Table of Contents

Ian Faber - ASEN 2004 Rocket Equation Model 1
Housekeeping 1
Setup - Single flight 1
Simulation - Single flight 2
Setup - Monte Carlo simulation 2
Simulation - Monte Carlo 3
Extraction - Single Flight 3
Plotting 4
rocketEOM Function 6
getConst Function 9
analyzeISP Function 11
analyzeWind Function 12
phase Function 14
color_line3d Function 15
mArrow3 Function (from MATLAB forums) 15

Ian Faber - ASEN 2004 Rocket Equation Model

```
%--------------------------------------------------------------------------
%   By: Ian Faber
%   SID: 108577813
%   Started: 4/3/22, 6:25 PM
%   Finished: 4/11/22, 9:55 PM
%
%   Runs a bottle rocket simulation subject to a set of initial parameters
%   according to delta-V calculated from the rocket equation with ode45 in
%   3 dimensions. The simulation includes wind, but only to control the
%   heading of the rocket (no side forces from wind). A Monte Carlo
%   simulation is alo run to analyze uncertainty in the rocket's landing
%   coordinates.
%
%--------------------------------------------------------------------------
```

Housekeeping

```
clc; clear; close all;

% Choose LA trial to reference
rocketTrial = 1;
```

Setup - Single flight

```
const = getConst(rocketTrial, 0, 0, 0, 0, 1);

% Calculate initial x, y, and z velocities
vx0 = const.deltaV*cosd(const.thetaInit);
```

```
vy0 = 0;
vz0 = const.deltaV*sind(const.thetaInit);

% Format the initial conditions vector, and by extension the variables to
% integrate
X0 = [const.xInit; const.yInit; const.zInit; vx0; vy0; vz0];

% Define events worthy of stopping integration, i.e. hitting the ground
options = odeset('Events', @phase);
```

Simulation - Single flight

```
% Integrate! Solves for the trajectory of the rocket by integrating the
% variables in X0 over tspan according to the derivative information
% contained in rocketEOM. Also stops integration according to "options," a
% predefined set of stopping conditions
[time, state, timePhases, ~, ~] = ode45(@(t,state)rocketEOM(t,state,const, 1),
 const.tspan, X0, options);

% Extract intermediate variables from rocketEOM for debugging, particularly
% weight, drag, friction, and wind. Found this approach on the MATLAB
% forums.
[~,gravCell, dragCell, fricCell, windCell] =
 cellfun(@(t,state)rocketEOM(t,state.',const, 1), num2cell(time),
 num2cell(state,2), 'uni', 0);

%Allocate space for intermediate variables
gravity = zeros(length(time),1);
drag = zeros(length(time),1);
friction = zeros(length(time),1);
wind = zeros(length(time),1);

% Extract intermediate variables from their cells
for i = 1:length(time)
    gravity(i) = norm(gravCell{i});
    drag(i) = norm(dragCell{i});
    friction(i) = norm(fricCell{i});
    wind(i) = norm(windCell{i});
end
```

Setup - Monte Carlo simulation

```
% Run 100 cases
nSims = 100;

% Generate a new contant structure with vectors for each simulation
monteConst = getConst(rocketTrial, 0.005, 0.005, 1, 1, nSims);

% Calculate initial velocities
monteVx0 = monteConst.deltaV.*cosd(monteConst.thetaInit);
monteVy0 = zeros(nSims, 1);
monteVz0 = monteConst.deltaV.*sind(monteConst.thetaInit);
```

Simulation - Monte Carlo

```
% Preallocate structures and arrays for speed
monteX = struct([]);
monteY = struct([]);
monteZ = struct([]);
monteRange = zeros(nSims, 1);
monteCrossRange = zeros(nSims, 1);
monteHeight = zeros(nSims, 1);

% Run simulations with randomized values for various parameters, extract
% trajectories and key performance parameters for plotting
for k = 1:nSims
    monteX0 = [monteConst.xInit, monteConst.yInit, monteConst.zInit,
 monteVx0(k), monteVy0(k), monteVz0(k)];
    [monteTime, monteState, monteTimePhases, ~, ~] =
 ode45(@(t,state)rocketEOM(t,state,monteConst, k), monteConst.tspan, monteX0,
 options);

    monteX{k} = monteState(:,1);
    monteY{k} = monteState(:,2);
    monteZ{k} = monteState(:,3);

    [~, maxR] = max(abs(monteX{k}));
    [~, maxCR] = max(abs(monteY{k}));
    [~, maxH] = max(abs(monteZ{k}));

    monteRange(k) = monteX{k}(maxR);
    monteCrossRange(k) = monteY{k}(maxCR);
    monteHeight(k) = monteZ{k}(maxH);
end
```

Extraction - Single Flight

```
% Extract variables of interest
rocketX = state(:,1);
rocketY = state(:,2);
rocketZ = state(:,3);
rocketVx = state(:,4);
rocketVy = state(:,5);
rocketVz = state(:,6);

% Find maximum values of interest
maxRange = max(rocketX);
maxCrossRange = max(abs(rocketY));
distance = norm([maxRange, maxCrossRange]);
maxHeight = max(rocketZ);
maxVx = max(rocketVx);
maxVy = max(abs(rocketVy));
maxVz = max(rocketVz);

% Calculate drift angle from launch azimuth
```

```
alpha = atand(maxCrossRange/maxRange);

% Equations for plotting north and east lines, as well as the wind
% direction
northLine = @(x) (sind(const.thetaAim)/cosd(const.thetaAim))*x;
eastLine = @(x) (sind(const.thetaAim + 90)/cosd(const.thetaAim + 90))*x;

[~, thetaAloft, thetaGround] = analyzeWind(const, const.hAloft, 1);

windLine = @(x) (sind(const.thetaAim - thetaAloft)/cosd(const.thetaAim -
 thetaAloft))*x;
```

Plotting

```
% Plot the trajectory and variables of interest for the bottle rocket's
% flight!
f = figure();
%f.Position = [100 100 740 740];

% Single Flight Trajectory
%subplot(2,2,1)
hold on;
title("Bottle Rocket Full Trajectory - Ian Faber, Isp Model");
color_line3d(time, rocketX, rocketY, rocketZ);

% Create arrows showing cardinal and wind directions (found on forums)
mArrow3([0, northLine(0), 0], [80, northLine(80),
 0], 'color', 'r', 'stemWidth', 0.25, 'tipWidth', 1);
mArrow3([0, eastLine(0), 0], [80, eastLine(80), 0], 'color', 'g', 'stemWidth',
 0.25, 'tipWidth', 1);
mArrow3([0, northLine(0), 0], [-80, northLine(-80),
 0], 'color', 'b', 'stemWidth', 0.25, 'tipWidth', 1);
mArrow3([0, eastLine(0), 0], [-80, eastLine(-80),
 0], 'color', 'm', 'stemWidth', 0.25, 'tipWidth', 1);
mArrow3([-80, windLine(-80), 0], [80, windLine(80),
 0], 'color', 'c', 'stemWidth', 0.25, 'tipWidth', 1);

% Plot x and y axes
plot3(-80:1:80, zeros(161,1), zeros(161,1), 'k--');
plot3(zeros(161,1), -80:1:80, zeros(161,1), 'k-.');

xlim([-90, 90]);
ylim([-90, 90]);
zlim([0, 30]);
view([0 90]); % Look at XY plane
%view([30 35]);
xlabel("Range (m)");
ylabel("Crossrange (m)");
zlabel("Height (m)");

windLabel = sprintf("Wind aloft: from %s", const.windDirAloft);
```

```
legend("Trajectory", "North", "East", "South", "West", windLabel, "x Axis", "y
 axis")

hold off;

%{
% Drag
subplot(2,2,2)
hold on;
title("Bottle Rocket Drag Force");
plot(time, drag);
xlabel("Time (sec)");
ylabel("Drag (N)");
hold off;

% Wind
subplot(2,2,3)
hold on;
title("Bottle Rocket Wind");
plot(time, wind);
xlabel("Time (sec)");
ylabel("Windspeed (m/s)");
hold off;

% Friction
endTime = 40;
subplot(2,2,4)
hold on;
title("Bottle Rocket Friction from Stand");
plot(time(1:endTime), friction(1:endTime));
xlabel("Time (sec)");
ylabel("Friction (N)");
hold off;
%}


% Monte Carlo Coordinates
g = figure();

% Plot raw landing positions
plot(monteRange,monteCrossRange,'k.','markersize',6)
axis equal;
grid on;
title("Monte Carlo Landing Coordinates - Ian Faber, Isp Model")
xlabel('Range [m]');
ylabel('Crossrange [m]');
hold on;

% Given error ellipse code below, substituted "x" for "monteRange" and "y"
% for "monteCrossRange"

% Calculate covariance matrix
P = cov(monteRange,monteCrossRange);
mean_x = mean(monteRange);
```

```
mean_y = mean(monteCrossRange);

% Calculate the define the error ellipses
n=100; % Number of points around ellipse
p=0:pi/n:2*pi; % angles around a circle

[eigvec,eigval] = eig(P); % Compute eigen-stuff
xy_vect = [cos(p'),sin(p')] * sqrt(eigval) * eigvec'; % Transformation
x_vect = xy_vect(:,1);
y_vect = xy_vect(:,2);

% Plot the error ellipses overlaid on the same figure
sigma1 = plot(1*x_vect+mean_x, 1*y_vect+mean_y, 'b');
sigma2 = plot(2*x_vect+mean_x, 2*y_vect+mean_y, 'g');
sigma3 = plot(3*x_vect+mean_x, 3*y_vect+mean_y, 'r');

legend([sigma1, sigma2, sigma3], "1\sigma", "2\sigma", "3\sigma")

% Monte Carlo Trajectories
h = figure();

title("Monte Carlo Trajectories - Ian Faber, Isp Model")

% Plot all simulated trajectories
for k = 1:nSims
    hold on;
    plot3(monteX{k}, monteY{k}, monteZ{k})
end

% Overlay erorr ellipses on landing
plot(monteRange,monteCrossRange,'k.','markersize',6);
sigma1 = plot(1*x_vect+mean_x, 1*y_vect+mean_y, 'b');
sigma2 = plot(2*x_vect+mean_x, 2*y_vect+mean_y, 'g');
sigma3 = plot(3*x_vect+mean_x, 3*y_vect+mean_y, 'r');

legend([sigma1, sigma2, sigma3], "1\sigma", "2\sigma", "3\sigma")

xlabel("Range (m)");
ylabel("Crossrange (m)");
zlabel("Height (m)");
view([30 35]);

hold off;
```

rocketEOM Function

```
function [dX, fGrav, fDrag, fFric, w] = rocketEOM(t,X,const, k)
% Function that defines the equations of motion and rates of change of
% various variables important for the flight of a water/air propelled
% bottle rocket.
%   Inputs: Time vector, t, state vector, X, formatted as
%   [x;y;z;vx;vy;vz], constant structure, const
%
```

```matlab
%   Outputs: Rates of change, dX, formatted as
%   [vx;vy;vz;ax;ay;az], intermediate weight force variable,
%   fGrav, intermediate drag variable, fDrag
%

    % Extract current state variables
    x = X(1);
    y = X(2);
    z = X(3);
    vx = X(4);
    vy = X(5);
    vz = X(6);

    % Calculate areas of various parts of the bottle
    Abottle = pi*(const.dBottle/2)^2; % m^2

    % Define a velocity vector for heading calculations
    v = [vx; vy; vz];

    % Define a wind velocity vector for heading calculations
    [w, ~, ~] = analyzeWind(const, z, k);

    % State determination for the rocket heading state machine:
    %
    %   "ONSTAND" if the rocket has not travelled the length of the launch
    %   stand, the heading will be fixed at the angle of the launch stand
    %
    %   "FREEFLIGHT" if the rocket has travelled the length of the launch
    %   stand, heading will be free to rotate as the rocket flies through
    %   the air
    %
    if(norm([(x-const.xInit), (y-const.yInit), (z-const.zInit)]) <
 const.lStand)
        headingState = "ONSTAND";
    else
        headingState = "FREEFLIGHT";
    end

    % State determination for the rocket flight phase state machine:
    %
    %   "FRICTION" if the rocket is still contacting the launch stand
    %
    %   "BALLISTIC" if the rocket has left the launch stand
    %
    %   "GROUND" if the rocket's z coordinate aligns with ground level,
    %   which stops the flight and simulation
    %
    if headingState == "ONSTAND" && z > 0
        flightState = "FRICTION";
    elseif headingState == "FREEFLIGHT" && z > 0
        flightState = "BALLISTIC";
    else
        flightState = "GROUND";
    end
```

```
% Rocket heading state machine
switch headingState
    case "ONSTAND"
        % Heading fixed at "thetaInit"
        vRel = v;
        h = [cosd(const.thetaInit(k)); 0; sind(const.thetaInit(k))];
    case "FREEFLIGHT"
        % Heading based on velocity
        vRel = v - w;
        h = vRel/norm(vRel);
    otherwise
        % In case something funky happens ;)
        h = [0; 0; 0];
end

% Rocket flight phase state machine
switch flightState
    case "FRICTION"
        % Calculate weight, drag, and thrust forces
        fGrav = [0; 0; -const.mDry(k)*const.g];
        fDrag = -h*(0.5*const.Cdrag*Abottle*const.rhoAmb*norm(v)^2);
        fFric = -h*(const.muStand*norm(fGrav)*cosd(const.thetaInit(k)));
        fThrust = h*0;

    case "BALLISTIC"
        %Calculate weight, drag and thrust forces
        fGrav = [0; 0; -const.mDry(k)*const.g];
        fDrag = -h*(0.5*const.Cdrag*Abottle*const.rhoAmb*norm(v)^2);
        fFric = [0; 0; 0];
        fThrust = h*0;

    otherwise
        vx = 0;
        vy = 0;
        vz = 0;

        % No forces acting on the rocket, "fGrav" assumed to include
        % normal force from the ground
        fGrav = [0; 0; 0];
        fDrag = [0; 0; 0];
        fFric = [0; 0; 0];
        fThrust = [0; 0; 0];
end

% Calculate the net force on the rocket with equation 1, modified for
% sign convention
fNet = fThrust + fDrag + fGrav + fFric;

% Calculate x and z accelerations
ax = fNet(1) / const.mDry(k);
ay = fNet(2) / const.mDry(k);
az = fNet(3) / const.mDry(k);
```

```
    % Assign rates of change
    dX = [vx; vy; vz; ax; ay; az];

    % Debugging/rocket monitoring stream
    fprintf("Output for simulation %d -- Time: %.3f, Location: [%.3f, %.3f,
 %.3f], Velocity: [%.3f, %.3f, %.3f], Wind Velocity: [%.3f, %.3f, %.3f],
 Relative Velocity: [%.3f, %.3f, %.3f], Heading state: %s, Heading: [%.3f,
 %.3f, %.3f], net force: [%.3f, %.3f, %.3f], flight state: %s\n", k, t, x,
 y, z, vx, vy, vz, w(1), w(2), w(3), vRel(1), vRel(2), vRel(3), headingState,
 h(1), h(2), h(3), fNet(1), fNet(2), fNet(3), flightState);

end
```

getConst Function

```
function const = getConst(rocketTrial, stdMWater, stdMDry, stdThetaInit,
 stdWindSpeed, nSims)
% Defines a constant structure with values outlined in the LA data
% sheets
%
%   Inputs: Which LA rocket trial to pull from, standard deciations for
%           Monte Carlo paramater randomizing
%
%   Outputs: Constant structure, const
%

    const.rocketTrial = rocketTrial;

    [const.ISP, const.stdISP] = analyzeISP(1, stdMWater, nSims);

    const.g = 1*9.81; % Acceleration from gravity, m/s^2

    const.g0 = 9.81; % Acceleration from gravity on Earth, m/s^2

    const.rhoAmb = 1.14; % Ambient air density, kg/m^3

    const.dBottle = 0.105; % Bottle diameter, m

    if rocketTrial == 1 % Jacob Wilson data sheet

        const.mDry = 0.125 + stdMDry*randn(nSims, 1); % Empty bottle mass, kg

        const.mWater = 1 + stdMWater*randn(nSims, 1); % Water mass, kg

        const.Cdrag = 0.2; % Drag coefficient

        const.thetaInit = 45 + stdThetaInit*randn(nSims,1); % Initial angle of
 rocket from ground, deg

        const.thetaAim = 30; % Initial angle of launch azimuth from north, deg

        const.xInit = 0; % Initial downrange distance, m
```

```
        const.yInit = 0; % Initial crossrange distance, m

        const.zInit = 0.25; % Initial vertical height, m

        const.windSpeedGround = (0 + stdWindSpeed*randn(nSims, 1))*0.44704; %
 Ground wind speed, m/s

        const.windDirGround = "N/A"; % Ground wind direction

        const.windSpeedAloft = (10 + stdWindSpeed*randn(nSims, 1))*0.44704; %
 Aloft wind speed, converted from mph to m/s

        const.windDirAloft = "W"; % Direction of wind source


    elseif rocketTrial == 2 % Esther Revenga data sheet

        const.mDry = 0.129 + stdMDry*randn(nSims,1); % Empty bottle mass, kg

        const.mWater = 0.983 + stdMWater*randn(nSims,1); % Water mass, kg

        const.Cdrag = 0.2; % Drag coefficient

        const.thetaInit = 45 + stdThetaInit*randn(nSims,1); % Initial angle of
 rocket from ground, deg

        const.thetaAim = 30; % Initial angle of launch stand from north, deg

        const.xInit = 0; % Initial downrange distance, m

        const.yInit = 0; % Initial crossrange distance, m

        const.zInit = 0.25; % Initial vertical height, m

        const.windSpeedGround = (0 + stdWindSpeed*randn(nSims, 1))*0.44704; %
 Ground wind speed, m/s

        const.windDirGround = "N/A"; % Ground wind direction

        const.windSpeedAloft = (2 + stdWindSpeed*randn(nSims,1))*0.44704; %
 Aloft wind speed, converted from mph to m/s

        const.windDirAloft = "NNW"; % Direction of wind source

    end

    const.hGround = 0;

    const.hAloft = 22;

    const.deltaV = const.ISP*const.g0.*log((const.mDry+const.mWater)./
const.mDry); % Calculated rocket deltaV, m/s

    const.lStand = 0.5; % Length of the test stand, m
```

```
    const.muStand = 0.2; % Coefficient of friction between bottle and launch
 stand (found online, verify)

    const.tspan = [0 5]; % Time span of integration [start stop], sec

end
```

analyzeISP Function

```
function [avgISP, stdISP] = analyzeISP(meanMWater, stdMWater, nSims)
% Function that finds average ISP for a bottle rocket
%   Inputs: Average water mass, standard deviation of water mass, number of
%           simulations to run
%   Outputs: Average ISP, ISP standard deviation

    sampFreq = 1652; % Sampled at 1.652 kHz
    maxTime = 5; % Maximum allowable time for data start and stop calcs
    relError = 0.05; % Relative error between thrust averages for start and
 stop calcs
    g0 = 9.81; % Freefall acceleration on Earth
    mWater = meanMWater + stdMWater*randn(nSims,1); % Mass of water propellant
 (1000 g)

    % Preallocate for speed
    avgISP = zeros(nSims, 1);
    stdISP = zeros(nSims, 1);

    % Data Extraction from test data files
    numFiles = 16;

    for i = 1:numFiles
        files{i} = dir(['.\Static Test Stand Data\Fixed Mass
\LA_Test_FixedMass_Trial',num2str(i)]);
        files{i}.path = ['.\Static Test Stand Data\Fixed Mass
\LA_Test_FixedMass_Trial',num2str(i)];
    end

    % Data Processing from files
    for i = 1:numFiles
       files{i}.data = 4.44822*load(files{i}.path); %Convert from lbf to N
       files{i}.thrust = files{i}.data(:,3);
       files{i}.peakThrust = max(files{i}.thrust);
       files{i}.time = (linspace(0,length(files{i}.thrust)/
sampFreq,length(files{i}.thrust)))'; % Compute time based on sampling
 frequency

       % Calculate a 5-value moving average for detecting trend changes
       thrustAvg = (files{i}.thrust(1:end-4) + files{i}.thrust(2:end-3)
 + files{i}.thrust(3:end-2) + files{i}.thrust(4:end-1) +
 files{i}.thrust(5:end))/5;
```

```
        % Check for abrupt changes in the data and if time is less than 5
seconds
        conditionStart = ischange(thrustAvg) & files{i}.time(1:end-4) <
maxTime;

        % Check to see if data changes by less than relError*100% and if time
is less than 5 seconds
        conditionStop = thrustAvg(1:end-1)./thrustAvg(2:end) - 1 > relError &
files{i}.time(1:end-5) < maxTime;

        start(i) = find(conditionStart, 1, 'first');
        stop(i) = find(conditionStop, 1, 'last');

        files{i}.thrustTime = files{i}.time(stop(i))- files{i}.time(start(i));
    end

    % Calculate Isp for each simulation
    for k = 1:nSims
        %fprintf("Setting up simulation %d of %d... Standby...\n", k, nSims);

        for i = 1:numFiles
            files{i}.waterFlow = mWater(k)/files{i}.thrustTime; % Calculate
mass flow rate of the water
            files{i}.impulse = trapz(files{i}.time(start(i):stop(i)),
files{i}.thrust(start(i):stop(i)) -
files{i}.waterFlow.*files{i}.time(start(i):stop(i)));
            files{i}.isp = files{i}.impulse./(mWater(k)*g0);

            isp(i,1) = files{i}.isp;
        end

        avgISP(k) = mean(isp);
        stdISP(k) = std(isp);
    end
end
```

analyzeWind Function

```
function [w, thetaAloft, thetaGround] = analyzeWind(const, height, k)
% Function that determines the wind velocity vector based on height
%   Inputs: constant structure (wind directions and speeds), current rocket
%           height, current simulation index
%   Outputs: wind velocity vector [wx; wy; wz]
%
%   If wind is "north," it blows from the north to the south

switch const.windDirGround
    case "N"
        thetaGround = 0; % Deg
    case "NNE"
        thetaGround = 22.5; % Deg
    case "NE"
        thetaGround = 45; % Deg
```

```
    case "ENE"
        thetaGround = 67.5; % Deg
    case "E"
        thetaGround = 90; % Deg
    case "ESE"
        thetaGround = 112.5; % Deg
    case "SE"
        thetaGround = 135; % Deg
    case "SSE"
        thetaGround = 157.5; % Deg
    case "S"
        thetaGround = 180; % Deg
    case "SSW"
        thetaGround = 202.5; % Deg
    case "SW"
        thetaGround = 225; % Deg
    case "WSW"
        thetaGround = 247.5; % Deg
    case "W"
        thetaGround = 270; % Deg
    case "WNW"
        thetaGround = 292.5; % Deg
    case "NW"
        thetaGround = 315; % Deg
    case "NNW"
        thetaGround = 337.5; % Deg
    otherwise
        thetaGround = NaN;
end

switch const.windDirAloft
    case "N"
        thetaAloft = 0; % Deg
    case "NNE"
        thetaAloft = 22.5; % Deg
    case "NE"
        thetaAloft = 45; % Deg
    case "ENE"
        thetaAloft = 67.5; % Deg
    case "E"
        thetaAloft = 90; % Deg
    case "ESE"
        thetaAloft = 112.5; % Deg
    case "SE"
        thetaAloft = 135; % Deg
    case "SSE"
        thetaAloft = 157.5; % Deg
    case "S"
        thetaAloft = 180; % Deg
    case "SSW"
        thetaAloft = 202.5; % Deg
    case "SW"
        thetaAloft = 225; % Deg
    case "WSW"
```

```
        thetaAloft = 247.5; % Deg
    case "W"
        thetaAloft = 270; % Deg
    case "WNW"
        thetaAloft = 292.5; % Deg
    case "NW"
        thetaAloft = 315; % Deg
    case "NNW"
        thetaAloft = 337.5; % Deg
    otherwise
        thetaAloft = NaN;
end

% Do some sanity checks, also account for last comment in function header
if isnan(thetaGround)
    thetaGround = 0;
else
    thetaGround = thetaGround - 180;
end

if isnan(thetaAloft)
    thetaAloft = 0;
else
    thetaAloft = thetaAloft - 180;
end

%theta = (thetaAloft/const.hAloft)*(height-const.hGround) + thetaGround;

theta = thetaAloft;

% ALL THETAS UP TO HERE FROM NORTH, NEED TO REORIENT TO X AXIS

%w = ((const.windSpeedAloft / const.hAloft)*(height - const.hGround) +
 const.windSpeedGround)*[cosd(const.thetaAim - theta); sind(const.thetaAim -
 theta); 0];

w = const.windSpeedAloft(k)*[cosd(const.thetaAim - theta); sind(const.thetaAim
 - theta); 0];

end
```

phase Function

```
function [value, isterminal, direction] = phase(t,X)
% Define events of interest to ODE45, specifically integration termination
% events
%   Inputs: Time vector, t, and state vector, X, formatted as
%   [x;z;vx;vz;m;mAir;Vair]
%
%   Outputs: Value to watch, value, whether the value will terminate
%   integration, isterminal, and what direction to watch the value change,
%   direction
%
```

```
    %Extract current height
    z = X(3);

    value = z; % which variable to use (hint, this indicates when the z-
coordinate hits the GROUND level, not sea level)
    isterminal = 1; % terminate integration? (0 or 1)
    direction = -1; % test when the value first goes negative (-1) or positive
 (+1)
end
```

color_line3d Function

```
function h = color_line3d(c, x, y, z)
% color_line3 plots a 3-D "line" with c-data as color
%
%       color_line3d(c, x, y)
%
%  in:  x      x-data
%       y      y-data
%       z      z-data
%       c      coloring
%

    h = surface(...
      'XData',[x(:) x(:)],...
      'YData',[y(:) y(:)],...
      'ZData',[z(:) z(:)],...
      'CData',[c(:) c(:)],...
      'FaceColor','interp',...
      'EdgeColor','interp',...
      'Marker','none', ...
      'LineWidth',2);

    colorbar;

end
```

mArrow3 Function (from MATLAB forums)

```
function h = mArrow3(p1,p2,varargin)
%mArrow3 - plot a 3D arrow as patch object (cylinder+cone)
%
% syntax:   h = mArrow3(p1,p2)
%           h = mArrow3(p1,p2,'propertyName',propertyValue,...)
%
% with:     p1:         starting point
%           p2:         end point
%           properties: 'color':      color according to MATLAB specification
%                                     (see MATLAB help item 'ColorSpec')
%                       'stemWidth':  width of the line
%                       'tipWidth':   width of the cone
%
```

```
%           Additionally, you can specify any patch object properties. (For
%           example, you can make the arrow semitransparent by using
%           'facealpha'.)
%
% example1: h = mArrow3([0 0 0],[1 1 1])
%           (Draws an arrow from [0 0 0] to [1 1 1] with default properties.)
%
% example2: h = mArrow3([0 0 0],[1 1
 1],'color','red','stemWidth',0.02,'facealpha',0.5)
%           (Draws a red semitransparent arrow with a stem width of 0.02
 units.)
%
% hint:     use light to achieve 3D impression
%
    propertyNames = {'edgeColor'};
    propertyValues = {'none'};
    % evaluate property specifications
    for argno = 1:2:nargin-2
        switch varargin{argno}
            case 'color'
                propertyNames = {propertyNames{:},'facecolor'};
                propertyValues = {propertyValues{:},varargin{argno+1}};
            case 'stemWidth'
                if isreal(varargin{argno+1})
                    stemWidth = varargin{argno+1};
                else
                    warning('mArrow3:stemWidth','stemWidth must be a real
 number');
                end
            case 'tipWidth'
                if isreal(varargin{argno+1})
                    tipWidth = varargin{argno+1};
                else
                    warning('mArrow3:tipWidth','tipWidth must be a real
 number');
                end
            otherwise
                propertyNames = {propertyNames{:},varargin{argno}};
                propertyValues = {propertyValues{:},varargin{argno+1}};
        end
    end
    % default parameters
    if ~exist('stemWidth','var')
        ax = axis;
        if numel(ax)==4
            stemWidth = norm(ax([2 4])-ax([1 3]))/300;
        elseif numel(ax)==6
            stemWidth = norm(ax([2 4 6])-ax([1 3 5]))/300;
        end
    end
    if ~exist('tipWidth','var')
        tipWidth = 3*stemWidth;
    end
    tipAngle = 22.5/180*pi;
```

```
    tipLength = tipWidth/tan(tipAngle/2);
    ppsc = 50;  % (points per small circle)
    ppbc = 250; % (points per big circle)
    % ensure column vectors
    p1 = p1(:);
    p2 = p2(:);
    % basic lengths and vectors
    x = (p2-p1)/norm(p2-p1); % (unit vector in arrow direction)
    y = cross(x,[0;0;1]);    % (y and z are unit vectors orthogonal to arrow)
    if norm(y)<0.1
        y = cross(x,[0;1;0]);
    end
    y = y/norm(y);
    z = cross(x,y);
    z = z/norm(z);
    % basic angles
    theta = 0:2*pi/ppsc:2*pi; % (list of angles from 0 to 2*pi for small
circle)
    sintheta = sin(theta);
    costheta = cos(theta);
    upsilon = 0:2*pi/ppbc:2*pi; % (list of angles from 0 to 2*pi for big
circle)
    sinupsilon = sin(upsilon);
    cosupsilon = cos(upsilon);
    % initialize face matrix
    f = NaN([ppsc+ppbc+2 ppbc+1]);
    % normal arrow
    if norm(p2-p1)>tipLength
        % vertices of the first stem circle
        for idx = 1:ppsc+1
            v(idx,:) = p1 + stemWidth*(sintheta(idx)*y + costheta(idx)*z);
        end
        % vertices of the second stem circle
        p3 = p2-tipLength*x;
        for idx = 1:ppsc+1
            v(ppsc+1+idx,:) = p3 + stemWidth*(sintheta(idx)*y +
costheta(idx)*z);
        end
        % vertices of the tip circle
        for idx = 1:ppbc+1
            v(2*ppsc+2+idx,:) = p3 + tipWidth*(sinupsilon(idx)*y +
cosupsilon(idx)*z);
        end
        % vertex of the tiptip
        v(2*ppsc+ppbc+4,:) = p2;
        % face of the stem circle
        f(1,1:ppsc+1) = 1:ppsc+1;
        % faces of the stem cylinder
        for idx = 1:ppsc
            f(1+idx,1:4) = [idx idx+1 ppsc+1+idx+1 ppsc+1+idx];
        end
        % face of the tip circle
        f(ppsc+2,:) = 2*ppsc+3:(2*ppsc+3)+ppbc;
        % faces of the tip cone
```

```
        for idx = 1:ppbc
            f(ppsc+2+idx,1:3) = [2*ppsc+2+idx 2*ppsc+2+idx+1 2*ppsc+ppbc+4];
        end
    % only cone v
    else
        tipWidth = 2*sin(tipAngle/2)*norm(p2-p1);
        % vertices of the tip circle
        for idx = 1:ppbc+1
            v(idx,:) = p1 + tipWidth*(sinupsilon(idx)*y + cosupsilon(idx)*z);
        end
        % vertex of the tiptip
        v(ppbc+2,:) = p2;
        % face of the tip circle
        f(1,:) = 1:ppbc+1;
        % faces of the tip cone
        for idx = 1:ppbc
            f(1+idx,1:3) = [idx idx+1 ppbc+2];
        end
    end
    % draw
    fv.faces = f;
    fv.vertices = v;
    h = patch(fv);
    for propno = 1:numel(propertyNames)
        try
            set(h,propertyNames{propno},propertyValues{propno});
        catch
            disp(lasterr)
        end
    end
end
```

Published with MATLAB® R2022a