NUMERICAL MODELLING FOR A BADMINTON-PLAYING ROBOT

Mr D Jhugroo

30th November 2018

Abstract

As part of an artificial intelligence software, it was required to create a program for computing the pitch angle required to hit a shuttlecock from a given initial point to a target point. The problem-solving steps involved deriving ordinary differential equations from FBDs and solving the IVPs and BVPs using Runge-Kutta's method and the shooting method. The trajectory for the flight of the shuttlecock was plotted in 2D, then in 3D and animations were added to enhance the realism of the model.

1. Introduction [1]

The Robomintoner is a badminton-playing robot, developed at the University of Electronic Science and Technology of China. Detecting the shuttlecock and the opponent through its advanced vision system, it can move about the court on its wheels in order to hit the shuttlecock back to the other side of the court. Moreover, a sports technology company is designing a new equivalent robot which uses artificial intelligence (AI) to compete against human players. As part of the AI software, it is required to numerically model the predicted shuttlecock trajectory, so as to intercept incoming shots and aim outgoing shots effectively. Consequently, a MATLAB software needs to be developed to compute the pitch angle required for the shuttlecock to land on a user-defined ground location on the court.

2. Definition of the problem [2]

The task to be accomplished by the software is to calculate the required pitch angle for the robot to hit a shuttlecock such that it lands on the location provided by the user. To demonstrate the functionality of the program, the latter has to consider the condition of a "low serve" where the shuttlecock would be hit at 2.1 m before the net to a distance of 2.1 m after the net.

The impart velocity of the shuttlecock is 75 m/s acting at an angle between 0 and 90 degrees from the horizontal. The shuttlecock has a mass of 5 g and a complicated flight dynamics. The aero-dynamic properties can be simplified into a two-stage model: unstable and stable flight. The unstable stage is a result of the flip from a backwards into a forwards orientation, just after impact, and lasts for 50 ms. During the unstable stage, the average exposed cross-sectional area is 0.012 m^2 and the drag coefficient is 0.8. For the stable stage, this cross-sectional area is 0.009 m^2 while the coefficient of drag is 0.6.

3. Design of a solution

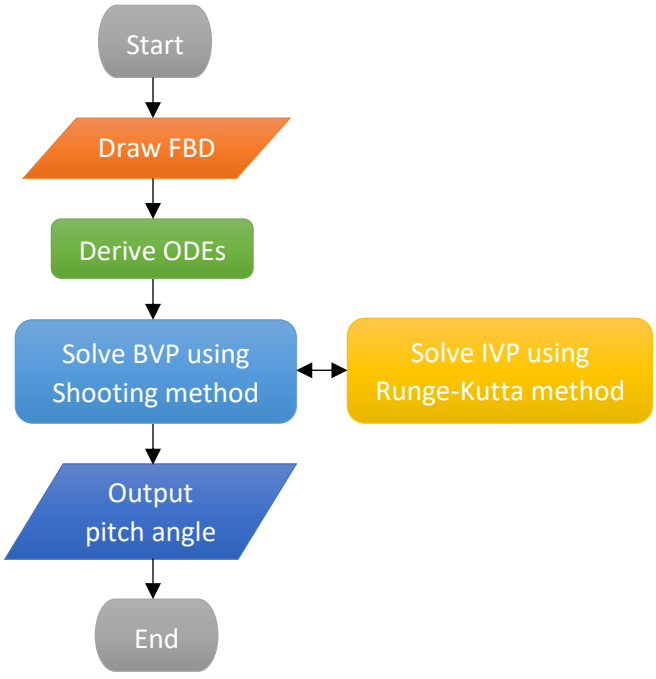

Figure 1: Solution Design Flowchart

The steps to solve the problem is illustrated in *Figure 1*.

4. Synthesizing the problem

4.1. The free-body diagram (FBD) of the motion of the shuttlecock was drawn as shown in *Figure 2*.

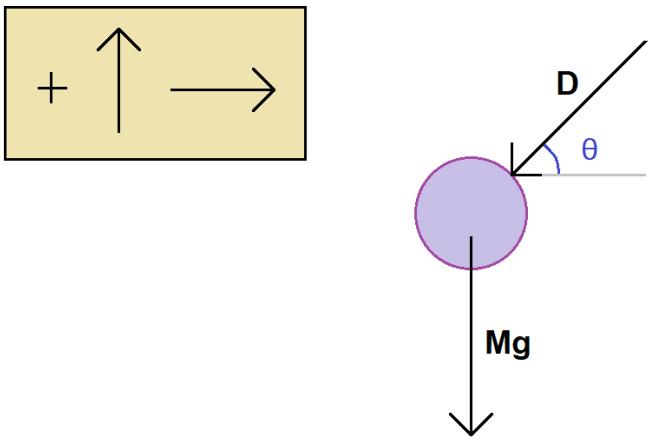

Figure 2: FBD of shuttlecock

$\boldsymbol{M}$ is the mass of the shuttlecock and $\boldsymbol{D}$ is the drag force acting on it at an angle $\boldsymbol{\theta}$ to the horizontal. $\boldsymbol{D}$ is calculated using *Equation 1* [3],

$$\boldsymbol{D = \frac{C\rho AV^2}{2}} \qquad \text{(Equation 1)}$$

where $\boldsymbol{C}$ is the drag coefficient, $\boldsymbol{\rho}$ is the density of air, $\boldsymbol{A}$ is the average cross-sectional area, and $\boldsymbol{V}$ is the velocity of the shuttlecock.

The resolved forces are shown in *Equation 2* and *Equation 3* below:

$$\boldsymbol{Mx'' = -D\cos\theta} \qquad \text{(Equation 2)}$$

$$\boldsymbol{My'' = -D\sin\theta - Mg} \qquad \text{(Equation 3)}$$

where $\mathbf{x''}$ is the acceleration in the x-direction and $\mathbf{y''}$ is the acceleration in the y-direction.

4.2. The ordinary differential equations (ODEs) governing the motions in x and y were derived from the FBD resolved equations as follows:

$$\boldsymbol{x'' = -\frac{D}{M}\cos\theta} \qquad \text{(Equation 4)}$$

$$\boldsymbol{y'' = -\frac{D}{M}\sin\theta - g} \qquad \text{(Equation 5)}$$

Equation 4 and *Equation 5* were obtained by dividing *Equation 2* and *Equation 3* by $\boldsymbol{M}$.

$$\boldsymbol{z = \begin{pmatrix} z_1 = x \\ z_2 = x' \\ z_3 = y \\ z_4 = y' \end{pmatrix}} \qquad \text{(Equation 6)}$$

Decomposing the ODEs in *Equation 4* and *Equation 5* into coupled 1st-order ODEs, *Equation 6* was obtained where $\boldsymbol{z}$ is a state vector containing state variables $\boldsymbol{z_1}$, $\boldsymbol{z_2}$, $\boldsymbol{z_3}$ and $\boldsymbol{z_4}$, where $\boldsymbol{x}$ and $\boldsymbol{y}$ are the displacements, and $\boldsymbol{x'}$ and $\boldsymbol{y'}$ are the velocities in the x and y directions respectively.

$$z' = \begin{pmatrix} z'_1 = z_2 \\ z'_2 = \frac{C\rho AV^2}{2M}\cos\frac{z_2}{V} \\ z'_3 = z_4 \\ z'_4 = \frac{C\rho AV^2}{2M}\sin\frac{z_4}{V} - g \end{pmatrix} \quad \text{(Equation 7)}$$

Taking the derivatives of the state variables of z in *Equation 6*, $\boldsymbol{z'}$ is given by *Equation 7*.

4.3. In MATLAB R2018b, this $\boldsymbol{z'}$ vector was used in a function named ***stateDeriv***. To calculate the pitch angle and plot a graph of the height against the distance travelled by the shuttlecock, 4 more different functions were used: i***vpSolver***, s***tepRungeKutta***, ***shootingMethod*** and ***pathGenerator***.

The ***pathGenerator*** function took in the start location and the target location of the shuttlecock and plotting the trajectory of the shuttlecock on the x and y axes, it also displayed the pitch angle required to achieve the target location. This pitch angle was calculated by the ***shootingMethod*** function and provided to the ***pathGenerator*** function.

In order to calculate the pitch angle, the ***shootingMethod*** worked on the principle of guess and evaluation of error caused by the guess. Initially, the ***shootingMethod*** provided two guesses. Then using the ***ivpSolver*** function, each guess was processed to provide the resulting landing points. Finding the difference between the landing points and the target point, the two errors and guesses were used to compute a third, but this time, intelligent guess. By comparing the error of this new guess and the minimum error required, a new guess was computed in a loop until the new error was nearly zero, thereby outputting the pitch angle.

For the ***shootingMethod*** function to be fed with landing distances, the ***ivpSolver*** took the current value of the guessed angle and start location and passed it through the ***stepRungeKutta*** function which accessed the ***stateDeriv*** function in order to compute the ODEs for a set of time and z vector values – required to plot the graph.

The MATLAB codes for these five functions can be found in *Appendix A*.

5. Assumptions

5.1 To simplify the complicated flight dynamics of the shuttlecock, a basic two-stage model was implemented. This induced some errors as in practice, the cross-sectional area and coefficient of drag varied continuously with the motion of the shuttlecock during the unstable state.

5.2 Moreover, the effect of the environmental wind blows, or droughts was not taken into consideration. As a result, the flight path of the shuttlecock will differ on a physical court.

5.3 The motion of the shuttlecock was modelled to be in a straight forward direction with no deflection to the left or right. Else, for a given pitch angle, the shuttlecock will not reach the target location.

6. Outcome of Solution

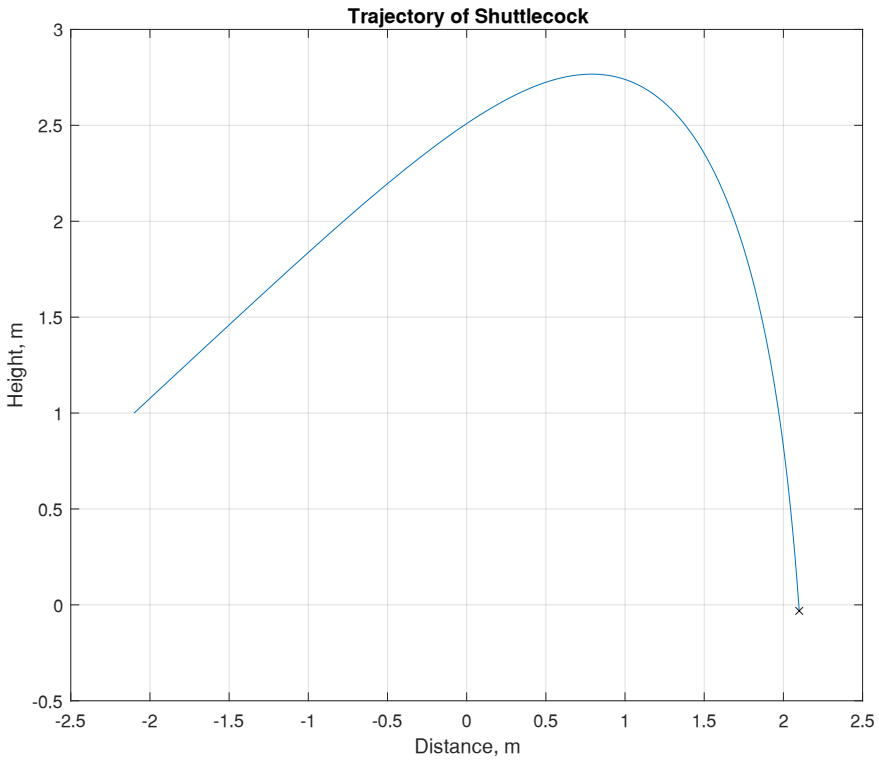

Figure 3: 2D plot of shuttlecock trajectory

Using the ***pathGeneration*** function, the graph shown in *Figure 3* was obtained. The trajectory replicated the required path; starting from 2.1 m before the net at a height of 1 m, to a distance of 2.1 m after the net where the point of impact is marked with an "X". However, the plot did not indicate the points during which the shuttlecock was experiencing an unstable flight due to the flip.

7. Extensions Added

7.1. For the purpose of enabling the user to type in the initial and target locations of the shuttlecock, a graphical user interface (GUI) was implemented using the ***appdesigner*** tool in MATLAB. As shown in *Figure 4*, the GUI features two text boxes: one for the initial location and the other for the target location to be input. To generate the graph, the ***Plot*** button was clicked. Furthermore, the GUI performs a validation check of the input values, where the user is constrained to input a location which is within the length of the court and, to always aim at a target location which is at the other side of the net.

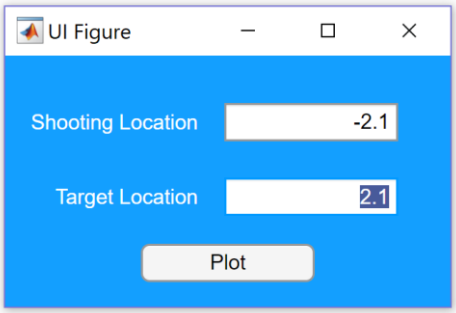

Figure 4: GUI to input initial and target location

7.2. To improve the realism of the software, the trajectory of the shuttlecock was plotted in 3-dimensional (3D) space where the court and the net can be visualised, in addition to the path. A 3D plot featuring the court and the net, can be shown in *Figure 5* below.

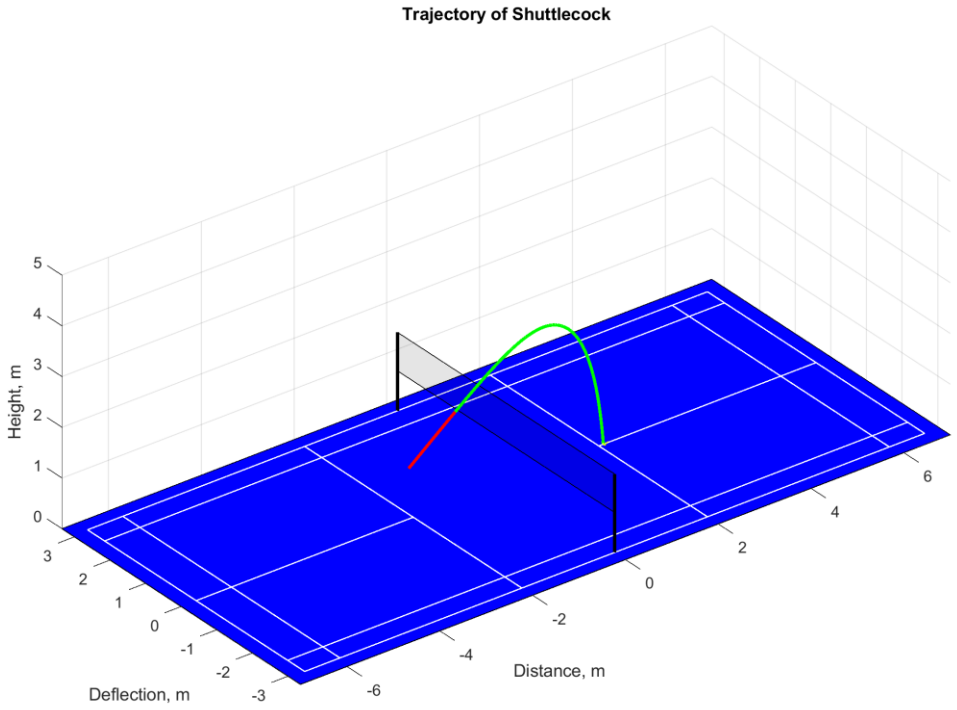

Figure 5: 3D plot of shuttlecock trajectory

To produce the white lines on the court and the net, MATLAB's ***plot3*** function was used together with the ***fill3*** function to fill up the blue rectangle. The code for the 3D plot functions and the ***plotCourt*** and ***plotNet*** scripts are found in *Appendix B.*

7.3. Furthermore, to visualise the period of unstable flight, the path was plotted in two colours: red for unstable flight and green for stable flight. This was done by using an ***if-else*** statement to choose the colour to be plotted.

7.4. As an animation effect, the code can be run to visualise the motion of the shuttlecock. To enable this functionality, a ***for loop*** was used in MATLAB together with the ***pause*** function. This is part of the ***pathGenerator*** function, which is in *Appendix B.*

7.5 The ***pathGenerator*** function also displays several information: the pitch angle, the ground impact speed, and the distance covered when the transition from unstable to stable flight occurs. This is shown below in *Figure 6.*

```
Command Window
  >> TrajectoryGen
  Pitch Angle = 37.5166 degrees
  Ground Impact Speed = 3.791 m/s
  Transition from unstable to stable flight occured 1.0026 m from the hit spot
fx >>
```

Figure 6: Output information

8. References

[1][2] ME20014 Assignment 2, ‘Numerical Modelling for a Badminton-Playing robot’: paper handout. Bath: University of Bath Department of mechanical engineering, 2018.

[3] Glenn Research Centre, NASA, (2016). The Drag Force Equation [Online]. Available from: https://www.grc.nasa.gov/www/k-12/airplane/drageq.html [Accessed 23 November 2018].

Appendices

Appendix A

```
function dz = stateDeriv(t,z)
% stateDeriv    Calculate the state derivative for a shuttlecock trajectory
%
%     DZ = stateDeriv(T,Z) computes the derivative DZ = [Vx;Vy;
%     Ax;Ay] of the state vector Z = [Dx;Dy; Vx;Vy], where D is
%     displacement, V is velocity, and A is acceleration in the x,y axis

% Set constants
M = 5*(10^-3); % Mass (kg)
r = 1.225; % Density of air (kg/m^3)
g = 9.80665; % Acceleration due to gravity (m/s^2)

if t < 0.05
    C = 0.8; % Drag coefficient
    A = 0.012; % Cross-sectional area (m^2)
else
    C = 0.6; % Drag coefficient
    A = 0.009; % Cross-sectional area (m^2)
end

theta = atand((z(4))/(z(2))); % Angle of resultant velocity to the ground
V2 = ((z(2))^2)+((z(4))^2); % Square of resultant velocity

dz1 = z(2); % Velocity in the x-direction
dz2 = -(((C*r*A)/(2*M))*(V2)*cosd(theta)); % Acceleration in the x-
direction
dz3 = z(4); % Velocity in the y-direction
dz4 = -(((C*r*A)/(2*M))*(V2)*sind(theta)) - g; % Acceleration in the y-
direction

dz = [dz1; dz2; dz3; dz4]; % Derivative of state vector z
```

```
function [t,z,d,ImpactSpeed] = ivpSolver(Q, start)
% ivpSolver    Solve initial value problem (IVP) and find target
location
%
%     [T,Z,D,IMPACTSPEED] = ivpSolver(Q, START) computes the IVP
solution
%     using an angle Q and initial location of START to generate the
target
%     location, D, and the impact velocity IMPACTSPEED

% Set initial conditions
v = 75; % Initial velocity (m/s)
t(1) = 0; % First time element of the time vector (seconds)
z(:,1) = [start, v*cosd(Q), 1, v*sind(Q)]; % Initial coordinates

dt = 0.01; % Time step (seconds)

% Continue stepping until the height of the shuttlecock becomes zero
n=1;
while z(3,:) > 0
    % Increment the time vector by one time step
    t(n+1) = t(n) + dt;

    % Apply Runge-Kutta's method for one time step
    z(:,n+1) = stepRungeKutta(t(n), z(:,n), dt);

    n = n+1;

end

d = z(1,end); % Extract target location for output
ImpactSpeed = sqrt(((z(2,end))^2)+((z(4,end))^2)); % Calculate
impact speed
```

```
function znext = stepRungeKutta(t,z,dt)
% stepRungeKutta   Compute one step using the Runge-Kutta method
%
%     ZNEXT = stepRungeKutta(T,Z,DT) computes the state vector ZNEXT
at the next
%     time step T+DT

% Calculate the state derivative from the current state

A = dt*stateDeriv(t, z);
B = dt*stateDeriv(t+(dt/2), z+(A/2));
C = dt*stateDeriv(t+(dt/2), z+(B/2));
D = dt*stateDeriv(t+dt, z+C);


% Calculate the next state vector from the previous one using Runge-
Kutta's
% update equation
znext = z + (A+(2*B)+(2*C)+D)/6;
```

```
function [Q,z,t,ImpactSpeed] = shootingMethod(start, target)
%shootingMethod Calculates the required pitch angle,Q, for the path
%
%   [Q,Z,T,IMPACTSPEED] = shootingMethod(START, TARGET) takes in the
%   initial and target location and computes the values using
ivpSolver to
%   calculate the pitch angle, Q, required for the path

Q1 = 30; % Guess 1 (degrees)
Q2 = 60; % Guess 2 (degrees)

% Computes the error for the first guess
[t,z,d,ImpactSpeed]= ivpSolver(Q1, start); % Calculate target
location
D(1) = d;
G(1) = Q1;
e(1) = D(1) - target;

% Computes the error for the second guess
[t,z,d,ImpactSpeed]= ivpSolver(Q2, start); % Calculate target
location
D(2) = d;
G(2) = Q2;
e(2) = D(2) - target;

n=2;

G(3) = G(n)-((e(n)*(G(n)-G(n-1))/(e(n)-e(n-1)))); % Intelligent
guess

% Computes the error for the third guess
[t,z,d,ImpactSpeed]= ivpSolver(G(3), start); % Calculate target
location
D(3)= d;
e(3) = D(3) - target;

n=3;
while round(e(n),4) ~= 0 % Generates a new intelligent guess until
error is nearly zero

    G(n+1) = G(n)-((e(n)*(G(n)-G(n-1))/(e(n)-e(n-1)))); %
Intelligent guess
    [t,z,d,ImpactSpeed]= ivpSolver(G(n+1), start); % Calculate
target location
    D(n+1) = d;
    e(n+1)= D(n+1) - target; % Computes the error for the (n+1)th
guess

    n = n+1;

end

Q = G(end); % Extract required pitch angle, Q, for output

End
```

```
function [] = pathGenerator(start,target)
%pathGenerator Produces Graph showing shuttlecock's flight
trajectory
%
%   [] = pathGenerator(START,TARGET) plots a graph showing the
flight
%   trajectory of a shuttlecock using START and TARGET in the
%   shootingMethod function to calculate required pitchAngle to
reach
%   target

if (target - start) > 4.55 || (target - start) < 1.46  % Prompt user
to input valid inputs

    disp("Please enter a start location and target location within a
separation between 1.46 and 4.55 m")

else

% Obtain the required pitch angle and impact speed using start and
target
% locations in shootingMethod function
[pitchAngle, z, ~,ImpactSpeed] = shootingMethod(start, target);



% Plot the shuttlecock trajectory
plot(z(1,:),z(3,:))
grid on
hold on
xlabel('Distance, m')
ylabel('Height, m')
title('Trajectory of Shuttlecock')
plot(z(1,end),z(3,end),'kX') % Plot an "X" at the point of landing

disp("Pitch Angle = " + num2str(pitchAngle) + " degrees")    %
Display pitch angle
disp("Ground Impact Speed = " + num2str(ImpactSpeed) + " m/s")  %
Display ground impact speed
end
end
```

Appendix B

```
function dz = stateDeriv_3D(t,z)
% stateDeriv_3D    Calculate the state derivative for a shuttlecock
trajectory in 3D
%
%     DZ = stateDeriv_3D(T,Z) computes the derivative DZ =
[Vx;Vz;Vy;
%     Ax;Az;Ay] of the state vector Z = [Dx;Dz;Dy; Vx;Vz;Vy], where
D is
%     displacement, V is velocity, and A is acceleration in the
x,z,y axis

% Set constants
M = 5*(10^-3); % Mass (kg)
r = 1.225; % Density of air (kg/m^3)
g = 9.80665; % Acceleration due to gravity (m/s^2)

if t < 0.05
    C = 0.8; % Drag coefficient
    A = 0.012; % Cross-sectional area (m^2)
else
    C = 0.6; % Drag coefficient
    A = 0.009; % Cross-sectional area (m^2)
end

theta = atand((z(4))/(z(2))); % Angle of resultant velocity to the
ground
V2 = ((z(2))^2)+((z(4))^2)+((z(6))^2); % Square of resultant
velocity
alpha = 0; % Angle of resultant velocity to the x-axis

dz1 = z(2); % Velocity in the x-direction
dz2 = -(((C*r*A)/(2*M))*(V2)*cosd(theta))*cosd(alpha); %
Acceleration in the x-direction
dz3 = z(4); % Velocity in the z-direction
dz4 = -(((C*r*A)/(2*M))*(V2)*sind(theta)) - g; % Acceleration in the
z-direction
dz5 = z(6); % Velocity in the y-direction
dz6 = -(((C*r*A)/(2*M))*(V2)*cosd(theta))*sind(alpha); %
Acceleration in the y-direction

dz = [dz1; dz2; dz3; dz4; dz5; dz6]; % Derivative of state vector z
```

```
function [t,z,d,ImpactSpeed] = ivpSolver_3D(Q, start)
% ivpSolver_3D    Solve initial value problem (IVP) and find target
location
%
%     [T,Z,D,IMPACTSPEED] = ivpSolver_3D(Q, START) computes the IVP
solution
%     using an angle Q and initial location of START to generate the
target
%     location, D, and the impact velocity IMPACTSPEED

% Set initial conditions
v = 75; % Initial velocity (m/s)
t(1) = 0; % First time element of the time vector (seconds)
z(:,1) = [start, v*cosd(Q), 1, v*sind(Q), 0, 0]; % Initial
coordinates

dt = 0.01; % Time step (seconds)

% Continue stepping until the height of the shuttlecock becomes zero
n=1;
while z(3,:) > 0
    % Increment the time vector by one time step
    t(n+1) = t(n) + dt;

    % Apply Runge-Kutta's method for one time step
    z(:,n+1) = stepRungeKutta_3D(t(n), z(:,n), dt);

    n = n+1;

end

d = z(1,end); % Extract target location for output
ImpactSpeed = sqrt(((z(2,end))^2)+((z(4,end))^2)); % Calculate
impact speed
```

```
function znext = stepRungeKutta_3D(t,z,dt)
% stepRungeKutta_3D   Compute one step using the Runge-Kutta method
%
%     ZNEXT = stepRungeKutta_3D(T,Z,DT) computes the state vector
ZNEXT at the next
%     time step T+DT

% Calculate the state derivative from the current state

A = dt*stateDeriv_3D(t, z);
B = dt*stateDeriv_3D(t+(dt/2), z+(A/2));
C = dt*stateDeriv_3D(t+(dt/2), z+(B/2));
D = dt*stateDeriv_3D(t+dt, z+C);


% Calculate the next state vector from the previous one using Runge-
Kutta's
% update equation
znext = z + (A+(2*B)+(2*C)+D)/6;
```

```
function [Q,z,t,ImpactSpeed] = shootingMethod_3D(start, target)
%shootingMethod_3D Calculates the required pitch angle,Q, for the
path
%
%   [Q,Z,T,IMPACTSPEED] = shootingMethod_3D(START, TARGET) takes in
the
%   initial and target location and computes the values using
ivpSolver_3D to
%   calculate the pitch angle, Q, required for the path

Q1 = 30; % Guess 1 (degrees)
Q2 = 60; % Guess 2 (degrees)

% Computes the error for the first guess
[t,z,d,ImpactSpeed]= ivpSolver_3D(Q1, start); % Calculate target
location
D(1) = d;
G(1) = Q1;
e(1) = D(1) - target;

% Computes the error for the second guess
[t,z,d,ImpactSpeed]= ivpSolver_3D(Q2, start); % Calculate target
location
D(2) = d;
G(2) = Q2;
e(2) = D(2) - target;

n=2;

G(3) = G(n)-((e(n)*(G(n)-G(n-1))/(e(n)-e(n-1)))); % Intelligent
guess

% Computes the error for the third guess
[t,z,d,ImpactSpeed]= ivpSolver_3D(G(3), start); % Calculate target
location
D(3)= d;
e(3) = D(3) - target;

n=3;
while round(e(n),4) ~= 0 % Generates a new intelligent guess until
error is nearly zero

    G(n+1) = G(n)-((e(n)*(G(n)-G(n-1))/(e(n)-e(n-1)))); %
Intelligent guess
    [t,z,d,ImpactSpeed]= ivpSolver_3D(G(n+1), start); % Calculate
target location
    D(n+1) = d;
    e(n+1)= D(n+1) - target; % Computes the error for the (n+1)th
guess

    n = n+1;

end

Q = G(end); % Extract required pitch angle, Q, for output
end
```

```
function [] = pathGenerator_3D(start,target)
%pathGenerator_3D Produces 3D Graph showing shuttlecock's flight
trajectory
%
%   [] = pathGenerator_3D(START,TARGET) plots a 3D graph showing the
flight
%   trajectory of a shuttlecock using START and TARGET in the
%   shootingMethod_3D function to calculate required pitchAngle to
reach
%   target

if (target - start) > 4.55 || (target - start) < 1.46  % Prompt user
to input valid inputs

    disp("Please enter a start location and target location within a
separation between 1.46 and 4.55 m")

else

% Obtain the required pitch angle and impact speed using start and
target
% locations in shootingMethod_3D function
[pitchAngle, z, t,ImpactSpeed] = shootingMethod_3D(start, target);

plotNet;         % Plot the net
plotCourt;       % Plot the Court
axis('equal')

n = size(z,2);  % Obtain the number of elements in the state vector
z

for count=2:n

if t(count) < 0.05
    colour = 'r';   % Set trajectory plot for unstable flight to
colour red
    pauseTime = 0.2;
    transCount = count; % Saves the count value for the transition
point
else
    colour = 'g';   % Set trajectory plot for stable flight to
colour green
    pauseTime = 0.000001;
end

% Plot the shuttlecock trajectory
plot3(z(1,count-1:count), z(5,count-1:count),z(3,count-
1:count),colour,'LineWidth',2)
grid on
zlim([0 5])
xlabel('Distance, m')
ylabel('Deflection, m')
zlabel('Height, m')
title('Trajectory of Shuttlecock')

pause(pauseTime)    % Enable animation effect
```

```
end
plot3(z(1,end),z(3,end),z(5,end),'yX') % Plot the point of landing
as X
hold off
transpoint = num2str((z(1,transCount))-start);
disp("Pitch Angle = " + num2str(pitchAngle) + " degrees")    %
Display pitch angle
disp("Ground Impact Speed = " + num2str(ImpactSpeed) + " m/s")  %
Display ground impact speed
% Display distance travelled upon transition from unstable to stable
flight
disp("Transition from unstable to stable flight occured " +
transpoint + " m from the hit spot")
end

end
```

```
% Net dimensions
NetX = [0 0 0 0 0];
NetY = [-3.05 -3.05 3.05 3.05 -3.05];
NetZ = [0.79 1.55 1.55 0.79 0.79];

% Plotting Net
plot3(NetX,NetY,NetZ,'k','LineWidth',1)
Mesh = fill3(NetX(1:4),NetY(1:4),NetZ(1:4),'k');
alpha(Mesh,0.1)     % Sets translucency of net

hold on

% Plotting Poles
plot3([0 0],[-3.05 -3.05],[0 1.55],'k','LineWidth',2)
plot3([0 0],[3.05 3.05],[0 1.55],'k','LineWidth',2)
```

```
% Court dimensions
Court1X = [-6.68 6.68 6.68 -6.68 -6.68];     % Full court boundary
Court1Y = [3.05 3.05 -3.05 -3.05 3.05];
Court1Z = [0 0 0 0 0];

Court2X = [-6.68 6.68 6.68 -6.68 -6.68];     % Court boundary
excluding side lines for doubles
Court2Y = [2.57 2.57 -2.57 -2.57 2.57];
Court2Z = [0 0 0 0 0];

Court3X = [-5.92 5.92 5.92 -5.92 -5.92];     % Court boundary
excluding long service line for singles
Court3Y = [3.05 3.05 -3.05 -3.05 3.05];
Court3Z = [0 0 0 0 0];

Court4X = [-2 2 2 -2 -2];                    % Court between the 2
short service lines
Court4Y = [3.05 3.05 -3.05 -3.05 3.05];
Court4Z = [0 0 0 0 0];

Court5X = [-6.68 -2 -2 -6.68 -6.68];         % Left side left service
court
Court5Y = [3.05 3.05 0 0 3.05];
Court5Z = [0 0 0 0 0];

Court6X = [2 6.68 6.68 2 2];                 % Right side right
service court
Court6Y = [3.05 3.05 0 0 3.05];
Court6Z = [0 0 0 0 0];

Court7X = [2 6.68 6.68 2 2];                 % Right side left
service court
Court7Y = [0 0 -3.05 -3.05 0];
Court7Z = [0 0 0 0 0];

Court8X = [-6.68 -2 -2 -6.68 -6.68];         % Left side right
service court
Court8Y = [0 0 0 0 0];
Court8Z = [0 0 0 0 0];

% Plotting court

Width = 1; % Width of white lines of court

%Sets Court Colour
fill3([-6.7-0.3,6.7+0.3,6.7+0.3,-6.7-0.3],[3.05+0.3,3.05+0.3,-3.05-
0.3,-3.05-0.3],[0,0,0,0],'b');

plot3(Court1X,Court1Y,Court1Z,'w','LineWidth',Width)
plot3(Court2X,Court2Y,Court2Z,'w','LineWidth',Width)
plot3(Court3X,Court3Y,Court3Z,'w','LineWidth',Width)
plot3(Court4X,Court4Y,Court4Z,'w','LineWidth',Width)
plot3(Court5X,Court5Y,Court5Z,'w','LineWidth',Width)
plot3(Court6X,Court6Y,Court6Z,'w','LineWidth',Width)
plot3(Court7X,Court7Y,Court7Z,'w','LineWidth',Width)
plot3(Court8X,Court8Y,Court8Z,'w','LineWidth',Width)
```