Modeling and Simulating Dice

Nidhi Sinha and Hanna Yu

October 30, 2020

Abstract

We simulate throwing a die on the table by modeling the object as a rigid body. The model tracks translational motion and rotational motion in three dimensions, taking into account forces exerted by the table's friction, stiffness, and damping. Results from the simulation show that to maintain numerical stability, increasing the stiffness of the table requires decreasing the step size of our numerical scheme.

1 Introduction

Following what was taught about rigid-body motion and modeling the table in the course Modeling and Simulation taught by Professor Charles Peskin at New York University, we develop a model that simulates throwing a die on a table. We model both translational motion and rotational motion in three dimensions, taking into account forces exerted by gravity and the table's friction, stiffness, and damping. Note that while we may at times refer to our cube as a die, we do not plot or track individual faces.

In this paper, we will introduce how we configure the die and the table, present our governing equations, and explain our numerical scheme and the way we calculate the inputs of the governing equations. We will also discuss the result of varying stiffnesses in the table. Finally, we suggest ideas for further study. The simulation was programmed using Matlab and the code can be found in the appendix.

2 Configuring the Cube and the Table

2.1 The Cube

We model the die as a cube with uniformly distributed total mass m and side length s. m and s will be the parameters for the cube we create. We will refer to the center of the cube as the center of mass.

It will be useful for us later to calculate the moment of inertia tensor. Luckily, because a cube is a simple shape, a formula exists to calculate the moment of inertia tensor I in terms of its side length:

$$I = (1/6)ms^2$$

Note that I is a scalar, made possible by the fact that the side length of the cube is the same in all three dimensions and the mass is evenly distributed. As a result, the moment of inertia tensor does not change under rotation; it stays constant throughout our simulation, hence we only need to compute it once.

2.2 The Table

We represent the table as a plane through the origin with unit normal n. We let the table be elastic with the following parameters:

$$\begin{aligned} S &= \text{stiffness of the table} \\ D &= \text{damping of the table} \\ \mu &= \text{coefficient of sliding friction} \end{aligned}$$

3 Governing Equations

Our simulation models the cube as a rigid body. The central assumption in rigid bodies is that the motion of the body is only affected by external forces (e.g. gravity, the table). This eliminates any internal forces such as the force individual point masses on the body exert on each other.

Our principal physical equations are:

$$F = ma$$

$$L = I\omega$$

where F denotes the total force exerted on the body, m denotes mass, a is the acceleration, L is the angular moment of the body, I is the moment of inertia, and ω is the angular velocity.

In addition, it is helpful to know these things about the torque τ:

$$dL/dt = \tau$$

$$\tau_k = \tilde{X_k} \times F_k$$

Here $\tilde{X_k}$ denotes the position of corner k relative to the center of mass and F_k denotes the force acted upon corner k.

We manipulate these equations to derive our governing differential equations, which determine the motion of a rigid body. Let x_{cm} and u_{cm} be the position and velocity at the center of mass. For now, assume that $\tilde{X_k}$ and F_k for each moment in time are known:

$$\frac{d}{dt}u_{cm} = F/m = \sum_k F_k/m \tag{1}$$

$$\frac{d}{dt}x_{cm} = u_{cm} \tag{2}$$

$$\frac{d}{dt}L = \tau = \sum_k \tilde{X_k} \times F_k \tag{3}$$

Equation (1) is derived from Newton's Second Law and the definition of acceleration. Equation (2) is the definition of velocity. Equation (3) comes from the definition of torque. These derivatives change when $\tilde{X_k}$ and the forces F_k change.

In addition to these governing equations, to animate our simulation and calculate the interaction with the table later, we need to know the absolute positions X and velocities U of our corners.

Instead of calculating the corner positions and velocities directly, we can derive them from x_{cm}, u_{cm} and the angular velocity ω. Note that knowing L, we can calculate ω easily since $L = I\omega$. So for each corner k,

$$X_k = x_{cm} + \tilde{X_k} \tag{4}$$

$$U_k = u_{cm} + \omega \times \tilde{X_k} \tag{5}$$

.

4 Numerical Scheme

We adapt our governing equations using a modified forward Euler method:

$$u_{cm}(t + \Delta t) = u_{cm}(t) + \Delta t * F/m \tag{6}$$

$$x_{cm}(t + \Delta t) = x_{cm}(t) + \Delta t * u_{cm} \tag{7}$$

$$L(t + \Delta t) = L(t) + \Delta t \sum_k (\tilde{X_k} \times F_k) \tag{8}$$

Note that $F = \sum_k F_k$.

Previously we assumed that $\tilde{X_k}$ and F_k for each moment in time were known. We now calculate the changes which determine $\tilde{X_k}$ and F_k for each time step.

4.1 Updating $\tilde{X_k}$

To update $\tilde{X_k}$, we use the equation

$$\tilde{X_k} = R\tilde{X_k} \tag{9}$$

Here, R is a rotation matrix updating $\tilde{X_k}$ according to how much the body has spun in time Δt:

$$R = P + cos(||\omega||\Delta t)(E - P) + sin(||\omega||\Delta t)\left(\frac{\omega_{cross}}{||\omega||}\right)$$

where

$$P = \frac{\omega}{||\omega||}\left(\frac{\omega}{||\omega||}\right)^\top, \quad E = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \omega_{cross} = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix}$$

4.2 Updating F_k

Now we update the force exerted on corner k. The forces at play are gravitational force and the force exerted by the table:

$$F_k = F_{gravity} + F_{table} \tag{10}$$

We know that $F_{gravity} = mg$, where g is the gravitational constant. Finding the force of the table will take more work.

Any point k that goes below the table experiences a force pushing it up, the normal force F_n, and a tangential force, the friction force F_f. Then the total force exerted by the table is

$$F_{table} = F_n + F_f$$

The normal force experienced by a corner at position X_k with velocity U_k is calculated as

$$F_n = ||F_n||n$$

where $||F_n||$ is the magnitude of the normal force. In our model, we set this to

$$||F_n|| = S\left(-n^\top X_k\right) - D\left(n^\top U_k\right)$$

Recall that S and D are the stiffness and damping constants of the table. Note that $-n^\top X_k$ is a dot product which calculates the corner's distance into the table and $n^\top U_k$ calculates how much of the corner's velocity is pointing into the table.

The force of friction experienced by the corner follows the physical equation

$$F_f = -\mu||F_n||\left(\frac{U_{tan}}{||U_{tan}||}\right)$$

where U_{tan} is the component of U_k in the direction tangent to the table. This is calculated as

$$U_{\text{tan}} = U_k - \left(n^\top U_k\right)n$$

At this point, we can go back to Equation 10 to evaluate the force experienced by a corner k below the table:

$$F_{table} = F_n + F_f$$

We have finished calculating the $\tilde{X}_k$ and F_k needed to update X_k, U_k, and our governing equations.

5 Code

To simulate our model in Matlab, we first wrote a function (cube.m) that initialized the variables describing a cube. One note about the cube we haven't mentioned yet is that our code indexes all eight corners of a cube and also indexes the edges (referred to as links) connecting those corners. Keeping track of links in rigid bodies isn't as important as when modeling objects as a network of springs and dashpots, but it is still useful in simplifying our code for plotting.

In a separate code (real_throw_cube.m), we called the cube function to create the cube. We then added the table, set the initial values for our governing equations, created an initial plot, and set the time variables for our numerical scheme. Note that the initial angular momentum is set to be random with uniform distribution within a specified range in order to mimic the randomness of throwing a die, but the initial angular momentum can also be set to a predetermined value. The for-loop is where we update our variables and animation according to the methods described under Numerical Scheme. We also record the coordinates of the cube's center of mass and the cube's total energy for each time step so we can plot it at the end of our simulation run.

In addition, we have written a program that does not take spinning into account so that we can focus on just the cube's interaction with the table (ground_cube.m). The angular velocity stays constant here. This simplified version of our simulation allows us to better experiment with different parameters for the table. The results of those experiments will be discussed in the next section.

6 Results and Discussion

6.1 Running the Simulation

Running real_throw_cube.m yields an animation of a cube with initial velocity hitting the table a couple times before coming to a stop. For all our tests, we set the radius of the cube to be 1, the total mass to be 1, the initial x_cm to be [0,0,3], the initial u_cm to be [5,0,5], and the run time to be 10 seconds. Figures 1-3 show the simulation at the beginning at the middle, and at the end of a test run.

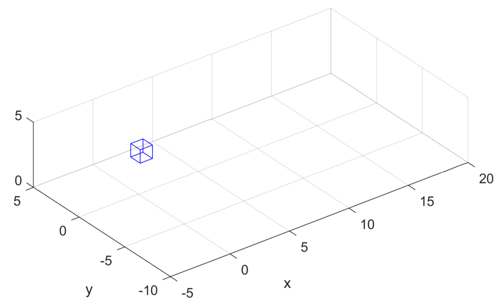

Figure 1: Initial Position

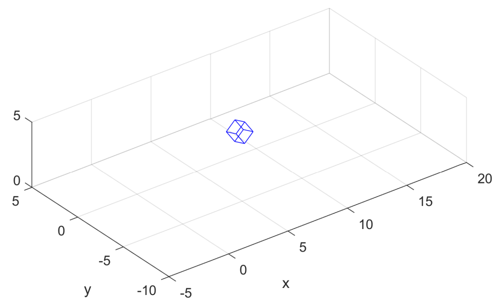

Figure 2: Middle

Figures 4 shows the coordinates of the cube's center of mass in a span of ten seconds for a another similar test run. Notice how the z position (in yellow) shows the cube bouncing from the table, each time with a rebound height lower than the previous. This is due to the table's damping. The x position increases steadily due to the cube's initial velocity in the x direction but tapers off due to friction with the table. Finally, note that while there is no initial velocity in the y direction, the cube ends up at a nonzero y position because spinning causes the cube to bounce from the table at varying angles.

Figure 5 shows the total energy (kinetic energy plus potential energy) of our cube during our simulation. The plot is consistent with the fact that our cube should lose energy through damping

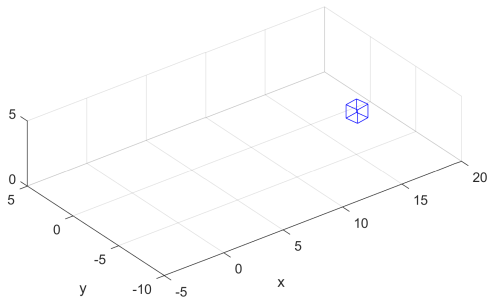

Figure 3: End Position

and friction. Note that the plotted total energy tapers off but does not reach zero; this is because we have calculated potential energy based on the position of the cube's center of mass, and when the cube is at rest at the end of the run, the center of mass is still above the table.

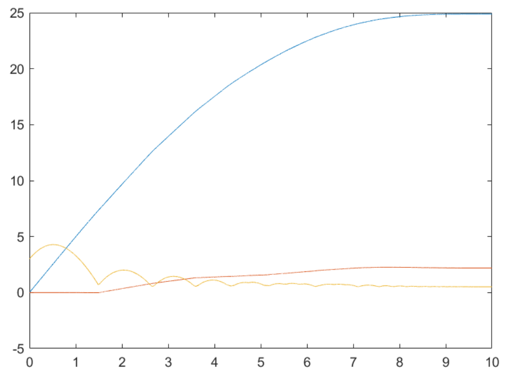

Figure 4: Positions vs Time

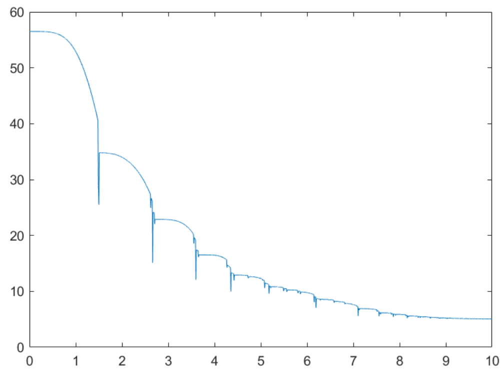

Figure 5: Energy vs Time

6.2 Experimenting with Table Parameters

We used ground_cube.m to consider the effects of changing our table parameters. The original parameters for the table are $S = 10^4$, $D = 0.2$, and $\mu = 0.075$. The number of time steps throughout our run is $clockmax = 10000$ and the total run time is 10.

Figure 6 shows the cube's position if there is no friction. Because of damping, the cube eventually stops bouncing, but the x plot shows that the cube keeps on sliding forever. Figure 7 shows the cube's position if there is friction but no damping. In this case, the cube stops increasing its x position at a certain point, but it never stops bouncing up and down. Essentially, it will keep bouncing in place forever. Figure 8 shows the cube's position when the stiffness is low. Here, the cube dips below 0 on the z axis because the table surface is soft. Figure 9 shows the cube's position when stiffness is high. Keep in mind that the number of time steps is the same as for previous runs.

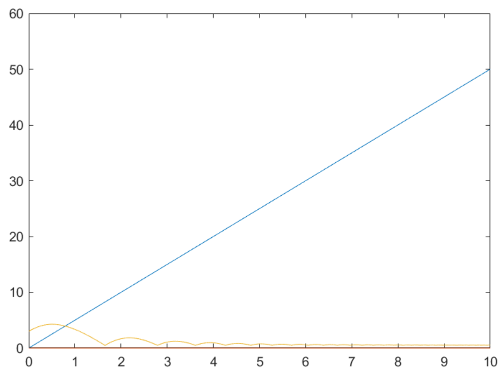

Figure 6: Positions for $\mu = 0$

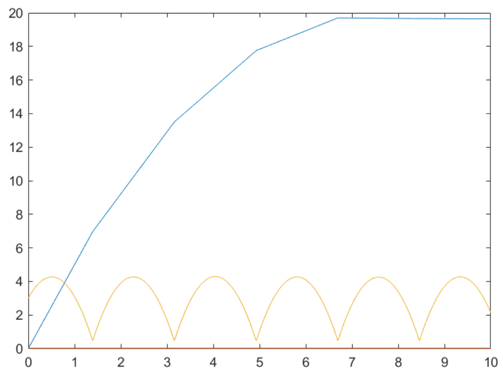

Figure 7: Positions for $D = 0$

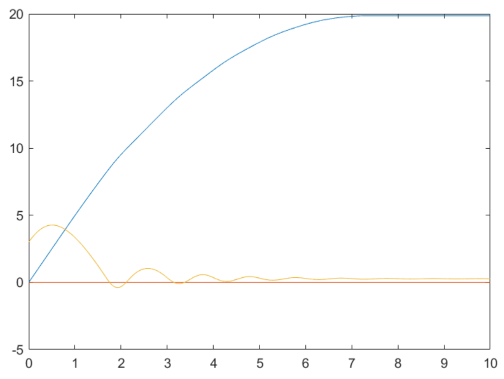

Figure 8: Positions for $S = 10^1$

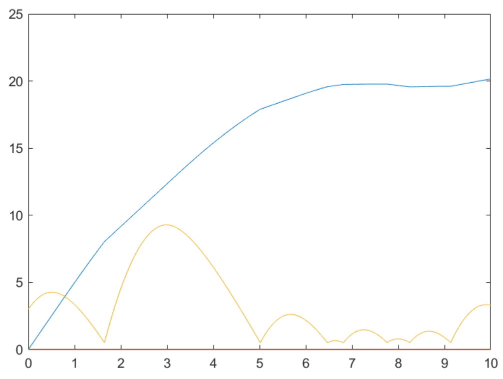

Figure 9: Positions for $S = 10^6$

Figure 9 shows that after the first collision with the table, the cube bounces to a height even higher than before. It seems that our cube may be bounding erratically from the table. Indeed, that's what it looks like when running the animation. Figure 10 shows the corresponding plot for total energy when stiffness is high. We see that the total energy is also erratic, violating the law of conservation of energy. These are signs that something went wrong in our simulation, because these things don't happen in real life.

Whenever working with numerical methods, it is important to verify that we have chosen an appropriate step size. What we realized is that in order to maintain numerical stability, as the stiffness of the table increases, the step size has to decrease. Otherwise, as we have shown, our simulation becomes unstable. Indeed, if we use the same stiffness but halve our step size (by doubling *clockmax*), then our plots look normal again (Figures 11 and 12):

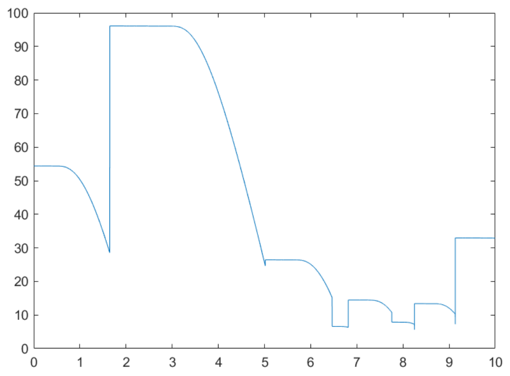

Figure 10: Energy for $S = 10^6$

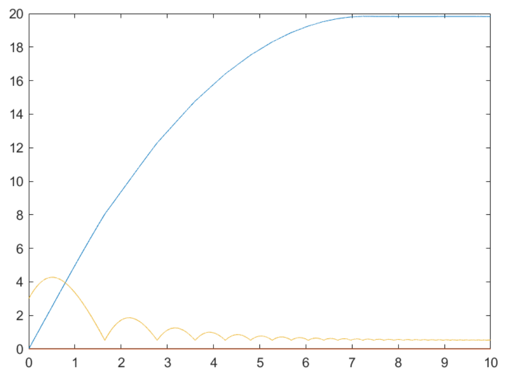

Figure 11: Positions for $S = 10^6$ and $clockmax = 20000$

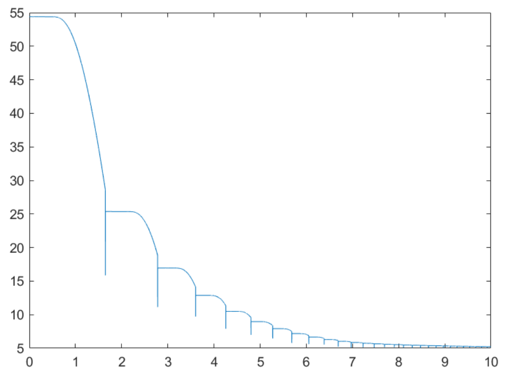

Figure 12: Energy for $S = 10^6$ and $clockmax = 20000$

In Figure 13, we have plotted the number of time steps we need as a function of stiffness. Note that the step size is the run time divided by *clockmax*. We should mention that the time steps needed to achieve accurate outcomes is a little subjective, because even with different "appropriate" step sizes, simulations of the exact same parameters can lead to slightly different cube trajectories.

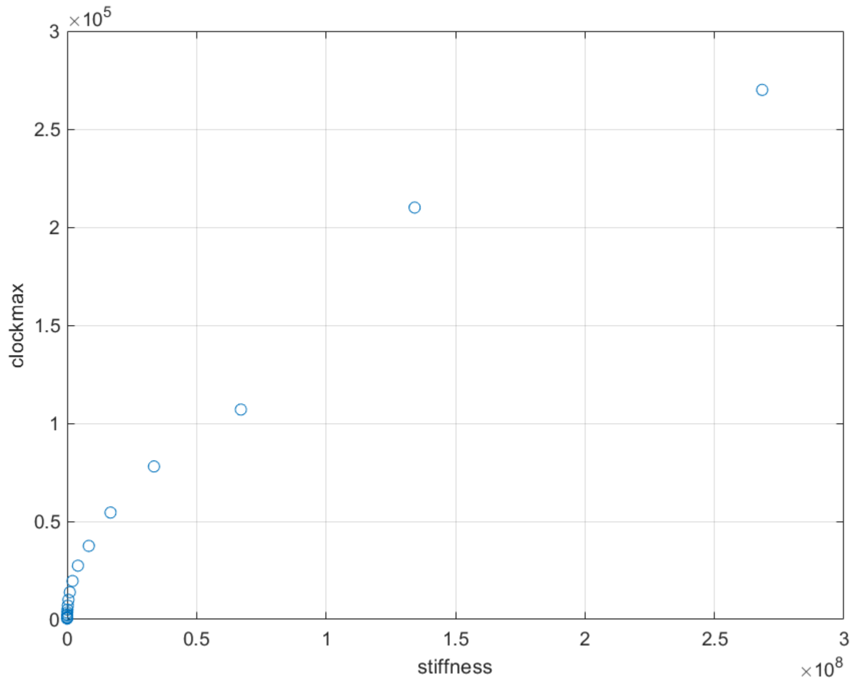

Figure 13

7 Ideas for Further Study

In our study, we have modeled and simulated the trajectory of a spinning cube thrown on an elastic table surface. We verified various physical facts by changing the parameters of the table and we found that to maintain numerical stability, increasing the stiffness of the table requires decreasing the step size.

A natural extension of our simulation is to fully configure our cube as a die by adding and identifying distinct faces to each side. Experiments could track the probability of any side landing face up.

Recall also that in our simulation, we only calculated the moment of inertia tensor once because the properties of our cube lent nicely to a moment of inertia tensor that did not change under rotation. If the mass of our cube was instead unevenly distributed, we would need to recalculate the moment of inertia tensor at every time step. In such a simulation, we could then study the effects of dealing with an unfair die. How would this change the probability of having any side land face up?

Lastly, further study could involve studying how two cubes would interact with each other when thrown onto the table. If they were to bump into each other, how would they react? How would their angular velocities change?

8 Acknowledgements

We'd like to thank Professor Charles Peskin and Scott Weady from the Courant Institute of Mathematical Sciences at New York University for their assistance with this project.

9 References

Charles Peskin lecture on "Dynamics of Rigid Bodies" (Fall 2020)
https://www.math.nyu.edu/faculty/peskin/modsim_lecture_notes/rigid_body_motion.pdf

Charles Peskin lecture on "Interaction of Dynamic Structures with the Ground" (Fall 2020)
https://www.math.nyu.edu/faculty/peskin/modsim_lecture_notes/interaction_with_ground.pdf

Wikipedia article on "List of moments of inertia"
https://en.wikipedia.org/wiki/List_of_moments_of_inertia

10 Appendix

Listing 1: cube.m

```
function [kmax,lmax,X_rel,jj,kk,M,I] = cube(r,total_mass)

%input parameters:
% r = diagonal length from center of mass to corner
% M = total mass

%outputs:
% X_rel(k,:) = coordinates of node k relative to the center of mass
% jj(l),kk(l) = indices of points connected by link l
% M = matrix of point masses
% I = moment of inertia for the cube

kmax = 8;       %8 nodes
lmax = 12;      %12 links
s = -sqrt(2) + 2*sqrt(0.5 + r^2);   %side length

%create position matrix
sign = [1,1,1; 1,-1,1; -1,-1,1; -1,1,1;
```

```
    1,1,−1; 1,−1,−1; −1,−1,−1; −1,1,−1];
X_rel = sign.*s/2;

%fill jj and kk for each link
jj=zeros(lmax,1); kk=zeros(lmax,1);
jj(1) = 1; kk(1) = 2;
jj(2) = 2; kk(2) = 3;
jj(3) = 3; kk(3) = 4;
jj(4) = 4; kk(4) = 1;
jj(5) = 5; kk(5) = 6;
jj(6) = 6; kk(6) = 7;
jj(7) = 7; kk(7) = 8;
jj(8) = 8; kk(8) = 5;
jj(9) = 1; kk(9) = 5;
jj(10) = 2; kk(10) = 6;
jj(11) = 3; kk(11) = 7;
jj(12) = 4; kk(12) = 8;

M = [(total_mass/8)*ones(kmax,1)];  %mass of each point
%moment of inertia tensor
I = diag([(1/6)*total_mass*s^2 (1/6)*total_mass*s^2 (1/6)*total_mass*s^2]);
```

Listing 2: real_throw_cube.m

```
clear all
close all

%% Create the cube
r = 1;
total_mass = 1;
[kmax,lmax,X_rel,jj,kk,M,I] = cube(r,total_mass);

%% Create the ground
%ground will be plane through origin with unit normal [ag,bg,cg]
ag = 0; bg = 0; cg = 1;
nabcg = sqrt(ag^2+bg^2+cg^2);
ag = ag/nabcg; bg = bg/nabcg; cg = cg/nabcg;  %normalize

S_ground = 10^4;    %stiffness of ground (Kg/s^2)
D_ground = 0.1;     %damping of ground (Kg/s^2)
mu_ground = .1;     %friction coefficient of ground

%set cutoff velocity for ground friction to avoid
%division by zero later if tangential velocity is zero
umin = (10^−6)*r*ones(kmax,1);

%% Initial setup
```

```
g = 9.8;
x_cm = [0,0,3];                 %initial position
u_cm = [5,0,5];                 %initial velocity
L = rand(3,1)*0.75 + (10^-6);   %random initial angular momentum
omega = I\L;                    %initial angular velocity

%create matrices for node positions and velocities
X = repmat(x_cm,kmax,1) + X_rel;
U = repmat(u_cm,kmax,1);

%% Initial drawing
figure(1)
link = 1:lmax;
h = plot3( [X(jj(link),1), X(kk(link),1)]', [X(jj(link),2), X(kk(link),2)]',...
           [X(jj(link),3), X(kk(link),3)]', 'b' );
%plot settings
grid on
xlabel('x'); ylabel('y')
axis equal; axis([-5,30,-10,10,0,5])
set(gcf, 'Position', get(0, 'Screensize'));
drawnow

%% Run the simulation
%time and recording
nskip = 200;            %frame iteration for animation
tmax = 10;              %duration of simulation (s)
clockmax = 100000;      %number of time steps
dt = tmax/clockmax;     %step size (s)

%for future plotting
X_save = zeros(clockmax,3);
t_save = zeros(clockmax,1);

for clock=1:clockmax
    t = clock*dt;

    %% Add gravity to each node
    F = zeros(kmax,3);
    F(:,3) = - (total_mass/8)*g;

    %% Add force of ground to each node
    %for each node, evaluate:
    dg = max(0, -(ag*X(:,1)+bg*X(:,2)+cg*X(:,3)) );   %distance into the ground
    un = ag*U(:,1) + bg*U(:,2) + cg*U(:,3);           %velocity normal to ground
    fn = max(0,S_ground*dg - D_ground*un);            %magnitude of normal force

    Utan = U - un*[ag,bg,cg];                           %tangential velocity vector
    Utan_norm = vecnorm(Utan,2,2) + umin;               %norm of tangential velocity
    Utan_dir = Utan./[Utan_norm,Utan_norm,Utan_norm]; %normalize

    %add normal force and friction force
    F = F + fn*[ag,bg,cg] - mu_ground*[fn,fn,fn].*Utan_dir;

    %% Update governing equations
    %update center of mass
    u_cm = u_cm + dt*sum(F,1)./total_mass;
    x_cm = x_cm + dt*u_cm;
    %update angular momentum (changes due to force of ground)
    torque = zeros(3,1);
    for k=1:kmax
        torque = torque + cross(X_rel(k,:),F(k,:))';
    end
    L = L + dt*torque;

    %% Calculate for animation
    %update X_rel due to spinning
    omega = I\L;
    P = (omega./norm(omega))*(omega./norm(omega))';
    omega_cross = [0,-omega(3),omega(2); omega(3),0,-omega(1);
        -omega(2),omega(1),0];
    R = P + cos(norm(omega)*dt)*(eye(3)-P) + ...
        sin(norm(omega)*dt)*(omega_cross/norm(omega));
    for k=1:kmax
        X_rel(k,:) = (R*X_rel(k,:)')';
    end

    %update nodes velocities and positions
    product = zeros(kmax,3);
    for k=1:kmax
        product(k,:) = cross(omega',X_rel(k,:));
    end
    U = repmat(u_cm,kmax,1) + product;  %needed for force of the ground
    X = repmat(x_cm,kmax,1) + X_rel;    %needed for animation

    %% Update animation
    if(mod(clock,nskip)==0)
        c = 0;
        for i=link
            c = c+1;
            h(c).XData = [X(jj(i),1),X(kk(i),1)];
            h(c).YData = [X(jj(i),2),X(kk(i),2)];
            h(c).ZData = [X(jj(i),3),X(kk(i),3)];
        end
        drawnow
    end

    %% Store results for future plotting
    X_save(clock,:) = x_cm;
    t_save(clock) = t;
    kinetic_save(clock) = .5*total_mass*norm(u_cm)^2 + .5*norm(L)^2 / I(1,1);
    potential_save(clock) = total_mass*g*x_cm(3);
    energy_save(clock) = kinetic_save(clock) + potential_save(clock);

end

%% Graph trajectory
figure(2)
plot(t_save',X_save')
figure(3)
plot(t_save',energy_save')
```

Listing 3: ground_cube.m

```
clear all
close all

%% Create the cube
r = 1;
total_mass = 1;
[kmax,lmax,X_rel,jj,kk,M,I] = cube(r,total_mass);

%% Create the ground
%ground will be plane through origin with unit normal [a,b,c]
ag = 0; bg = 0; cg = 1;
nabcg = sqrt(ag^2+bg^2+cg^2);
ag = ag/nabcg; bg = bg/nabcg; cg = cg/nabcg;  %normalize

    S_ground = 10^4;     %stiffness of ground (Kg/s^2)
    D_ground = 0.2;      %damping of ground (Kg/s^2)
    mu_ground = 0.075;   %friction coefficient of ground

%set cutoff velocity for ground friction to avoid
%division by zero later if tangential velocity is zero
umin = (10^-6)*r*ones(kmax,1);

%% Initial setup
g = 9.8;
x_cm = [0,0,3];     %initial position
u_cm = [5,0,5];     %initial velocity
```

```

%create matrices for node positions and velocities
X = repmat(x_cm,kmax,1) + X_rel;
U = repmat(u_cm,kmax,1);

%% Initial drawing
figure(1)
link = 1:lmax;
h = plot3( [X(jj(link),1), X(kk(link),1)]', [X(jj(link),2), X(kk(link),2)]',...
           [X(jj(link),3), X(kk(link),3)]', 'b' );
grid on
xlabel('x')
ylabel('y')
axis equal
axis([-5,30,-5,5,0,5])
drawnow

%% Animation
%time and recording
    nskip = 10;              %frame iteration for animation
    tmax = 10;               %duration of simulation (s)
    clockmax = 10000;        %number of time steps
dt = tmax/clockmax;          %(s)

%for future plotting
X_save = zeros(clockmax,3);
t_save = zeros(clockmax,1);

for clock=1:clockmax
    t = clock*dt;

    %% Add gravity to each node
    F = zeros(kmax,3);
    F(:,3) = - (total_mass/8)*g;

    %% Add force of ground to each node
    %for each node, evaluate:
    dg = max(0, -(ag*X(:,1)+bg*X(:,2)+cg*X(:,3)) );   %distance into the ground
    un = ag*U(:,1) + bg*U(:,2) + cg*U(:,3);          %velocity normal to ground
    fn = max(0,S_ground*dg - D_ground*un);           %magnitude of normal force

    Utan = U - un*[ag,bg,cg];                        %tangential velocity vector
    Utan_norm = vecnorm(Utan,2,2) + umin;            %norm of tangential velocity
    Utan_dir = Utan./[Utan_norm,Utan_norm,Utan_norm]; %normalize

    %add normal force and friction force
    F = F + fn*[ag,bg,cg] — mu_ground*[fn,fn,fn].*Utan_dir;

    %% Update velicities and positions
    %for center of mass
    u_cm = u_cm + dt*sum(F,1)./total_mass;
    x_cm = x_cm + dt*u_cm;

    %for each node
    X = repmat(x_cm,kmax,1) + X_rel;
    U = repmat(u_cm,kmax,1);

    %% Update animation
    if(mod(clock,nskip)==0)
        c = 0;
        for i=link %I don't understand what this means but it works
            c = c+1;
            h(c).XData = [X(jj(i),1),X(kk(i),1)];
            h(c).YData = [X(jj(i),2),X(kk(i),2)];
            h(c).ZData = [X(jj(i),3),X(kk(i),3)];
        end
        drawnow
    end

    %for future plotting
    X_save(clock,:) = x_cm;
    t_save(clock) = t;
    kinetic_save(clock) = .5*total_mass*norm(u_cm)^2;
    potential_save(clock) = total_mass*g*x_cm(3);
    energy_save(clock) = kinetic_save(clock) + potential_save(clock);
end

figure(2)
plot(t_save',X_save')
figure(3)
plot(t_save',energy_save')
```