MatLab: Mathematically modelling the motion of a projectile with a parachute

Basic Programme Brief

MatLab was to be used to model the following situation:

A projectile is launched, after **15s a parachute opens**. The projectile must land a **distance of 10,000m** from the launch site. The model was to use the parameters in Table 1 below. The programme is to calculate the required launch angle and final absolute velocity.

Table 1 : Modelling Parameters.

Parameter	Value	Parameter	Value
Exit Velocity , v_0 (m/s)	900	Parachute Frontal Area, A_{pa} (m^2)	1.1
Projectile Mass, m (kg)	600	Parachute Drag Coefficient, C_{pa}	0.9
Projectile Frontal Area, A_{pr} (m^2)	0.25	Air Density at Sea Level, ρ (kg/m^3)	1.207
Projectile Drag Coefficient, C_{pr}	0.38		

Basic system odinary differential equation (ODE) derivation

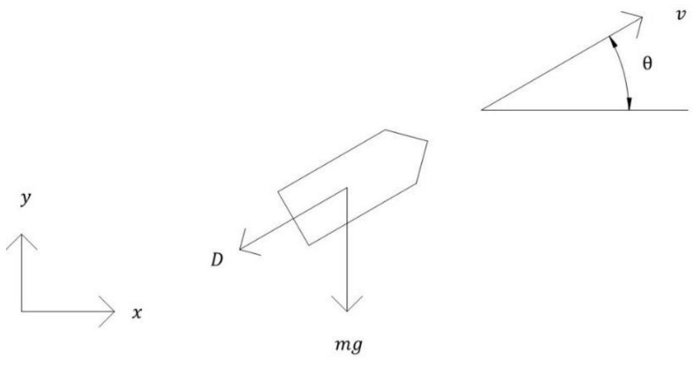


Figure 1: Free body diagram of the basic projectile

Drag of Projectile: $D_{pr} = \frac{1}{2} C_{pr} \rho A_{pr} v^2$

Drag of Parachute: $D_{pa} = \frac{1}{2} C_{pa} \rho A_{pr} v^2$

When the parachute was open the drag values were added to give the total drag

Total Drag: $D_T = D_{pr} + D_{pa} = \frac{1}{2} \rho v^2 (C_{pr} A_{pr} + C_{pa} A_{pa})$

The forces on the projectile, with and without the parachute, were resolved both horizontally and vertically, using Newton's second law, $F = ma$ (Table 2)

Table 2: Projectile ODE equations generated with Newton's second law

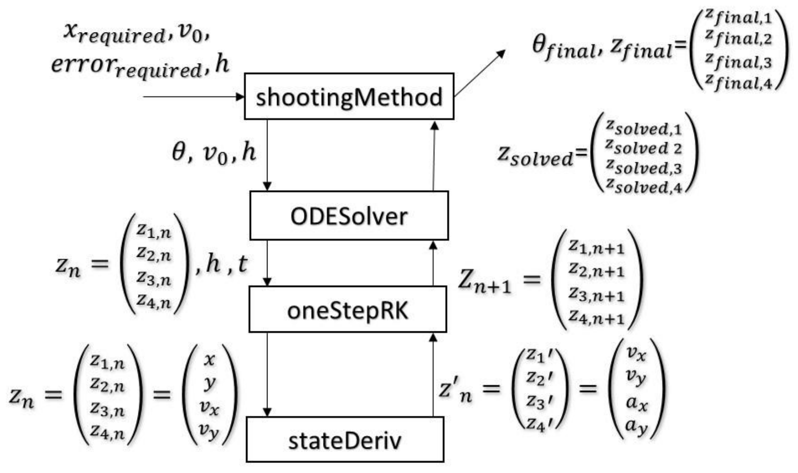
Plane	Variable	Without Parachute	With Parachute
⇨	$a_x = \ddot{x}$	$\frac{-D_{pr} \cos\theta}{m}$	$\frac{-D_T \cos\theta}{m}$
⇧	$a_y = \ddot{y}$	$-g - \frac{D_{pr} \sin\theta}{m}$	$-g - \frac{D_T \sin\theta}{m}$

Where the angle of the projectile at any point, $\theta = \arctan \frac{v_y}{v_x}$

The projectile was therefore modelled as a coupled systems of second order equations in displacement. This system of ODEs was simplified into 4 first order equations to be used with the Runge-Kutta 4 mathematical modelling method.

State Variables	Derivative Variables
$z_1 = x$	${z_1}' = z_2$
$z_2 = x'$	${z_2}' = D \cos\theta$
$z_3 = y$	${z_3}' = z_4$
$z_4 = y'$	$z_4' = -mg - D \sin\theta$

Basic Function



$x_{required}, v_0,$
$error_{required}, h$
shootingMethod
$\theta_{final}, z_{final} = \begin{pmatrix} z_{final,1} \\ z_{final,2} \\ z_{final,3} \\ z_{final,4} \end{pmatrix}$
$\theta,\ v_0, h$
$z_{solved} = \begin{pmatrix} z_{solved,1} \\ z_{solved\ 2} \\ z_{solved,3} \\ z_{solved,4} \end{pmatrix}$
ODESolver
$z_n = \begin{pmatrix} z_{1,n} \\ z_{2,n} \\ z_{3,n} \\ z_{4,n} \end{pmatrix}, h, t$
oneStepRK
$z_{n+1} = \begin{pmatrix} z_{1,n+1} \\ z_{2,n+1} \\ z_{3,n+1} \\ z_{4,n+1} \end{pmatrix}$
$z_n = \begin{pmatrix} z_{1,n} \\ z_{2,n} \\ z_{3,n} \\ z_{4,n} \end{pmatrix} = \begin{pmatrix} x \\ y \\ v_x \\ v_y \end{pmatrix}$
$z'_n = \begin{pmatrix} z_1' \\ z_2' \\ z_3' \\ z_4' \end{pmatrix} = \begin{pmatrix} v_x \\ v_y \\ a_x \\ a_y \end{pmatrix}$
stateDeriv

Variable	Explanation
$x_{required}\ \ (m)$	Required distance
v_0	Initial Launch Velocity
h	Time step used in RK method
$error_{required}$	Allowable distance error.
z_n	(4x1) Current state vector
z'_n	(4x1) Current derivative state vector
z_{n+1}	(4x1) Next state vector
z_{solved}	(4xm) Solved projectile motion state matrix

Figure 2: Flowchart representation of the 4 basic Matlab Scripts:

A brief summary of each basic function is outlined below in Table 3. A full and comprehensive explanation of each function is contained within the code comments in Appendix i.

Table 3: Brief overview of the basic programme functions.

Function	Explanation
Shooting Method (Appendix iA)	Solves the boundary value projectile problem for the launch angle using a series of intelligent guesses using gradient approximation of two initial test angles. $$\theta_3 = \theta_2 + \frac{\theta_2 - \theta_1}{x_2 - x_1}$$ It repeatedly calls ODE solver. A graph of the solution is then plotted.
ODE Solver (Appendix iB)	Solves the initial value projectile problem (IVP) for the launch distance of the projectile given the launch angle, θ passed to it by the shooting method function.
One Step Runge Kutta (Appendix iC)	Takes the initial state vector z_n and generates the next state vector, z_{n+1} after time step, h using the Runge-Kutta 4 method.
State Deriv (Appendix iD)	Takes the current state vector and generates the current derivative state using the 4 first order ODE equations. The derivative state vector is then used in the Runge-Kutta 4 method to generate the next state vector.

The basic programme then plots a graph of the projectile motion from the final state matrix. A plot of the projectile solution to the problem posed in the initial brief is shown in

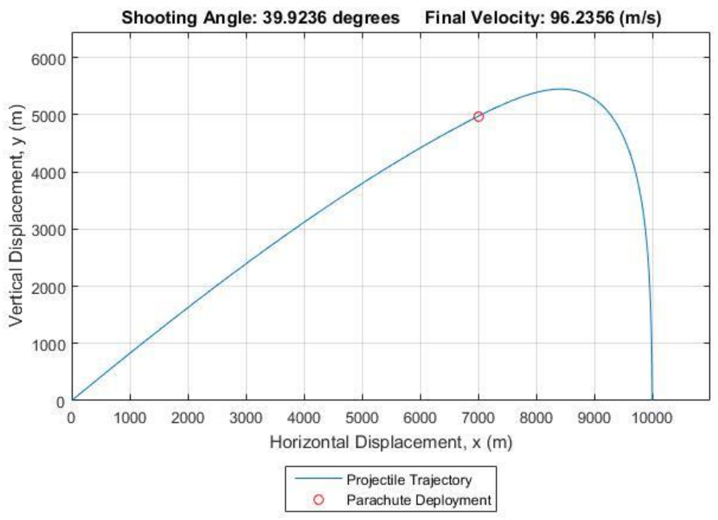


Figure 3: Plot of the assigned BVP solution: $v_0 = 900, x_{required} = 10000,\ h = 0.1,\ error_{required} = 10$

Limitations of the Basic Program:

There were a number of clear limitations to the basic programme in both the underlying maths and the coding approach. These are listed below.

ODE derivation

- **Wind**
- Coriolis Effect
- **Density**
- Curved Earth
- Fixed Area of Projectile

Code:

- **False inputs**
- **Speed**
- **Ease of Use**

The limitations emphasised in bold were addressed in an expansion of the basic programme in the section below.

Programme Improvements

Density

A function was written to incorporate the Earth's atmosphere into the mathematical model. It generates the density of the air at a given point above sea level. It was based on 'NASA's atmosphere model'. Density against height was plotted and shown in Figure 4.

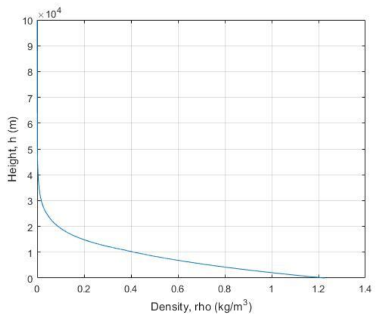


Figure 4: Plot of Density against height used in the improved programme.

Variable Wind

The entire programme was updated to incorporate the concept of a crosswind and a third axis of motion (throughout Appendix ii). During a launch a constant wind at angle, θ_{wind} (always measured from the positive z-axis Figure 5) would affect the motion of the projectile. The drag values would be changed by introducing a relative velocity between the projectile and the wind. It was assumed that the wind would not have any effect on the vertical motion (y-axis) of the projectile and that it would not cause the projectile to rotate.

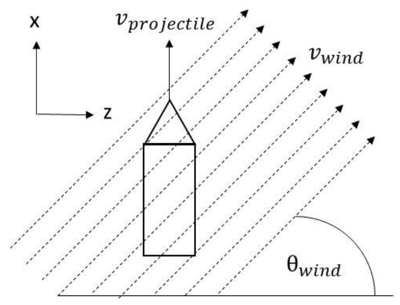


Figure 5: Visualisation of the angle of crosswind on the projectile

3 second order ODE's (Table 4) were derived to represent the new projectile motion using a 3D FBD (Figure 6). Subsequently a system of 6 first order ODEs were then generated. To incorporate the 3D motion all the basic programme functions had to be updated to accept the larger state vectors. To use the updated functions the entire programme had to be modified to pass through the input wind information and the 3 dimensional state vectors.

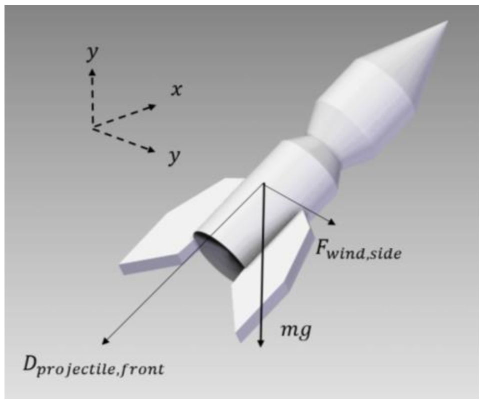


Figure 6: 3D FBD of the projectile.

Table 4: 3D ODEs

Plane	Variable	Without Parachute	With Parachute
x	$a_x = \ddot{x}$	$\frac{C_{side}A_{side}\rho(V_w cos\theta_w)^2}{2m}$	
y	$a_y = \ddot{y}$	$-g - \frac{D_{pr}\sin\theta}{m}$	$-g - \frac{D_T\sin\theta}{m}$
z	$a_z = \ddot{y}$	$\frac{C_{pr}A_{pr}\rho\left(V_{pr} + V_w sin\theta\right)^2}{2m} (= D_{prz})$	$D_{prz} + \frac{C_{pa}A_{pa}\rho\left(V_{pa} + V_w sin\theta\right)^2}{2m}$

Candidate Number: 10079

It was hoped that a 3D shooting method could be coded into the programme to account for wind. However, it became clear that it was not possible to run a shooting method for the distance and then run a second shooting method for the wind compensation angle. This is because the two components are inextricably linked via the derived ODEs and a true 3D shooting method was too difficult to generate.

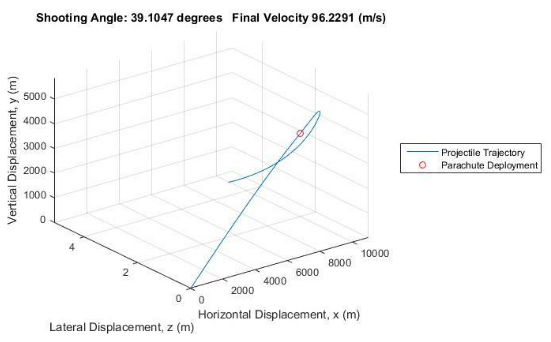


Figure 7: 3D plot of the projectile motion accounting for wind, $v_0 = 900, x_{required} = 10000,\ h = 0.1,\ error_{required} = 10, v_{wind} = 5, \theta_{wind} = 20$

Error Checking

Maximum Distance

An issue with the original code was that if the projectile did not reach the desired target range the shooting method would enter an infinite loop as it continued to look for an impossible angle. An error catch was put in place. A function called maxDisplacement (Appendix ii F) was written to check the maximum launch distance against the desired distance. This was achieved by incrementing shooting tests down from 90° launch angle until the launch distance reached a maximum. If the programme failed the maximum distance check it was halted with a custom error. (shown below)

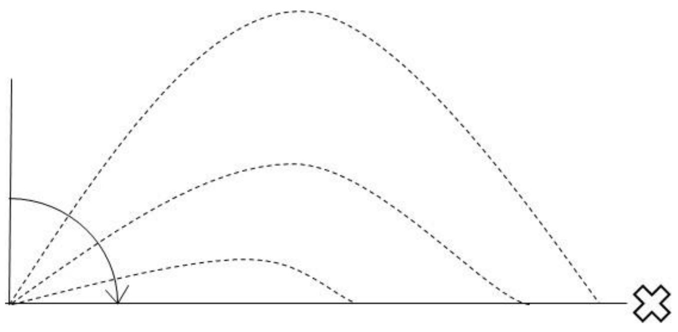


Figure 8: Visualisation of how the maximum possible distance of a launch was calculated.

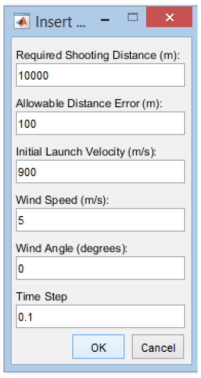
```
Error using shootingMethod (line 28)
Projectile not fast enough to reach destination, increase speed or decrease distance.
```

Intelligent Initial Guesses

For certain distance values, especially lower values, the shooting method would enter an unstable loop where the test shooting angles would diverge from the correct value. As a result the programme would stall or output 'not a number'. This was due to the initial guesses. As computation time was already being used to calculate the maximum distance, the function was expanded to create intelligent overshoot and undershoot guesses to allow the shooting method to run correctly. It operated like maxDisplacement and stepped through possible angles to find the correct initial guess. There was a knock on effect on the timing of the programme but it was far more robust, more than compensating for the short time delay.

Input Box

A basic input dialogue programme was implemented into the programme (shown Figure 9). It had two uses, firstly, the ability to easily input variables into the functions and secondly, it allowed for beginning of programme error checks. Each variable was checked before it was input into the shooting method to prevent unnecessary runs of the software.



Required Shooting Distance (m): 10000
Allowable Distance Error (m): 100
Initial Launch Velocity (m/s): 900
Wind Speed (m/s): 5
Wind Angle (degrees): 0
Time Step 0.1

Figure 9: Input Box

Conclusion

The practicality of a simulator lies in the ability to trust the results output from the simulation. For this reason, the majority of the additional work made to the basic programme was undertaken to improve the robustness of the simulation rather than hiding an error prone code under a flashy graphical user interface. The programming process has shown that when trying to increase the realism of the simulation the increase in coding complexity is exponential. Prematurely opening up the programme to a huge amount of user definable variables would have generated many potential errors. This was especially evident after the incorporation of 3D modelling. It quickly became difficult to link conceptually simple ideas, such as the twisting of the projectile in the wind, to a mathematical model. Hence limiting the ability to realistically model a situation.

Appendices

Appendix i: – Basic Programme

Appendix i A: shootingMethod.m

```
function []= shootingMethod (x_de, e_d, v_0, h)

%INPUTS:
% x_de = Required delivery distance (m)
% e_d  = Desired delivery error (m) ie. to within 100m
% initialise
%Initialise the Guess angles, very inaccurate guess to cover all
%possibilities

%Run two initial shoot guesses, one for undershoot and one for overshoot.
theta_1 = 20*(pi/180); %Guess angle for undershoot
theta_2 = 46*(pi/180); %Guess angle overshoot

%Generate final state matrices of the two test angles and extract final
% x displacement values
z_1 =odeSolver(theta_1, v_0,h);
z_2 =odeSolver(theta_2, v_0,h);
x_1= z_1(end,1);
x_2= z_2(end,1);

%initialise z_3 so that code doesn't error out if the inital guess meets
%the projectile requirements
z_3 = z_2;
theta_3 = theta_2;

%Calculate the initial error
e_c= x_de-x_2;

% While condition compares the current error to the desired error
while abs(e_c) > e_d

    %Calculates the Error state each loop
    e_c = x_de - x_2;

    %Generates an approximation of the next theta angle to test based on a
    %gradient interpretation of the first two test angles
    theta_3 = theta_2 + (e_c*((theta_2 - theta_1)/(x_2- x_1)));

    %Generates a new state vector based on the generated launch angle and
    %extracts the displacement angle
    z_3=odeSolver (theta_3,v_0,h);
    x_3= z_3(end,1);

    %Re-assigns values for use in the next loop iteration.
    x_1=x_2;
    x_2=x_3;
    theta_1=theta_2;
    theta_2=theta_3;

end

%Extracts the x and y state vectors from the final state matrix for plotting
```

```
x= z_3(:,1);
y= z_3(:,2);
theta_final= theta_3 * (180/pi);

%Extracts the the parachute lauch values for plotting of parachute.
t_pa=15;
stepNo= t_pa/h;
x_pa= z_3(stepNo,1);
y_pa= z_3(stepNo,2);

%Graph Generation of final x and y values for use in the axes
x_max = z_3(end,1)+1000;
y_max = max(z_3(:,2))+1000;

%Calculation of the absolute final velocity of the projectile
v_xend= z_3(end,3);
v_yend= z_3(end,4);
v_end = sqrt(v_xend^2 + v_yend^2);

%Plot the final state matrix and the parachute deployment
plot(x,y);
hold on
plot(x_pa,y_pa, 'ro');

%Format Graph
title(['Shooting Angle: ' num2str(theta_final) ' degrees     Final Velocity: '
num2str(v_end) ' (m/s)'] );
grid on
axis ([0 x_max 0 y_max]);
xlabel ('Horizontal Displacement, x (m)');
ylabel ('Vertical Displacement, y (m)');
legend ('Projectile Trajectory', 'Parachute Deployment', 'Location', 'SouthOutside');
hold off
```

Appendix i B: ODESolver.m

```
function [z] = odeSolver(theta, v_0,h)

% Solve the inital value problem ODE surrounding the motion of a projectile
%
% Inputs:
% tlim  = time vector in the form [1 tend] where tend is the end time
% theta = angle of projectile launch (radians)
% v_0   = Absolute launch speed (m/s)
% h     = Time step used for calculations

%Values to be used for Shooting Method Function
tlim    = [1 200];

%Set Initial Launch conditions
x   = 0;                    % x displacement
y   = 0;                    % y displacement
v_x = v_0*cos(theta);       % x velocity
v_y = v_0*sin(theta);       % y velocity

%Enter Launch Conditions into an initial state vector z
z(1,:)=[x y v_x v_y];

% Set initial conditions
t(1) = tlim(1);
```

```
% Continue stepping until the end time is exceeded OR the projectile hits
% the ground
n=1;
while t(n) <= tlim(2)&& z(n,2)>=0
    % Increment the time vector by one time step
    t(n+1) = t(n) + h;

    % Apply Runge Kutta's method for one time step to find the next values
    % and append to the updated state vector to a value matrix

    [z(n+1,:)] = oneStepRK(z(n,:), t(n), h);

    %increment the while loop
    n = n+1;
end;
```

Appendix i C: oneStepRK.m

```
function [znext] = oneStepRK (z,t,h)
% Uses the Runge-Kutta 4 method on a state vector of the form [x y v_x v_y] to generate
the next state vector.

% INPUTS:
% z = state vector input in form [x y v_x v_y]
% t = current time
% h = selected time step

%Calculate the required Runge-Kutta 4 coefficients A, B, C & D
A = h * stateDeriv(z,t);
B = h * stateDeriv(z+(A/2),t+(h/2));
C = h * stateDeriv(z+(B/2),t+(h/2));
D = h * stateDeriv(z+C,t+h);

%Generate the next state vector. This is the function output
znext = z + ((A+(2*B)+(2*C)+D)/6);
```

Appendix i D: stateDeriv.m

```
function [dz] = stateDeriv(z,t)

%Calculates finds out the time derivatives of a projectile state vector of
%the form [x y v_x v_y] and out puts a state derivative vector of the form
%[v_x v_y a_x a_y]

%Inputs:
% z = state vector in form [x y v_x v_y]
% t = current time (s)

v_0  =   900;     % Inital Velocity of projectile (m/s)
m    =   600;     % Mass of Projectile (kg)
A_pr =   0.25;    % Frontal Area of Projectile (m^2)
C_pr =   0.38;    % Drag Coefficient of Projectile
A_pa =   1.1;     % Frontal Area of Parachute (m^2)
C_pa =   0.9;     % Drag Coefficient of Parachute
rho  =   1.207;   % Air Density at sea level (kg/m^3)
g    =   9.81;    % Gravitational Constant

%Extracting Velocity Values from the state vector to make
%code more readable
```

```
v_x = z(3);
v_y = z(4);

%Work out the angle of the rocket at a given point based on the velocity
%values
theta = atan2(v_y,v_x);

%Work out the drag coefficient of the projectile and the parachute
d_nopara     = (C_pr * rho * A_pr)/(2*m);
d_para       = (C_pa * rho * A_pa)/(2*m);

%Choose which drag coefficient to use knowing that the parachute opens after 15s
if t<15
    d= d_nopara;
else
    d=d_para+d_nopara;
end

%Generates the state's absolute velocity for use in drag calculations
v = sqrt((v_x^2) + (v_y^2));

%Generate the differential state vector in the form [v_x v_y dv_x dv_y]
%Using the derived projectile ODES
dv_x= (-1)*d*(v^2)* cos(theta);
dv_y= -g -(d*(v^2)* sin(theta));
dz = [v_x v_y dv_x dv_y];
```

Appendix ii: Modified Programme

Appendix ii A: shootingMethod.m

```
function []= shootingMethod (x_de, e_d, v_0, wind,h)

%Shooting method uses an underlying Runge-Kutta-4 method to
%INPUTS:
% x_de = Required delivery distance (m)
% e_d  = Desired delivery error (m) ie. to within 100m
% initialise

%Extract Velocity and Angle values from the wind vector.
v_w = wind(1);
theta_w = wind(2);

% Begin Error Catching for Wind Angle
if theta_w>360
    while theta_w>360
        theta_w=theta_w -360;
    end
end

%Convert corrected theta_w to radians and redefine the new wind vector
theta_w = theta_w * (pi/180);
wind = [v_w,theta_w];



%Run Error catching processes and generate two intelligent initial guesses
[a, b, c] = maxDisplacement(v_0,wind,h,x_de);
x_max = a;
%Initialise the intelligent guesses, theta_2 is the overshoot
theta_2 = b;
```

```
theta_1 = c;

%Create the initial guess state vectors and fine out the respective distances
z_1 =odeSolver(theta_1, v_0, wind,h);
z_2 =odeSolver(theta_2, v_0, wind,h);
x_1= z_1(end,1);
x_2= z_2(end,1);

%initialise z_3 so that code doesn't error out if the inital guess meets
%the projectile requirements
z_3 = z_2;
theta_3 = theta_2;

%Calculate the initial error
e_c= x_de-x_2;

% While condition compares the current error to the desired error
while abs(e_c) > e_d

    %Calculates the Error state each loop
    e_c = x_de - x_2;

    %Generates an approximation of the next theta angle to test based on a
    %gradient interpretation of the first two test angles
    theta_3 = theta_2 + (e_c*((theta_2 - theta_1)/(x_2- x_1)));

    %Generates a new state vector based on the generated launch angle and
    %extracts the displacement angle
    z_3=odeSolver (theta_3, v_0, wind,h);
    x_3= z_3(end,1);


    %Re-assigns values for use in the next loop iteration.
    x_1=x_2;
    x_2=x_3;
    theta_1=theta_2;
    theta_2=theta_3;

end

% Generates x,y,z coordinates of the Parachute opening using the timestep h
% and the final state matrix
t_pa = 15;
stepNo= t_pa/h;
x_pa= z_3(stepNo,1);
y_pa= z_3(stepNo,2);
z_pa= z_3(stepNo,3);

%Extracts the x and y coordinates from the final state vector for plotting
%purposes
x= z_3(:,1);
y= z_3(:,2);
z= z_3(:,3);
theta_final= theta_3 * (180/pi);

%Works out the final absolute velocity for the graph title this uses 3D
%vector resolution.
v_x= z_3(end,4);
v_y= z_3(end,5);
v_z= z_3(end,6);
v_final= sqrt((v_x^2) +(v_y^2) +(v_z^2));
```

```
%Graph Generation of final x and y and z values for use in the graph axes, NB Z and Y
axis swapped for
%correct Matlab 3D graph visualisation.
x_max = z_3(end,1)+(0.1*z_3(end,1));
y_max = max(z_3(:,2))+(0.1*max(z_3(:,2)));
z_max = z_3(end,3)+(0.1*z_3(end,3));

%Plots the Projectile Motion and the Parachute Opening Point.
plot3(x,z,y);
hold on
plot3(x_pa,z_pa,y_pa, 'ro');

%Establishes the formatting and options in the graph
rotate3d on
title(['Shooting Angle: ' num2str(theta_final) ' degrees   Final Velocity '
num2str(v_final) ' (m/s)'] );
grid on
axis ([0 x_max 0 z_max 0 y_max]);
legend ('Projectile Trajectory', 'Parachute Deployment', 'Location', 'EastOutside');
xlabel ('Horizontal Displacement, x (m)');
zlabel ('Vertical Displacement, y (m)');
ylabel ('Lateral Displacement, z (m)');
hold off
```

Appendix ii B: ODESolver.m

```
function [z] = odeSolver(theta, v_0, wind,h)
% Solve the inital value ODE surrounding the motion of a projectile
%
% Inputs:
% tlim  = time vector in the form [1 tend] where tend is the end time
% theta = angle of projectile launch (radians)
% v_0   = Absolute launch speed (m/s)
% h     = Time step used for calculations
% wind  = Vector of wind information in the form [wind speed, wind angle)
%Values to be used for Shooting Method Function
tlim    = [1 200];

%Set Initial Launch conditions
x   = 0;                    % x displacement
y   = 0;                    % y displacement
z   = 0;                    % z displacement
v_x = v_0*cos(theta);       % x velocity
v_y = v_0*sin(theta);       % y velocity
v_z = 0;                    % z velocity

%Enter Launch Conditions into an initial state vector z
z=[x y z v_x v_y v_z];

% Set initial conditions
t(1) = tlim(1);

% Continue stepping until the end time is exceeded OR the projectile hits
% the ground
n=1;
while t(n) <= tlim(2)&& z(n,2)>=0
    % Increment the time vector by one time step
    t(n+1) = t(n) + h;

    % Apply Runge Kutta's method for one time step to find the next values
    % and append to the updated state vector to a value matrix
    [z(n+1,:)] = oneStepRK(z(n,:), t(n), h, wind);

    %increment the while loop
    n = n+1;
end;
```

Appendix ii C: oneStepRK.m

```
function [znext] = oneStepRK (z,t,h,wind)
% Uses the Runge-Kutta 4 method on a state vector of the form [x y v_x v_y] to generate
the next state vector.

% INPUTS:
% z = state vector input in form [x y z v_x v_y v_z]
% t = current time
% h = selected time step

%Calculate the required Runge-Kutta 4 coefficients A, B, C & D
A = h * stateDeriv(z,t,wind);
B = h * stateDeriv(z+(A/2),t+(h/2),wind);
C = h * stateDeriv(z+(B/2),t+(h/2),wind);
D = h * stateDeriv(z+C,t+h,wind);

%Generate the next state vector. This is the function output
znext = z + ((A+(2*B)+(2*C)+D)/6);
```

Appendix ii D: stateDeriv.m

```
function [dz] = stateDeriv(z,t,wind)

%Calculates finds out the time derivatives of a projectile state vector of
%the form [x y z v_x v_y v_z] and out puts a state derivative vector of the form
%[v_x v_y v_z a_x a_y a_z]

%Inputs:
% z = state vector in form [x y v_x v_y]
% t = current time (s)
%wind = vecotr of wind information in the form of [V_w, theta_w)



%Initialise the wind information to make code more readable
v_w     = wind(1);
theta_w = wind(2);

%Work Out the x and z components of the wind velocity
v_wx = v_w*(sin(theta_w));
v_wz = v_w*(cos(theta_w));

%Initialise the numerical model constants

v_0   =   900;    % Inital Velocity of projectile (m/s)
m     =   600;    % Mass of Projectile (kg)
A_pr =    0.25;   % Frontal Area of Projectile (m^2)
C_pr =    0.38;   % Drag Coefficient of Projectile
A_pa =    1.1;    % Frontal Area of Parachute (m^2)
C_pa =    0.9;    % Drag Coefficient of Parachute
g     =   9.81;   % Gravitational Constant

%Side of the projectile has it's own drag coefficient and area
C_side = 0.30;   % Drag Coefficient of Side Profile
```

```
A_side = 0.30;    % Side Profile Area

%Extracting Displacement and Velocity Values from the state vector to make
%code more readable
y   = z(2);
v_x = z(4);
v_y = z(5);
v_z = z(6);

%Derivation of rho at the current height using NASA's atmosphere model
rho = densityCalculator(y);

%Work out the angle of the rocket at a given point based on the velocity
%values
theta = atan2(v_y,v_x);

%Work out the drag coefficients of the front of the projectile, side of the
%projectile and the parachute.
d_nopara      = (C_pr * rho * A_pr)/(2*m);
d_para        = (C_pa * rho * A_pa)/(2*m);
d_side        = (C_side *rho *A_side)/(2*m);

%Choose which drag coefficient to use knowing that the parachute opens after 15s
if t<15
d= d_nopara;
else
    d=d_para+d_nopara;
end

%Generates the state's absolute velocity for use in drag calculations
v = sqrt((v_x^2) + (v_y^2));

%Generate the differential state vector in the form [v_x v_y v_z dv_x dv_y a_z]
%Using the derived projectile ODES
dv_x= (-1)*d*((v+v_wx)^2)* cos(theta);
dv_y= -g -(d*(v^2)* sin(theta));
dv_z= d_side*(v_wz^2);

%Finalise the current derivative state vector to be passed out of the
%function
dz = [v_x v_y v_z dv_x dv_y dv_z];
```

Appendix ii E: densityCalculator.m

```
function [rho] = densityCalculator (h)
%Calculates the air density at a certain height using Nasa's 'Earth
%atmosphere model https://www.grc.nasa.gov/www/k-12/airplane/atmosmet.html

% initialise variables
T=0; %Temperature (degrees)
p=0; %Pressure (kPa)

if h<11000
    %Values for the Troposhpere
    T   = 15.04 -0.00649*h;
    p   = 101.29 * (((T+273.1)/(288.08))^5.256);

elseif h<25000
    %Values for the lower stratosphere
    T=-56.46;
    p=(22.65*exp(1.73-(0.000157*h)));
```

```
else
    %Values for the upper stratosphere
    T=-131.21+(0.00299*h);
    p=2.488*(((T+273.1)/(216.6))^-11.388);
end
    rho = p/(0.2869*(T+273.1));
```

Appendix ii F: maxDisplacement.m

```
function [x_max, theta_over, theta_under] = maxDisplacement(v_0, wind,h,x_de)

%Function to generate the furthest possible displacement given an initial
%launch velocity. Additionally, as a by-product it outputs intelligent first guess
angles
%for use in the shooting method

%INPUTS
%Launch Velocity, v_0
%Wind Information, wind
%Time Step , h
%Required Shooting Distance, x_de

%OUTPUTS
%x_max: Maximum launch distance
%theta_over: estimate of the projectile overshoot
%theta_under: estimate of the projectile undershoot.


theta = 90*(pi/180); %Initialise theta at 90, the lowest distance launch angle
h_x=2;               %Step in the value of theta (determines the accuracy of the x_max
value)

%Initialise Variables for use in the while loop
    z_1     = odeSolver (theta,v_0, wind,h);
    z_2     = odeSolver (theta-(h*(pi/180)),v_0, wind,h);
    x_1     = z_1 (end,1);
    x_2     = z_2 (end,1);

%Iterates down from 90 degrees using the ODEsolver to find the maximum
%shooting distance.
    while x_2-x_1>0
        theta   = theta-(h_x*(pi/180));
        z_1     = odeSolver (theta,v_0,wind,h);
        z_2     = odeSolver (theta-(h*(pi/180)),v_0,wind,h);
        x_1     = z_1 (end,1);
        x_2     = z_2 (end,1);

    end
    x_max = x_1;

    % Run test to check if the projectile will reach the destination
    if x_de>x_max
        error('Projectile not fast enough to reach destination, increase speed or
decrease distance.')
    end

    % If the projectile can reach the destination, begin a while loop to find an
optimum test angle to use in the
    % shooting method.
    if x_max> x_de
        while x_1>x_2
```

```
            theta = theta+(1*(pi/180));
            z_1      = odeSolver (theta,v_0,wind,h);
            x_1      = z_1 (end,1);
        end
    end

    % Finalise the undershoot and overshoot guesses to be used in the
    % shooting method.
   theta_over = theta;
   theta_under= theta + 0.3;
```

Appendix ii G: shootProjectile.m

```
%shootProjectile is a script to initialise the simulation of a delivery
%projectile

%Construct the Input boxes and default values for the various launch
%parameters
prompt   = {'Required Shooting Distance (m):', 'Allowable Distance Error (m):','Initial
Launch Velocity (m/s):','Wind Speed (m/s):', 'Wind Angle (degrees):', 'Time Step'};
name     =  'Insert Launch Variables';
defaultAnswer  = {'10000','100','900','5','0','0.1'};
numlines = 1;

%Register the inputs from the input box
inputs = inputdlg(prompt,name,numlines,defaultAnswer);
x_de     = str2double(inputs{1});
e_d      = str2double(inputs{2});
v_0      = str2double(inputs{3});
v_w      = str2double(inputs{4});
theta_w  = str2double(inputs{5});
wind     = [v_w theta_w];
h        = str2double(inputs{6});



%Run error checks on the user inputs
if x_de < 0
    error('Please enter a positive input')

elseif e_d > 100
    error('You can get more accurate than that!')

elseif v_0 < 0
    error('Please enter a positive velocity, you are firing away from the target')

elseif v_w > 20
    error('Did not know a hurricane was out today, lower the wind speed!')

elseif theta_w < 0
    error('Negative angle, please set to positive')
end



%Run the Shooting Method to begin the simulation
shootingMethod(x_de, e_d, v_0, wind,h);
```