Numerically Simulating the Solar System in *Mathematica*

Charles Chen
Northwood High School, Irvine, USA
Email: charlesbattlechen@gmail.com

***Abstract*—The planetary motion within our solar system is a topic that has been studied for hundreds of years and has given rise to the science of astronomy. It is very important to know the positions of the planets in our solar system, as many of our current scientific research depends on it. Space exploration, for example, is a perfect example of when we need to know the exact positions of the planets in our solar system. Since it takes many years to send a rover or satellite to a planet, we will need to be able to predict the position of that planet many years into the future. Therefore, I present a second order Runge-Kutta simulation to predict the future position and velocity of the planets in our solar system based on Newtonian laws of motion. The equations of motion are implemented into a Mathematia script which animates the motion of each planet by generating a single static plot at each iteration within the while loop, stepping forward in time, re-plotting overtop the previous frame. This step-by-step numerical simulation is typically overlooked as an animation technique available in Mathematica. I herein provide an introduction to the software, an intuitive comparison of numerical vs analytical solutions to differential equations, and finally present the results of the simulation.**

***Index Terms*—Mathematica, Newton's second law of motion, Runge-Katta, Universal Law of Gravitation**

I. Mathematica

Mathematica is a mathematical computation software that uses the Wolfram programming language, and it offers better symbolic manipulation than many other programming languages. Mathematica documents are called 'notebooks' and are organized into cells that can be individually evaluated. The Wolfram language has many built in functions, but it also allows the creation of custom functions. Functions make use of the fact that Wolfram is case-sensitive by using capital letters for built-in functions, which is why it is good practice to start custom functions and variables with lowercase letters. In order to define a custom function, $f[x\_]:=$ is used. Variables in Mathematica work like variables in any other programming language and are defined by using the 'set' assignment operator $var =$. Modules in Mathematica allows different programs to not conflict with each other by localizing the scope of variables used in the program, as the default scope for Mathematica variables is global across all notebooks. This works by assigning a serial number $\$snnn$ to the end of all variable names, making them unique.

Due to its user-friendly notebook format, Mathematica is commonly used as an elaborate graphing calculator and sometimes overlooked as a programming language for numerical simulations. Furthermore, Mathematica's selection of helpful built-in functions such as "Animate", "Manipulate", and "Dynamic", which allow the user to control input parameters (via a slide bar) for graphical plots of analytic functions, often cause users to overlook Mathematica's ability to animate numerical solutions in real time. The code presented in this paper shows a creative method to achieve step-by-step, real time animations by updating a particle's position in a while loop. To highlight the difference between analytical and numerical solutions, this paper begins with a discussion on differential equations, using the spring-mass system as an example. The spring-mass problem is solved both analytically and numerically to demonstrate the difference between the two approaches.

II. Differential Equations: Analytic VS Numerical Solutions

A differential equation is any equation that involves the derivative of a function, with the derivative being the instantaneous rate of change of a function. Take for example:

$$\frac{dy(x)}{dx} = 3x^5$$

Notice the solution to the above differential equation is itself a function, $y(x)$. This is true for all differential equations. Due to the time-derivative relationship between acceleration, velocity, and position, any physical system wherein the position is affected by the velocity or acceleration (i.e. essentially physical systems) will be described by a differential equation.

There are two approaches to solving differential equations, analytical and numerical. The difference between analytical and numerical solutions is that analytical solutions are more accurate, as their solutions are continuous functions, while numerical solutions are discrete functions. However, many real-life situations must be solved numerically, as their analytical solution either does not exist or is too difficult to solve for. A common real-world example where the analytical solution of a differential equation does not exist would

Manuscript received May 11, 2019; revised September 26, 2019.

the Blasius equation, used in fluid mechanics: $f''' + \frac{1}{2}ff'' = 0$ [1].

In order to demonstrate the two ways of solving differential equations, a simple spring-mass system will be used. Since a spring-mass system is much simpler than the solar system, it serves as an effective example to highlight the difference between the two methods.

The example system that will be used is a horizontal spring with a mass on the end of it, resting on a frictionless surface with no gravity involved. The goal is to solve for the spring's exact motion in time $x(t)$, given an initial position $x(t = 0) = x_0$ and an initial velocity $v(t = 0) = v_0$ (schematic shown in Fig. 1).

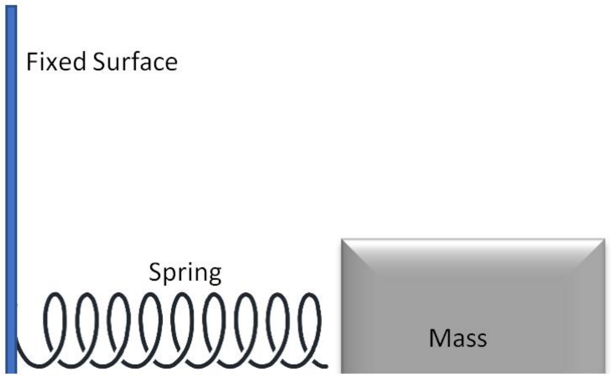

Figure 1. Schematic of a simple spring-mass system.

To solve for $x(t)$ analytically, it is important to note that velocity is the derivative of position as a function of time, and acceleration is the derivative of velocity as a function of time. Therefore, $v = \frac{dx}{dt}$ and $a = \frac{dv}{dt}$ [2]. Thus, acceleration is equal to the double derivative of position as a function of time, $a = \frac{d^2x}{dt^2}$. Newton's second law of motion states that $F = ma$, and Hooke's law states that $F = -kx$, where k is the force constant of a spring and x is the displacement of the spring from its equilibrium position. Setting the forces of the equations above equal to each other gives the equation $-kx = ma$. Setting acceleration to be the second derivative of position with respect to time, and dividing by m, gives the following differential equation:

$$-\frac{k}{m}x = \frac{d^2x}{dt^2} \tag{1}$$

Although it looks complicated, the equation simply means that the second derivative of x(t) is the same function, $x(t)$, multiplied by a constant, $-\frac{k}{m}$. With this new information, an ansatz that $x(t) = Ce^{-iAt}$ can be made. Plugging this ansatz into the different equation results in $-\frac{kCe^{-iAt}}{m} = \frac{d^2(Ce^{-iAt})}{dt^2}$. Taking the first derivative of Ce^{-iAT} gives $-iACe^{-iAt}$, and taking the second derivative of that gives CA^2e^{iAt}, which can then be substituted back into the equation, giving $CA^2e^{-iAt} = -\frac{ke^{-iAT}}{m}$. Diving both sides by $C\, e^{-iAt}$ gives the equation $A^2 = \frac{k}{m}$, which means:

$$x(t) = e^{-i\sqrt{\frac{k}{m}}t} \quad \rightarrow$$

$$x(t) = A\cos\sqrt{\frac{k}{m}}t - iB\sin\sqrt{\frac{k}{m}}t \tag{2}$$

here, Euler's identity has allowed to the exponential to be rewritten in terms of oscillating sine and cosine functions [3]. The unknown complex constants A and B are easily solved for when the initial conditions of the system are known. One can substitute $x(t) \rightarrow x_0$ and $t \rightarrow 0$, as well as $v(t) \rightarrow v_0$ and $t \rightarrow 0$ to yield two equations with two unknowns, solve for A and B, and obtain an exact solution for $x(t)$ at all times.

To highlight the difference between analytical and numerical solutions, the numerical solution to the same problem is now presented. Again, the goal is to find an unknown $x(t)$ given a set of initial conditions: $x(t = 0) = x_0$ and $v(t = 0) = 0$. In other words, "what is the motion of a mass after an initial perturbation?"

First, the horizontal axis (time) is broken into discrete segments, as shown above in Fig. 2, and each $x(t)$ must have some value at each point: $\{x_1, x_2, x_3 \ldots\}$. Next, it is assumed that the slope (i.e. derivative) between two neighboring points x_n and x_{n+1} is simply the slope of a line that connects the two points, these are shown as red lines in Fig. 2. The slope of a line is given by mnemonic "rise over run", or slope $\approx \frac{x_{n+1} - x_n}{t_{n+1} - t_n}$. Since velocity is the derivative (i.e. slope) of position with respect to time, the following equation is obtained:

$$\frac{x_{n+1} - x_n}{t_{n+1} - t_n} = v_n \tag{3}$$

Since acceleration is the derivative of velocity with respect to time:

$$\frac{v_{n+1} - v_n}{\Delta t} = a_n \tag{4}$$

where $\Delta t = t_{n+1} - t_n$ has been substituted in. From Newton's second law, the acceleration of the mass must be $-\frac{kx}{m}$:

$$\frac{v_{n+1} - v_n}{\Delta t} = -\frac{k}{m}x_n \tag{5}$$

Solving for x_{n+1} and v_{n+1} results in $x_{n+1} = v_n\Delta t + x_n$ and $v_{n+1} = -\frac{kx}{m}\Delta t + v_n$. These final two equations are well suited for computation. A while loop can be implemented to continuously calculate the new position and velocity $\{x_{n+1}, v_{n+1}\}$ using the previous position and velocity $\{x_n, v_n\}$.

As simple as Euler's method is, it isn't the most accurate, as one side of the equation is centered with time, and the other side isn't. For example, in $\frac{x_{n+1} - x_n}{\Delta t} = v_n$, the left side is centered halfway between t_{n+1} and t_n, while the right side is centered at t_n. Runge-Kutta is one solution to the issue, which attempts to solve the issue by centering both sides of the equation at the same t through a half step using Euler's method to get x and v at $t_{n+1/2}$, which can later be used to more accurately solve for the next step. Applying this to the spring-mass system results in $v_{half} = -\frac{kx}{m} * \Delta t_{half} + v_n$ and $x_{half} = v_n * \Delta t_{half} + x_n$, which are then used to solve for the next step with $x_{n+1} = v_{half} * \Delta t + x_n$ and $v_{n+1} = -\frac{kx_{half}}{m} * \Delta t + v_n$.

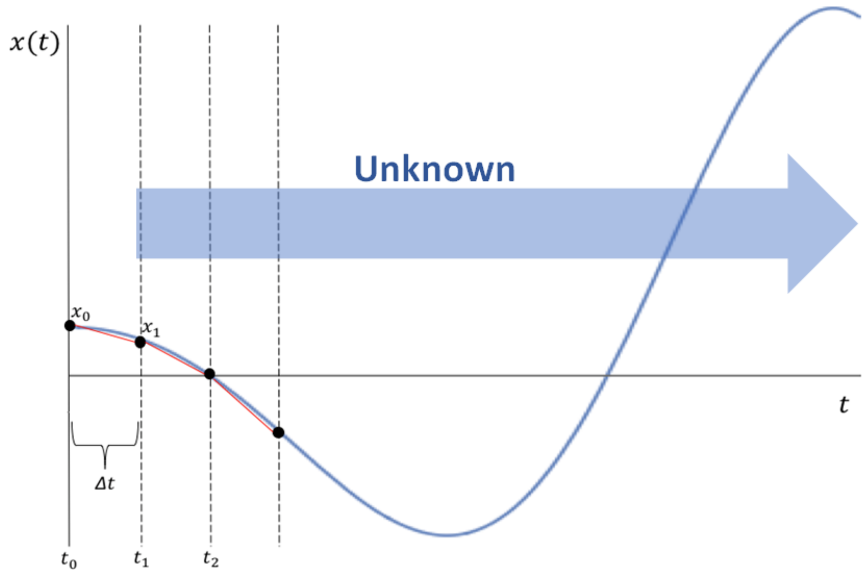

Figure 2. Graphical representation of Euler's method applied to find an unknown function x(t).

III. Numerical Simulation of the Solar System

Moving on to the solar system, the mathematics is very similar to the mathematics in the spring-mass system. As there are many factors to consider in a system with multiple items, the system will be solved for numerically. Starting with Newton's second law of motion $F = ma$, and Newton's universal law of gravitation $F = -G\frac{m_{planet}m_{sun}}{r^2}$, the two equations can be substituted, then solved for acceleration, giving the equation:

$$\vec{F} = -G\frac{m_{planet} * m_{sun}}{r^2}\hat{r} = m_{planet}\vec{a} \quad \Rightarrow \quad \vec{a} = -G\frac{m_{sun}}{r^2}\hat{r} \tag{6}$$

Noting that the mass of the planet cancels out. Using the fact that velocity is the derivative of position over time, the following formula is given: $\frac{x_{n+1}-x_n}{dt} = v_{xn}$, where x can be replaced by y to find the vertical vector. This equation can be solved for x_{n+1}, resulting in the equation:

$$x_{n+1} = v_x * dt + x_n$$

Now using the fact that acceleration is the derivative of velocity over time, gives the equation $-G\frac{m_{sun}}{r^2} = \frac{v_{n+1}+v_n}{dt}$, and solving for v_{n+1} gives the equation $v_{n+1} = G\frac{m_{sun}}{r^2} * dt + v_n$. However, as the simulation is two-dimensional, the equation for velocity needs to be split into v_x and v_y. To do this, the scalar a is multiplied by its unit vector, $\hat{a}$, to get vector $\vec{a}$. The unit vector is found by summing the x and y vectors of the planet, and dividing by the magnitude of $\vec{r}$, where $|r| = \sqrt{x^2+y^2}$. This means that $\hat{r} = \frac{-\hat{x}-\hat{y}}{\sqrt{x^2+y^2}}$. Together, this gives the equations:

$$v_{x+1} = G\frac{m_{sun}}{x^2+y^2} * \frac{-\hat{x}}{\sqrt{x^2+y^2}} * dt + v_x \tag{7}$$

Using Runge-Kutta, the half steps of the previous equations are then taken, giving the final equations:

$$p_{half} = v_p * dt_{half} + p_n \tag{8}$$

$$v_{phalf} = G\frac{m_{sun}}{x^2+y^2} * \frac{-\hat{p}}{\sqrt{x^2+y^2}} * dt_{half} + v_p \tag{9}$$

$$p_{n+1} = v_{phalf} * dt + p_n \tag{10}$$

$$v_{p+1} = G\frac{m_{sun}}{x_{half}^2+y_{half}^2} * \frac{-\hat{p}_{half}}{\sqrt{x^2+y^2}} * dt + v_p \tag{11}$$

where p can be replaced by either x or y depending on the component that is being found. These last four equations are good for computational purposes, so given the initial positions and velocities of planets, as well as universal gravitational constant G, the equations can predict the future positions the planets.

It should be noted that there are factors of consideration that have not been accounted for; however, these factors do not produce a significant impact on the results. One assumption that is made about planetary movement is that the planets move in two dimensions (snapshots of the animation are shown in Fig. 3). In reality, the solar system exists in three dimensions, but due to the common orbital angular moment of the gas cloud that evolved into the solar system, the planets rotate in a single 2-D plane to a good approximation. At most, the orbital inclinations of the celestial planets only differ by 7° with respect to on another, meaning very little accuracy is sacrificed in making this assumption [4]. A second assumption that is made is that the planets only feel a gravitational attraction to the sun. However, since the mass of the sun is far greater than any other object in our solar system, with the second most massive thing, Jupiter, being 1000 times less massive, a largest gravitational force a planet feels will be coming from the sun.

Before entering the initial values of all the planets into the code, the units of these values must be converted so Mathematica can run the simulation without having to deal with numbers that tens of digits long, thousands of times per second. To do this, all the distances were originally in meters, and were brought down by 10^9. Times, originally in units of seconds, were divided by 33554.5, in order to set the velocity of the Earth at unity. The units of mass were kept at kilograms, as only the mass of the sun is ever used. The gravitational constant (units of $m^3kg^{-1}s^{-2}$) was converted to a value of $G = 7.514 * 10^{-29}$ so as to incorporate the distance and time conversions used.

IV. Conclusion

Since the employed algorithm only used second-order Runge-Kutta and simplified the assumptions used (circular orbits in a two-dimensional plane), the presented simulation is less accurate than state of the art numerical simulations of the solar system, e.g. Runge-Kutta-Nystrom algorithms used by NASA [5]. However, the final script (attached at bottom of this paper), maintained impressive accuracy, matching true orbital patterns to within roughly ± 0.15 earth years for 40 earth years into the simulation. Fig. 3 shows the position of the planets from 0 to 100 years into the future. The overall accuracy of each planet is inversely proportional to its orbital period, meaning the planet with the shortest period, Mercury, is the first to deviate from its true orbit. If there exists even a slight discrepancy between the initial velocity and the initial position of the planet, then the planet will be thrown off axis very easily. This problem may be fixed if a higher order Runge-Kutta had been used. This simulation uses second-order Runge-Kutta, only taking the half step once, but a higher order, e.g. sixth-order Runge-Kutta could help even out the inaccuracies. Excluding Mercury, the other planets stay in orbit for ~500 years before being thrown off axis.

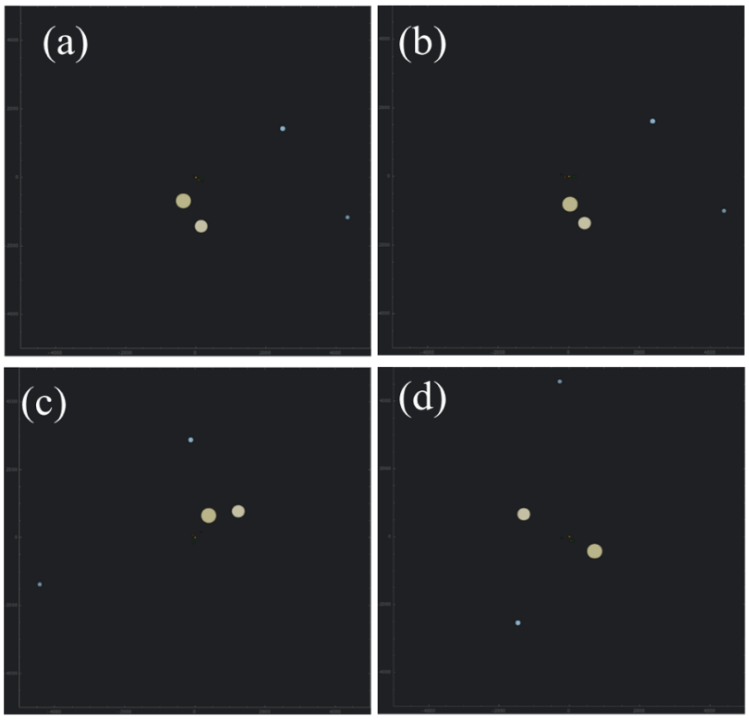

Figure 3. Snapshot of animation after (a) 0 years (b) 1 year (c) 50 years and (d) 100 years.

The central purpose of this study, to develop a flexible method for simulating and animating physics in Mathematica was met. Although this paper only explicitly deals with the spring-mass system and celestial bodies orbiting the sun, the script (attached below) can be easily modified to simulate any number of physical phenomenon by substituting the necessary force in Eq. 6 and following the subsequent steps of Euler's (or Runge-Kutta's) method outlined in the paper.

Acknowledgment

The author would like to thank his family and friends, especially Kameron, for their help. He would also like to thank his high school physics teacher for piquing his interest in physics.

References

[1] J. He, "Approximate analytical solution of Blasius' equation," *Communications in Nonlinear Science and Numerical Simulation*, vol. 3, no. 4, pp. 260-263, 1998.
[2] W. Thomson, *Theory of Vibration with Applications*, CRC Press, 2018.
[3] N. H. Asmar, *Partial Differential Equations with Fourier Series and Boundary Value Problems*, Courier Dover Publications, 2016.
[4] C. D. Murray and S. F. Dermott, *Solar System Dynamics*, Cambridge University Press, 1999.
[5] J. R. Dormand and P. J. Prince, "New Runge-Kutta algorithms for numerical simulation in dynamical astronomy," *Celestial Mechanics*, vol. 18, no. 3, pp. 223-232, 1978.

Charles Chen was born in New Brunswick, NJ, USA in September 2001. He graduated from Northwood High School. Now he currently enrolled at University of California, Berkeley. His research interests include Computational Physics and Applied Mathematics.