Simulation of a bouncing ball on a trampoline in 3D

Zan Ahmad   Chongkun Zhao

October 2020

Contents

1 Introduction **2**

2 Equations **2**

2.1 Network of Linear Springs in 3D 2

2.2 Construction of Trampoline Geometry 3

2.3 Modeling 3D Sphere interaction 4

3 Numerical Methods **5**

4 Implementation in MATLAB **6**

4.1 plotTram.m 6

4.2 testradius.m 7

4.3 trampoline.m 7

5 Results and Discussion **9**

A Complete code for the project **12**

A.1 plotTram.m 12

A.2 testradius.m 14

A.3 trampoline.m 15

1 Introduction

In this report, we will be constructing a square trampoline grid and simulating various kinematic trajectories regarding its interactions with spherical ball. The structure of the trampoline will be generated by adapting equations for an arbitrary network of nodes connected by springs equipped with dash-pots. The interactions between objects will be governed by Newtonian mechanics. In this report, we exclude considerations of spin, so the interactions between the ball and the trampoline are frictionless. Several variations of this system will be considered, beginning from a simple, flat trampoline with a ball being dropped from rest and advancing to a system of several trampolines rotated at various angles with a ball interacting between them.

2 Equations

2.1 Network of Linear Springs in 3D

First, let us consider an arbitrary three dimensional network of nodes linked with linear springs that are equipped with dampers. The goal is to use this general framework to build a realistic trampoline in the following subsection. [1]

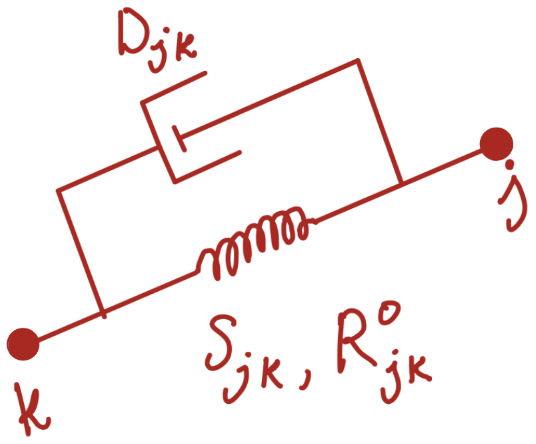

Figure 1: Schematic of two arbitrary nodes j and k connected by a linear spring equipped with a dashpot

A spring is defined as something where the force depends on length. A linear spring between j and k has a rest length R^0_{jk}, a stiffness S_{jk}, and a damping constant D_{jk}. Here, j and k are nodes with respective masses. This rest length is the distance between the two nodes that results in zero spring force between them. If we stretch or compress the spring so that the nodes they connect are a length away from each other that is not equal to the rest length, then the stiffness gives information on how much the spring force between them changes. The dash-pot that each spring is equipped with is essentially a shock absorber that exerts a force in the opposite direction of velocity, slowing the motion and absorbing energy, to ensure that the spring does not oscillate forever. The magnitude of this effect is related to D_{jk}. From Newton's Second Law, $\mathbf{F} = m\mathbf{a}_k$, where m is mass and $\mathbf{a}_k$ is acceleration of node k, we can derive the following relationship for the total link force on node k within the spring network:

$$\mathbf{F}_{l,k} = M_k \frac{d\mathbf{U}_k}{dt} = \sum_{j \in N(k)} T_{jk} \frac{\mathbf{X}_j - \mathbf{X}_k}{||\mathbf{X}_j - \mathbf{X}_k||} \tag{1}$$

where

$$\mathbf{U}_k = \frac{d\mathbf{X}_k}{dt} \tag{2}$$

$$T_{jk} = S_{jk}(||\mathbf{X}_j - \mathbf{X}_k|| - R^0_{jk}) + D_{jk}\frac{d}{dt}||\mathbf{X}_j - \mathbf{X}_k|| \tag{3}$$

In (1), M_k is the mass of node k, T_{jk} is the tension in the spring connecting nodes j and k, and $\mathbf{X_j}$ is the position vector corresponding to node j. Therefore, $\mathbf{X}_j - \mathbf{X}_k$ is the vector that points from node k to j, and $||\mathbf{X}_j - \mathbf{X}_k||$ is the length of such vector. So $\frac{\mathbf{X}_j - \mathbf{X}_k}{||\mathbf{X}_j - \mathbf{X}_k||}$ is a unit vector pointing from node k to j. The change in velocity over change in time is defined as acceleration, $\mathbf{a}_k = \frac{d\mathbf{U}_k}{dt}$. In essence, the force on k, $\mathbf{F}_k = M_k\mathbf{a}_k$, is equal to the sum of the tension forces in the direction j to k for all j in the set of neighboring nodes that k is connected to, denoted as $N(k)$. The spring and the dashpot both work together to create tension. The stiffness of the spring contributes to the tension when there is a deviation from the rest length $S_{jk}(||\mathbf{X}_j - \mathbf{X}_k|| - R^0_{jk})$, and the damping constant of the dashpot contributes to the tension when there is any change in the distance between the nodes, $\frac{d}{dt}||\mathbf{X}_j - \mathbf{X}_k||$, or in other words, if there is any nonzero velocity of the spring, since:

$$\frac{d}{dt}||\mathbf{X}_j - \mathbf{X}_k|| = \frac{\mathbf{X}_j - \mathbf{X}_k}{||\mathbf{X}_j - \mathbf{X}_k||} \cdot (\mathbf{U}_j - \mathbf{U}_k) \tag{4}$$

The state of an arbitrary network of springs is the position and the velocity of every node. The above formulations thus describe an equation for the rate of change of the state as a function of the state itself. Therefore, as long as we know the initial position and velocity of each node, we can solve the ordinary differential equations (1) and (2) to simulate the movement of the nodes over time.

2.2 Construction of Trampoline Geometry

To construct the trampoline structure, we arrange the nodes in such a way that they become a uniformly spaced square grid connected with horizontal and vertical links at time t=0.

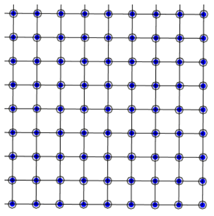

Figure 2: 9 by 9 grid of nodes

We assume that all nodes have the same mass and that each link between them has the same stiffness, damping constant, and rest length. To mimic a trampoline surface, we set the rest length for all links to be smaller than the distance between adjacent nodes on the grid plane at $t = 0$. The nodes on the boundaries are taken to be fixed in space and we release the interior nodes from rest and the force of gravity is applied on each node k:

$$\mathbf{F}_{g,k} = -M_k g \hat{\mathbf{k}} \tag{5}$$

where g is the gravitational acceleration constant and $\hat{\mathbf{k}}$ is the unit vector in the z direction.

2.3 Modeling 3D Sphere interaction

To model a ball bouncing on a trampoline, consider a sphere with a center location $\mathbf{C}$ and a radius R in $\mathbb{R}^3$. Let all of its mass, M_{ball}, be concentrated at the center so that it is essentially a point mass with a specified radius. An important assumption made here is that the ball is frictionless, meaning it does not spin. Now, there are two instances that we need to consider for the ball: the case where it is above the trampoline and the case where it is in contact with the trampoline. When the ball is not in contact with the trampoline, the only force acting on it is gravity:

$$\mathbf{F}_{g,\text{ball}} = -M_{\text{ball}} g \hat{\mathbf{k}} \tag{6}$$

When the ball is in contact with the nodes of the trampoline, this means that there are a nonzero number of nodes that are within distance R of the center of mass of the ball, $\mathbf{C}$. The ball is exerting a force $\mathbf{F}_{\text{ball},i}$ onto each node i in this cohort of nodes inside the radius of the ball at time t, $\mathscr{B}(t)$, and by Newton's third law, each node $i \in \mathscr{B}(t)$ is exerting an equal and opposite force back onto the center of mass of the sphere, $\mathbf{F}_{i,\text{ball}}$. This interaction between the center of mass of the sphere and the node is spring-like, where the rest length is the radius R and the stiffness of the spring is K_{ball}.
This particular spring-like interaction does not feel any effects when the spring is "stretched" (i.e. $||\mathbf{X}_i - \mathbf{C}|| \geq R$), however, it does feel the effects when the spring is compressed and $||\mathbf{X}_i - \mathbf{C}|| < R$. The following equations reiterate the ideas stated above:

$$\mathbf{F}_{\text{ball},i} = K_{\text{ball}}(R - ||\mathbf{X}_i - \mathbf{C}||) \cdot \frac{\mathbf{X}_i - \mathbf{C}}{||\mathbf{X}_i - \mathbf{C}||} \tag{7}$$

$$\mathbf{F}_{i,\text{ball}} = -\mathbf{F}_{\text{ball},i} \tag{8}$$

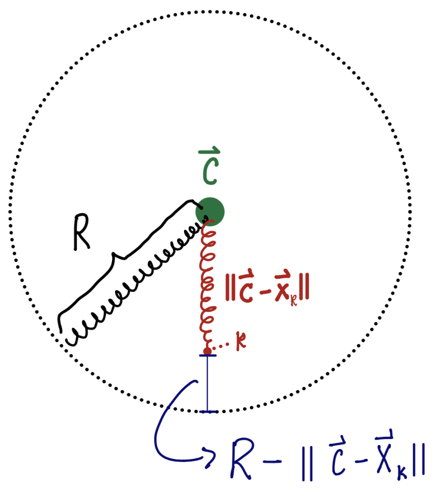

Figure 3: 2D cross sectional schematic of ball contacting the trampoline (node k is within radius distance from the center of the point mass of the ball).

The resultant vector from the sum of these equal and opposite forces acting on the ball from the nodes can be approximated to be pointing in the normal direction when we take n to be large. Thus the normal force exerted on the ball by the trampoline is:

$$\mathbf{F}_{\text{Normal}} = \sum_{i\in\mathscr{B}(t)} \mathbf{F}_{i,\text{ball}} \tag{9}$$

Of course, when the nodes are exerting a force on the ball, gravity is still in effect, so the total force on the ball at any point in time is as follows:

$$\mathbf{F}_{\text{ball}} = \mathbf{F}_{g,\text{ball}} + \mathbf{F}_{i,\text{ball}} \tag{10}$$

Note that when $\mathscr{B} = \varnothing$, then $\mathbf{F}_{i,\text{ball}} = 0$. The movement of the ball can be modeled by the following ODEs:

$$\mathbf{F}_{\text{ball}} = M_{\text{ball}}\frac{d\mathbf{U}_{\text{ball}}}{dt} \tag{11}$$

$$\mathbf{U}_{\text{ball}} = \frac{d\mathbf{C}}{dt} \tag{12}$$

3 Numerical Methods

To solve the ODEs in (1) and (2), we replace the time derivatives with discretized difference quotients regarding the next time step. This is called forward Euler's method:

$$M_k\frac{\mathbf{U}_k(t+\Delta t)-\mathbf{U}_k(t)}{\Delta t} = \sum_{j\in N(k)} T_{jk}(t)\frac{\mathbf{X}_j(t)-\mathbf{X}_k(t)}{||\mathbf{X}_j(t)-\mathbf{X}_k(t)||} \tag{13}$$

$$\frac{\mathbf{X}_k(t+\Delta t)-\mathbf{X}_k(t)}{\Delta t} = \mathbf{U}_k(t+\Delta t) \tag{14}$$

We solve the ODEs in (11) and (12) in the same way:

$$M_{\text{ball}}\frac{\mathbf{U}_{\text{ball}}(t+\Delta t)-\mathbf{U}_{\text{ball}}(t)}{\Delta t} = -\sum_{i\in\mathscr{B}} K_{\text{ball}}(R-||\mathbf{X}_i(t)-\mathbf{C}(t)||)\cdot\frac{\mathbf{X}_i(t)-\mathbf{C}(t)}{||\mathbf{X}_i(t)-\mathbf{C}(t)||} - M_{\text{ball}}g\hat{\mathbf{k}} \tag{15}$$

$$\frac{\mathbf{C}(t+\Delta t)-\mathbf{C}(t)}{\Delta t} = \mathbf{U}_{\text{ball}}(t+\Delta t) \tag{16}$$

This is actually a slight improvement from Euler's method. In Euler's method, in equations (14) and (16) we would have $\mathbf{U}_{\text{ball}}$ and $\mathbf{U}_k$ be evaluated at the previous time step, t, but since we know its value at $t+\Delta t$, which is the next time step, our approximation is slightly more accurate.

4 Implementation in MATLAB

There are various programs involved in this simulation. We have a function to construct the trampoline, a function to compute the distance between the ball and every node on the trampoline, and a driver script from which we run the simulation. The following subsections outline in detail how these programs work.

4.1 plotTram.m

The plotTram.m function builds the trampoline. As input, we need parameters **n** and **a**, where **n** is the number of nodes on one side of the trampoline and **a** is the distance between adjacent nodes at time $t = 0$. Also, the function takes in a rotation angle θ and a 1×3 translation matrix. These two parameters allow user to rotate the trampoline around x axis by initializing an angle in the driver script trampoline.m.

With these inputs, the function returns the location matrix, **X**, of each node, link matrix **jj**, **kk** and a matrix of all nodes that we want to fix, called **fixed**. The link table **jj** is a matrix that stores the node on one end of each link, while **kk** stores the other end. For example, **jj**`(first_link)` = `first_node`, and **kk**`(first_link)` = `second_node`, etc.
In terms of the dimensions of each output, **X** is a $N\times 3$ matrix, where N is the total number of nodes and 3 stands for x, y, z coordinates. **fixed** has dimension $N\times 1$, and each value is 1 if the node is fixed, 0 if it can move. One thing that we should keep in mind is that, we enumerate the nodes from 1 to N. For every matrix that describes the nodes, the n-th index coresponds to node n.
Constructing the link table **jj** and **kk** is straightforward, so we make an intermediate matrix **NB** that memorizes what neighbours each node has. The number of neighbours depends on how we construct the trampoline, in our simulation, links only come in vertical and horizontal directions, that is to say, the interior nodes will have 4 neighbours. Therefore, **NB** is a $N\times 4$ matrix. In order to specify the neighbours of each node, we use the following nested loop to visit every node and set conditions for the boundary nodes that have fewer neighbours.

```
nb = zeros(N, 4);
for iy = 1:n
    for ix = 1:n
        idx = ix + n * (iy-1);
        if ix > 1; nb(idx,1) = idx - 1; end%if the node is left-most, ...
           it does not have left neighbour
        if ix < n; nb(idx,2) = idx + 1; end%right-most nodes don't ...
           have right neighbour.
        if iy > 1; nb(idx,3) = idx - n; end
        if iy < n; nb(idx,4) = idx + n; end
    end
end
```

With the help of this neighbour matrix, it will be convenient to build up the link tables **jj** and **kk**. Link tables are $2NN \times 1$ matrix where NN is the total number of links. The following code reshapes NB into **jj** and **kk**.

```
jj = []; kk = [];
for link = 1:size(nb,2) %size(nb,2) is the number of neighbour each ...
    node has
  inds = (nb(:,link) > 0);
  jj = [jj;nodes(inds)]; %The neighbours of a node are some ends of ...
      some links.
  kk = [kk;nb(inds,link)];
  %we actually count each link twice.
end
```

The reason why we doubled NN is that when we loop through the neighours, for example, if one end of a link **P** is $\mathbf{a}_k$ and the other end is **B**, **P** will be counted when we find out **B** is a neighbour of $\mathbf{a}_k$, and it will be counted again when we find out $\mathbf{a}_k$ is a neighbour of **B**. To solve this problem, we just need to divide the real stiffness by 2 when we calculate the link force.

4.2 testradius.m

This function is very crucial when the the ball interacts with the trampoline. As we can infer from its name, the function determines which nods are inside the radius of the ball. The result is a logical matrix, namely **in_out**, whose dimension is $N \times 1$. Each node will have value logical **1** if its distance to **C** is less or equal to the radius **R**. Testradius.m also returns the vector between **C** the node that is within **R**. These vectors are important for calculating our imaginary spring force of the ball.

4.3 trampoline.m

With these two function, we are good to go. We are quite ambitious about our project, so we are not satisfied with only one trampoline. The code has been highly modularized, and it allows us to build several trampolines that rotate in different angles and can be translated to different position. The following set up reads in as many trampolines as we want.

```
num_tram = 4; %number of trampolines
kmax = n^2; %size of X
lmax = 4 * n * (n-1); %size of jj: (twice as many as the total links)
thetax = [0 -pi/4 +pi/6 -pi/4]; %angle of rotation
mtrans = [0 0 0;
          0 -6 4;
          0 -6 -15;
          0 +6 -15];// each row correspond to the translation matrix ...
              of a trampoline.
%-----------------preallocate everything-----------------
N0 = zeros(1,num_tram); //
NN0 = zeros (1,num_tram);
X0 = zeros(kmax, 3, num_tram );
jj0 = zeros(lmax, 1, num_tram);
kk0 = zeros(lmax, 1, num_tram);
fixed0 = zeros(kmax, 1, num_tram );
U0 = zeros(kmax,3,num_tram); %preallocate velocity of trampoline nodes
% These are array of matrices that we can read and edit.

%-------------------trampoline mega loop ------------------------
for i = 1:num_tram
    thetax_temp = thetax(i);
    mtrans_temp = mtrans(i,:);
    [N_temp,NN_temp,X_temp,jj_temp,kk_temp,fixed_temp] = ...
       plotTram(n,a,thetax_temp,mtrans_temp);%build the structure
    N0(i) = N_temp;
    NN0(i) = NN_temp;
    X0(:,:,i) = X_temp;
    jj0(:,:,i) = jj_temp;
    kk0(:,:,i) = kk_temp;
    fixed0(:,:,i) = fixed_temp;
end
```

After this mega preallocation, we set up some global variables such as gravity, stiffness and damping constant. We also need to set up the time step and maximal time for our forward Euler's method. In our simulation with relatively small trampolines, $dt = 0.001$ will be small enough to perform a beautiful bouncing ball. We also need to initialize the the properties of the ball. Especially, we need to set up a proper value for the imaginary stiffness of the ball. According to our test results, we find that it will be appreciate to establish a linear relation between the mass of the ball and the imaginary stiffness. Therefore we set

$$\mathbf{K_{imaginary}} = \mathbf{M_{ball}} \times 10^5.$$

Within each time step, we need to loop through every trampoline, then:

1. Read data from the big arrays of matrix.

2. Calculate link vectors, link length, link tension. Recall that we need to divide the Stiffness by 2 here to cancel the effect of counting links twice.

```
        T = (S/2).*(R-R0) + (D./R).*sum(DX.*DU,2); %S for ...
            stiffness, R-R0 for length difference, D for damping ...
            constant, DX for link vector and DU for velocity ...
            difference.
```

3. Calculate forces:

   (a) link force:

```
            % For each link, add its force with correct sign
            % to the point at each end of the link:
            for link=1:NN
                F(kk(link),:) = F(kk(link),:) + FF(link,:);
                F(jj(link),:) = F(jj(link),:) - FF(link,:);
            end
```

   (b) Calculate the force when the ball and the trampoline interacts.

```
            [in_out, ndir, DisF]= testradius(C, X, ...
               Radius);%DisF is the vector of difference ...
               between distance and R.
            TEST = sum(in_out);
            in_out = logical(in_out);
            if(TEST ≠ 0)%if at list one index in in_out is ...
               nonezero, TEST will be nonzero.
                F(in_out,:) = F(in_out,:) + K_pen .* ...
                    (DisF(in_out,:));
                F_ball = F_ball - sum(K_pen .* (DisF(in_out,:)),1);
            end
```

4. Apply Euler's method on acceleration and velocity of both the ball and the trampoline using Newton's second law.

5. Store the location of ball and trampoline back into the big array of matrices so that we can read it in the next time step.

We also implemented some interesting ways to plot the whole scene. The complete code will be attached in the final section of the paper.

5 Results and Discussion

The results of our simulation show realistic behaviors of the balls interaction with the trampoline as can be seen in the trajectories shown. After much experimentation, found that the ideal parameters for these bounces include a small rest length for the springs in the network, so that the trampoline is always under some tension. A nice size for the ball is $2 * a$ or double the distance between adjacent nodes on the trampoline at $t = 0$. Some careful considerations when running the program include scaling the mass of the ball and the mass of the individual nodes on the trampoline in such a way that the ball mass is not too heavy for the trampoline, otherwise the trampoline will collapse. Recommended parameters for the stiffness of the links between nodes and the damping constant can be found in the code below. If the damping constant for the springs is too large when the rest length is near 0, then the interior nodes will break off. Also, it is important to choose a time step small enough for the parameters chosen for the ball. Namely, the larger the quantity K_{ball} or in the code referred to as the ball's imaginary stiffness: $\mathbf{K}_{\text{pen}}$, the smaller the time step dt should be to avoid instability where the system spontaneously gains energy. The final variation in our parameter initializations were the number of trampolines, location of each trampoline, angle of rotation of each trampoline, starting location of the ball, and starting time for ball to be released.

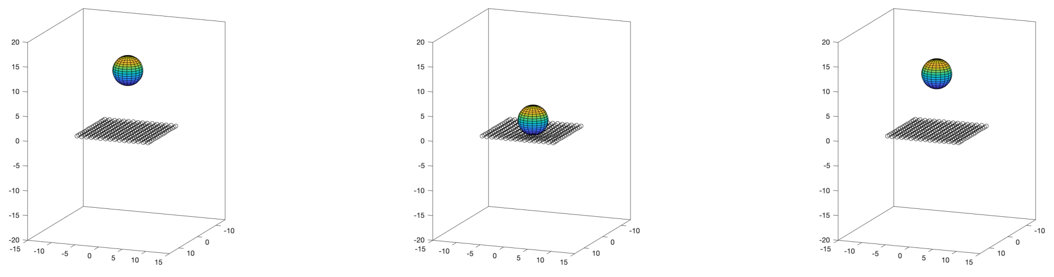

Figure 4: Frames from animation of ball and trampoline at three different time steps.

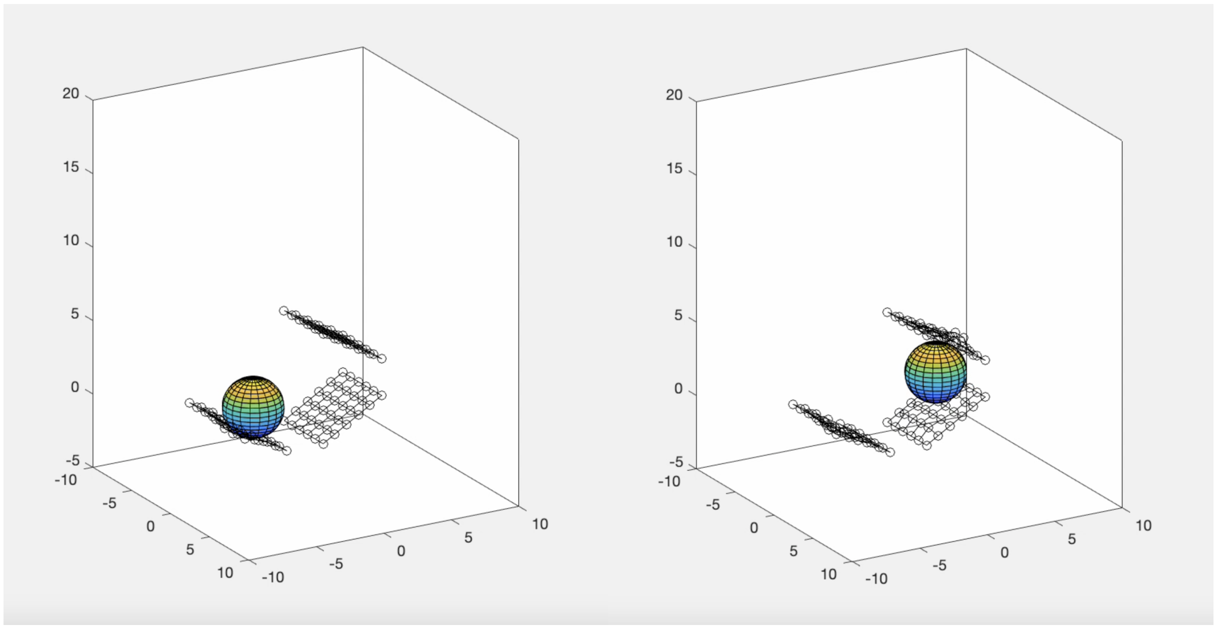
Figure 5: Frames from animation of ball and three trampolines at two different time steps.

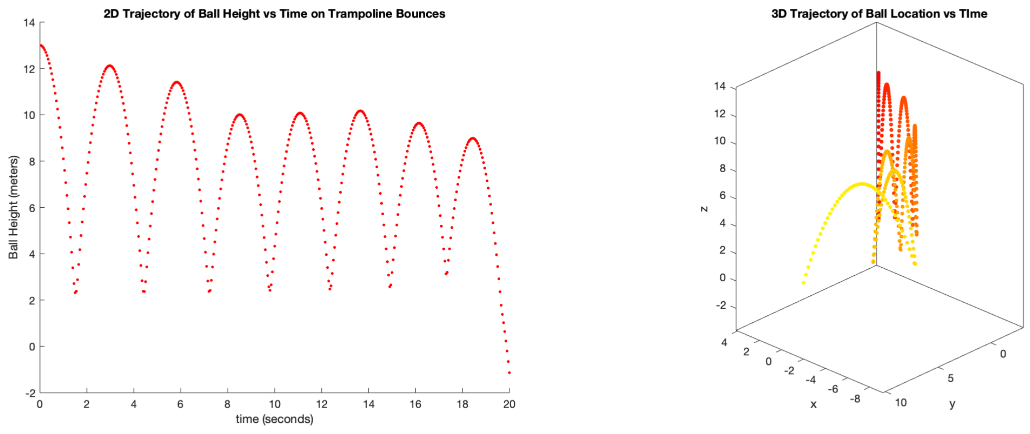

Figure 6: 2D and 3D kinematic trajectories as a function of time of the ball as it bounces on the same simple trampoline until it falls off. In the 3D plot note that red represents the earliest time and yellow represents the latest time.

A Complete code for the project

A.1 plotTram.m

```matlab
function [N,NN,X,jj,kk, fixed] = plotTram(n,a, theta, mtrans)

%This function constructs a square trampoline with n points on one ...
    side of
%the trampoline.
%The trampoline is a square net structure.
%That means the only links are vertical and horizontal, no cross ...
    links will
%be implemented.
%Eventually, the trampoline will look like a net.

%input parameters:

% n = number of vertices on each side, n is at least 3.

if n < 3
    error("There should be at least 3 vertices on each side")
end

%*** then it will be a n-1 by n-1 net.
% a = length of links

%theta is the angle of rotation.
%mtrans = is the 1x3 translation matrix.


%outputs:
% X(k,:) = coordinates of point k
% jj(l),kk(l) = indices of points connected by link l
% Rzero(l) = rest length of link l

% There are kmax points in the grid:
kmax = n^2;

%There are these many links given n*n vertices (proved by induction):


X = zeros(kmax, 3); %Matrix of vertices

%points on trampoline:
for k=1:n
    for j = 1:n
       X((k-1)*n+j,:) = [j*a, k*a, 0];
    end
end

```

```


nodes = (1:kmax)'; %nodes [1 2 3 4 ...]'
nb = zeros(kmax, 4);%the neighbour matrix, each node has at most 4 ...
    neighbours


%construct neighbours
for iy = 1:n
    for ix = 1:n
        idx = ix + n * (iy-1);
        if ix > 1; nb(idx,1) = idx - 1; end%if the node is left most, ...
            it does not have left neighbour
        if ix < n; nb(idx,2) = idx + 1; end%right most nodes
        if iy > 1; nb(idx,3) = idx - n; end
        if iy < n; nb(idx,4) = idx + n; end
    end
end

%the below part is to reshape the neighbour matrix nb into link table
%you can print [nodes X nb] out to see it more clearly.
jj = []; kk = [];
for link = 1:size(nb,2)
  inds = (nb(:,link) > 0);
  jj = [jj;nodes(inds)]; %we actually count each link twice.
  kk = [kk;nb(inds,link)];
end




mtrans_0 = [-(a*(n-1))/2-1 -(a*(n-1))/2-1 0];
X = X + mtrans_0; %translate to origin.




rotation = [        1           0           0
                    0    cos(theta)  -sin(theta)
                    0    sin(theta)   cos(theta)]; %3x3 matrix X = nx3 ...
                        matrix.

X = X * rotation;

X = X + mtrans;%translate to target position.


%fixed = (X(:,1) == 1*a | X(:,1) == n*a | X(:,2) == 1*a | X(:,2) == n*a);
fixed = zeros(kmax,1);

for f = 1:kmax
    if(f ≤ n )  fixed(f) = 1;
    elseif(f > (kmax - n))  fixed(f) = 1;
    elseif(mod(f,n) == 1) fixed(f) = 1;
    elseif(mod(f,n) == 0) fixed(f) = 1;
    end
end

fixed = logical(fixed);

N = length(X);
NN = length(jj);


plot3(X(:,1),X(:,2),X(:,3),'.b','markersize',10);
hold on;
plot3([X(jj,1),X(kk,1)]',[X(jj,2),X(kk,2)]',[X(jj,3),X(kk,3)]','ko-');


axis equal
box on
axis([-10 10 -10 10 -5 20])
```

A.2 testradius.m

```
function [in_out, ndir, DisF] = testradius(C, X, R)
%X: n * 3 matrix
%C: 1 * 3 matrix
%penalty force points from Center to Nodes
in_out = zeros(size(X,1), 1);
ndir = zeros(size(X,1), 3);
DisF = zeros(size(X,1), 3);


for i = 1: size(X,1)
   Dis = sqrt(sum((X(i,:) - C).^2) );
   direction = X(i,:) - C;
   if(Dis <= R) %if it's within the radius
        in_out(i) = true;
        ndir(i,:) =  (direction)./Dis; % calculate direction.
        DisF(i,:) = (R - Dis).* (direction/Dis); %distance with direction


   else
        in_out(i) = false;
        ndir(i,:) = 0;
   end
end
```

A.3 trampoline.m

```
clear all
close all

n = 7; %dimensions
a = 1; %length of each link;
num_tram = 4; %numher of trampolines
kmax = n^2; %size of X
lmax = 4 * n * (n-1); %size of jj
thetax = [+pi/6 -pi/4 +pi/6 -pi/4];
mtrans = [0 +6 0;
          0 -6 4;
          0 -6 -15;
          0 +6 -15];%some fancy structure of four trampolines
%------------------preallocate everything------------------
N0 = zeros(1,num_tram);
NN0 = zeros (1,num_tram);
X0 = zeros(kmax, 3, num_tram );
jj0 = zeros(lmax, 1, num_tram);
kk0 = zeros(lmax, 1, num_tram);
fixed0 = zeros(kmax, 1, num_tram );
U0 = zeros(kmax,3,num_tram); %preallocate velocity of trampoline nodes


%--------------------trampoline mega loop ------------------------
for i = 1:num_tram
    thetax_temp = thetax(i);
    mtrans_temp = mtrans(i,:);
    [N_temp,NN_temp,X_temp,jj_temp,kk_temp,fixed_temp] = ...
       plotTram(n,a,thetax_temp,mtrans_temp);%build the structure
    N0(i) = N_temp;
    NN0(i) = NN_temp;
    X0(:,:,i) = X_temp;
    jj0(:,:,i) = jj_temp;
    kk0(:,:,i) = kk_temp;
    fixed0(:,:,i) = fixed_temp;

end

%------------------global varibale------------------

g = 9.8; %gravity
M = 0.1; %mass of each node
K = 500; %stiffness
Dam = 5;% Damping constant

tmax = 3; %final time
dt = 0.001; %time step
nskip = floor(0.05/dt); %plotting time
clockmax = floor(tmax/dt+0.1);
```

```
R0 = 0; %link lengths

S = K * ones(lmax,1); %stiffness table
D = Dam * ones(lmax,1); %damping table




%------------------ball information --------
C = [0 4.8 7];
Radius = 2*a;
M_ball = 1; %mass of ball
U_ball = zeros(1,3);%initialize ball speed
K_pen = M_ball * 1e5;

t_ball_start = 0; %time that the ball drops (s)
%----------------------------------------




%----------make video------------------

%   video = VideoWriter('easy_rotate','MPEG-4'); %create the video object
%   open(video); %open the file for writing
%   writeVideo(video,getframe(gcf)); %write the image to ...
    v i d e o close (video)


for clock=1:clockmax

    for i = 1:num_tram
        %---------initialization-------------
        N = N0(i);
        NN = NN0(i) ;
        X = X0(:,:,i);
        jj = jj0(:,:,i);
        kk = kk0(:,:,i);
        fixed =logical(fixed0(:,:,i));
        U = U0(:,:,i);
        %-------forward euler's method-----------------
        t = clock*dt; %get current time
        DX = X(jj,:) - X(kk,:); %link vectors
        DU = U(jj,:) - U(kk,:); %link velocity difference vectors
        R = sqrt(sum(DX.^2,2)); %link lengths

        T = (S/2).*(R-R0) + (D./R).*sum(DX.*DU,2); %link tensions
        TR = T./R; %link tensions divided by link lengths
        FF = [TR,TR,TR].*DX; %link force vectors

        F = zeros(N,3); %initialize force array for mass points
        F(:,3) = -M*g;    %apply force of gravity to each link point

        F_ball = zeros(1,3); %initialize force array for mass points
        F_ball(:,3) = -M_ball * g;    %apply force of gravity to each ...
            link point


        % For each link, add its force with correct sign
        % to the point at each end of the link:
        for link=1:NN
            F(kk(link),:) = F(kk(link),:) + FF(link,:);
            F(jj(link),:) = F(jj(link),:) - FF(link,:);
        end
        %-------------test if the ball hits the ...
            trampoline----------------

        [in_out, ndir, DisF]= testradius(C, X, Radius);
        TEST = sum(in_out);
        in_out = logical(in_out);

        if(TEST ≠ 0)
            F(in_out,:) = F(in_out,:) + K_pen .* (DisF(in_out,:));
            F_ball = F_ball - sum(K_pen .* (DisF(in_out,:)),1);
        end

        %-------------------------------------------------------------




        %---------------------ball movements--------------


        U_ball = U_ball + dt*F_ball./[M_ball,M_ball,M_ball]; %update ...
            velocities of all points
        if(t < t_ball_start)
            U_ball = zeros(1,3);%initialize ball speed
        end

        C = C + dt*U_ball; %update positions of all points

        %------------Trampoline movements------------
        U = U + dt*F./[M,M,M]; %update velocities of all points
        U(fixed,:) = 0; %don't move the ground
        X = X + dt*U; %update positions of all points


        %---------------update information-------------------
        X0(:,:,i) = X;
        U0(:,:,i) = U;
    end
    %----------------plot---------------------------------

    clc; fprintf('t = %1.4f\n',t);
    if(mod(clock,nskip)==0)
        figure(1);
        cla;
        for i = 1:num_tram
            X = X0(:,:,i);
            if(i == 1) hold on;end

            %plot3([X(jj,1),X(kk,1)]',[X(jj,2),X(kk,2)]',[X(jj,3),X(kk,3)]','ko-','Mar
            scatter3(X(:,1),X(:,2),X(:,3),'k') %faster to plot, but ...
                not as nice


            [sphere1, sphere2, sphere3] = sphere(20);
            surf(Radius*sphere1 + C(1), Radius*sphere2 + ...
                C(2),Radius*sphere3 + C(3));
            axis equal
            box on
            axis([-15 15 -15 15 -20 20])
            %axis([1 (n+1)*a 1 (n+1)*a -10 +10]);
            %view(90+clock * 1/30,20)
            view(100,20)
        end
        drawnow

%                       writeVideo(video,getframe(gcf)); %write the ...
    image to  v i d e o close (video)

        figure(2)
        hold on;
        %plot3(C(1),C(2),C(3),'.','markersize',10,'color',[1 ...
            clock/clockmax 0])
        plot(clock,C(3),'.r');
        axis equal
        box on
        axis([-15 15 -15 15 -20 20])
        %view(60,20)
        view(clock/4,20)


    end






end
%       close(video)
```

References

[1] Charles S Peskin. Modeling & simulation in science, engineering and economics lecture notes: Dynamics of structures: Networks of springs and dashpots with point masses at the nodes. Sep 2018.