbitrary precision code we were able to create using this library made obtaining results for varying values of precision much faster and easier. If we had used the standard IEEE floating-point types we would have had to create multiple versions of our code, where each version had the variables declared as a different type and we would have had to find software libraries for each type without a hardware implementation. In our arbitrary precision code we simply loop through an array of significand values that correspond to those used IEEE half, IEEE single, IEEE double, C99 long double, and IEEE quad types. This way we are able to obtain the solution for all these different values for the same initial condition during a single execution of our code.

3.3 Using R to Analyze Results

R is a programming language used for statistical computing and graphing [11]. In this project we wrote R scripts to read in the data generated by the C code and analyze the results.

In the results section we provide plots of the ℓ^2-norm for various initial conditions. The ℓ^2-norm of the double pendulum at some time t given by $\vec{v} = (\theta_1, w_1, \theta_2, w_2)$ is given by

$$||\vec{v}||^2 = \sqrt{|\theta_1|^2 + |w_1|^2 + |\theta_2|^2 + |w_2|^2}.$$

We calculate the ℓ_2-norm at each time step after executing the Runge-Kutta method in our C code. We then output the results to a file so we can import the data into RStudio and plot the evolution of these values over time. We choose to calculate the ℓ_2-norm in C so that we can endure high precision calculations and R does not support quad precision.

4 Results

We will be considering the initial condition $\vec{v_0} = (\theta_1, w_1, \theta_2, w_2) = (120°, 0, 120°, 0)$ which results in a Liapunov Exponent $\lambda > 2.5$ [2]. Hence, $\vec{v_0}$ produces highly chaotic results. We will also consider the initial conditions

$$\vec{v_1} = (121°, 0, 120°, 0)$$
$$\vec{v_2} = (120°, 1, 120°, 0)$$
$$\vec{v_3} = (120°, 0, 121°, 0)$$
$$\vec{v_4} = (120°, 0, 120°, 1)$$

where each $\vec{v_i}$ perturbs the value of one of the parameters given in $\vec{v_0}$. Through analyzing the differences in these solutions we will be able to determine which variable the system is most sensitive to change in.

For comparison, we will also consider an initial condition with the Liapunov Exponent $\lambda = 0$, $\vec{u_0} = (45°, 0, 45°, 0)$. This will allow us to see how chaotic solutions are more sensitive to rounding errors than non chaotic solutions are. Just as with $\vec{v_0}$, we will also consider initial conditions

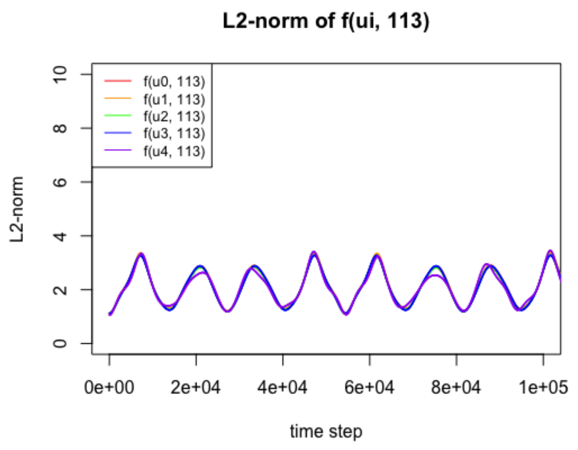
$$\vec{u_1} = (46°, 0, 45°, 0)$$
$$\vec{u_2} = (45°, 1, 45°, 0)$$
$$\vec{u_3} = (45°, 0, 46°, 0)$$
$$\vec{u_4} = (45°, 0, 45°, 1)$$

where each $\vec{u_i}$ perturbs a single parameter in $\vec{u_0}$ by 1.

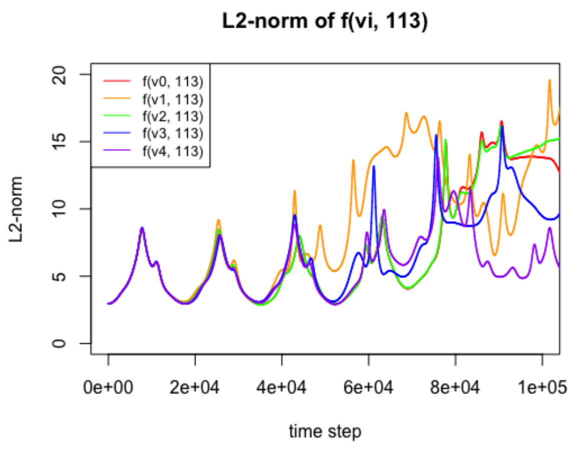
For ease of discussion we will refer to the long term solution of the double pendulum with initial condition $\vec{v_i}$ calculated using n-bits as $f(\vec{v_i}, n)$. For each $\vec{v_i}$ and $\vec{u_i}$ that we have defined we will run the solutions for all significand values of $n \in \{11, 24, 53, 64, 113\}$.

4.1 Initial Conditions: Chaotic vs. Non-Chaotic

First we will consider the differences in the trajectories generated using initial conditions that lead to chaotic behavior, $\vec{v_i}$, and initial conditions that do not, $\vec{u_i}$. Consider Fig. 4. In (a) we see how the ℓ^2 norm of $f(u_i, 113)$ for each $\vec{u_i}$ evolves over time. We observe how the values oscillate and remain close together even though the parameter values in each $\vec{u_i}$ vary slightly. Clearly, the initial conditions given by each $\vec{u_i}$ do not lead to chaotic behavior. However, in (b) the ℓ^2 norm of $f(v_i, 113)$ for each $\vec{v_i}$ diverges onto its own trajectory. Fig. 4 provides a clear depiction of how some initial conditions cause the double pendulum to behave chaotically and some do not.



(a) Evolution of the ℓ^2 norm of $f(u_i, 113)$ for each $i \in \{0, 1, 2, 3, 4\}$ over time.



(b) Evolution of the ℓ^2 norm of $f(v_i, 113)$ for each $i \in \{0, 1, 2, 3, 4\}$ over time.

Figure 4: Evolution of the ℓ^2-norm for chaotic and non-chaotic initial conditions.

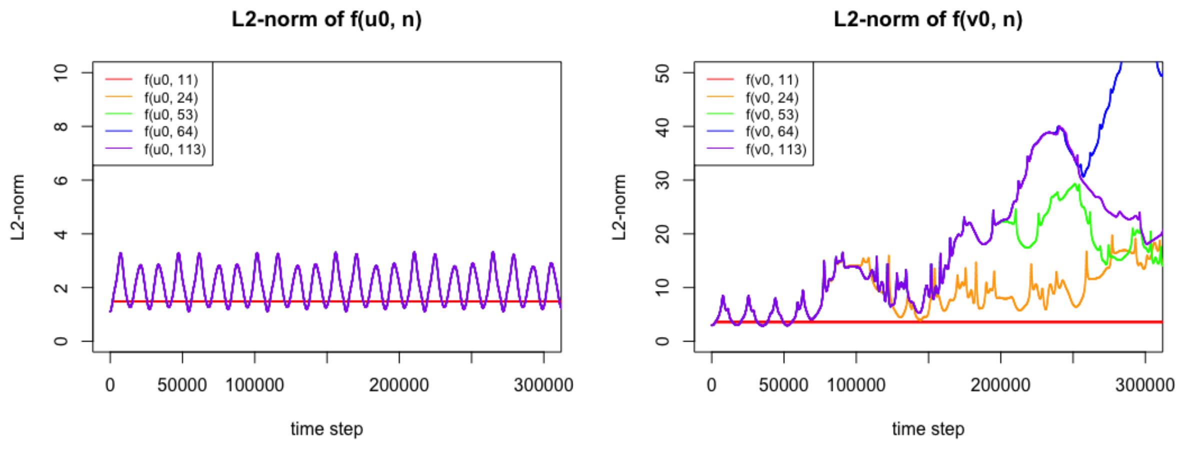
The solutions for $f(u_i, 113)$ and $f(v_i, 113)$ were calculate under the same conditions. Since the only difference was the initial condition given at the start of the simulation we know that the erratic behavior in (b) is intrinsic of the double pendulum's tendency to behave chaotically for high energy values and not a result of error induced by the numerical method used to approximate the solutions.

4.2 Word Size: Chaotic vs. Non-Chaotic

The system clearly exhibits sensitivity to small perturbations in the initial values of θ_1, w_1, θ_2, and w_2. Recall that when the computer calculates the solution at each time step, rounding errors occur as a result of the word size of the number and the operations being performed on that number. These rounding errors perturb the trajectory at each time step. This perturbed trajectory is used to calculate the next trajectory. We want to determine how these rounding errors affect the next approximated trajectories of the double pendulum over a long period of time.

Consider Fig. 5 where we plot the evolution of the ℓ^2-norm for $f(u_0, n)$ and $f(v_0, n)$ for $n \in \{11, 24, 53, 63, 113\}$. Both $f(u_0, n)$ and $f(v_0, n)$ were run under the exact same conditions where the only difference were the initial conditions. Notice how in (a) each n-bit solution generated with the initial condition u_0 remains the same over time while each n-bit solutions generated with the v_0 initial condition diverges early on in the simulation. From this we can conclude that the effects of finite precision have a greater influence on the accuracy of chaotic simulations than they do on non-chaotic simulations.

One point of interest in Fig. 5 is the results obtained for $f(u_0, 11)$ and $f(v_0, 11)$. For both of these



(a) Evolution of the ℓ^2 norm of $f(u_0, n)$ for each $n \in \{11, 24, 53, 64, 113\}$ over time.

(b) Evolution of the ℓ^2 norm of $f(v_0, n)$ for each $n \in \{11, 24, 53, 64, 113\}$ over time.

Figure 5: Graph of ℓ^2-norm over time for chaotic and non-chaotic initial conditions calculated using varying word size.

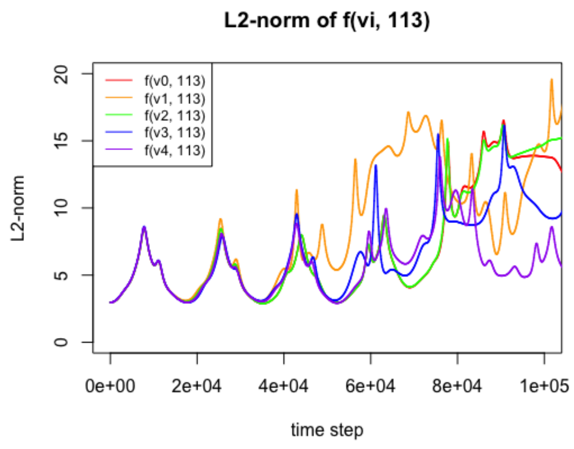
solutions we observe that the l^2-norm is constant. This is a result of the step size used in to calculate the trajectories. As mentioned, our step size was chosen to be $h = 10^{-4}$ based on results found in [2]. This step size results in underflow error when using 11-bit precision which is why we obtain a constant result. When we calculate the solution under the same conditions but let $h = 10^{-3}$, we obtain values for the L^2-norm that fluctuate over time. We have chosen to stick with the step size of $h = 10^{-4}$ because it gives more accurate results for the other solutions and because the trajectories for $f(u_0, 11)$ and $f(v_0, 11)$ diverge after the first few iterations when $h = 10^{-3}$. Thus, for either step size the results generated using 11-bit precision are uninteresting because they diverge immediately. We choose to leave them in our graphs for reference and because it may be of interest to future researchers.

4.3 Word Size: Time to Diverge

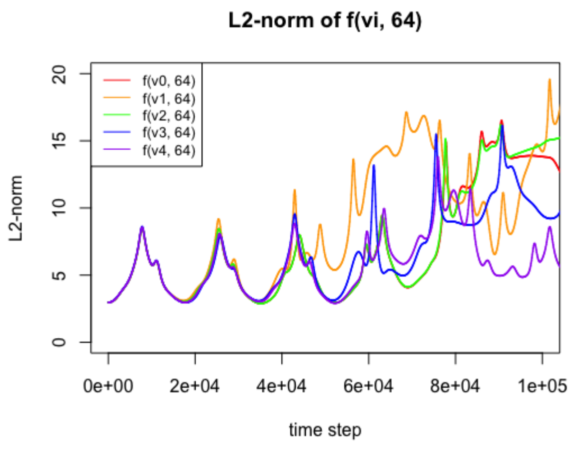
Referring back to (b) in Fig. 4 we can make observations regarding which variables the system is most sensitive to perturbations. In (b) we see that $f(v_2, 113)$ remains close to $f(v_0, 113)$ for a longer period of time than $f(v_4, 113)$ does. From this we can claim that the system expresses more sensitivity to perturbations in w_2 than it does in w_1. Similarly, we observe that the system is more sensitive to perturbations in θ_1 than θ_2.

Clearly, in Fig. 4 $f(v_1, 113)$ diverges from the solution generated using v_0 first. The next solution to diverge is $f(v_3, 113)$ followed by $f(v_4, 113)$ and then $f(v_2, 113)$. Thus, the system is most sensitive to perturbations in the values for θ_1, θ_2, w_2, and then w_1, in that order.

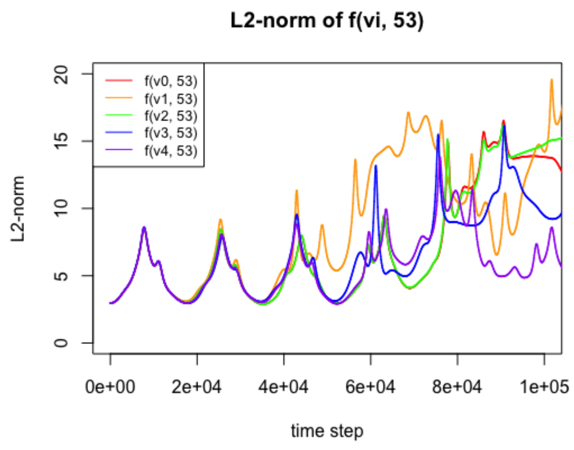
We want to determine if this parameter sensitivity is the same for all values of n. That is, we want to see $f(v_1, n)$ is always the first to diverge; followed by $f(v_3, 113)$, $f(v_4, 113)$ and then $f(v_2, 113)$. Consider Fig. 6. Here we see that no matter the size of n the solutions diverge from $f(v_0, n)$ at relatively the same time. Thus, when given the starting point v_0, word size does not have an impact on the time it takes the nearby trajectories v_1, v_2, v_3, and u_4 to diverge. In fact, all the graphs are relatively the same with the exception of (d) which begins to differ after 60000 iterations. The system does not begin to exhibit highly chaotic behavior until after 40000 iterations. In the next section we will examine each v_i for all our values of n and determine when rounding error begins to impact the solution.



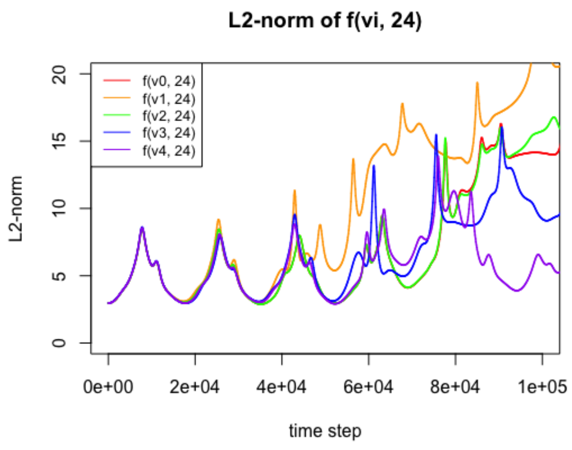
(a) Evolution of the ℓ^2 norm of $f(u_i, 113)$ for each $i \in \{0, 1, 2, 3, 4\}$ over time.



(b) Evolution of the ℓ^2-norm of $f(v_i, 64)$ for each $i \in \{0, 1, 2, 3, 4\}$ over time.



(c) Evolution of the ℓ^2-norm of $f(u_i, 53)$ for each $i \in \{0, 1, 2, 3, 4\}$ over time.



(d) Evolution of the ℓ^2-norm of $f(v_i, 24)$ for each $i \in \{0, 1, 2, 3, 4\}$ over time.

Figure 6: Evolution of the ℓ^2-norm for chaotic initial condition for various word sizes.

4.4 Rounding Error

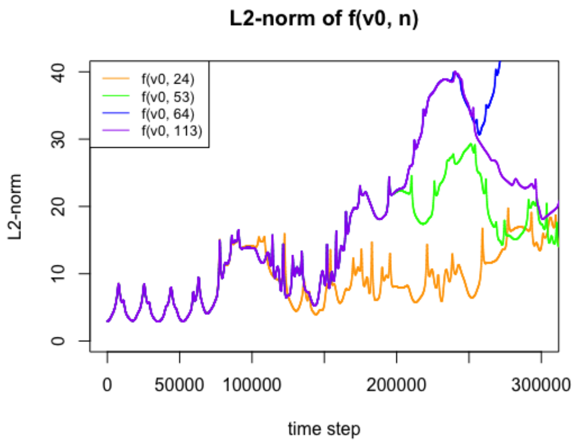
Consider Fig. 7. Each graph shows the multiple long term trajectories given a particular initial condition, v_i, each calculated using a different significand size. As stated previously, the system does not begin to exhibit chaotic behavior until after 40,000 iterations. In Fig. 7 we observe that none of the trajectories where $n = 24, 53, 64$ diverge from $f(v_i, 113)$ until after the $40000th$ iteration. This verifies our claim that rounding errors do not impact the accuracy of a solution unless the pendulum is exhibiting chaotic behavior.

In Fig. 7 we also observe that $f(v_i, 24)$ diverges from $f(v_i, 113)$ first followed by $f(v_i, 53)$ and then by $f(v_i, 64)$ for each v_i. This is consistent with what one would predict; higher precision achieves accurate results for a longer period of time.

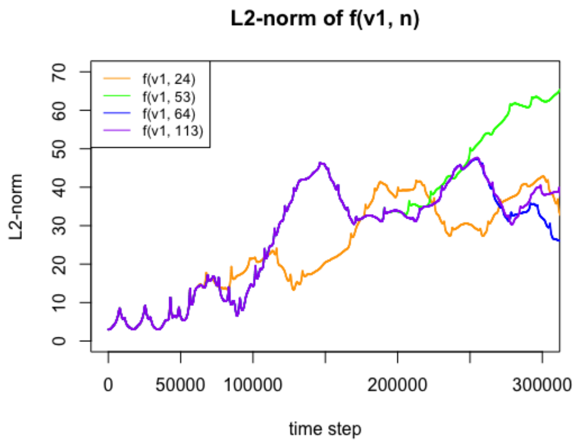
4.5 Changing the Step Size

Upon recommendation by my readers I computed $f(v_0, n)$ using a new step size $h = 2^{-13} \approx 0.000122$ as opposed to $h = 10^{-4} = 0.0001$. While similar in value, the significant difference between these two choices of h is that 2^{-13} can be exactly represented in floating point notation while 10^{-4} cannot. In Fig. 8 we observe the impact changing h has on the time it takes for each n-bit solution to diverge from the 113-bit solution. Recall that in Fig. 5 (b) the n-bit solutions all diverge from the 113-bit solution by the 250,000th iteration. In Fig. 8 however they do not all diverge until after the 400,000th iteration. We observe that when $h = 2^{-13}$ is used as the step size it takes the 53-bit and 64-bit solutions longer to diverge from the 113-bit solution.

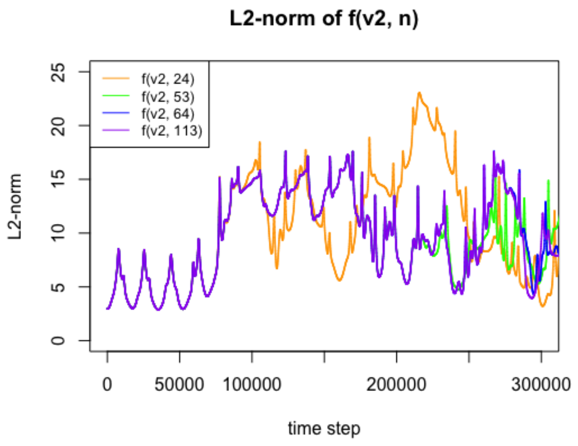
These results are interesting because of how much these solutions vary with regard to step size. When we execute the Runge-Kutta method and multiply by $h = 2^{-13}$ we are simply shifting the bits to the right. No rounding error occurs as a result of this. However, when we multiply by $h = 10^{-4}$ the computer preforms binary multiplication which can result in rounding. Here we see how by eliminating the rounding error that occurs at each iteration we obtain results that are extremely close to the 113-bit solution for a longer period of time.



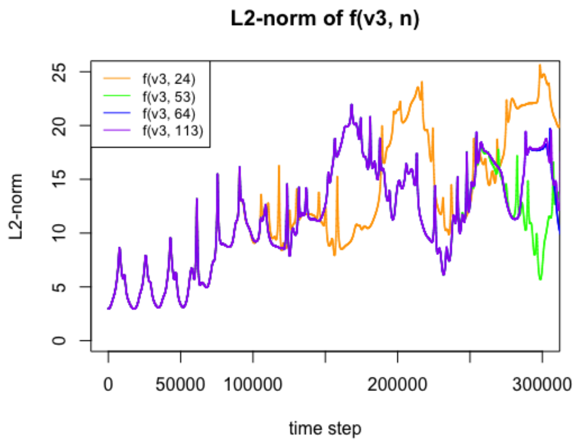
(a) Evolution of the ℓ^2-norm of $f(v_0, n)$ for each $n \in \{11, 24, 53, 64, 113\}$ over time.



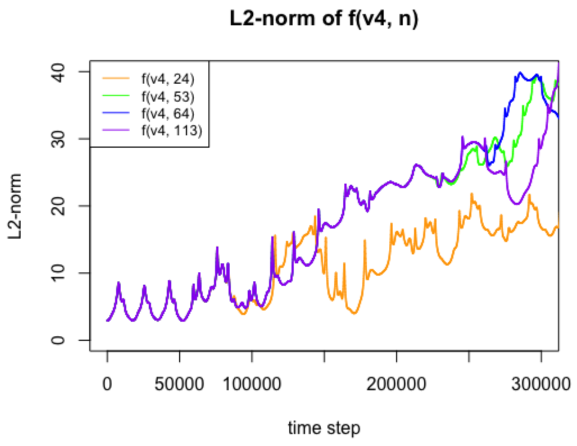
(b) Evolution of the ℓ^2-norm of $f(v_1, n)$ for each $n \in \{11, 24, 53, 64, 113\}$ over time.



(c) Evolution of the ℓ^2-norm of $f(v_2, n)$ for each $n \in \{11, 24, 53, 64, 113\}$ over time.



(d) Evolution of the ell^2-norm of $f(v_3, n)$ for each $n \in \{11, 24, 53, 64, 113\}$ over time.



(e) Evolution of the ℓ^2-norm of $f(v_4, n)$ for each $n \in \{11, 24, 53, 64, 113\}$ over time.

Figure 7: Evolution of the ℓ^2-norm for perturbed initial conditions.

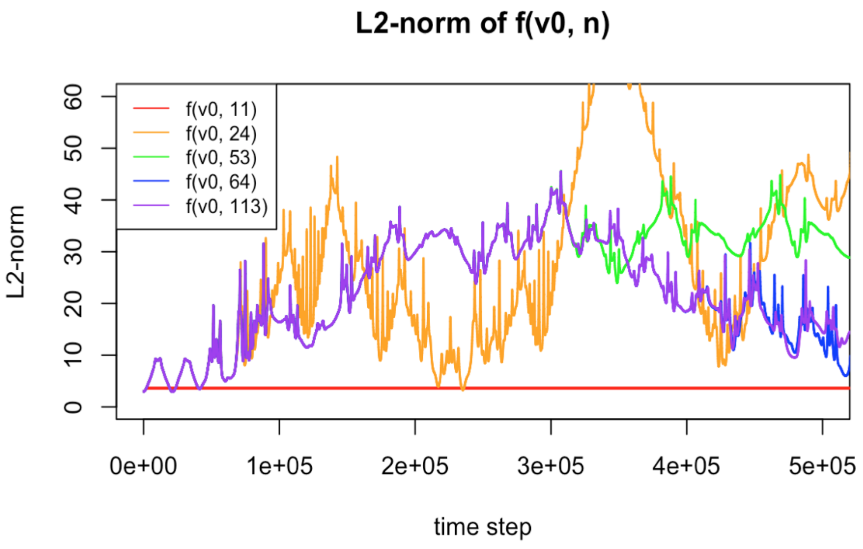


Figure 8: Evolution of the ℓ^2-norm for $f(v_0, n)$ where $h = 2^{-13}$.

5 Future Work

The next step in this research would be to numerically calculate the Liapunov Exponent for various initial conditions. In this project, we obtained the exponent values for our initial conditions based on results found in [2]. If we were able to reproduce these results, we could then do further analysis on how finite precision effects the chaos in a system. Do the solutions diverge faster given an initial condition with a greater exponent value? If we have multiple initial conditions that all generate approximately the same Liapunov Exponent, will the rate of divergence of each n-bit solution be the same for all of them?

In order to calculate the exponent value we will have to use the Gram-Schmidt process so that in each iteration we can normalize the vectors before calculating the next step. By comparing how much these vectors become skewed after an iteration and summing this value we can find the exponent. A detailed description of an algorithm that calculates the Liapunov Exponent utilizing the Gram-Schmidt process can be found in [2].

Many methods can be used to enhance the algorithms used in these simulations to reduce error. For example, an adaptive Runge-Kutta method described in [12] can be used to reduce error by shadowing the value, measuring the difference between them and comparing them to some user defined error tolerance, usually on the order of the machine epsilon for the precision being used. If the error is greater than what is desired, then you convert to higher precision, perform the calculation, and then convert back to single precision. Then the single precision error occurs in only one operation rather than 2 or more. This also saves resources meaning we could obtain faster results. This adaptive method is similar to the inner product method discussed in [5]. It could be of interest for future research to compare our results with the results obtained from using one or more of these methods. We could also try other methods besides Runge-Kutta such as the Parker-Sochacki method [13] which is an extension of the Picard iteration.

It could also be of interest to increase the precision of just the angles and then of just the angular velocities and see what kind of results are obtained and how they differ. Much more numerical work is required to integrate the angles so floating point rounding errors would have greater impact. It would be good to test this with non-chaotic initial conditions first (since the real trajectory is known). Then one could see if it is the same for chaotic conditions.

Currently, the mass and length of both pendula are set to 1. It could be interesting to see how the dynamics of the system changes with respect to these parameters. We could change the attributes of the pendulum to have different lengths and masses and see if any of the results obtained in this thesis hold in general for these different versions of the pendulum.

Currently the solution is restricted to the interval [-2,2]. Floating point values are very dense in this region. What would happen if we changed this interval to one where floating point values are not as dense? Given a particular initial condition that is known to become periodic after n iterations in the interval [-2,2], will it take fewer iterations to become periodic if we change the interval?

6 Conclusion

In this thesis we sought to gain insight on how finite precision affects long term solutions to the double pendulum. We have shown that the word size chosen to calculate long term solutions to the double pendulum has a more significant impact on the accuracy of chaotic solutions than non-chaotic solutions. We have also shown that the effects of rounding error do not impact the accuracy of the solution until after the system is known to enter a chaotic state.

We have also observed that the double pendulum is most sensitive to perturbations in θ_1 and least sensitive to perturbations in w_1 for the initial condition analyzed in this thesis. Our results show that this sensitivity is not influenced by the word size used in the calculation.

7 Appendix

Listing 1: dpend.h

```
#include <stdio.h>
#include <gmp.h>
#include<mpfr.h>



#define N 4     /* number of equations to be solved */
#define G 9.8   /* gravity (m/s^2) */
#define L1 1.0 /* length of pendulum 1 (m) */
#define L2 1.0 /* length of pendulum 2 (m) */
#define M1 1.0 /* mass of pendulum 1 (kg) */
#define M2 1.0 /* mass of pendulum 2 (kg) */

int nbits;  /* number of bits to use for mantissa */
double h;  /* step size */

typedef struct {
  mpfr_t th1;      /* angle of pend 1 */
  mpfr_t w1;       /* angular velocity of pend 1 */
  mpfr_t th2;      /* angle of pend 2 */
  mpfr_t w2;       /* angular velocity of pend 2 */
} y_t;

char dir_name[30];
int dir_count;

FILE *polar_output;
FILE *cartesian_output;
FILE *mag_output;

FILE *all_ics;
```

Listing 2: dpend_main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <gmp.h>
#include <mpfr.h>
#include "dpend.h"
#include "dpend_out.h"
#include "dpend_math.h"
#include <dirent.h>
#include <sys/stat.h>
#include <errno.h>
#include <math.h>

/*
 * dpend_main.c - Arbitrary precision solution to double pendulum ODEs
 *       using fourth order Runge-Kutta.
 *
 * Parameters are passed in at the command line:
 *
 * $./dpend_main.c TH10 W10 TH20 W20 NSTEP H
 *
 * TH10, TH20 - initial angles of the pendulums (degrees)
 * W10, W20 - initial angular velocities of the pendulums (degrees/sec)
 * NSTEP - number of times steps
 * H - step size
 */
int main(int argc, char *argv[])
{
  int NSTEP = atoi(argv[5]);
  int mantissa_sz[5] = {113,11,24,53,64};
  h = atoi(argv[6]);

  create_output_directory();
  output_initial_conditions(argv);
```

```
y_t * y_actual;
y_actual = malloc(NSTEP*sizeof(y_t));

/* Preform RK4 method to solve the system for each mantissa size. */
for (int j = 0; j < 5; j++) {
  mpfr_t t_curr, t_next;
  mpfr_t mag;
  y_t yin, yout;

  nbits = mantissa_sz[j];

  /* Create constant for converting angles to radians. */
  mpfr_t radian_conv;
  mpfr_init2(radian_conv, nbits);
  mpfr_set_d(radian_conv, 3.14159, MPFR_RNDN);
  mpfr_div_ui(radian_conv, radian_conv, 180, MPFR_RNDN);

  mpfr_inits2(113, mag, dot, r_error, NULL);
  mpfr_set_d(r_error, 0.0, MPFR_RNDN);

  mpfr_inits2(nbits, yin.th1, yin.w1, yin.th2, yin.w2,
              yout.th1, yout.w1, yout.th2, yout.w2, NULL);

  /* Create output files to store results. */
  create_out_files(mantissa_sz[j]);

  /* Initilize and set time values. */
  mpfr_inits2(113, t_curr, t_next, NULL);
  mpfr_set_d(t_curr, 0.0, MPFR_RNDN);
  mpfr_set_d(t_next, 0.0, MPFR_RNDN);

  /* Set initial values converting angles to radians. */
  mpfr_mul_d(yin.th1, radian_conv, atof(argv[1]), MPFR_RNDN);
  mpfr_mul_d(yin.w1, radian_conv, atof(argv[2]) , MPFR_RNDN);
mpfr_mul_d(yin.th2, radian_conv, atof(argv[3]), MPFR_RNDN);
mpfr_mul_d(yin.w2, radian_conv, atof(argv[4]) , MPFR_RNDN);

magnitude(&yin, &mag);
if (nbits != 113) {
  output_mag(&t_curr, &mag);
} else {
  y_actual[0] = yin;
  output_mag(&t_curr, &mag);
}

/* Output initial values. */
output_polar(&t_curr, &yin);
output_cartesian(&t_curr, &yin);

printf("Computing␣%d-bit␣result...␣", nbits); fflush(stdout);
/* Perform the integration. */
for (int i = 1; i < NSTEP; i++) {
  mpfr_add_d(t_next, t_curr, h, MPFR_RNDN);
  runge_kutta(t_curr, &yin, &yout);

  /* Print output to files. */
  output_polar(&t_next, &yout);
  output_cartesian(&t_next, &yout);

  /* Set yin to yout. */
  mpfr_set(yin.th1, yout.th1, MPFR_RNDN);
  mpfr_set(yin.w1, yout.w1, MPFR_RNDN);
  mpfr_set(yin.th2, yout.th2, MPFR_RNDN);
  mpfr_set(yin.w2, yout.w2, MPFR_RNDN);
  mpfr_set(t_curr, t_next, MPFR_RNDN);

  if (nbits == 113) { y_actual[i] = yin; }

  magnitude(&yout, &mag);
```

```
      output_mag(&t_curr, &mag);
    }
    printf("Done.\n");

    /* Clean up. */
    mpfr_clears(radian_conv, NULL);
    mpfr_clears(mag, dot, t_curr, t_next, yin.th1, yin.w1, yin.th2,
                yin.w2,yout.th1, yout.w1, yout.th2, yout.w2, NULL);
    mpfr_free_cache();

    /* Close files. */
    fclose(polar_output);
    fclose(cartesian_output);
    fclose(mag_output);
  }

  free(y_actual);

  return 0;
}
```

Listing 3: dpend_math.h

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <gmp.h>
#include <mpfr.h>

void runge_kutta(mpfr_t t, y_t *yin, y_t *yout);
void derivs(y_t *yin, y_t *dydx);

void magnitude (y_t *y, mpfr_t *magnitude);
```

Listing 4: dpend_math.c

```
#include "dpend.h"
#include <stdio.h>
#include <stdlib.h>

/*
 * Calculates the derivitive of each component in yin.
 * Result is stored in dydx.
 */
void derivs(y_t *yin, y_t *dydx) {

  /* Declare variables. */
  mpfr_t num1, den1;          // numerator and denomenator of w1'
  mpfr_t num2, den2;          // numerator and denomenator of w2'
  mpfr_t mass_sum;            // sum of pendulum masses, M1 + M2
  mpfr_t del;                 // difference of pend angles, th2 - th1
  mpfr_t cos_del, sin_del;    // cos(del), sin(del)
  mpfr_t sin_th1, sin_th2;    // sin(th1), sin(th2)
  mpfr_t sin_cos_del;         // sin(del) * cos(del)
  mpfr_t w1_sqr, w2_sqr;      // w1^2, w2^2
  mpfr_t aux;                 // auxilary variable

  /* Initilize variable precision. */
  mpfr_inits2(nbits, mass_sum, w1_sqr, w2_sqr, num1,
              den1, num2, den2, aux, del, cos_del,
              sin_del, sin_cos_del, sin_th1, sin_th2, NULL);

  /* Initilize variables. */
  mpfr_set_d(mass_sum, M1 + M2, MPFR_RNDN);
  mpfr_sub(del, yin->th2, yin->th1, MPFR_RNDN);

  mpfr_sin_cos(sin_del, cos_del, del, MPFR_RNDN);
  mpfr_mul(sin_cos_del, sin_del, cos_del, MPFR_RNDN);
  mpfr_sin(sin_th1, yin->th1, MPFR_RNDN);
  mpfr_sin(sin_th2, yin->th2, MPFR_RNDN);
```

```
mpfr_sqr(w1_sqr, yin->w1, MPFR_RNDN);
mpfr_sqr(w2_sqr, yin->w2, MPFR_RNDN);

/* Calculate th1' and th2'. */
mpfr_set(dydx->th1, yin->w1, MPFR_RNDN);
mpfr_set(dydx->th2, yin->w2, MPFR_RNDN);

/* Calculate w1'. */
// den1
mpfr_mul_d(den1, mass_sum, L1, MPFR_RNDN);
mpfr_set_d(aux, M2, MPFR_RNDN);
mpfr_mul_d(aux, aux, L1, MPFR_RNDN);
mpfr_mul(aux, aux, cos_del, MPFR_RNDN);
mpfr_mul(aux, aux, cos_del, MPFR_RNDN);
mpfr_sub(den1, den1, aux, MPFR_RNDN);

// num1
mpfr_mul_d(num1, w1_sqr, L1, MPFR_RNDN);
mpfr_mul(num1, num1, sin_cos_del, MPFR_RNDN);
mpfr_mul_d(aux, sin_th2, G, MPFR_RNDN);
mpfr_mul(aux, aux, cos_del, MPFR_RNDN);
mpfr_add(num1, num1, aux, MPFR_RNDN);
mpfr_mul_d(aux, w2_sqr, L2, MPFR_RNDN);
mpfr_mul(aux, aux, sin_del, MPFR_RNDN);
mpfr_add(num1, num1, aux, MPFR_RNDN);
mpfr_mul_d(num1, num1, M2, MPFR_RNDN);
mpfr_mul_d(aux, mass_sum, G, MPFR_RNDN);
mpfr_mul(aux, aux, sin_th1, MPFR_RNDN);
mpfr_sub(num1, num1, aux, MPFR_RNDN);

mpfr_div(dydx->w1, num1, den1, MPFR_RNDN);


/* Calculate w2'. */
```

```
  // den2
  mpfr_mul_d(den2, mass_sum, L2, MPFR_RNDN);
  mpfr_set_d(aux, M2, MPFR_RNDN);
  mpfr_mul_d(aux, aux, L2, MPFR_RNDN);
  mpfr_mul(aux, aux, cos_del, MPFR_RNDN);
  mpfr_mul(aux, aux, cos_del, MPFR_RNDN);
  mpfr_sub(den2, den2, aux, MPFR_RNDN);

  // num2
  mpfr_mul_d(num2, sin_th1, G, MPFR_RNDN);
  mpfr_mul(num2, num2, cos_del, MPFR_RNDN);
  mpfr_mul_d(aux, w1_sqr, L1, MPFR_RNDN);
  mpfr_mul(aux, aux, sin_del, MPFR_RNDN);
  mpfr_sub(num2, num2, aux, MPFR_RNDN);
  mpfr_mul_d(aux, sin_th2, G, MPFR_RNDN);
  mpfr_sub(num2, num2, aux, MPFR_RNDN);
  mpfr_mul(num2, num2, mass_sum, MPFR_RNDN);
  mpfr_set_d(aux, M2 * L2, MPFR_RNDN);
  mpfr_mul(aux, aux, w2_sqr, MPFR_RNDN);
  mpfr_mul(aux, aux, sin_cos_del, MPFR_RNDN);
  mpfr_sub(num2, num2, aux, MPFR_RNDN);

  mpfr_div(dydx->w2, num2, den2, MPFR_RNDN);

  /*  Clean up. */
  mpfr_clears(num1, num2, den1, den2, aux, mass_sum,
              del, cos_del, sin_del, sin_cos_del, sin_th1,
              sin_th2, w1_sqr, w2_sqr, NULL);
  mpfr_free_cache();
}

/*
 * Preforms runge-kutta method to integrate yin at time t.
 * Result is stored in yout.
 */
```

```
void runge_kutta(mpfr_t t, y_t *yin, y_t *yout)
{

  y_t dydx, dydxt, yt, k1, k2, k3, k4;
  mpfr_t aux;

  // Initilize y_t struct content
  mpfr_inits2(nbits, dydx.th1, dydx.w1, dydx.th2, dydx.w2, NULL);
  mpfr_inits2(nbits, dydxt.th1, dydxt.w1, dydxt.th2, dydxt.w2, NULL);
  mpfr_inits2(nbits, yt.th1, yt.w1, yt.th2, yt.w2, NULL);
  mpfr_inits2(nbits, k1.th1, k1.w1, k1.th2, k1.w2, NULL);
  mpfr_inits2(nbits, k2.th1, k2.w1, k2.th2, k2.w2, NULL);
  mpfr_inits2(nbits, k3.th1, k3.w1, k3.th2, k3.w2, NULL);
  mpfr_inits2(nbits, k4.th1, k4.w1, k4.th2, k4.w2, NULL);

  // Initilize temps
  mpfr_init2(aux, nbits);

  /* First step. */
  derivs(yin, &dydx);

  mpfr_mul_d(k1.th1, dydx.th1, h, MPFR_RNDN);
  mpfr_mul_d(yt.th1, k1.th1, 0.5, MPFR_RNDN);
  mpfr_add(yt.th1, yt.th1, yin->th1, MPFR_RNDN);

  mpfr_mul_d(k1.w1, dydx.w1, h, MPFR_RNDN);
  mpfr_mul_d(yt.w1, k1.w1, 0.5, MPFR_RNDN);
  mpfr_add(yt.w1, yt.w1, yin->w1, MPFR_RNDN);

  mpfr_mul_d(k1.th2, dydx.th2, h, MPFR_RNDN);
  mpfr_mul_d(yt.th2, k1.th2, 0.5, MPFR_RNDN);
  mpfr_add(yt.th2, yt.th2, yin->th2, MPFR_RNDN);

  mpfr_mul_d(k1.w2, dydx.w2, h, MPFR_RNDN);
  mpfr_mul_d(yt.w2, k1.w2, 0.5, MPFR_RNDN);
```

```
mpfr_add(yt.w2, yt.w2, yin->w2, MPFR_RNDN);


/* Second step. */
derivs(&yt, &dydxt);

mpfr_mul_d(k2.th1, dydxt.th1, h, MPFR_RNDN);
mpfr_mul_d(yt.th1, k2.th1, 0.5, MPFR_RNDN);
mpfr_add(yt.th1, yt.th1, yin->th1, MPFR_RNDN);

mpfr_mul_d(k2.w1, dydxt.w1, h, MPFR_RNDN);
mpfr_mul_d(yt.w1, k2.w1, 0.5, MPFR_RNDN);
mpfr_add(yt.w1, yt.w1, yin->w1, MPFR_RNDN);

mpfr_mul_d(k2.th2, dydxt.th2, h, MPFR_RNDN);
mpfr_mul_d(yt.th2, k2.th2, 0.5, MPFR_RNDN);
mpfr_add(yt.th2, yt.th2, yin->th2, MPFR_RNDN);

mpfr_mul_d(k2.w2, dydxt.w2, h, MPFR_RNDN);
mpfr_mul_d(yt.w2, k2.w2, 0.5, MPFR_RNDN);
mpfr_add(yt.w2, yt.w2, yin->w2, MPFR_RNDN);


/* Third step. */
derivs(&yt, &dydxt);

mpfr_mul_d(k3.th1, dydxt.th1, h, MPFR_RNDN);
mpfr_add(yt.th1, k3.th1, yin->th1, MPFR_RNDN);

mpfr_mul_d(k3.w1, dydxt.w1, h, MPFR_RNDN);
mpfr_add(yt.w1, k3.w1, yin->w1, MPFR_RNDN);

mpfr_mul_d(k3.th2, dydxt.th2, h, MPFR_RNDN);
mpfr_add(yt.th2, k3.th2, yin->th2, MPFR_RNDN);
```

```
mpfr_mul_d(k3.w2, dydxt.w2, h, MPFR_RNDN);
mpfr_add(yt.w2, k3.w2, yin->w2, MPFR_RNDN);



/* Fourth step. */
derivs(&yt, &dydxt);

mpfr_mul_d(k4.th1, dydxt.th1, h, MPFR_RNDN);

mpfr_set(yout->th1, yin->th1, MPFR_RNDN);
mpfr_add(aux, k1.th1, k4.th1, MPFR_RNDN);
mpfr_div_d(aux, aux, 6.0, MPFR_RNDN);
mpfr_add(yout->th1, yout->th1, aux, MPFR_RNDN);
mpfr_add(aux, k2.th1, k3.th1, MPFR_RNDN);
mpfr_div_d(aux, aux, 3.0, MPFR_RNDN);



mpfr_mul_d(k4.w1, dydxt.w1, h, MPFR_RNDN);

mpfr_set(yout->w1, yin->w1, MPFR_RNDN);
mpfr_add(aux, k1.w1, k4.w1, MPFR_RNDN);
mpfr_div_d(aux, aux, 6.0, MPFR_RNDN);
mpfr_add(yout->w1, yout->w1, aux, MPFR_RNDN);
mpfr_add(aux, k2.w1, k3.w1, MPFR_RNDN);
mpfr_div_d(aux, aux, 3.0, MPFR_RNDN);
mpfr_add(yout->w1, yout->w1, aux, MPFR_RNDN);



mpfr_mul_d(k4.th2, dydxt.th2, h, MPFR_RNDN);

mpfr_set(yout->th2, yin->th2, MPFR_RNDN);
mpfr_add(aux, k1.th2, k4.th2, MPFR_RNDN);
mpfr_div_d(aux, aux, 6.0, MPFR_RNDN);
mpfr_add(yout->th2, yout->th2, aux, MPFR_RNDN);
mpfr_add(aux, k2.th2, k3.th2, MPFR_RNDN);
```

```
  mpfr_div_d(aux, aux, 3.0, MPFR_RNDN);
  mpfr_add(yout->th2, yout->th2, aux, MPFR_RNDN);


  mpfr_mul_d(k4.w2, dydxt.w2, h, MPFR_RNDN);

  mpfr_set(yout->w2, yin->w2, MPFR_RNDN);
  mpfr_add(aux, k1.w2, k4.w2, MPFR_RNDN);
  mpfr_div_d(aux, aux, 6.0, MPFR_RNDN);
  mpfr_add(yout->w2, yout->w2, aux, MPFR_RNDN);
  mpfr_add(aux, k2.w2, k3.w2, MPFR_RNDN);
  mpfr_div_d(aux, aux, 3.0, MPFR_RNDN);
  mpfr_add(yout->w2, yout->w2, aux, MPFR_RNDN);


 /* Clean up. */
  mpfr_clears(dydx.th1, dydx.w1, dydx.th2, dydx.w2,
              dydxt.th1, dydxt.w1, dydxt.th2, dydxt.w2,
              yt.th1, yt.w1, yt.th2, yt.w2,
              k1.th1, k1.w1, k1.th2, k1.w2,
              k2.th1, k2.w1, k2.th2, k2.w2,
              k3.th1, k3.w1, k3.th2, k3.w2,
              k4.th1, k4.w1, k4.th2, k4.w2,
              aux, NULL);
  mpfr_free_cache();
}

/*
 * Calculates the magnitude of y_t and stores it in magnitude.
 */
void magnitude (y_t *y, mpfr_t *magnitude) {
  mpfr_t aux;
  mpfr_init2(aux, 113);

  mpfr_set(*magnitude, y->th1, MPFR_RNDN);
  mpfr_sqr(*magnitude, *magnitude, MPFR_RNDN);
  mpfr_sqr(aux, y->w1, MPFR_RNDN);
  mpfr_add(*magnitude, *magnitude, aux, MPFR_RNDN);
  mpfr_sqr(aux, y->th2, MPFR_RNDN);
  mpfr_add(*magnitude, *magnitude, aux, MPFR_RNDN);
  mpfr_sqr(aux, y->w2, MPFR_RNDN);
  mpfr_add(*magnitude, *magnitude, aux, MPFR_RNDN);
  mpfr_sqrt(*magnitude, *magnitude, MPFR_RNDN);

  mpfr_clear(aux);
  mpfr_free_cache();
}
```

Listing 5: dpend_out.h

```
#include <stdio.h>
#include <stdlib.h>
#include <gmp.h>
#include <mpfr.h>

void output_polar(mpfr_t* t, y_t* y);
void output_cartesian(mpfr_t* t, y_t* y);
void output_mag(mpfr_t* t, mpfr_t *mag);

void create_out_files(int num);
void create_output_directory();
void output_initial_conditions(char* argv[]);
```

Listing 6: dpend_out.c

```c
#include "dpend.h"
#include <stdio.h>
#include <stdlib.h>
#include <gmp.h>
#include <mpfr.h>
#include <dirent.h>
#include <sys/stat.h>

/*
 * Used by dpend_mpfr.c to write out data.
 */

/* Record initial conditions given at program start. */
void output_initial_conditions(char* argv[]) {
  all_ics = fopen("./mpfr_data/all_ics.txt", "a");
  fprintf(all_ics, "ic_%d,%s,%s,%s,%s,%s,%f\n",
          dir_count, argv[1], argv[2], argv[3], argv[4], argv[5], h);
  fclose(all_ics);
}

/* Create folder to store all output files. */
void create_output_directory() {
  /* Count number of files in mpfr_data. */
  dir_count = 0;
  DIR *dirp;
  struct dirent *entry;

  dirp = opendir("./mpfr_data");
  while ((entry = readdir(dirp)) != NULL) {
    if (entry->d_type == DT_DIR) { /* If the entry is a regular file */
      dir_count++;
    }
  }
  closedir(dirp);
  /* Create ic_# directory to store output in. */
  snprintf(dir_name, 30, "./mpfr_data/ic_%d", dir_count);
  mkdir(dir_name, 0777);
}

/* Create all files needed to store output in. */
void create_out_files(int num) {
    /* Create output file for polar coordinates. */
    snprintf(polar_fn, 50, "%s/polar%d.csv", dir_name, num);
    polar_output = fopen(polar_fn, "w");
    fprintf(polar_output, "time,th1,w1,th2,w2\n");

    /* Create output file for cartesian coordinates. */
    snprintf(cartesian_fn, 50, "%s/cartesian%d.txt", dir_name, num);
    cartesian_output = fopen(cartesian_fn, "w");
    fprintf(cartesian_output, "time,x1,y1,x2,y2\n");

    /* Create output file for magnitude. */
    snprintf(mag_fn, 50, "%s/mag%d.csv", dir_name, num);
    mag_output = fopen(mag_fn, "w");
    fprintf(mag_output, "time,magnitude,dot_product,r_error\n");
}

/* Output the magnitude at time t. */
void output_mag(mpfr_t* t, mpfr_t *mag) {
  mpfr_fprintf(mag_output, "%0.32RNF,%0.32RNF\n",
               *t, *mag);
}

/* Output the polar coordinates at time t. */
void output_polar(mpfr_t* t, y_t* y) {
  mpfr_fprintf(polar_output, "%0.32RNf,%0.32RNf,%0.32RNF,%0.32RNF,
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣%0.32RNF\n", *t, y->th1, y->w1, y->th2, y->w2);
}
```

```
/* Output the cartesian coordinates at time t. */
void output_cartesian(mpfr_t* t, y_t* y) {

  /* Convert to cartesian coordinates. */
  mpfr_t x1, y1, x2, y2, temp;
  mpfr_inits2(nbits, x1, y1, x2, y2, temp, NULL);

  // Calculate x1
  mpfr_set_d(x1, L1, MPFR_RNDN);
  mpfr_sin(temp, y->th1, MPFR_RNDN);
  mpfr_mul(x1, x1, temp, MPFR_RNDN);

  // Calculate y1
  mpfr_set_d(y1, -L1, MPFR_RNDN);
  mpfr_cos(temp, y->th1, MPFR_RNDN);
  mpfr_mul(y1, y1, temp, MPFR_RNDN);

  // Calculate x2
  mpfr_set_d(x2, L2, MPFR_RNDN);
  mpfr_sin(temp, y->th2, MPFR_RNDN);
  mpfr_mul(x2, x2, temp, MPFR_RNDN);
  mpfr_add(x2, x2, x1, MPFR_RNDN);

  // Calculate y2
  mpfr_set_d(y2, -L2, MPFR_RNDN);
  mpfr_cos(temp, y->th2, MPFR_RNDN);
  mpfr_mul(y2, y2, temp, MPFR_RNDN);
  mpfr_add(y2, y2, y1, MPFR_RNDN);

  mpfr_fprintf(cartesian_output, "%0.32RNf,%0.32RNf,%0.32RNF,%0.32RNF,
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣%0.32RNF\n", *t, x1, y1, x2, y2);
}
```

References

[1] S. H. Strogatz, *Nonlinear Dynamics and Chaos: With Applications to Physics, Biology, Chemistry and Engineering.* Westview Press, 2000.

[2] A. M. Calvão and T. J. P. Penna, "The double pendulum: a numerical study," *European Journal of Physics*, vol. 36, p. 045018, July 2015.

[3] M. K. Gupta, K. Bansal, and A. K. Singh, "Mass and length dependent chaotic behavior of a double pendulum," *IFAC Proceedings Volumes*, vol. 47, no. 1, pp. 297 – 301, 2014. 3rd International Conference on Advances in Control and Optimization of Dynamical Systems (2014).

[4] E. W. Weisstein, "Double pendulum." Accessed: 2018-09-30.

[5] N. J. Higham, *Accuracy and Stability of Numerical Algorithms.* Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2nd ed., 2002.

[6] IEEE, "IEEE Standard for Floating-Point Arithmetic (IEEE 754-2008)," tech. rep., IEEE, New York, Aug 2008.

[7] P. Góra and A. Boyarsky, "Why computers like lebesgue measure," *Computers & Mathematics with Applications*, vol. 16, no. 4, pp. 321 – 329, 1988.

[8] R. Corless, C. Essex, and M. Nerenberg, "Numerical methods can suppress chaos," *Physics Letters A*, vol. 157, no. 1, pp. 27 – 36, 1991.

[9] Free Software Foundation, Inc., *GNU MPFR*, 4.0.2 ed., 2019.

[10] Free Software Foundation, Inc., *The GNU Awk User's Guide*, 4.2 ed., 2018.

[11] "The r project for statistical computing." Accessed: 2019-03-20.

[12] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C (2Nd Ed.): The Art of Scientific Computing.* New York, NY, USA: Cambridge University Press, 1992.

[13] G. E. Parker and J. S. Sochacki, "Implementing the picard iteration," *Neural, Parallel Sci. Comput.*, vol. 4, pp. 97–112, Mar. 1996.