ME 2080 - Dynamics

Dynamics Group Project 1:
Ball Kinematics Modelling and Experiment

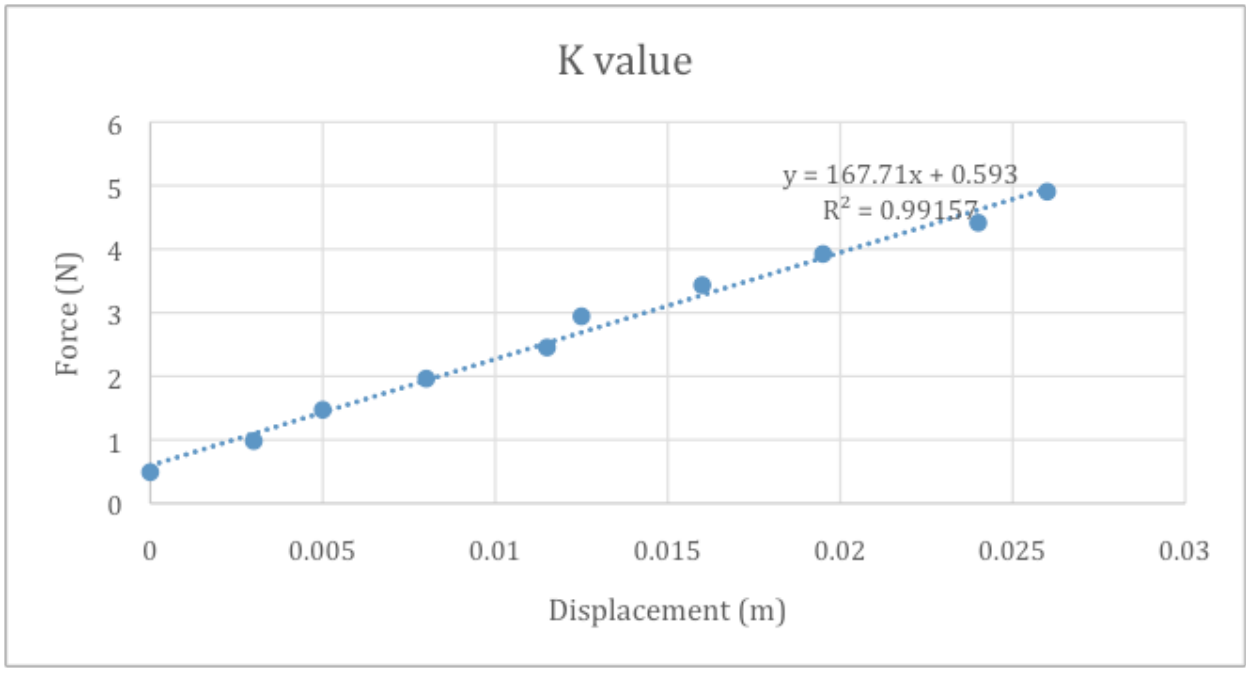
Authors: David Orr, Sam Feinman, Brian Sheng

Introduction

For this project, we modeled the equations of motion for a problem involving launching a ball horizontally off a table and into a bowl placed a certain distance away. The goal was to numerically model the trajectory of a ping pong ball launched from a certain height and have it bounce once and land in a bowl. The launching mechanism used was a rubber band that was deflected a particular distance in order to achieve our horizontal velocity. For this project, we had to account for: force due to air drag; the effective spring constant, k, of the rubber band; and, the coefficient of restitution between the floor and the ball. After solving those values, we used computer simulations to find the distance, d, needed to have the ball reach the bowl. Finally, we conducted the experiment and successfully had the ball reach the bowl in one bounce. This report will elucidate on how we solved for our values, used the computer simulation, and conducted the experiment.

Measurements

The first value we solved for was our effective spring constant, k, of the rubber band. To find out this value, we used a series of weights and plotted the corresponding displacements and used a best-fit line to find the slope, which is the spring constant. We started with an initial weight of 50 grams and an initial displacement of 0.06 meters. From there, in increments of 50 grams, we measured the displacement of the rubber band until we had 500 grams on the rubber band. We then took the force in Newtons and measured it against its respective displacement. Finally, we used a line of best fit to find the slope. Below is a graph of Force vs. Displacement:



the effective spring constant turned out to be **167.71 N/m.**

The next variable to solve for was the coefficient of restitution between the ball and floor. To solve for this, we had to determine the velocity of the ball right before it makes an impact with the ground, and then find the velocity immediately after it bounces off the floor. We did this using the conservation of mechanical energy and measure the initial height and the maximum height achieved after the bounce and used the following equation:

$$e = \frac{v_f}{v_i} = \frac{\sqrt{2gh_f}}{\sqrt{2gh_i}}$$

Our final value to solve for, the force due to air drag, was solved using the following equation:

$$F_d = 1/2\, pC_dV^2A$$

F_d = Force due to Air Drag
p = density of air (1.2 kg/m^3)
C_d = drag coefficient (.445)
A = Projected area (.001256637 m^2)
V = velocity of ping pong ball

Using this equation and a solved velocity of the ball, we found a drag coefficient of **0.000359**.

Computer Simulation

After solving for those critical values, we developed a Matlab codebase to help us model the ball trajectory and kinematics. We created three functions, ball_bounce_interp.m, launch_ball.m, and RK4.m, which helped set up particular values to solve for d. It is important to note that the approximate ball mass used in our initial velocity calculation was found to be **0.0027 kg** via internet sources.

The RK4.m function performs both a numerical single and double integration on acceleration to find the ball's velocity and position based on its acceleration, previous position, and previous velocity. It does so using the Runge-Kutta Fourth Order numerical method, which uses four calculated k values, calculated from the function to be integrated, to compute the integral for the current time step. In our function this is done for acceleration and velocity, effectively performing a single and double integration to produce velocity and acceleration values, respectively, for the current iteration.

The ball_bounce_interp.m function is called when the ball passes through or contacts the ground, essentially when the y position is less than or equal to zero. It handles bouncing the ball based on its velocity upon impact and its coefficient of restitution, e. It first computes the time from impact to a current index along with the impact velocity using linear interpolation. It then

uses the RK4 function to solve for the ball's new y position and velocity by using the newly generated time step and the computed impact velocity.

The launch_ball.m function is used to run a "ball launch" simulation. It takes a ball structure with the initial position fields already specified, a spring stiffness, and initial spring displacement as its arguments. It first computes the initial velocity of the ball using the equation below, then it calls the RK4 function iteratively to model the ball's kinematics, calling the ball_bounce_interp function if the ball passes through the ground, and terminating the simulation on the second bounce (the termination condition for the while loop). Some bookkeeping regarding the form of the acceleration function depending on direction of the velocity vector is also handled in this function. This whole process is applied to find the kinematics of the ball in the x and y directions, while simultaneously incrementing a time array (the ball.time field). The resulting x and y components of position, velocity, and acceleration are stored in the respective fields of the ball structure (i.e. ball.x, ball.y, ball.vx, etc.), and the ball structure is then returned.

$$v_i = \sqrt{\frac{k}{m}}x$$

The equation used to determine the initial x-velocity for the ball.

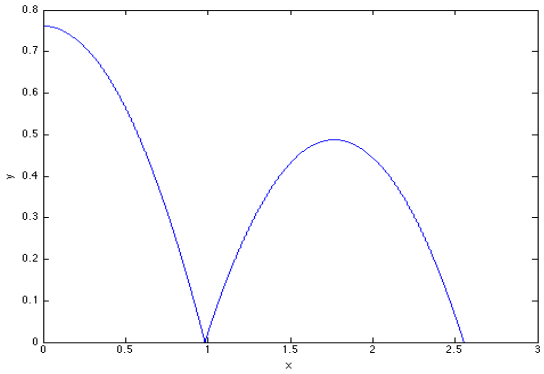
Finally, we have the launch_script.m, which calls all of the previously defined functions and executes them to run a simulation of our experiment and generate a useful data set contained in a ball structure, along with descriptive plots. In general, this script contains the numerous constants and values given, or determined earlier, which are then used to populate fields of a ball structure to be passed to the launch_ball function. It starts by populating fields for initial position in a ball structure, using the value of the variable h for the initial y position, and setting the initial x position equal to zero. Next, it initializes the x and y velocities (the launch ball function will calculate the initial x velocity) and defines acceleration function handles for the ball structure for the cases of positive and negative velocities, relative to our coordinate system (right and up are positive). It then loads the acceleration functions into corresponding fields of ball structure, and subsequently loads the ball's mass, coefficient of restitution, and the initial time (zero) into their respective fields (ball.m, ball.e, etc.). After all this, the launch_ball function is called and runs the simulation with the given input arguments of the ball structure, k_stiffness variable, and initial spring displacement variable (displacement). The function then returns a ball structure with fields containing arrays for time as well as x and y position, velocity, and acceleration that are then plotted for analysis. All of the functions and scripts described can be found in Appendix 2 of this report.

We verified the code using several different methods. The first was by analyzing the code's accuracy modelling a dragless ball bounce scenario and comparing with the analytic solution and our approximate solution. The analytic solution gave a second bounce distance of 2.5547 m and while the model predicted 2.5543 m. We also compared the projected distance with and without air drag (drag coefficient = 0.000359). The results were very similar, with the model giving a projected distance of 2.5543 m, and the model with drag giving a projected distance of 2.5537 m (less than the distance travelled in the dragless model, as predicted). The

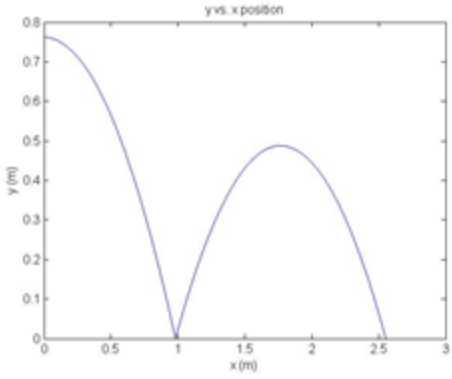
graph outputs for the analytic model, and our simulation were also in very close agreement. More quantitatively, the maximum percent error in our numerical model's position data points relative to the analytic solution was very low, to the point of being negligible (sub-millimeter deviations in distance values). We also verified our code by checking the edge case of a very large drag coefficient coupled with a large drop height, where terminal velocity is achieved relatively quickly. In this case, our code did behave as expected, confirming that our drag model was implemented correctly. As a final verification method for our code, we qualitatively analyzed the kinematic plots (x and y position vs. time, velocity vs. time, acceleration vs. time) output by our code to check that they agreed with our mathematical model for the kinematics. We feel verification of the code via the three aforementioned methods is adequate in showing that our code performs as expected, and that our air drag model was properly implemented.

All of our kinematic plots with respect to time (i.e. x vs. time, y vs. time, x velocity vs. time, etc.) can be found in Appendix 2.

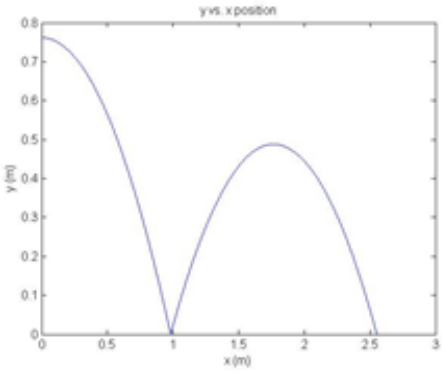
Analytic without Drag



Model Without Drag



Model With Drag



Experiment

To set up the experiment, we built a launching mechanism out of a wooden plate, two nails, and a rubber band. We set the nails 2 inches apart which would serve as holders for the rubber band. We used ping pong ball displacement in our spring displacement calculations instead of rubber band displacement because it proved to be a better measure of effective spring displacement. Due to the design of the launcher, the extra displacement of the rubber band with respect to the ping pong ball doesn't factor into work done on the ball by the rubber band. This is because upon releasing the ping pong ball, the rubber band "grabs" onto the back of the ball by pulling out the slack between the back of the ball and the rubber band. The rubber band also grips the ball from the sides, so pulling out the slack doesn't actually move it forward.

We placed the launching apparatus at a height of **30 inches (0.762 m)** off of the floor, and set the bowl on the floor **101.5 inches (2.57 m)** horizontally from our launcher based on some trials run before placing the bowl. To execute the experiment, we placed the ball in between the nails against the rubber band, and pulled the rubber band back, such that the ping pong ball was displaced **0.01 m** back. We then released the rubber band and the ball shot off the table, bounced on the ground once, and managed to make it into the bowl. It did, however, take a few attempts before we could successfully get the ball into the bowl. This could be due to slight human error, such as not pulling the rubber band such that the ping pong ball was displaced "exactly" 0.01 m. Other sources of error could be induced spin of the ball or bumps or cracks on the floor, which would affect its bounce. The experiment was considered a success, considering we were only 0.02 meters off from our predicted d value of 2.55 m.

Appendix 1 - Matlab Code

"RK4.m" Matlab function

```
% RK4 function performs both a numerical single and double integration on acceleration
% to find velocity and position based on acceleration, previous position, and previous
% velocity.

% accel_func is a function handle that holds the acceleration function for the ball.

function [new_x, new_vel] = RK4(x_prev, v_prev, accel_func, time_step)

    v1 = v_prev;                        %corresponds to k1 for finding position
    a1 = accel_func(v1)*time_step;  %corresponds to k1 for finding velocity

    v2 = v_prev + 0.5*a1*time_step; %corresponds to k2 for finding position
    a2 = accel_func(v2)*time_step;  %corresponds to k2 for finding velocity

    v3 = v_prev + 0.5*a2*time_step; %corresponds to k3 for finding position
    a3 = accel_func(v3)*time_step;  %corresponds to k3 for finding velocity


    v4 = v_prev + a3*time_step;     %corresponds to k4 for finding position
    a4 = accel_func(v3)*time_step;  %corresponds to k4 for finding velocity

    new_x = x_prev + time_step*(v1 + 2*v2 + 2*v3 + v4)/6;
    new_vel = v_prev + (a1 + 2*a2 + 2*a3 +a4)/6;

end
```

“ball_bounce_interp.m” Matlab function

```
% ball_bounce_interp function is called when the ball's y position goes below zero.
% It handles "bouncing" the ball based on its velocity on impact (calculated by linear interpolation)
% and its coefficient restitution "e".

% accel_func1 is an acceleration function handle that contains the acceleration function for postive
y-velocity case.

function [x_new,v_new] = ball_bounce_interp(x_prev, x_curr, v_prev, v_curr, accel_func1, time_step, e)

   %Computes time step from impact to current index using linear
   %interpolation
   dt2 = time_step/(x_prev- x_curr)*(-x_curr);

   x_hit = 0;  %Sets "y" position = 0 at impact

   %Computes velocity at impact using linear interpolation
   v_hit = v_prev + (v_curr-v_prev)/(x_prev-x_curr)*(-x_curr);

   v_hit = -v_hit*e; %Computes velocity after imapct using restitution coeff

   %RK4 step from impact to current index using calculated time_step
   [x_new, v_new] = RK4(x_hit, v_hit, accel_func1, dt2);

end
```

"launch_ball.m" Matlab function

```
% Function used to run "launch" simulation on ball.
% Calls ball_bounce_interp function and RK4 function.
% Called by launch_script to generate simulation results.

% "ball" input argument is a structure with fields: x, y, vx, vy, ax, ay, m, etc.

function ball = launch_ball(ball, k_stiffness, initial_displacement, time_step)

   %Computes initial x velocity from k_stiffness and initial_displacement
   ball.vx(1) = sqrt(k_stiffness/ball.m)*initial_displacement;

   i = 2;              %Initializes index to start iterating from i = 2
   num_bounces = 0;      %Initializes # of bounces

   %Computes intial x acceleration based on sign of x velocity
   if ball.vx(1) >= 0
         ball.ax(1) = ball.accel_funcx1(ball.vx(1));
   else
         ball.ax(1) = ball.accel_funcx2(ball.vx(1));
   end

   %Computes intial y acceleration based on sign of y velocity
   if ball.vy(1) >= 0
         ball.ay(1) = ball.accel_funcy1(ball.vy(1));
   else
         ball.ay(1) = ball.accel_funcy2(ball.vy(1));
   end

   while num_bounces < 2

         %Picks appropriate x acceleration function based on sign of x velocity
         if ball.vx(i-1) >= 0
         accel_funcx = ball.accel_funcx1;
         else
         accel_funcx = ball.accel_funcx2;
         end
        %Picks appropriate y acceleration function based on sign of y velocity
         if ball.vy(i-1) >= 0
         accel_funcy = ball.accel_funcy1;
         else
         accel_funcy = ball.accel_funcy2;
         end
```

```
        %Computes current x and y velocity and position using RK4 and
        %increments time array
        [ball.x(i),ball.vx(i)] = RK4(ball.x(i-1), ball.vx(i-1), accel_funcx, time_step);
        [ball.y(i),ball.vy(i)] = RK4(ball.y(i-1), ball.vy(i-1), accel_funcy, time_step);
        ball.time(i) = ball.time(i-1) + time_step;

        %Checks if ball has passed through or hit the floor, and either
        %calls the bounce function or handles the ball bounce manually.
        if ball.y(i) <= 0
                if ball.y(i) < 0
                        [ball.y(i),ball.vy(i)] = ball_bounce_interp(ball.y(i-1), ball.y(i), ball.vy(i-1), ball.vy(i), ...
                        ball.accel_funcy1, time_step, ball.e);
                num_bounces = num_bounces + 1;
                else
                        ball.vy(i) = -ball.vy(i)*e;
                end
        end

        %Computes current x and y acceleration
        if ball.vx(i) >= 0
                ball.ax(i) = ball.accel_funcx1(ball.vx(i));
        else
                ball.ax(i) = ball.accel_funcx2(ball.vx(i));
        end

        if ball.vy(i) >= 0
                ball.ay(i) = ball.accel_funcy1(ball.vy(i));
        else
                ball.ay(i) = ball.accel_funcy2(ball.vy(i));
        end

        i = i + 1; %Index incrementation

    end

end
```

“launch_script.m” Matlab script

```
% Modelling Script for Ball
% Populates fields in a ball structure with given values.
% Calls the launch_ball function and plots relevant results.

g = 9.81;          %gravity (m/s^2)
mass = 0.0027;  %ball mass (kg)
e = 0.8;           %coefficient of restitution (unitless)
time_step = 0.0001; %time_step (seconds)

k = 0.000359;                    %Drag Coefficient (unitless)
initial_displacement = 0.01;     %Initial Spring Displacement (m)
k_stiffness = 167.71;            %Spring Stiffness (N/m)
initial_height = 0.762           %Initial Ball Height (m)

accel_funcx1 = @(v)(-k*v^2);     %x acceleration function assuming vx >= 0
accel_funcx2 = @(v)(k*v^2);      %x acceleration function assuming vx < 0
accel_funcy1 = @(v)(-g-k*v^2);  %y acceleration function assuming vy >= 0
accel_funcy2 = @(v)(-g+k*v^2);  %y acceleration function assuming vy <= 0

%Loads initial conditions for position and y velocity into "ball" structure fields
ball.x(1) = 0;
ball.y(1) = initial_height;
ball.vy(1) = 0;

%Initializes x velocity, and x & y acceleration fields for "ball" structure
ball.vx(1) = 0; %ball.vx(1), ball.ax(1), ball.ay(1) will change later in script
ball.ax(1) = 0;
ball.ay(1) = 0;

%Loads acceleration functions into corresponding fields of "ball" structure
ball.accel_funcx1 = accel_funcx1;
ball.accel_funcx2 = accel_funcx2;
ball.accel_funcy1 = accel_funcy1;
ball.accel_funcy2 = accel_funcy2;

%Loads ball mass, Coeff of Restitution, and initial time into corresponding
%fields of "ball" structure
ball.m = mass;
ball.e = e;
ball.time(1) = 0;
```

```
%launch_ball function runs the simulation with the given inputs and outputs
%another "ball" struct with arrays for each of its x, y, vx, vy, ax, ay, and
%time fields.
ball = launch_ball(ball,k_stiffness, initial_displacement, time_step);

%Usage examples for "ball" struct: ball.x(15) gets ball position at index 15 ball.time(15) gets time at index
15

plot(ball.x,ball.y)
title('y vs. x position')
xlabel('x (m)')
ylabel('y (m)')

figure
plot(ball.time,ball.x)
title('x position vs. time')
xlabel('time (seconds)')
ylabel('x (m)')

figure
plot(ball.time, ball.y)
title('y position vs. time')
xlabel('time (seconds)')
ylabel('y (m)')

figure
plot(ball.time, ball.vx)
title('x velocity vs. time')
xlabel('time (seconds)')
ylabel('x (m)')

figure
plot(ball.time, ball.vy)
title('y velocity vs. time')
xlabel('time (seconds)')
ylabel('y (m)')

figure
plot(ball.time, ball.ax)
title('x acceleration vs. time')
xlabel('time (seconds)')
ylabel('x acceleration (m/s^2)')

figure
plot(ball.time, ball.ay)
title('y acceleration vs. time')
xlabel('time (seconds)')
ylabel('y acceleration (m/s^2)')
```

Appendix 2 - Plots

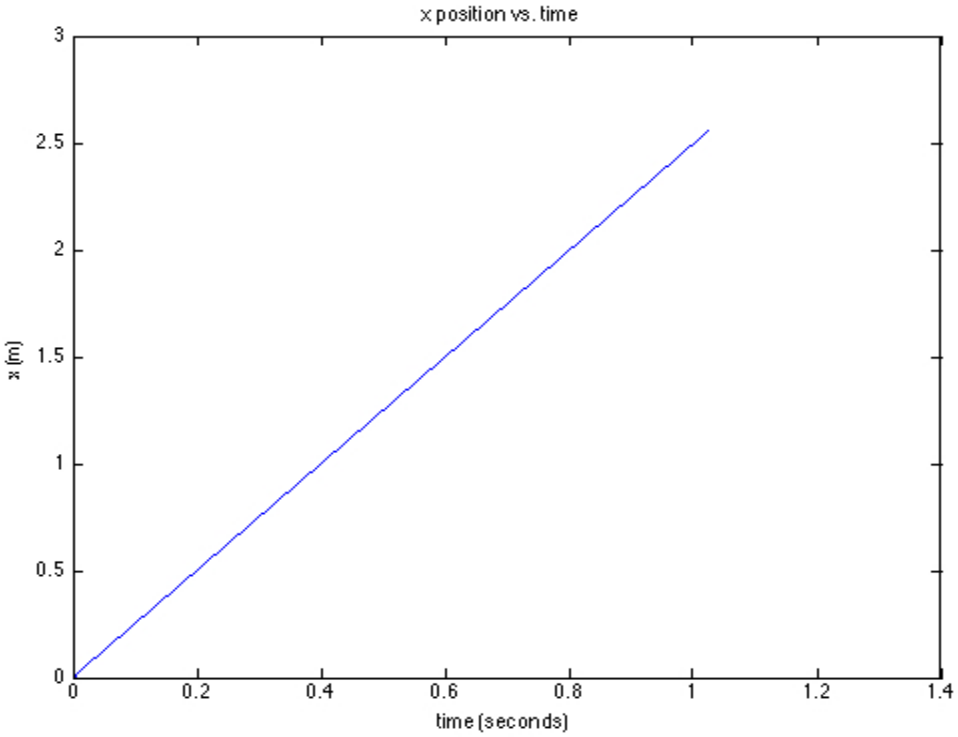


Figure 1. x position vs. time plot for computer simulation

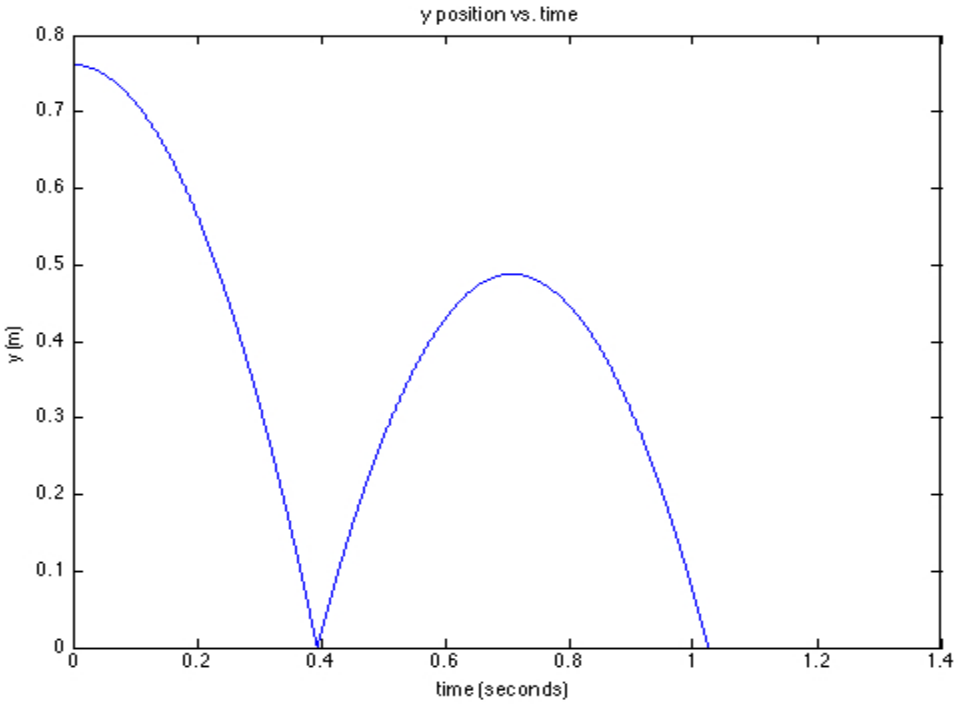


Figure 2. y position vs. time plot for computer simulation

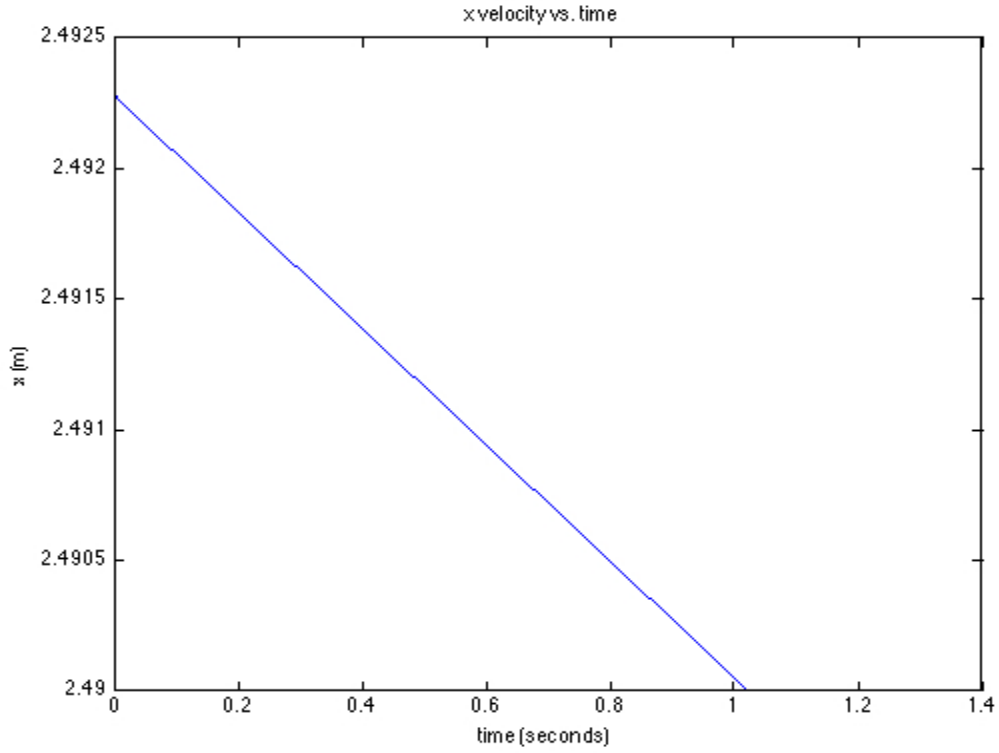


Figure 3. x velocity vs. time plot for computer simulation

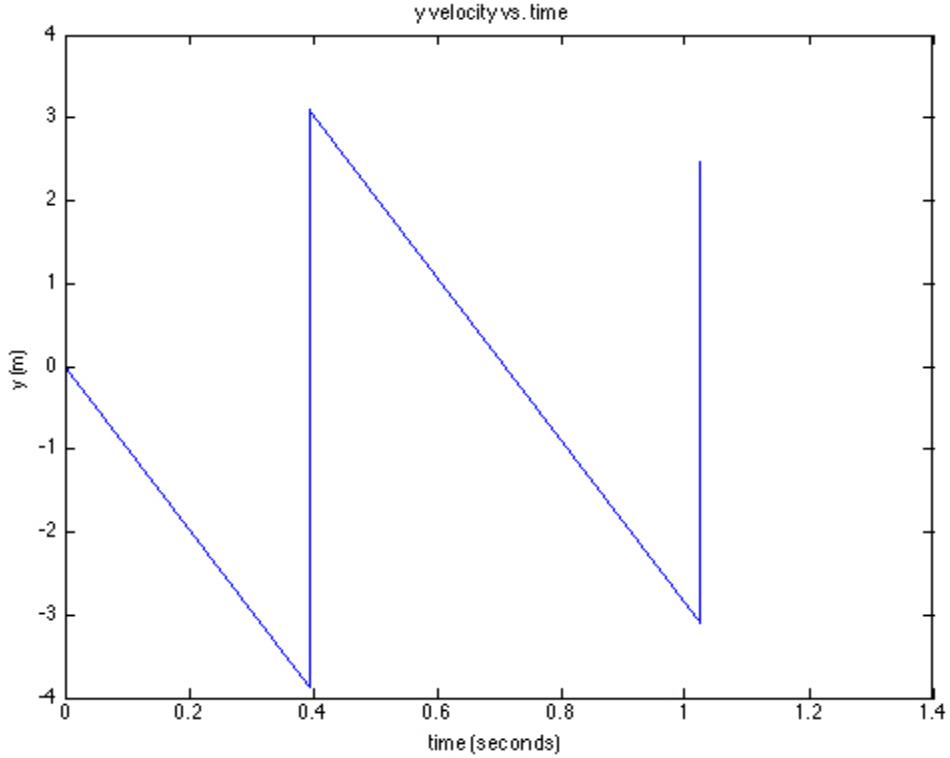


Figure 4. y velocity vs. time plot for computer simulation

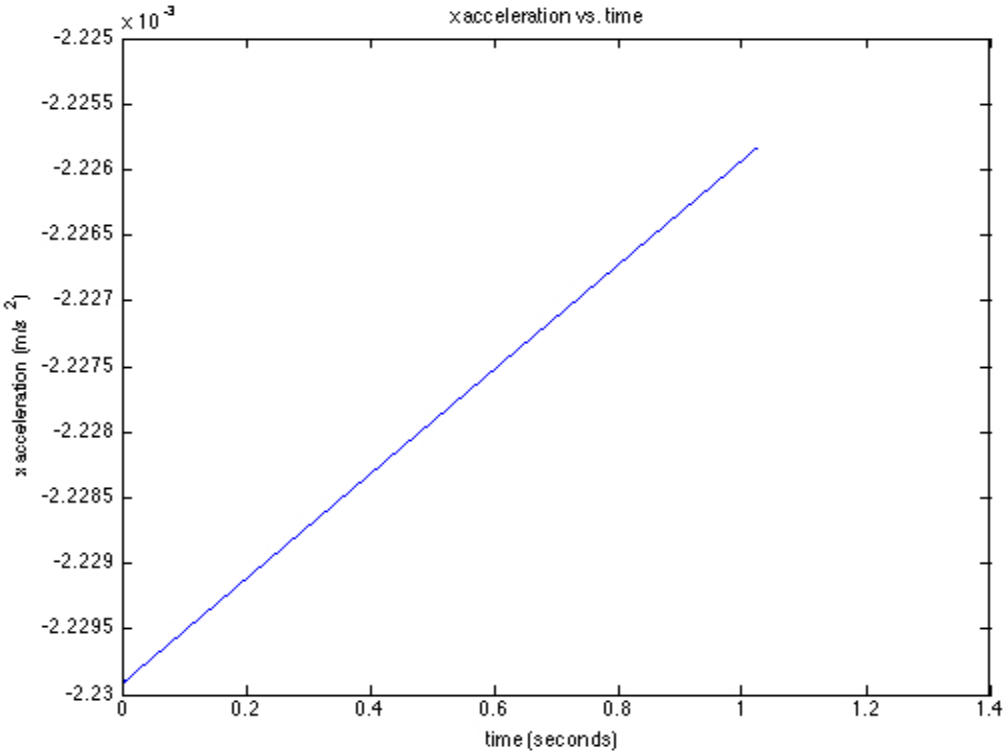


Figure 5. x acceleration vs. time plot for computer simulation

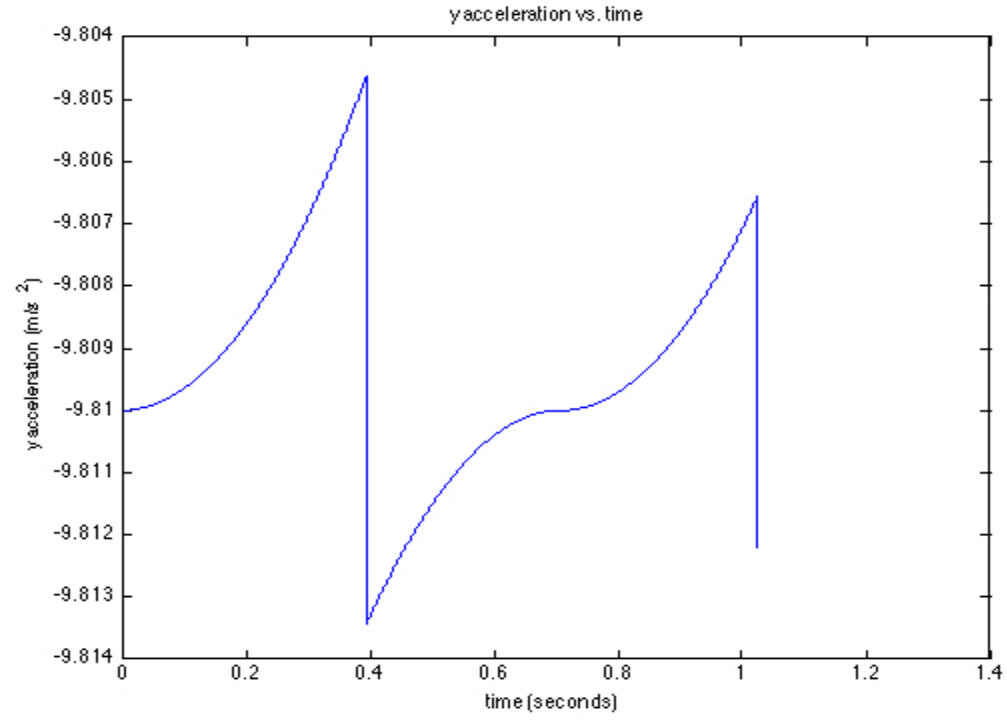


Figure 6. y acceleration vs. time plot for computer simulation

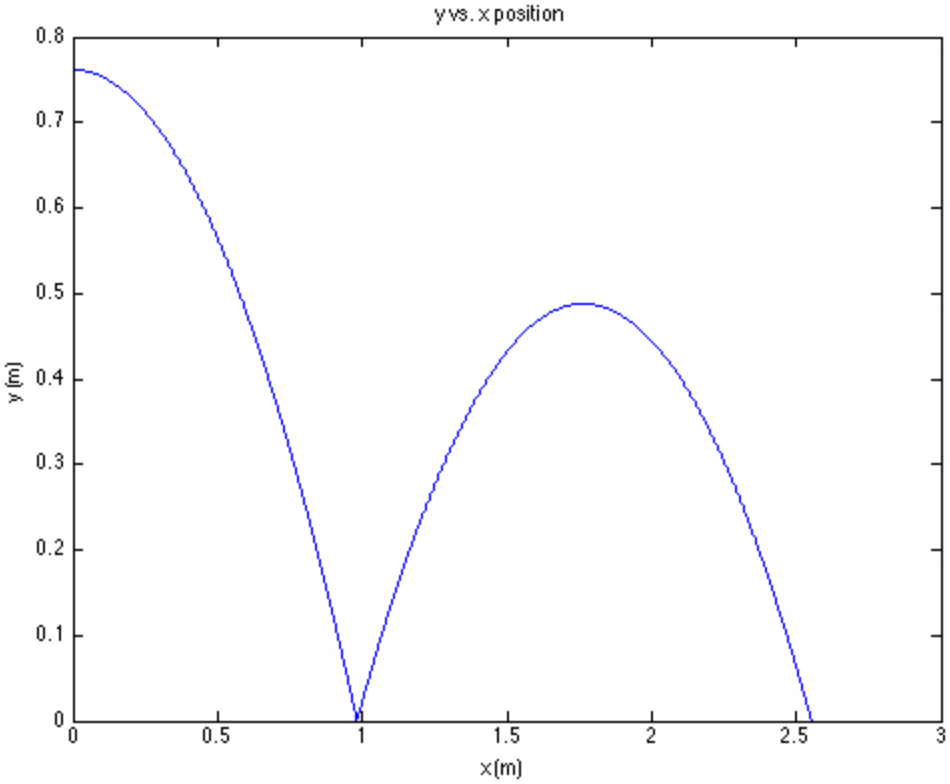


Figure 7. y position vs. x position (trajectory) plot for computer simulation

Appendix 3 - Data Tables

Distance (m)	Displacement (m)	Mass (kg)	Weight (N)
0.06	0	0.05	0.4905
0.063	0.003	0.1	0.981
0.065	0.005	0.15	1.4715
0.068	0.008	0.2	1.962
0.0715	0.0115	0.25	2.4525
0.0725	0.0125	0.3	2.943
0.076	0.016	0.35	3.4335
0.0795	0.0195	0.4	3.924
0.084	0.024	0.45	4.4145
0.086	0.026	0.5	4.905

Table 1. Effective Spring Stiffness Data

Initial Height (in)	Initial Height (ft)	Initial Velocity (ft/s)	Max Bounce Height (in)	Max Bounce Height (ft)	Bounce Velocity (ft/s)
25	2.083333333	11.58303357	16	1.333333333	9.266426855
25	2.083333333	11.58303357	16.5	1.375	9.410100956
25	2.083333333	11.58303357	16	1.333333333	9.266426855
25	2.083333333	11.58303357	16.25	1.354166667	9.338540214
25	2.083333333	11.58303357	16.25	1.354166667	9.338540214
25	2.083333333	11.58303357	16	1.333333333	9.266426855
25	2.083333333	11.58303357	16.25	1.354166667	9.338540214
25	2.083333333	11.58303357	16.25	1.354166667	9.338540214
25	2.083333333	11.58303357	16.25	1.354166667	9.338540214
25	2.083333333	11.58303357	16	1.333333333	9.266426855

Table 2. Coefficient of Restitution Data - Trials on Tabletop

Initial Height (in)	Initial Height (ft)	Initial Velocity (ft/s)	Max Bounce Height (in)	Max Bounce Height (ft)	Bounce Velocity (ft/s)
25	2.083333333	11.58303357	16.25	1.354166667	9.338540214
25	2.083333333	11.58303357	16	1.333333333	9.266426855
25	2.083333333	11.58303357	16	1.333333333	9.266426855
25	2.083333333	11.58303357	16.25	1.354166667	9.338540214
25	2.083333333	11.58303357	16.25	1.354166667	9.338540214
25	2.083333333	11.58303357	16	1.333333333	9.266426855
25	2.083333333	11.58303357	16	1.333333333	9.266426855
25	2.083333333	11.58303357	15.75	1.3125	9.193747876
25	2.083333333	11.58303357	16	1.333333333	9.266426855
25	2.083333333	11.58303357	16	1.333333333	9.266426855

Table 3. Coefficient of Restitution Data - Trials on Floor