1 Abstract

In this paper I will cover three dynamical systems: The swinging Atwood’s machine, the double pendulum, and Julia sets. The goal was to create as many visual representations of these systems as I could to gain a better understanding of how the systems behave. In order to solve the first two systems I broke the equations of motion down into first order equations and I wrote my own fourth order Runge-Kutta integrator that solves them. For the Julia sets I set an upper bound of 2 on the absolute value of the point to see if the point is leaving the set or not. In this paper I will not use a rigorous definition of chaos because there are many conflicting definitions available and I would like to just show that my results exhibit features closely related to chaos. I found some great examples of chaos in the swinging Atwood’s machine with orbits corresponding to \( \mu = 2.536 \) and nearby neighborhoods. I also found some great aperiodic orbits of the double pendulum in the simplified case of \( m_1 = m_2 = l_1 = l_2 = 1 \). The development of Julia sets also displayed some interesting behaviors when taking walks around the complex plane.
2 Introduction and Background

In this project I study 3 important dynamical systems that tend to exhibit chaos and model them in Octave through code. First I will explore the numerical methods available to integrate the first two systems, and after developing numerical solutions for all 3 systems, I plot many intervals of initial conditions that demonstrate chaos and other visually aesthetic features and animate them through Mencoder.

The first of the three mathematical systems I study is the swinging atwood’s machine, which is a physical system that consists of two masses hanging from opposite ends of a massless string. This problem is known to be sensitive to the ratio of mass between the two hanging masses and may develop aperiodic and chaotic behavior.

The second system I study is the double pendulum, which is a physical system that consists of a pendulum that is allowed to swing freely about a pivot, and a second pendulum that swings completely on the end of the first. This system also displays odd aperiodic and chaotic motion.

The final system I study is Julia sets, which is a set of points in the complex plane that (when given a point c in \( \mathbb{C} \)) converges under the influence of the sequence \( z_{n+1} = z_n^2 + c \). By moving through the complex plane on parametrically defined equations, it’s possible to discover paths in \( \mathbb{C} \) that exhibit interesting features.
3 Problem Solving Methods

When considering the problem of integrating a swinging atwood’s machine and a double pendulum (I’ll cover the specifics of those in their respective sections) there are unlimited options available including Euler’s method, the midpoint method, and an unlimited number of Runge-Kutta methods, adaptive methods, and simplectic methods. Since Euler’s method and the midpoint method are accurate up to first and second order errors, respectively, I decided that the classical Runge-Kutta method would be the most cost effective. Classical Runge-Kutta is accurate up to fourth order errors, and although it doesn’t force itself to take smaller steps when it senses the system becoming sensitive like adaptive methods and it doesn’t preserve system energy like simplectic methods, it seemed very accurate without the loss of speed that more advanced methods require.

For classical Runge-Kutta, which is sometimes referred to as RK4 or the Runge-Kutta method. After defining the gradient function $F(\vec{x})$ the method becomes very straightforward.

Consider the vector of the equation’s variables $x_n$ and a solution step size $h$. Then we define the vectors:

\[ v_1 = F(x_n) \]
\[ v_2 = F(x_n + \frac{h}{2}v_1) \]
\[ v_3 = F(x_n + \frac{h}{2}v_2) \]
\[ v_4 = F(x_n + hv_3) \]

One iteration of the Runge-Kutta method then assigns as follows:

\[ x_{n+1} = x_n + \frac{h}{6}(v_1 + 2v_2 + 2v_3 + v_4) \]

This equation completes one iteration of the method by stepping forward by $h$. By creating a function that mimics $F(\vec{x})$ in Octave and another that computes each $v_i$ using $F(\vec{x})$, where $F(\vec{x})$ is the gradient function I obtained from the differential equations of motion, it’s possible to loop through the main function while calling the 2 subroutines to numerically integrate the swinging Atwood’s machine and the double pendulum. The integration looped through a matrix by performing the Runge-Kutta method on the previous row, and then storing another row of variables before moving forward another step. After integration, appropriate variables were picked by columns from it and plotted, which formed the results in this project.
4 The Swinging Atwood’s Machine

The Swinging Atwood’s Machine, commonly called a SAM, is a physical system in which two masses, M and m (sometimes - not in this paper - referred to as m1 and m2), are connected by a massless string that may not stretch, and the string is connected to two pulleys that have no radius but are sufficiently separated to prevent object collision.

The first mass, m, is allowed to swing freely within the two dimensional plane; however, the second, M, may only move up and down to counteract the forces exerted by m.

SAM’s are frequently studied because of their chaotic orbits as M and m are varied within consistent initial conditions to the system. An example of a SAM is below.

![Figure 1: Swinging Atwood’s Machine](image)

After applying Lagrange analysis on the kinetic and potential energies for the machine, Wikipedia provided the following second order differential equations:

\[ r\ddot{\theta} + 2\dot{r}\dot{\theta} + g\sin\theta = 0 \]
\[ mr\ddot{\theta}^2 - Mg + mg\cos\theta = (M + m)\ddot{r} \]

Solving the equations for \( \ddot{\theta} \) and \( \ddot{r} \) myself yields the following:

\[ \ddot{\theta} = \frac{mr(\dot{\theta}^2) - Mg + mg\cos\theta}{M + m} \]
\[ \ddot{r} = \frac{-2\dot{r}\dot{\theta}}{r} - \frac{g\sin\theta}{r} \]
Applying the change of variables $\rho = \dot{r}, \dot{\rho} = \ddot{r}, \tau = \dot{\theta}, \dot{\tau} = \ddot{\theta}$ reduces this system down to 4 first order equations, and our gradient function $F(\vec{x})$ is obvious from this.

\[
\begin{pmatrix}
\dot{r} \\
\dot{\theta} \\
\dot{\rho} \\
\dot{\tau}
\end{pmatrix} = \begin{pmatrix}
\rho \\
\tau \\
\frac{m_r(\dot{\theta}^2) - M g + m g \cos \theta}{m + \frac{m}{r}} \\
\frac{-2 r \dot{\theta}}{r} - g \sin \theta
\end{pmatrix} = F \begin{pmatrix}
r \\
\theta \\
\rho \\
\tau
\end{pmatrix}
\]

Once this system of equations was established, I constructed the gradient function like the one above and the Runge-Kutta method like in equation 1. Looping through and iterating continuously provided a matrix and some very interesting graphics which will be covered later in the results section.

As a final note on my work with the SAM, since all of the output in the functions I cite in the code appendix are jpeg images, I’d like to mention that in order to transform a folder full of ordered jpeg images into the video files I used in my presentation, I used the following command in the terminal:

```
```
5 The Double Pendulum

The Double Pendulum is a physical system with one pendulum with length $l_1$ and mass $m_1$ and a second pendulum with length $l_2$ and mass $m_2$ attached to the end of the first pendulum. The configuration in the image below is the general setup for the double pendulum.

![Figure 2: Double Pendulum](image)

Given a set of initial conditions, $\theta_1, \dot{\theta}_1, \theta_2, \text{ and } \dot{\theta}_2$, it is possible to observe the trajectories of the masses, but it requires two important differential equations that can be obtained from the Lagrangians, which Wolfram Scienceworld provided:

\begin{align*}
(m_1 + m_2)l_1 \ddot{\theta}_1 + m_2 l_2 \ddot{\theta}_2 \cos(\theta_1 - \theta_2) + m_2 l_2 \dot{\theta}_2^2 \sin(\theta_1 - \theta_2) + g(m_1 + m_2) \sin(\theta_1) &= 0 \\
m_2 l_2 \ddot{\theta}_2 + m_2 l_1 \dot{\theta}_1 \cos(\theta_1 - \theta_2) - m_2 l_1 \dot{\theta}_2^2 \sin(\theta_1 - \theta_2) + m_2 g \sin(\theta_2) &= 0
\end{align*}

And after I performed excruciating simplification and divisions, solving for $\ddot{\theta}_1$ and $\ddot{\theta}_2$ yields:

\begin{align*}
\ddot{\theta}_1 &= \frac{-m_2 \sin(\theta_1 - \theta_2)(l_2 \dot{\theta}_2^2 + l_1 \dot{\theta}_1 \cos(\theta_1 - \theta_2)) + g((m_1 + m_2) \sin(\theta_1) - m_2 g \sin(\theta_2) \cos(\theta_1 - \theta_2))}{(l_1 (m_1 + m_2 \sin(\theta_1 - \theta_2)^2))} \\
\ddot{\theta}_2 &= \frac{\sin(\theta_1 - \theta_2)(m_2 l_2 \dot{\theta}_2^2 \cos(\theta_1 - \theta_2) + l_1 \dot{\theta}_1^2 (m_1 + m_2)) + (m_1 + m_2) (\cos(\theta_1 - \theta_2) \sin(\theta_1) - \sin(\theta_2))}{(l_2 (m_2 \sin(\theta_1 - \theta_2)^2 + m_1))}
\end{align*}

Of course, now we simply define two more variables to reduce the system into 4 first order differential equations. Set $\tau_1 = \dot{\theta}_1$ and $\tau_2 = \dot{\theta}_2$ so $\ddot{\tau}_1 = \ddot{\theta}_1$, which is one of the complicated expressions above, and $\ddot{\tau}_2 = \ddot{\theta}_2$ which another of our complicated expressions. Now that we have all first order derivatives, we then define the function for our gradient $F(\vec{x})$ as follows:
5 The Double Pendulum

\[
F \begin{pmatrix} \theta_1 \\ \theta_2 \\ \tau_1 \\ \tau_2 \end{pmatrix} = \begin{pmatrix} \tau_1 \\ \tau_2 \\ -m_2 \sin(\theta_1 - \theta_2)(l_2 \tau_2^2 + l_1 \tau_1^2 \cos(\theta_1 - \theta_2)) + g((m_1 + m_2) \sin(\theta_1) - m_2 g \sin(\theta_2) \cos(\theta_1 - \theta_2)) \\ \sin(\theta_1 - \theta_2)(m_2 l_2^2 \tau_2^2 \cos(\theta_1 - \theta_2) + m_2 l_1^2 (m_1 + m_2)) + \frac{g(m_1 + m_2)(\cos(\theta_1 - \theta_2) \sin(\theta_1) - \sin(\theta_2))}{(l_2^2 \sin(\theta_1 - \theta_2)^2 + m_1)} \end{pmatrix}
\]

Once this system of equations was established, I constructed the gradient function like the one above and the Runge-Kutta method like in equation 1. Looping through and iterating the code to plot a single orbit continuously provided a matrix and some very interesting graphics.

Again, with the double pendulum, in order to make a video with Mencoder I entered the following command into the terminal:

6 Julia Sets

Unlike the previous two systems, which were based purely upon the existence of real physical systems, the Julia set is a mathematical object which does not have a nice physical representation. The Julia set of the point \( c \in \mathbb{C} \) is the set consisting of all points in \( \mathbb{C} \) that do not diverge under the iteration function \( z_{n+1} = z_n^2 + c \). Figure 3 shows an example of the Julia set corresponding to \( c = 0 + 0.750i \).

![Julia Set with c = 0.000 + 0.750 i](image)

**Figure 3: Julia Set with** \( c = 0 + 0.750i \)

Frequently these sets are colored so that points in the set are one color but then points that diverge are colored based on how many iterations it took for the point to diverge. Frequently a large "depth", which is the number of iterations of the formula above to approximate as \( \infty \) for convergence/divergence of our sequence, is needed in order to make sure that the boundary of the set takes the correct form, but I just used 32 for my depth in generating all of my work, and the boundaries look quite accurate compared to standard Julia set documentation.
7 Results

With the Julia sets, I merely found each corresponding folder of graphics and entered the following command in the terminal:

```
```

7 Results

Here are some of the most interesting images produced by the graphs of the swinging Atwood’s machine. These were produced along with about 15000 other images with the following command:

```
GraphicsSAM(0.001,1,15,0.01,5000)
```

![Figure 4: Swinging Atwood's Machine with \( u = 2.812 \)](image_url)
Figures 4 and 5 show nice unique periodic behavior of the SAM orbit, but figure 6 shows that a small deviation from the parameter used in Figure 4 provides a drastically different result. This shows that the function can be chaotic based on this parameter, and I believe that figure 6 has no numerical deficiencies because if we accidentally generated energy in the system by a numerical defect, then the orbit would have escaped the unit circle drastically.
7 Results

Here are some of the most interesting images produced by the graphs of the double pendulum. In all of my double pendulum simulations I used \( m_1 m_2 = l_1 = l_1 = 1 \). These were produced with the following commands:

\[
\text{GraphicsDP}(\pi/2, \pi/2, 0, -4, 1, 1, 1, 0.01, 1000) \\
\text{GraphicsDP}(\pi, \pi, 0, -2, 1, 1, 1, 0.01, 2000)
\]

**Figure 7:** Double Pendulum with \( \theta_1 = \pi/2, \theta_2 = \pi/2, \dot{\theta}_1 = -4, \dot{\theta}_2 = 0 \)

**Figure 8:** Double Pendulum with \( \theta_1 = \pi, \theta_2 = \pi, \dot{\theta}_1 = 0, \dot{\theta}_2 = -2 \)

When looking at both figure 7 and 8, the only striking features are the fact that the pendulums do not escape a circle of radius 2. Otherwise, it appears that they swing
7 Results

around wildly with a sort of chaotic motion.

Here are some of the most interesting images produced by the graphs of Julia sets. These were produced with the following commands:

GraphicsJulia(100, [0.250, -0.750], [1.000, 0], true)
GraphicsJuliaC(25, 1, true)

![Figure 9: Julia Set with $c = -0.020 + 0.770i$](image)

**Figure 9: Julia Set with $c = -0.020 + 0.770i$**

![Figure 10: Julia Set with $c = 0.750 + 0.000i$](image)

**Figure 10: Julia Set with $c = 0.750 + 0.000i$**

When looking at figures 9 and 10, the most important points to consider are the boundaries of the sets, because the boundaries take on shapes have infinitely repeating
self-similarities when zooming in on most points on the boundary. The nice symmetries of the sets but infinite self-similarities and strange features and the way this set changes while moving around the complex plane are good examples of how chaotically it can evolve.

Since there are some great examples of periodic motion, aperiodic motion, and self-similarities and symmetries, we have successfully found examples of chaos in these particular dynamical systems. Separate from this paper are also several animations that were generated that show better detail in how chaotically these systems behave. These images were only meant to summarize the actual behavior, and I think they did so very well.
8 Code Appendix

Code to plot the SAM:

```matlab
function [r,th] = PlotSAM(r,th,rho,tau,M,m,its,doplota,doplotb)

%% Given the scalar initial conditions for the SAM r,theta,rho,tau, and
%% the scalar variables for the masses M, m, PlotSAM(...) plots the
%% orbit of this specific scenario using a RK4 integration with h as
%% the step size out to its iterations doplota and doplotb are booleans
%% that determine whether to draw the graphs for the r vs. theta and X
%% vs. Y plots or not, respectively.

%% Initialize the variables so that things run a little more smoothly
% g=9.80665;
mat=zeros(its,4); % Set our matrix empty for speed
mat(1,:)=[r,th,rho,tau]; % Define our initial conditions

%% Let physics take it away!!
for i=2:its
    mat(i,:)=RK(mat(i-1,:),M,m,h); % Iterate on each previous row
end
r=mat(:,1); % strip r values from column 1
th=mat(:,2); % strip theta values from column 2
u=M/m; % calculate the mass ratio

% t1=sprintf('Plot of R vs. Theta with u = %1.3f',u); % Title both graphs
% t2=sprintf('SAM Orbit with u = %1.3f',u);

if doplota
    figure(1) % Focus on figure 1
    plot(r,th,'k') % Plot r vs. theta
    title(t1) % Title it
    xlabel('R Axis') % Label axes
    ylabel('Theta Axis')
end

x=r.*cos(th-pi/2); % Extract x and y, but remember from our
% definition, the angles are 90 degrees off
y=r.*sin(th-pi/2);
```

8 Code Appendix
if doplotb
    figure(2) % Focus on figure 2
    plot(x,y,'r'); % Plot x and y
    title(t2) % Title accordingly
    xlabel('X Axis') % Label axes
    ylabel('Y Axis')
    axis([-2,2,-2,2]) % Grab an appropriate region
end
endfunction

%% Have a function set up to run the differential equations
function w = SamEngine(v,M,m)
    %% SamEngine(v,M,m) takes the vector of r, theta, rho, and tau and
    %% calculates the derivative of each
    % Remember v = [r,th,rho,tau]
    % Remember w = [rd,thd,rhod,taud]
    w=zeros(1,4); % For some miniscule speed
    g=9.80665; % Reset g
    w(1)=v(3); % Equations of motion, which should be correct
    w(2)=v(4);
    w(3)=(m*v(1)*v(4)^2-M*g+m*g*cos(v(2)))/(M+m);
    w(4)=(-2*v(3)*v(4)-g*sin(v(2)))/v(1);
endfunction

%% Have a function set up for the runge-kutta method
function w = RK(v1,M,m,h)
    %% Set up the standard RK4 as its normally defined
    v2=v1+h/2*SamEngine(v1,M,m);
    v3=v1+h/2*SamEngine(v2,M,m);
    v4=v1+h*SamEngine(v3,M,m);
    w=v1+h*(SamEngine(v1,M,m)+2*SamEngine(v2,M,m)+2*SamEngine(v3,M,m)
    +SamEngine(v4,M,m))/6;
Code to generate SAM graphics:

```matlab
function ret = GraphicsSAM(res,min,max,h,its)

%% GraphicsSAM(res,min,max,h,its) takes scalars res, min, max, h, and
%% its and draws the orbits of the swinging atwood's machine between
%% u=min and u=max with res being the step size between each change in
%% u. h is the step size to use for the plot, and its is the number of
%% iterations to use to solve the SAM for the plot. It assumes initial
%% conditions [1,-pi/2,0,0] for all 4 variables in the sam and plots
%% both the R vs. Theta graph and the X vs. Y graph.

n=ceil((max-min)/res)+1; % Find out how many images we're going to need
u=linspace(min,max,n); % Use linspace to vary u from min to max
for i=1:n % Loop for the number of plots we need
    PlotSAM(1,-pi/2,0,0,u(i),1,h,its,true,true);
    figure(2) % Focus on the X vs. Y plot
    name=sprintf('SamXY%1.3fwH%dwIts%i.jpg',u(i),h,its);
    print(name) % Save the image
end
endfunction
```

Code to generate Double Pendulum Orbit:

```matlab
function [th1,th2] = PlotDP(th1,th2,tau1,tau2,l1,l2,m1,m2,h,its)

%% Given the scalar initial conditions for the Double Pendulum
%% th1,th2,tau1,tau2, and the scalar variables for the lengths of both
%% pendulums l1, l2, and their masses m1, m2, PlotDP(...) plots the
%% orbit of this specific scenario using a RK4 integration with h as
%% the step size out to its iterations. Currently, changing l1, l2, m1,
%% and m2 has no effect because they're set to 1 on line 16

%% Initialize the variables so that things run a little more smoothly
g=9.80665; % Use an accurate g because strange things may happen
```
mat=zeros(its,4); % Initialize for some speed (hopefully)
mat(1,:)=[th1,th2,tau1,tau2]; % Set up first row with initial conditions

for i=2:its % For the rows 2 through its
    mat(i,:)=RK(mat(i-1,:),1,1,1,1,h); % Iterate on the previous row with step size h to get the next row
end

th1=mat(:,1); % Strip the first theta values from the matrix
th2=mat(:,2); % Strip the second theta values from the matrix

t1=sprintf('Plot of Theta1 vs. Theta2'); % Generate plot names
t2=sprintf('Double Pendulum Orbit'); % " " "

figure(1) % Focus on the first image
plot(th1,th2,'k') % Plot theta1 vs. theta2
title(t1) % Title the plot
xlabel('Theta1 Axis') % Label axes
ylabel('Theta2 Axis') % " "

x=l1.*cos(th1-pi/2)+l2.*cos(th2-pi/2); % Remember to adjust the
% angles by 90 degrees, then find where we are in the plane
y=l1.*sin(th1-pi/2)+l2.*sin(th2-pi/2); % "

figure(2) % Focus on the second image
plot(x,y,'r'); % Plot the X vs. Y graph
title(t2) % Title it
xlabel('X Axis') % Label axes
ylabel('Y Axis') % " "
axis([-2,2,-2,2]) % Make sure our axes stay in a nice range

endfunction

%%% Have a function set up to run the differential equations
function w = DPEngine(v,m1,m2,l1,l2)

%%% DPEngine takes the vector of conditions v (which is
%%% theta1,theta2,tau1,tau2), and the global settings to iterate the
%%% physics engine on the vector once

% Recall what v looks like
v = [th1,th2,tau1,tau2]
% Recall what \( w \) will look like  \( w = [\text{th1d}, \text{th2d}, \tau_1d, \tau_2d] \)
\( w = \text{zeros}(1, 4); \)  % Predefine \( w \) for a tiny amount of speed (it adds up?)

\( \text{th1} = v(1); \)  % Change our vector variables to names that make the
% equations a little easier to follow
\( \text{th2} = v(2); \)  % 
\( \text{thd1} = v(3); \)  % 
\( \text{thd2} = v(4); \)  % 
\( d = \text{th1} - \text{th2}; \)  % 
\( g = 9.80665; \)  % Reset \( g \)

\( w(1) = \text{thd1}; \)  % These equations are extremely messy, but they're the
% correct equations of motion solved
\( w(2) = \text{thd2}; \)
\( w(3) = \frac{(-m2*\sin(d) \ast (12*\text{thd2}^2 + 11*\text{thd1}^2 \ast \cos(\text{th1} - \text{th2}))) \ast g \ast ((m1+m2) \ast \sin(\text{th1}) - m2 \ast g \ast \sin(\text{th2}) \ast \cos(d)))}{(11 \ast (m1+m2 \ast \sin(d)^2))}; \)
\( w(4) = \frac{\sin(d) \ast (m2 \ast 12 \ast \text{thd2}^2 \ast \cos(\text{d}) + 11 \ast \text{thd1}^2 \ast (m1+m2))/\left(12 \ast (m2 \ast \sin(\text{d})^2 + m1))\right) \ast g \ast (m1+m2) \ast (\cos(d) \ast \sin(\text{th1}) - \sin(\text{th2}))/\left(12 \ast (m2 \ast \sin(\text{d})^2 + m1))\right)}{6}; \)
endfunction

%% Have a function set up for the runge-kutta method
function \( w = \text{RK}(v1, m1, m2, l1, l2, h) \)

%% Define our RK4 method to normal standards
\( v2 = v1 + h/2 \ast \text{DPEngine}(v1, m1, m2, l1, l2); \)
\( v3 = v1 + h/2 \ast \text{DPEngine}(v2, m1, m2, l1, l2); \)
\( v4 = v1 + h \ast \text{DPEngine}(v3, m1, m2, l1, l2); \)

\( w = v1 + h \ast (\text{DPEngine}(v1, m1, m2, l1, l2) + 2 \ast \text{DPEngine}(v2, m1, m2, l1, l2) + 2 \ast \text{DPEngine}(v3, m1, m2, l1, l2) \ast \text{DPEngine}(v4, m1, m2, l1, l2))/6; \)
endfunction

Code to generate Double Pendulum Graphics:

function \([\text{th1}, \text{th2}] = \text{GraphicsDP} (\text{th1}, \text{th2}, \text{tau1}, \text{tau2}, l1, l2, m1, m2, h, its) \)

%% Given the scalar initial conditions for the Double Pendulum
%% \( \text{th1}, \text{th2}, \text{tau1}, \text{tau2}, \) and the scalar variables for the lengths of both
%% pendulums \( l1, l2, \) and their masses \( m1, m2, \) \text{PlotDP}(...) plots the
orbit of this specific scenario using a RK4 integration with h as the step size out to its iterations. Currently, changing l1, l2, m1, \%
m2 has no effect because they’re set to 1 on line 16

Initialize the variables so that things run a little more smoothly g=9.80665; %Use a really accurate g because strange things may happen as errors build up in iterations Initialize our matrix of important variables so it isn’t constantly resizing Set up the first row with our initial conditions

% Name the graph we’ll draw
%For the rows 2 through its
% Iterate on the previous row with step size h to get the next row
% Strip the first theta
% Strip the second theta
% Remember to adjust the angles by 90 degrees, then find where we are in the plane
% Focus on the second image
% Plot the X vs. Y graph
% Title it
% Label axes
%" "
% Make sure our axes stay in a nice range in case something gets out of control
% Pick a number so naming
% Write a file name
% Save the file

Have a function set up to run the differential equations
function w = DPEngine(v,m1,m2,l1,l2)
%% DPEngine takes the vector of conditions \( \mathbf{v} \) (which is \( \mathbf{v} = [\theta_1, \theta_2, \tau_1, \tau_2] \)), and the global settings to iterate the physics engine on the vector once

% Recall what \( v \) looks like \( \mathbf{v} = [\theta_1, \theta_2, \tau_1, \tau_2] \)
% Recall what \( w \) will look like \( \mathbf{w} = [\dot{\theta}_1, \dot{\theta}_2, \dot{\tau}_1, \dot{\tau}_2] \)
\textbf{w} = \textbf{zeros}(1, 4); % Predefine \textbf{w} for a tiny amount of speed (it adds up?)

\textbf{th1} = \textbf{v}(1); % Change our vector variables to names that make the equations a little easier to follow
\textbf{th2} = \textbf{v}(2); % "
\textbf{thd1} = \textbf{v}(3); % "
\textbf{thd2} = \textbf{v}(4); % "
\textbf{d} = \textbf{th1} - \textbf{th2}; % "

\textbf{g} = 9.80665; % Reset \textbf{g}

\textbf{w}(1) = \textbf{thd1}; % These equations are extremely messy, but they’re the correct equations of motion solved correctly
\textbf{w}(2) = \textbf{thd2};
\textbf{w}(3) = (-\textbf{m}_2 * \textbf{sin} (\textbf{d}) * (12 * \textbf{thd2}^2 + 11 * \textbf{thd1}^2 * \textbf{cos} (\textbf{th1} - \textbf{th2}))) + g * ((\textbf{m}_1 + \textbf{m}_2) * \textbf{sin} (\textbf{th1}) - \textbf{m}_2 * \textbf{g} * \textbf{sin} (\textbf{th2}) * \textbf{cos} (\textbf{d})) / (12 * (\textbf{m}_2 * \textbf{sin} (\textbf{d})^2));
\textbf{w}(4) = \textbf{sin} (\textbf{d}) * (\textbf{m}_2 * \textbf{thd2}^2 * \textbf{cos} (\textbf{d}) + 11 * \textbf{thd1}^2 * (\textbf{m}_1 + \textbf{m}_2))/(12 * (\textbf{m}_2 * \textbf{sin} (\textbf{d})^2 + \textbf{m}_1)) + g * (\textbf{m}_1 + \textbf{m}_2) * (\textbf{cos} (\textbf{d}) * \textbf{sin} (\textbf{th1}) - \textbf{sin} (\textbf{th2}))/12 * (\textbf{m}_2 * \textbf{sin} (\textbf{d})^2 + \textbf{m}_1));
endfunction

%% Have a function set up for the runge-kutta method
function \textbf{w} = \textbf{RK} (\textbf{v}_1, \textbf{m}_1, \textbf{m}_2, \textbf{l}_1, \textbf{l}_2, \textbf{h})

%% Define our RK4 method to normal standards
\textbf{v}2 = \textbf{v}1 + \textbf{h}/2 * \textbf{DPEngine} (\textbf{v}1, \textbf{m}_1, \textbf{m}_2, \textbf{l}_1, \textbf{l}_2);
\textbf{v}3 = \textbf{v}1 + \textbf{h}/2 * \textbf{DPEngine} (\textbf{v}2, \textbf{m}_1, \textbf{m}_2, \textbf{l}_1, \textbf{l}_2);
\textbf{v}4 = \textbf{v}1 + \textbf{h} * \textbf{DPEngine} (\textbf{v}3, \textbf{m}_1, \textbf{m}_2, \textbf{l}_1, \textbf{l}_2);

\textbf{w} = \textbf{v}1 + \textbf{h} * (\textbf{DPEngine} (\textbf{v}1, \textbf{m}_1, \textbf{m}_2, \textbf{l}_1, \textbf{l}_2) + 2 * \textbf{DPEngine} (\textbf{v}2, \textbf{m}_1, \textbf{m}_2, \textbf{l}_1, \textbf{l}_2) + 2 * \textbf{DPEngine} (\textbf{v}3, \textbf{m}_1, \textbf{m}_2, \textbf{l}_1, \textbf{l}_2)) / 6;
endfunction
Code to plot the Julia set:

```matlab
function ret = PlotJulia(Z,c)

%% PlotJulia(Z,c), which was constructed purely to be called from
%% GraphicsJulia will take a large mesh matrix Z and an imaginary
%% number c and plot the julia set associated with them

W=zeros(size(Z)); %Create our color matrix from the size of Z to match

for k=1:32 % Use a depth of only 32 colors
    Z=Z.^2+c; % Traditional julia set iteration
    W(Z>2 & W==0)=k; % All points that stepped out of our upper limit of
    %2 for the first time are marked with the iteration that they escaped on
end

W=W/32; % Normalize all the colors
colormap('jet') % Use a cool color scheme
pcolor(W); % Draw our julia set
axis('square','equal','off'); % Plot a Size x Size square with no labels
shading('flat'); % Take off messy grid lines
title=sprintf('Julia Set with c = %1.3f + %1.3f i',real(c),imag(c)); %Name the graph with the position in the complex plane
drawnow(); % Make sure Octave draws it immediately (associated
%with bug that it wouldn't draw things in the GraphicsJulia loop)
endfunction
```

Code to generate Julia set graphics in a circle:

```matlab
function ret = GraphicsJuliaC(its,r,p)

%% GraphicsJuliaC(its,r,p) acts almost exactly the same as its
%% counterpart GraphicsJulia, except that this loop travels around a
%% circle in the complex plane with scalar radius r

its=its+1; % If we want its segments, then we need its+1 points

for k=1:32 % Use a depth of only 32 colors
    Z=Z.^2+c; % Traditional julia set iteration
    W(Z>2 & W==0)=k; % All points that stepped out of our upper limit of
    %2 for the first time are marked with the iteration that they escaped on
end

W=W/32; % Normalize all the colors
colormap('jet') % Use a cool color scheme
pcolor(W); % Draw our julia set
axis('square','equal','off'); % Plot a Size x Size square with no labels
shading('flat'); % Take off messy grid lines
title=sprintf('Julia Set with c = %1.3f + %1.3f i',real(c),imag(c)); %Name the graph with the position in the complex plane
drawnow(); % Make sure Octave draws it immediately (associated
%with bug that it wouldn't draw things in the GraphicsJulia loop)
% Use this convention in case I want to go in and change the values
% later, although I probably will not. The purpose of this block is to
% expedite calculations and improve runtime when replotting in the same
% plane multiple times

% Center the plot at (0+0i)
cx=0;
cy=0;
sx=1.5;  % Add padding of 1.5 on both x and y around our plot
sy=1.5;

% Define x=rcos(theta) to go around the circle
xt=r*cos(t);
% y=rsin(theta)

% Define this set of points as x+iy
yt=r*sin(t);
zt=xt+i*yt;

% Construct x and y values in the plane
x=linspace(cx-sx,cx+sx,size);
y=linspace(cy-sy,cy+sy,size);

[X,Y]=meshgrid(x,y);  % Build a meshgrid of x and y values

Z=X+i*Y;  % Construct the complex plane to pass to PlotJulia

for r=1:its
    PlotJulia(Z,zt(r))  % Pass on what we have to the plot function
    if p  % If we want to save the images
        abc=r+10000;  % Create a big number that will prevent 1
                     % from coming before 10 in file names
        name=sprintf('JuliaSetC%i.jpg',abc);  % Print out our name with the
        print(name);  % Save the file
    end
end
endfunction

Code to generate Julia set graphics in a straight line:

function ret = GraphicsJulia(its,l1,l2,p)

% GraphicsJulia(its,l1,l2,p) takes two points l1 and l2 as row vectors
% and connects them with a parametric line through the complex plane
% broken into its segments, where its is a natural number. It then
% plots each julia set along the line according to PlotJulia, and if p
% (a boolean value) is true, then the function prints each graph to a
% jpeg file in the current directory for later animations.

its=its+1;  % If we want segments, we need its+1 points
\begin{verbatim}
t=linspace(l1(1),l1(2),its)+1i*linspace(l2(1),l2(2),its);
\% Construct our line through the complex plane between both points
size=1000; \% Set our resolution to 1000x1000

\% Use this convention in case I want to go in and change the values
\% later, although I probably will not. The purpose of this block is to
\% expedite calculations and improve runtime when replotting in the same
\% plane multiple times
cx=0; \% Center the plot at (0+0i)
cy=0;
sx=1.5; \% Add padding of 1.5 on both x and y around our plot
sy=1.5;
x=linspace(cx-sx,cx+sx,size); \% Construct x and y values to form plane
y=linspace(cy-sy,cy+sy,size);
[X,Y]=meshgrid(x,y); \% Build a meshgrid of x and y values
Z=X+i*Y; \% Construct the complex plane to pass to PlotJulia

for r=1:its
    PlotJulia(Z,t(r)) \% Pass on what we have to the plot function
    if p \% If we want to save the images
        abc=r+10000; \% Create a big number that will prevent 1 from
        \% coming before 10 in file names
        name=sprintf('JuliaSet%0i.jpg',abc);
        \% Print out our name with the file extension
        print(name); \% Save the file
    end
end
endfunction
\end{verbatim}
9 References

Information regarding Runge-Kutta methods:

http://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods

Information regarding the swinging Atwood’s machine:

http://en.wikipedia.org/wiki/Swinging_Atwood%27s_machine

Other work on swinging Atwood’s machines:

http://openprocessing.org/sketch/27494
https://www.youtube.com/watch?v=oVsfdm9sacI

Information regarding the double pendulum:


Information regarding Julia sets:


Background on chaos theory:

http://en.wikipedia.org/wiki/Topological_mixing#Topological_mixing