Semi-empirical Simulation Method for High-Powered Model Rocketry

by
Jacob A. Hales

A senior thesis submitted to the faculty of
Brigham Young University – Idaho
in partial fulfillment of the requirements for the degree of

Bachelor of Science

Department of Physics
Brigham Young University – Idaho
December 2014
DEPARTMENT APPROVAL

of a senior thesis submitted by

Jacob A. Hales

This thesis has been reviewed by the research committee, senior thesis coordinator, and department chair and has been found to be satisfactory.

Date: ___________________________  David Oliphant, Advisor

Date: ___________________________  Evan Hansen, Senior Thesis Coordinator

Date: ___________________________  Stephen Turcotte, Committee Member

Date: ___________________________  Stephen McNeil, Department Chair
ABSTRACT

SEMI-EMPIRICAL SIMULATION METHOD FOR HIGH-POWERED MODEL ROCKETRY

Jacob A. Hales
Department of Physics
Bachelor of Science

High-power model rocket trajectories are difficult to predict because they are highly sensitive to the vehicle shape and weather conditions. This thesis presents a numerical strategy for simulating the flight path based on two known semi-empirical techniques: the Barrowman method and the United States Air Force Stability and Control Datcom method. The methods were implemented in a C++ program named BIRD (BYU-I Rocket Dynamics). The simulator includes six degrees of freedom and accepts detailed input including the rocket geometry and thrust curves from static motor tests. To estimate measurement error, a Gaussian-weighted Monte Carlo framework is included in the simulator. Specifically, this allows for estimates in the error for apogee (maximum altitude) and landing position. Program output is compared with OpenRocket and Cambridge Rocketry Simulator for validation. BIRD predicts apogee positions consistent with these other simulators, but differs in upwind landing position. Further work is needed to clarify the cause of the discrepancies and validate other flight parameters. Also, further work is recommended in comparing the program to GPS data from real launches.
ACKNOWLEDGEMENTS

I would like to thank my wife Tanya and all of the members of my thesis committee for their help in writing and revising this thesis. I would especially like to thank Dr. Evan Hansen for directing the thesis and spending several long sessions with me in his office critically reviewing the thesis content. Finally, I would like to thank all of the members of the BYU-Idaho physics department. Education is not an individual effort, and without all of you I wouldn’t be where I am today. Truly, thank you.
# CONTENTS

Abstract ................................................................................................................................. v
Semi-empirical Simulation Method for High- ................................................................. v
Powered Model Rocketry ................................................................. v
Acknowledgements ............................................................................................................. vii
List of Figures ....................................................................................................................... x
List of Tables ......................................................................................................................... x

## Chapter 1 Introduction ........................................................................................................ 11
  1.1 Background ..................................................................................................................... 11
  1.2 High Powered Rocketry .................................................................................................. 12
  1.3 Coordinate Systems ........................................................................................................ 14
  1.4 Forces and Stability ......................................................................................................... 16

## Chapter 2 Flight Parameters ............................................................................................... 19
  2.1 Rocket Geometry ............................................................................................................. 19
  2.2 Mass and Moments ......................................................................................................... 21
  2.3 Aerodynamic Properties ................................................................................................. 22
  2.4 Coefficient of Normal Drag ......................................................................................... 24
  2.5 Coefficient of Axial Drag ............................................................................................... 25
  2.6 Atmospheric Properties ................................................................................................. 26

## Chapter 3 Simulation Framework ....................................................................................... 29
  3.1 Simulation Models .......................................................................................................... 29
  3.2 Database and State Variables ....................................................................................... 30
  3.3 Mechanics ....................................................................................................................... 32
  3.4 Numerical Methods ......................................................................................................... 34

## Chapter 4 Simulator Comparison ....................................................................................... 35
  4.1 Method ............................................................................................................................ 35
  4.2 Results ............................................................................................................................ 36
  4.3 Discussion ......................................................................................................................... 37
  4.4 Conclusion ....................................................................................................................... 38

Bibliography .......................................................................................................................... 40
LIST OF FIGURES

Figure 1 A simple rocket design with components listed.......................................................... 13
Figure 2 Phases of flight during the first few seconds................................................................. 14
Figure 3 Phases of flight after the coasting phase................................................................. 14
Figure 4 Roll and pitch axes....................................................................................................... 15
Figure 5 Yaw, pitch, and roll axes............................................................................................ 15
Figure 6 Coordinate systems for aerodynamic forces............................................................. 16
Figure 7 Forces acting on a rocket ............................................................................................ 18
Figure 8 Center of mass and center of pressure........................................................................ 18
Figure 9 Fin geometry.................................................................................................................. 19
Figure 10 OpenRocket Model .................................................................................................. 35
Figure 11 Altitude performance.................................................................................................. 36
Figure 12 Upwind performance .................................................................................................. 37
Figure 13 Center of pressure of OR and BIRD........................................................................... 38

LIST OF TABLES

Table 1 Fin symbols...................................................................................................................... 19
Table 2 Standard Atmosphere Conditions.................................................................................. 28
Table 3 Rocket state variables...................................................................................................... 30
Table 4 Rocket state derivatives.................................................................................................. 30
Table 5 Dynamic database parameters ..................................................................................... 32
Table 6 Altitude Performance...................................................................................................... 36
Table 7 Landing performance....................................................................................................... 37
CHAPTER 1 INTRODUCTION

1.1 Background

Flight simulation for model rockets is both a practical tool for engineers and a method of studying aerodynamic effects on high velocity vehicles. The goal of a rocket simulator is to predict the flight trajectory of a rocket based on its physical properties and the external conditions. Engineers may use a simulator to understand the flight characteristics of a specific model design before investing time and resources into actual construction. Before launch, they may simulate local weather effects such as wind to see if they should make pre-flight adjustments. Research interests may include understanding aerodynamic drag based on the geometrical features of the rocket body and the effects of compressible flow at high velocities. Both scenarios benefit from accurate predictions.

It is challenging to create an accurate simulator because of a rocket’s complex interaction with the atmosphere. The wind is very difficult to predict and tends to contribute a large amount of uncertainty to the overall rocket trajectory. As a model rocket moves through the air, it experiences aerodynamic drag forces that are likewise difficult to estimate. The parameters of these forces can be measured directly in wind tunnel experiments. However, many universities and individuals do not have access to the necessary equipment, and the process is very time consuming. Instead, the aerodynamic parameters are often estimated using semi-empirical or computational methods derived from fluid mechanics.

The semi-empirical methods used for model rocketry are often based on a Masters dissertation completed by Barrowman in the late 1960’s (1) (the “Barrowman method”). His work consolidates various theoretical treatments of slender-body aerodynamics in subsonic and supersonic flight conditions. Mandell et al. followed Barrowman’s work with a full textbook on rocketry flight dynamics (2). They provide a detailed analysis of model rocket design and simulation. More recent literature
includes extensions to the Barrowman method to increase the range of valid rocket configurations (see references (3) and (4)). Also, Box et al. detail the mechanics needed for implementing a six degrees of freedom flight simulation for the major phases of flight (5).

1.2 High Powered Rocketry

High powered rocketry (HPR) is the most interesting scenario for model rocket simulations. A model rocket is classified as high powered when its weight and motor classifications exceed the limits for unrestricted model rocket flight. The National Association of Rocketry (NAR) has documented guidelines that differentiate between normal and high powered vehicles, which can be found online (6). For example, high powered rockets use motors with a total impulse exceeding 160 Newton seconds and/or weighs more than 1.5 kilograms. HPR rockets exhibit all of the characteristics of less powerful rockets, but with additional complexity: they travel faster, higher, and for longer periods of time which allows them to experience a wider range of aerodynamic and environmental variation. Since many of them approach or exceed the speed of sound, compressibility of the air must be taken into account. Also, HPR rockets are more capable of varying payloads such as altimeters which them to take measurements that can be used to validate the simulation software.

Most model rocket bodies are created using a standard set of axisymmetric components. Figure 1 shows a simple rocket design. A rocket begins with a nose cone that typically comes to a point. The main body consists of cylindrical body tubes and conical changes in transition (such as a boat tail at the end). Fins are often trapezoidal, with three or four equally spaced around the bottom of the body tube. Many rockets also include a boat tail, which is a conical transition placed after the main body tube. A launch lug or bolt is responsible for attaching the rocket to a launch rail. A launch rail helps stabilize the rocket during the initial stages of flight.
The interior of the rocket includes one or more motors, one or more parachutes, and a number of other components. For example, an electronics bay may be added with an altimeter, GPS, accelerometer, embedded processor, and other equipment.

High power rockets experience several phases of flight (see Figure 2 and Figure 3). Initially, the rocket is placed on a launch rail or other supporting structure to stabilize it in the first moments of flight. The rocket may be attached using a launch lug or a pair of bolts running along the body of the rocket that slide along the rail. Without a guiding structure and with no initial momentum, a rocket would tend to become unstable. After clearing the launch rail, the rocket will continue accelerating upward until its fuel has burned completely. After the powered phase, the rocket will continue to coast upward and lose velocity under the influences of gravity and drag. Finally, a peak altitude is reached (apogee). More advanced rockets may have multiple stages for thrust, but these cases won’t be examined in this thesis.

As the rocket begins to descend, it may begin to tumble and will deploy a parachute. Some rockets may have multiple parachutes which may be deployed at apogee or other predefined altitudes. An ejection charge is placed near the nose cone which will eject the chute and disrupt the normal geometry of the rocket. During this phase, wind and gravity tend to be the dominating forces.
1.3 Coordinate Systems

Several coordinate systems are used to describe the shape and state of a rocket during simulation. A cylindrical coordinate system will be used to specify the shape of a rocket. In the simulation, the rocket’s position and orientation will be represented in Cartesian world coordinates. World coordinates will be defined as the position relative to the initial position, oriented normal to the earth’s surface. In some occasions, it will be useful to represent the rocket in local Cartesian coordinates with the rocket placed at the origin and aligned along the z axis. Additionally, drag forces have two different representations that need some explanation.

It is convenient to use a cylindrical coordinate system to represent a rocket’s shape because the rocket body is radially symmetric. The rocket body is aligned along the z axis (see Figure 4). This axis is often referred to as the roll, longitudinal, or axial axis and it will be labeled $\hat{R}$. Rocket components are usually aligned with the roll axis, but they may have a radial offset $r$ such as in the case of a launch lug. Figure 5 shows the axes used to specify the spatial rotation of a rocket as commonly used in aviation. A model rocket tends to spin about its roll axis and tilt about the pitch axis. The yaw axis can be ignored for practical purposes as it is usually indistinguishable from pitch.
World coordinates and local coordinates describe the rocket’s position in the world’s reference frame and the rocket’s reference frame, respectively. Simulators track the rocket in world coordinates with the origin at the initial position and an orientation normal to the earth’s surface (i.e. aligned with a launch rail that has not been tilted). Local coordinates place the rocket’s center of mass at the origin, and it is aligned with the roll axis (similar to Figure 4). The local reference frame is used for calculating the forces and torques acting on a simulated vehicle.

There are two coordinate systems used in rocket literature to describe the aerodynamic forces (see Figure 6). In flight, a rocket’s velocity direction vector $\hat{V}$ is not always aligned with the roll axis $\hat{R}$. The angle between these two unit vectors defines the angle of attack $\alpha$. When the angle of attack is non-zero, drag forces can be viewed as acting opposite either to the velocity vector or opposite to the roll axis. The first representation of drag forces defines an axial drag force $\vec{F}_A$ directed against the roll axis, and a normal drag force $\vec{F}_N$ pushing sideways against the rocket body. The other interpretation defines the drag force $\vec{F}_D$ which acts opposite to the velocity vector, and the perpendicular lift force $\vec{F}_L$. 

Figure 4 Roll and pitch axes.

Figure 5 Yaw, pitch, and roll axes.
which acts sideways. Results from one coordinate system can be converted to another as needed. BIRD will use the normal/axial coordinate system.

![Coordinate systems for aerodynamic forces.](image)

**Figure 6 Coordinate systems for aerodynamic forces.**

### 1.4 Forces and Stability

Model rocket design involves understanding the forces acting on a rocket and how they influence its flight stability and performance. There are three dominant forces in rocket flight: thrust, gravity, and drag. Figure 7 shows each of the forces, with the drag force split into two components (the axial and normal aerodynamic forces). Thrust and gravity are straightforward to measure and approximate. A complete discussion of the aerodynamic forces is outside of the scope of this thesis and the underlying theory will only be mentioned briefly. However, Mandell presents a good introduction to the theory in reference (2).

Thrust delivers the force needed to overcome gravity and propel the rocket skyward. Model rockets typically use solid propellant that burns completely within a few seconds. All thrust is typically directed forward along the roll axis, with none of the fuel being used for corrective steering. As the fuel
burns, the motor’s internal pressure builds and combustion gases are rapidly expelled from the motor nozzle. The gases have little mass, but their high exit velocity contributes significantly to their momentum. The rocket conserves the change in momentum by launching forward with appreciable force. Thrust from a motor can be measured directly through a static engine test and recorded as a thrust versus time curve. Rocket motors come in many varieties which offer different thrust curve characteristics and total impulse.

Gravity acts constantly in the negative \( z \) direction in world coordinates and varies with time. Although high powered rockets can achieve impressive altitudes, the gravitational force variation with altitude can usually be neglected. More importantly, a rocket’s weight changes during the powered phase as the fuel burns. Propellant is a significant fraction of a rocket’s total weight. Its depletion results in a significant change in the total gravitational force acting on the rocket.

Model rockets react strongly with the surrounding atmosphere, resulting in aerodynamic forces. The vehicle’s motion creates pressure differences in the air (pressure drag), and the rocket’s surface experiences friction forces through contact with the fluid (viscous drag). If the rocket’s roll axis and velocity vector are aligned (i.e. the angle of attack is zero), the net aerodynamic force will be entirely directed opposite to the roll axis (i.e. no sideways force. It should be noted that in this case, the two coordinate systems in Figure 6 are identical). If the angle of attack is non-zero, then there will be a sideways force on the rocket that may induce a torque on the vehicle. This occurs as one side of the rocket becomes somewhat shielded from the incoming air flow, creating an imbalance in aerodynamic forces. Aerodynamic forces play a critical role in a rocket’s overall stability.

Especially in high powered rocketry, flight stability and safety is a critical concern. The stability of a rocket can be predicted based on the relationship between the center of gravity and center of pressure. Most forces act on the rocket’s center of mass. Normal aerodynamic forces, however, act at a location known as the center of pressure (COP) which is determined by the rocket shape and
atmospheric pressure distribution. At zero velocity and horizontal wind, the COP is the centroid of the side profile of a rocket. Wind is evenly distributed over the side of the rocket, and will impart more force to the fins because of their larger area. At non-zero velocity, the center of pressure depends on the angle of attack and can be estimated using the Barrowman method.

A rocket is stable if the center of pressure is located below the center of mass (see Figure 8). The difference between the center of mass and center of pressure defines the moment arm when a torque is imparted on the rocket. As the moment arm length increases, the magnitude of corrective torques increases making the vehicle more stable. Stability causes the rocket to turn towards the wind until the roll axis has the same direction as the relative atmospheric velocity (i.e. the angle of attack goes to zero). Angle of attack tends to be greatest just after launch since the launch rail constrains the roll axis irrespective of the wind. The launch rail can be tilted to face toward the wind to help reduce the initial angle of attack for a more stable launch.
CHAPTER 2 FLIGHT PARAMETERS

2.1 Rocket Shape

Calculating the aerodynamic properties of a rocket requires knowing its geometric properties in detail. This includes finding the volume, surface area, and cross-sectional areas of the rocket body and fins. Because of cylindrical symmetry, the entire rocket body shape can be defined with a piecewise function \( r(z) \) revolved about an axis to generate a surface (with \( r \) and \( z \) as the radius and height in a cylindrical coordinate system). Each rocket component contributes a line segment \( r_i(z) \) to the overall curve:

\[
r(z) = \begin{cases} 
  r_0(z) & 0 \leq z < z_0 \\
  r_1(z) & z_0 \leq z < z_1 \\
  \vdots & \vdots \\
  r_i(z) & z_{i-1} \leq z < i 
\end{cases}
\]

Fins are usually trapezoidal in shape and equally spaced around a rocket. A trapezoidal fin is specified by the symbols shown in Figure 9 and defined in Table 1:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_\theta )</td>
<td>Sweep angle</td>
</tr>
<tr>
<td>( S_l )</td>
<td>Sweep length</td>
</tr>
<tr>
<td>( S )</td>
<td>Fin span (height)</td>
</tr>
<tr>
<td>( C_r )</td>
<td>Root chord</td>
</tr>
<tr>
<td>( C_m )</td>
<td>Mid chord</td>
</tr>
<tr>
<td>( C_t )</td>
<td>Tip chord</td>
</tr>
<tr>
<td>( d_f )</td>
<td>Body diameter at the fins</td>
</tr>
<tr>
<td>( t )</td>
<td>Fin thickness</td>
</tr>
</tbody>
</table>

Table 1 Fin symbols

There are two cross-sectional areas of interest: the axial and radial cross-sections (where the axis is normal to the cross-sectional plane). The axial cross-section (usually just referred to as the cross-
section) is a top-down projection of the rocket. Its area is the same as a circle generated by the radius \( r(z) \) at the specified height:

\[
A = \pi r(z)^2
\]  

(2)

Fin axial cross-sections will be treated as negligible. The radial cross-section (planform area) is a sideways projection of the rocket (e.g. Figure 1). It can be calculated for the rocket body using equation (3):

\[
A_b = 2 \int_{z_i}^{z_f} r(z) \, dz
\]

(3)

For trapezoidal fins, the equation for the planform area is simply the area of a trapezoid:

\[
A_f = \frac{1}{2} (C_r + C_t)S
\]

(4)

The planform area of the fins is often projected into the body of the rocket to account for interference effects with the body. It then includes the area created by extending the leading and trailing edges of the fin to the rocket centerline. The projected fin planform area can be derived from the fin planform area with equation (5) as shown in reference (7):

\[
A_{fp} = A_f + \frac{1}{2} d_f C_r
\]

(5)

Surface area of the rocket body is calculated by revolving the rocket’s defining curve \( r(z) \) about an axis. Reference (8 p. 467) provides a general formula for determining the surface area of an arbitrary curve on the \( x \) axis by revolving it about the \( y \) axis:

\[
S = \int_{a}^{b} 2\pi y \sqrt{1 + \left( \frac{dy}{dx} \right)^2} \, dx = \int_{a}^{b} 2\pi y \sqrt{1 + (f'(x))^2} \, dx
\]

(6)

\( x \) and \( y \) can be replaced with \( z \) and \( r \), respectively. \( a \) and \( b \) represent the starting and ending \( z \) position of the rocket (from 0 to the total length of the rocket). The nose of the rocket should come to a single point and therefore has no frontal area at \( z = 0 \). However, the area at the base of the rocket has not been included and can be added as an extra step if needed.

Finally, the volume of the rocket body can be calculated by generating a solid of revolution. The disk method will be used as specified by reference (8) on page 467:
\[ V = \int_a^b A(x)dx = \int_a^b \pi[R(x)]^2dx \]  
(7)

Again, \( x \) and \( y \) can be replaced with \( z \) and \( r \), and the integration limits are the same.

### 2.2 Mass and Moments

The mass-related quantities of model rockets can be calculated directly by dividing the rocket into components and then recombining their individual influences. For example, the mass of the rocket is the sum of the mass of each part. Although in practice measuring the mass directly is more accurate, the theoretical approach serves as a good estimate.

Taylor (9 p. 87) defines the center of mass of a system as the mass-weighted average of each particle in the system with respect to a common origin:

\[ \vec{R} = \frac{1}{M} \sum_{\alpha=1}^{N} m_{\alpha} \vec{r}_{\alpha} \]  
(8)

Here \( \vec{R} \) is the center of mass, \( M \) is the system mass, and \( m_{\alpha} \) is the mass of particle \( \alpha \). This vector equation can be simplified by assuming an axisymmetric rocket parallel to the Z axis. Each rocket component is modeled as a particle positioned at its center of mass in the rocket’s reference frame:

\[ Z_{cm}(R) = \frac{1}{M(R)} \sum_{c \in R} [Z_{c} + Z_{cm}(c)] \]  
(9)

\( R \) and \( c \) represent the rocket and one of its components, respectively. \( Z_{c} \) is the component’s offset from the tip of the nose cone, and \( Z_{cm} \) is the center of mass from the top of the component. \( M(R) \) is the total mass of the rocket.

The moment of inertia is also defined by Taylor (9). Below, \( I \) represents the moment inertia for a given axis, and \( \rho_{\alpha} \) is the distance of the mass from the axis:

\[ I = \sum_{\alpha=1}^{N} m_{\alpha} \rho_{\alpha}^2 \]  
(10)
The parallel axis theorem is used to compute the moment of inertia for the entire rocket from its individual parts. As an example, the moment of inertia about the Z axis is calculated as follows:

\[
I_{zz}(R) = \frac{1}{M(R)} \sum_{c \in R} (I_{zz}(c) + m(c)[Z_{cm}(R) - Z_{cm}(c)]^2)
\]  

(11)

Assuming axial symmetry and calculating moments about the center of mass, the moment of inertia tensor is a diagonal matrix that is easily inverted:

\[
\begin{bmatrix}
I_{xx} & 0 & 0 \\
0 & I_{yy} & 0 \\
0 & 0 & I_{zz}
\end{bmatrix}^{-1} = \begin{bmatrix}
I_{xx}^{-1} & 0 & 0 \\
0 & I_{yy}^{-1} & 0 \\
0 & 0 & I_{zz}^{-1}
\end{bmatrix}
\]  

(12)

During powered flight, the mass, center of mass, and moments of inertia change as the propellant burns. Box et al. estimate the mass as varying by the ratio of the current impulse over the total thrust impulse (7). The mass of the fuel at time \( t \) is related to the original mass \( M_0 \) by:

\[
M(t) = \left(1 - \int_0^t T \, dt / \int_0^\infty T \, dt\right) M_0
\]  

(13)

\( T \) represents the time-varying thrust of the rocket and can be interpolated from the thrust-time curve of the rocket motor. After calculating the new mass, the center of mass and moments of inertia can be updated for the current time \( t \).

2.3 Aerodynamic Properties

Several aerodynamic properties play a critical role in characterizing the air streams flowing over a model rocket. Aerodynamic coefficients encode the shape of the rocket and how the rocket responds to fluid flow. The ratio of the fluid inertia force to the friction force is represented by the Reynolds number \( R_e \). Speed relative to the atmosphere is often given in terms of the Mach number \( M_a \), which represents the ratio of the vehicle’s speed to the local speed of sound. All of the aerodynamic coefficients are functions of the angle of attack, Mach number, and Reynolds number.
Drag forces are analyzed by considering a dimensionless quantity known as the drag coefficient that represents the atmospheric effects on a particular shape. Mandell et al. (7) give the drag coefficient as

\[ C_D = \frac{D}{\frac{1}{2} \rho A_r V^2} \]  

where \( D \) is the drag force, \( \rho \) is the fluid density, \( A_r \) is the reference area, \( V \) is the object velocity, and \( C_D \) is the drag coefficient. In rocketry, the reference area is typically the area at the base of the nose or the maximum cross-sectional area. Knowing the drag coefficient allows the drag force to be calculated at varying velocity and density. However, the drag coefficient is often a function of other variables such as the angle of attack, Mach number, and Reynolds number.

The Reynolds number is used to determine the effects of frictional forces on the rocket. It is defined as:

\[ R_e = \frac{\rho V L}{\mu} \]  

\( \rho \) is the atmospheric density, \( L \) is the characteristic length, \( V \) is the object velocity, and \( \mu \) is the coefficient of viscosity. The characteristic length differs for the rocket body and fins. For the rocket body, \( L \) is the total rocket length (without fins). According to Box et al., \( L \) is the mid chord for fins (7).

Mach number is the velocity given in terms of the local speed of sound, thereby characterizing the velocity in terms of the kinetic properties of the fluid. Box et al. (7) calculate the Mach number as follows:

\[ Ma = \frac{V}{\sqrt{\gamma R \Theta_A}} \]  

Here \( V \) is the velocity, \( \gamma \) is the ratio of specific heats for air, \( R \) is the gas constant for air, and \( \Theta_A \) is the ambient (local) temperature.

The Prandtl-Glauert compressibility correction adjusts the aerodynamic coefficients to account for compressible flow. In the derivations for drag coefficients, the air is approximated as an incompressible fluid. In reality, air will compress significantly at high velocities. Box et al. (7) account for
this using a stepwise function that also avoids a singularity in the Prandtl-Glauert correction that occurs as Mach number approaches one:

\[
C_i' = \begin{cases} 
  \frac{C_i}{\sqrt{1 - Ma^2}} & Ma \leq 0.8 \\
  \frac{C_i}{\sqrt{1 - 0.8^2}} & 0.8 < Ma < 1.1 \\
  \frac{C_i}{\sqrt{Ma^2 - 1}} & Ma \geq 1.1 
\end{cases}
\] (17)

\(Ma\) equals Mach number, and \(C_i\) represents the value of any given drag coefficient. This method can be used to adjust both the normal and axial drag to account for compressible flow.

\section*{2.4 Coefficient of Normal Drag}

Barrowman provides a method for estimating the coefficient of normal force and the center of pressure for a rocket that does not violate several assumptions. These assumptions place restrictions on the rocket flight, including small angles of attack, steady flow, rigidity of the rocket, and the assumption that the nose comes to a sharp point. Barrowman examines the fins and body separately.

First, the coefficient of normal force and center of pressure will be calculated. Barrowman (1) derives an expression for the coefficient of normal force for the rocket body:

\[
C_N(b) = \frac{2\alpha}{A_r} [A(l_0) - A(0)] 
\] (18)

\(l_0\) is the position at the base of the rocket, \(A(z)\) is the area, and \(\alpha\) is the angle of attack. \(A_r\) is the reference area which will be defined as the area at the base of the nose cone. Barrowman also derives the center of pressure along the \(z\) axis:

\[
Z_{cp}(b) = l - \frac{V}{A_b} 
\] (19)

\(l\) is the length of the rocket, \(V\) is the volume of the rocket body, and \(A_b\) is the area at the base of the rocket. The coefficient of normal force for the fins is given in equation (1):
\[ C_N(f) = \alpha \left( 1 + \frac{d_f}{Z} \right) \left( \frac{4n \left( \frac{S}{d_n} \right)^2}{1 + \left( \frac{2C_m}{C_r + C_t} \right)^2} \right) \]

\( d_n \) is the diameter at the base of the nose cone. Fin center of pressure can be found using the following equation, where \( Z_f \) is the vertical displacement from the top of the root chord to the tip of the nose:

\[ Z_{cp}(f) = Z_f + \frac{C_m(C_r + 2C_t)}{3(C_r + C_t)} + \frac{1}{6} \left[ C_r + C_t - \frac{C_r C_t}{C_r + C_t} \right] \]

The fin and body coefficients of normal force and centers of pressure can then be combined together as follows:

\[ C_N(R) = C_N(b) + C_N(f) \]

\[ Z_{cp}(R) = \frac{C_N(b)Z_{cp}(b) + C_N(f)Z_{cp}(f)}{C_N(b) + C_N(f)} \]

### 2.5 Coefficient of Axial Drag

The coefficient of axial drag will be estimated using the United States Air force Stability and Control Datcom method (referred to as the DATCOM method in this paper). This method was employed by Mandell et al. in reference (2). Their results will be presented here, with some simplifications and adjustments. The DATCOM method splits the rocket into three components (fins, forebody, and afterbody) and recombines them to determine the overall drag coefficient.

The fin coefficient includes the friction forces acting on the fins as well as an interference effect that occurs at the junction between the body and the fins:

\[ C_A(f) = 2C_f(f) \left( 1 + \frac{t}{C_r + C_t} \right) \left( \frac{2A_{fp} - A_f}{A_r} \right) \]

\( A_{fp} \) is the total extended fin planform area, \( A_p \) is the normal fin planform area, and \( A_r \) is the reference area (in this case the reference area is the maximum axial cross-section). \( C_f(f) \) is the fin coefficient of friction, which will be defined later in equation (28). The equation for the forebody is:
\[
C_A(fb) = C_f(b) \left[ 1 + \frac{60}{(l/d_m)^3} + 0.0025 \left( \frac{l}{d_m} \right) A_b / A_r \right] \tag{25}
\]

\(l\) is the length of the rocket, \(d_m\) is the maximum diameter, and \(A_b\) is the surface area of the entire body, excluding the circular area at the base. The afterbody drag coefficient is then given by:

\[
C_A(ab) = 0.029 \frac{(l/d_m)^3}{\sqrt{C_A(fb)}} \tag{26}
\]

The total axial drag coefficient is simply the sum of the fin, forebody, and afterbody coefficients:

\[
C_A(R) = C_A(f) + C_A(fb) + C_A(ab) \tag{27}
\]

Viscous friction is estimated from the Reynolds number for turbulent and non-turbulent flow.

The fluid surrounding the rocket must slide past the body and fins during flight. Air at the surface of the rocket must have the same velocity as the vehicle, and a velocity gradient occurs as air further from the rocket reduces to match the atmospheric wind velocity. Rocket momentum is transferred to the surrounding fluid, resulting in drag. This boundary layer formed by the velocity gradient is laminar and smooth for low Reynolds number, and turbulent beyond the critical Reynolds number \(R_c\) (resulting in higher drag). The coefficient of friction drag is given for both the body and fins using their respective Reynolds numbers:

\[
C_f(R_e, R_c) = \begin{cases} 
\frac{1.328}{\sqrt{R_c}} & \text{when } R_e < R_c \\
0.074 \left( \frac{R_c}{R_e} \right)^{1/5} - 0.074 \left( \frac{R_c}{R_e} \right)^{1/5} - \frac{1.328}{\sqrt{R_c}} & \text{when } R_e \geq R_c 
\end{cases} \tag{28}
\]

The axial drag equations are valid at zero angle of attack. However, the DATCOM method actually produces coefficients in the drag/lift coordinate system (see Figure 6). At non-zero angle of attack, the coefficients calculated in this section will be inaccurate. To correct this, the coefficients need to be converted to the normal/axial coordinate system:

\[
C_A \equiv C_D \cos(\alpha) \tag{29}
\]

### 2.6 Atmospheric Properties

The International Standard Atmosphere (ISA) specifies standard conditions for estimating atmospheric properties. Cavcar (10) delineates the assumptions and equations used in ISA weather
modeling which will be summarized here. This model will be used for estimating the atmospheric
temperature, pressure, and density used by the simulator.

The following table lists the mean sea level conditions assumed by the International Standard
Atmosphere (10 p. 2):
<table>
<thead>
<tr>
<th>Condition</th>
<th>Symbol</th>
<th>Sea Level Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pressure</td>
<td>$P_0$</td>
<td>101,325 N/m²</td>
</tr>
<tr>
<td>Density</td>
<td>$\rho_0$</td>
<td>1.225 kg/m³</td>
</tr>
<tr>
<td>Temperature</td>
<td>$T_0$</td>
<td>288.15 K</td>
</tr>
<tr>
<td>Speed of Sound</td>
<td>$a_0$</td>
<td>340.294 m/s</td>
</tr>
<tr>
<td>Acceleration of Gravity</td>
<td>$g_0$</td>
<td>9.80665 m/s²</td>
</tr>
<tr>
<td>Real gas constant for air</td>
<td>$R$</td>
<td>287.04 m²/K s²</td>
</tr>
</tbody>
</table>

Table 2 Standard Atmosphere Conditions

Temperature linearly decreases with altitude until tropopause (11,000 m), after which it remains relatively constant. The decrease is linear and can be expressed as (10 p. 3):

$$T = T_0 - 6.5 \frac{h(m)}{1000}$$

where $T$ is the temperature in kelvin, $T_0$ is the mean temperature at sea level, and $h(m)$ expresses the height above sea level in meters.

Pressure is derived using the Ideal Gas Law, hydrostatic equation, and temperature lapse rate. Air is treated as a perfect gas. Below the tropopause, it can be modeled as (10 p. 5):

$$P = P_0 \left(1 - 0.0065 \frac{h}{T_0}\right)^{5.2561}$$

$h$ is the height in meters, $P_0$ is the mean sea level pressure in Pascals, and $T_0$ is the mean sea level temperature in kelvin.

Having calculated the temperature and pressure, the density of air can be estimated using the ideal gas law:

$$\rho = \frac{P}{RT}$$

Note that $R$ is $R_{specific}$ for air, not just the ideal gas constant (which would lead to a number density, not a mass density).
Chapter 3 Simulation Framework

3.1 Simulation Models

The flight simulator is a numerical integrator that maintains a database of variables and constants. It consists of several simulation models that represent different phases of the integration. Each model determines how to calculate the forces and torques on the rocket for a particular phase of flight. The inputs into the simulator are files that define the characteristics of the rocket, atmosphere, and thrust curve. Results are stored in a CSV file. There is a standard simulation mode that runs a single simulation, and a Monte Carlo mode used to estimate the error in flight path predictions.

Currently the simulator supports three different models. The first is the launch model. Initially, a model rocket is attached to a launch rail. A launch rail allows the rocket to accelerate in order to become stable before entering free flight. The launch model differs from the others in that the rocket’s orientation will be fixed. The rocket is not be allowed to slide backward. The rail may have some friction, however it is considered negligible. This portion of the simulation ends as soon as the rocket displacement exceeds the length of the rail.

After leaving the launch rail, the launch model transitions to the ascent model. During this phase, the rocket will continue to accelerate until the fuel has been spent. The ascent model is the most complex as it involves six degrees of freedom (three for position and three for rotation). This model ends as soon as the rocket’s momentum in the z direction drops below zero, meaning the rocket has reached apogee and is beginning to fall.

The descent model handles the final phase of rocket flight. After reaching apogee, rockets will typically deploy a parachute. BIRD will assume the parachute deploys instantaneously. Once this occurs, the drag of the rocket body becomes negligible relative to the drag caused by the parachute. Rotations
are ignored, reducing the degrees of freedom to three for position. Aerodynamic forces on the rocket are ignored except on the parachute itself which has a constant coefficient of drag. The model ends as soon as the rocket reaches the ground.

### 3.2 Database and State Variables

Five state variables uniquely identify the state of the rocket during simulation (see Table 3). All other values—center of pressure, coefficients of drag, temperature, thrust, etc.—are functions of these variables. The state variables are differentiated at each time step by the numerical integrator in order to solve the kinematic equations for the rocket (Table 4). During differentiation, they are used to index into a database of parameters that represent the properties of the rocket and the environment (Table 5).

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T )</td>
<td>Time (in seconds) since launch</td>
</tr>
<tr>
<td>( \vec{Q} )</td>
<td>Orientation quaternion</td>
</tr>
<tr>
<td>( \vec{x} )</td>
<td>Position</td>
</tr>
<tr>
<td>( \vec{v} )</td>
<td>Linear velocity</td>
</tr>
<tr>
<td>( \vec{ω} )</td>
<td>Angular velocity</td>
</tr>
</tbody>
</table>

**Table 3 Rocket state variables**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( δt )</td>
<td>Time step</td>
</tr>
<tr>
<td>( \vec{Q} )</td>
<td>Quaternion derivative</td>
</tr>
<tr>
<td>( \vec{v} )</td>
<td>Velocity</td>
</tr>
<tr>
<td>( \vec{a} )</td>
<td>Linear acceleration</td>
</tr>
<tr>
<td>( \vec{α} )</td>
<td>Angular acceleration</td>
</tr>
</tbody>
</table>

**Table 4 Rocket state derivatives**

The state variables used by the simulator are encountered frequently in kinematics and will not be explained in detail here, with the exception of a brief introduction to quaternions. Quaternions may be represented by a scalar and a vector part (for a total of four components), where the scalar can be visualized as an angular rotation around the vector:

\[
\vec{Q} = [s, \vec{v}] \tag{33}
\]
They have many uses, particularly for representing three-dimensional rotations and orientations. Each state variable has a corresponding state derivative. Linear and angular acceleration are calculated by summing the forces and torques acting on the rocket:

\[
\mathbf{a} = \frac{1}{M} \sum \mathbf{F}_i, \quad (34)
\]
\[
\mathbf{a} = R(I)^{-1}R^T \sum \mathbf{\tau}_i. \quad (35)
\]

Here \( M \) represents the mass, \( R \) represents the rotation from world coordinates to local coordinates, and \( I \) is the moment of inertia tensor. It will often be easier to work with forces and torques in local coordinates. The quaternion derivative was given by Box et al. as follows (see reference (5)):

\[
\dot{\mathbf{Q}} = \left[ \frac{1}{2} (\mathbf{\omega} \cdot \mathbf{v}), \frac{1}{2} (s\mathbf{\omega} + (\mathbf{\omega} \times \mathbf{v})) \right] \quad (36)
\]

The parameter database contains the rocket geometry, environment data, and constants needed for the simulator. Some of these values are dependent on the current state of the rocket. Thrust is a time-dependent quantity that will go to zero after the fuel has been burned. This in turn affects the mass, center of mass, and moments of inertia. Each of the drag coefficients depend on the angle of attack and velocity. Center of pressure depends on angle of attack only. Wind, density, and temperature are all related to the altitude of the rocket.
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>Thrust</td>
<td>$t$</td>
</tr>
<tr>
<td>$M$</td>
<td>Mass</td>
<td>$t$</td>
</tr>
<tr>
<td>$X_{cm}$</td>
<td>Center of mass (relative to nose tip)</td>
<td>$t$</td>
</tr>
<tr>
<td>$I$</td>
<td>Moment of inertia tensor</td>
<td>$t$</td>
</tr>
<tr>
<td>$C_A$</td>
<td>Axial drag coefficient</td>
<td>$\alpha, R_e, M_a$</td>
</tr>
<tr>
<td>$C_N$</td>
<td>Normal drag coefficient</td>
<td>$\alpha, R_e, M_a$</td>
</tr>
<tr>
<td>$X_{cp}$</td>
<td>Center of pressure (relative to nose tip)</td>
<td>$\alpha$</td>
</tr>
<tr>
<td>$\vec{W}$</td>
<td>Wind velocity</td>
<td>$z$</td>
</tr>
<tr>
<td>$\rho$</td>
<td>Air density</td>
<td>$z$</td>
</tr>
<tr>
<td>$\Theta_A$</td>
<td>Temperature</td>
<td>$z$</td>
</tr>
</tbody>
</table>

Table 5 Dynamic database parameters

### 3.3 Mechanics

In order to calculate the forces and torques acting on the rocket, the database dependencies need to be resolved. Time and altitude are given directly in the state variables. Angle of attack ($\alpha$) and the atmospheric velocity ($\vec{V}$) will need to be derived. This section is based on the work of Box et al. in reference (5) and the same equations are found in their article:

\[
\vec{V} = \vec{V}_{cm} + \vec{V}_\omega \tag{37}
\]

\[
\alpha = \cos^{-1}(\vec{V} \cdot \vec{R}_A) \tag{38}
\]

$\vec{V}_{cm}$ is the center of mass velocity relative to the atmosphere, which is simply $\dot{X} + \vec{W}$. $\vec{V}_\omega$ is the linear velocity of the center of pressure, which for simplicity (and to a good approximation) will be taken as zero. $\vec{R}_A$ is the roll axis (which points in the same direction as the rocket nose). Note that the roll axis needs to be rotated into rocket coordinates based on the rocket's orientation by converting the quaternion into a rotation matrix:

\[
R = \begin{bmatrix}
1 - 2v_y^2 - 2v_z^2 & 2v_xv_y - 2sv_z & 2v_xv_z + 2sv_y \\
2v_xv_y + 2sv_z & 1 - 2v_x^2 - 2v_z^2 & 2v_yv_z - 2sv_x \\
2v_xv_z - 2sv_y & 2v_yv_z - 2sv_x & 1 - 2v_x^2 - 2v_y^2
\end{bmatrix}
\tag{39}
\]

\[
\vec{R}_A = RR^T_{A,0} \tag{40}
\]
Each model will involve one or more forces. Thrust is a time-dependent quantity that has a positive component in the direction of the roll axis:

\[ \vec{F}_T = -T \vec{R}_A \] (41)

Gravity can be calculated as in (42), where \( g \) is approximated in (43) shown below:

\[ \vec{F}_g = [0, 0, -mg]^T \] (42)
\[ g = G \frac{Me}{(r_E + z)^2} \] (43)

\( G \) is the gravitational constant, \( Me \) is the mass of the earth, and \( r_E \) is the radius of the earth. Note that this model assumes a spherical earth and does not take into account the centrifugal force resulting from a rotating reference frame.

Drag force is split into two components. Axial drag acts in the direction opposite of the roll axis. The direction of normal drag is more involved as it occurs in the plane formed by the roll axis and apparent velocity, and it is orthogonal to the roll axis:

\[ F_A = \frac{1}{2} \rho V^2 A_r C_A \] (44)
\[ \vec{F}_A = -F_A \vec{R}_A \] (45)
\[ F_N = \frac{1}{2} \rho V^2 A_r C_N \] (46)
\[ \vec{F}_N = F_N \left( \vec{R}_A \times (\vec{R}_A \times \vec{V}) \right) \] (47)

\( A_r \) is the area at the base of the nose for the normal force, and the maximum cross-sectional area for the axial force. \( \rho \) is the atmospheric density, \( V \) is the apparent velocity, and \( C_A \) and \( C_N \) are the axial and normal drag coefficients, respectively.

In this model, torque on the rocket comes entirely from the normal force. The normal force acts on the center of pressure, not the center of gravity. This leads to a force that tends to push the rocket (47), as well as a torque that tends to turn the rocket into the wind:

\[ \tau_N = F_N Z (\vec{R}_A \times \vec{V}) \] (48)

Where \( Z \) is the difference between the center of pressure and the center of mass.
3.4 Numerical Methods

The simulator uses a fourth-order Runge-Kutta method integrator with a Monte-Carlo wrapper. Integration proceeds until the current model reaches its end condition (such as apogee). Models are arranged in a sequence. The termination of one model will begin the next until all models have run. Most often, a simulation will consist of the launch, ascent, and descent models in that order. Each state vector is saved after every iteration so that the results can be used to form a full analysis of the rocket’s trajectory. Weather conditions, variation in thrust curves from fuel, and imperfections in rocket design can all lead to a high level of error in the simulated results. The goal of the Monte-Carlo wrapper is to predict the total error in the rocket trajectory based on each error contribution.

The Monte-Carlo method can be used to predict the error in simulated values by using a mean value and standard deviation for each measurement. A single Monte-Carlo simulation will consist of a large number of runs. Before each iteration, the parameter database will be populated with the mean value for each measurement. Then, a Gaussian-weighted random number generator will add a random error offset to each database variable using its associated standard deviation. The iteration will then proceed as normal. After all of the iterations have finished, the mean and standard deviation for the apogee and landing position are calculated using the output from each run.
CHAPTER 4 SIMULATOR COMPARISON

4.1 Method

In order to validate BIRD, the simulator was compared with two other programs: Cambridge Rocketry Simulator (CR) and OpenRocket (OR). A common rocket model was designed and duplicated among the three simulators in their respective file formats. Tests were run under the same atmospheric conditions. The primary aim of this analysis was to evaluate the altitude and landing position estimates generated by BIRD. Conditions were chosen to exercise the treatment of wind and compressibility effects.

Both of the chosen simulators are open source software. Cambridge Rocketry simulator was created by Box et al. based on their two papers (references (5) and (7)). It is not widely used, but its methods are similar to BIRD (Barrowman and DATCOM). OpenRocket is widely popular among model rocket hobbyists and is based on a Master’s thesis by Niskanen (4). OR uses the Barrowman method with several enhancements.

The rocket model was originally designed in OpenRocket (Figure 10). The length is over two meters with a weight in excess of ten kilograms. The selected motor has an estimated impulse of 1227 \( N \cdot s \). This model is capable of breaking the sound barrier, making compressible flow adjustments necessary.
Each of the simulators were run with the same rocket design and atmospheric conditions. The wind was set to 5 m/s which should cause observable changes in the flight path. A parachute was deployed at apogee. For BIRD, error estimates were included for all of the measurements except thrust. Thrust error was ignored for this revision of the simulator until a reasonable error model is designed. Wind error was set to a modest 0.1 m/s. The Monte-Carlo simulator ran with 1,000 iterations.

4.2 Results

The following plots and tables show the results of the simulations. Figure 11 shows the altitude performance of the three simulators (compare to Figure 2). The altitude of the rocket is plotted against time. The first image shows the entire flight. All three simulators produced similar results. The differences are easier to see in the second image which is zoomed in on the apogee (~30 seconds). BIRD predicted a lower altitude than the others, although the differences are rather small. Table 6 summarizes the apogee measurements. For BIRD, the Monte-Carlo estimate is shown in parenthesis which is in agreement with all of the measured values.

<table>
<thead>
<tr>
<th>Simulator</th>
<th>Apogee (m)</th>
<th>Deviation from BIRD (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cambridge Rocketry</td>
<td>5064</td>
<td>46 (0.92%)</td>
</tr>
<tr>
<td>OpenRocket</td>
<td>5083</td>
<td>65 (1.30%)</td>
</tr>
<tr>
<td>BIRD</td>
<td>5018 (5000 ± 100)</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 6 Altitude Performance

![Figure 11 Altitude performance](image-url)
Upwind position (distance from the launch rail after landing) differs significantly between each program (Figure 12). CR predicts a landing position six times greater than OR (see Table 7). BIRD reverses direction around 10 seconds. After apogee, all of the simulators have a similar slope for the parachute descent. The Monte-Carlo estimate predicts a narrow error margin which none of the other simulators agree with.

<table>
<thead>
<tr>
<th>Simulator</th>
<th>Landing Position (m)</th>
<th>Deviation from BIRD (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cambridge Rocketry</td>
<td>83</td>
<td>1162 (-108%)</td>
</tr>
<tr>
<td>OpenRocket</td>
<td>-449</td>
<td>630 (-58%)</td>
</tr>
<tr>
<td>BIRD</td>
<td>-1079 (-1100± 39)</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 7 Landing performance

Figure 12 Upwind performance

4.3 Discussion

Apogee measurements are in good agreement between all of the simulators, but the landing position measurements are not. The general shapes of the graphs for OpenRocket and Cambridge Rocketry Simulator are similar, but BIRD reverses direction suddenly before apogee. Because all of them disagree, it is impossible at the present to come to any conclusions about what may or may not be wrong. However, there are some features missing in BIRD that may explain its behavior.

OR and BIRD have different methods of calculating the center of pressure. Figure 13 was created using the detailed output from the two programs (which is unavailable in CR). BIRD has a constant center of pressure around 1.6m from the nose tip, whereas OR starts at the same point and moves
downward towards the base. There are at least two reasons why this might occur: Galejs noted that the Barrowman equations do not take into account additional lift forces that occur with long models at a non-zero angle of attack (see reference (3)). OpenRocket uses Galejs’ equations, whereas they are not implemented in BIRD. Also, Barrowman provides corrections to the fin center of pressure to account for compressible flow (1). As a rocket accelerates, the center of pressure of the fins will begin to lower and move towards the base. This also has not been implemented in BIRD. It is reasonable that the center of pressure in the BIRD simulation is too high, causing the rocket to be less stable than in the other simulators.

![Figure 13 Center of pressure of OR and BIRD](image)

Unlike BIRD, OR and CR have been tested against real rocket flights with good results. There is a possibility that BIRD still needs more validation and testing. Also, it would be helpful to test different rocket designs to see if any erroneous patterns emerge.

### 4.4 Conclusion

The BYU-I rocket dynamics simulator has demonstrated results consistent with theory and other rocket simulators. Each of the phases of flight behaved qualitatively as expected. Its apogee closely matched those produced by OpenRocket and Cambridge Rocketry Simulator. However, the upwind
displacements measured by BIRD differed significantly from the other simulators. BIRD needs more testing and validation to ensure that it is functioning correctly. Also, missing features such as fin center of pressure variation with Mach number need to be implemented.

There are several other features not yet included in BIRD. Rocket fins that are canted (not parallel with the rocket body) produce torque on the rocket. Canted fins have their own local area of attack that results in a variety of effects on rocket flight, including causing the rocket to roll. Motor thrust tends to have a roll dampening effect. Also, if the motor nozzle isn’t strictly parallel to the roll axis, it too may induce torques on the vehicle. Thrust also disrupts base drag by altering the pressure at the base of the rocket.

Finally, a better wind model is essential for accurate error measurements. Wind has a substantial impact on landing position. Currently, the simulator treats wind as constant. Real wind varies with altitude and is unpredictable. However, some win models take into account local land features and atmospheric layers. If accurate atmospheric measurements are available, these can be used to predict conditions during flight. With a superior wind model in place, the Monte Carlo simulator can take into account the random wind variations and provide a more realistic estimate on the error in landing position.


APPENDIX A: SOURCE CODE

column.hpp
..........................................................................................................................#ifndef COMMON_HPP
#define COMMON_HPP

#include <cmath>
#include <ctime>
#include <sstream>
#include <iostream>

const double PI = 3.1415926535897932384626433832795028841971693993751058209749;

inline double sqrt(double a) { return a * a; }
inline double cube(double a) { return a * a * a; }
inline double min(double a, double b) { return a < b ? a : b; }
inline double max(double a, double b) { return a > b ? a : b; }

inline double sum(double *a, int n)
{ double result = 0;
  for (int i = 0; i < n; i++)
    result += a[i];
  return result;
}

inline std::string TimeStamp()
{
#pragma warning(disable : 4996)
  std::stringstream ss;
  time_t now = time(0);
  tm *t = localtime(&now);
  ss << 1900 + t->tm_year;
  if (t->tm_mon + 1 < 10) ss << 0;
  ss << t->tm_mon + 1;
  if (t->tm_mday < 10) ss << 0;
  ss << t->tm_mday;
  if (t->tm_hour < 10) ss << 0;
  ss << t->tm_hour;
  if (t->tm_min < 10) ss << 0;
  ss << t->tm_min;
  if (t->tm_sec < 10) ss << 0;
  ss << t->tm_sec;
  return ss.str();
}

inline double BoundedAcos(double val)
{ if (val > 1.0)
  return 0;
}

inline void FatalError(std::string message)
{ std::cerr << "Fatal error: " << message << std::endl;
  exit(1); }
#endif

csv.hpp
..........................................................................................................................#ifndef CSV_HPP
#define CSV_HPP

#include <fstream>
#include <iostream>
#include <string>
#include <map>
#include <vector>
class DataCsvReader
{
private:

std::ifstream m_fileStream;

public:

DataCsvReader(std::string file)
{
    m_fileStream.open(file);
    if (m_fileStream.fail())
    {
        throw std::invalid_argument(file.c_str());
    }
    std::string header;
    getline(m_fileStream, header);
}

~DataCsvReader()
{
    m_fileStream.close();
}

bool NextRow(std::vector<double>& row)
{
    if (m_fileStream.eof())
    {
        return false;
    }
    std::string buffer;
    std::getline(m_fileStream, buffer);
    row.clear();
    for (auto p = strtok((char*)buffer.c_str(), ","); p != nullptr; p = strtok(nullptr, ","))
    {
        row.push_back(strtod(p, nullptr));
    }
    return row.size() > 0;
}

};
#endif

curve.hpp

#ifndef CURVE_HPP
#define CURVE_HPP

#include <vector>
#include <algorithm>

class Curve
{
private:

std::vector<double> mX;
std::vector<double> mY;
std::vector<double> mD;
std::vector<double> mI;

double Interpolate(double v, const std::vector<double>& x, const std::vector<double>& y)
{
    if (v <= x[0])
    {
        return y[0];
    }
    if (v >= x[x.size() - 1])
    {
        return y[y.size() - 1];
    }
    int i = lower_bound(x.begin(), x.end(), v) - x.begin() - 1;
    return y[i] + (y[i + 1] - y[i]) * (v - x[i]) / (x[i + 1] - x[i]);
}

public:

Curve(const std::vector<double>& x, const std::vector<double>& y)
:
    mX(x), mY(y)
{
    mD.push_back(0);
    mI.push_back(0);
}
for (size_t i = 1; i < x.size(); i++)
{
    double dx = x[i] - x[i - 1];
    double dy = y[i] - y[i - 1];
    mD.push_back((y[i] - y[i - 1]) / (x[i] - x[i - 1]));
    mI.push_back(mI[i - 1] + (y[i - 1] + 0.5 * dy) * dx);
}

bool InBounds(double x)
{
    return !(x < mX[0] || x > mX[mX.size() - 1]);
}

double Value(double x)
{
    if (!InBounds(x))
    {
        throw std::out_of_range("Input x value is out of range.");
    }
    return Interpolate(x, mX, mV);
}

double Integral()
{
    return mI[mI.size() - 1];
}

double Integral(double x)
{
    return Integral(mI[0], x);
}

double Integral(double x1, double x2)
{
    if (!InBounds(x1) || !InBounds(x2))
    {
        throw std::out_of_range("Input x value is out of range.");
    }
    return Interpolate(x2, mX, mI) - Interpolate(x1, mI, mI);
}

double Derivative(double x)
{
    if (!InBounds(x))
    {
        throw std::out_of_range("Input x value is out of range.");
    }
    return Interpolate(x, mX, mD);
}

#endif
database.cpp

#include "database.hpp"
#include "part.hpp"
#include "csv.hpp"

using namespace std;

Database::Database(bool random, Settings& settings) : m_random(random), n_settings(&settings)
{
    Load(settings);
    SetT(0);
    SetS(Vector(0, 0, 0));
    SetA(0);
    SetV(0);
}

Database::~Database()
{
    for (auto kvp : m_parts)
    {
        delete kvp.second;
    }
    m_parts.clear();
}

Part* Database::GetPart(std::string name)
{
    try
    {
        return m_parts.at(name);
    }
    catch (out_of_range ex)
    {
    }
throw invalid_argument("Unable to locate specified part in database: "+ name);
}

void Database::Load(Settings& settings)
{
    LoadEnvironment(settings.Get("Environment")->
    LoadParachute(settings.Get("Parachute");
    LoadLaunchRail(settings.Get("LaunchRail");
    LoadThrust(settings.Get("Thrust");
    LoadParts(settings.GetParts());
    InitializeThrustDatabase();
}

double Database::LoadItem(ItemMenu& menu, std::string name)
{
    auto item = menu.Get(name);
    double val = 0.0;
    double err = 0.0;
    try
    {
        val = std::stod(item.Value);
    }
    catch (invalid_argument ex)
    {
        val = 0.0;
    }
    try
    {
        err = std::stod(item.Error);
    }
    catch (invalid_argument ex)
    {
        err = 0.0;
    }
    if (m_random && err > 0)
    {
        return m_settings->GaussianRandom(val, err);
    }
    else
    {
        return val;
    }
}

void Database::LoadEnvironment(ItemMenu& menu)
{
    T = LoadItem(menu, "Temperature");
    Elevation = LoadItem(menu, "Elevation");
    Wind = Vector(
        LoadItem(menu, "Wx");
        LoadItem(menu, "Wy");
        LoadItem(menu, "Wz");
    );
}

void Database::LoadParachute(ItemMenu& menu)
{
    Pap = LoadItem(menu, "Ap");
    Pcd = LoadItem(menu, "Cd");
}

void Database::LoadLaunchRail(ItemMenu& menu)
{
    Lrl = LoadItem(menu, "Length");
    Lrcf = LoadItem(menu, "Cf");
    Lru = Vector(
        LoadItem(menu, "Ux");
        LoadItem(menu, "Uy");
        LoadItem(menu, "Uz");
    ).Normalized();
}

void Database::LoadThrust(ItemMenu& menu)
{
    // Load the time and thrust force data
    DataCsvReader csv(menu.AsString("ThrustFile");
    vector<double> row;
    vector<DynamicsEntry> thrust;
    m_thrust.clear();
    while (csv.NextRow(row))
    {
        m_thrust.push_back(DynamicsEntry(row[0], row[1]));
    }
    for (double t = 0, maxT = (m_thrust.end() - 1); t <= maxT + 0.01; t += .01)
    {
        DynamicsEntry entry;
void InitializeThrustDatabase()
{
    // Calculate the total impulse (used to calculate the remaining mass)
    double totalImpulse = 0;
    double tlast = 0;
    for (auto entry : m_thrust)
    {
        totalImpulse += entry.F * (entry.T - tlast);
        tlast = entry.T;
    }
    // Update the dynamic variables for each thrust entry
    auto fuel = GetPart("Fuel");
    double impulse = 0.0;
    double mfuel = fuel->M;
    tlast = 0.0;
    for (auto& entry : m_thrust)
    {
        // Set the current impulse and mass of the remaining fuel
        impulse += entry.F * (entry.T - tlast);
        fuel->M = mfuel * (1 - impulse / totalImpulse);
        // Calculate the mass and center of mass
        for (auto kvp : m_parts)
        {
            entry.M += kvp.second->M;
            entry.Xcm += kvp.second->M * kvp.second->Xcm();
        }
        entry.Xcm /= entry.M;
        // Calculate the moment of inertia tensor
        for (auto kvp : m_parts)
        {
            double temp1 = kvp.second->Ix(entry.Xcm);
            double temp2 = kvp.second->Iy(entry.Xcm);
            entry.Ix += kvp.second->Iz(entry.Xcm);
            entry.Iy += kvp.second->Iz(entry.Xcm);
            entry.Iz += kvp.second->Iz(0);
        }
        tlast = entry.T;
    }
    Tthrust = m_thrust[m_thrust.size() - 1].T;
    fuel->M = mfuel;
}

/****************************************************************************
* Interpolate two thrust entries for a given time
* t: time of burn for the rocket
* e1: thrust entry below time t
* e2: thrust entry above time t
* a: weight for e1
* b: weight for e2
*****************************************************************************/
void InterpolateDynamics(double t, DynamicsEntry& e)
{
    auto n = m_thrust.size() - 1;
    auto min = m_thrust[0].T;
    auto max = m_thrust[n].T;
    // Make sure specified time is in bounds
    if (t > max || t < min)
    {
        e = (t > max) ? m_thrust[n] : m_thrust[0];
        return;
    }
    // Find the index before and after the specified time
    auto dt = (max - min) / n;
    auto i2 = (size_t)(t / dt);
    i2 = i2 > 0 ? i2 : 1;
    i2 = i2 < n ? i2 : n - 1;
    for (; m_thrust[i2].T > t && i2 > 1; i2--);
    for (; m_thrust[i2].T < t && i2 < n - 1; i2++);
    // Use a linear interpolation before and after the data point
    size_t i1 = i2 - 1;
    InterpolateDynamics(t, entry);
    thrust.push_back(entry);
}

m_thrust = thrust;
auto e1 = &m_thrust[i1];
auto e2 = &m_thrust[i2];
auto b = (t - e1->T) / (e2->T - e1->T);
auto a = 1 - b;
e.T = t;
e.F = a * e1->F + b * e2->F;
if (e.F < 0)
{
e.F = 0;
}
e.M = a * e1->M + b * e2->M;
e.Ix = a * e1->Ix + b * e2->Ix;
e.Iy = a * e1->Iy + b * e2->Iy;
e.Iz = a * e1->Iz + b * e2->Iz;
e.Xcm = a * e1->Xcm + b * e2->Xcm;

void Database::LoadParts(std::map<std::string, ItemMenu>& parts)
{
for (auto kvp : parts)
{
    m_parts[kvp.first] = CreatePart(kvp.second);
}

auto nose = (NosePart*)GetPart("Nose");
Geometry.ln = nose->l;
Geometry.dn = nose->d;
Geometry.nose = nose->GetShape();
Geometry.Xr = Geometry.ln;

auto body = (CylinderPart*)GetPart("Body");
Geometry.lb = body->l;
Geometry.db = body->d;
Geometry.Xc = Geometry.ln + Geometry.lb;

auto fins = (FinSet*)GetPart("Fins");
Geometry.n = fins->n;
Geometry.df = fins->d;
Geometry.lr = fins->r;
Geometry.Lm = fins->Lm;
Geometry.Lt = fins->Lt;
Geometry.ls = fins->ls;
Geometry.lts = fins->lts;
Geometry.Tf = fins->Tf;
Geometry.Xf = fins->Xf;

auto tail = (ConicalFrustrumPart*)GetPart("Tail");
Geometry.du = tail->d;
Geometry.dd = tail->d;
Geometry.lc = tail->l;
Geometry.ltr = Geometry.ln + Geometry.lb + Geometry.lc;
Geometry.Update();
Xrb = Geometry.ltr;
Arb = Geometry.Ar;
}

Part* Database::CreatePart(ItemMenu& menu)
{

auto shape = StringToShape(menu.AsString("Shape"));
auto m = LoadItem(menu, "Mass");
auto x = LoadItem(menu, "Offset");
auto l = LoadItem(menu, "Length");

switch (shape)
{
case Shape::Cylinder:
    return new CylinderPart(m, x, l, LoadItem(menu, "InnerDiameter"),
                            LoadItem(menu, "OuterDiameter"));
case Shape::Cube:
    return new CubePart(m, x, l, LoadItem(menu, "Width"));
case Shape::ConicalFrustrum:
    return new ConicalFrustrumPart(m, x, l, LoadItem(menu, "LowerDiameter"),
                                    LoadItem(menu, "UpperDiameter"), LoadItem(menu, "Thickness"));
case Shape::Trapezoid:
    return new TrapezoidPart(m, x, l, menu.AsInt("Count"),
                              LoadItem(menu, "Thickness"), LoadItem(menu, "BodyDiameter"),
                              LoadItem(menu, "TipChord"), LoadItem(menu, "MidChord"),
                              LoadItem(menu, "Span"), LoadItem(menu, "TotalSpan"));
case Shape::Ogive:
    return new OgiveNosePart(m, x, l, LoadItem(menu, "Diameter"));
case Shape::Cone:
    return new ConicalNosePart(m, x, l, LoadItem(menu, "Diameter"));
case Shape::Parabolic:
    return new ParabolicNosePart(m, x, l, LoadItem(menu, "Diameter"));
default:
    throw invalid_argument("Unexpected shape: " + shape);
}
void Database::SetT(double t)
{
  if (t == m_t)
  {
    return;
  }

  InterpolateDynamics(t, m_dynamics);
  Ft = m_dynamics.F;
  M = m_dynamics.M;
  Xcm = m_dynamics.Xcm;
  I = Matrix::Zero();
  I(0, 0) = m_dynamics.Ix;
  I(1, 1) = m_dynamics.Iy;
  I(2, 2) = m_dynamics.Iz;
  m_t = t;
}

void Database::SetS(const Vector& s)
{
  if (s.Z == m_z)
  {
    return;
  }

  // Update the atmospheric conditions
  const double p0 = 101325;
  const double rho0 = 1.225;
  const double T0 = 288.15;
  const double a0 = 340.294;
  const double g0 = 9.80665;
  const double R = 287.04;
  Tatm = T0 - 6.5 * s.Z / 1000.0;
  P = p0 * pow(1 - 0.0065 * s.Z / Tatm, 5.2561);
  Rho = P / (R * Tatm);
  m_z = s.Z;
}

void Database::SetA(double a)
{
  if (a == m_a)
  {
    return;
  }

  Xcp = Geometry.Xcp(a);
  m_a = a;
}

void Database::SetV(double v)
{
  if (v == m_v)
  {
    return;
  }

  Re = Reynolds(Rho, v, Xrb, Tatm);
  Ma = Mach(v, Tatm);
  Cn = Geometry.Cn(m_a, Re, Ma);
  Ca = Geometry.Ca(m_a, Re, Ma, Cn);
  m_v = v;
}

database.hpp
DynamicsEntry() : T(0), F(0), M(0), Xcm(0), Ix(0), Iy(0), Iz(0) { }
DynamicsEntry(double t, double f) : T(t), F(f), M(0), Xcm(0), Ix(0), Iy(0), Iz(0) { }

class Database
{
private:
    std::map<std::string, Part*> m_parts;
    std::vector<DynamicsEntry> m_thrust;

    double m_t = -1; // Time
    double m_a = -1; // Angle of attack
    double m_v = -1; // Relative velocity
    double m_z = -1; // Height
    bool m_random = false; // True to add random offsets to all values
    DynamicsEntry m_dynamics;
    Settings* m_settings;
    Vector S; // Position

    void Load(Settings& settings);
    void LoadEnvironment(ItemMenu& menu);
    void LoadParachute(ItemMenu& menu);
    void LoadLaunchRail(ItemMenu& menu);
    void LoadThrust(ItemMenu& menu);
    void LoadParts(std::map<std::string, ItemMenu>& parts);
    double LoadItem(ItemMenu& menu, std::string name);
    Part* CreatePart(ItemMenu& menu);
    void InitializeThrustDatabase();
    void InterpolateDynamics(double t, DynamicsEntry& e);

public:
    RocketGeometry Geometry;
    const Vector AxisY0 = Vector(1, 0, 0);
    const Vector AxisP0 = Vector(0, 1, 0);
    const Vector AxisR0 = Vector(0, 0, 1);
    double Gu = 398600.4418e9; // Gravitational constant
    double Rearth = 6.371e6; // Mean radius of the earth
    double Elevation; // Elevation
    double Ft; // Thrust
    double M; // Mass
    double Ms; // Mach number
    double Re; // Reynolds number
    double Cn; // Coefficient of normal drag
    double Ca; // Coefficient of axial drag
    double Xcm; // Center of mass
    double Xcp; // Center of pressure
    double Xrb; // Rocket body length
    double Arb; // Rocket body cross-sectional area
    double Pap; // Parachute planform area
    double Pcd; // Parachute coefficient of drag
    double Lrl; // Launch rail length
    double Lrcf; // Launch rail coefficient of friction
    double Rho; // Atmospheric density
    double Tatm; // Atmospheric temperature
    double Tthrust; // Maximum thrust time
    double P; // Air pressure
    Vector Wind; // Wind speed vector
    Vector Lru; // Launch rail orientation vector
    Matrix I; // Moment of inertia matrix

    Database(bool random, Settings& settings);
    ~Database();
    Part* GetPart(std::string name);
    void SetT(double t);
    void SetS(const Vector& s);
    void SetA(double a);
    void SetV(double v);
};

#endif

dynamics.hpp
-----------------------------------------------------------------------------------------------------------------------
#endif
#define PHYSICS_HPP

enum Shape
{
    Unknown, Cylinder, Tube, Cube, ConicalFrustrum, Trapezoid, Ogive, Cone, Parabloid
};

inline Shape StringToShape(std::string type)
{
    if (type == "Cylinder")
    {
        return Shape::Cylinder;
    }
    else if (type == "Cube")
    {
        return Shape::Cube;
    }
    else if (type == "Ogive")
    {
        return Shape::Ogive;
    }
    else if (type == "Cone")
    {
        return Shape::Cone;
    }
    else if (type == "Parabolid")
    {
        return Shape::Parabolid;
    }
    else if (type == "Trapezoid")
    {
        return Shape::Trapezoid;
    }
    else if (type == "ConicalFrustum")
    {
        return Shape::ConicalFrustum;
    }
    throw std::invalid_argument("Unknown shape: " + type);
}

/******************************************************************************/
/* Applies the parallel axis theorem to a given moment of inertia */
/* 1: original moment of inertia */
/* m: mass of the object */
/* x: position of the axis */
/* xcm: center of mass of the object */
/******************************************************************************/
inline double IxyzParallelAxis(double i, double m, double x, double xcm)
{
    return i + m * sqr(abs(x - xcm));
}

/******************************************************************************/
/* Determines the x and y moment of inertia of a cone */
/* m: mass */
/* r: radius of the base */
/* l: length */
/******************************************************************************/
inline double IxyCone(double m, double r, double l)
{
    return m * ((sqr(l) / 10.0) + (3.0 * sqr(r) / 20.0));
}

/******************************************************************************/
/* Determines the z moment of inertia of a cone */
/* m: mass */
/* r: radius of the base */
/******************************************************************************/
inline double IzCone(double m, double r)
{
    return 3.0 * m * sqr(r) / 10.0;
}

/******************************************************************************/
/* Determines the x moment of inertia of a cube */
/* m: mass */
/* y: y length */
/* z: z length */
/******************************************************************************/
inline double IxCube(double m, double y, double z)
{
    return m * (sqr(y) + sqr(z)) / 12.0;
}

/******************************************************************************/
/* Determines the y moment of inertia of a cube */
/* m: mass */
/* x: x length */
/* z: z length */
/******************************************************************************/
inline double IyCube(double m, double x, double z)
{
    return m * (sqr(x) + sqr(z)) / 12.0;
}

/******************************************************************************/
* Determines the z moment of inertia of a cube
  * m: mass
  * x: x length
  * y: y length
******************************************************************************/
inline double IzCube(double m, double x, double y)
{
  return m * (sqr(x) + sqr(y)) / 12.0;
}
/*****************************************************************************/
* Determines the x and y moment of inertia of a cylinder
  * m: mass
  * r1: inner radius
  * r2: outer radius
  * h: height
******************************************************************************/
inline double IxyCylinder(double m, double r1, double r2, double h)
{
  return (m / 12.0) * (3.0 * (sqr(r2) + sqr(r1)) + sqr(h));
}
/*****************************************************************************/
* Determines the z moment of inertia of a cylinder
  * m: mass
  * r1: inner radius
  * r2: outer radius
******************************************************************************/
inline double IzCylinder(double m, double r1, double r2)
{
  return (m / 2.0) * (sqr(r2) + sqr(r1));
}
/*****************************************************************************/
* Approximates axial force based on the drag and normal force coefficients
  * cd: coefficient of drag force
  * cn: coefficient of normal force
  * a: angle of attack
******************************************************************************/
inline double DragToAxialForce(double cd, double cn, double a)
{
  return cd * cos(a);
}/*double c1 = cos(a);
 double c2 = sin(a);
 double c3 = cd * cos(a);
 double c4 = 0.5 * cn * sin(2 * a);
 double c5 = 1 - sqr(sin(a));
 return (cd * cos(a) - 0.5 * cn * sin(2 * a)) / (1 - sqr(sin(a)));*/
/*****************************************************************************/
* Reynolds number (air approximation)
  * rho: atmospheric density
  * v: apparent velocity vector
  * l: characteristic dimension (total body length or fin mid chord)
  * t: atmospheric temperature
******************************************************************************/
inline double Reynolds(double rho, double v, double l, double t)
{
  const double lambda = 1.512041288e-6; // Sutherland gas constant for air
  const double c = 120.0; // Sutherland’s constant
  double mu = lambda * pow(t, 3.0 / 2.0) / (t + c);
  return rho * v * l / mu;
}
/*****************************************************************************/
* Mach number (for air)
  * v: apparent velocity
  * t: atmospheric temperature
******************************************************************************/
inline double Mach(double v, double t)
{
  const double g = 1.4; // Ratio of specific heats for air
  const double r = 287.0; // Gas constant for air (diff. in specific heats)
  return v / sqrt(g * r * t);
}
/*****************************************************************************/
* Corrects drag coefficients based on mach number
  * c: drag coefficient value
  * m: mach number
******************************************************************************/
inline double CompressibilityCorrection(double c, double m)
{
  if (m < 0.8)
  {
    return c / sqrt(1 - sqr(m));
  }
  else if (m > 1.1)
  {
    return c / sqrt(sqr(m) - 1);
  }
else
{
    return c / sqrt(1 - sqr(0.8));
}

/***************************************************************************/
/* Viscous drag coefficient                                               */
/* re: Reynolds number                                                   */
/***************************************************************************/
inline double ViscousDrag(double re)
{
    const double Re0 = 5e5; // Critical Reynolds number
    double c1 = (1.328 / sqrt(re));
    if (re <= Re0)
    {
        return re >= 1 ? c1 : 0; // TODO: is this a good assumption?
    }
    else
    {
        double c2 = (0.874 / pow(re, 1.0 / 5.0));
        return c2 - (Re0 * (c2 - c1) / re);
    }
}

/***************************************************************************/
/* Polynomial approximation for delta alpha drag component               */
/* a: angle of attack                                                     */
/***************************************************************************/
inline double Delta(double a)
{
    const double c[] = { 0.0001, 23.991, -289.02, 1950.5, -7337.5, 14290.0, -11198.0};
    double result = 0;
    for (int i = 0; i <= 6; i++)
    {
        result += c[i] * pow(a, i);
    }
    return result;
}

/***************************************************************************/
/* Polynomial approximation for eta alpha drag component                  */
/* a: angle of attack                                                     */
/***************************************************************************/
inline double Eta(double a)
{
    const double c[] = { 4.0e-5, 21.168, -293.39, 2111.4, -8871.8, 15624.0, -12860.8};
    double result = 0;
    for (int i = 0; i <= 6; i++)
    {
        result += c[i] * pow(a, i);
    }
    return result;
}

/***************************************************************************/
/* Variables determined by the geometry of a rocket                      */
/***************************************************************************/
struct RocketGeometry
{
    const double K = 1.0;
    Shape nose;
    double Xb;
    double XF;
    double XC;
    double ln;
    double lb;
    double lc;
    double lr;
    double lm;
    double lt;
    double ls;
    double lts;
    double ltr;
    double dn;
    double db;
    double df;
    double du;
    double dd;
    double n;
    double TF;
}
double Ap;
double Ar;
double KCna1;
double KCna2;
double KXcpa1;
double KXcpa2;
void Display();
void Update();

void DragBody (double cf) 
{

double c1 = 1.0;
double c2 = 60.0 / pow(ltr / db, 3);
double c3 = 0.0025 * (lb / db);
double c4 = 2.7 * (ln / db);
double c5 = 4.0 * (lb / db);
double c6 = 2.0 * (1 - (dd / db)) * (lc / db);
}
return (c1 + c2 + c3) * (c4 + c5 + c6) * cf;
}

double DragBase(double cd_fb)
{
    if (cd_fb == 0) return 0;
    double dddb = (du == 0 || dd == 0 || lc == 0) ? 1.0 : dd / db;
    return 0.025 * pow(dddb, 3) / sqrt(cd_fb);
}

double DragFin(double cf)
{
    double afp = Afp();
    double c1 = (1.0 + 2.0 * (Tf / lm));
    double c2 = 4.0 * n * afp / (PI * sqr(df));
    return 2.0 * cf * c1 * c2;
}

double DragInterference(double cf)
{
    double afp = Afp();
    double afp = Afp();
    double c1 = (1.0 + 2.0 * (Tf / lm));
    double c2 = 4.0 * n * (afp - afe) / (PI * sqr(df));
    return 2.0 * cf * c1 * c2;
}

double DragAlphaBody(double a)
{
    double delta = Delta(a);
    double eta = Eta(a);
    double c1 = 2 * delta * sqr(a);
    double c2 = 3.6 * cube(a) * eta * ((1.36 * ltr) - (0.55 * ln)) / (PI * db);
    return c1 + c2;
}

double DragAlphaFin(double a)
{
    double afp4 = Afp() * Afp() * Afp() * Afp();
    double afe4 = Afe() * Afe() * Afe() * Afe();
    double c1 = sqr(a) / (PI * sqr(df));
    double c2 = 1.2 * afp4;
    double c3 = 3.12 * (kfb + kbf - 1.0) * afe4;
    return c1 * (c2 + c3);
}

double Xcp(double a);

double Cn(double a, double r, double m);

double Ca(double a, double r, double m, double cn);

void StabilityConstants(double& kcna1, double& kcna2, double& kxcpa1, double& kxcpa2);
};

inline void RocketGeometry::Update()
{
    Ar = PI * sqr(dn / 2);
    StabilityConstants(KCna1, KCna2, KXcpa1, KXcpa2);
}

inline void RocketGeometry::StabilityConstants(double& kcna1, double& kcna2, double& kxcpa1, double& kxcpa2)
{
    double ncna[] = { NCnaB(), NCnaF(), NCnaN(), NCnaT() };
    double lcna[] = { LCnaB(), LCnaF(), LCnaN(), LCnaT() };
    double nxcp[] = { NXcpB(), NXcpF(), NXcpN(), NXcpT() };
    double lxcp[] = { LXcpB(), LXcpF(), LXcpN(), LXcpT() };
    kcna1 = kxcpa1 = kxcpa2 = 0;
    for (int i = 0; i < 4; i++)
    {
        kcna1 += ncna[i];
        lcna1 += lcna[i];
        nxcp1 += nxcp[i];
        lxcp1 += lxcp[i];
        } kxcpa1 = kxcpa1 = kxcpa2 = 0;
}

**************************************************************************
**** Calculate the center of pressure on the rocket
* a: angle of attack
* cp: center of pressure
**************************************************************************
inline double RocketGeometry::Xcp(double a)
{
    return (KXcpa1 + (KXcpa2 * a)) / (KCna1 + (KCna2 * a));
}

****************************************************************************/
* Calculate the total normal drag coefficient and center of pressure
* g: rocket geometry
**************************************************************************
* a: angle of attack
* r: Reynolds number
* m: mach number
* cn: coefficient of normal drag
* cp: center of pressure

******************************************************************************/
inline double RocketGeometry::Cn(double a, double r, double m)
{
    auto cn = (KCna1 + (KCna2 * a)) * a;
    return CompressibilityCorrection(cn, m);
}

/*************************************************************************************************
* Calculate the total axial drag coefficient
* g: rocket geometry
* a: angle of attack
* r: Reynolds number
* m: mach number
* cn: coefficient of normal drag
* ca: coefficient of axial drag
**********************************************************************************************/
inline double RocketGeometry::Ca(double a, double r, double m, double cn)
{
    double cf = ViscousDrag(r);
    double cdfb = DragBody(cf);
    double cdb = DragBase(cdfb);
    double cdf = DragFin(cf);
    double cdi = DragInterference(cf);
    double cdba = DragAlphaBody(a);
    double cdfa = DragAlphaFin(a);
    double cd = cdfb + cdb + cdf + cdi + cdba + cdfa;
    double ca = DragToAxialForce(cd, cn, a);
    double cca = CompressibilityCorrection(ca, m);
    return cca;
}
inline void RocketGeometry::Display()
{
    double v = 30;
    double t = 273.15 + 20;
    double rho = 1.225;
    double r = Reynolds(rho, v, ltr, t);
    double a = 0.01;
    double m = Mach(v, t);
    double cf = ViscousDrag(r);
    double xcp = Xcp(a);
    double cn = Cn(a, r, m);
    double ca = Ca(a, r, m, cn);
    std::cout << "Rocket Geometry: 
    " Planform Areas
    - Afe: " << Afe() << 
    - Afp: " << Afp() << 
    Stability Derivatives
    - CnaN: " << NCnaN() << 
    - CnaB: " << NCnaB() << 
    - CnaT: " << NCnaT() << 
    - CnaF: " << NCnaF() << 
    Lift Force CP
    - NcpN: " << NXcpN() << 
    - NcpB: " << NXcpB() << 
    - NcpT: " << NXcpT() << 
    - NcpF: " << NXcpF() << 
    Drag Force
    - DragB: " << DragBody(cf) << 
    - DragA: " << DragBase(DragBody(cf)) << 
    - DragF: " << DragFin(cf) << 
    - DragI: " << DragInterference(cf) << 
    - DragBA: " << DragAlphaBody(a) << 
    - DragFA: " << DragAlphaFin(a) << 
    Results
    - Cn: " << cn << 
    - Ca: " << ca << 
    - Xcp: " << xcp << std::endl;
}
#endif

main.cpp

#include "simulator.hpp"
#include "settings.hpp"
```cpp
#include "database.hpp"
#include <iostream>
#include <string>
using namespace std;
#include <chrono>
#include <random>
#include <sstream>
#include <cstdlib>

void ConsoleMode();
void DisplayHelp();

int main(int argc, char** argv)
{
    if (argc < 3)
    {
        ConsoleMode();
        return 0;
    }
    for (int i = 1; i < argc; i++)
    {
        cout << argv[i] << endl;
    }
    Simulator simulator(argv[2]);
    if (string(argv[1]) == "mc")
    {
        simulator.RunMonteCarlo(atoi(argv[3]));
    }
    else if (string(argv[1]) == "run")
    {
        SimulationResult result;
        simulator.Run(true, false, 0.01, 0.1, result);
    }
    return 0;
}

void ConsoleMode()
{
    cout << "BYU - I Rocket Simulator by Jacob Hales" << endl << endl;
    DisplayHelp();
    string file = "..
    Simulation
    Simple Rocket (0).xml";
    while (true)
    {
        string line;
        getline(cin, line);
        if (line == "clear")
        {
            system("cls");
        }
        else if (line == "db")
        {
            //db.ExportDragProfile("C:\\Temp\\DragProfile.csv");
            //db.ExportAerodynamics("C:\\Temp\\Aerodynamics.csv");
            //db.ExportDynamics("C:\\Temp\\Dynamics.csv");
        }
        else if (line == "help")
        {
            DisplayHelp();
        }
        else if (line == "monte")
        {
            Simulator simulator(file);
            simulator.RunMonteCarlo(1000);
        }
        else if (line == "quit")
        {
            return;
        }
        else if (line == "test")
        {
        }
        else if (line == "run")
        {
            Simulator simulator(file);
            SimulationResult result;
            simulator.Run(true, false, 0.0001, 0.1, result);
        }
        else
        {
            cout << "Unrecognized command: " << line << endl;
        }
    }
}
```
void DisplayHelp()
{
    cout << "Available commands:" << endl << endl
        << "db: output the contents of the database" << endl
        << "clear: clears the display" << endl
        << "help: displays this help menu" << endl
        << "quit: quits the program" << endl
        << "run: runs the current simulation" << endl
        << "test: runs a predefined test" << endl;
}

math.hpp

#ifndef MATH_HPP
#define MATH_HPP

class Vector;
class Matrix;
class Quaternion;

class Matrix
{
    private:
        double mA[9];
        int ToIndex(int row, int col) const
        {
            return row * 3 + col;
        }

    public:

        Matrix()
        {
        }

        Matrix(const Matrix& other)
        {
            (*this) = other;
        }

        Matrix DiagonalInverse() const
        {
            Matrix m = Zero();
            m(0, 0) = 1.0 / mA[ToIndex(0, 0)];
            m(1, 1) = 1.0 / mA[ToIndex(1, 1)];
            m(2, 2) = 1.0 / mA[ToIndex(2, 2)];
            return m;
        }

        Matrix Transpose() const
        {
            Matrix m;
            for (int row = 0; row < 3; row++)
            {
                for (int col = 0; col < 3; col++)
                {
                    m.mA[ToIndex(col, row)] = mA[ToIndex(row, col)];
                }
            }
            return m;
        }

        static Matrix Zero()
        {
            Matrix m;
            memset(m.mA, 0, sizeof(double) * 9);
            return m;
        }

        double& operator()(const int row, const int col)
        {
            return mA[ToIndex(row, col)];
        }

        Matrix & operator-(const Matrix &rhs)
        {
            for (int i = 0; i < 9; i++)
                mA[i] = rhs.mA[i];
            return *this;
        }

        Matrix & operator+(const Matrix &rhs)
        {
            for (int i = 0; i < 9; i++)
                mA[i] += rhs.mA[i];
            return *this;
        }

        Matrix & operator=(const Matrix &rhs)
        {
            return *this;
        }

        Matrix & operator=(const Matrix& rhs)
        {
            for (int i = 0; i < 9; i++)
                mA[i] = rhs.mA[i];
            return *this;
        }

    }
{ for (int i = 0; i < 9; mA[i++] = rhs.mA[i]);
  return *this;
}

Matrix & operator*=(const Matrix &rhs)
{
  Matrix m = Zero();
  for (int row = 0; row < 3; row++)
  {
    for (int col = 0; col < 3; col++)
    {
      for (int k = 0, i = ToIndex(row, col); k < 3; k++)
        { m.mA[i] = mA[ToIndex(row, k)] * rhs.mA[ToIndex(k, col)];
        }
    }
  }
  (*this) = m;
  return *this;
}

Matrix & operator*=(const double &rhs)
{
  for (int i = 0; i < 9; mA[i++] *= rhs);
  return *this;
}

Matrix & operator/=(const double &rhs)
{
  for (int i = 0; i < 9; mA[i++] /= rhs);
  return *this;
}

const Matrix operator+(const Matrix &other) const
{
  return Matrix(*this) += other;
}

const Matrix operator-(const Matrix &other) const
{
  return Matrix(*this) -= other;
}

const Matrix operator*(const Matrix &other) const
{
  return Matrix(*this) *= other;
}

const Matrix operator*(const double &other) const
{
  return Matrix(*this) *= other;
}

const Matrix operator/(const double &other) const
{
  return Matrix(*this) /= other;
}

};

class Vector
{
 public:
  double X;
  double Y;
  double Z;

  Vector(const Vector &v) : X(v.X), Y(v.Y), Z(v.Z) { }
  Vector(double x, double y, double z) : X(x), Y(y), Z(z) { }

  double Norm() const
  {
    return sqrt(X * X + Y * Y + Z * Z);
  }

  void Normalize()
  {
    *this /= Norm();
  }

  Vector Normalized()
  {
    return *this / Norm();
  }

  double Dot(const Vector v) const
  {
Vector operator-(const Vector &rhs)
{
    X = rhs.X;
    Y = rhs.Y;
    Z = rhs.Z;
    return *this;
}
Vector & operator+=(const Vector &rhs)
{
    X += rhs.X;
    Y += rhs.Y;
    Z += rhs.Z;
    return *this;
}
Vector & operator-=(const Vector &rhs)
{
    X -= rhs.X;
    Y -= rhs.Y;
    Z -= rhs.Z;
    return *this;
}
Vector & operator*=(const double &rhs)
{
    X *= rhs;
    Y *= rhs;
    Z *= rhs;
    return *this;
}
Vector & operator/=(const double &rhs)
{
    X /= rhs;
    Y /= rhs;
    Z /= rhs;
    return *this;
}
const Vector operator+(const Vector &other) const
{
    return Vector(*this) += other;
}
const Vector operator-(const Vector &other) const
{
    return Vector(*this) -= other;
}
const Vector operator*(const double &other) const
{
    return Vector(*this) *= other;
}
const Vector operator/(const double &other) const
{
    return Vector(*this) /= other;
}
friend Vector operator*(double d, const Vector v)
{
    return Vector(v) *= d;
}
friend Vector operator*(Matrix m, Vector v)
{
    return Vector(
        m(0, 0) * v.X + m(0, 1) * v.Y + m(0, 2) * v.Z,
        m(1, 0) * v.X + m(1, 1) * v.Y + m(1, 2) * v.Z,
        m(2, 0) * v.X + m(2, 1) * v.Y + m(2, 2) * v.Z);
};
class Quaternion
{
    public:
    double W;
double X;
double Y;
double Z;

Quaternion()
{
}

Quaternion(double w, double x, double y, double z) : W(w), X(x), Y(y), Z(z)
{
}

double Norm()
{
    return sqrt(W * W + X * X + Y * Y + Z * Z);
}

void normalize()
{
    double n = Norm();
    W /= n;
    X /= n;
    Y /= n;
    Z /= n;
}

Matrix toRotationMatrix() const
{
    Matrix m;
    double s = sin(W);
    double X2 = sqr(X);
    double Y2 = sqr(Y);
    double Z2 = sqr(Z);
    m(0, 0) = 1 - 2 * (Y2 - Z2);
    m(0, 1) = 2 * (X * Y - W * Z);
    m(0, 2) = 2 * (X * Z + W * Y);
    m(1, 0) = 2 * (X * Y + W * Z);
    m(1, 1) = 1 - 2 * (X2 - Z2);
    m(1, 2) = 2 * (Y * Z - W * X);
    m(2, 0) = 2 * (X * Z - W * Y);
    m(2, 1) = 2 * (Y * Z + W * X);
    m(2, 2) = 1 - 2 * (X2 - Y2);
    return m;
}

Vector vec() const
{
    return Vector(X, Y, Z);
}

};

#endif

---

#include "model.hpp"
#include "common.hpp"

using namespace std;

ModelState& ModelState::operator=(const ModelState& rhs)
{
    if (this != &rhs)
    {
        T = rhs.T;
        S = rhs.S;
        P = rhs.P;
        L = rhs.L;
        Q = Quaternion(
            Q.W + rhs.Q.W,
            Q.X + rhs.Q.X,
            Q.Y + rhs.Q.Y,
            Q.Z + rhs.Q.Z);
        return *this;
    }
}

const ModelState ModelState::operator+(const ModelState& other) const
{
    return ModelState(*this) += other;
}

ModelState& ModelState::operator+=(const ModelState& rhs)
{
    S += rhs.S;
    P += rhs.P;
    L += rhs.L;
    Q = Quaternion(
        Q.W + rhs.Q.W,
        Q.X + rhs.Q.X,
        Q.Y + rhs.Q.Y,
        Q.Z + rhs.Q.Z);
    return *this;
}
const ModelState ModelState::operator-(const ModelState& other) const
{
    return ModelState(*this) -= other;
}

ModelState& ModelState::operator-=(const ModelState& rhs)
{
    S -= rhs.S;
    P -= rhs.P;
    L -= rhs.L;
    Q = Quaternion(
        Q.W - rhs.Q.W,
        Q.X - rhs.Q.X,
        Q.Y - rhs.Q.Y,
        Q.Z - rhs.Q.Z);
    return *this;
}

const ModelState ModelState::operator*(const ModelState& other) const
{
    return ModelState(*this) *= other;
}

ModelState& ModelState::operator*=(const ModelState& rhs)
{
    S = Vector(S.X * rhs.S.X, S.Y * rhs.S.Y, S.Z * rhs.S.Z);
    P = Vector(P.X * rhs.P.X, P.Y * rhs.P.Y, P.Z * rhs.P.Z);
    L = Vector(L.X * rhs.L.X, L.Y * rhs.L.Y, L.Z * rhs.L.Z);
    Q = Quaternion(
        Q.W * rhs.Q.W,
        Q.X * rhs.Q.X,
        Q.Y * rhs.Q.Y,
        Q.Z * rhs.Q.Z);
    return *this;
}

const ModelState ModelState::operator*(double n) const
{
    return ModelState(*this) *= n;
}

ModelState& ModelState::operator*=(double n)
{
    S *= n;
    P *= n;
    L *= n;
    Q = Quaternion(Q.W * n, Q.X * n, Q.Y * n, Q.Z * n);
    return *this;
}

std::ostream& operator<<(std::ostream &out, ModelState& state)
{
    out << state.T << ", "
        << state.S.X << ", "
        << state.S.Y << ", "
        << state.S.Z << ", "
        << state.P.X << ", "
        << state.P.Y << ", "
        << state.P.Z << ", "
        << state.L.X << ", "
        << state.L.Y << ", "
        << state.L.Z << ", "
        << state.Q.W << ", "
        << state.Q.X << ", "
        << state.Q.Y << ", "
        << state.Q.Z << std::endl;
    return out;
}

void Model::Integrate(double dt,ModelState& state, double recordInterval, std::vector<ModelState>& history)
{
    for (double dt2 = dt / 2.0, nextRecord = recordInterval; !IsFinished(state.T, state); state.T += dt)
    {
        auto k1 = Differentiate(state.T, state);
        auto k2 = Differentiate(state.T + dt2, state + k1 * dt2);
        auto k3 = Differentiate(state.T + dt2, state + k2 * dt2);
        auto k4 = Differentiate(state.T + dt, state + k3 + 2.0 + k4) * (dt / 6.0);
        state += (k1 + k2 * 2.0 + k3 * 2.0 + k4) * (dt / 6.0);
        state.Q.normalize();
        if (state.T >= nextRecord)
        {
            history.push_back(state);
            nextRecord += recordInterval;
        }
    }
    if (history.size() > 0 && history.back().T < state.T)
    {
        history.push_back(state);
    }
bool LaunchModel::IsFinished(double t, const ModelState& state)
{
    return t > 0 && ((state.S.Norm() > m_db->lrl) || (state.T > m_db->Thrust));
}

ModelState LaunchModel::Differentiate(double t, const ModelState& state)
{
    m_db->SetT(state.T);
    m_db->SetS(state.S);
    m_db->SetA(0);
    auto v = state.P + m_db->Wind;
    auto vnorm = v.Norm();
    auto u = m_db->u;
    m_db->SetV(vnorm);
    auto drag = 0.5 * m_db->Rho * sqr(vnorm) * m_db->Arb;
    auto cfn = drag * m_db->Cn;
    auto cfa = drag * m_db->Ca;
    auto cft = m_db->Ft;
    auto cfg = m_db->Gu * m_db->M / sqr(m_db->Rearth + state.S.Z);
    auto fn = Vector::Zero();
    auto fa = -cfa * u;
    auto ft = cft * u;
    auto fg = -cfg * u;
    auto fd = Vector::Zero();
    ModelState dstate;
    dstate.T = state.T;
    dstate.S = state.P;
    auto r = state.Q.toRotationMatrix();
    auto w = state.L;
    auto wnorm = w.Norm();
    auto what = (wnorm > 0) ? (Vector)(w / wnorm) : u;
    auto v = Vector(0, 0, 0);
    auto vnorm = (double)0.0;
    auto vhat = Vector(0, 0, 0);
    auto vcm = state.P + m_db->Wind;
    auto xbar = 0.0;
    auto tw = sin(acos(u.Dot(what))) * u.Cross(w);
    for (double a = 0, aold = INFINITY; abs(a - aold) > 0.001; aold = a)
    {
        v = vcm + xbar * tw;
        vnorm = v.Norm();
        what = (wnorm > 0) ? (Vector)(v / vnorm) : u;
        a = BoundedAcos(what.Dot(u));
        m_db->SetA(a);
        xbar = abs(m_db->Xcp - m_db->Xcm);
    }
    m_db->SetV(vnorm);
    auto drag = 0.5 * m_db->Rho * sqr(vnorm) * m_db->Arb;
    auto cfn = drag * m_db->Cn;
    auto cfa = drag * m_db->Ca;
    auto cft = m_db->Ft;
    auto cfg = m_db->Gu * m_db->M / sqr(m_db->Rearth + state.S.Z);
    Vector fn = cfn * u.Cross(u.Cross(what));
    auto fa = -cfa * u;
    auto ft = cft * u;
    return dstate;
}

bool AscentModel::IsFinished(double t, const ModelState& state)
{
    return state.P.Z < 0;
}

ModelState AscentModel::Differentiate(double t, const ModelState& state)
{
    m_db->SetT(state.T);
    m_db->SetS(state.S);
    m_db->SetA(0);
    auto r = state.Q.toRotationMatrix();
    auto w = state.L;
    auto wnorm = w.Norm();
    auto what = (wnorm > 0) ? (Vector)(w / wnorm) : u;
    auto v = Vector(0, 0, 0);
    auto vnorm = (double)0.0;
    auto vhat = Vector(0, 0, 0);
    auto vcm = state.P + m_db->Wind;
    auto xbar = 0.0;
    auto tw = sin(acos(u.Dot(what))) * u.Cross(w);
    for (double a = 0, aold = INFINITY; abs(a - aold) > 0.001; aold = a)
    {
        v = vcm + xbar * tw;
        vnorm = v.Norm();
        what = (wnorm > 0) ? (Vector)(v / vnorm) : u;
        a = BoundedAcos(what.Dot(u));
        m_db->SetA(a);
        xbar = abs(m_db->Xcp - m_db->Xcm);
    }
    m_db->SetV(vnorm);
    auto drag = 0.5 * m_db->Rho * sqr(vnorm) * m_db->Arb;
    auto cfn = drag * m_db->Cn;
    auto cfa = drag * m_db->Ca;
    auto cft = m_db->Ft;
    auto cfg = m_db->Gu * m_db->M / sqr(m_db->Rearth + state.S.Z);
    Vector fn = cfn * u.Cross(u.Cross(what));
    auto fa = -cfa * u;
    auto ft = cft * u;
    return dstate;
}
auto fg = -cfg * m_db->AxisR0;
auto tn = cfn * xbar * u.Cross(vhat);
auto sdot = 0.5 * w.Dot(state.Q.vec());
auto vdot = 0.5 * (state.Q.W * w + w.Cross(state.Q.vec()));
ModelState dstate;
dstate.T = state.T;
dstate.S = state.P;
dstate.Q = Quaternion(sdot, vdot.X, vdot.Y, vdot.Z);
dstate.P = (fn + fa + ft + fg) / m_db->M;
dstate.L = r * m_db->I.DiagonalInverse() * r.Transpose() * tn;
return dstate;
}

bool ParachuteModel::IsFinished(double t, const ModelState& state) {
    return state.S.Z <= 0;
}

ModelState ParachuteModel::Differentiate(double t, const ModelState& state) {
    m_db->SetT(state.T);
    m_db->SetS(state.S);
    m_db->SetA(0);
    auto v = state.P + m_db->Wind;
    auto vnrm = v.Norm();
    auto vhat = v / vnrm;
    auto fd = -0.5 * m_db->Rho * sqrt(vnrm) * m_db->Pcd * m_db->Pap * vhat;
    auto fg = -m_db->Gu * m_db->M / sqrt(m_db->Rearth + state.S.Z) * m_db->AxisR0;

    ModelState dstate;
    dstate.T = state.T;
    dstate.S = state.P;
    dstate.Q = Quaternion(1, 0, 0, 0);
    dstate.P = (fd + fg) / m_db->M;
    dstate.L = Vector(0, 0, 0);
    return dstate;
}

std::vector<Output> Model::GetOutput(const vector<ModelState>& state) {
    vector<Output> output;
    for (ModelState m : state) {
        output.push_back(GetOutput(m.T, m));
    }
    return output;
}

Output Model::GetOutput(double t, const ModelState& state) {
    Output o;
    n_db->SetT(state.T);
    n_db->SetS(state.S);
    auto r = state.Q.toRotationMatrix();
    auto u = r * m_db->AxisR0;
    auto w = r * n_db->I.DiagonalInverse() * r.Transpose() * state.L;
    auto vnrm = w.Norm();
    auto what = (vnrm > 0) ? (Vector)(w / vnrm) : u;
    auto v = Vector(0, 0, 0);
    auto vnrm = (double)0.0;
    auto what = Vector(0, 0, 0);
    auto vcm = state.P + m_db->Wind;
    auto xbar = 0.0;
    auto tw = sin(acos(u.Dot(what))) * u.Cross(w);
    double a = 0;

    for (double aold = INFINITY; abs(a - aold) > 0.001; aold = a) {
        v = vcm + xbar * tw;
        vnrm = v.Norm();
        what = (vnrm > 0) ? (Vector)(v / vnrm) : u;
        a = BoundedAcos(what.Dot(u));
        m_db->SetA(a);
        xbar = abs(m_db->Xcp - m_db->Xcm);
    }
    n_db->SetV(vnrm);
    auto drag = 0.5 * m_db->Rho * sqrt(vnrm) * m_db->Arb;
    auto cfn = drag * m_db->Cn;
auto cf = drag * m_db->Ca;
auto ct = m_db->Ft;
auto cg = m_db->Gu * m_db->M / sqrt(m_db->Rearth + state.S.Z);

Vector fn = cfn * u.Cross(u.Cross(vhat));
auto fa = -cf * a;
auto ft = cft * u;
auto fg = -cfg * m_db->AxisR0;

auto tn = cfn * xbar * u.Cross(vhat);
auto sdot = 0.5 * w.Dot(state.Q.vec());
auto vdot = 0.5 * (state.Q.W * w + w.Cross(state.Q.vec()));

ModelState dstate;
dstate.T = state.T;
dstate.S = state.P;
dstate.Q = Quaternion(sdot, vdot.X, vdot.Y, vdot.Z);
dstate.P = (fn + fa + ft + fg) / m_db->M;
dstate.L = tn;

ModelState o;
  o.Time = t;
  o.Altitude = state.S.Z;
  o.VerticalVelocity = state.P.Z;
  o.VerticalAcceleration = state.P.Norm();
  o.TotalVelocity = state.P.Norm();
  o.PositionUpwind = state.S.X;
  o.PositionParallelToWind = state.S.Y;
  o.LateralDistance = Vector(state.S.X, state.S.Y, 0).Norm();
  o.LateralDirection = 0;
  o.LateralVelocity = Vector(state.P.X, state.P.Y, 0).Norm();
  o.LateraAcceleration = Vector(state.P.X, state.P.Y, 0).Norm();
  o.Latitude = 0;
  o.Longitude = 0;
  o.GravitationalAcceleration = m_db->Gu / sqrt(m_db->Rearth + state.S.Z);
  o.AngleOfAttack = a;
  o.RollRate = w.X;
  o.YawRate = w.Y;
  o.PitchRate = w.Z;
  o.Mass = m_db->M;
  o.PropellantMass = 0;
  o.LongitudalMomentOfInertia = m_db->I(0, 0);
  o.RotationalMomentOfInertia = m_db->I(2, 2);
  o.PLocation = m_db->Xcp;
  o.CGLocation = m_db->Xcm;
  o.StabilityMarginCalibers = 0;
  o.MachNumber = m_db->Ma;
  o.ReynoldsNumber = m_db->Re;
  o.Thrust = m_db->Ft;
  o.DragForce = (fn + fa).Norm();
  o.AxialDragCoefficient = m_db->Ca;
  o.NormalForceCoefficient = m_db->Cn;
  o.ReferenceArea = m_db->Geometry.Ap;
  o.ReferenceLength = m_db->Geometry.ltr;
  o.AirTemperature = m_db->Tatm;
  o.AirPressure = m_db->P;
  o.Altitude = state.S.Z;
  o.WindVelocity = m_db->Wind.Norm();

return o;
}

model.hpp

#endif MODEL_HPP
#define MODEL_HPP

#include <vector>
#include <iostream>
#include <string>
#include <sstream>
#include "database.hpp"
#include "database.hpp"
#include "math.hpp"

class Output;

class ModelState {
public:
  double T;
  Vector S;
  Vector P;
  Vector L;
  Quaternion Q;

  ModelState& operator=(const ModelState& rhs);
  const ModelState operator+(const ModelState& other) const;
  ModelState& operator+=(const ModelState& rhs);
  const ModelState operator-(const ModelState& other) const;
  ModelState& operator-=(const ModelState& rhs);
  ModelState& operator-=(const ModelState& rhs);

  ModelState(){}
};
const ModelState operator*(const ModelState& other) const;
ModelState& operator*(const ModelState& rhs);
const ModelState operator*(double n) const;
ModelState& operator=(double n);
friend std::ostream& operator<<(std::ostream &out, ModelState& state);
ModelState() : T(0), S(0, 0, 0), P(0, 0, 0), L(0, 0, 0), Q(0, 0, 0, 0)
{ }
};
class Model
{
protected:
  Database* m_db;
public:
  Model(Database* db) : m_db(db) { }
  virtual bool IsFinished(double t, const ModelState& state) = 0;
  virtual ModelState Differentiate(double t, const ModelState& state) = 0;
  Output GetOutput(double t, const ModelState& state);
};
class LaunchModel : public Model
{
public:
  LaunchModel(Database* db) : Model(db) { }
  bool IsFinished(double t, const ModelState& state);
  ModelState Differentiate(double t, const ModelState& state);
};
class AscentModel : public Model
{
public:
  AscentModel(Database* db) : Model(db) { }
  bool IsFinished(double t, const ModelState& state);
  ModelState Differentiate(double t, const ModelState& state);
};
class ParachuteModel : public Model
{
public:
  ParachuteModel(Database* db) : Model(db) { }
  bool IsFinished(double t, const ModelState& state);
  ModelState Differentiate(double t, const ModelState& state);
};
class Output
{
public:
  double Time;
  double Altitude;
  double VerticalVelocity;
  double VerticalAcceleration;
  double TotalVelocity;
  double TotalAcceleration;
  double PositionUpwind;
  double PositionParallelToWind;
  double LateralDistance;
  double LateralDirection;
  double LateralVelocity;
  double LateralAcceleration;
  double Latitude;
  double Longitude;
  double GravitationalAcceleration;
  double AngleOfAttack;
  double RollRate;
  double YawRate;
  double PitchRate;
  double Mass;
  double PropellantMass;
  double LongitudinalMomentOfInertia;
  double RotationalMomentOfInertia;
  double CPLocation;
  double CGLocation;
  double StabilityMarginCalibers;
  double MachNumber;
  double ReynoldsNumber;
  double Thrust;
  double DragForce;
  double DragCoefficient;
  double AxialDragCoefficient;
  double FrictionDragCoefficient;
double PressureDragCoefficient;
double BaseDragCoefficient;
double NormalForceCoefficient;
double PitchMomentCoefficient;
double YawMomentCoefficient;
double SideForceCoefficient;
double RollMomentCoefficient;
double RollForcingCoefficient;
double RollDampingCoefficient;
double PitchDampingCoefficient;
double ReferenceLength;
double ReferenceArea;
double VerticalOrientationZenith;
double LateralOrientationAzimuth;
double WindVelocity;
double AirTemperature;
double AirPressure;
double SpeedOfSound;

Output()
{
    Time = 0;
    Altitude = 0;
    VerticalVelocity = 0;
    TotalVelocity = 0;
    TotalAcceleration = 0;
    PositionUpwind = 0;
    PositionParallelToWind = 0;
    LateralDistance = 0;
    LateralDirection = 0;
    LateralVelocity = 0;
    LateralAcceleration = 0;
    Latitude = 0;
    Longitude = 0;
    GravitationalAcceleration = 0;
    AngleOfAttack = 0;
    RollRate = 0;
    YawRate = 0;
    PitchRate = 0;
    Mass = 0;
    PropellantMass = 0;
    LongitudinalMomentOfInertia = 0;
    RotationalMomentOfInertia = 0;
    CPOrientation = 0;
    CGLocation = 0;
    StabilityMarginCalibers = 0;
    MachNumber = 0;
    ReynoldsNumber = 0;
    Thrust = 0;
    DragForce = 0;
    DragCoefficient = 0;
    AxialDragCoefficient = 0;
    FrictionDragCoefficient = 0;
    PressureDragCoefficient = 0;
    BaseDragCoefficient = 0;
    NormalForceCoefficient = 0;
    YawMomentCoefficient = 0;
    SideForceCoefficient = 0;
    RollMomentCoefficient = 0;
    RollForcingCoefficient = 0;
    RollDampingCoefficient = 0;
    PitchDampingCoefficient = 0;
    ReferenceLength = 0;
    ReferenceArea = 0;
    VerticalOrientationZenith = 0;
    LateralOrientationAzimuth = 0;
    WindVelocity = 0;
    AirTemperature = 0;
    AirPressure = 0;
    SpeedOfSound = 0;
}

std::string Header()
{
    std::stringstream ss;
    ss << "Time, ";
    ss << "Altitude, ";
    ss << "VerticalVelocity, ";
    ss << "VerticalAcceleration, ";
    ss << "TotalVelocity, ";
    ss << "TotalAcceleration, ";
    ss << "PositionUpwind, ";
    ss << "PositionParallelToWind, ";
    ss << "LateralDistance, ";
    ss << "LateralDirection, ";
    ss << "LateralVelocity, ";
    ss << "LateralAcceleration, ";
    ss << "Latitude, ";
    ss << "Longitude, ";
    return ss.str();
}
GravitationalAcceleration, AngleOfAttack, RollRate, YawRate, PitchRate, Mass, PropellantMass, LongitudinalMomentOfInertia, RotationalMomentOfInertia, CPLocation, CGLocation, StabilityMarginCalibers, MachNumber, ReynoldsNumber, Thrust, DragForce, DragCoefficient, AxialDragCoefficient, FrictionDragCoefficient, PressureDragCoefficient, BaseDragCoefficient, NormalForceCoefficient, PitchMomentCoefficient, YawMomentCoefficient, SideForceCoefficient, RollMomentCoefficient, RollForcingCoefficient, RollDampingCoefficient, PitchDampingCoefficient, ReferenceLength, ReferenceArea, VerticalOrientationZenith, LateralOrientationAzimuth, WindVelocity, AirTemperature, AirPressure, SpeedOfSound

return ss.str();
}

std::string Row()
{
  std::stringstream ss;
  ss << Time << ", ",
      Altitude << ", ",
      VerticalVelocity << ", ",
      VerticalAcceleration << ", ",
      TotalVelocity << ", ",
      TotalAcceleration << ", ",
      PositionUpwind << ", ",
      PositionParallelToWind << ", ",
      LateralDistance << ", ",
      LateralDirection << ", ",
      LateralVelocity << ", ",
      LateralAcceleration << ", ",
      Latitude << ", ",
      Longitude << ", ",
      GravitationalAcceleration << ", ",
      AngleOfAttack << ", ",
      RollRate << ", ",
      YawRate << ", ",
      PitchRate << ", ",
      Mass << ", ",
      PropellantMass << ", ",
      LongitudinalMomentOfInertia << ", ",
      RotationalMomentOfInertia << ", ",
      CPLocation << ", ",
      CGLocation << ", ",
      StabilityMarginCalibers << ", ",
      MachNumber << ", ",
      ReynoldsNumber << ", ",
      Thrust << ", ",
      DragForce << ", ",
      DragCoefficient << ", ",
      AxialDragCoefficient << ", ",
      FrictionDragCoefficient << ", ",
      PressureDragCoefficient << ", ",
      BaseDragCoefficient << ", ",
      NormalForceCoefficient << ", ",
      PitchMomentCoefficient << ", ",
      YawMomentCoefficient << ", ",
      SideForceCoefficient << ", ",
      RollMomentCoefficient << ", ",
      RollForcingCoefficient << ", ",
      RollDampingCoefficient << ", ",
      PitchDampingCoefficient << ", ",
      ReferenceLength << ", ",
      ReferenceArea << ", ",
      VerticalOrientationZenith << ", ",
      LateralOrientationAzimuth << ", 

  return ss.str();
}
<< WindVelocity << "," << AirTemperature << "," << AirPressure << "," << SpeedOfSound << std::endl;
return ss.str();
}
double Iy() { return IxyCylinder(M, R1(), R2(), L); }
double Iz() { return IzCylinder(M, R1(), R2()); }
double R1() { return D1 / 2.0; }
double R2() { return D2 / 2.0; }

ConicalFrustrumPart(double m, double x, double l, double d1, double d2, double w) : Part(m, x, l), W(w), D1(d1), D2(d2) {
};

/****************************************************************************
* Represents the nose of a rocket
******************************************************************************/
class NosePart : public Part
{
public:
    double D;
    double Xcm() { return (2.0 / 3.0) * L; }
    double Ix() { return IxyCone(M, R(), L); }
    double Iy() { return IxyCone(M, R(), L); }
    double I1() { return D / 2.0; }
    virtual Shape GetShape() = 0;

    NosePart(double m, double x, double l, double d) : Part(m, x, l), D(d) {
    }
};

/**************************************************************************
* Conical nose
******************************************************************************/
class ConicalNosePart : public NosePart
{
public:
    Shape GetShape() { return Shape::Cone; }

    ConicalNosePart(double m, double x, double l, double d) : NosePart(m, x, l, d)
    {
    }
};

/**************************************************************************
* Ogive nose
******************************************************************************/
class OgiveNosePart : public NosePart
{
public:
    Shape GetShape() { return Shape::Ogive; }

    OgiveNosePart(double m, double x, double l, double d) : NosePart(m, x, l, d)
    {
    }
};

/**************************************************************************
* Parabolic nose
******************************************************************************/
class ParabolicNosePart : public NosePart
{
public:
    Shape GetShape() { return Shape::Parabloid; }

    ParabolicNosePart(double m, double x, double l, double d) : NosePart(m, x, l, d)
    {
    }
};

/**************************************************************************
* Represents the fins of a rocket
******************************************************************************/
class FinSet : public Part
{
public:
    int N;
    double W, DF, LT, LN, LS, LTS;
    double Ix() { return IxCube(M, W, L); }
    double Iy() { return IyCube(M, W, L); }
    double Iz() { return IzCube(M, W, W); }

    FinSet(double m, double x, double l, int n, double w, double df, double lt, double ln, double ls, double lts) : Part(m, x, l), N(n), W(w), DF(df), LT(lt), LN(ln), LS(ls), LTS(lts)
    {
    }
};
#ifndef settings_hpp
#endif

#include "settings.hpp"
#include "pugixml.hpp"
#include <iostream>
#include <chrono>

using namespace std;
using namespace pugi;

double AttributeToDouble(const xml_node& node, const string& name)
{
    try
    {
        return stod(node.attribute(name.c_str()).value());
    }
    catch (invalid_argument)
    {
        return 0.0;
    }
}

Settings::Settings(string file)
    : m_generator((int)chrono::system_clock::now().time_since_epoch().count())
{
    Load(file);
}

void Settings::Load(string file)
{
    xml_document doc;
    auto result = doc.load_file(file.c_str());
    if (result == 0)
        throw exception(result.description());

    for (auto child : doc.child("Simulation").children("Menu"))
    {
        ParseItemMenu(child);
    }
}

void Settings::ParseItemMenu(const xml_node& node)
{
    ItemMenu menu;
    menu.Name = node.attribute("Name").value();
    menu.Description = node.attribute("Description").value();
    menu.Type = node.attribute("Type").value();

    for (auto child : node.children("Item"))
    {
        ItemElement item;
        item.Name = child.attribute("Name").value();
        item.Value = child.attribute("Value").value();
        item.Error = child.attribute("Error").value();
        item.Description = child.attribute("Description").value();
        menu.Items[item.Name] = item;
    }
    m_menus[menu.Name] = menu;
}

ItemMenu::ItemMenu::Get(std::string item)
{
    try
    {
        return Items.at(item);
    }
    catch (std::out_of_range ex)
    {
        throw std::out_of_range("Unable to locate item " + item + " in menu " + Name);
    }
}

std::string ItemMenu::AsString(std::string item)
{
    return Get(item).Value;
}

double ItemMenu::AsDouble(std::string item)
{
    try
    {
        return stod(AsString(item));
    }
    catch (std::invalid_argument ex)


```
(
    return 0.0;
)

int ItemMenu::AsInt(std::string item)
{
    return (int)AsDouble(item);
}

ItemMenu Settings::Get(std::string name)
{
    try
    {
        return m_menus.at(name);
    }
    catch (std::out_of_range ex)
    {
        throw std::out_of_range("Unable to locate menu " + name + " in settings");
    }
}

std::map<std::string, ItemMenu> Settings::GetParts()
{
    map<string, ItemMenu> parts;
    for (auto kvp : m_menus)
    {
        if (kvp.second.Type != "Part")
        {
            continue;
        }
        parts[kvp.second.Name] = kvp.second;
    }
    return parts;
}

double Settings::GaussianRandom(double mean, double std)
{
    std::normal_distribution<double> distribution(mean, std);
    return abs(distribution(m_generator));
}

settings.hpp

#pragma once

#include <string>
#include <vector>
#include <map>
#include <random>
#include "pugixml.hpp"
#include "dynamics.hpp"

struct ItemElement
{
    std::string Name;
    std::string Value;
    std::string Error;
    std::string Description;
};

struct ItemMenu
{
    std::string Name;
    std::string Description;
    std::string Type;
    std::map<std::string, ItemElement> Items;

    ItemElement Get(std::string item);
    std::string AsString(std::string item);
    double AsDouble(std::string item);
    int AsInt(std::string item);
};

class Settings
{
private:
    std::mt19937 m_generator;
    std::map<std::string, ItemMenu> m_menus;
    void load(std::string file);
    void ParseItemMenu(const pugi::xml_node& node);
public:
    Settings(std::string file);
```
ItemMenu Get(std::string name);
std::map<std::string, ItemMenu> GetParts();

double GaussianRandom(double mean, double stdev);

};
#endif

simulator.hpp

#ifndef SIMULATOR_HPP
#define SIMULATOR_HPP

#include <fstream>
#include <iostream>
#include <vector>
#include <map>
#include <algorithm>
#include <numeric>
#include "database.hpp"
#include "dynamics.hpp"
#include "pugixml.hpp"
#include "model.hpp"
#include "settings.hpp"

using namespace std;
using namespace pugi;

#include "settings.hpp"
#include "database.hpp"

inline void Stdev(double& mean, double& stdev, vector<double> v)
{
    double sum = std::accumulate(v.begin(), v.end(), 0.0);
    mean = sum / v.size();
    double sq_sum = std::inner_product(v.begin(), v.end(), v.begin(), 0.0);
    stdev = std::sqrt(sq_sum / v.size() - mean * mean);
}

struct SimulationResult
{
    ModelState Launch;
    ModelState Apogee;
    ModelState Landing;
    std::vector<ModelState> History;
};

class Simulator
{
private:
    Settings* m_settings;

    void Print(std::vector<Output>& a, std::vector<Output>& b, std::vector<Output>& c)
    {
        std::ofstream fout(m_settings->Get("Settings").AsString("OutputFileName"));
        if (fout.fail())
        {
            std::cerr << "Failed to open output file."
        }
        Output o;
        fout << o.Header();
        for (auto o : a)
        {
            fout << o.Row();
        }
        for (auto o : b)
        {
            fout << o.Row();
        }
        for (auto o : c)
        {
            fout << o.Row();
        }
    }

declare:
    void Run(bool exportCsv, bool monteCarlo, double dt, double recordInterval, SimulationResult& result);
    void GenerateFlightPlot();
    void RunMonteCarlo(int iterations);
    SimulationResult Run(bool mc, double dt);

    Simulator(std::string settingsFile)
    m_settings = new Settings(settingsFile);
);
}

void Simulator::Run(bool exportCsv, bool monteCarlo, double dt, double recordInterval, SimulationResult& result)
{
    ModelState state;
    state.T = 0;
    state.P = Vector(0, 0, 0);
    state.L = Vector(0, 0, 0);
    state.S = Vector(0, 0, 0);
    state.Q = Quaternion(1, 0, 0, 0);
    Database m_db(monteCarlo, *m_settings);
    LaunchModel launch(&m_db);
    launch.Integrate(dt, state, recordInterval, result.History);
    result.Launch = state;
    AscentModel ascent(&m_db);
    ascent.Integrate(dt, state, recordInterval, result.History);
    result.Apogee = state;
    if (exportCsv)
    {
        std::cout << "Apogee: " << state.S.Z << ", " << state.S.X << ", " << state.S.Y << std::endl;
    }
    ParachuteModel parachute(&m_db);
    parachute.Integrate(dt, state, recordInterval, result.History);
    result.Landing = state;
    if (exportCsv)
    {
        std::cout << "Landing position: " << state.S.X << ", " << state.S.Y << std::endl << std::endl;
    }
    if (exportCsv)
    {
        Print(ascent.GetOutput(result.History), std::vector<Output>(), std::vector<Output>());
    }
}

void Simulator::RunMonteCarlo(int iterations)
{
    double dt = 0.001;
    double recordInterval = 0.1;
    vector<SimulationResult> results;
    for (size_t i = 0; i < iterations; i++)
    {
        results.push_back(SimulationResult());
        Run(false, true, dt, recordInterval, results.back());
        if (i % 100 == 0 && i > 0)
        {
            std::cout << i << " / " << iterations << std::endl;
        }
    }
    std::cout << std::endl;
    // Make all of the results have the same history size
    int maxHistory = 0;
    for (size_t i = 0; i < results.size(); i++)
    {
        if (((int)results[i].History.size() > maxHistory) maxHistory = results[i].History.size();
    }
    for (size_t i = 0; i < results.size(); i++)
    {
        while (((int)results[i].History.size() < maxHistory) results[i].History.push_back(results[i].Landing);
    }
    // Calculate the mean trajectory
    SimulationResult mean;
    mean.History.resize(maxHistory);
    for (size_t i = 0; i < results.size(); i++)
    {
        mean.Launch += results[i].Launch;
        mean.Apogee += results[i].Apogee;
        mean.Landing += results[i].Landing;
        for (size_t j = 0; j < mean.History.size(); j++)
        {
            mean.History[j] += results[i].History[j];
        }
    }
    mean.Launch *= 1.0 / results.size();

mean.Apogee *= 1.0 / results.size();
mean.Landing *= 1.0 / results.size();
for (size_t i = 0; i < mean.History.size(); i++)
{
  mean.History[i] *= 1.0 / results.size();
}

// Calculate the standard deviation
SimulationResult stdev;
stdev.resize(maxHistory);
for (size_t i = 0; i < results.size(); i++)
{
  ModelState d1 = mean.Launch - results[i].Launch;
  ModelState d2 = mean.Apogee - results[i].Apogee;
  ModelState d3 = mean.Landing - results[i].Landing;
  stdev.Launch += d1 * d1;
  stdev.Apogee += d2 * d2;
  stdev.Landing += d3 * d3;
}
for (size_t i = 0; i < mean.History.size(); i++)
{
  ModelState d = mean.History[i] - results[i].History[i];
  stdev.History[i] += d * d;
}
stdev.Launch *= 1.0 / (results.size() - 1);
stdev.Apogee *= 1.0 / (results.size() - 1);
stdev.Landing *= 1.0 / (results.size() - 1);
for (size_t i = 0; i < mean.History.size(); i++)
{
  ModelState h = mean.History[i];
  ModelState s = stdev.History[i];
  stdev.History[i] *= 1.0 / (maxHistory - 1);
}

// Print out the statistics
cout << std::endl << " Results" << std::endl << "----------------------------------------" << std::endl;
cout << " Apogee: " << std::endl;
  cout << " X: " << mean.Apogee.S.X << " +/- " << stdev.Apogee.S.X << std::endl;
  cout << " Y: " << mean.Apogee.S.Y << " +/- " << stdev.Apogee.S.Y << std::endl;
  cout << 
  cout << " Landing: " << std::endl;
  cout << " X: " << mean.Landing.S.X << " +/- " << stdev.Landing.S.X << std::endl;
  cout << " Y: " << mean.Landing.S.Y << " +/- " << stdev.Landing.S.Y << std::endl;

std::ofstream fout(m_settings->Get("Settings").AsString("OutputFileName"));
if (fout.fail())
{
  std::cerr << "Failed to open output file." << std::endl;
}
  fout << "t, X, dX, Y, dY, Z, dZ" << std::endl;
for (int i = 0; i < maxHistory; i++)
{
  ModelState h = mean.History[i];
  ModelState s = stdev.History[i];
  fout << i * recordInterval << "," " i h.S.X " i " s.S.X " " i h.S.Y " i " s.S.Y " " i h.S.Z " i " s.S.Z " << std::endl;
}

#endif