Bouncing Ball Simulations with Acceleration and Damping Effects

1 Introduction

A computer simulation is a simulation, run on a computer, or a network of computers, meant to reproduce the behavior of a system. Computer simulations use computer models to simulate the system. A computer model consists of the algorithms and equations used to represent the behavior of the system being modeled. I will be presenting 3 simulations which each have very similar models. A 2-Dimensional simulation of ball bouncing in a walled space, and 2 simulations that go hand in hand. They represent a 3-Dimensional simulation of a ball bouncing in a walled room. One is a plot of the trajectory of the center of the ball, and one is an animation of a ball bouncing. Here are the parameters of the systems:

- 2-Dimensional

$$\begin{cases} \texttt{dimensions of walled space: } 1m \texttt{ x } 1m \\ p = (x, y) \texttt{ (position)} \\ v = (x, y) \texttt{ (velocity)} \\ a = (x, y) \texttt{ (acceleration)} \\ x : \texttt{horizontal component, } y : \texttt{vertical component} \\ r = .05m \texttt{ (radius of the ball)} \\ p(0) = (.5, 1 - r)m \texttt{ (initial position)} \\ v(0) = (.3, 0)\frac{m}{s} \texttt{ (initial velocity)} \\ a = (0, .0981)\frac{m}{s^2} \\ \delta t = .03s \texttt{ time between calculated states of the system} \\ \alpha = .8 \texttt{ (normal damping coefficient)} \\ \beta = .99 \texttt{ (tangential damping coefficient)} \end{cases} \tag{1}$$

- 3-Dimensional

$$\begin{cases} \texttt{dimensions of walled space: } 1m \texttt{ x } 1m \texttt{ x } 1m \\ p = (x, y, z) \texttt{ (position)} \\ v = (x, y, z) \texttt{ (velocity)} \\ a = (x, y, z) \texttt{ (acceleration)} \\ x : \texttt{horizontal component, } y : \texttt{depth component, } z : \texttt{vertical component} \\ r = .05m \texttt{ (radius of the ball)} \\ p(0) = (.5, 1 - r, .5)m \texttt{ (initial position)} \\ v(0) = (.3, .1, .2)\frac{m}{s} \texttt{ (initial velocity)} \\ a = (0, 0, .0981)\frac{m}{s^2} \\ \delta t = .03s \texttt{ time between calculated states of the system} \\ \alpha = .8 \texttt{ (normal damping coefficient)} \\ \beta = .99 \texttt{ (tangential damping coefficient)} \end{cases} \tag{2}$$

2 Models

I simulated 2 different situations described above. As I said, the models for both situations are very similar. Acceleration is constant for both systems so it never has to be adjusted. The simulation runs as follows:

1. Set up initial conditions
2. Draw the first frame
3. Begin the simulation
   - Calculate next position and velocity vectors
   - Clear the previous frame
   - Draw the new frame
4. End simulation and output simulation run time.

The Euler Method is used to calculate consecutive position and velocity vectors in both cases:

$$p_n = p_{n-1} + v_{n-1} \cdot \delta t \tag{3}$$

$$v_n = v_{n-1} + a \cdot \delta t \tag{4}$$

The next position is calculated before calculating the next velocity. When a collision occurs it is handled as such:

- Position
  1. Determine if the ball is out of bounds

$$\texttt{out1} = p > \texttt{lower bounds} \tag{5}$$

$$\texttt{out2} = p < \texttt{upper bounds} \tag{6}$$

where out1 and out2 are logical matrices:

$$\begin{cases} \texttt{2D:out1} = [\ \texttt{logical logical}\ ] \\ \texttt{3D:out1} = [\ \texttt{logical logical logical}\ ] \\ \texttt{logical} = 0\ \texttt{(if ball is beyond lower bounds)} \\ \texttt{logical} = 1\ \texttt{(if ball is in bounds)} \end{cases} \tag{7}$$

$$\begin{cases} \texttt{2D:out2} = [\ \texttt{logical logical}\ ] \\ \texttt{3D:out2} = [\ \texttt{logical logical logical}\ ] \\ \texttt{logical} = 0\ \texttt{(if ball is beyond upper bounds)} \\ \texttt{logical} = 1\ \texttt{(if ball is in bounds)} \end{cases} \tag{8}$$

2. Transform logical matrices into usable matrices:
   - 2D: out1 and out2 look like this after transformation

$$\begin{pmatrix} \texttt{logical} & 0 \\ 0 & \texttt{logical} \end{pmatrix} \tag{9}$$

   - 3D: out1 and out2 look like this after transformation

$$\begin{pmatrix} \texttt{logical} & 0 & 0 \\ 0 & \texttt{logical} & 0 \\ 0 & 0 & \texttt{logical} \end{pmatrix} \tag{10}$$

3. If the ball is beyond the bounds, place ball in bounds touching the wall, else do nothing:

$$p = p \cdot out1 \tag{11}$$

   This will multiply a 1 x 2 matrix by a 2 x 2 matrix producing a new 1 x 2 position matrix or multiply a 1 x 3 matrix by a 3 x 3 matrix producing a new 1 x 3 position matrix. The multiplication sets the position of a component beyond the lower bounds to the minimum position 0 (touching the wall) and keeps in bound components the same.

$$p = p \cdot out2 + \texttt{ upper bounds } \cdot \texttt{ !}out2 \tag{12}$$

   !out2 is the logical complement of out2. This will multiply a 1 x 2 matrix by a 2 x 2 matrix producing a new 1 x 2 position matrix or multiply a 1 x 3 matrix by a 3 x 3 matrix producing a new 1 x 3 position matrix. The multiplication sets the position of a component beyond the upper bounds to the minimum position 0 and keeps in bound components the same. Then the addition will add the maximum position to the components that are beyond the upper bounds. When the calculation is complete the out of bounds components will be touching the upper bounds and the in bound indexes of the position matrix will have remained the same.

- Velocity

1. Determine if the ball is out of bounds

$$\texttt{out1} = p > \texttt{lower bounds} \tag{13}$$

$$\texttt{out2} = p < \texttt{upper bounds} \tag{14}$$

   where out1 and out2 are logical matrices:

$$\begin{cases} \texttt{2D:out1} = [\ \texttt{logical logical}\ ] \\ \texttt{3D:out1} = [\ \texttt{logical logical logical}\ ] \\ \texttt{logical} = 0\ \texttt{(if ball is beyond lower bounds)} \\ \texttt{logical} = 1\ \texttt{(if ball is in bounds)} \end{cases} \tag{15}$$

$$\begin{cases} \texttt{2D:out2} = [\ \texttt{logical logical}\ ] \\ \texttt{3D:out2} = [\ \texttt{logical logical logical}\ ] \\ \texttt{logical} = 0\ \texttt{(if ball is beyond upper bounds)} \\ \texttt{logical} = 1\ \texttt{(if ball is in bounds)} \end{cases} \tag{16}$$

These logical matrices also gain additional meaning for velocity collision calculations. A logical value of 1 corresponds to a velocity component that is tangential to the wall during collision. A logical value of 0 corresponds to a velocity component that is normal to the wall during collision

2. If the ball is beyond or touching the bounds, reverse the velocity components normal to the wall during the collision and multiply the velocity vector by the correct damping coefficients.
   - Make out1 and out2 usable:
   - 2D: out1 and out2 look like this after first transformation

$$\begin{pmatrix} \texttt{logical} & 0 \\ 0 & \texttt{logical} \end{pmatrix} \tag{17}$$

   - 3D: out1 and out2 look like this after first transformation

$$\begin{pmatrix} \texttt{logical} & 0 & 0 \\ 0 & \texttt{logical} & 0 \\ 0 & 0 & \texttt{logical} \end{pmatrix} \tag{18}$$

   - Further transformation is required:

$$\texttt{out1} = (\texttt{out1} * (\texttt{normal damp + tangential damp})) - \ \texttt{normal damp} \tag{19}$$

$$\texttt{out2} = (\texttt{out2} * (\texttt{normal damp + tangential damp})) - \ \texttt{normal damp} \tag{20}$$

     Where the damping coefficients are scalars and all elements of the logical matrices have the operations of the scalars performed on them.
   - 2D: out1 and out2 will look like this after second transformation

$$\begin{pmatrix} \texttt{damping coefficient} & 0 \\ 0 & \texttt{damping coefficient} \end{pmatrix} \tag{21}$$

   - 3D: out1 and out2 will look like this after second transformation

$$\begin{pmatrix} \texttt{damping coefficient} & 0 & 0 \\ 0 & \texttt{damping coefficient} & 0 \\ 0 & 0 & \texttt{damping coefficient} \end{pmatrix} \tag{22}$$

   - Final operation:

$$v = v \cdot out1\ \texttt{iff collision occurs with a lower bound(s)} \tag{23}$$

$$v = v \cdot out2\ \texttt{iff collision occurs with an upper bound(s)} \tag{24}$$

This will multiply a 1 x 2 matrix by a 2 x 2 matrix producing a new 1 x 2 velocity matrix or multiply a 1 x 3 matrix by a 3 x 3 matrix producing a new 1 x 3 velocity matrix. The multiplication multiplies the velocity components by the damping coefficients they should be scaled by for a given collision and reverses the normal velocity components. The matrix transformations and final velocity multiplication occurs if and only if there is a collision. Else, no transformations occur and the velocity is unchanged after Euler's scheme is performed on it.

The damping coefficients come into effect when the ball comes into contact with a wall. The collision is not elastic so the magnitude of the velocities after hitting the wall are equal to the magnitude of the velocities hitting the wall multiplied by the damping coefficients. When a velocity component is normal to the wall during a collision it is multiplied by the normal damping coefficient and when a component is tangential to the wall it is multiplied by the tangential damping coefficient.

2.1 My Code

In my code I followed these models almost exactly. The only difference is in my code there are minor adjustments made for position depending on the shape/object representing the ball. For example, in the 2D simulation, I use a curved rectangle to represent the ball so collision calculations have to take into account a rectangle uses its lower left two coordinates as its position. In the 3D simulation I use the DrawSphere() method the professor provided us to represent the ball, so the radius of the sphere has to be accounted for in collision calculations.

***Note: The simulation does not implement a solution to the Zeno Phenomenon, so the ball will bounce very slightly forever.

3 Results

In `MatLab`, my simulation code is invoked at the prompt by calling the function `BouncingBall( timeStep, simLength1, simLength2, simLength3, windowWidth, windowHeight )`:

- timeStep: δt `(value for homework is .03)`

- simLength1: Length of time in seconds the 2D simulation should simulate for (recommended value is 50)

- simLength2: Length of time in seconds the 3D trajectory plot should simulate for (recommended value is 42)

- simLength3: Length of time in seconds the 3D simulation should simulate for (recommended value is 42)

- windowWidth, windowHeight: The desired width and height of the window the simulation is animated in

The command

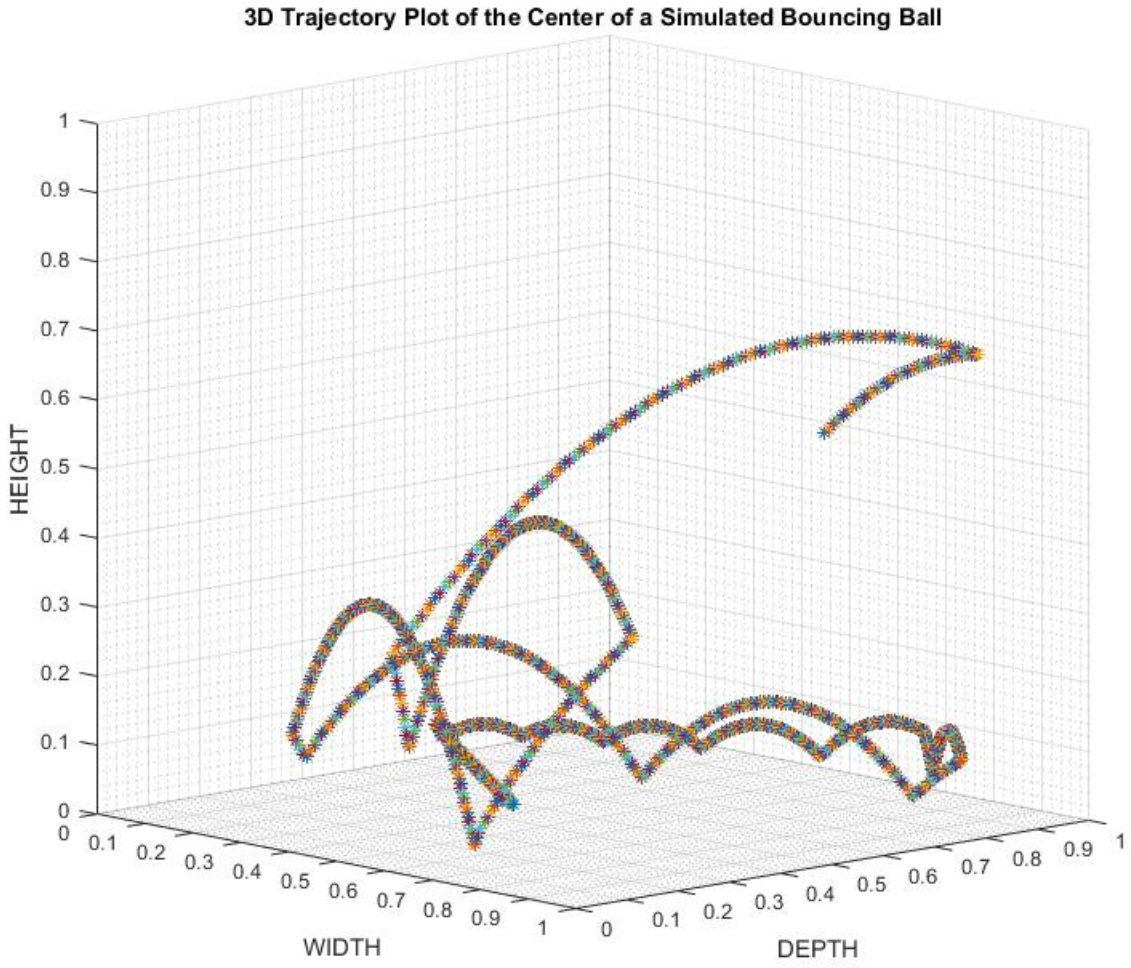
```
>> BouncingBall(.03, 50, 42, 42, 700, 700)
```

will run the 3 simulations back to back:

1. 2D simulation of the ball with initial conditions in the homework
2. 3D trajectory plot of the center of the ball being simulated with initial conditions in the homework
3. 3D simulation of the ball with initial conditions in the homework

The reason for including the trajectory plot is because although the 3D simulation is accurate, it does not look like it is, because the sphere does not shrink as it moves farther back in the room, and does not change shades from shadowing either. In other words it is hard to tell exactly what direction the ball is moving since depth cannot be visualized. This plot shows the path the center of the sphere in the 3D simulation takes. Once the simulation has been run for the desired simLength, the plot can be rotated in MatLab to visualize the path of the ball better.

Here is the 3D trajectory plot:



4 Conclusion

There are limitations to my code. I have created a simulation engine that can simulate many similar systems to the one simulated for the homework. With my code I can increase the size of the rooms, change the initial conditions, including the size of the ball, change the acceleration vector, change the time step, and pass different function handles to calculate subsequent position and velocity vectors all by simply adjusting the parameters of my Movie()

functions.

My collision system is very rudimentary. It simply places a ball that is out of bounds in bounds and does not return any special values adjusting the time step for the calculation. For this reason, if the initial velocity or acceleration is too high, the ball will not move because in one time step, it will traverse the space and simply be reset to a position in bounds over and over, never moving. This is also due to the low order of accuracy of Euler's method. This behavior is also affected by the time step. If the time step is lowered as much as the acceleration and initial velocity are increased, the odd behavior mentioned above will not occur, but the simulation will be very slow on an average PC.

Lastly, I came across the Zeno Phenomenon for the first time while creating this simulation. Due to time constraints, I was not able to implement a solution to the Zeno Phenomenon. So my simulation keeps running forever with the ball bouncing indefinitely very slightly. To combat this problem, I simply have the user specify the time the simulation should be simulated for and end the simulation after that time has been reached. Therefore, my simulation is only accurate up until the Zeno Phenomenon is observed. The recommended simLength times I gave run the simulation up until just after the Zeno Phenomenon begins to be observed, at which point in real life, the ball would most likely come to a stop.

A Implementation in MatLab

The MatLab implementation with comments is given here:

BouncingBall.m:

```
function BouncingBall( timeStep, simLength1, simLength2, simLength3, ...
                       windowWidth, windowHeight )

%** FOR PURPOSE OF THE HOMEWORK CALL: BouncingBall(.03, 50, 42, 42, 700, 700)***
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%BouncingBall Runs different simulations of a bouncing ball
%   Arguments:
%       timeStep: The time step for which the simulations should be run at. The
%       smaller the value here, the more accurate the simulation will be. The
%       smaller the value gets however the longer the run time of the simulation
%       will be.
%
%       simLength1,simLength2,simLength3: The run time for various simulations.
%       The first is for the run time of the 2D simulation, the second is for
%       the run time of the 3D trajectory simulation, and the third one is for
%       the 3D simulation. If set to 0 or anything less the corresponding
%       simulation will not play.
%
%       windowWidth,windowHeight: The width and height of the window the
%       animation is contained in respectively
%       ***** This will affect the size of the animation as well, but
%       the axes will always be equal length so having a longer width will not
%       make the animation wider if the width of the window exceeds the height.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    %Setting up variables to be consistent with the assignment
    roomWidth = 1.0;
    roomHeight = 1.0;
    roomDepth = 0;
    initVelocity = [0.3, 0];
    ballRadius = .05;
    initPosition = [.5, 1-ballRadius];
    dampingMatrix = [.8, .99];
    gravity = [0, -.0981];

    %Run 2D simulation
    if ( simLength1 > 0 )
```

```
        BouncyBallMovie(@EulersVelocity, @EulersPosition, initPosition, ...
                        initVelocity, gravity, timeStep, dampingMatrix, ...
                        ballRadius, roomWidth, roomHeight, roomDepth, ...
                        simLength1, windowWidth, windowHeight);
    end

    %Setting up variables to be consistent with the assignment
    roomWidth = 1.0;
    roomHeight = 1.0;
    roomDepth = 1.0;
    initVelocity = [0.3,0.1,0.2];
    ballRadius = .05;
    initPosition = [.5,1-ballRadius,.5];
        dampingMatrix = [.8, .99];
    gravity = [0,0, -.0981];

    %Run 3D trajectory simulation
    if ( simLength2 > 0 )
        BouncyBallMovie3DPlot(@EulersVelocity, @EulersPosition, initPosition,...
                        initVelocity, gravity, timeStep, dampingMatrix, ...
                        ballRadius, roomWidth, roomHeight, roomDepth, ...
                        simLength2, windowWidth, windowHeight);
    end

    %Run 3D simulation
    if ( simLength3 > 0 )
        BouncyBallMovie3D(@EulersVelocity, @EulersPosition, initPosition, ...
                        initVelocity, gravity, timeStep, dampingMatrix, ...
                        ballRadius, roomWidth, roomHeight, roomDepth, ...
                        simLength3, windowWidth, windowHeight);
    end
end
```

EulersVelocity.m:

```
function velocity = EulersVelocity ( position, initVelocity, acceleration, ...
                                     timeStep, bounds0, boundsMax, ...
                                     ballRadius, normalDamp, tangentialDamp )
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%EulersVelocity Calculate subsequent velocity of the ball using Euler's Method
%   Arguments:
%       position: 2D or 3D position vector of the ball. The vector is a
%       1 x 2 vector as such - [x-position y-position] where y is the vertical
%       component and x is the horizontal component or a 1 x 3 vector as such -
%       [x-position y-position z-position] where z is the vertical component,
%       y is the depth component, and x is the horizontal component. Units of
%       the vector are in meters. Used for collision calculations
%
%       initVelocity: 2D or 3D velocity vector of the ball. The vector is a
%       1 x 2 vector as such - [x-velocity y-velocity] where y is the vertical
%       component and x is the horizontal component or a 1 x 3 vector as such -
%       [x-velocity y-velocity z-velocity] where z is the vertical component,
%       y is the depth component, and x is the horizontal component. Units of
%       the vector are in meters per second.
%
%       acceleration: 2D or 3D acceleration vector of the ball. The vector is a
%       1 x 2 vector as such - [x-acceleration y-acceleration] where y is the
%       vertical component and x is the horizontal component or a 1 x 3 vector
%       as such - [x-acceleration y-acceleration z-acceleration] where z is the
%       vertical component, y is the depth component, and x is the horizontal
%       component. For this simulation, the acceleration is constant. Units of
%       the vector are in meters per second^2.
%
%       timeStep: The time interval between successive position and velocity
%       calculations.
%
%       bounds0: The lower bounds of the system to be used in collision
%       calculations
%
%       boundsMax: The upper bounds of the system to be used in collision
%       calculations
%
%       ballRadius: The radius of the ball to be used in collision calculations
%
%       normalDamp: The damping coefficient of the velocity component normal to
%       the wall during a collision. Used in collision calculations
```

```
%
%       tangentialDamp: The damping coefficient of the velocity component
%       tangential to the wall during a collision. Used in collsion calculations
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    %CALCULATE THE NEXT VELOCITY OF THE BALL
    velocity = initVelocity + acceleration * timeStep;

    %CHECK AND ADJUST FOR COLLISIONS. THIS IS A VERY SIMPLE COLLISION HANDLER.
    %IF THERE IS A COLLISION DETERMINE WHICH VELOCITY COMPONENTS ARE NORMAL TO
    %THE WALL AND MULTIPLY THEM BY -normalDamp AND DETERMINE WHICH VELOCITY
    %COMPONENTS ARE TANGENTIAL TO THE WALL AND MULTIPLY THEM BY tangentialDamp.

    %2 DIMENSIONAL CHECK
    if (length(position) == 2)

        %DETERMINE IF PART OF THE BALL IS OUTSIDE OF THE LOWER BOUNDS OF ROOM
        out1 = position > bounds0;

        %DETERMINE IF PART OF THE BALL IS BEYOND THE UPPER BOUNDS OF THE ROOM
        out2 = position + 2*ballRadius < boundsMax;

      %IF THERE IS A COLLISION DETERMINE WHICH VELOCITY COMPONENTS ARE NORMAL TO
      %THE WALL AND MULTIPLY THEM BY -normalDamp AND DETERMINE WHICH VELOCITY
      %COMPONENTS ARE TANGENTIAL TO THE WALL AND MULTIPLY THEM BY tangentialDamp

        if (out1*[1;1]<2)
            out1 = (out1 * (normalDamp+tangentialDamp)) - normalDamp;
            out1 = [out1*[1;0],0;0,out1*[0;1]];

            velocity = velocity * out1;
        end

        if (out2*[1;1]<2)
            out2 = (out2 * (normalDamp+tangentialDamp)) - normalDamp;
            out2 = [out2*[1;0],0;0,out2*[0;1]];

            velocity = velocity * out2;
        end

    %3 DIMENSIONAL CHECK
    else
```

```
        %DETERMINE IF PART OF THE BALL IS OUTSIDE OF THE LOWER BOUNDS OF ROOM
        out1 = position - ballRadius > bounds0;

        %DETERMINE IF PART OF THE BALL IS BEYOND THE UPPER BOUNDS OF THE ROOM
        out2 = position + ballRadius < boundsMax;

      %IF THERE IS A COLLISION DETERMINE WHICH VELOCITY COMPONENTS ARE NORMAL TO
      %THE WALL AND MULTIPLY THEM BY -normalDamp AND DETERMINE WHICH VELOCITY
      %COMPONENTS ARE TANGENTIAL TO THE WALL AND MULTIPLY THEM BY tangentialDamp
        if (out1*[1;1;1]<3)
            out1 = (out1 * (normalDamp+tangentialDamp)) - normalDamp;
            out1 = [out1*[1;0;0],0,0;0,out1*[0;1;0],0;0,0,out1*[0;0;1]];

            velocity = velocity * out1;
        end

        if (out2*[1;1;1]<3)
            out2 = (out2 * (normalDamp+tangentialDamp)) - normalDamp;
            out2 = [out2*[1;0;0],0,0;0,out2*[0;1;0],0;0,0,out2*[0;0;1]];

            velocity = velocity * out2;
        end
    end

end
```

EulersPosition.m:

```
function position = EulersPosition( initPosition, velocity, timeStep, ...
                                    bounds0, boundsMax, ballRadius )
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%EulersPosition Calculate subsequent position of the ball using Euler's Method
%   Arguments:
%       initPosition: 2D or 3D position vector of the ball. The vector is a
%       1 x 2 vector as such - [x-position y-position] where y is the vertical
%       component and x is the horizontal component or a 1 x 3 vector as such -
%       [x-position y-position z-position] where z is the vertical component,
%       y is the depth component, and x is the horizontal component. Units of
%       the vector are in meters.
%
%       velocity: 2D or 3D velocity vector of the ball. The vector is a
%       1 x 2 vector as such - [x-velocity y-velocity] where y is the vertical
%       component and x is the horizontal component or a 1 x 3 vector as such -
%       [x-velocity y-velocity z-velocity] where z is the vertical component,
%       y is the depth component, and x is the horizontal component. Units of
%       the vector are in meters per second.
%
%       timeStep: The time interval between successive position and velocity
%       calculations.
%
%       bounds0: The lower bounds of the system to be used in collision
%       calculations
%
%       boundsMax: The upper bounds of the system to be used in collision
%       calculations
%
%       ballRadius: The radius of the ball to be used in collision calculations
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    %CALCULATE THE NEXT POSITION OF THE BALL
    position = initPosition + velocity * timeStep;

    %CHECK AND ADJUST FOR COLLISIONS. THIS IS A VERY SIMPLE COLLISION HANDLER
    %THAT SIMPLY RESETS THE POSITION IN BOUNDS IF IT IS OUT OF BOUNDS

    %2 DIMENSIONAL CHECK MADE FOR A CURVED RECTANGLE CREATED BY RECTANGLE()
    if (length(position) == 2)

        %DETERMINE IF PART OF THE BALL IS OUTSIDE OF THE LOWER BOUNDS OF ROOM
```

```
        out1 = position > bounds0;

        %DETERMINE IF PART OF THE BALL IS BEYOND THE UPPER BOUNDS OF THE ROOM
        out2 = (position + 2*ballRadius) < boundsMax;

        %ADJUST FOR PARTS OF THE BALL OUTSIDE OF THE LOWER BOUNDS OF THE ROOM
        position = position * [out1*[1;0],0;0,out1*[0;1]];

        %ADJUST FOR PARTS OF THE BALL BEYOND THE UPPER BOUNDS OF THE ROOM
        position = position * [out2*[1;0],0;0,out2*[0;1]] + ...
             [(boundsMax*[1;0] - 2*ballRadius)*(~out2*[1;0]),...
              (boundsMax*[0;1] - 2*ballRadius)*(~out2*[0;1])];

    %3 DIMENSIONAL CHECK FOR A SPHERE MADE WITH Draw_Sphere()
    else

        %DETERMINE IF PART OF THE BALL IS OUTSIDE OF THE LOWER BOUNDS OF ROOM
        out1 = position - ballRadius > bounds0;

        %DETERMINE IF PART OF THE BALL IS BEYOND THE UPPER BOUNDS OF THE ROOM
        out2 = (position + ballRadius) < boundsMax;

        %ADJUST FOR PARTS OF THE BALL OUTSIDE OF THE LOWER BOUNDS OF THE ROOM
        position=position*[out1*[1;0;0],0,0;0,out1*[0;1;0],0;0,0,out1*[0;0;1]]...
                             + ballRadius*(~out1);

        %ADJUST FOR PARTS OF THE BALL BEYOND THE UPPER BOUNDS OF THE ROOM
        position=position*[out2*[1;0;0],0,0;0,out2*[0;1;0],0;0,0,out2*[0;0;1]]+...
            [(boundsMax*[1;0;0] - ballRadius)*(~out2*[1;0;0]),...
             (boundsMax*[0;1;0] - ballRadius)*(~out2*[0;1;0]),...
             (boundsMax*[0;0;1] - ballRadius)*(~out2*[0;0;1])];

    end
end
```

BouncyBallMovie.m:

```
function BouncyBallMovie(cVelocity, cPosition, initPosition, ...
                         initVelocity, acceleration, timeStep, dampingMatrix,...
                         ballRadius, roomWidth, roomHeight, roomDepth, ...
                         simLength, windowWidth, windowHeight)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%BouncyBallMovie Plays a movie of a 2D ball bouncing in a walled 2D space
%   Arguments:
%       cVelocity: Calculates velocity for the next step, can work anyway you
%       want as long as it takes a position vector, an initial velocity vector,
%       an acceleration vector, a time step, the lower and upper bounds of the
%       2D space, the ball radius, the normal damping force, and the tangential
%       damping force. This function needs so many arguments because it must
%       also take care of the cases when the ball hits a wall.
%
%       cPosition: Calculates postion for the next step, can work anyway you
%       want as long as it takes an initial position vector, a velocity vector,
%       a time step, the lower and upper bounds of the 2D space, and the ball
%       radius. This function also take care of the cases when the ball hits a
%       wall.
%       ****cPosition and cVelocity should coordinate to take care of the case
%           when the ball hits the wall.
%
%       initPosition: Initial 2D position vector of the ball. The vector is a
%       1 x 2 vector as such - [x-position y-position] where y is the vertical
%       component and x is the horizontal component. Units of the vector are in
%       meters.
%
%       initVelocity: Initial 2D velocity vector of the ball. The vector is a
%       1 x 2 vector as such - [x-velocity y-velocity] where y is the vertical
%       component and x is the horizontal component. Units of the vector are in
%       meters per second.
%
%       acceleration: The acceleration vector for the system. The vector is a
%       1 x 2 vector as such - [x-acceleration y-acceleration] where y is the
%       vertical component and x is the horizontal component. For this
%       simulation, the acceleration is constant. Units of the vector are in
%       meters per second^2.
%
%       timeStep: The time interval between successive position and velocity
%       calculations.
%
```

```
%       dampingMatrix: The collision damping maxtrix for the system. The matrix
%       is a 1 x 2 vector as such - [alpha beta] where alpha is the collision
%       damping coefficient of the velocity component normal to the wall and
%       beta is the collision damping coefficient of the velocity component
%       tangential to the wall.
%
%       ballRadius: The radius of the ball in meters.
%
%       roomWidth, roomHeight, roomDepth: The width of the room in meters, the
%       height of the room in meters, and the depth of the room in meters
%       respectively.
%       ***** Note that for a 2D space the depth should always be zero.
%             Also note that the axes of the animation will always make a square
%             so if the height and width are not the same, the ball will look
%             like an ellipse.
%
%       simLength: Time in seconds the system should be simulated for.
%       ***** This is important for this function because the Zeno Phenomenon is
%       not accounted for here so without a duration for the simulation, it may
%       run forever.
%
%       windowWidth,windowHeight: The width and height of the window the
%       animation is contained in respectively
%       ***** This will affect the size of the animation as well, but remember
%       the axes will always make a square so having a longer width will not
%       make the animation wider if the width of the window exceeds the height.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    program_start=tic(); %STARTS THE CLOCK FOR THE RUN TIME OF THE PROGRAM

    %ADJUST INITIAL POSITION FOR RECTANGLE() SO BALL APPEARS IN CORRECT PLACE
    initPosition = initPosition - ballRadius;

    %SET UP THE DAMPING COEFFICIENTS AS SEPERATE VARIABLES FOR EASIER
    normalDamp = dampingMatrix * [1;0];                    %CALCULATION LATER ON
    tangentialDamp = dampingMatrix * [0;1];

    %SET UP LOWER AND UPPER BOUNDS FOR FUTURE CALCULATIONS
    bounds0 = [0, 0];
    boundsMax = [roomWidth, roomHeight];

    %SET TIME VARIABLE TO 0. TO BE USED TO SIMULATE PROGRAM FOR DESIRED
```

```
    t = 0;                                                          %LENGTH OF TIME

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% DRAWING FIRST FRAME %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %DRAW BALL IN INITAL POSITION
    figure('position', [250, 50, windowWidth, windowHeight])
    title('A 2D Simulation of a Bouncing Ball');
    rectangle('Position',[initPosition 2*ballRadius 2*ballRadius],...
              'Curvature',[1,1],...
              'FaceColor','r');

    %CREATING AXES
    axis([0 roomWidth 0 roomHeight]);
    axis('square');
    axis('on');
    grid on;
    grid minor;
    xticks([0 .1*roomWidth .2*roomWidth .3*roomWidth .4*roomWidth ...
        .5*roomWidth .6*roomWidth .7*roomWidth .8*roomWidth .9*roomWidth ...
        roomWidth]);
    yticks([0 .1*roomHeight .2*roomHeight .3*roomHeight .4*roomHeight ...
        .5*roomHeight .6*roomHeight .7*roomHeight .8*roomHeight .9*roomHeight...
        roomHeight]);

    %LABELING AXES
    xlabel('WIDTH');
    ylabel('HEIGHT');
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    %SETTING POSITION AND VELOCITY TO INITIAL CONDITIONS
    position = initPosition;
    velocity = initVelocity;

    %RUN SIMULATION
    while abs(t) < abs(simLength)

        %TIME COUNTER FOR LOOP ITERATION
        t_loopstart=tic();

        %CALCULATE NEXT POSITION
        position = cPosition(position, velocity, timeStep, bounds0, ...
                             boundsMax, ballRadius);
```

```
%CALCULATE NEXT VELOCITY
velocity = cVelocity(position, velocity, acceleration, timeStep, ...
                     bounds0, boundsMax, ballRadius, ...
                     normalDamp, tangentialDamp);

%CLEAR FIGURE
clf;

%%%%%%%%%%%%%%%%%%%%%% DRAW NEXT FRAME %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%DRAW CIRCLE IN UPDATED POSITION
rectangle('Position',[position 2*ballRadius 2*ballRadius],...
    'Curvature',[1,1],...
    'FaceColor','r');

%PRESERVE AXES
axis([0 roomWidth 0 roomHeight]);
axis('square');
axis('on');
grid on;
grid minor;
xticks([0 .1*roomWidth .2*roomWidth .3*roomWidth .4*roomWidth ...
    .5*roomWidth .6*roomWidth .7*roomWidth .8*roomWidth .9*roomWidth ...
    roomWidth]);
yticks([0 .1*roomHeight .2*roomHeight .3*roomHeight .4*roomHeight ...
    .5*roomHeight .6*roomHeight .7*roomHeight .8*roomHeight ...
    .9*roomHeight roomHeight]);

%PRESERVE LABELS
title('A 2D Simulation of a Bouncing Ball');
xlabel('WIDTH');
ylabel('HEIGHT');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%UPDATE TIME BALL HAS BEEN SIMULATED FOR
t = t + timeStep;

%PAUSING ANIMATION TO MAKE IT VIEWABLE
el_time=toc(t_loopstart);     %CALCULATE TIME IT TOOK FOR LOOP ITERATION
if (el_time < timeStep)       %
    pause(timeStep-el_time);  %IF THE SYSTEM IS FAST ENOUGH TO
```

```
        else                            %ITERATE THROUGH THE LOOP FASTER THAN
            pause(.02);                 %timeStep, MAKE ANIMATION VERY CLOSE TO
        end                             %REAL TIME, ELSE PAUSE TO MAKE VIEWABLE
    %END OF ANIMATION
    end

    %OUTPUT TIME COMPUTER TOOK TO ANIMATE THE SYSTEM SO IT CAN BE COMPARED TO
    %THE TIME IT SIMULATED FOR TO SEE IF ANIMATION WAS IN REAL TIME OR NOT
    %IF program_run_time_2D IS ABOUT EQUAL TO simLength THEN ANIMATION WAS VERY
    %CLOSE TO REAL TIME, ELSE ANIMATION WAS NOT REAL TIME, BUT STILL SPANNED THE
    %LENGTH OF TIME IN THE REAL WORLD SPECIFIED BY simLength
    program_run_time_2D=toc(program_start)

end
```

BouncyBallMovie3D.m:

```
function BouncyBallMovie3D( cVelocity, cPosition, initPosition, ...
                        initVelocity, acceleration, timeStep, dampingMatrix,...
                        ballRadius, roomWidth, roomHeight, roomDepth, ...
                        simLength, windowWidth, windowHeight )
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%BouncyBallMovie3D Plays a movie of a 3D ball bouncing in a walled 3D space
%   Arguments:
%       cVelocity: Calculates velocity for the next step, can work anyway you
%       want as long as it takes a position vector, an initial velocity vector,
%       an acceleration vector, a time step, the lower and upper bounds of the
%       3D space, the ball radius, the normal damping force, and the tangential
%       damping force. This function needs so many arguments because it must
%       also take care of the cases when the ball hits a wall.
%
%       cPosition: Calculates postion for the next step, can work anyway you
%       want as long as it takes an initial position vector, a velocity vector,
%       a time step, the lower and upper bounds of the 3D space, and the ball
%       radius. This function also take care of the cases when the ball hits a
%       wall.
%       ****cPosition and cVelocity should coordinate to take care of the case
%           when the ball hits the wall.
%
%       initPosition: Initial 3D position vector of the ball. The vector is a
%       1 x 3 vector as such - [x-position y-position z-position] where z is the
%       vertical component, y is the depth component, and x is the horizontal
%       component. Units of the vector are in meters.
%
%       initVelocity: Initial 3D velocity vector of the ball. The vector is a
%       1 x 3 vector as such - [x-velocity y-velocity z-velocity] where z is the
%       vertical component, y is the depth component, and x is the horizontal
%       component. Units of the vector are in meters per second.
%
%       acceleration: The acceleration vector for the system. The vector is a
%       1 x 3 vector as such - [x-acceleration y-acceleration z-acceleration]
%       where z is the vertical component, y is the depth component, and x is
%       the horizontal component. For this simulation, the acceleration is
%       constant. Units of the vector are in meters per second^2.
%
%       timeStep: The time interval between successive position and velocity
%       calculations.
%
```

```
%       dampingMatrix: The collision damping maxtrix for the system. The matrix
%       is a 1 x 2 vector as such - [alpha beta] where alpha is the collision
%       damping coefficient of the velocity component normal to the wall and
%       beta is the collision damping coefficient of the velocity component
%       tangential to the wall.
%
%       ballRadius: The radius of the ball in meters.
%
%       roomWidth, roomHeight, roomDepth: The width of the room in meters, the
%       height of the room in meters, and the depth of the room in meters
%       respectively.
%       ***** Note that the axes of the animation will always make a cube
%             so if the height, depth, and width are not the same, the ball will
%             look like an ellipse.
%
%       simLength: Time in seconds the system should be simulated for.
%       ***** This is important for this function because the Zeno Phenomenon is
%       not accounted for here so without a duration for the simulation, it may
%       run forever.
%
%       windowWidth,windowHeight: The width and height of the window the
%       animation is contained in respectively
%       ***** This will affect the size of the animation as well, but remember
%       the axes will always make a cube so having a longer width will not
%       make the animation wider if the width of the window exceeds the height.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    program_start=tic(); %STARTS THE CLOCK FOR THE RUN TIME OF THE PROGRAM

    %SET UP THE DAMPING COEFFICIENTS AS SEPERATE VARIABLES FOR EASIER
    normalDamp = dampingMatrix * [1;0];                    %CALCULATION LATER ON
    tangentialDamp = dampingMatrix * [0;1];

    %SET UP LOWER AND UPPER BOUNDS FOR FUTURE CALCULATIONS
    bounds0 = [0, 0, 0];
    boundsMax = [roomWidth, roomDepth, roomHeight];

    %SET TIME VARIABLE TO 0. TO BE USED TO SIMULATE PROGRAM FOR DESIRED
    t = 0;                                                 %LENGTH OF TIME

%%%%%%%%%%%%%%%%%%%%%%%%%%%%% DRAWING FIRST FRAME %%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %DRAW BALL IN INITAL POSITION
```

```
    figure('position', [250, 50, windowWidth, windowHeight])
    title('A 3D Simulation of a Bouncing Ball');
    Draw_Sphere(initPosition*[1;0;0],initPosition*[0;1;0], ...
                initPosition*[0;0;1],ballRadius);

    %CREATING AXES
    axis([0 roomWidth 0 roomDepth 0 roomHeight]);
    axis('square');
    axis('on');
    grid on;
    grid minor;
    xticks([0 .1*roomWidth .2*roomWidth .3*roomWidth .4*roomWidth ...
        .5*roomWidth .6*roomWidth .7*roomWidth .8*roomWidth .9*roomWidth ...
        roomWidth]);
    yticks([0 .1*roomDepth .2*roomDepth .3*roomDepth .4*roomDepth ...
        .5*roomDepth .6*roomDepth .7*roomDepth .8*roomDepth .9*roomDepth...
        roomDepth]);
    zticks([0 .1*roomHeight .2*roomHeight .3*roomHeight .4*roomHeight ...
        .5*roomHeight .6*roomHeight .7*roomHeight .8*roomHeight .9*roomHeight...
        roomHeight]);

    %LABELING AXES
    xlabel('WIDTH');
    ylabel('DEPTH');
    zlabel('HEIGHT');
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    %SETTING POSITION AND VELOCITY TO INITIAL CONDITIONS
    position = initPosition;
    velocity = initVelocity;

    %RUN SIMULATION
    while abs(t) < abs(simLength)

        %TIME COUNTER FOR LOOP ITERATION
        t_loopstart=tic();

        %CALCULATE NEXT POSITION
        position = cPosition(position, velocity, timeStep, bounds0, ...
                             boundsMax, ballRadius);

        %CALCULATE NEXT VELOCITY
```

```
velocity = cVelocity(position, velocity, acceleration, timeStep, ...
                     bounds0, boundsMax, ballRadius, ...
                     normalDamp, tangentialDamp);

%CLEAR FIGURE
clf;

%%%%%%%%%%%%%%%%%%%%%% DRAW NEXT FRAME %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%DRAW BALL IN UPDATED POSITION
Draw_Sphere(position*[1;0;0],position*[0;1;0],position*[0;0;1], ...
            ballRadius);

%PRESERVE AXES
axis([0 roomWidth 0 roomDepth 0 roomHeight]);
axis('square');
axis('on');
grid on;
grid minor;
xticks([0 .1*roomWidth .2*roomWidth .3*roomWidth .4*roomWidth ...
    .5*roomWidth .6*roomWidth .7*roomWidth .8*roomWidth .9*roomWidth ...
    roomWidth]);
yticks([0 .1*roomDepth .2*roomDepth .3*roomDepth .4*roomDepth ...
    .5*roomDepth .6*roomDepth .7*roomDepth .8*roomDepth .9*roomDepth...
    roomDepth]);
zticks([0 .1*roomHeight .2*roomHeight .3*roomHeight .4*roomHeight ...
    .5*roomHeight .6*roomHeight .7*roomHeight .8*roomHeight ...
    .9*roomHeight roomHeight]);

%PRESERVE LABELS
title('A 3D Simulation of a Bouncing Ball');
xlabel('WIDTH');
ylabel('DEPTH');
zlabel('HEIGHT');
view(45,25);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%UPDATE TIME BALL HAS BEEN SIMULATED FOR
t = t + timeStep;

%PAUSING ANIMATION TO MAKE IT VIEWABLE
```

```
        el_time=toc(t_loopstart);     %CALCULATE TIME IT TOOK FOR LOOP ITERATION
        if (el_time < timeStep)       %
            pause(timeStep-el_time); %IF THE SYSTEM IS FAST ENOUGH TO
        else                          %ITERATE THROUGH THE LOOP FASTER THAN
            pause(.02);               %timeStep, MAKE ANIMATION VERY CLOSE TO
        end                           %REAL TIME, ELSE PAUSE TO MAKE VIEWABLE
    %END OF ANIMATION
    end

    %OUTPUT TIME COMPUTER TOOK TO ANIMATE THE SYSTEM SO IT CAN BE COMPARED TO
    %THE TIME IT SIMULATED FOR TO SEE IF ANIMATION WAS IN REAL TIME OR NOT
    %IF program_run_time_3D IS ABOUT EQUAL TO simLength THEN ANIMATION WAS VERY
    %CLOSE TO REAL TIME, ELSE ANIMATION WAS NOT REAL TIME, BUT STILL SPANNED THE
    %LENGTH OF TIME IN THE REAL WORLD SPECIFIED BY simLength
    program_run_time_3D=toc(program_start)

end
```

BouncyBallMovie3dPlot.m:

```
function BouncyBallMovie3DPlot( cVelocity, cPosition, initPosition, ...
                        initVelocity, acceleration, timeStep, dampingMatrix,...
                        ballRadius, roomWidth, roomHeight, roomDepth, ...
                        simLength, windowWidth, windowHeight )
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%BouncyBallMovie3DPlot Pots the trajectory of a 3D ball bouncing in a walled 3D
%                       space.
%***** The arguments for this function are the same as they are for
%         BouncyBallMovie3D. This function exists because although the
%         simulation produced by BouncyBallMovie3D is accurate, it does not look
%         like it is, because the sphere does not shrink as it moves farther
%         back in the room and does not change shades from shadowing either. In
%         other words it is hard to tell exactly what direction the ball is
%         moving since depth cannot be seen. This plot shows the path the center
%         of the sphere in BouncyBallMovie3D took. Once the simulation has been
%         run for the desired simLength, the plot can be rotated in matlab to
%         visualize the path of the ball better.
%
%   Arguments:
%       cVelocity: Calculates velocity for the next step, can work anyway you
%       want as long as it takes a position vector, an initial velocity vector,
%       an acceleration vector, a time step, the lower and upper bounds of the
%       3D space, the ball radius, the normal damping force, and the tangential
%       damping force. This function needs so many arguments because it must
%       also take care of the cases when the ball hits a wall.
%
%       cPosition: Calculates postion for the next step, can work anyway you
%       want as long as it takes an initial position vector, a velocity vector,
%       a time step, the lower and upper bounds of the 3D space, and the ball
%       radius. This function also take care of the cases when the ball hits a
%       wall.
%       ****cPosition and cVelocity should coordinate to take care of the case
%           when the ball hits the wall.
%
%       initPosition: Initial 3D position vector of the ball. The vector is a
%       1 x 3 vector as such - [x-position y-position z-position] where z is the
%       vertical component, y is the depth component, and x is the horizontal
%       component. Units of the vector are in meters.
%
%       initVelocity: Initial 3D velocity vector of the ball. The vector is a
%       1 x 3 vector as such - [x-velocity y-velocity z-velocity] where z is the
```

```
%       vertical component, y is the depth component, and x is the horizontal
%       component. Units of the vector are in meters per second.
%
%       acceleration: The acceleration vector for the system. The vector is a
%       1 x 3 vector as such - [x-acceleration y-acceleration z-acceleration]
%       where z is the vertical component, y is the depth component, and x is
%       the horizontal component. For this simulation, the acceleration is
%       constant. Units of the vector are in meters per second^2.
%
%       timeStep: The time interval between successive position and velocity
%       calculations.
%
%       dampingMatrix: The collision damping maxtrix for the system. The matrix
%       is a 1 x 2 vector as such - [alpha beta] where alpha is the collision
%       damping coefficient of the velocity component normal to the wall and
%       beta is the collision damping coefficient of the velocity component
%       tangential to the wall.
%
%       ballRadius: The radius of the ball in meters.
%
%       roomWidth, roomHeight, roomDepth: The width of the room in meters, the
%       height of the room in meters, and the depth of the room in meters
%       respectively.
%       ***** Note that the axes of the animation will always make a cube
%             so if the height, depth, and width are not the same, the ball will
%             look like an ellipse.
%
%       simLength: Time in seconds the system should be simulated for.
%       ***** This is important for this function because the Zeno Phenomenon is
%       not accounted for here so without a duration for the simulation, it may
%       run forever.
%
%       windowWidth,windowHeight: The width and height of the window the
%       animation is contained in respectively
%       ***** This will affect the size of the animation as well, but remember
%       the axes will always make a cube so having a longer width will not
%       make the animation wider if the width of the window exceeds the height.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    program_start=tic(); %STARTS THE CLOCK FOR THE RUN TIME OF THE PROGRAM

    %SET UP THE DAMPING COEFFICIENTS AS SEPERATE VARIABLES FOR EASIER
```

```
    normalDamp = dampingMatrix * [1;0];                    %CALCULATION LATER ON
    tangentialDamp = dampingMatrix * [0;1];

    %SET UP LOWER AND UPPER BOUNDS FOR FUTURE CALCULATIONS
    bounds0 = [0, 0, 0];
    boundsMax = [roomWidth, roomDepth, roomHeight];

    %SET TIME VARIABLE TO 0. TO BE USED TO SIMULATE PROGRAM FOR DESIRED
    t = 0;                                                 %LENGTH OF TIME

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% DRAWING FIRST FRAME %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %DRAW BALL IN INITAL POSITION
    figure('position', [250, 50, windowWidth, windowHeight])
    title('3D Trajectory Plot of the Center of a Simulated Bouncing Ball');
    plot3(initPosition*[1;0;0],initPosition*[0;1;0], ...
          initPosition*[0;0;1],'*');hold on;

    %CREATING AXES
    axis([0 roomWidth 0 roomDepth 0 roomHeight]);
    axis('square');
    axis('on');
    grid on;
    grid minor;
    xticks([0 .1*roomWidth .2*roomWidth .3*roomWidth .4*roomWidth ...
        .5*roomWidth .6*roomWidth .7*roomWidth .8*roomWidth .9*roomWidth ...
        roomWidth]);
    yticks([0 .1*roomDepth .2*roomDepth .3*roomDepth .4*roomDepth ...
        .5*roomDepth .6*roomDepth .7*roomDepth .8*roomDepth .9*roomDepth...
        roomDepth]);
    zticks([0 .1*roomHeight .2*roomHeight .3*roomHeight .4*roomHeight ...
        .5*roomHeight .6*roomHeight .7*roomHeight .8*roomHeight .9*roomHeight...
        roomHeight]);

    %LABELING AXES
    xlabel('WIDTH');
    ylabel('DEPTH');
    zlabel('HEIGHT');
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    %SETTING POSITION AND VELOCITY TO INITIAL CONDITIONS
    position = initPosition;
    velocity = initVelocity;
```

```
%RUN SIMULATION
while abs(t) < abs(simLength)

    %TIME COUNTER FOR LOOP ITERATION
    t_loopstart=tic();

    %CALCULATE NEXT POSITION
    position = cPosition(position, velocity, timeStep, bounds0, ...
                         boundsMax, ballRadius);

    %CALCULATE NEXT VELOCITY
    velocity = cVelocity(position, velocity, acceleration, timeStep, ...
                         bounds0, boundsMax, ballRadius, ...
                         normalDamp, tangentialDamp);

    %%%%%%%%%%%%%%%%%%%%% DRAW NEXT FRAME %%%%%%%%%%%%%%%%%%%%%%%%%%%%

    %DRAW BALL IN UPDATED POSITION
    plot3(position*[1;0;0],position*[0;1;0],position*[0;0;1],'*');hold on;

    %PRESERVE AXES
    axis([0 roomWidth 0 roomDepth 0 roomHeight]);
    axis('square');
    axis('on');
    grid on;
    grid minor;
    xticks([0 .1*roomWidth .2*roomWidth .3*roomWidth .4*roomWidth ...
        .5*roomWidth .6*roomWidth .7*roomWidth .8*roomWidth .9*roomWidth ...
        roomWidth]);
    yticks([0 .1*roomDepth .2*roomDepth .3*roomDepth .4*roomDepth ...
        .5*roomDepth .6*roomDepth .7*roomDepth .8*roomDepth .9*roomDepth...
        roomDepth]);
    zticks([0 .1*roomHeight .2*roomHeight .3*roomHeight .4*roomHeight ...
        .5*roomHeight .6*roomHeight .7*roomHeight .8*roomHeight ...
        .9*roomHeight roomHeight]);

    %PRESERVE LABELS
    title('3D Trajectory Plot of the Center of a Simulated Bouncing Ball');
    xlabel('WIDTH');
    ylabel('DEPTH');
    zlabel('HEIGHT');
```

```
        view(45,25);

        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        %UPDATE TIME BALL HAS BEEN SIMULATED FOR
        t = t + timeStep;

        %PAUSING ANIMATION TO MAKE IT VIEWABLE
        el_time=toc(t_loopstart);     %CALCULATE TIME IT TOOK FOR LOOP ITERATION
        if (el_time < timeStep)       %
            pause(timeStep-el_time);  %IF THE SYSTEM IS FAST ENOUGH TO
        else                          %ITERATE THROUGH THE LOOP FASTER THAN
            pause(.02);               %timeStep, MAKE ANIMATION VERY CLOSE TO
        end                           %REAL TIME, ELSE PAUSE TO MAKE VIEWABLE
    %END OF ANIMATION
    end

    %OUTPUT TIME COMPUTER TOOK TO ANIMATE THE SYSTEM SO IT CAN BE COMPARED TO
    %THE TIME IT SIMULATED FOR TO SEE IF ANIMATION WAS IN REAL TIME OR NOT
    %IF program_run_time_3DPlot IS ABOUT EQUAL TO simLength THEN ANIMATION WAS
    %CLOSE TO REAL TIME, ELSE ANIMATION WAS NOT REAL TIME, BUT STILL SPANNED THE
    %LENGTH OF TIME IN THE REAL WORLD SPECIFIED BY simLength
    program_run_time_3DPlot=toc(program_start)

end
```

Draw_Sphere.m:

```
function Draw_Sphere(xc,yc,zc,R)
    [x,y,z] = sphere;
    surfl(R*x+xc,R*y+yc,R*z+zc);
    shading interp; colormap(autumn);
end
```