# Estimation of Drag Coefficient of a Particle Projectile

CHEN,YI-JUI
*Department of AeroSpace Engineering*
*Tamkang University*
New Taipei, Taiwan

***Abstract*—In this study, the finite differential method is used to estimate drag coefficients from trajectory data, which is a crucial method when wind tunnels or other tools for measuring aerodynamic coefficients are unavailable. This technique enables the derivation of approximate coefficients for application in future launches.**

***Index Terms*—Differential method, Drag coefficient, Particle trajectory, Aerodynamic analysis, Particle dynamics**

## I. INTRODUCTION

Accurately forecasting the aerodynamic drag of airborne vehicles is crucial for optimizing their performance and conserving fuel. Traditional wind tunnel testing, although reliable, often proves costly and impractical, especially for smaller organizations seeking alternative methods for coefficient determination. While such alternatives may carry a margin of error, they can yield acceptable results if the error converges within a permissible range. However, data noise can lead to divergence in these methods. To potentially address this, future work could involve implementing a Kalman filter to smooth the data, which would ensure that the noise does not adversely affect the results obtained through the differential method.

## II. THEORETICAL BACKGROUND

### A. Equation of Motion of Particle

In this study, we assume that the particle is two-dimensional (x-z plane), influenced only by gravity and drag, excluding any other external forces like thrust. Consequently, its equation of motion is articulated as follows:

$$\vec{r}(t) = x(t)\vec{i} + z(t)\vec{k} \tag{1}$$

$$\frac{\mathrm{d}\vec{r}}{\mathrm{d}t} = \vec{V}(t) = v_x(t)\vec{i} + v_z(t)\vec{k} \tag{2}$$

$$m\frac{\mathrm{d}\vec{V}}{\mathrm{d}t} = -k|\vec{V}|^2\vec{e}_v + m\vec{g} \tag{3}$$

where $\vec{r}$ denotes the particle's position, x the position on x axis, z the position on z axis, $m$ the particle's mass, $\vec{V}$ the velocity vector, $v_x$ the position on i axis, $v_z$ the position on z axis , $k = \frac{1}{2}\rho C_D S$ the drag parameter, $\rho$ the air density, $C_D$ the drag coefficient, $S$ the reference area, $\vec{e}_v$ the unit vector in the direction of velocity, and $\vec{g}$ the gravitational acceleration vector. This equation models the force equilibrium on the particle, factoring in the drag opposing the motion and gravity pulling towards the Earth's core.

## III. PROCEDURES

The procedures followed in this study encompass data generation, noise simulation, and calculation of the drag coefficient. Each step is designed to replicate real-world conditions within a controlled environment, leveraging mathematical models to estimate aerodynamic parameters.

### A. Data Generation

Data was synthetically generated utilizing an analytical model of 2D particle projectile motion incorporating quadratic drag effects. This method permits controlled manipulation of variables to emulate real-world scenarios by implementing Gaussian-distributed stochastic noise. The employed analytical solutions are detailed in [2].

*1) Analytical Solution with 2-D Quadratic Drag:* The equations underpinning the analytical solution are as follows:

- $k = \frac{1}{2}\rho C_D S$ - Parameter for air resistance.
- $H = \frac{V_0^2 \sin^2\theta_0}{g(2+kV_0^2\sin\theta_0)}$ - maximum height of ascent of the point mass.
- $T = 2\sqrt{\frac{2H}{g}}$ - motion time.
- $V_a = \frac{V_0\cos\theta_0}{\sqrt{1+kV_0^2(\sin\theta_0+\cos^2\theta_0\ln\tan(\frac{\theta_0}{2}+\frac{\pi}{4}))}}$ - the velocity at the trajectory apex.
- $L = V_a T$ - flight range.
- $t_a = \frac{T-kHV_a}{2}$ - the time of ascent.
- $x_a = \sqrt{LH\cot\theta_0}$ - the abscissa of the trajectory apex.
- $a_1 = \frac{L}{x_a}$ - Ratio of flight range and the abscissa of the trajectory apex.
- $c = \frac{2(a_1-1)}{a_1}$ - Constant for trajectory adjustment.
- $w_1 = t - t_a$, $w_2 = \frac{2t(T-t)}{a_1}$ - Temporal factors.
- $x(t) = \frac{L(w_1^2+w_2+w_1\sqrt{w_1^2+cw_2})}{2w_1^2+a_1w_2}$ - Function defining horizontal displacement over time.
- $z(t) = \frac{Ht(T-t)}{t_a^2+(T-2t_a)t}$ - Function for vertical displacement.

For more details, see Appendix A.

With these equations, we can construct trajectories is illustrated in Figs. 1 2. These graphs depict motion paths without the application of stochastic noise, serving as a baseline for further analysis:

```matlab
function [X, Z, cd] = dragkv22D(t,m0)
% Constants and Initial Conditions
rho_a = 1.2; % Air density (kg/m^3)
cd = 0.25; % Drag coefficient
d = 0.13;
r = d / 2; % Radius (m)
m = m0; % Mass (kg)
V0 = 100; % Initial velocity (m/s)
theta0 = deg2rad(70); % Initial angle (rad)  input (deg)
g = 9.81; % Acceleration due to gravity (m/s^2)
A = pi * r^2; % Cross-sectional area (m^2)

% Display constants
disp(['CD : ', num2str(cd)])
disp(['rho : ', num2str(rho_a)])
disp(['d : ', num2str(d)])

% Drag coefficient calculation
k = rho_a * cd * A / (2 * m * g);

% Derived quantities
H = (V0^2 * sin(theta0)^2) / (g * (2 + k * V0^2 * sin(theta0)));
T = 2 * sqrt(2 * H / g);
Va = (V0 * cos(theta0)) / sqrt(1 + k * V0^2 * (sin(theta0) + cos(theta0)^2 * log(tan(theta0 / 2 + pi / 4))));
L = Va * T;
ta = (T - k * H * Va) / 2;
xa = sqrt(L * H * cot(theta0));
a1 = L / xa;

% Corrected definitions of w1, w2, and c
w1 = t - ta;
w2 = 2 .* t .* (T - t) / a1;
c = 2 * (a1 - 1) / a1;

% Calculate X and Z
X = (L .* (w1.^2 + w2 + w1 .* sqrt(w1.^2 + c .* w2))) ./ (2 .* w1.^2 + a1 .* w2);
Z = H .* t .* (T - t) ./ (ta^2 + (T - 2 * ta) .* t);
end
```

Fig. 1. Theoretical trajectory without noise.

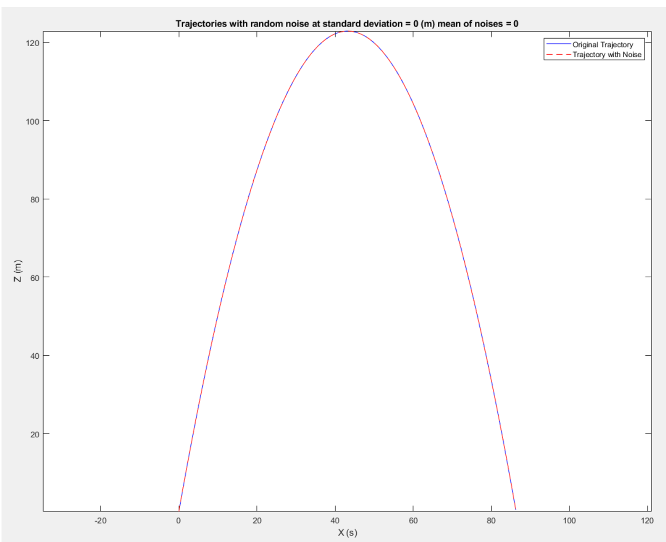


Fig. 2. 2D representation of the theoretical trajectory without noise.

We now proceed to introduce noise to the model to better simulate the variable conditions encountered in real-world data collection.

### B. Generating Gaussian Distribution Noise

We input the pseudo-real data into the functions `calVaXYZ()` 10 and `calVnorm()` 11 to calculate the velocity, velocity norm, and acceleration needed for Equation (4). These values are then used in Equation (4) to find $C_D$. The brief implementation code is illustrated in Fig. 3. The detailed derivation of $C_D$ is discussed in Appendix B, Derivation of Drag Coefficient.

```matlab
clc
clear
close all

m0 = 100;% kg
T = 0:0.01:1000;
%% Trajectory generation (1-D)
%[Z_origin,CD_true] = dragS(T,m0);
%index = Z_origin(:)>=0;
%X = zeros(length(T(index)),1);
%Y = zeros(length(T(index)),1);
%T = T(index);
%Z_origin = Z_origin(index);

%$ Trajectory generation (2-D)
[X_origin,Z_origin,CD_true] = dragkv22D(T,m0);
index = Z_origin(:)>=0;
Y = zeros(1,length(T(index)));
T = T(index);
Z_origin = Z_origin(index);
X_origin = X_origin(index);
%% add noise for standard deviation 0.3 meter (mean of noise is 0)
sigma = 0.3; % standard deviation %
mu = 0;

noise_2X = sigma.*generateNormalRandomNumbers(size(X_origin), mu, sigma); % original maybe < 0 so let T = 0 Z = 0
noise2X(1) = 0;
noise_2Z = sigma.*generateNormalRandomNumbers(size(Z_origin), mu, sigma); % original maybe < 0 so let T = 0 Z = 0
noise2Z(1) = 0;
X = X_origin + noise_2X;
Z = Z_origin + noise_2Z;
```

Fig. 3. Adding noise to the analytical solution.

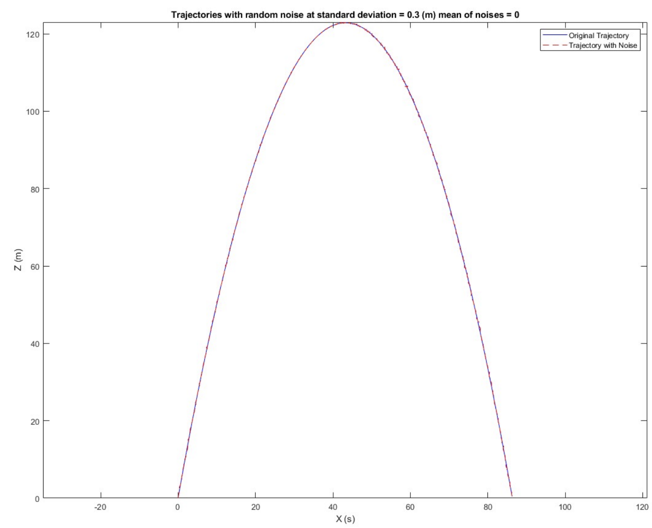


Fig. 4. Trajectory with noise.

### C. Calculation of Drag Coefficient ($C_D$)

The derivation of the equations for $C_D$ is included at the end of this paper. We first apply these equations to determine the current drag coefficient. The formula used is as follows:

$$C_D = \frac{2m(\vec{g} - \vec{a}) \cdot \vec{V}}{\rho \left\| \vec{V} \right\|^3 A} \tag{4}$$

We input the pseudo-real data into the function `calVaXYZ()` to calculate the velocity and acceleration needed for Equation (4). These values are then used in Equation (4) to find $C_D$. The detailed derivation of $C_D$ and the code for `calVaXYZ()` are discussed in Appendix B.

### D. Results Analysis

```
%% calculate velocity and acceleration or use(diff())

S = [X;Y;Z];

[V,a] = calVaXZ(S',T);

Vx =V(:,1);
Vy =V(:,2);
Vz =V(:,3);

ax =a(:,1);
ay =a(:,2);
az =a(:,3);

% Environement Condition
W = [0,0,0];
g = [0,0,-9.81];
rho = 1.225;
d = 0.13;
V_norm = calVnorm(V,T);

% Calculate CD
A = (a - g);
B = V(1:end-1,:) - W;

CD = (-8 .* m0 .* dot(A,B,2)) ./ (pi .* d^2 .* rho * V_norm(1:end-1,:).^3);

% Calculate the squared differences between the calculated CD and the used CD
squared_diff = (CD - CD_true).^2;

% Calculate the mean squared error (RMSE)
MSE = mean(squared_diff);
RMSE = sqrt(MSE);
```

Fig. 5. code of calculate CD.

we first use the data without noise is illustrated in Figs.6.

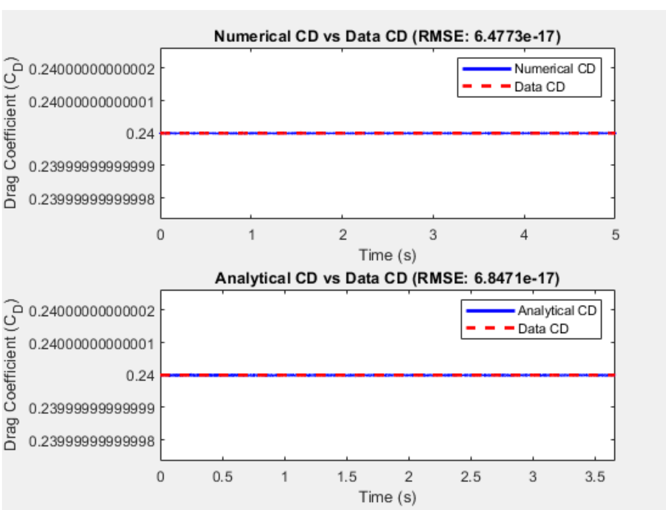


Fig. 6. Comparison of actual and estimated drag coefficients: Numerical CD vs. Data CD and Analytical CD vs. Data CD without noise.

Initially, we analyzed the data in the absence of noise. Although the results appeared smoother, they did not fully align with the actual $C_D$ , suggesting discrepancies in the estimation process or model assumptions.

Subsequently, we incorporated noise into the trajectory data:

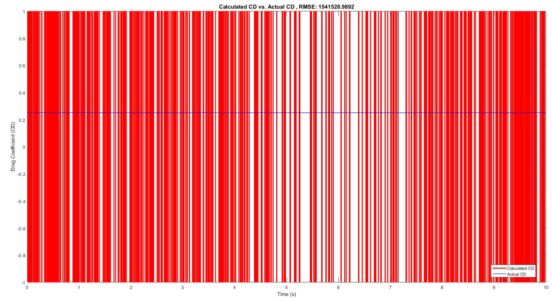


Fig. 7. Comparison of actual and estimated drag coefficients with noise.

We found that when the trajectory data is added with noise, the RMSE (root-mean-square error) rises rapidly. This observation is illustrated in Figs. 7.

## IV. Future Work

Intention to find why the result didn't fully consistent with the Actul $C_D$.
and Implement a Kalman filter to predict, estimate, and smooth the position data, enabling the use of the differential method to obtain an appropriate drag coefficient.

## References


[1] M. Doso et al., "Assessment of Drag Prediction Techniques for a Flying Vehicle Based on Radar-Tracked Data," *International Journal of Aeronautical and Space Sciences*, [Online]. Available: https://doi.org/10.1007/s42405-023-00656-7

[2] P. S. Chudinov, "Approximate Analytical Investigation of Projectile Motion in a Medium with Quadratic Drag Force," *International Journal of Sports Science and Engineering*, vol. 5, no. 1, pp. 027–042, 2011.


## V. Appendix

### A. Analytical Solution with 2-D Quadratic Drag Detail

- $k$: A coefficient that incorporates air density $\rho$, drag coefficient $C_D$, and the cross-sectional area $S$ of the object. This coefficient is crucial in calculating the effects of air resistance on the projectile.

$$k = \frac{1}{2}\rho C_D S$$

It reflects how air density, shape, and size of the object interact to affect its motion through air.

- $H$: maximum height of ascent of the point mass. It is calculated based on the initial velocity $V_0$, launch angle $\theta_0$, gravitational acceleration $g$, and the coefficient $k$.

$$H = \frac{V_0^2 \sin^2\theta_0}{g(2 + kV_0^2 \sin\theta_0)}$$

- $T$: motion time. It is calculated based on the peak altitude $H$ and gravitational acceleration $g$.

$$T = 2\sqrt{\frac{2H}{g}}$$

- $V_a$: the velocity at the trajectory apex. It is calculated based on the initial velocity $V_0$, launch angle $\theta_0$, and the coefficient $k$.

$$V_a = \frac{V_0 \cos\theta_0}{\sqrt{1 + kV_0^2(\sin\theta_0 + \cos^2\theta_0 \ln\tan(\frac{\theta_0}{2} + \frac{\pi}{4}))}}$$

- $L$: flight range. It is calculated based on the velocity at a specified instance $V_a$ and total flight time $T$.

$$L = V_a T$$

- $t_a$: the time of ascent. It is calculated based on the total flight time $T$, maximum height of ascent of the point mass $H$, and velocity at a specified instance $V_a$.

$$t_a = \frac{T - kHV_a}{2}$$

- $x_a$: the abscissa of the trajectory apex. It is calculated based on the total horizontal traversal $L$, peak altitude $H$, and launch angle $\theta_0$.

$$x_a = \sqrt{LH\cot\theta_0}$$

- $a_1$: Ratio of position. It is calculated based on the total horizontal traversal $L$ and specific horizontal coordinate $x_a$.

$$a_1 = \frac{L}{x_a}$$

- $c$: Constant for trajectory adjustment. It is calculated based on the ratio of position $a_1$.

$$c = \frac{2 \cdot (a_1 - 1)}{a_1}$$

- $w_1$: Temporal factors. It represents the time difference from the time of ascent $t_a$ to any given time $t$.

$$w_1 = t - t_a$$

- $w_2$Temporal factors, potentially influenced by the launch angle $a_1$ and motion time $T$

$$w_2 = \frac{2t(T - t)}{a_1}$$

- $x(t)$: Function defining horizontal displacement over time. It calculates the horizontal position at any given time $t$.

$$x(t) = \frac{L(w_1^2 + w_2 + w_1\sqrt{w_1^2 + cw_2})}{2w_1^2 + a_1 w_2}$$

- $z(t)$: Function for vertical displacement. It calculates the vertical position at any given time $t$.

$$z(t) = \frac{Ht(T - t)}{t_a^2 + (T - 2t_a)t}$$

### B. The complement of Derivation of Drag Coefficient

The drag force equation is given by:

$$D = \frac{1}{2}\rho \left\|\vec{V}\right\|^2 C_D S$$

We assume that the external forces include only gravity and drag force:

$$\vec{F}_{\text{ex}} = \vec{F}_g + \vec{D}$$

According to Newton's second law, the external force is equal to the mass times acceleration:

$$\vec{F}_{\text{ex}} = m\vec{a}$$

Therefore:

$$\vec{F}_g + \vec{D} = m\vec{a}$$

Considering the gravitational force $\vec{F}_g = m\vec{g}$, we can rearrange the terms to get:

$$\vec{D} = D(-\vec{e}_v) = m(\vec{a} - \vec{g})$$

Taking the dot product with $-\vec{e}_v$ on both sides:

$$D = m(\vec{a} - \vec{g}) \cdot (-\vec{e}_v) = -m(\vec{a} - \vec{g}) \cdot \frac{\vec{V}}{\left\|\vec{V}\right\|}$$

Substituting the drag force equation and solving for the drag coefficient $C_D$:

$$C_D = \frac{-2m(\vec{a} - \vec{g}) \cdot \vec{V}}{\rho \left\|\vec{V}\right\|^3 S} \tag{5}$$

The velocity vector and its magnitude are given by:

$$\vec{V} = v_x\vec{i} + v_z\vec{k}, \quad \left\|\vec{V}\right\| = \sqrt{\vec{V} \cdot \vec{V}}, \quad \vec{e}_v = \frac{\vec{V}}{\left\|\vec{V}\right\|}$$

Once the velocity and acceleration vectors are obtained, the drag coefficient $C_D$ can be estimated using the above equation, where $m$ is the mass of the vehicle, $\vec{g}$ is the gravitational acceleration vector, $\vec{V}$ is the velocity vector, $\rho$ is the air density, and $S$ is the reference area for the drag coefficient.

*1) Matlab code:* Use Position data to computer velocity and acceleration

```
%% calculate velocity and acceleration or use(diff())

S = [X;Y;Z];

[V,a] = calVaXZ(S',T);

Vx =V(:,1);
Vy =V(:,2);
Vz =V(:,3);

ax =a(:,1);
ay =a(:,2);
az =a(:,3);

% Environement Condition
W = [0,0,0];
g = [0,0,-9.81];
rho = 1.225;
d = 0.13;
V_norm = calVnorm(V,T);

% Calculate CD
A = (a - g);
B = V(1:end-1,:) - W;

CD = (-8 .* m0 .* dot(A,B,2)) ./ (pi .* d^2 .* rho * V_norm(1:end-1,:).^3);

% Calculate the squared differences between the calculated CD and the used CD
squared_diff = (CD - CD_true).^2;

% Calculate the mean squared error (RMSE)
MSE = mean(squared_diff);
RMSE = sqrt(MSE);
```

Fig. 8. Calculate $C_D$

```
%% plot result
figure
plot(T(1:end-2),CD,'r',LineWidth=2)
hold on
plot(T, CD_true * ones(size(T)), 'b', 'LineWidth', 1);

% Adding labels and legend
xlabel('Time (s)');
ylabel('Drag Coefficient (CD)');
xlim([T(1),T(end)])
ylim([-1,1])
legend('Calculated CD', 'Actual CD', 'Location', 'best');
% Displaying the MSE on the plot
title(['Calculated CD vs. Actual CD , RMSE: ', num2str(RMSE)], 'FontSize', 12);
% Displaying the plot
hold off;

% trajectory compare (1-D)
%figure
%plot(T, Z_origin, 'b-', T, Z, 'r--')
%legend('Original Trajectory', 'Trajectory with Noise')
%xlabel('Time (s)')
%ylabel('Position (m)')
%title(['Trajectories with random noise at standard deviation = ',num2str(sigma),' (m) mean of noises = ',num2str(mu)])

% trajectory compare (2-D)
figure
plot(X_origin, Z_origin, 'b-', X, Z, 'r--')
legend('Original Trajectory', 'Trajectory with Noise')
xlabel('X (s)')
ylabel('Z (m)')
title(['Trajectories with random noise at standard deviation = ',num2str(sigma),' (m) mean of noises = ',num2str(mu)])
axis equal
```

Fig. 9. Plot $C_D$ v.s. $t$ and Trajectory (x-z plane)

```
function [V,a] = calVaXYZ(S,T)

    X = S(:,1);
    Y = S(:,2);
    Z = S(:,3);

    % Calculate velocity
    V = zeros(length(T)-1,3);
    for i = 1:length(T)-1
        V(i,1) = (X(i+1) - X(i)) / (T(i+1) - T(i)); %Vx
        V(i,2) = (Y(i+1) - Y(i)) / (T(i+1) - T(i)); %Vy
        V(i,3) = (Z(i+1) - Z(i)) / (T(i+1) - T(i)); %Vz
    end

    % Calculate acceleration
    a = zeros(length(T)-2,3);
    for i = 1:length(T)-2
        a(i,1) = (V(i+1,1) - V(i,1)) / (T(i+1) - T(i)); %ax
        a(i,2) = (V(i+1,2) - V(i,2)) / (T(i+1) - T(i)); %ay
        a(i,3) = (V(i+1,3) - V(i,3)) / (T(i+1) - T(i)); %az
    end
end
```

Fig. 10. function of calVaXYZ

```
function V_norm = calVnorm(V,T)
    V_norm = zeros(length(T)-1,1);
        for i = 1:length(T)-1
            V_norm(i) = norm(V(i,:));
        end
end
```

Fig. 11. function of calVnorm

### *C. The complement of Gaussian random number generator*

*1) Application of Linear Congruential Generator and Box-Muller Transform in Generating Random Numbers*
*:* First, we implement a linear congruential generator (LCG) from scratch to create uniformly distributed random numbers. This method involves calculating each random number using the formula $X = \text{mod}(a \times X + c, m)$. Here, I select one seed to keep my simulation easy to test. and use U to let the data involove in [0,1)

```
function Z = generateNormalRandomNumbers(dimensions, mu, sigma) % only 2-D matrix
seed = 1234;        % Initial seed X(1)
a = 1664525;        % Multiplier
c = 1013904223;     % Increment
m = 2^32;           % Modulus
A = dimensions(1);
B = dimensions(2);
n = 2*A*B;              % Number of uniformly distributed random numbers to generate
% Generate uniformly distributed random numbers
U = zeros(n, 1);
X = seed; % Initial X
for i = 1:n
    X = mod(a * X + c, m);
    U(i) = X / m;  % Scale random numbers to [0, 1) interval
end
```

Fig. 12. Implementation of the Linear congruential generator.

Second, we use the Box-Muller transform to transfer uniform distribution to normal distribution.

```
% Box-Muller transform
Z = zeros(A, B);
index = 1;
for i = 1:A
    for j = 1:B
        Z(i,j) = mu + sigma * sqrt(-2 * log(U(index))) * cos(2 * pi * U(index+1));
        %Z(i,j) = mu + sigma * sqrt(-2 * log(U(index))) * sin(2 * pi * U(index+1));
        index = index + 2;
    end
end

end
```

Fig. 13. Application of the Box-Muller transform to generate normally distributed random numbers.

Then, I implement the code and add a density line to check if my code is correct.

```
clc
clear
close all
%%
mu = 0;
sigma = 1;
Z = generateNormalRandomNumbers([10000,2],mu,sigma);
x = -4*sigma:0.01:4*sigma;
a = -((x-mu).^2/(2*sigma^2));
f = (1/(sigma*sqrt(2*pi)))*exp(a);

figure
% Plot histogram of normal distribution random numbers with 50 bins
histogram(Z, 50, 'FaceColor', 'blue','Normalization', 'pdf');
hold on
plot(x,f,'Color','red','LineWidth',2)
hold off
title('Histogram and PDF of Normal Distribution Random Numbers');
xlabel('Value');
ylabel('Probability Density');
```

Fig. 14. imply randn code.

The results are presented here.

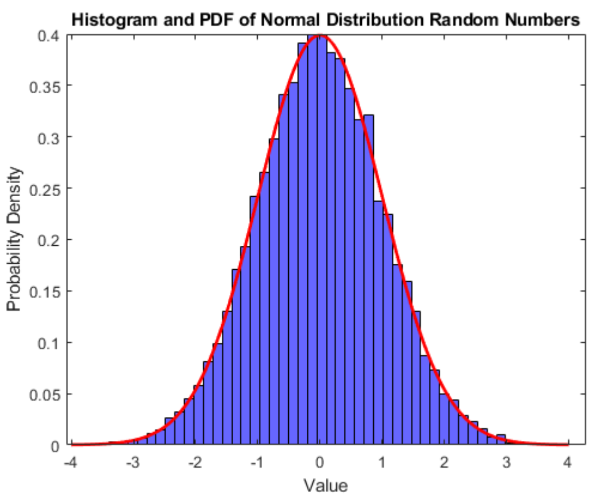


Fig. 15. Histogram and PDF of Normal Distribution Random Numbers.

## VI. Other Matlab Code I Write

### A. Integrator

Because we wanted to understand how numerical integrators work, we built several integrators to gain a deeper understanding of their mechanisms. Furthermore, we derive or refer to analytical solutions and dynamic models that allow us to test the reliability of the integrator. Below is a description of our integrators.

*1) Runge-Kutta:* The Runge-Kutta methods are a series of iterative techniques, which significantly improve the accuracy of approximations to differential equations. They use multiple intermediate steps (slopes) to achieve a better estimate of the derivative at each interval.

**Definition**

Consider a differential equation expressed as $\frac{dy}{dt} = f(t, y)$, where $f$ is a known function. The 4th order Runge-Kutta method, one of the most common, calculates the next value $y_{k+1}$ by using the current value $y_k$ and the weighted average of four increments, where each increment is a product of the size of the interval, $h$, and an estimated slope:

$$
\begin{aligned}
k_1 &= hf(t_k, y_k), \\
k_2 &= hf\left(t_k + \frac{1}{2}h, y_k + \frac{1}{2}k_1\right), \\
k_3 &= hf\left(t_k + \frac{1}{2}h, y_k + \frac{1}{2}k_2\right), \\
k_4 &= hf(t_k + h, y_k + k_3),
\end{aligned}
$$

Then, the update formula for $y_{k+1}$ is given by:

$$y_{k+1} = y_k + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4).$$

This formulation ensures that each step combines information from several estimates of the derivative, providing a high degree of accuracy with reasonable computational effort.

```
function [t,y] = RungeKutta( y0 , h , tf ,U)
% RungeKutta method for numerical integration with less accuracy.
% Inputs:
%   y0: Initial state vector.
%   h: Time step size.
%   tf: Final time.
%   U: Control parameter vector.
% Outputs:
%   t: Time vector.
%   y: State matrix.
N = tf / h;
y = zeros(6, N+1);% N+1 is iteration time plus initial time
t = 0:h:tf; % initial condition already known so we only have to iterate N time
y(:,1) = y0;% place initial condition in list
```

Fig. 16. code for RungeKutta 1

```

for n = 1:N %%
    U(3) = t(n); % current time
    f = @(y) particle_model3DRungeKutta(y, U);
    k1 = f(y(:,n));

    U(3) = t(n) + h/2; % update t(n) + h/2
    k2 = f(y(:,n) + h/2 * k1);

    U(3) = t(n) + h/2; % hold t(n) + h/2
    k3 = f(y(:,n) + h/2 * k2);

    U(3) = t(n) + h; % update t(n) + h
    k4 = f(y(:,n) + h * k3);

    y(:,n+1) = y(:,n) + h/6*(k1+2*k2+2*k3+k4);

    if y(6, n+1) <= 0
        y = y(:, 1:n+1);
        t = t(1:n+1);
        break;
    end
end
end
```

Fig. 17. code for RungeKutta 2

```
function XDOT = particle_model3DRungeKutta(y,U)
%% State space
Vx = y(1); %Vx
Vy = y(2); %Vy
Vz = y(3); %Vz
X = y(4); %X
Y = y(5); %Y
Z = y(6); %Z
v =[Vx;Vy;Vz];
s =[X;Y;Z];
c = U(1);          %mdot
Vex = U(2);        %Vex
t = U(3);          %time
tbo = U(4);        %s
m0 = U(5);         %kg
rho = U(6);        %kg/m^3
Cd = U(7);
g = U(8);          %m/s^2
S = U(9);          %m^2
thetar = U(10);     %theta degree
%%
if t == 0
    gamma = thetar;
else
    gamma = atan2(Vz,sqrt(Vx^2+Vy^2));
end
Cgamma = [cos(-gamma)  0         -sin(-gamma);
          0         1         0       ;
          sin(-gamma)  0         cos(-gamma)];

%% m(t)  mf = m0 - integral(u1,0,t)
mf = 0;
if t <= tbo
    mf = m0 -c.*t;
elseif t > tbo
    mf = m0 -c.*tbo;
end
```

Fig. 18. Dynamic model of particle assumption 1

```
%% FA    FA = 0.5*rho*V^2*Cd*S      opposite to velocity direction
k= 0.5*rho*Cd*S;
FAV = -k*norm([Vx;Vy;Vz]).*[Vx;Vy;Vz];

%% FT    FT = mdot*Vex     = g0*Isp*mdot     Isp (s)
if t <= tbo
    FT = c*Vex;
elseif t > tbo
    FT = 0;
end

FTV = FT*Cgamma.'*[1;0;0];

%% FG
FG=mf*g;
if t == 0
    FGV = [0;0;0];
elseif t > 0
    FGV = FG*[0;0;-1];
end

FB=FTV+FGV+FAV;

%% a = F/m(t)

a = 1/mf.*FB;

%% XDOT
%disp('Size of a:');
%disp(size(a));
%disp('Size of v:');
%disp(size(v));
XDOT = [a;v];

end
```

Fig. 19. Dynamic model of particle assumption 2

```
clc;
clear;
close all;

% Parameters
tbo = 7;              % s
m0 = 42;              % kg
mfuel=30.149;         % kg
rho = 1.225;          % kg/m^3
Cd = 0.24;
g = 9.81;             % m/s^2
S = 0.065^2*pi;       % m^2
theta = deg2rad(80);  % rad input(deg) launch angle
c = 0;%4.307;              % mdot
Vex = 502;            % Vex
t0 = 0;             % starttime
V0 = 100;

% Initial conditions
v0 = Ctheta(theta)*[V0;0;0];
Vx0 =v0(1);
Vy0 =v0(2);
Vz0 =v0(3);
s0 = [0;0;0];
X0 =s0(1);
Y0 =s0(2);
Z0 =s0(3);

% combime Initial conditions and integral parameters
y0 = [Vx0; Vy0; Vz0; X0; Y0; Z0]; % IC
h = 1e-4;              % timestep
tf = 1000;               % final time

U = [c; Vex; t0; tbo; m0; rho; Cd; g; S; theta];
% solve by Runge-Kutta
[t, y] = RungeKutta(y0, h, tf, U);
```

Fig. 20. Imply RungeKutta and Dynamic model 1

```
%% plot
Vx = y(1,:);
Vy = y(2,:);
Vz = y(3,:);
X = y(4,:);
Y = y(5,:);
Z = y(6,:);
%% analytical solution
Zm = masschangingaS(t,U,V0,s0(3),mfuel);
ZD = dragaS(t,U,V0,s0(3));

[X2D, Z2D, cd] = dragkv22D(t,V0,U);

Sa = analysissolution(t,Vx0,Vz0,X0,Z0,g);
Xa = Sa(1,:);
Za = Sa(2,:);
%% RMS
% masschangeRMS
squared_errorm = (Zm - Z).^2  ;
mean_squared_errorm = mean(squared_errorm);
RMSm = sqrt(mean_squared_errorm);

% dragkV2RMS
squared_errorD = (ZD - Z).^2  ;
mean_squared_errorD = mean(squared_errorD);
RMSD = sqrt(mean_squared_errorD);

% dragkV22DRMS
squared_errorD2D = (X2D - X).^2+(Z2D - Z).^2  ;
mean_squared_errorD2D = mean(squared_errorD2D);
RMSD2D = sqrt(mean_squared_errorD2D);

% analyticalsolutionRMS
squared_errorA = (Xa - X).^2+(Za - Z).^2  ;
mean_squared_errorA = mean(squared_errorA);
RMSA = sqrt(mean_squared_errorA);
```

Fig. 21. Imply RungeKutta and Dynamic model 2

```
%
if Cd == 0 && c ~= 0 && theta == 90
    figure
    plot(t,Z,'r' ,'LineWidth', 2)
    hold on
    plot(t,Zm,'b','LineWidth', 2)
    xlabel('t s')
    ylabel('Z m')
    grid on
    title(['Z-t,\theta = ', num2str(theta),' c = ', num2str(c),' Vex = ', num2str(Vex),' Vz0 = ' ...
        , num2str(Vz0),' Z0 = ', num2str(Z0),' RMS = ', num2str(RMSm),' masschanging 1-D, sample time = ', num2str(h)])
    legend('Numerical Solution','Analytical Solution')
elseif Cd ~= 0 && c == 0 && theta == 90
    figure
    plot(t,Z,'r','LineWidth', 2)
    hold on
    plot(t,ZD,'b','LineWidth', 2)
    xlabel('t s')
    ylabel('Z m')
    grid on
    title(['X-t,\theta = ', num2str(theta),' \rho = ', num2str(rho),' Cd = ', num2str(Cd),' S = ' ...
        , num2str(S),' Vz0 = ', num2str(Vz0),' Z0 = ', num2str(Z0),' RMS = ', num2str(RMSD),' drag(kV^2), sample time = ', num2str(h)])
    legend('Numerical Solution','Analytical Solution')
elseif Cd ~= 0 && c == 0 && theta ~= 90
    figure
    plot(X,Z,'r','LineWidth', 2)
    hold on
    plot(X2D,Z2D,'b','LineWidth', 2)
    xlabel('X m')
    ylabel('Z m')
    grid on
    title(['X-t,\theta = ', num2str(theta),' \rho = ', num2str(rho),' Cd = ', num2str(Cd),' S = ' ...
        , num2str(S),' Vz0 = ', num2str(Vz0),' Z0 = ', num2str(Z0),' RMS = ', num2str(RMSD2D),' drag(kV^2), sample time = ', num2str(h)])
    legend('Numerical Solution','Analytical Solution')
```

Fig. 22. Imply RungeKutta and Dynamic model 3

```
elseif c == 0 && Cd == 0
    if theta == 90
        figure
        plot(t,Z,'r','LineWidth', 2)
        hold on
        plot(t,Za,'b','LineWidth', 2)
        xlabel('t s')
        ylabel('Z m')
        grid on
        title(['X-Z,\theta = ', num2str(theta),' RMS = ', num2str(RMSA),' sample time = ', num2str(h)])
        legend('Numerical Solution','Analytical Solution')
    else
        figure
        plot(X,Z,'r','LineWidth', 2)
        hold on
        plot(Xa,Za,'b','LineWidth', 2)
        xlabel('X m')
        ylabel('Z m')
        grid on
        title(['X-Z,\theta = ', num2str(theta),' RMS = ', num2str(RMSA),' sample time = ', num2str(h)])
        legend('Numerical Solution','Analytical Solution')
    end

else
    figure
    plot(X,Z,'r','LineWidth', 2)
    xlabel('X m')
    ylabel('Z m')
    title(['X-Z,\theta = ', num2str(theta),' \rho = ', num2str(rho),' Cd = ', num2str(Cd),' S = ' ...
        , num2str(S),' c = ', num2str(c),' Vex = ', num2str(Vex),' Vx0 = ', num2str(Vx0),' Vy0 = ' ...
        , num2str(Vy0),' Vz0 = ', num2str(Vz0), ' Z0 = ', num2str(Z0),' sample time = ', num2str(h)])
    grid on
    axis equal
end
```

Fig. 23. Imply RungeKutta and Dynamic model 4

*2) Dormand-Prince Method:* We employ the 5th-order solution to test the convergence of the 4th-order solution. If the error between these two solutions exceeds a predefined tolerance, we calculate an optimal timestep to re-simulate the time point where convergence was not achieved.

**Definition**

The single-step calculation in the Dormand-Prince method is performed as follows:

$$
\begin{aligned}
k_1 &= hf(t_k, y_k), \\
k_2 &= hf\left(t_k + \frac{1}{5}h, y_k + \frac{1}{5}k_1\right), \\
k_3 &= hf\left(t_k + \frac{3}{10}h, y_k + \frac{3}{40}k_1 + \frac{9}{40}k_2\right), \\
k_4 &= hf\left(t_k + \frac{4}{5}h, y_k + \frac{44}{45}k_1 - \frac{56}{15}k_2 + \frac{32}{9}k_3\right), \\
k_5 &= hf\left(t_k + \frac{8}{9}h, y_k + \frac{19372}{6561}k_1 - \frac{25360}{2187}k_2\right. \\
&\qquad \left. + \frac{64448}{6561}k_3 - \frac{212}{729}k_4\right), \\
k_6 &= hf\left(t_k + h, y_k + \frac{9017}{3168}k_1 - \frac{355}{33}k_2\right. \\
&\qquad \left. - \frac{46732}{5247}k_3 + \frac{49}{176}k_4 - \frac{5103}{18656}k_5\right), \\
k_7 &= hf\left(t_k + h, y_k + \frac{35}{384}k_1 + \frac{500}{1113}k_3 + \frac{125}{192}k_4\right. \\
&\qquad \left. - \frac{2187}{6784}k_5 + \frac{11}{84}k_6\right),
\end{aligned}
$$

The next step value $y_{k+1}$ is then calculated as:

$$
\begin{aligned}
y_{k+1} = y_k &+ \frac{35}{384}k_1 + \frac{500}{1113}k_3 + \frac{125}{192}k_4 \\
&- \frac{2187}{6784}k_5 + \frac{11}{84}k_6.
\end{aligned}
$$

Next, we calculate the 5th-order step value $z_{k+1}$ as:

$$
\begin{aligned}
z_{k+1} = y_k &+ \frac{5179}{57600}k_1 + \frac{7571}{16695}k_3 + \frac{393}{640}k_4 \\
&- \frac{92097}{339200}k_5 + \frac{187}{2100}k_6 + \frac{1}{40}k_7.
\end{aligned}
$$

We then determine the difference between the two step values $|z_{k+1} - y_{k+1}|$:

$$
\begin{aligned}
|z_{k+1} - y_{k+1}| = \bigg| &\frac{5179}{57600}k_1 - \frac{7571}{16695}k_3 + \frac{393}{640}k_4 \\
&- \frac{92097}{339200}k_5 + \frac{187}{2100}k_6 + \frac{1}{40}k_7 \bigg|.
\end{aligned}
$$

This difference is considered as the error in $y_{k+1}$. We calculate the optimal time interval $h_{opt}$ as follows:

$$
s = \left(\frac{1}{2}\frac{\varepsilon h}{|z_{k+1} - y_{k+1}|}\right)^{\frac{1}{5}}, \quad h_{opt} = sh,
$$

where $\varepsilon$ is the desired accuracy level.

```
function [t,y] = Dormand_Prince(tf,y0,h,maxh,minh,tolerance,U)
t = 0;
y = zeros(6,1);
z = zeros(6,1);
n = 0;  % Initialize counter for actual iterations
y(:,1) = y0;
z(:,1) = y0;
f = @particle_model3DDormandPrince;



maxIterations = 1e10;

for iter = 1:maxIterations
    if t(end) + h > tf
        h = tf - t(end);  % ensure the time does not exceed tf
    end

    % Dormand-Prince calculations
    k1 = h * f(t(end), y(:,end),U);
    k2 = h * f(t(end) + 1/5*h, y(:,end) + 1/5*k1,U);
    k3 = h * f(t(end) + 3/10*h, y(:,end) + 3/40*k1 + 9/40*k2,U);
    k4 = h * f(t(end) + 4/5*h, y(:,end) + 44/45*k1 - 56/15*k2 + 32/9*k3,U);
    k5 = h * f(t(end) + 8/9*h, y(:,end) + 19372/6561*k1 - 25360/2187*k2 + 64448/6561*k3 - 212/729*k4,U);
    k6 = h * f(t(end) + h, y(:,end) + 9017/3168*k1 - 355/33*k2 + 46732/5247*k3 + 49/176*k4 - 5103/18656*k5,U);
    k7 = h * f(t(end) + h, y(:,end) + 35/384*k1 + 500/1113*k3 + 125/192*k4 - 2187/6784*k5 + 11/84*k6,U);


    new_y = y(:,end) + 35/384 * k1 + 500/1113 * k3 + 125/192 * k4 -2187/6784 * k5 + 11/84 * k6;
    new_z = y(:,end) + 5179/57600 * k1 +7571/16695 * k3 +393/640 * k4 -92097/339200 * k5 + 187/2100 * k6 +1/40 * k7;
```

Fig. 24. code for Dormand-Prince 1

```
    e = norm(new_z - new_y);

    if e <= tolerance
        n = n + 1;
        t = [t, t(end) + h];
        y = [y, new_y];
        z = [z, new_z];

    end

    s = (tolerance * h/(2*e))^0.2;  % calculate the scaling factor

    hopt = s * h;  % calculate the optimal time step

    if hopt > maxh
        h = maxh;
    elseif hopt < minh
        h = minh;
    else
        h = hopt;
    end

    if y(6,end) < 0 || t(end) >= tf  % || means 'or'
        break;  % Break if  landed or the final time is reached
    end
end

end
```

Fig. 25. code for Dormand-Prince 2

```
function XDOT = rocketDynamics(t,y,U)
%% State space

Vx = y(1); %Vx
Vy = y(2); %Vy
Vz = y(3); %Vz
X = y(4); %X
Y = y(5); %Y
Z = y(6); %Z
v =[Vx;Vy;Vz];
s =[X;Y;Z];
c = U(1);           %mdot
Vex = U(2);         %Vex
tbo = U(3);         %s
m0 = U(4);          %kg
rho = U(5);         %kg/m^3
Cd = U(6);
g = U(7);           %m/s^2
S = U(8);           %m^2
thetar = U(9);      %theta radian
%%
if t == 0
    gamma = thetar;
else
    gamma = atan2(Vz,sqrt(Vx^2+Vy^2));
end
if thetar == 90*pi/180
    Cgamma = [0             0       -sin(-gamma);
              0             1       0           ;
              sin(-gamma)   0       0];
else
    Cgamma = [cos(-gamma)  0          -sin(-gamma);
          0           1          0          ;
          sin(-gamma)  0         cos(-gamma)];
end
```

Fig. 26. Dynamic model of particle assumption 1

```
%% m(t)  mf = m0 - integral(u1,0,t)
mf = 0;
if t <= tbo
    mf = m0 -c.*t;
elseif t > tbo
    mf = m0 -c.*tbo;
end
%% FA    FA = 0.5*rho*V^2*Cd*S       opposite to velocity direction
k= 0.5*rho*Cd*S;
FAV = -k*norm([Vx;Vy;Vz]).*[Vx;Vy;Vz];

%% FT    FT = mdot*Vex     = g0*Isp*mdot    Isp (s)
if t <= tbo
    FT = c*Vex;
elseif t > tbo
    FT = 0;
end

FTV = FT*Cgamma.'*[1;0;0];

%% FG
FG=mf*g;
if t == 0
    FGV = [0;0;0];
elseif t > 0
    FGV = FG*[0;0;-1];
end

FB=FTV+FGV+FAV;

%% a = F/m(t)
a = 1/mf.*FB;

%% XDOT
XDOT = [a;v];

end
```

Fig. 27. Dynamic model of particle assumption 2

```
clc
%clear
close all

% Parameters
tbo = 7;            % s
m0 = 42;            % kg
mfuel=30.149;       % kg
rho = 1.225;        % kg/m^3
Cd = 0.24;
g = 9.81;           % m/s^2
S = 0.065^2*pi;     % m^2
theta = deg2rad(80);  % launch angle rad input(deg)
c = 0;%4.307;            % mdot
Vex = 502;          % Vex
t0 = 0;
% initial state
V0 = 100;
v0 = Ctheta(theta)*[V0;0;0];
Vx0 =v0(1);
Vy0 =v0(2);
Vz0 =v0(3);
s0 = [0;0;0];
X0 =s0(1);
Y0 =s0(2);
Z0 =s0(3);
% initial value
tf = 1000;              % final time
y0 = [Vx0; Vy0; Vz0; X0; Y0; Z0]; % initial position vilocity
h = 1e-4;             % initail step
maxh = 1;
minh = 1e-5;
tolerance = 1e-4;

%%%%%%%%%%%%%%%%--------%%%%%%%%%%%%%%%%--------%%%%%%%%%%%%%%%%--------%%%%%%%%%%%%%%%%
```

Fig. 28. Imply Dormand-Prince and Dynamic model 1

```matlab
U = [c; Vex; tbo; m0; rho; Cd; g; S; theta];
% solve by Dormand_Prince
%[t, y] = Dormand_Prince(tf,y0,h,maxh,minh,tolerance,U);
[t, y] = DormandPrincetest(tf,y0,h,maxh,minh,tolerance,U);

%% plot
thetaplot = theta/(pi*180);
Vx = y(1,:);
Vy = y(2,:);
Vz = y(3,:);
X = y(4,:);
Y = y(5,:);
Z = y(6,:);
%% analytical solution
Zm = masschangingaS(t,U,V0,s0(3),mfuel);
ZD = dragaS(t,U,V0,s0(3));

[X2D, Z2D, cd] = dragkv22D(t,V0,U);

Sa = analysissolution(t,Vx0,Vz0,X0,Z0,g);
Xa = Sa(1,:);
Za = Sa(2,:);
%% RMS
% masschangeRMS
squared_errorm = (Zm - Z).^2  ;
mean_squared_errorm = mean(squared_errorm);
RMSm = sqrt(mean_squared_errorm);

% dragkV2RMS
squared_errorD = (ZD - Z).^2  ;
mean_squared_errorD = mean(squared_errorD);
RMSD = sqrt(mean_squared_errorD);

% dragkV22DRMS
squared_errorD2D = (X2D - X).^2+(Z2D - Z).^2  ;
mean_squared_errorD2D = mean(squared_errorD2D);
RMSD2D = sqrt(mean_squared_errorD2D);
```

Fig. 29. Imply Dormand-Prince and Dynamic model 2

```matlab
% analyticalsolutionRMS
squared_errorA = (Xa - X).^2+(Za - Z).^2  ;
mean_squared_errorA = mean(squared_errorA);
RMSA = sqrt(mean_squared_errorA);

%
if Cd == 0 && c ~= 0 && theta == 90
    figure
    plot(t,Z,'r' ,'LineWidth', 2)
    hold on
    plot(t,Zm,'b','LineWidth', 2)
    xlabel('t s')
    ylabel('Z m')
    grid on
    title(['Z-t,\theta = ', num2str(theta),' c = ', num2str(c),' Vex = ', num2str(Vex),' Vz0 = ' ...
        , num2str(Vz0),' Z0 = ', num2str(Z0),' RMS = ', num2str(RMSm),' masschanging 1-D, sample time = ', num2str(h)])
    legend('Numerical Solution','Analytical Solution')
elseif Cd ~= 0 && c == 0 && theta == 90
    figure
    plot(t,Z,'r','LineWidth', 2)
    hold on
    plot(t,ZD,'b','LineWidth', 2)
    xlabel('t s')
    ylabel('Z m')
    grid on
    title(['X-t,\theta = ', num2str(theta),' \rho = ', num2str(rho),' Cd = ', num2str(Cd),' S = ' ...
        , num2str(S),' Vz0 = ', num2str(Vz0),' Z0 = ', num2str(Z0),' RMS = ', num2str(RMSD),' drag(kV^2), sample time = ', num2str(h)])
    legend('Numerical Solution','Analytical Solution')
```

Fig. 30. Imply Dormand-Prince and Dynamic model 3

```matlab
elseif Cd ~= 0 && c == 0 && theta ~= 90
    figure
    plot(X,Z,'r','LineWidth', 2)
    hold on
    plot(X2D,Z2D,'b','LineWidth', 2)
    xlabel('X m')
    ylabel('Z m')
    grid on
    title(['X-t,\theta = ', num2str(theta),' \rho = ', num2str(rho),' Cd = ', num2str(Cd),' S = ' ...
        , num2str(S),' Vz0 = ', num2str(Vz0),' Z0 = ', num2str(Z0),' RMS = ', num2str(RMSD2D),' drag(kV^2), sample time = ', num2str(h)])
    legend('Numerical Solution','Analytical Solution')
elseif c == 0 && Cd == 0
    if theta == 90
        figure
        plot(t,Z,'r','LineWidth', 2)
        hold on
        plot(t,Za,'b','LineWidth', 2)
        xlabel('t s')
        ylabel('Z m')
        grid on
        title(['X-Z,\theta = ', num2str(theta),' RMS = ', num2str(RMSA),' sample time = ', num2str(h)])
        legend('Numerical Solution','Analytical Solution')
    else
        figure
        plot(X,Z,'r','LineWidth', 2)
        hold on
        plot(Xa,Za,'b','LineWidth', 2)
        xlabel('X m')
        ylabel('Z m')
        grid on
        title(['X-Z,\theta = ', num2str(theta),' RMS = ', num2str(RMSA),' sample time = ', num2str(h)])
        legend('Numerical Solution','Analytical Solution')
    end

```

Fig. 31. Imply Dormand-Prince and Dynamic model 4

```matlab
else
    figure
    plot(X,Z,'r','LineWidth', 2)
    xlabel('X m')
    ylabel('Z m')
    title(['X-Z,\theta = ', num2str(theta),' \rho = ', num2str(rho),' Cd = ', num2str(Cd),' S = ' ...
        , num2str(S),' c = ', num2str(c),' Vex = ', num2str(Vex),' Vx0 = ', num2str(Vx0),' Vy0 = ' ...
        , num2str(Vy0),' Vz0 = ', num2str(Vz0), ' Z0 = ', num2str(Z0),' sample time = ', num2str(h)])
    grid on
    axis equal
end
```

Fig. 32. Imply Dormand-Prince and Dynamic model 5