Simulation 3: Mass Striking a Barrier

Brandon Reddish

November 1st 2018

1 Introduction

This simulation looks at a mass striking a barrier. The simulation is done by modeling the system with a bond graph and solving the state equations with MATLAB. The system that was modeled can be seen in Figure 1.

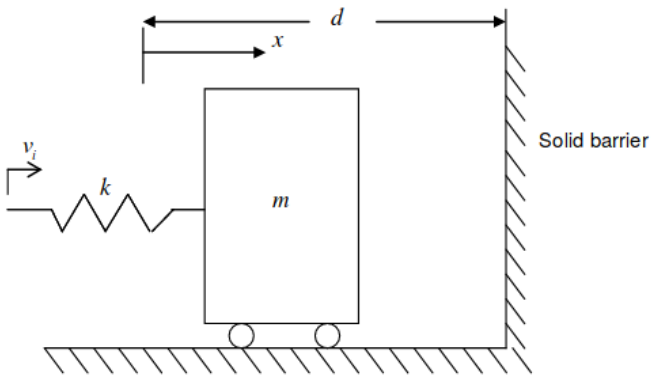

Figure 1: Model of Mass Striking a Barrier

For this example, the input velocity will be modeled as the harmonic frequency ω_i which is equal to the natural frequency $\omega_n = \sqrt{\frac{k}{m}}$ rads/sec. With this input, the response of the mass will continue to grow in the x direction until the barrier is struck.

The first step in the modeling process is to develop a bond graph that represents the system. Figure 2 shows the simplified model and the associated bond graph.

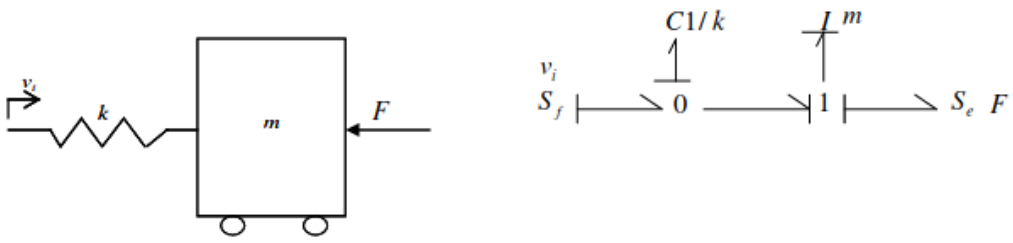

Figure 2: Simplified Model and Bond Graph

Using the bond graph from Figure 2, the state equations shown in Equation 1 and 2 can be used to model the system in MATLAB.

$$\dot{p} = q * k - F(t) \tag{1}$$

$$\dot{q} = v_i - \frac{p}{mass} \tag{2}$$

The simulation will run with these equations, but in order to know where the mass is, the state space must be expanded. This can be done my implementing Equation 3.

$$\dot{x} = \frac{p}{mass} \tag{3}$$

The initial condition for this problem will be 0.25m and the input periodic function will be modeled with Equation 4.

$$v_i = x_i \omega_i cos \omega_i t \tag{4}$$

The mass will be 1 kg and the spring constant will be modeled as $k = \omega_n^2$ with $\omega_n = 2\pi f_n$ and $f_n = 1.0hz$.

The distance to the barrier will be 0.5 meters and when the mass reaches that point, the interaction will begin.

1.1 Force Barrier Model

The first model for the barrier interaction will use an applied force to reverse the momentum of the mass in a very short amount of time. The goal is to have the initial velocity of when the mass hits the barrier be the final velocity in the opposite direction when the mass leaves the barrier. To do this Equation 5 will be used to calculate the force required over a single time step to reverse the momentum and ultimately the velocity.

$$F = \frac{2p_{m0}}{\Delta t} \tag{5}$$

1.2 Stiff Spring Model

The other method used to simulate a barrier is to use a stiff spring model. Figure 3 shows the additional element that will be used to reverse the momentum of the mass.

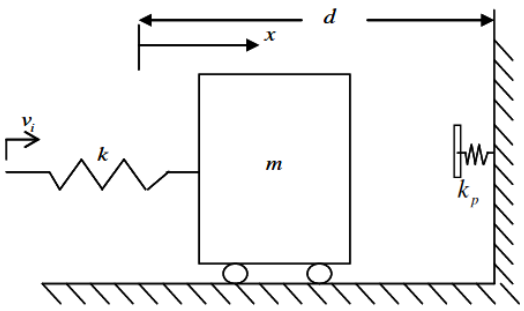

Figure 3: Stiff Spring Barrier

To implement this model, the force term used in the previous section will be replaced with Equation 6.

$$F_{spring} = (x - d) * k_p \tag{6}$$

Plugging F_{spring} into Equation 1 for $F(t)$ will replace the force with a more physical spring element. It is important to note that this equation is only valid when $x \geq d$. For values of x less than d, the value of F_{spring} is equal to zero.

2 Running the Simulation

2.1 Preliminary Run

The first simulation was to runt the code with no input velocity and no wall reaction. This was done to make sure that the model was behaving in a way that made physical sense. Figures 4 and 5 show the system response to an initial offset of 0.25 meters.

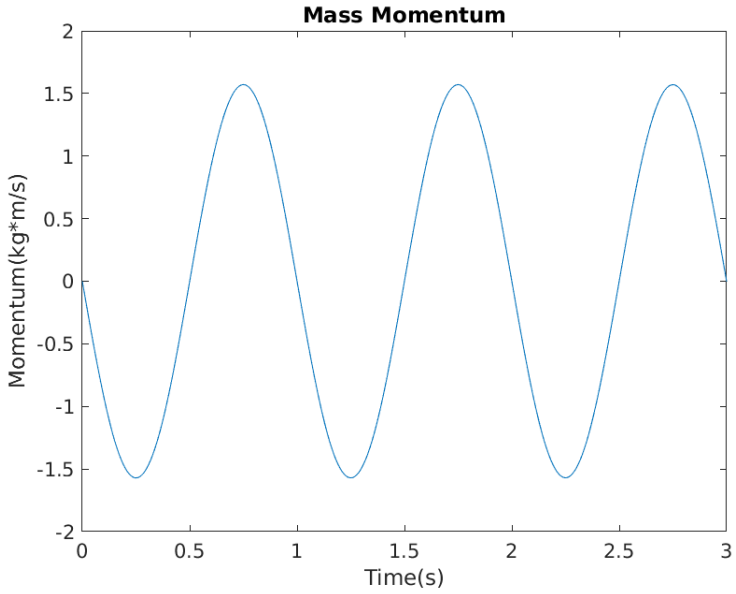

Figure 4: Momentum Response with no Input

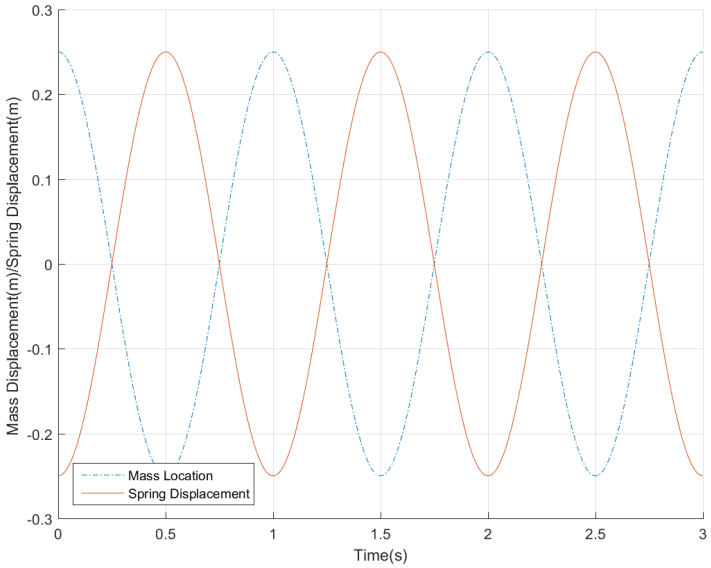

Figure 5: Mass Location Response with no Input

As one can see, the response of the mass is that is osculates about zero to -0.25 and +0.25 meters. Because there is no friction and no input, this correctly models the response of a mass and spring.

2.2 Force Model Results

The next step is to implement the force response of the barrier. Additionally, an input velocity will be applied to the spring element. Figures 6 and 7 show the response of the system when the barrier is modeled as force that reverses the momentum of the mass.

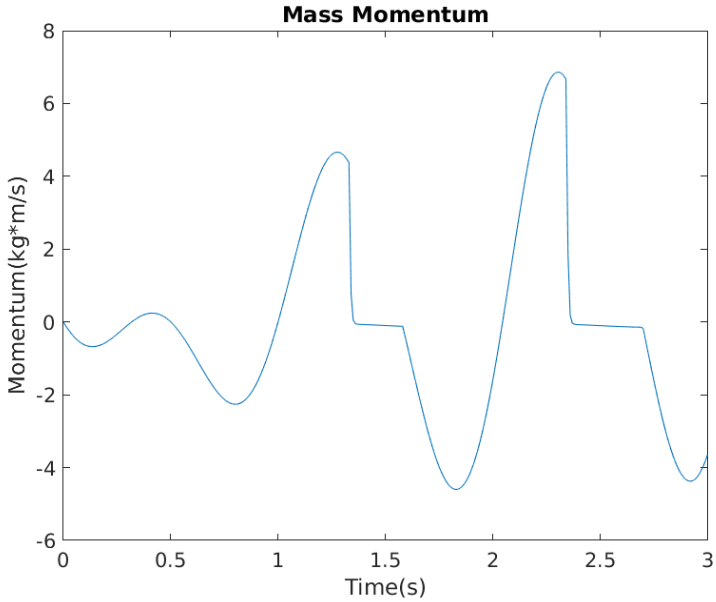

Figure 6: Momentum Response with Force Input

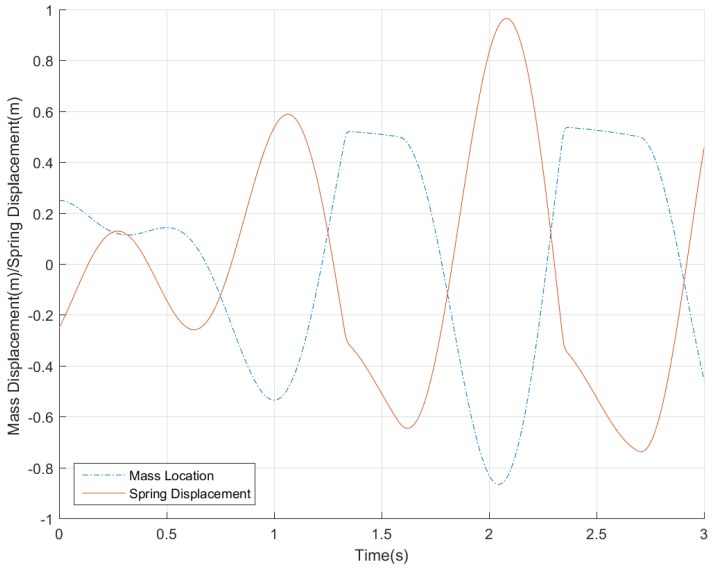

Figure 7: Mass Location Response with Force Input

In this model of the system, the solver was not able to adequately handle the force impulse. The simulation was able to bring the momentum down to zero when the mass reached the wall. This is a fault with the ode45 function and the force equations. In order to get around this, the system was then modeled using an Euler method in C++. Figures 8 and 9 show the result of an instantaneous reversal of momentum when evaluated with a basic Euler numerical method.

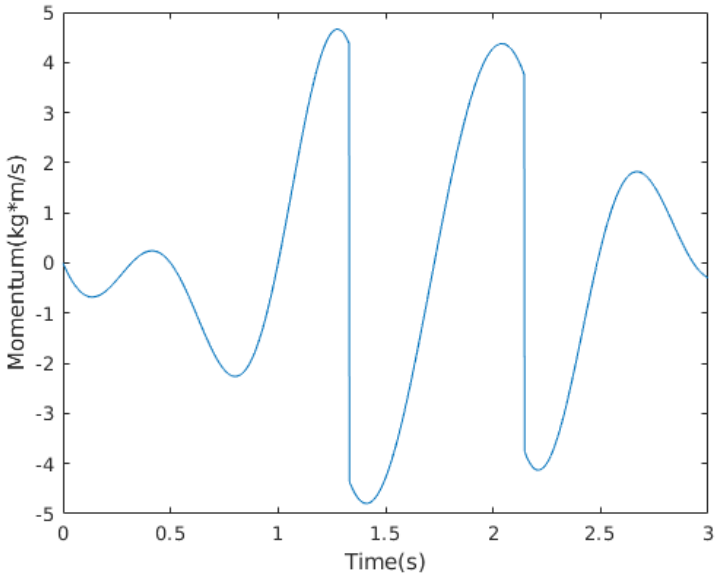

Figure 8: Momentum Response with Force Input Using Euler Solver

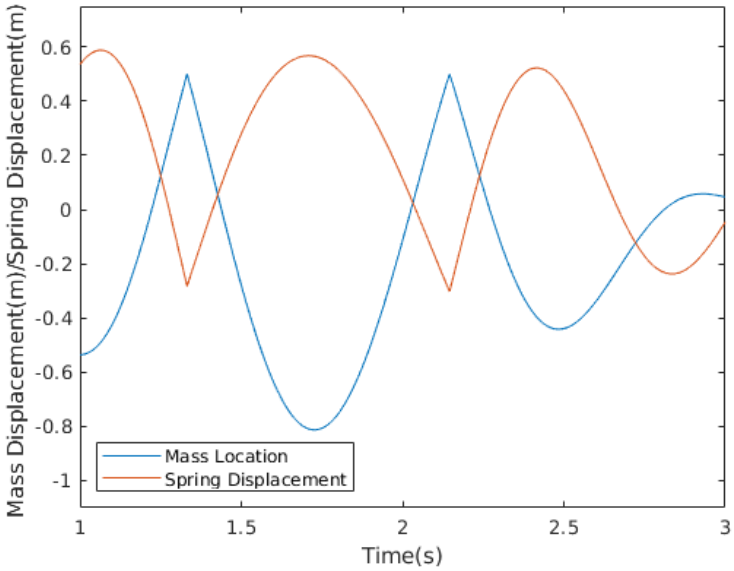

Figure 9: Mass Location Response with Force Input Using Euler Solver

Using the Euler method allowed for the instant change of the momentum state variable which occurs when the mass reaches the barrier.

2.3 Stiff Spring Model Results

Modeling the impact as a stiff spring did produce a more physical result than the impulse force model. In Figure 10, the momentum is a smooth curve that reverses sign over a short period of time.

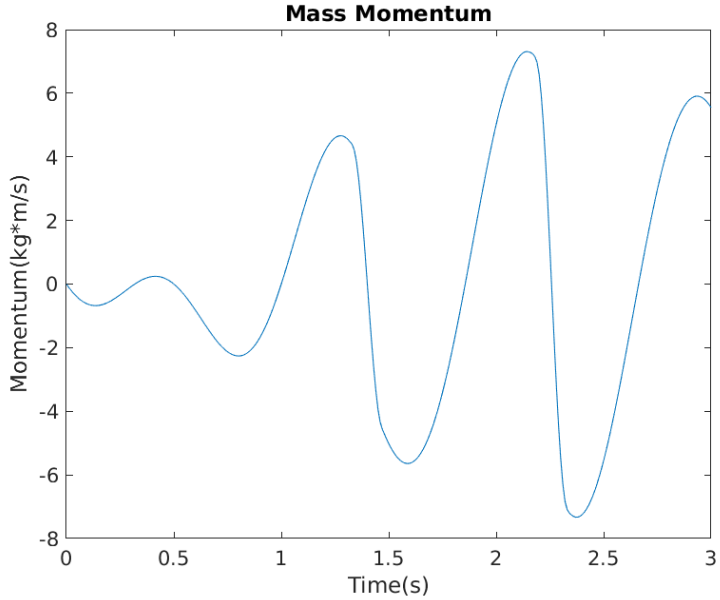

Figure 10: Momentum Response with Stiff Spring

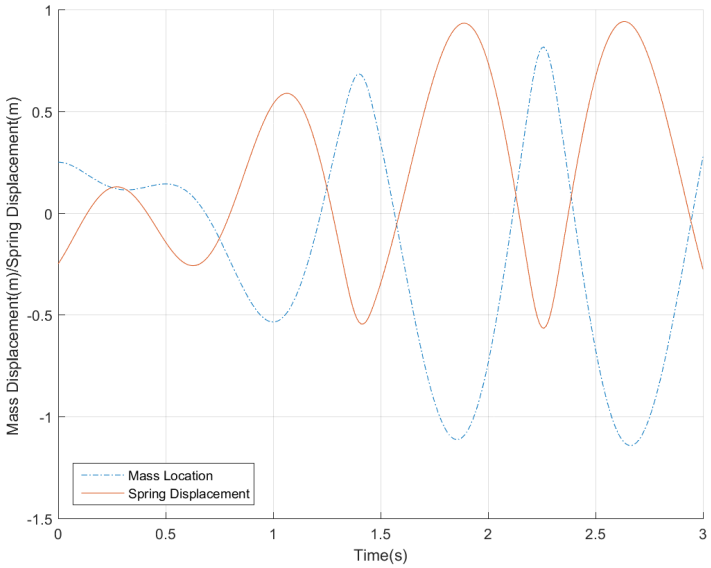

Figure 11: Mass Location Response with Stiff Spring

From the results of the spring model, it is clear that the model is more realistic. The ratio of k_p/k was ten for this model, and the response is slower than that of the force model. Because of the slower response, the distance of the mass went beyond 0.5 m but quickly revered its momentum. A higher ratio of k_p/k would result in a response closer to that of the force impulse model.

3 Source Code

3.1 MATLAB Code

The following code is was used to set up the run and execute the ode45 function.

```
function main()
%% Set parameters
global mass omegan k qi timeStep
close all
% Given values
mass = 1; % kg
fn = 1; % hz
omegan = 2 * pi * fn;
k = mass * omegan * omegan; % spring constant
timeStep = 0.01;
tfinal = 3;
tspan = 0:timeStep:tfinal;
%% Initial Conditions
qi = 0.25; %initial spring displacement
% spring state is positive in compression so when the mass is moved forward
% by 0.25 m then the spring has a state of -0.25 m as it is being elongated
initials = [-qi 0 qi];

%% Run ode
[t,x] = ode45(@equations,tspan,initials);
```

```
% Extract derivatives and other values
for i = 1:length(t)
    [dx(i,:), oth(i,:)] = equations(t(i),x(i,:));
end

%% Post Proc
% Plot momentum
figure(1);
grid on
plot(t(:,1),x(:,2))
title('Mass Momentum')
xlabel('Time(s)')
ylabel('Momentum(kg*m/s)')
print('spring_Momentum','-dpng')

%% Mass location and spring state
figure(2);
grid on
hold on
% plot mass location
plot(t(:,1),x(:,3))
% plot spring state
plot(t(:,1),x(:,1))
legend('Mass Location','Spring Displacement','Location','southwest')
xlabel('Time(s)')
ylabel('Mass Displacement(m)/Spring Displacement(m) ')
print('spring_Location','-dpng')
end
```

The main code calls the equation code shown below. There are multiple cases that this code can run. Various sections of the code are commented out, but when implemented can simulate the system with the force response, the spring response, and without a wall.

```
function [dx,oth] = equations(t,x)
% Global variables
global omegan qi k mass timeStep
% Unpack variables
q    = x(1); % spring state
p    = x(2); % momentum
xLoc = x(3); % mass location
v = qi* omegan* cos(omegan*t); % spring flow input
% v = 0;% if v = 0 the mass would oscilate around zero and up to +- 0.25

%% No wall
% inputForce = 0;

%% Wall force
% if xLoc >=0.5
%     inputForce = 2*p/timeStep;
% else
%     inputForce = 0;
% end

%% Wall spring
if xLoc >0.5
    inputForce = (xLoc-0.5)*k*10;
else
```

```
    inputForce = 0;
end

%% Calculate dp,dq and dxLoc
dp = q * k - inputForce; % change in momentum
dq = v -p/mass; % change in spring state
dxLoc = p/mass; % expanfing state space to get x of mass
dx =[dq;dp;dxLoc];
oth = v;
end
```

3.2 C++ Code

```
#include <iostream>
#include <math.h>
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
//declare variables
float mass = 1; // mass [kg]
float fn = 1;   // natural frequency [hz]
float omegan;   // radial requency  [rads/sec]
float k;        // spring constant

float tstep = 0.00002;// seconds [s]
float tstart = 0;     // initial time [s]
float tfinal = 3;     // final time [s]
int   timeSize = (tfinal-tstart)/tstep; // generate size of time array

float time[timeSize]={}; // time array
float p[timeSize] = {};  //momentum array
float q[timeSize] = {};  //spring state array
float x[timeSize] = {};  // location array
float dp[timeSize] = {}; // rate of change of momentum
float dq[timeSize] = {}; // rate of change of spring state
float dx[timeSize] = {}; // rate of change of mass location

float inputVel; // input velocity
float inputForce; // input force from wall
int flag = 1; // used to make sure the force inpulse happens only once

// calculation of variable
omegan = 2 * 3.1415 * fn;
k = mass * omegan * omegan;
    // initial conditions
p[0] = 0;
q[0] = -0.25;
x[0] = 0.25;
ofstream myFile;
myFile.open("sim3_data.csv");
myFile << "time,momentum,spring state,location\n";
// Euler method
```

```
for (int i=0; i<timeSize; i++)
{
time[i] = i*tstep;
inputVel = x[0] * omegan * cos(omegan*time[i]); // spring input
// Calculate the input force from the wall
    if(x[i]>=.5&flag==1)
{
inputForce = 2*p[i]/tstep;
flag=0;
}
else
{
inputForce = 0;
}
dp[i] = q[i] * k - inputForce;
dq[i] = inputVel - p[i]/mass;
dx[i] = p[i]/mass;
p[i+1]= p[i]+ dp[i]*tstep;
q[i+1]= q[i]+ dq[i]*tstep;
x[i+1]= x[i]+ dx[i]*tstep;
myFile << time[i]<<","<<p[i] << "," << q[i] << "," << x[i] <<"\n";
// Reset flag when mass leaves wall
if(x[i]<.5&flag==0)
{
flag=1;
}
}
// Done
cout << "Done" <<"\n";
return 0;
}
```