Multi-Pass Aerocapture Approach for Orbital Insertion

a project presented to
The Faculty of the Department of Aerospace Engineering
San José State University

in partial fulfillment of the requirements for the degree
Master of Science in Aerospace Engineering

by

Bohdan O. Wesely

May 2025

approved by

Dr. Perikilis Papadapolous
Faculty Advisor

Special thanks for additional
expertise and guidance on this project

Paul Wercinski
NASA Ames Research Center
Entry Vehicles and Systems Branch

ABSTRACT

Multi-Pass Aerocapture Approach for Orbital Insertion

Bohdan O. Wesely

Aerocapture is an orbital insertion method that utilizes atmospheric drag to decrease an interplanetary spacecraft's (S/C) velocity into a captured orbit. The technique has been studied for several decades but has yet to be utilized in a flight mission. Aerocapture has the potential to significantly reduce S/C propellant mass, increase science payload mass, and/or reduce mission duration, especially for large outer planet (Neptune, Uranus) missions. This study explores a novel multi-pass aerocapture approach where initial entry would only reduce the velocity enough for a minimum captured orbit. This would be followed by additional atmospheric passes to sequentially bring the spacecraft apoapsis close to its target science orbit. Traditional aerocapture is where the spacecraft apogee is at or near the required science orbit after just one atmospheric entry. This project aims to construct conceptual first order trajectory and atmospheric entry analysis tools to compare both approaches. A fair assessment would determine whether the potential benefits of multi-pass such as reduced TPS mass and required control authority outweigh the drawbacks such as increased mission duration.

Table of Contents

Table of Figures

Nomenclature

C_N = Normal Force Coefficient
C_A = Axial Force Coefficient
C_S = Side Force Coefficient
C_D = Drag Coefficient
C_L = Lift Coefficient
C_Z = Velocity Oriented Side Force Coefficient
C_p = Pressure Coefficient
C_τ = Shear Coefficient
B_C = Ballistic Coefficient
α = Angle of Attack (deg)
β = Angle of Sideslip (deg)
θ = Local Surface Inclination Angle (deg)
D = Entry Vehicle Diameter (m)
R_N = Sphere Cone Nose Radius (m)
δ_{flank} = Sphere Cone Half Angle (deg)
M_∞ = Free Stream Mach Number
p_∞ = Free Stream Pressure (Pa)
ρ_∞ = Free Stream Density (kg/m^3)
V_∞ = Free Stream Velocity (m/s)
V = Inertial Velocity (km/s)
T = Temperature (K)
T_∞ = Free Stream Temperature (K)
T_W = Wall Temperature (K)
G_i = Species Gibbs Free Energy (kJ/mole)
H_i = Species Total Enthalpy (kJ/mole)
p_i = Species Partial Pressure (Pa)
S_i = Species Entropy (kJ/mole K)
η_i = Species Mole-Mass Ratio
v_i = Species Stoichiometric Coefficient
p_t = Total Pressure (Pa)
σ = Particle collision cross section (pm^2)
d_i = Species particle kinetic diameter (pm)
$\mathcal{R}$ = Universal Gas Constant (kJ/kmol K)
γ = Specific Heat Ratio
K_p = Equilibrium Constant
MR = Initial Ratio of Gas Particles
q_s = Stagnation point convective heating (W/cm^2)
J_s = Stagnation point total heat load (J/cm^2)
k = Aerothermal constant
K_n = Knudsen Number
k_b = Boltzmann Constant (J/K)
n = Particle Number Density (1/m^3)
λ = Mean Free Path (m)

$\boldsymbol{r}$ = Orbital Position Vector
$\boldsymbol{v}$ = Orbital Velocity Vector
τ = Orbital Period
μ = Gravitational Constant (km^3/s^2)
J_2 = 2nd Zonal Harmonic
r_a = Radius at Apoapsis (km)
r_p = Radius at Periapsis (km)
e = Eccentricity
a = Semi Major Axis (km)
i = Inclination (deg)
ω = Argument of Periapsis (deg)
Ω = Longitude of the Ascending Node (deg)
Θ = True Anomaly (deg)
Az = Azimuth (deg)
fpa = Flight Path Angle (deg)
ΔV = Change in Velocity (km/s)
I_{sp} = Specific Impulse (s)
g_o = Specific Gravity (m/s^2)
t_b = Burn Time (s)

1. Introduction

The 2023-2032 National Academies Planetary Science and Astrobiology Decadal Survey listed the ice giants (Uranus, Neptune) as a top priority for science exploration missions [2]. These destinations have longer mission durations and higher arrival velocities compared to the gas giants or inner planets and therefore demand a higher spacecraft mass fraction allocated to the propulsion system and propellant. The mass penalty of a traditional propulsion systems makes aerocapture of particular interest for these ice giant destinations. Venus, Mars, and Earth return missions are also viable for aerocapture orbital insertion though with less mass savings due to their proximity to Earth. A great deal of literature and concept studies exist on aerocapture and the vast majority focus on a single pass approach (ref. [9]-[15], [19], [23], and [24]). The main takeaways are that Aerocapture may require several new technologies related to low-medium L/D entry vehicles, high performance thermal protection systems (TPS) such as 3DMCP, HEEET, etc. and novel guidance and control techniques to account for atmospheric uncertainties. New technologies present a degree of risk when it comes to mission planning.

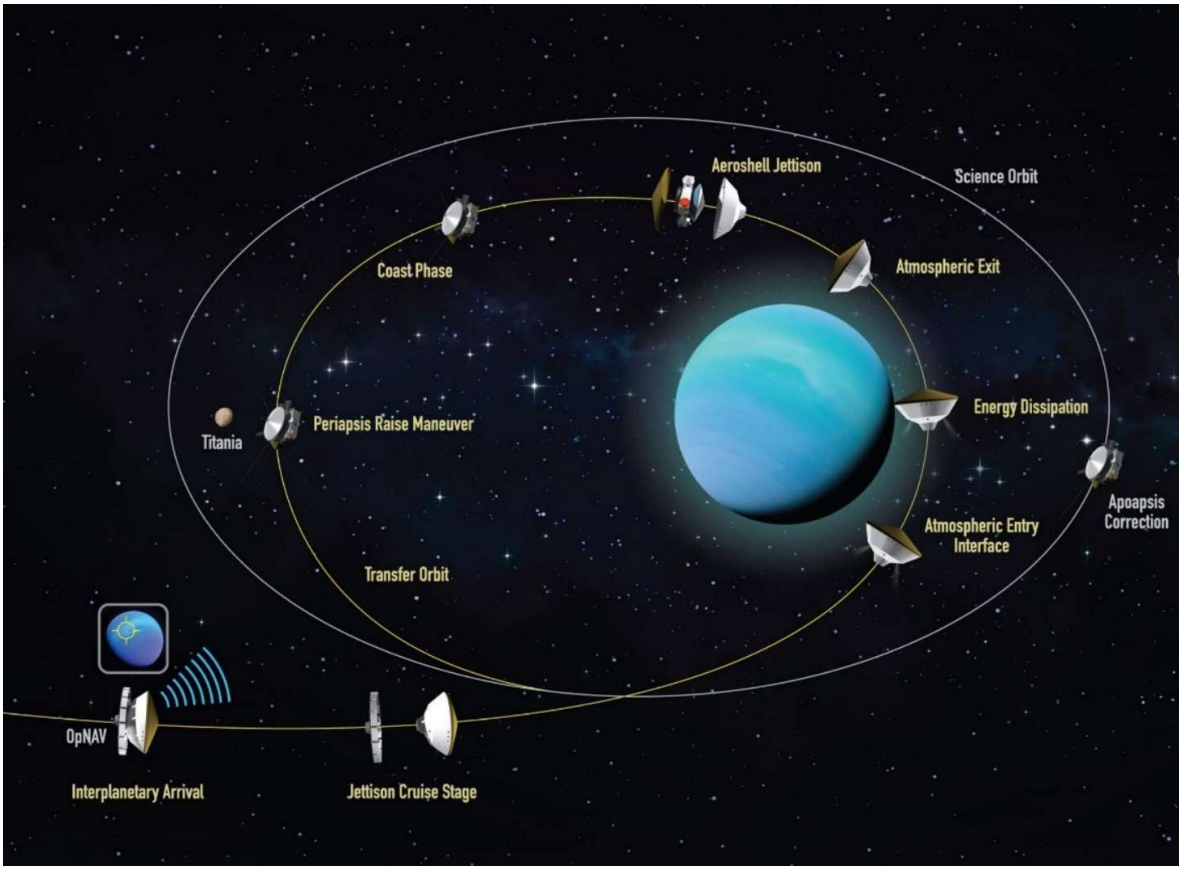


Figure 1.1 Traditional Single-Pass Aerocapture from ref. [9]

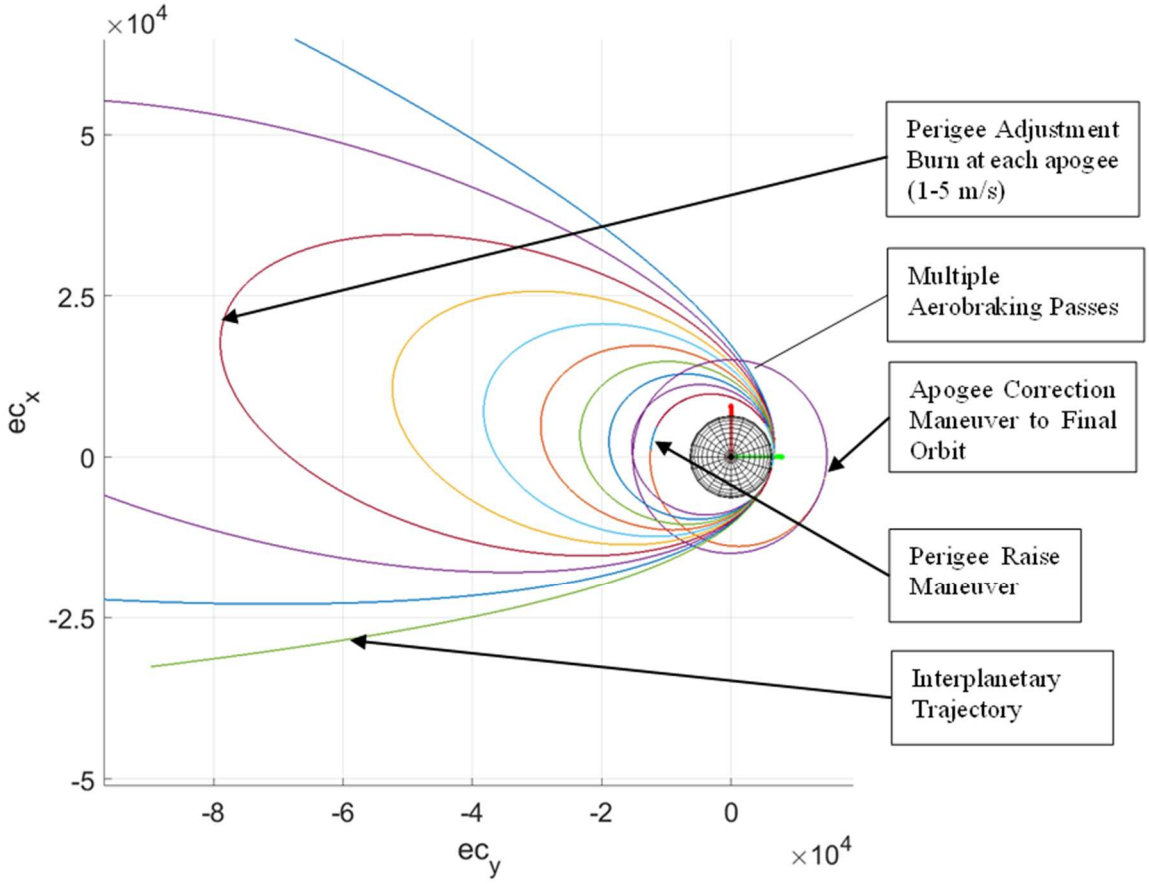


Figure 1.2 Multi-Pass Aerocapture Approach

2. Literature Review

2.1. Aerocapture and Entry Descent and Landing (EDL)

The aerocapture concept shares much of the same technology requirements as traditional entry descent and landing (EDL) through planetary atmospheres. Existing literature on aerocapture based missions contains analysis, propositions, and design methodologies derived from EDL. Atmospheric flight at orbital velocities creates extreme heating environments in both aerocapture and EDL mandating the need for TPS. Guidance, navigation, and control (GN&C) is similar for both mission cases. Aerocapture velocities remain in the orbital regime with a large portion of the trajectory at high altitudes in non-continuum flow. In traditional EDL, navigation and maneuvers can be performed at lower altitudes where the atmosphere is denser, offering more control authority and flexibility in targeting a precise landing location. With Aerocapture, all maneuvering action must be performed at orbital velocities in the outer fringes of the atmosphere.

The main benefit of aerocapture as discussed in section 1 is the propellant mass savings allowing for larger payloads and/or shorter mission durations. The TPS in traditional EDL is still a significant vehicle mass fraction which is determined by the entry environment and materials used. Systematic TPS and entry vehicle design methodologies have been used on all space missions requiring entry descent and landing through a planetary atmosphere. Determination of the TPS size and mass requirements allow for a quantitative comparison between traditional chemical propulsion, single-pass aerocapture, and multi pass aerocapture, which is the objective of this project.

2.2. Entry System Design

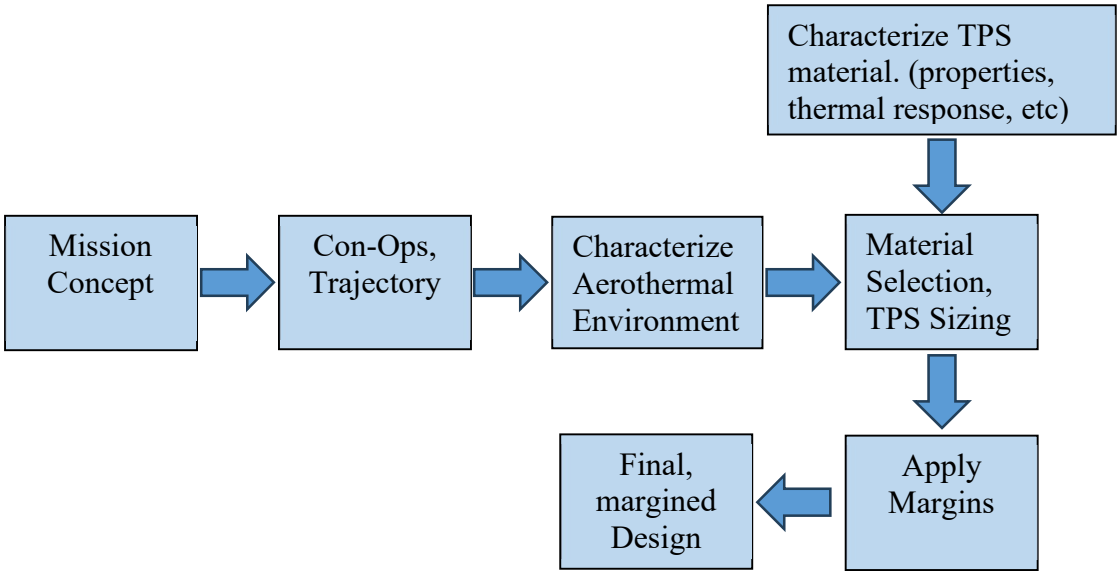


Figure 2.1 Entry System Design Flow

Science and/or space exploration objectives typically drive a mission concept and destination. The mission concept creates high level requirements and initiates the project

preliminary design phase. At this point mission planners can design a preliminary trajectory and navigation solution for the spacecraft to reach its destination. This spans from launch, injection maneuvers, cruise, orbital insertion, and entry descent and landing. Orbits and trajectories can be modeled in the simplest form by Kepler's law of motion (eq. 2.1).

$$F_g = \frac{Gm_1m_2}{r^2} \tag{2.1}$$

Orbit shapes are characterized by conic sections, ellipses and hyperbola being the most common. Patched conic methods can be used to form a complete mission trajectory by stitching together orbital segments [27]. In the conceptual design phase this offers a reasonable approximation. Additional physics such as atmospheric drag, oblateness, gravitation forces from other planets, and solar radiation can be added to the fundamental Kepler equations for increased fidelity. Reference [22] describes modeling 2-body Keplerian orbits extensively and presents algorithms for implementing models in a simulation environment such as MATLAB.

An atmospheric entry trajectory is also governed by Kepler's laws with the key addition of aerodynamic forces. Once a trajectory is outlined, the velocity magnitude and atmospheric conditions drive the aerothermal heating environment. At orbital velocities, the extreme compression as gases impinge on the entry vehicle and can produce temperatures up to 20000 K behind the shock front [1]. Aerothermal heating of an entry vehicle is an energy balance problem at the surface with convective and radiative heating being the dominant methods of heat transfer from the flow field to the vehicle TPS.

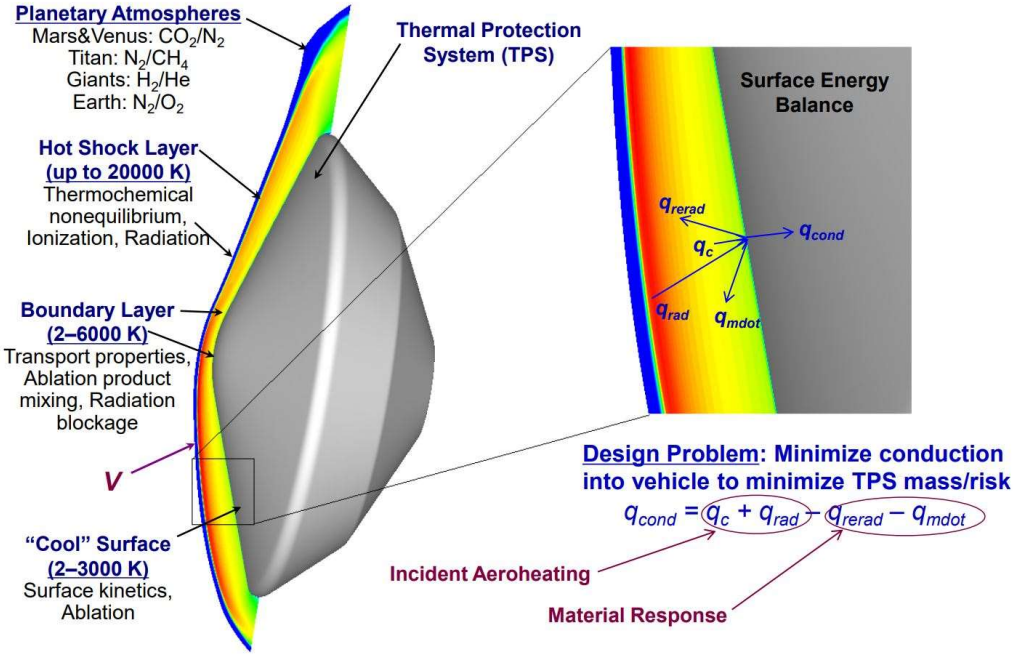


Figure 2.2 Aerothermal Environment ref. [1]

At the extreme environments of atmospheric entry, chemical reactions and diffusion greatly affect energy transfer in the shock layer region. Hypersonic and High Temperature Gas Dynamics" [17], by John D. Anderson describes the physics at play in these flow regimes in detail and has been a key resource for this project. Predictive design tools are invaluable for EDL mission designers to estimate how an entry vehicle will perform during atmospheric flight and how much TPS is required. The state of the art of entry environment predictions is highly parallelized CFD (Computational Fluid Dynamics) with extensive chemistry and radiation models as well as Direct Simulation Monte Carlo (DSMC) for non-continuum flows. Lower fidelity tools also exist such as the Sutton Graves approximation and generalized chapman method, these are often used in conjunction with CFD anchor points [1]. Various methods exist to predict the chemical reactions and resulting gas compositions in hypersonic flow fields. References [3]-[5] outline computational methods of predicting chemical compositions based on empirical thermodynamic data. Additional details on these first order aerothermal modeling tools will be discussed in sections 4.2 and 4.3.

Accurate vehicle aerodynamic force data is necessary for accurate trajectory results, especially for guided trajectories. CFD results can output lift, drag, and other aero coefficients but first order tools like Modified Newtonian Theory are a reasonable first order approximation [13]. Reference [17] describes a variety of aero force predictive methods in detail, [28] outlines a Newtonian panel method and was used to validate methods discussed in section 4.

A recent NASA early career initiative (ECI) studied and proved the viability for aerocapture at Uranus with present technologies, [9]-[15]. References [10], [13], and [14] discuss the mission concept and aerothermal environment predictions while [12] discusses the TPS design and sizing. A similar performance analysis for a Venus aerocapture is outlined in [23]. and is used one of the main validations for the methodologies described in section 4.

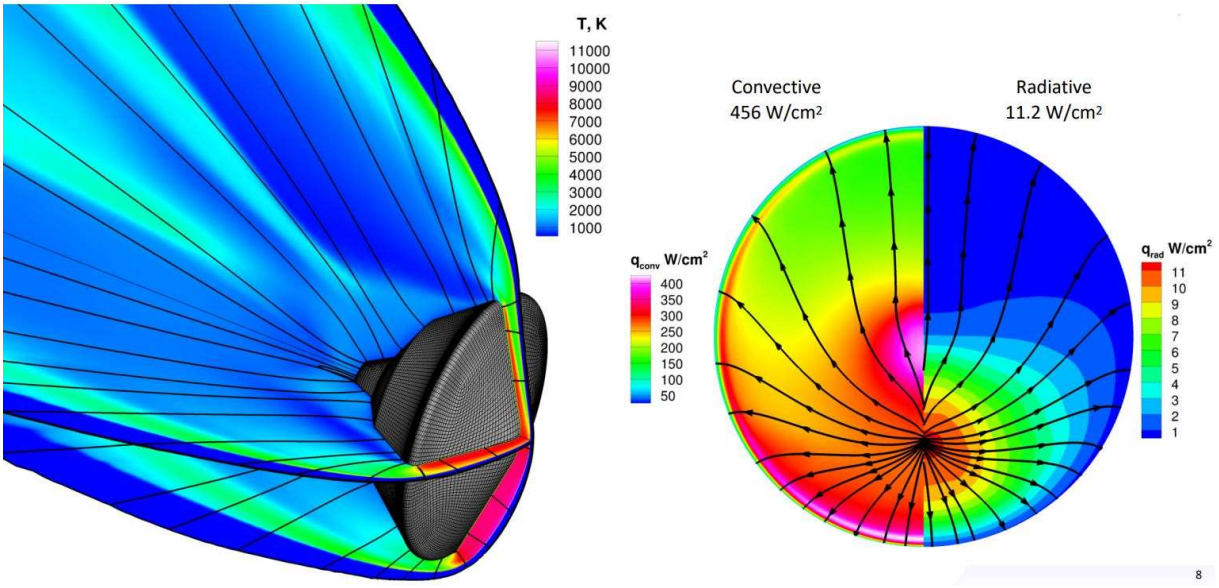


Figure 2.3 CFD Heating Predictions for Uranus Aerocapture [14]

3. Project Objective

This study aims to determine whether a multi-pass approach for a flagship outer planet or Venus/Earth return mission presents less risk and is more viable with current technologies than a traditional single-pass approach. The first phase of the study will incorporate rudimentary, first order analysis tools with several key assumptions to perform a fair assessment between the two approaches. The second phase will incorporate more advanced analysis tools, begin accounting for uncertainties, and increase the overall level of fidelity. The primary comparison metrics will be TPS mass and mission duration. The maximum heat flux and total heat load for each multi-pass will be compared to the single pass and contribute to the TPS sizing methodology. Higher fidelity models might assess whether each of the multiple passes introduces less trajectory dispersion than a single higher intensity pass. In a single-pass aerocapture, the aeroshell can be jettisoned immediately after atmospheric exit, but for a multi-pass approach it must be retained which may introduce issues with thermal soak back and communications. These drawbacks will also be assessed. In the end, the various analysis tools should demonstrate a clear assessment between the two mission architectures. Any difference between the low and high fidelity analysis tools will also be discussed.

3.1. Uranus and Venus for Multi-Pass Aerocapture

At the inception of this project, Uranus was chosen as the first mission destination to be tested. The NASA early career initiative (ECI) studied and proved the viability for aerocapture at Uranus with present technologies, [9]-[15]. The simulations run in the ECI targeted a post-aerocapture orbit of 4000 x 550000 (5.5e5) km. One of the objectives of this high apogee is to encounter Titania, the largest moon of Uranus. Brief 2-body orbital mechanics calculations illustrate that the 5.5e5 km target apogee is already close to a minimum captured orbit. There isn't significant allowance for meaningful ΔV savings by taking multiple lighter passes to get to the same apoapsis. Lower Uranus orbits can be achieved through gravity down-pumping maneuvers via Titania and other moons albeit more slowly than aerobraking passes. The above factors make a multi-pass aerocapture approach less viable for Uranus. Neptune, Jupiter, and Saturn also have large natural satellites that can provide gravity assists and pump-down maneuvers essentially for free, provided the initial orbit is high enough for a periodic encounter. Venus, lacking a natural satellite could potentially be more attractive for a multi-pass approach for low orbiting missions, and will therefore be the first destination to be analyzed. A concept study outlining a single-pass Venus aerocapture is outlined in [23] targets a low 500x500km orbit. Once a low fidelity simulation tool has been built and tested for the Venus case, performance to lower Uranus orbits or other destinations may be assessed.

4. Analysis Methodology

4.1. Orbital Mechanics

A simple 2-body orbital mechanics model will be used for the low fidelity flight simulation in this study. The simulation will start with an interplanetary state vector that is pulled from the data used in [23]. This will allow an initial comparison to current literature to validate the simulation. An altitude of 150 km will be used as a cutoff for atmospheric flight effects, the same as [23], though a higher cutoff may be considered in further refinements. The coordinate system for the orbit propagation will be a cartesian, equatorial centric inertial (ECI) reference frame (Figure 4.1). The following assumptions will be made in this 2-body model.

1) J2 spherical harmonic perturbations will be modeled, no perturbations from any other bodies.

2) No solar radiation pressure or any other perturbing body forces on vehicle.

3) Maneuvers assume perfect thrust alignment with vehicle velocity vector, constant ISP.

4) Vehicle modeled as a point mass, no rigid body or attitude dynamics.

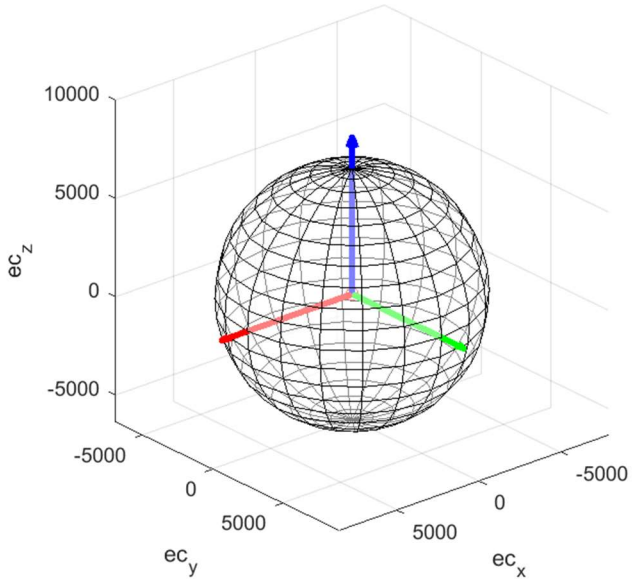


Figure 4.1 Coordinate System for Flight Simulation

$$\boldsymbol{r} = \begin{bmatrix} X\ \boldsymbol{ec_x} \\ Y\ \boldsymbol{ec_y} \\ Z\ \boldsymbol{ec_z} \end{bmatrix} \tag{4.1}$$

$$\mu = GM \tag{4.2}$$

$$\ddot{\boldsymbol{r}} = \frac{\mu}{r^3}\boldsymbol{r} \tag{4.3}$$

$$\ddot{\boldsymbol{r}} = \frac{\mu}{r^3}\boldsymbol{r} + \frac{F\dot{\boldsymbol{r}}}{mr} \tag{4.4}$$

4.2. Hypersonic Atmospheric Flight

Once the altitude threshold is reached, the simulation will switch to a separate integration scheme to account for atmospheric effects. The atmospheric drag model used in the initial low fidelity study is spelled out in eq. 4.5. This assumes a constant drag coefficient and constant vehicle orientation with the drag vector, $\boldsymbol{P}$, acting opposite to the velocity vector. The velocity vector $\boldsymbol{v_{rel}}$ shown in 4.6 factors in winds and planetary rotation.

$$\ddot{\boldsymbol{r}} = \frac{\mu}{r^3}\boldsymbol{r} + \boldsymbol{P} \tag{4.5}$$

$$\boldsymbol{P} = \frac{1}{2}\rho_\infty |\boldsymbol{v_{rel}}|^2 C_D A \frac{|\boldsymbol{v_{rel}}|}{\boldsymbol{v_{rel}}} \tag{4.6}$$

4.2.1. Modified Newtonian Method

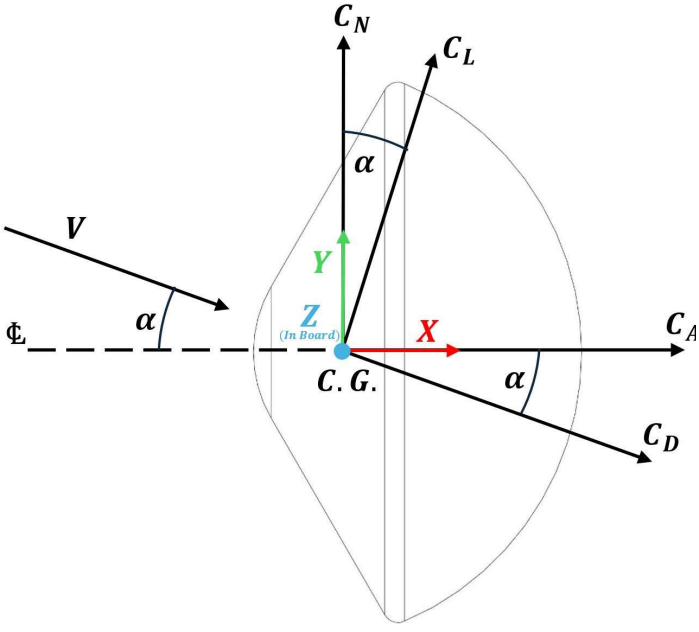


Figure 4.3 Vehicle Force Coefficients

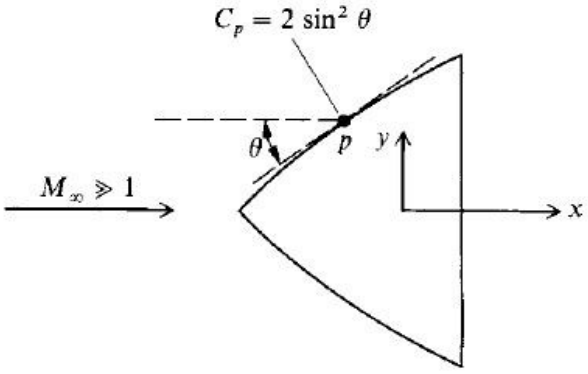


Figure 4.2 Newtonian Method Concept [17]

Validated aerodynamic data for the entry vehicle is critical to accurately predict a post aerocapture trajectory. Typically, blunt body entry vehicles are flown at a small angle of attack to allow for some form of control authority such as bank angle modulation. For simplicity, the initial analysis will assume the vehicle is at a constant zero angle of attack with the center of mass along the geometric centerline. With this assumption, $C_A = C_D$ $(\alpha = 0)$, making C_D the only aerodynamic coefficient that requires an accurate prediction. The Modified Newtonian Method is often utilized as a first order approximation for hypersonic aerodynamics. It assumes that the pressure coefficient at any point on a body surface is proportional to the sine squared of the local inclination angle to the flow, θ in Figure 4.2, (eq. 4.7). Modified refers to the addition of $C_{p_{max}}$ which comes from the isentropic relation shown in eq. 4.8, in the traditional Newtonian method the relationship is simply $C_p = 2\sin^2\theta$. As the Mach number increases the more accurate this prediction tends to be [17]. The Venus aerocapture study [23] that will be used for model validation utilizes a 60° sphere cone with a nose radius 25% of the vehicle diameter. A panel based Modified Newtonian method was used to achieve a first order approximation of the drag coefficient for this geometry. Derivation of this model is shown below.

Pressure coefficient of one segment i

$$C_{p_i} = C_{p_{max}} \sin^2\theta_i \tag{4.7}$$

$$C_{p_{max}} = \frac{2}{\gamma M_\infty^2}\left[\frac{p_{O_2}}{p_\infty} - 1\right] \tag{4.8}$$

$$C_{p_i} = \frac{p_i - p_\infty}{q_\infty} \tag{4.9}$$

$$C_D = \frac{D}{q_\infty S} \tag{4.10}$$

Drag of single segment i

$$D_i = (p_i - p_\infty)S_i \tag{4.11}$$

Shadow area for an axisymmetric vehicle profile at $\alpha = 0$

$$S_i = \pi(y_i^2 - y_{i-1}^2) \tag{4.12}$$

Combine 8, 8b, 8d, and 8e and sum all panel segments

$$D = q_\infty C_{p,max}\pi \sum_{i=2}^{n}(y_i^2 - y_{i-1}^2)\sin^2\theta_i \tag{4.13}$$

$$C_D = \frac{C_{p,max}\pi \sum_{i=2}^{n}(y_i^2 - y_{i-1}^2)\sin^2\theta_i}{S} \tag{4.14}$$

Equation 4.14 was solved numerically in MATLAB for the 60° sphere cone geometry which led to a drag coefficient of 1.3933. After ~Mach 10 $C_{p_{max}}$ becomes essentially Mach independent. [29] presents a similar Newtonian prediction method and in one section studied a vehicle with a 70° sphere cone. This case was mimicked with the internally developed tool and a C_D of 1.624 at 0° α was generated. The results from [29] are shown in Figure 4.4 and indicate good agreement with the internal model, adding confidence to the 1.3933 figure to be used for the aerocapture simulation. The modified Newtonian method is most accurate in continuum flow regimes which only partly represent aerocapture trajectories, though it is where the aerodynamic forces are greatest and have the most influence. Methods for modeling rarefied and free molecular flow are discussed in 9.2.

Parameter	*Sphere-Cone*	*Biconic*
r_n	0.3 ft.	1.0 in.
δ_1	70.0^o	17^o
δ_2	-	8^o
d	2.5 ft.	19.6 in.
h	-	48.0 in.

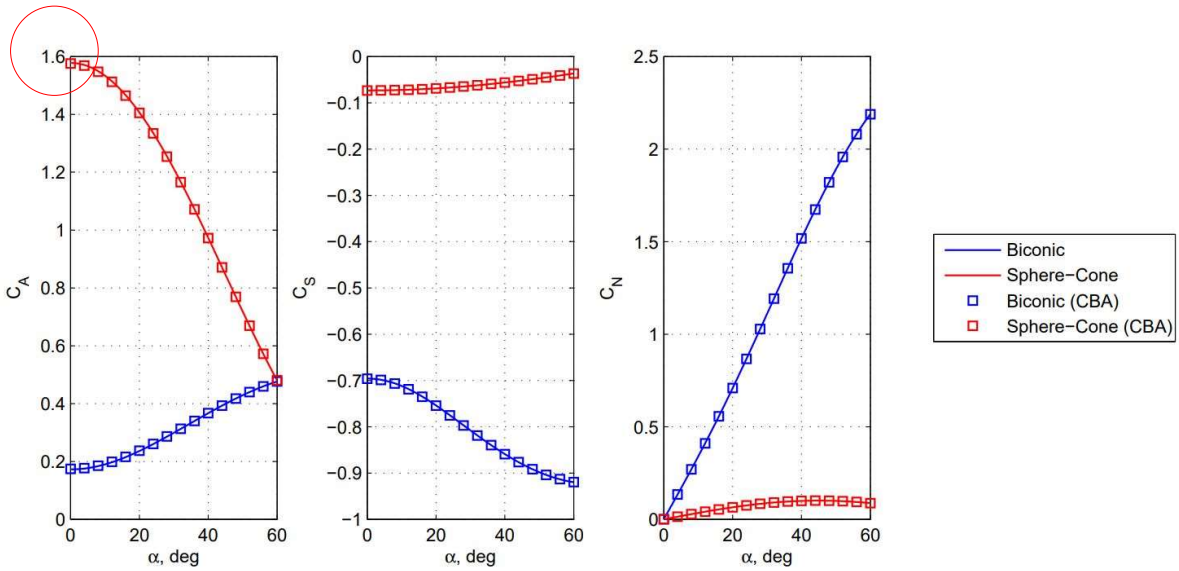


Figure 4.4 Force Coefficients for Model Validation [29]

4.2.2. Modified Newtonian Method $\boldsymbol{\alpha \neq 0}$

The panel method described in 0 must be significantly expanded for a vehicle in 3 dimensions to model aerodynamic effects of angle of attack or sideslip. This additional modeling allows for realistic lift-up and lift-down cases to be considered. L/D ratios can also be compared for different vehicle geometries. This opens the door to modeling moments, stability, and rudimentary guidance algorithms, though those topics are out of scope for this project. For entry vehicles, the AOA convention is opposite of traditional aircraft with positive α pointing below the velocity vector (Figure 4.3). For the 3D case a more general relationship must be built for axisymmetric bodies. Equation 4.15 describes the Newtonian method for any body shape in 3 dimensions to determine aerodynamic force coefficients [29]. Parameterization of the vehicle surface is needed to calculate the local inclination angle θ and normal vector $\boldsymbol{n}$.

$$\begin{bmatrix} -C_A \\ C_N \\ -C_S \end{bmatrix} = \frac{1}{A_{\text{ref}}} \iint_S C_p \begin{bmatrix} \boldsymbol{n}^T \hat{\boldsymbol{x}} \\ \boldsymbol{n}^T \hat{\boldsymbol{y}} \\ \boldsymbol{n}^T \hat{\boldsymbol{z}} \end{bmatrix} dA \tag{4.15}$$

$$sin\theta = \frac{\boldsymbol{V}_\infty}{|V_\infty|} \cdot \boldsymbol{n} \tag{4.16}$$

The methodology is similar to the $\alpha = 0$ case, where the geometry is broken into panels, though an additional summation revolved around the body is necessary as $sin\theta$ will vary radially for the $\alpha \neq 0, \beta \neq 0$ case. Both summations essentially solve the surface integral in 4.15 numerically, which adds flexibility for more complex shapes over an analytical solution. In [29], the surface integral is solved analytically using Mathematica tools. The origin for the discretization can be arbitrary as long as the XYZ convention is the same as the force coefficient reference frame. If moment coefficients are desired a reference point must be chosen. The process involves a series of vector rotations to sum the normal vectors and their reference areas.

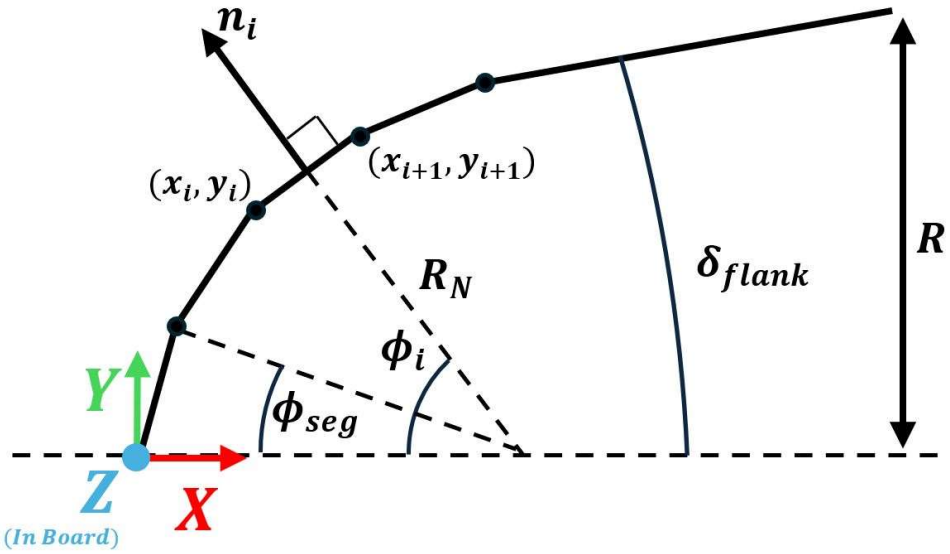


Figure 4.5 Side View Discretization with 4 Nose Segments

$$\phi_{seg} = \frac{\frac{\pi}{2} - \delta_{flank}}{N_{seg}} \qquad \phi_i = \frac{\phi_{seg}(2i-1)}{2} \qquad x_{i+1} = R_N - R_N \cos(\phi_{seg} i)$$

$$y_{i+1} = R_N \sin(\phi_{seg} i) \qquad \boldsymbol{n_i} = \begin{bmatrix} \cos\phi_i \\ \sin\phi_i \\ 0 \end{bmatrix}$$

The above relations are all that are necessary for the $\alpha = 0$ case where $\boldsymbol{n}$ is the same at any axisymmetric position around the vehicle body. If $\alpha \neq 0$ *or* $\beta \neq 0$, $\boldsymbol{n}$ varies and each radial segment contributes a different force component on the vehicle, necessitating an additional summation.

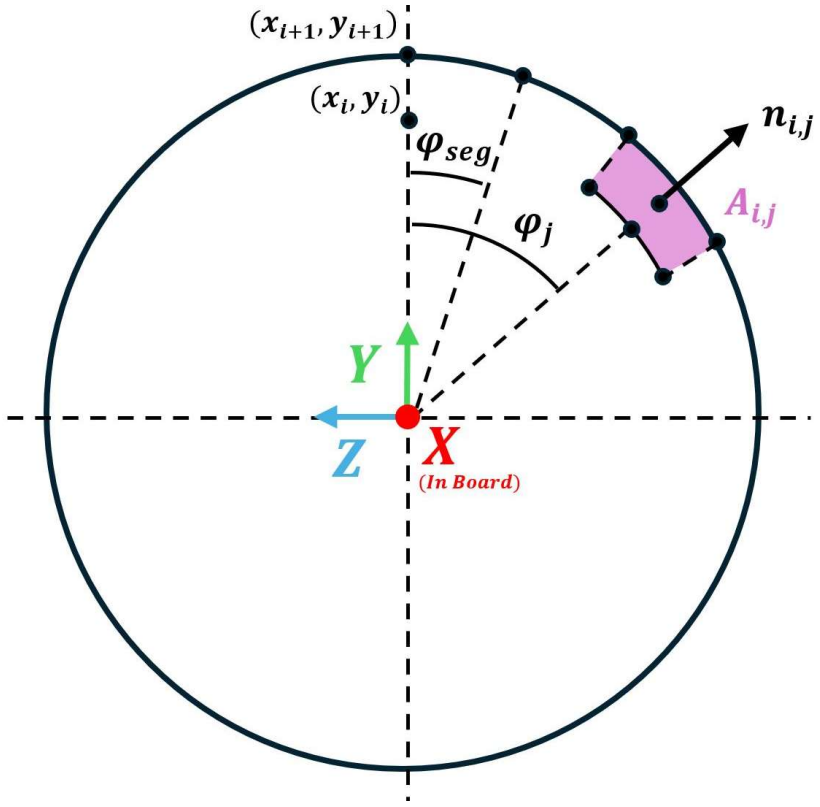


Figure 4.6 Nose View Discretization Example

$$\varphi_{seg} = \frac{2\pi}{M_{seg}} \qquad \varphi_j = \frac{\varphi_{seg}(2j-1)}{2} \qquad \boldsymbol{n_{i,j}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\varphi_j & sin\varphi_j \\ 0 & -sin\varphi_j & cos\varphi_j \end{bmatrix} \boldsymbol{n_i}$$

$$A_{tot} = \pi d(y_{i+1} + y_i) \qquad d_{seg} = \sqrt{(y_{i+1} - y_i)^2 + (x_{i+1} - x_i)^2} \qquad A_{i,j} = \frac{A_{tot}}{N_{seg}}$$

After the normal vector for each panel is computed, an additional rotation about α and β must be performed to obtain the final normal vector with respect to the vehicle velocity vector (4.17). The area of each segment ($A_{i,j}$) is the area of the entire frustrum (A_{tot}) that represents each lengthwise segment (i) divided by the desired number of radial divisions. After this rotation, the primary coordinate system remains in the velocity frame (X parallel with velocity vector), so the first force coefficients calculated are drag, lift, and sideslip. The reference area (A_{ref}) is just the vehicle surface area projected onto the Y-Z plane at $\alpha = 0$ which for an axisymmetric body is simply πR^2. The body force coefficients can be obtained by reversing the transformation in (12), represented by (14). As α and β increase, vehicle surfaces may become obscured from the flow, since the coordinate system remains in the velocity frame, a simple conditional statement can be implemented to check each panel for $\boldsymbol{N_{i,j}\hat{x}} < 0$ and assign $C_p = 0$ for those cases per Newtonian theory.

$$\boldsymbol{N_{i,j}} = \begin{bmatrix} cos\beta & 0 & -sin\beta \\ 0 & 1 & 0 \\ sin\beta & 0 & cos\beta \end{bmatrix} \begin{bmatrix} cos\alpha & -sin\alpha & 0 \\ sin\alpha & cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \boldsymbol{n_{i,j}} \tag{4.17}$$

$$\begin{bmatrix} C_D \\ -C_L \\ C_Z \end{bmatrix} = \frac{C_{p_{max}}}{A_{\text{ref}}} \sum_{i=2}^{N_{seg}} \sum_{j=2}^{M_{seg}} (\boldsymbol{N_{i,j}\hat{x}})^2 \begin{bmatrix} \boldsymbol{N_{i,j}\hat{x}} \\ \boldsymbol{N_{i,j}\hat{y}} \\ \boldsymbol{N_{i,j}\hat{z}} \end{bmatrix} A_{i,j} \tag{4.18}$$

$$\frac{\boldsymbol{V_\infty}}{|V_\infty|} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \qquad sin\theta = \frac{\boldsymbol{V_\infty}}{|V_\infty|} \cdot \boldsymbol{n} = \boldsymbol{N_{i,j}\hat{x}}$$

$$\begin{bmatrix} C_A \\ C_N \\ C_S \end{bmatrix} = \begin{bmatrix} cos\alpha & -sin\alpha & 0 \\ sin\alpha & cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}^T \begin{bmatrix} cos\beta & 0 & -sin\beta \\ 0 & 1 & 0 \\ sin\beta & 0 & cos\beta \end{bmatrix}^T \begin{bmatrix} C_D \\ -C_L \\ C_Z \end{bmatrix} \tag{4.19}$$

While this method adds complexity over the $\alpha = 0$ case, the matrix math is easily handled in MATLAB and can quickly run through various angle of attack and sideslip for a variety vehicle geometries. This is a powerful first order design tool that allows for quick generation of an aerodynamic database for axisymmetric entry vehicles. Setting the model up to compare with the sphere-cone case on page 14 of [29] with an initial 20° sideslip angle shows near exact agreement using 10 nose divisions and 20 radial divisions (Figure 4.8). Additional validations with [29] are discussed in appendix 9.1.1. The model was also compared with the pre-CFD aerodynamic modeling in [13] and the results are in family, though as previously mentioned the Newtonian method is only valid for continuum flow regimes (Figure 4.7). Ref. [13] discusses a Uranus aerocapture and utilizes a MSL like entry vehicle with a 70° cone angle. Integration of this model into the trajectory simulation is discussed in 4.2.4.

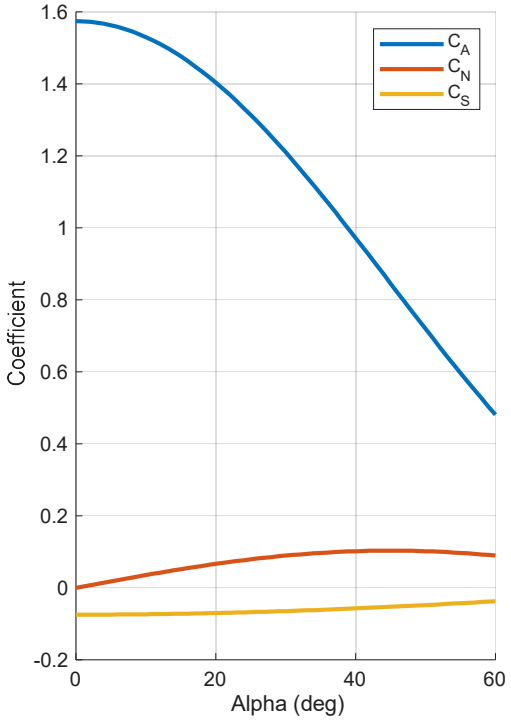


Figure 4.8 Aero Coefficient Validation with Figure 4.4, ref. [29]

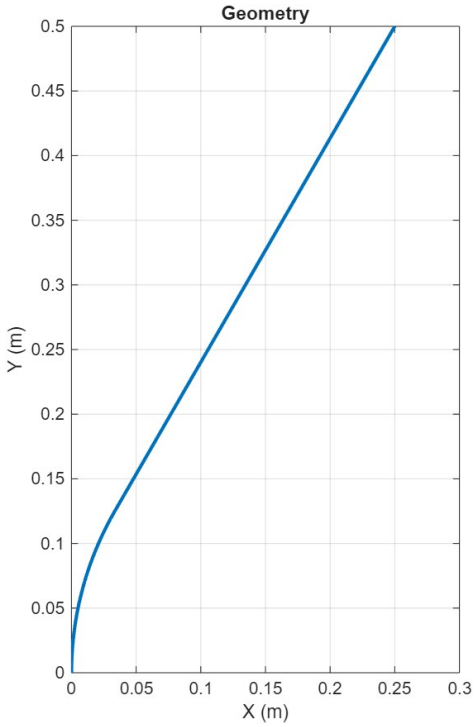


Figure 4.9 Modified Newtonian Geometry Output ($\delta_{flank} = 60°, \frac{R_N}{R} = 0.5$)

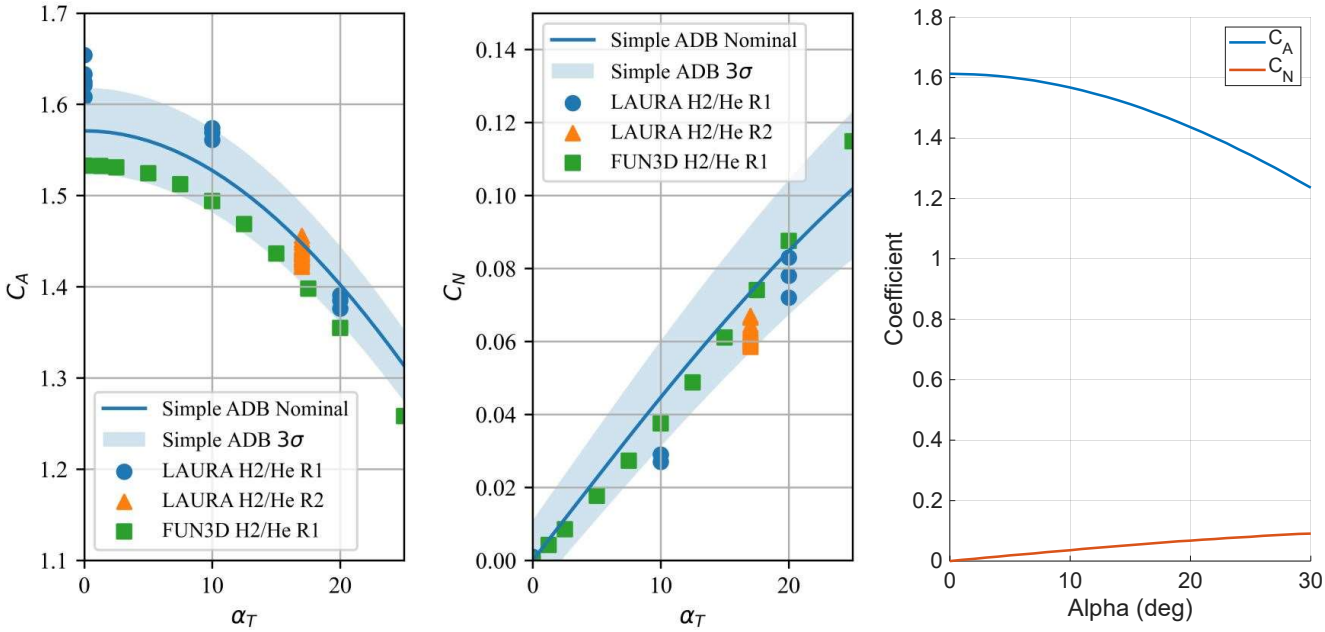


Figure 4.7 Aero Coefficient Validation with [13] (Uranus Aerocapture)

4.2.3. NASA Global Atmospheric Reference Model (GRAM)

The NASA developed Global Atmospheric Reference Model (GRAM) is an engineering design tool that can compute atmospheric data for every planet in the solar system that contains an atmosphere [7]. The program takes position and time inputs and utilizes ephemeris calculations and a host of empirical data to output conditions such as density, pressure, temperature, chemical mass fractions, winds, and more. GRAM can support monte carlo runs and 3-sigma dispersions on all properties, though only the nominal mean values are used for this initial study. This project is utilizing the latest 2.1 release from October 2024 that includes a MATLAB interface. Specifically, VenusGRAM [7] will be used in the initial analysis and validation with [23]. The following assumptions will be made for atmospheric flight.

1) Aerodynamic force coefficients $\boldsymbol{C_L}$, $\boldsymbol{C_D}$, and $\boldsymbol{C_S}$ allowed to vary with α and β
2) 3 DOF, Constant Vehicle Orientation, no GNC uncertainties or modeling
3) Temporal and special variations and winds will be modeled within GRAM
4) Atmospheric uncertainties and perturbations are not modeled

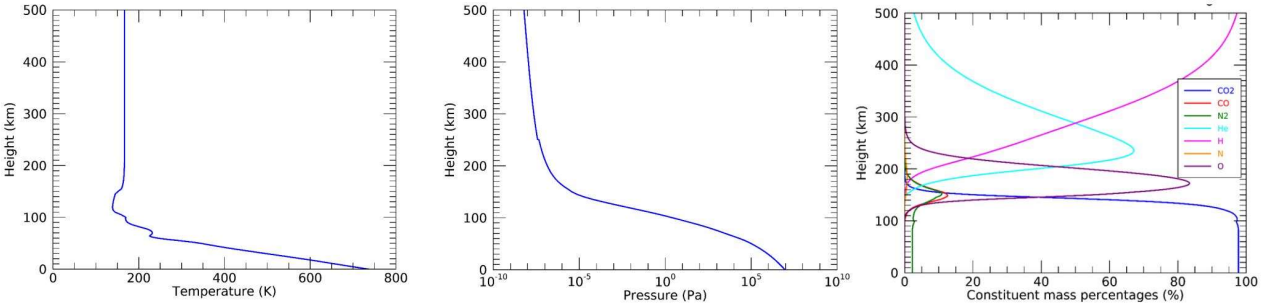


Figure 4.10 Venus Atmosphere Profile (ref. [7])

The winds on Venus are significant at higher altitudes and are faster than the planet's rotation. GRAM outputs winds in three components, north to south, east to west, and vertical. The wind velocities can reach well above 100 m/s which must be considered for accurate trajectory modeling. The winds show a clear difference between the day and night sides of the planet and are the strongest at the equator. The winds at lower altitudes move consistently against the planet's rotation though at higher altitudes (80+ km) there are significant variations and direction changes which can have a non-negligible effect on entry and aerocapture flight mechanics. Trajectories with and without winds are shown in Figure 4.15.

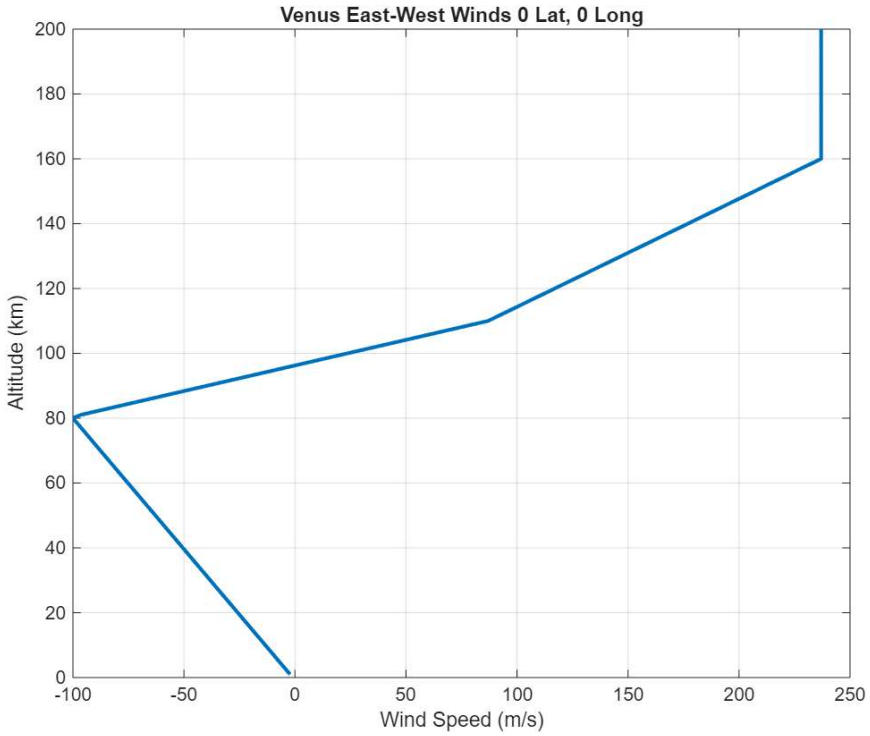


Figure 4.11 Venus Winds vs. Altitude

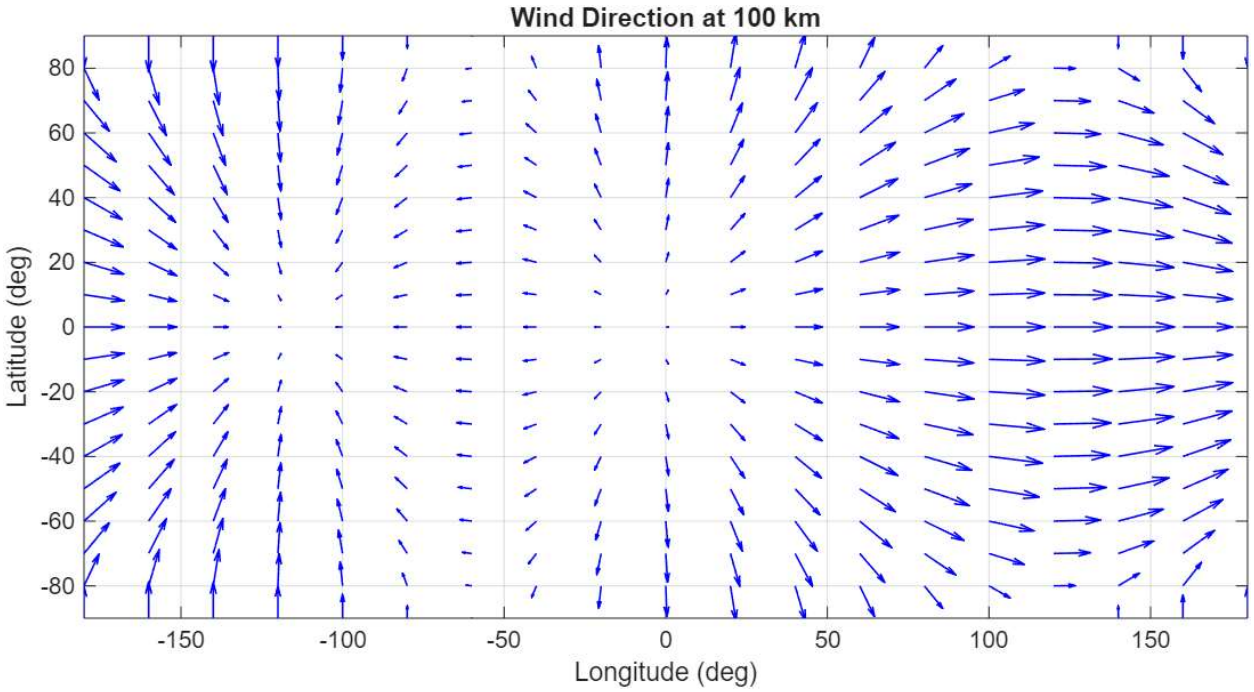


Figure 4.12 Venusian Wind Direction at 100 km Altitude (JD 2459138)

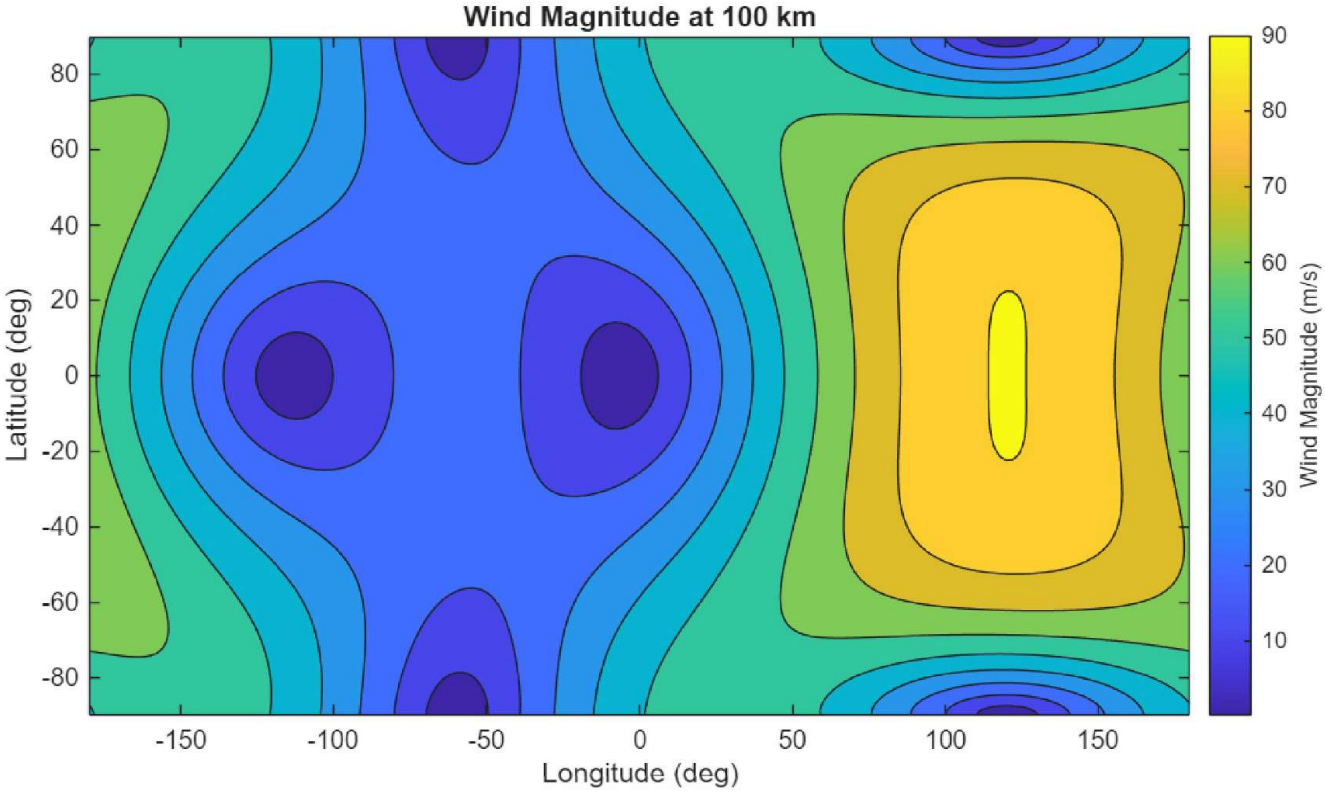


Figure 4.13 Venusian Wind Magnitude at 100 km Altitude (JD 2459138)

With the addition of aero coefficients in 3 dimensions, the effect of how the winds interact with the vehicle required evaluation. If the vehicle is flying at a nominal ballistic trajectory $\alpha = \beta = 0$, winds and to a small extend the Coriolis effect will produce a relative velocity vector different from the inertial velocity which will induce an effective angle of attack or sideslip on the vehicle, this is assuming 3DOF mechanics where the vehicle's orientation is fixed to the velocity frame. At a high-altitude condition on Venus with significant east to west winds around 200 m/s (~150 km), the worst case crosswind scenario would be an azimuth of 0° where the entry vehicle is flying south to north in a polar orbit. An entry velocity of 11 km/s would produce an effective sideslip angle of ~1.04°, which is non-negligible. To model this effect, an additional transform between the inertial and relative velocity frame had to be developed. With a sufficiently blunt body (~70° half angle), the α vs. C_L lift slope is the opposite of a slender body due to the blunt frontal forebody producing lift instead of the lengthwise chord surface area. This effect results in the entry vehicle being pulled in the direction of the wind with a net component to the resulting force in the +Y direction, almost like a sailboat sailing upwind (Figure 4.14). While counter intuitive at first, the effect checks out when the appropriate rotation matrices and conventions are applied.

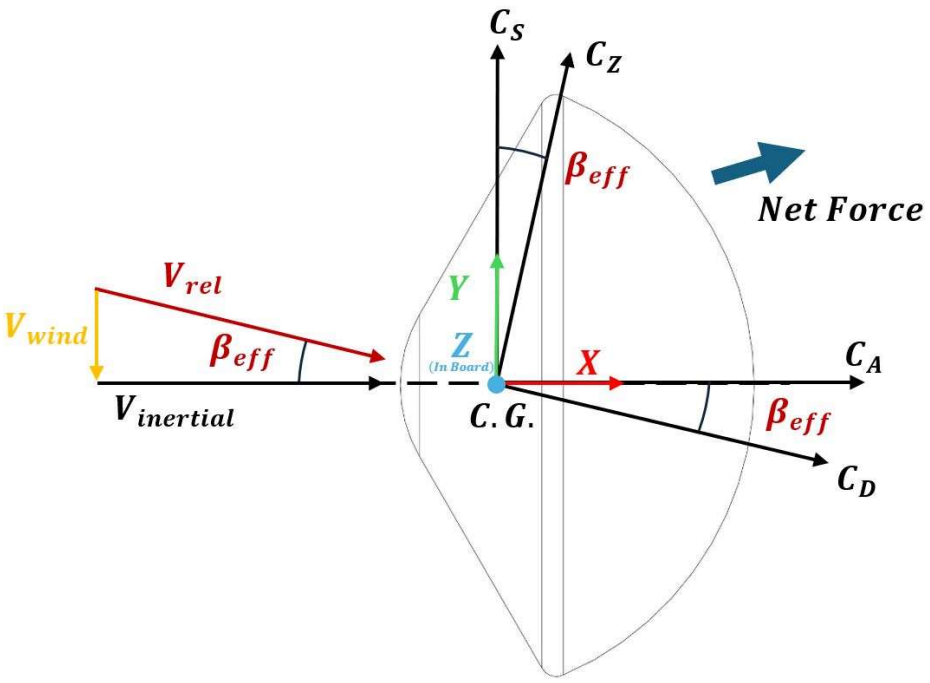


Figure 4.14 Blunt Body Crosswind Illustration

Polar Orbit
V = 11/km/s, Alt = 150 km, Fpa = -5.61°
Final Azimuth (Winds): 173.257°
Final Azimuth (No Winds): 174.242°

Equatorial Orbit
V = 11/km/s, Alt = 150 km, Fpa = -5.61°
Final Altitude (Winds): 828.1516 km
Final Altitude (No Winds): 375.0636 km

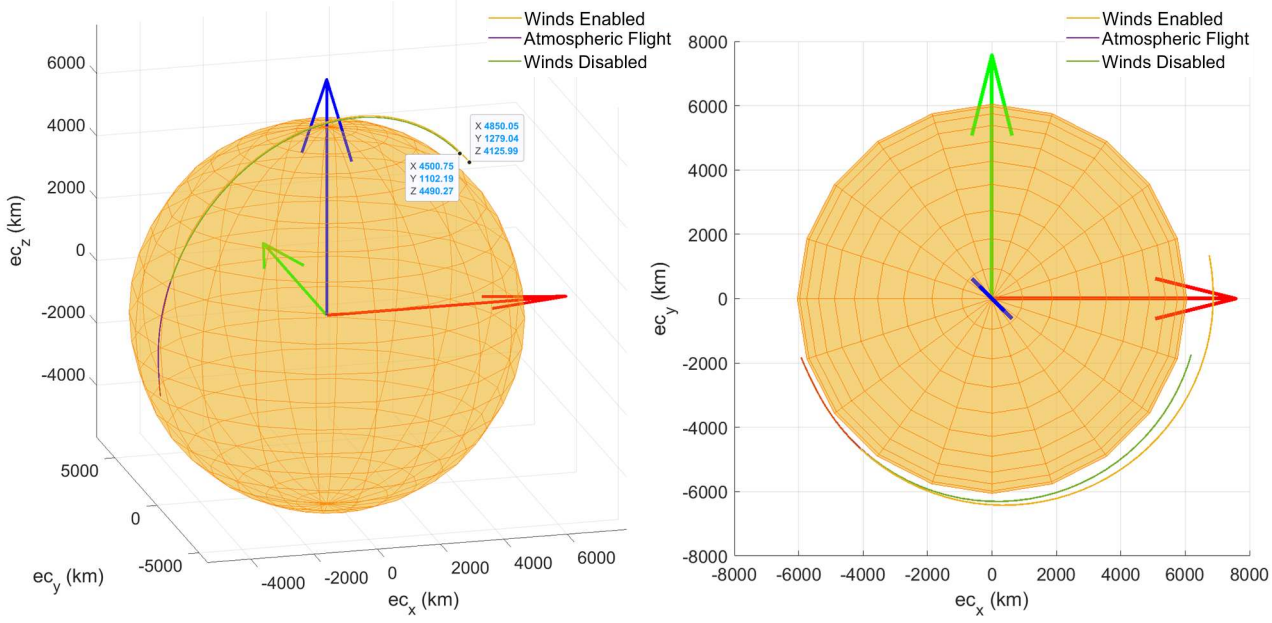


Figure 4.15 Polar and Equatorial Trajectory Wind Dispersions (JD = 2459138)

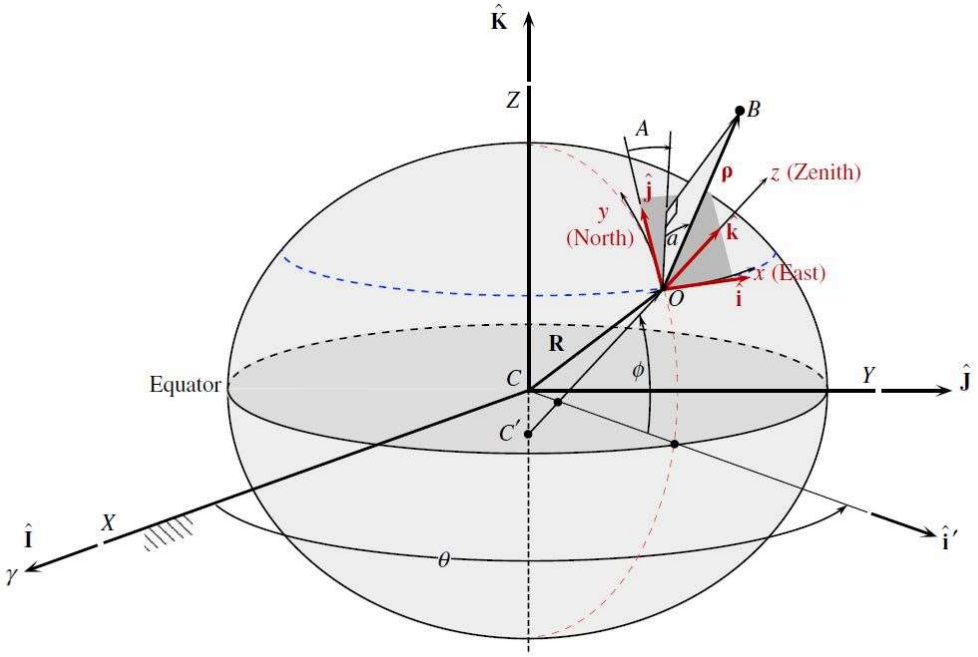
4.2.4. Reference Frames

The position inputs and outputs to GRAM are in the topocentric frame utilizing longitude and latitude coordinates. A conversion to the base ECI frame that the simulation runs on was necessary. A reference frame transposition was derived that also required the local sidereal time of the planet to account for rotation in comparison to the fixed inertial ECI frame. The rotation matrix QXx and its transpose shown in Figure 4.16 were used to convert to the East North Zenith (ENZ) topocentric frame necessary for the GRAM inputs. This transform was also used to support trajectory inputs in the form of azimuth, entry flight path angle, and velocity magnitude.

A time component was also necessary to extract the correct prime meridian location of Venus (0° Longitude). The latest report from the International Astronomical Union (IAU) [30] outlines measurements for planetary rotation positions at time since standard epoch (1/1/2000). The standard reference direction for this frame is Earth's vernal equinox at the time of epoch, this is known as the J2000 reference frame. The rotation angle (sidereal time) of Venus is given by eq. 4.20 from [30].

$$W = 160.20 - 1.4813688d \quad (4.20)$$

This is good approximation for the initial analysis over short timespans, investigation into seasonal atmospheric variations is possible in the GRAM model. GRAM is also capable of outputting an ephemeris state with various properties that can be used to calculate the exact sidereal time, though this has not yet been implemented. Figure 4.17 shows the sidereal time in relation to the J2000 frame. Point Q in the illustration corresponds to the +X axis in Figure 4.1 and Figure 4.16 since the simulation is using a fixed inertial frame rather than a rotating frame as shown in Figure 4.17.



$$[\mathbf{Q}]_{Xx} = \begin{bmatrix} -\sin\theta & \cos\theta & 0 \\ -\sin\phi\cos\theta & -\sin\phi\sin\theta & \cos\phi \\ \cos\phi\cos\theta & \cos\phi\sin\theta & \sin\phi \end{bmatrix}$$

Figure 4.16 ENZ and ECI Reference Frames [22]

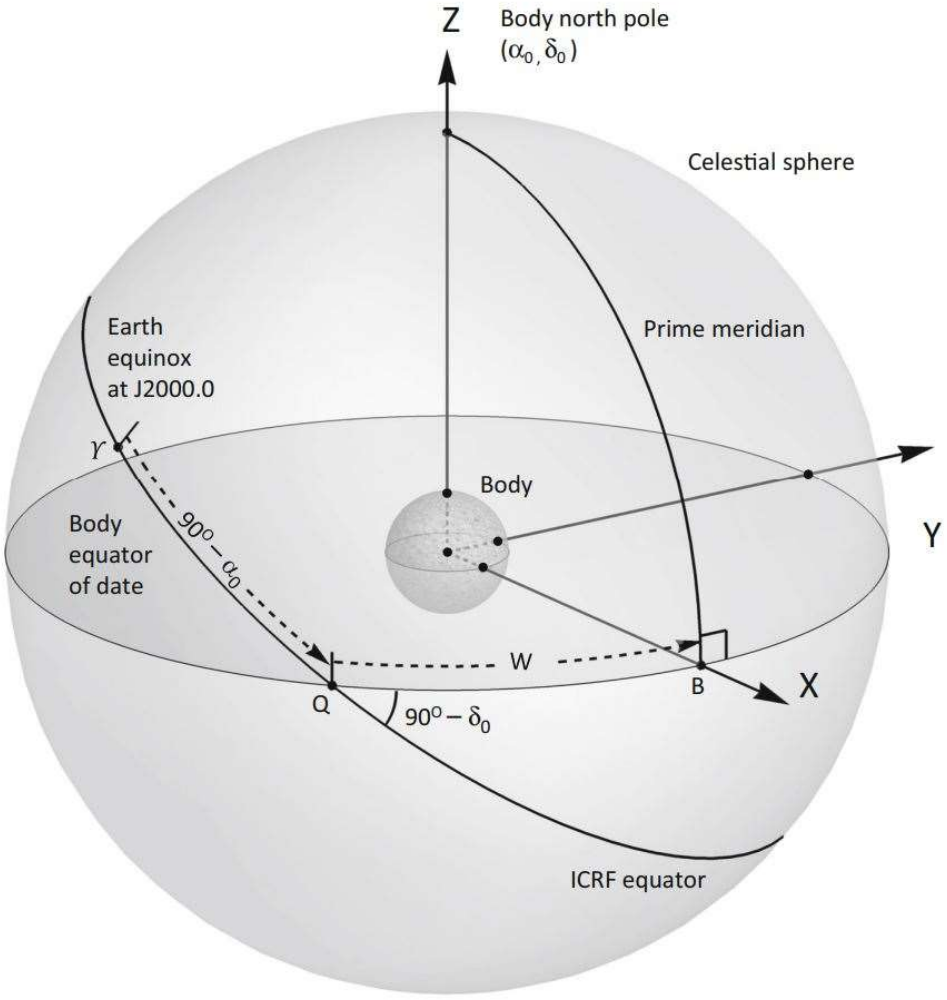


Fig. 1 Reference system used to define orientation of the planets and their satellites. For $\dot{W}(t) > 0$, body rotation is prograde (e.g., Mercury, Jupiter). For $\dot{W}(t) < 0$, body rotation is retrograde (e.g., Venus, Uranus)

Figure 4.17 Reference System for Time Dependent Planet Orientation [30]

Matrix rotations between frames need to be carried out carefully as matrix multiplication is not commutative. To obtain the rotation matrix between the enz frame and inertial velocity frame (Figure 4.18), a rotation about the flight path angle and azimuth is performed to derive Q_{enz}. The convention is switched due to 0° Az corresponding with due north Y+ in the enz frame which results in a row swap in the matrix.

$$\boldsymbol{v_{enz}} = Q_{enz}\boldsymbol{v_{vfi}} \tag{4.21}$$

$$Q_{enz} = \begin{bmatrix} cos(fpa)\sin(Az) & -\cos(Az) & -sin(fpa)\sin(Az) \\ cos(fpa)\cos(Az) & \sin(Az) & -sin(fpa)\cos(Az) \\ \sin(fpa) & 0 & \cos(fpa) \end{bmatrix} \tag{4.22}$$

To get from the inertial velocity frame to the relative velocity frame, another series of rotations is necessary about α and β, similar to the transform used in the panel aerodynamics (eq. 4.17). To perform a complete transformation to the ECI frame from the relative velocity frame, 4.24 is required. This transform is necessary as the aerodynamic coefficients C_D, C_Z, and C_L are expressed relative to the velocity vector. The vehicle body frame coefficients (C_A, C_S, and C_N) are important for a 6DOF simulation but in this case would simply add another transform.

$$Q_{vfi} = \begin{bmatrix} cos(\beta)\cos(\alpha) & \sin(\alpha) & cos(\beta)\sin(\alpha) \\ sin(\beta)\cos(\alpha) & cos(\beta) & -sin(\beta)\sin(\alpha) \\ -\sin(\alpha) & 0 & \cos(\alpha) \end{bmatrix} \tag{4.23}$$

$$\boldsymbol{v_{eci}} = Q_{Xx}Q_{enz}Q_{vfi}\boldsymbol{v_{vfr}} \tag{4.24}$$

$$\beta_{eff} = sin^{-1}(v_{vfr}\hat{\boldsymbol{y}}) \tag{4.25}$$

$$\alpha_{eff} = \cos^{-1}\left(\frac{v_{vfr}\hat{\boldsymbol{x}}}{\cos(\beta_{eff})}\right) \tag{4.26}$$

Figure 4.18 shows the simulation output with all forces on the vehicle shown as vectors. ECI is the equatorial centered inertial frame that the simulation runs on, VFi is the inertial velocity frame where +x is the direction of travel. The conditions were artificially set to exaggerate the effects of the winds for illustration purposes. To calculate the net perturbation vector on the vehicle which considers lift, drag, side force, and relative velocity eq. 4.5 and 4.6 can be expanded to produce the net perturbation vector in eq. 4.28. Due the relative velocity, $C_D(\alpha,\beta)$, $C_S(\alpha,\beta)$, $C_L(\alpha,\beta)$ are recalculated at each time step which adds computation time, a fixed L/D mode option was added that neglects these effects for speed. Eq. 4.27 represents the relative velocity vector in the ECI frame. The three wind components, east to west (eww), north

to south (nsw), and vertical (vw) are expressed in the ENZ frame from GRAM and are converted to the ECI frame with the rotation matrix in Figure 4.16.

$$\boldsymbol{v_{rel}} = \boldsymbol{V_{eci}} - \underbrace{\begin{bmatrix} 0 \\ 0 \\ \omega \end{bmatrix} \times \boldsymbol{R_{eci}}}_{Coriolis\ Effect} - \underbrace{Q_{Xx} \begin{bmatrix} -eww \\ -nsw \\ vw \end{bmatrix}}_{Winds} \tag{4.27}$$

$$\boldsymbol{P_{vfr}} = \frac{1}{2}\rho_{\infty}|\boldsymbol{v_{rel}}|^2 \begin{bmatrix} -C_D \\ C_Z \\ C_L \end{bmatrix} \tag{4.28}$$

$$\boldsymbol{P_{eci}} = Q_{Xx}Q_{enz}Q_{vfi}\boldsymbol{P_{vfr}} \tag{4.29}$$

$$\ddot{\boldsymbol{r}} = \frac{\mu}{r^3}\boldsymbol{r} + \boldsymbol{P_{eci}} \tag{4.30}$$

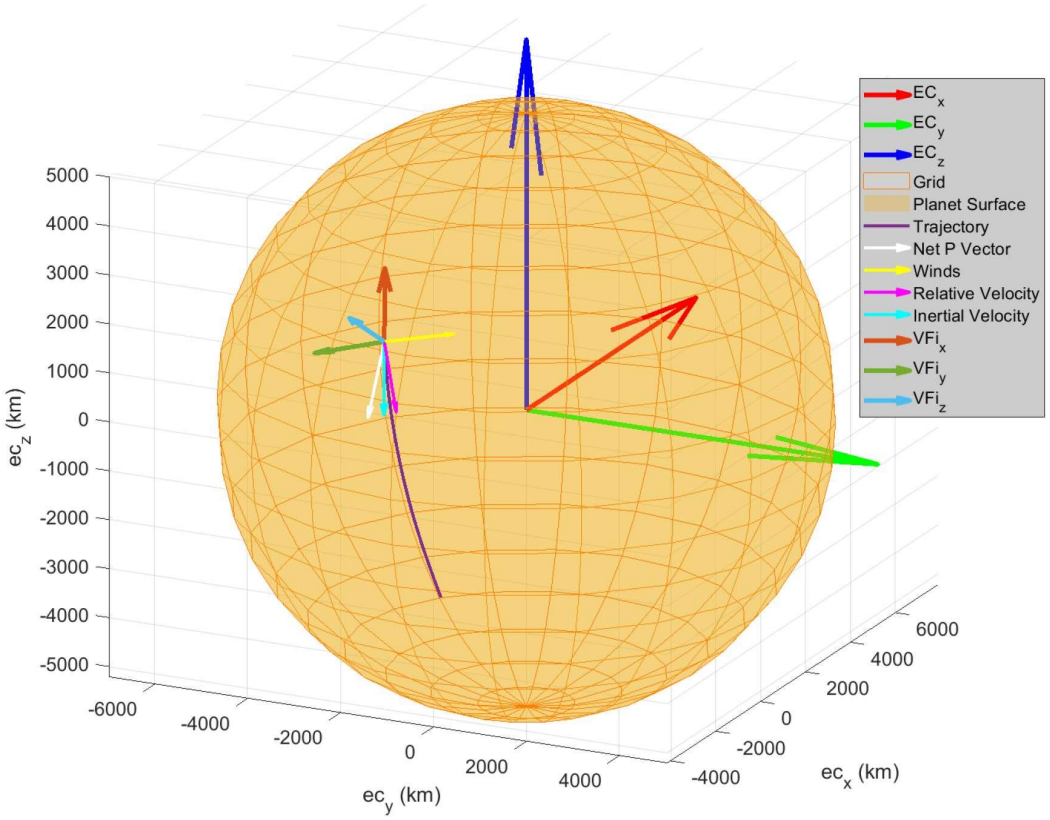


Figure 4.18 Inertial Velocity Frame and Forces on Entry Vehicle

4.3. Aerodynamic Heating and TPS Sizing

Aerodynamic heating at atmospheric entry velocities is a complex modeling problem. The state of the art is highly parallelized CFD with extensive chemistry and radiation models as well as Direct Simulation Monte Carlo (DSMC) for non-continuum flows. At the opposite end of the fidelity scale there are many first order approximate calculations for stagnation heat flux, radiative heating, and shock temperature based on a few key inputs like the vehicle radius of curvature, velocity, and atmospheric chemistry. These tools will be the primary means of determining the heating environment during the aerocapture atmospheric flight phase which will in turn drive the TPS material type and sizing.

4.3.1. Sutton Graves Approximation

$$q_s = k\left(\frac{\rho}{R_n}\right)^{\frac{1}{2}} V^3 \tag{4.31}$$

One method of approximating convective heating, k is a constant that is derived for the planetary body and R_n is the effective nose radius. This approximation will be the first tool used based on the vehicle state vector to approximate heating. The Sutton graves formula can be expanded into the generalized chapman method which incorporates additional exponents and a hot wall correction [1]. Modern, higher fidelity correlations have been derived from CFD runs and can be tailored for an individual planetary atmosphere, see [37], [38], and [39].

4.3.2. Normal Shock Wave Calculation for Thermochemical Equilibrium.

For hypersonic flow, the typical isentropic relations must be expanded to account for chemical reactions. If the concentrations of each chemical species in the atmosphere are known, equations 4.32-4.33 can be used to calculate the temperature and pressure just behind the shock at the stagnation point [17]. These properties are useful for TPS sizing at the vehicle stagnation point and can be scaled for other surface locations. Expanded relations exist for conical flow. To solve for the species enthalpy, thermodynamic table lookups or curve fit approximations must be performed [3]-[5]. Thermodynamic properties such as species entropy and enthalpy and are used to determine the Gibbs free energy and subsequent equilibrium constants, this allows for the calculation of partial pressures, mole fractions, and enthalpy of each species (eqns. 4.34-4.39). The species assumed to be in the flight environment on Venus are CO2, CO, N2, O2, N, O, and their respective ions [7]. These mass fractions can change drastically within the aerocapture flight corridor (Figure 4.10) so an accurate atmospheric model is necessary. The Uranus atmosphere is much simpler containing the species H2, He, H, H+, He+, and e- [6]. The mole fraction variations in the altitude flight corridor aren't as significant as Venus so it's possible that the calorically perfect case of H2/He, γ=1.45 is a good enough approximation.

$$p_2 = p_1 + \rho_1 u_1^2\left(1 - \frac{\rho_1}{\rho_2}\right) \tag{4.32}$$

$$h_2 = h_1 + \frac{u_1^2}{2}\left[1 - \left(\frac{\rho_1}{\rho_2}\right)^2\right] \tag{4.33}$$

$$G_i = H_i - TS_i \tag{4.34}$$

$$\Delta G^{p=1} \equiv \sum_i v_i G_i^{p_i=1} \tag{4.35}$$

$$\prod_i p_i^{v_i} = \mathrm{e}^{-\frac{\Delta G^{p=1}}{\mathcal{R}T}} \tag{4.36}$$

$$K_p(T) = \prod_i p_i^{v_i} \tag{4.37}$$

$$p_t = \sum_i p_i \tag{4.38}$$

$$h = \sum_i c_i h_i = \sum_i \eta_i H_i \tag{4.39}$$

As a proof of concept, a basic chemistry model for air was developed using only the species $N_2, O_2, O, N,$ and NO. The three reactions that are modeled are shown below.

$$N_2 \rightleftharpoons 2N \qquad O_2 \rightleftharpoons 2N \qquad O + N \rightleftharpoons NO$$

$$K_{p1} = \frac{P_N^2}{P_{N_2}} \tag{4.40}$$

$$K_{p2} = \frac{P_O^2}{P_{O_2}} \tag{4.41}$$

$$K_{p3} = \frac{P_N P_O}{P_{NO}} \tag{4.42}$$

$$p_t = P_{N_2} + P_{O_2} + P_N + P_O + P_{NO} \tag{4.43}$$

$$MR = \frac{2P_{N_2} + P_N + P_{NO}}{2P_{O_2} + P_O + P_{NO}} \tag{4.44}$$

Utilizing equation 4.37, equilibrium constant balance equations can be constructed based on the stoichiometric ratios of the reactants and products. Equation. 4.38 can be used to construct a pressure balance constraint and the known initial ratio of nitrogen and oxygen atoms can be used to construct a mole balance constraint. Equations 4.40-4.44 represent a system of 5 nonlinear equations with 5 unknowns. These can be solved independently through a numerical method such as newton Raphson or successive substitution, however the system is sensitive to how the equations are represented. When one or more chemical species are sufficiently low in concentration, it's not uncommon for the equilibrium constant to be in the realm of 1e-16 or less which can cause instability in the numerical method. The most robust solution is to reduce the system so that only the species that are expected to be present in the highest concentrations are solved for. It is possible to compile 4.40-4.44 into one singular equation as a function of N, N2, O2, or O through a symbolic math engine such as Mathematica or the MATLAB symbolic toolbox. N2 was solved for lower temperature conditions and N was used for high temperature, a cutoff of 5000-6000 K works well. Once the composition equations are derived the thermodynamic properties of each species must be obtained through an empirical data source. The database contained in [4] was used for this initial proof of concept chemistry model and specifies 7th order polynomial approximations listed in 4.45 and 4.46. A simple validation of the proof-of-concept chemistry model was performed by computing the mole fractions of each of the constituents from 1000K to 9000 K. Comparison with a plot in [17] indicates strong agreement.

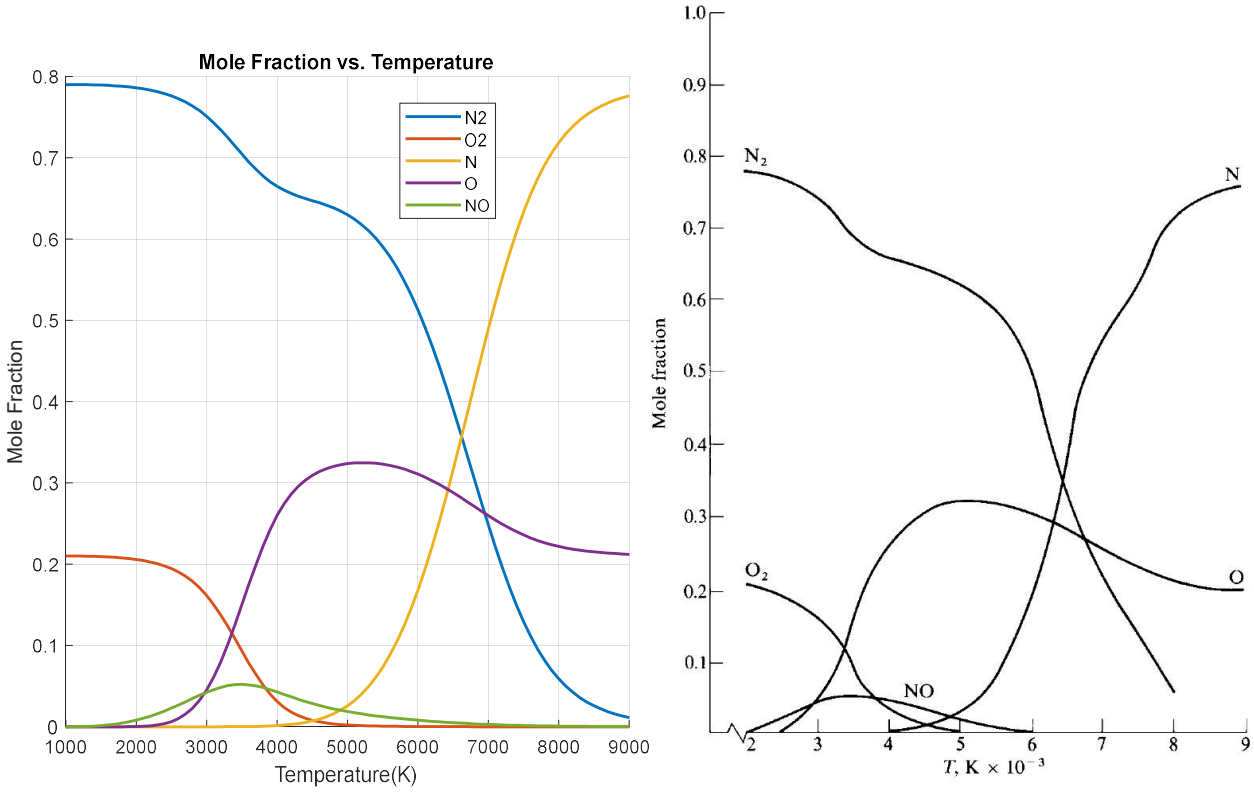


Figure 4.19 Chemistry Model Validation ([17], Fig 11.12, P.541)

$$\frac{H_T^o}{\mathcal{R}T} = a_1 + \frac{a_2}{2}T + \frac{a_3}{3}T^2 + \frac{a_4}{4}T^3 + \frac{a_5}{5}T^4 + \frac{a_6}{T} \tag{4.45}$$

$$\frac{S_T^o}{\mathcal{R}} = a_1 \ln T + a_2 T + \frac{a_3}{2}T^2 + \frac{a_4}{3}T^3 + \frac{a_5}{4}T^4 + a_7 \tag{4.46}$$

The properties contained in [3] are much more extensive than [4] and are used in the industry standard CEA (Chemical Equilibrium Applications) software. An expanded chemistry model utilizing [3] was developed for fast calculations at each timestep in the trajectory code to improve the aerothermal environment predictions. The expanded polynomial curve fits specified by [3] are shown in eqns. 4.47-4.49 and utilize 9 coefficients. A database was generated from [3] so that the 9 coefficients could be queried for any chemical species used in an analysis. The coefficients are provided over three temperature ranges with cutoffs at 1000K and 6000K. A normal shock wave solver was programmed with the same chemical species equations listed in eqns. 4.40-4.44 using the symbolic math method. CEA utilizes a scheme described in [5] that iteratively minimizes the Gibbs free energy term for each reaction, this method is extremely versatile and robust for any number of chemical species and reactions though is computationally expensive for the inner loop of a trajectory program. The NASA Ames developed TRAJ trajectory program utilizes a pre-generated Mollier diagram to determine the equilibrium thermodynamic state behind the shock at the stagnation point. The approach taken here falls in between the above two approaches where the exact expressions for the fixed quantity of species in the atmosphere are manipulated through symbolic math to allow the fastest possible numerical convergence for only a single species.

$$\frac{C_P^o(T)}{\mathcal{R}} = a_1 T^{-2} + a_2 T^{-1} + a_3 + \mathrm{a}_4 T + a_5 T^2 + a_6 T^3 + a_7 T^4 \tag{4.47}$$

$$\frac{H_T^o(T)}{\mathcal{R}T} = -a_1 T^{-2} + \frac{\mathrm{a}_2 lnT}{T} + a_3 + \frac{a_4}{2}T + \frac{a_5}{3}T^2 + \frac{a_6}{4}T^3 + \frac{a_7}{5}T^4 + \frac{b_1}{T} \tag{4.48}$$

$$\frac{S_T^o(T)}{\mathcal{R}} = -\frac{a_1 T^{-2}}{2} - a_2 T^{-1} + a_3 lnT + a_4 T + \frac{a_5}{2}T^2 + \frac{a_6}{3}T^3 + \frac{a_7}{4}T^4 + b_2 \tag{4.49}$$

$$P_{N_2} = f(K_{p1}, P_N) \tag{4.50}$$

$$P_{O_2} = f(K_{p2}, P_O) \tag{4.51}$$

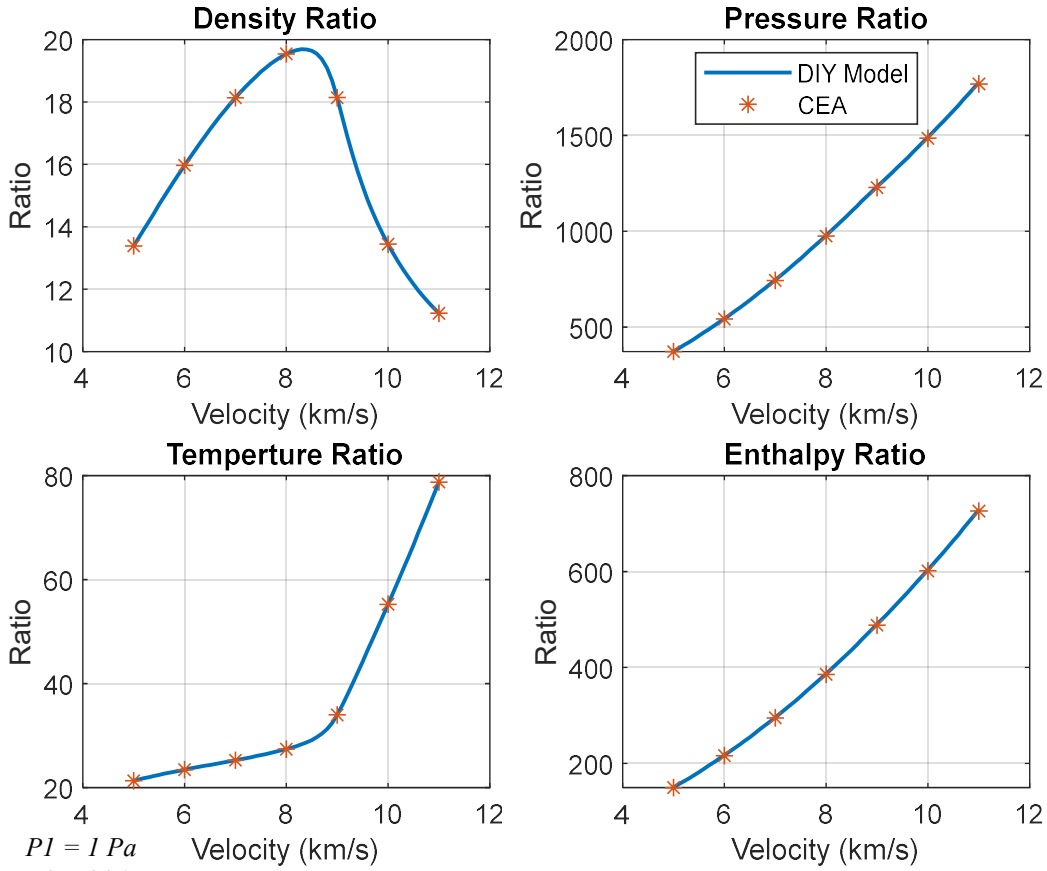
$$P_{NO} = f(K_{p3}, P_N, P_O) \tag{4.52}$$

Equations 4.50-4.52 are formed algebraically from 4.40-4.42 and are plugged into 4.43 to generate an expression that is only a function of P_N and P_O, this expression can then be solved symbolically for P_O, the intent is to isolate P_N which is expected to be the species of the highest concentration at high temperatures, the same reduction scheme can be performed for P_{N_2} for lower temperatures. 4.50-4.54 are all substituted into 4.44 to form an expression where P_N is the only unknown. This expression (4.55) can be solved efficiently with a minimization function like MATLAB's fzero function and the remaining species concentrations can be determined through simple expression evaluations. An additional increase in computation efficiency could be achieved if 4.55 could be solved for P_N, however due to the various root and exponential terms such a solution is not practical to reach with a symbolic math engine.

$$P_t = f\left(K_{p1}, K_{p2}, K_{p3}, P_N, P_O\right) \tag{4.53}$$

$$P_O = f\left(K_{p1}, K_{p2}, K_{p3}, P_N, P_t\right) \tag{4.54}$$

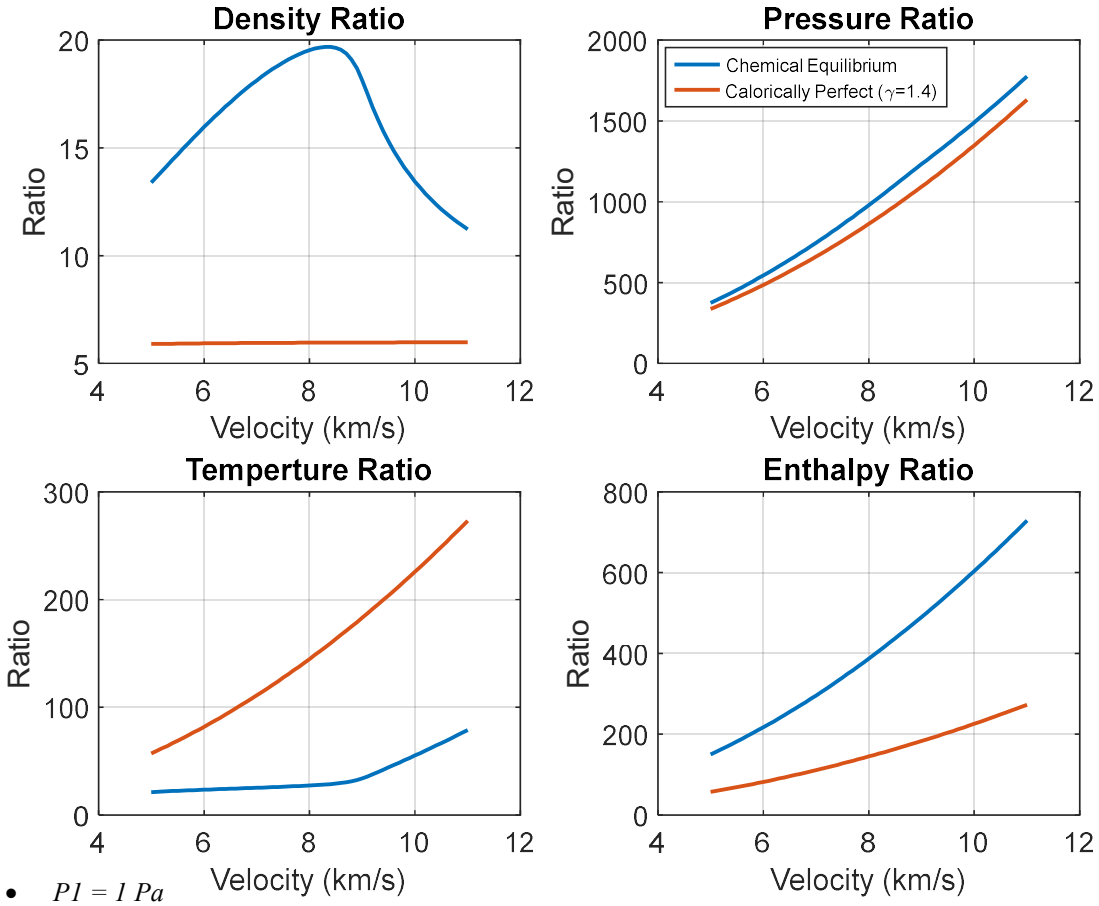
$$f_{P_N} = f\left(K_{p1}, K_{p2}, K_{p3}, P_N, P_t, MR\right) \tag{4.55}$$



- *P1 = 1 Pa*
- *T1 = 216.4 K*
- *0.79 mole N2*
- *0.21 mole O2*

Figure 4.20 Chemistry Model Validation with CEA

A validation case was prepared with CEA using conditions chosen to match an Earth altitude of 80 km. As shown in Figure 4.20, the agreement of the symbolic math model with CEA is excellent. All output parameters were tested and errors between the two models are 1e-4 or less for all property ratios and species concentrations. Figure 4.21 shows a comparison with the calorically perfect gas case which illustrates the importance of chemical reaction effects with high speed, hypersonic applications. Note that the pressure ratio is not strongly dependent on chemical effects as it is a "mechanical" property rather than a thermodynamically driven property. The current model still makes the significant assumption that the gas mixture is in thermochemical equilibrium just behind the shock, in reality equilibrium takes a set amount of time to reach. State of the art CFD solvers such as DPLR model non-equilibrium chemical effects which are important in determining the radiative heating environment.



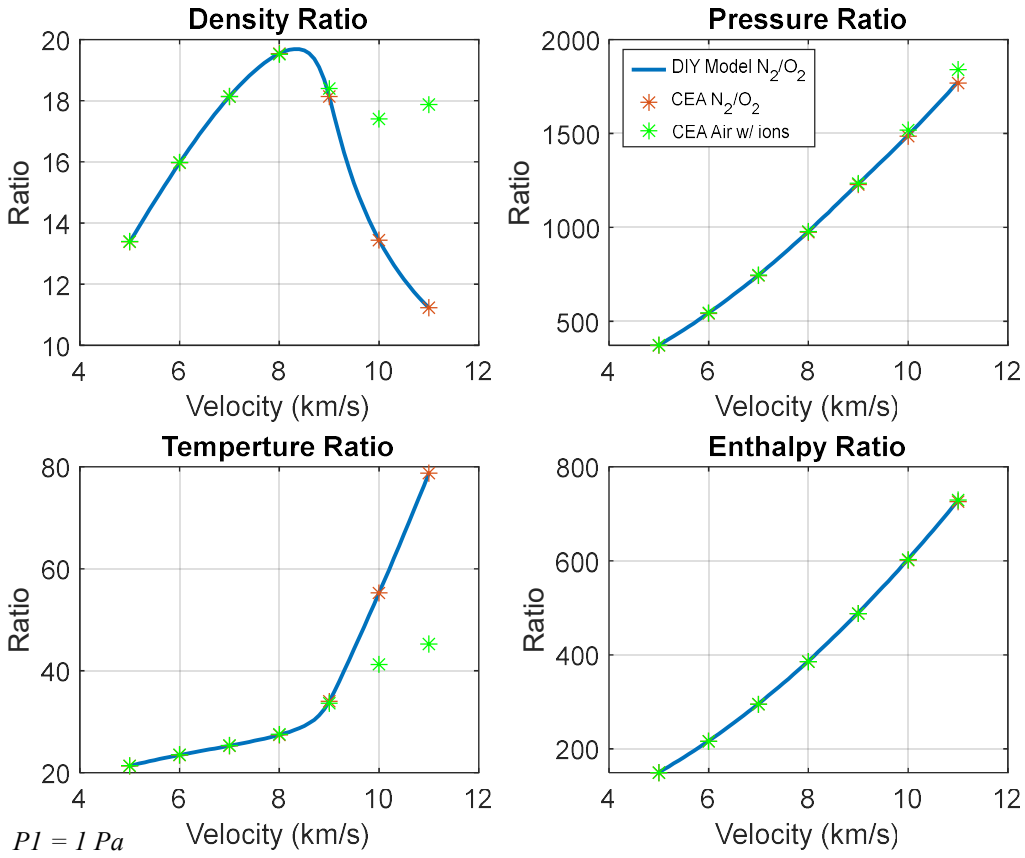
- *P1 = 1 Pa*
- *T1 = 216.4 K*
- *0.79 mole N2*
- *0.21 mole O2*

Figure 4.21 Calorically Perfect Gas Comparison

Another assumption at this stage is that ionization reactions are neglected. At high temperatures, gasses can ionize or lose electrons, in air the first dominant ionization reaction that begins to occur at higher temperatures is $O \rightleftharpoons O_+ + e_-$. To include ionization effects, an additional charge balance constraint equation must be added to the set of equilibrium relations. a_{ei} is the excess or deficiency of electrons, so $a_{ei} = -1$ for O_+, +1 for e_-, and +2 for a species like N_{++}, in other words it is the level of ionizations that have occurred.

$$\sum_{i=1}^{NG} X_i a_{ei} = 0 \tag{4.56}$$

The case in Figure 4.20 was re-run with the inclusion of ions and the CEA default composition for air, which models Argon, CO2 and other trace species. This is to evaluate the level of fidelity of a simple N_2, O_2 model. After around 9 km/s, the density and temperature ratios diverge significantly. It is evident that for Earth entry trajectories at super-orbital velocities such as lunar returns, ionization reactions can have a significant effect on the equilibrium environment.



- *P1 = 1 Pa*
- *T1 = 216.4 K*
- *0.79 mole N2*
- *0.21 mole O2*

Figure 4.22 CEA Validation with Ionization Effects

The inclusion of ionization reactions and effects into the air model represented by equations 4.40-4.44 would significantly increase the complexity of the symbolic math engine calculations. Venus and Mars also have atmospheres of similar complexity with the inclusion of CO2 and its related compounds. The Uranus/Neptune upper atmosphere is almost entirely H_2/He which simplifies the modeling and inclusion of ionization effects. A Uranus aerocapture trajectory was tested as an initial study into the equilibrium chemistry effects on the heating environment, the results are presented in Appendix 9.4.

4.4. Geometry

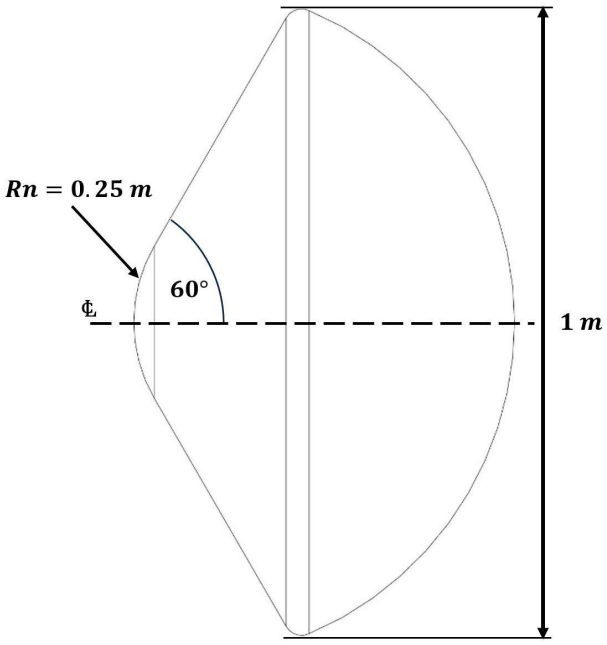


Figure 4.23 Validation Entry Vehicle Geometry (ref. [11])

The entry vehicle for the initial trajectory validations will consist of a standard sphere cone aeroshell with the same forebody geometry specified in [23]. Different geometries will be investigated such as a 70° sphere cone similar to the Mars Science Lab (MSL) vehicle. The forebody geometry is fed into the modified Newtonian aerodynamics calculations discussed in section 4.2 and the effective nose radius is used in the Sutton graves correlation. A scaled up MSL vehicle type was used in ECI study on a single-pass aerocapture approach for outer planets missions [12], [15]. TPS material candidates for the forebody are PICA-D (domestic materials), C-PICA (conformal), and HEEET (Heatshield for Extreme Entry Environment Technology), which is a high performance woven TPS [12]. Material selection and sizing will be determined by the entry environment. The aerothermal analysis and TPS sizing will need to take into consideration the effects of multiple entry pulses and any progressive degradation.

4.5. Simulation Framework

4.5.1. Simulation Version 1.0

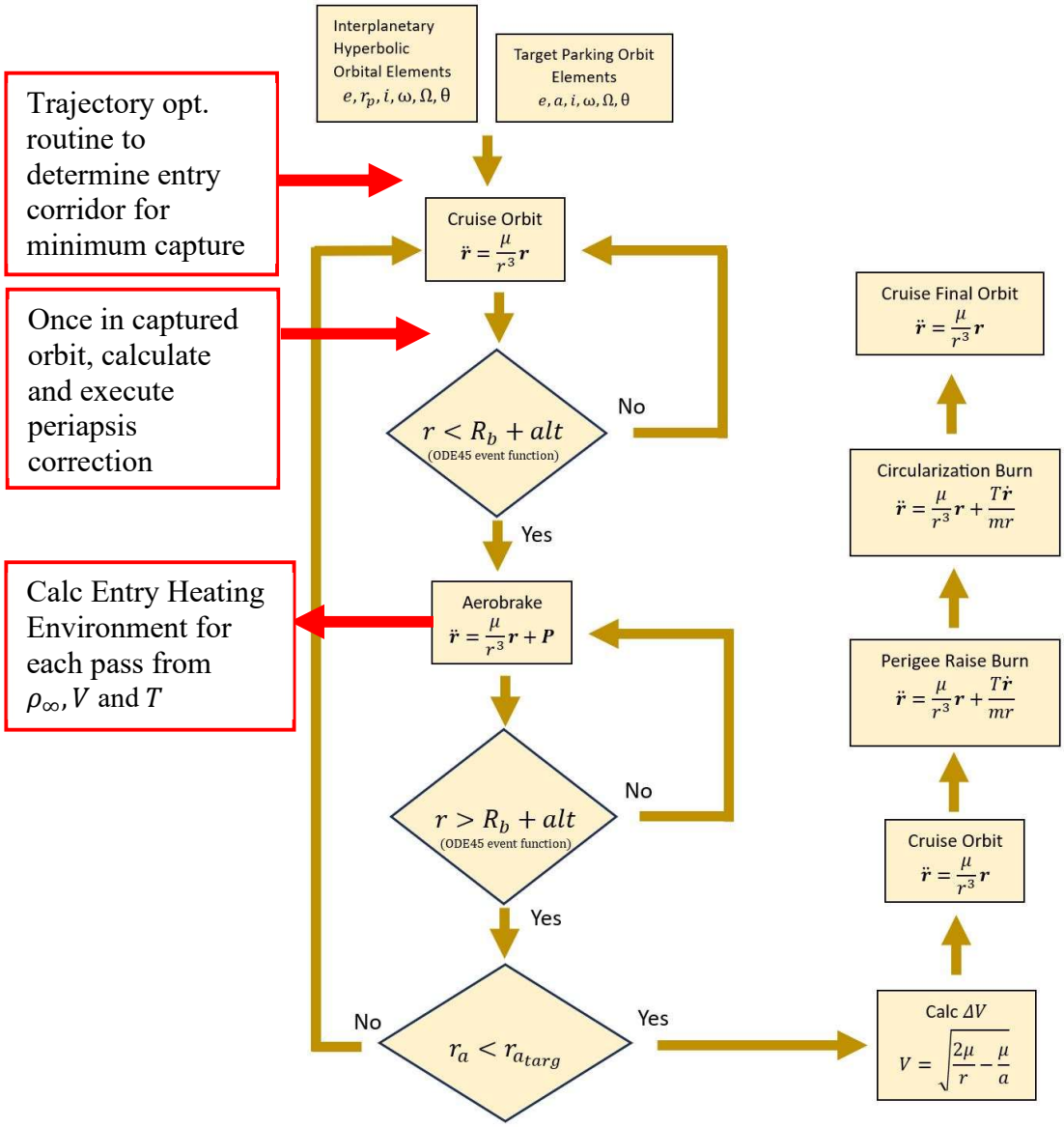


Figure 4.24 Aerobraking Model with Key Additions (red)

A past AE242 Orbital Mechanics project partially modeled an aerobraking trajectory for Earth and Mars cases and is a primary inspiration for this multi-pass aerocapture study. The program was written in MATLAB and has been reworked into an object-oriented format to allow for easier implementation of different calculations like aero heating and trajectory optimization.

The framework was validated with a flight proven trajectory analysis and TPS design tool such as BATSPEED in section 5.2.

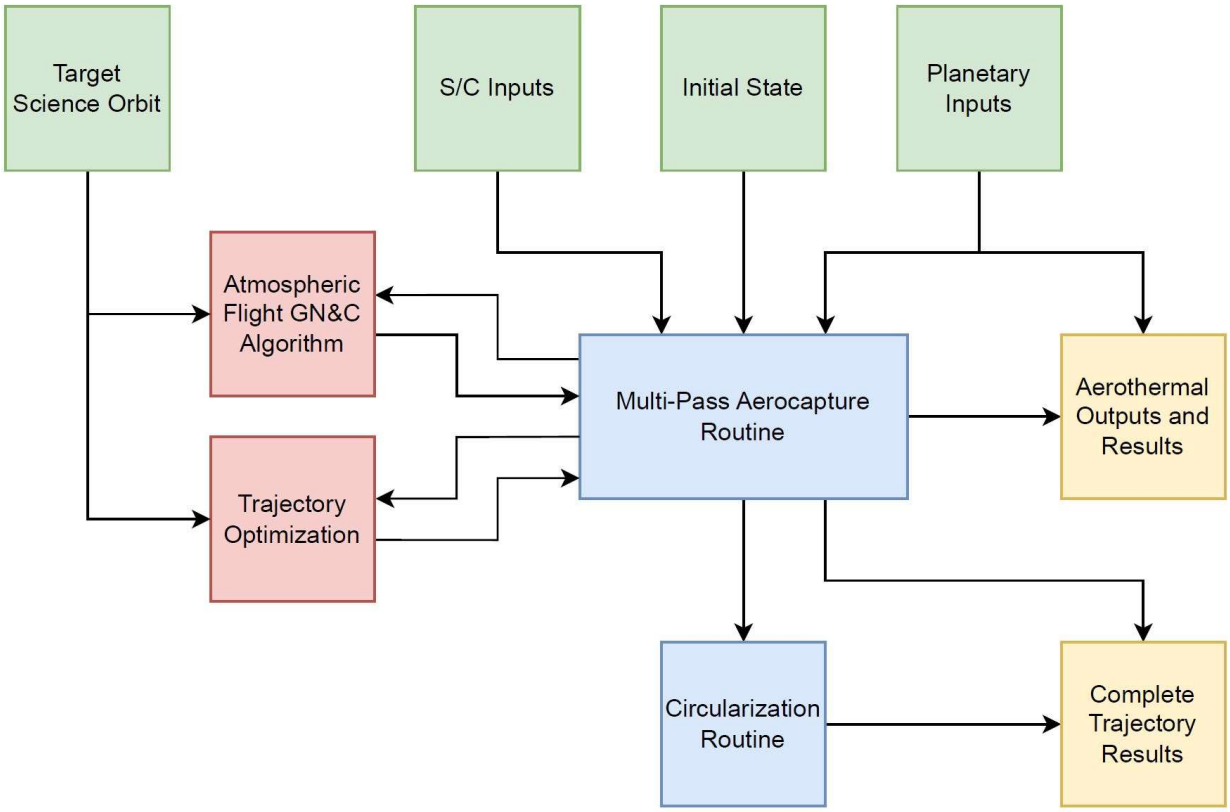


Figure 4.25 Model Framework (Version 1.0)

The simulation framework was developed in an object-oriented approach in MATLAB. The initial intent was to use system object blocks to build out the algorithm in Simulink. This has not been implemented yet as handing large blocks of vectorized data along with numerous other parameters was tedious in Simulink, a traditional scripted approach was used for the initial build of the simulation framework.

The aerocapture and circularization routines contain the core trajectory propagation methods, using a Runge Kutta (RK) 4/5 variable step integrator such as MATLAB's ODE45. Version 2.0 of the simulation allows the user to choose from a set of integrators within MATLAB, ODE89 is especially accurate for long duration smooth orbits. Event functions are implemented to command the integrator stop at critical points like atmospheric interface or apoapsis. Trajectory correction, perigee raise, and other propulsive maneuvers are modeled using eqn. 4.7. Additional methods within the orbit propagator class of objects generate new initial conditions and stitch time history results to previous orbit segments for post processing and visualization. The time history results are passed on to an Aerothermal calculations object.

The trajectory optimization routine is called after the atmospheric exit of each aero-pass. This routine will assess the current vehicle state and simulate a small perigee adjustment burn at apoapsis and the following aero-pass. This "look forward" trajectory will be iterated while

adjusting the maneuver burn time until the desired apoapsis is achieved. The output of this optimization is the ΔV and burn time for an optimal periapsis adjustment, typically on the order of 1-2 m/s. At the beginning of the simulation, an optimization routine determines the optimal number of passes based on the ΔV required for the initial orbital insertion pass. The outputs of this optimization are the number of aero-passes and the target apoapsis after each pass, this data is fed into the periapsis adjust routine.

Extensive literature exists on guidance and control methods for aerocapture ([10], [23], [26], [27]). Popular methods are bank angle modulation (BAM) and direct force control (DFC). One common control algorithm is fully numeric predictor corrector aerocapture guidance (FNPAG). Nominal unguided trajectories will be used for the initial analysis as developing and simulating a complete closed loop guidance system is at the edge of scope for this project. NASA developed tools such as GENESIS [17] already exist and can run 6-DOF simulations utilizing these guidance methods.

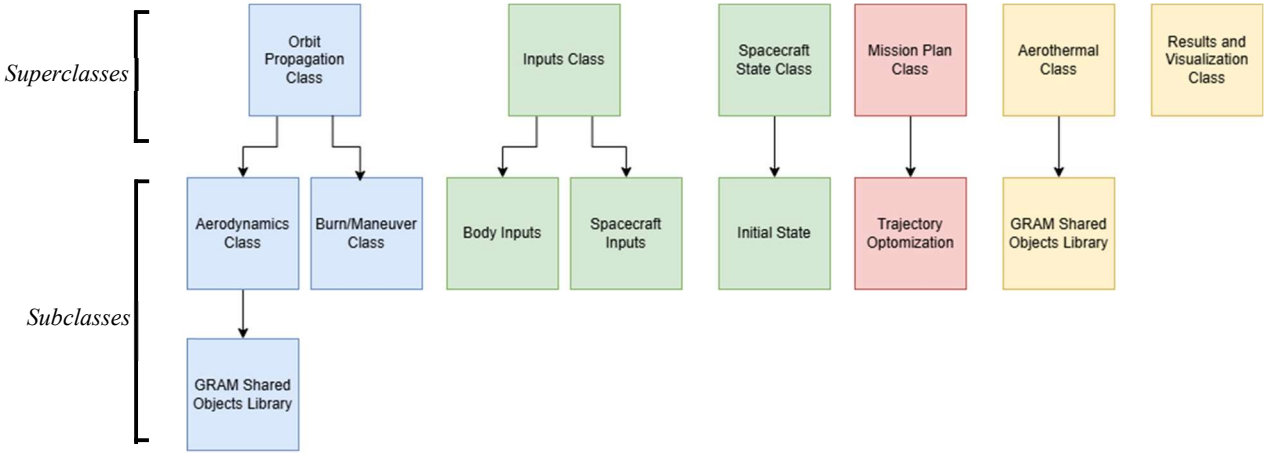


Figure 4.26 MATLAB Object Oriented Class Structure (Version 1.0)

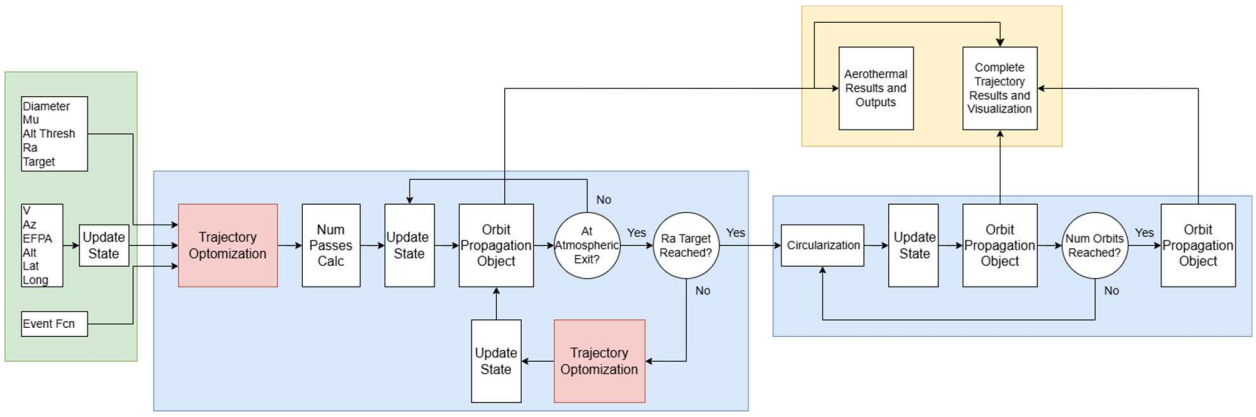


Figure 4.27 Expanded Model with Subsystems (Version 1.0)

4.5.2. Simulation Version 2.0

The first version of the multi-pass trajectory program utilized object oriented programming techniques but did not take full advantage of the various built-in aspects of the class and object formats within MATLAB. The two primary class types are value classes and handle classes. Value classes behave like normal variables where property values are tied to the variable name, if an object of a value class is assigned to a new variable, a new independent object is created. Modifying the properties of the new object do not affect those of the original object. Use of value classes in a simulation architecture requires objects to be passed in and out of functions to be modified and can be limiting for an environment that requires large numbers of parameters and state variables to be in sync at all times. Version 1.0 of the simulation environment utilized value classes for the majority of data management.

MATLAB handle classes can create multiple objects that are references to a single object. A handle object can be copied to new variables and passed into functions or assigned as properties and all instances reference the same underlying object. Any change to properties of a handle object will be reflected in all instances of that object. This behavior enables a massive amount of flexibility in the simulation environment. Handle classes are created by deriving them from the handle superclass.

```
1.      classdef MyHandleClass < handle
2.          ...
3.      end
```

An even more specialized type of handle objects are MATLAB system objects. They are of a handle class by definition but contain various built in features that allow them to be re-used in loops with step and reset functions as well as expanded load and save capabilities. This allows them to be used as system blocks within Simulink however the current simulation has not been implemented in Simulink and is run through a script. All the class definition files in version 2.0 were converted to the matlab system object format. A top level, encapsulating object was created to initialize the simulation and pass various objects as properties to other objects to enable the interconnected nature of handle classes. Once the handle objects were mutually shared, the simulation could be run continuously with all relevant properties like the spacecraft state, geometry, and trajectory results seamlessly shared between objects without having to pass inputs and outputs through the various functions and methods. Handle objects also enable other advanced behavior such as listeners and events that can automatically trigger callback functions when properties are changed. One example of this behavior is when the planet property is updated in an object of the BodyInputs class, the GRAM and time objects are automatically updated with the new planet and the GRAM interface is re-initialized with a new atmosphere.

Table 4.1 Program Class Summary

Shared Handle Classes	
SCInputs	Spacecraft geometry, aerodynamics, and rocket propulsion
BodyInputs	Planetary constants, shape model, atmospheric threshold
SCState	All state parameters, coordinates, frame transformations, etc.
TrajResults	Time history trajectory results, contains labels and names for plotting
timeMgr	Handles elapsed time and all time dependent planet orientations
gramMgr	Contains the GRAM interface library and all shared objects
chemMgr	Handles all atmospheric chemistry and thermodynamic calculations
Trajectory Propagation Classes	
OrbitProp	Superclass for any trajectory propagation, contains the numerical integration scheme and all associated properties
AeroPass	Subclass of OrbitProp for atmospheric flight
Burn	Subclass of OrbitProp for propulsive maneuvers
Post Processing and Visualization Classes	
plotProps	Generates 2D plots and handles all plotting options
Aerothermal	Primary trajectory postprocesser, handles all time history and aerothermal calculations
AerothermalStep	Subclass of aerothermal, performs calculations at one trajectory point
TrajPlot	Generates the 3D trajectory plots and stores all run history results for all previous trajectory segments
Optimization Classes	
MissionPlan	Contains all shooting method trajectory optimization routines, contains specific logic and maneuver calculations for a multi-pass aerocapture type mission
Low Level Helper Classes	
InitState	Initializes default values when a new configuration is created
AeroDB	Constructed by the SCInputs class, handles all aerodynamics calculations
inputSpecies	Contains properties of the chemical species within GRAM
optoIn	Creates and formats inputs to the trajectory optimizer

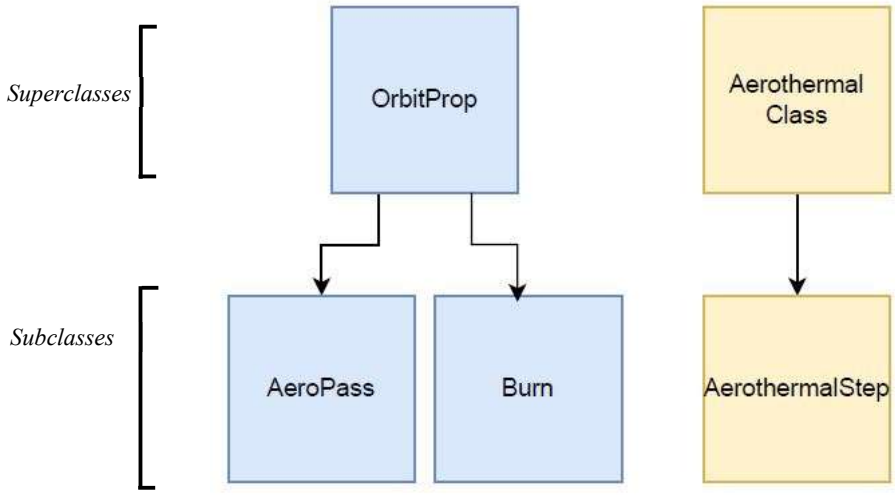


Figure 4.28 Version 2.0 Class Inheritance Hierarchy

In version 2.0, only a few classes utilize inheritance such as the trajectory propagators and aerothermal classes, compared to almost all the value classes in 1.0 (Figure 4.26). This streamlines the overall architecture and prevents subclasses from inheriting excessive properties and methods when most are not needed. Only objects with closely related functionality benefit from a sub-superclass hierarchy. Most of the classes contain other classes as properties so that the same set of simulation data can be readily available for any class method at any point. Without the reference behavior of handle objects, the amount of variables and structs that need to be passed between different classes and functions would be extremely cumbersome.

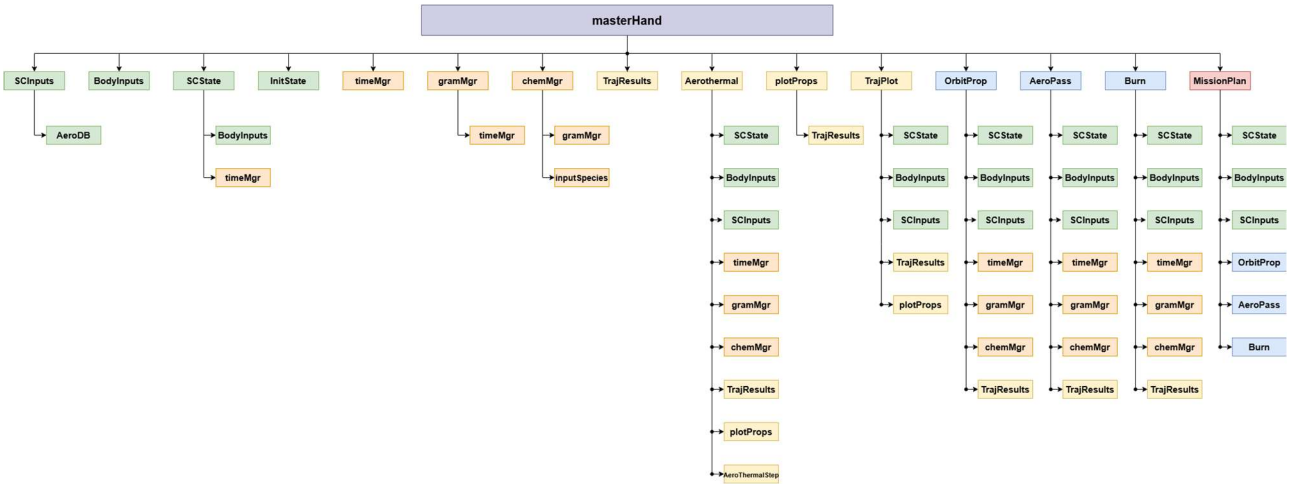


Figure 4.29 Class Containment Structure

To create a new simulation, the master hand constructor is called which initializes all 15 of the primary system objects and creates the connections shown in Figure 4.29. The master hand contains methods such as listener callbacks which handle unique interactions between the system

objects. Once a configuration is created, any instance of a particular handle object such as SCState will always contain the most up to date values as they are all references to the same object. This allows an atmospheric flight trajectory to be run and any following segments such as a coast or burn trajectory will already contain the updated state and initial conditions necessary to run. The same is true with the results as any of the three trajectory objects populate the same results object (TrajResults) which is also accessible from the post processing, aerothermal, and plotting objects. The save and load functions of system objects allow a masterhand object to be seamlessly saved to and loaded from a .MAT file which is useful for organizing different configurations. Lower level objects such as a spacecraft configuration (SCInputs) can also be saved and loaded into a simulation on their own. The mission plan object contains all the trajectory propagation objects to compute shooting trajectory optimizations. The multi pass routine contained within the mission plan still reflects the general functionality shown in Figure 4.27. The State object supports saving previous states as structs to allow resets at the start of another iteration or after an optimization.

A special method was implemented within masterhand using the "assignin" matlab function to flatten the architecture and assign the 15 primary system objects into the matlab base workspace, this makes for easier access to the various properties. There are hundreds of individual properties contained within the various objects, to ease in creating and managing individual missions and simulation setups a property editor was created within the MATLAB app designer. The property editor allows important input properties to be edited and is organized into 5 tabs, State, Spacecraft, Aerodynamics, Planet, and Options. The editor can create, load, and save MAT files which can then be easily loaded into a script running a simulation. A sample script as well as the source code for most of the system objects is contained in Appendix 9.6

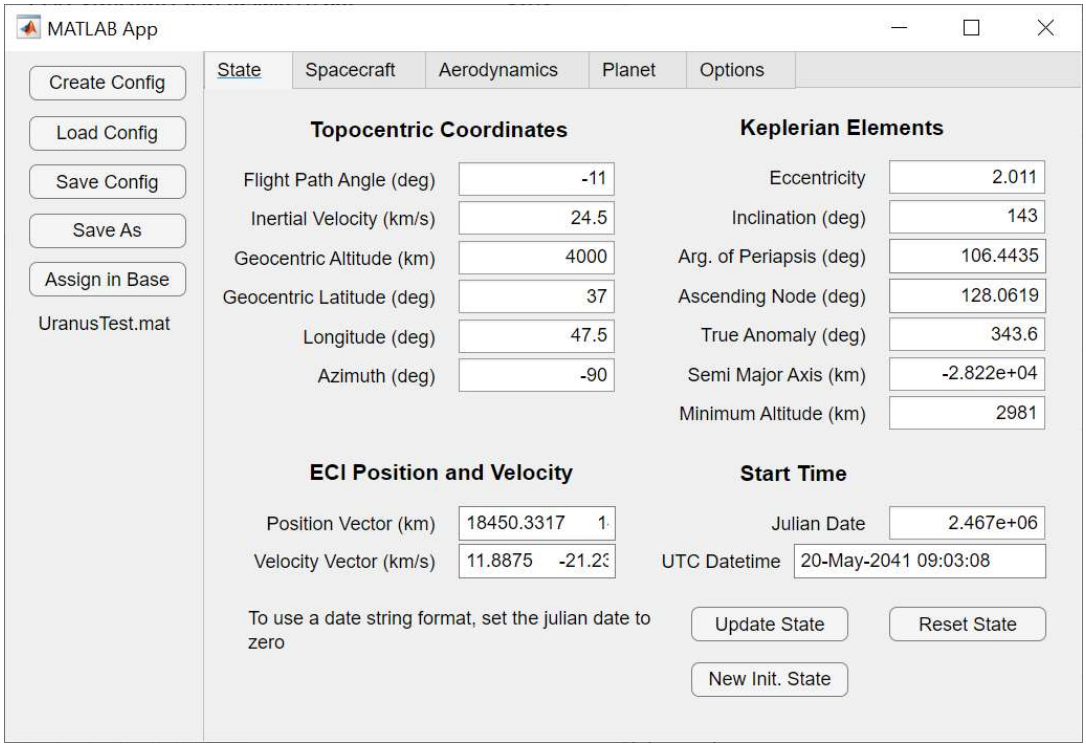


Figure 4.30 Configuration Editor

5. Model Validation and Comparison

5.1. Model Validation with SCITECH Venus Aerocapture Performance Analysis

The simulation methodology outlined in chapter 4 requires a robust validation scheme to verify the physics and modeling methods are sound. Recall reference [23] assesses the performance of various GN&C methods for a smallsat Venus aerocapture and presents preliminary aerothermal environment predictions. To capture the guided bank angle trajectory space in [23], lift-up, lift-down, and bank 90° trajectories were run using the 10° trim angle of attack specified in the paper. The initial conditions are shown in Figure 5.1 and were pulled directly from [23].

Parameter	Value
Velocity	11 km/s
Azimuth	-90°
Flight path angle	-5°
Altitude	150 km
Latitude	0
Longitude	0
Julian date	2455504.0

Figure 5.1 Model Validation Trajectory Inputs [23]

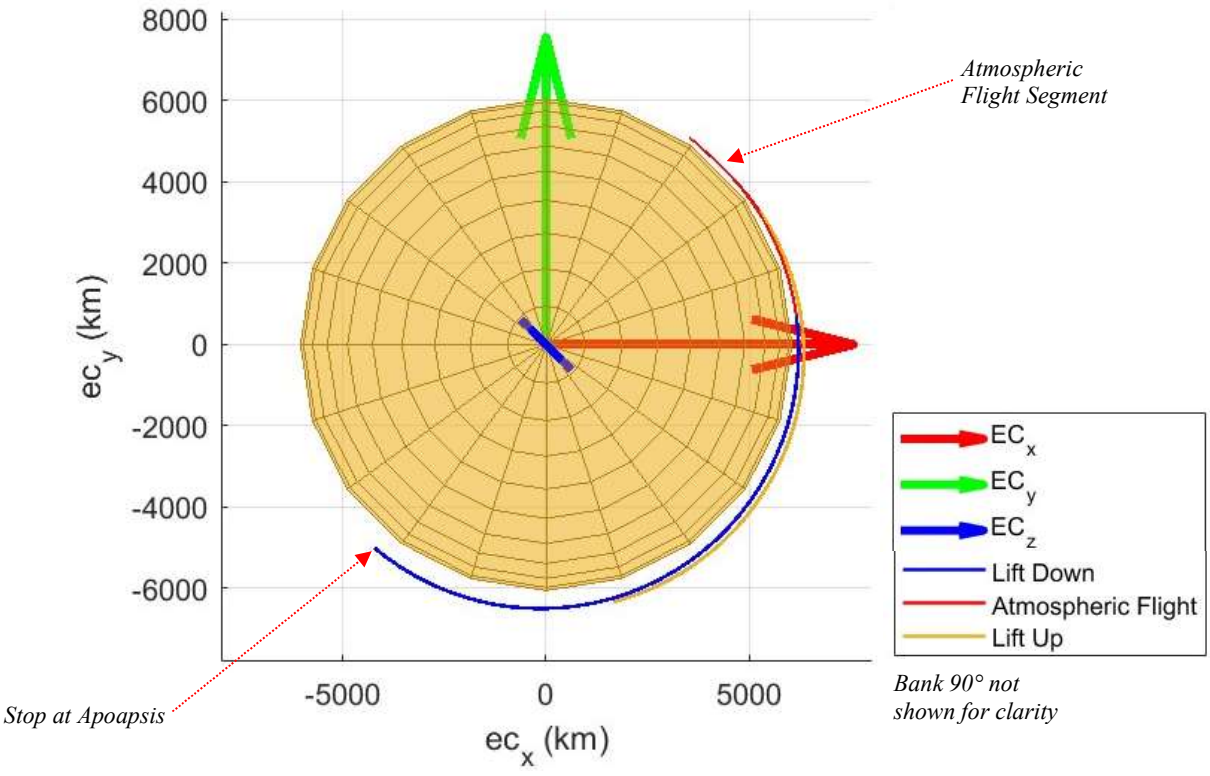


Figure 5.2 Unguided Aerocapture Validation Trajectory

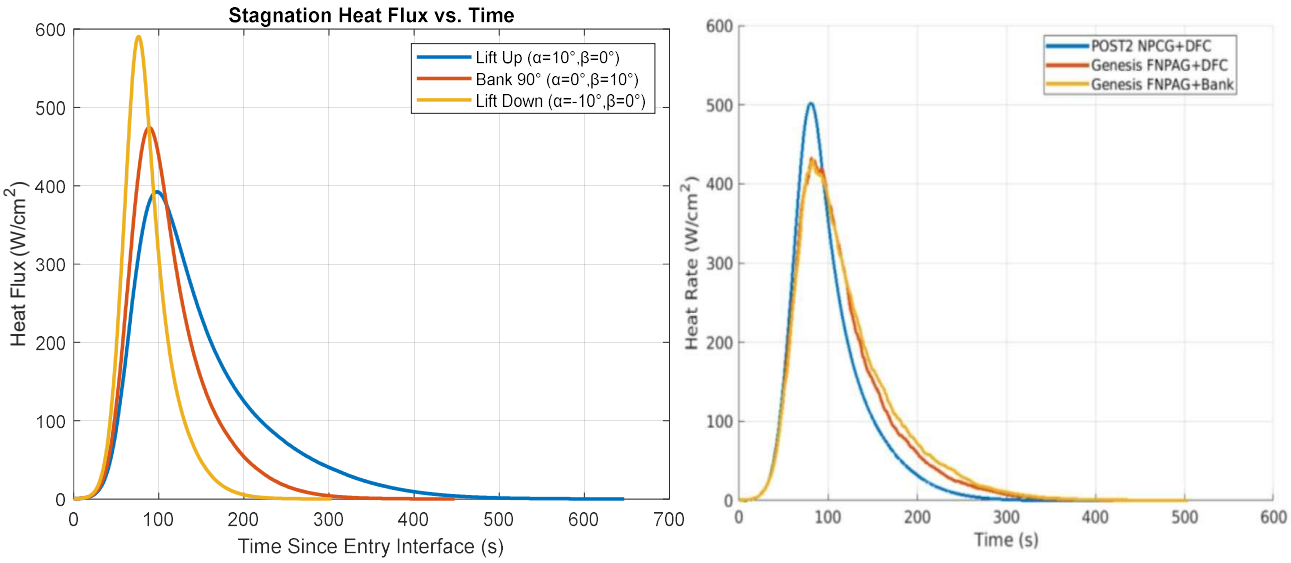


Figure 5.3 Trajectory Heat Flux Validation (Internal Model: left Ref. [23]: right)

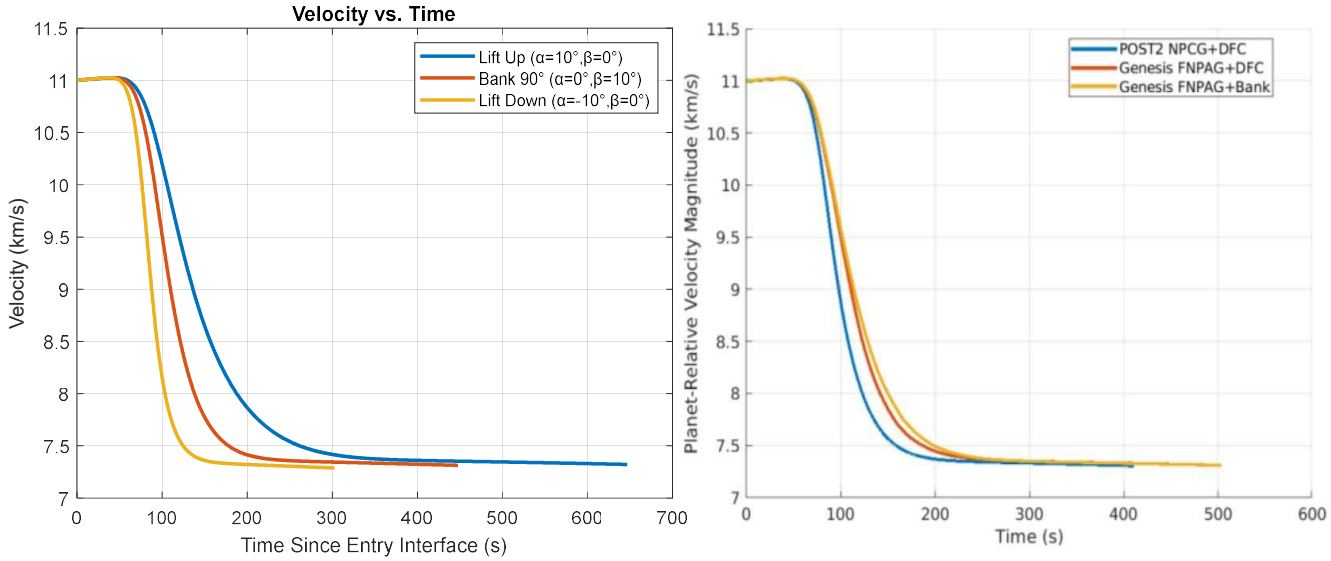


Figure 5.4 Trajectory Velocity Validation (Internal Model: left Ref. [23]: right)

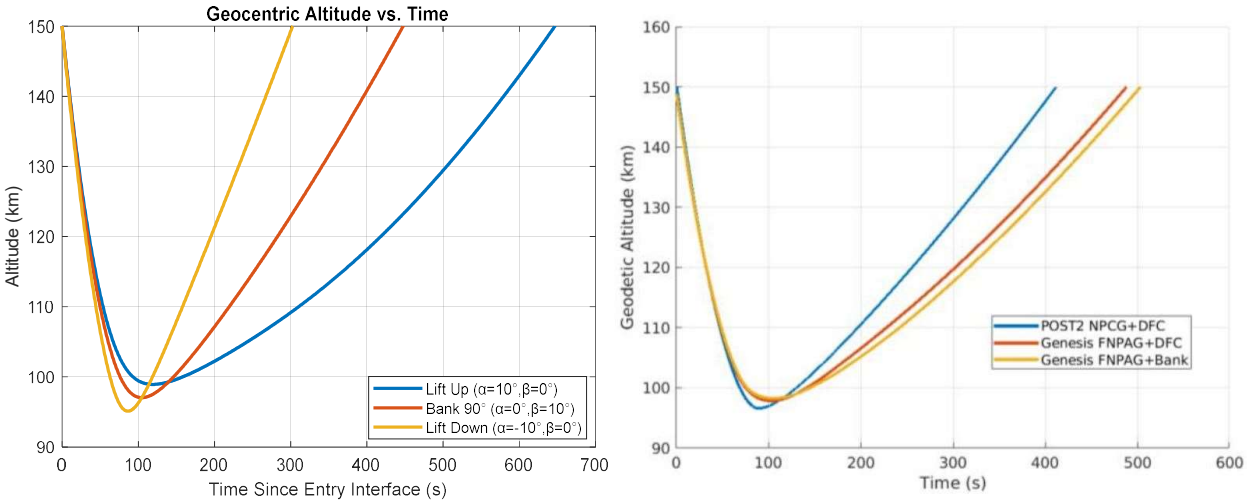


Figure 5.5 Trajectory Altitude Validation (Internal Model: left Ref. [23]: right)

Using the flight path angle as an optimization parameter, the simulation was able to achieve the 500 km target apogee altitude within 100 meters. The trajectory and aerothermal results are bounding of the results from [23] which utilized flight proven software tools. The heat flux and total heat load were calculated using Sutton graves outlined in section 4.3 using the same aerothermal constant provided in [23]. While the current simulation does not model guidance, navigation or 6 degrees of freedom, it is a reasonable 1st order estimate and allows for expansion to new entry conditions.

Table 5.1 Venus Aerocapture Validation Summary

	Peak Conv. Heat Flux (W/cm^2)	Conv. Heat Load (J/cm^2)	ΔV Lost Due to Drag (km/s)	FPA at Entry Interface (deg)
Lift Down	392.29	49653	3.6558	-5.4233
Bank 90°	474.32	39900	3.6304	-5.6182
Lift Up	590.60	32103	3.5535	-5.9737

5.2. Comparison with NASA TRAJ Software

The Entry Systems and Technology Division at the NASA Ames Research Center is the agency's hub for entry systems modeling and TPS materials research. The division has developed numerous software tools over the years to simulate atmospheric entry and model TPS material response. One of the internal tools to the materials branch (TSM) is a code called BATSPEED (Broad A priori TPS Sizing for Proposals and Efficient Engineering Design). This tool combines earlier developed codes TRAJ (Trajectory Analysis Program) with FIAT (Fully Implicit Ablation and Thermal Analysis Program). TRAJ is intended to simulate the entry

trajectory and generate aerothermal environments, almost an analogy to the MATLAB trajectory code developed as part of this project. TRAJ does support skip-out but does not simulate propulsive maneuvers or multiple atmospheric entries. FIAT is the TPS material response tool that reads the entry environment and spits out a required TPS thickness and can recommend TPS material options. BATSPEED combines these two codes in a convenient mission design tool that allows the user to specify an atmospheric entry state and simulate a range of trajectories to generate a bounded TPS design space. TRAJ was used as an initial validation for the MATLAB trajectory tool for multi-pass aerocapture. Future work will involve running FIAT for the entry environments of a single pass and multi pass aerocapture and comparing the resulting TPS thickness of each. BATSPEED is run in a linux ubuntu shell environment and all the various outputs and data are still being explored. Currently, TRAJ is able to simulate an aerocapture trajectory and target a post-capture apoapsis but requires a non-zero angle of attack to generate lift-up and lift-down results. TRAJ has a variety of options for atmosphere models and entry vehicle geometries with many based on empirical flight data. For an initial comparison, TRAJ was run at 0.5° α and the resulting entry flight path angles for the lift up and lift down trajectories should evenly split the 0° α case. The same entry vehicle geometry was used as the comparison with [23] in section 5.1 but the Julian date and longitude and latitude had to be updated as the values in Figure 5.1 were throwing errors in TRAJ.

The results of the MATLAB and TRAJ trajectories are in-family and there are many physics assumptions that differ between the two that likely make up the differences. The MATLAB EFPA result is slightly skewed towards the lift-up TRAJ result rather than splitting the difference. The MATLAB model also overpredicts the heating environment by around 20%, though this could be due to a slightly different aerothermal constant being used in TRAJ. Typically, uncertainties are high with heating predictions and appropriate margins are applied accordingly. Familiarity with the TRAJ and BATSPEED codes is low and there is much to learn for future simulations. Once issue is that when viewing the time history results of the TRAJ lift up trajectory, the vehicle does not actually skip out and falls all the way to the surface. TRAJ uses an iterative method similar to the in-house developed MATLAB script to home in on the EFPA for the post-capture apoapsis. More trial and error is necessary to diagnose this issue.

Table 5.2 MATLAB-TRAJ Comparison Inputs

Inputs	
Velocity	11 km/s
Azimuth	-90°
Longitude	305.61606°
Latitude	5.170641°
Julian Date	2456755
Mass	150 kg
Cone Half Angle	60°
Diameter	1 m
Nose Radius	0.25 m
Target Apoapsis	500 km

Table 5.3 MATLAB-TRAJ Comparison Results

	MATLAB	NASA TRAJ (Lift Up)	NASA TRAJ (Lift down)
Angle of Attack	0°	0.5°	-0.5°
Resulting EFPA	-5.6149	-5.6122	-5.586
Peak Convective Heat Flux	451.641 W/cm^2	386.53 W/cm^2	375.58 W/cm^2
Peak Radiative Heat Flux	-	15.75 W/cm^2	14.94 W/cm^2
Total Heat Load (Conv.)	38394.7 J/cm^2	41498.96 J/cm^2	34019.71 J/cm^2

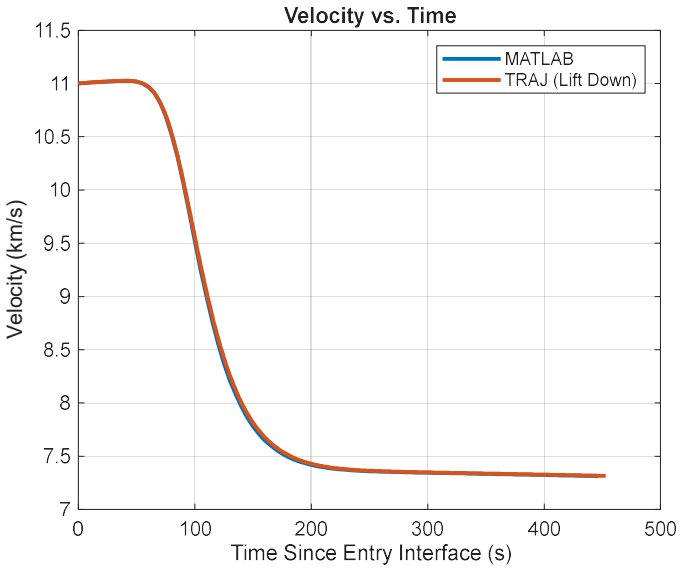


Figure 5.7 TRAJ-MATLAB Velocity Comparison

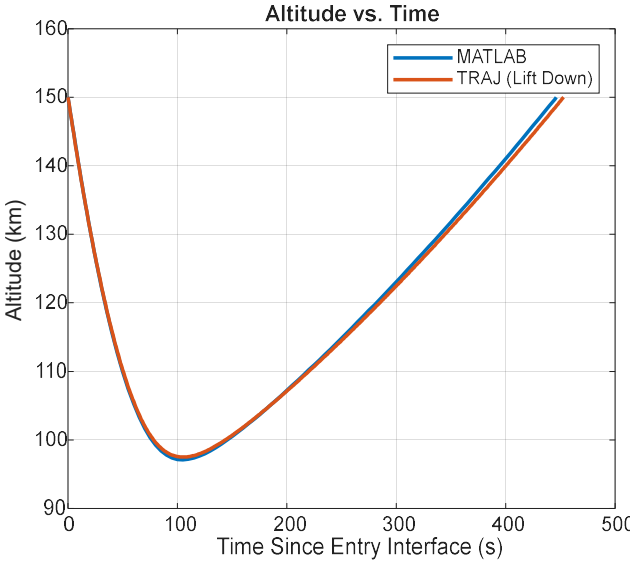


Figure 5.6 TRAJ-MATLAB Altitude Comparison

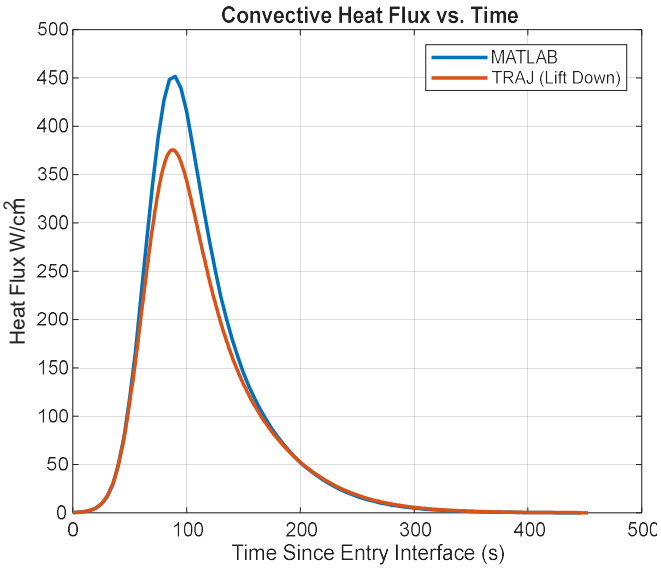


Figure 5.8 TRAJ-MATLAB Heat Flux Comparison

An additional comparison study was conducted with TRAJ with lift up and lift down conditions, the same vehicle configuration and input state were used as Table 5.2 with $\alpha = 10°$, $15°$, and $20°$ targeting a 500 km post aerocapture apoapsis. The error between any of the two resulting entry flight path angles is under 0.02° (Table 5.4) which is less than observed differences from varying the entry long, lat or Julian date. This agreement adds additional confidence to the MATLAB model and is a strong indication that the physics and modeling methods are sound. There are numerous assumptions and modeling methods that are different between the two solvers and the degree of variation seen between the two is expected. Both models match the expected behavior of trading higher total heat load for decreased maximum heat flux for decreasing α. Figure 5.10 and Figure 5.11 illustrates close agreement in the trajectory space in terms of velocity and altitude while the aerothermal environments are around 10-20% higher for the MATLAB model. While TRAJ does offer an option for Venus GRAM in its atmosphere selection, discussion with colleagues suggested that this is an altitude profile that was extracted from a separate run of GRAM and not an individual query of the GRAM model at each time step; this and possibly the aerodynamics model may account for some of the small differences.

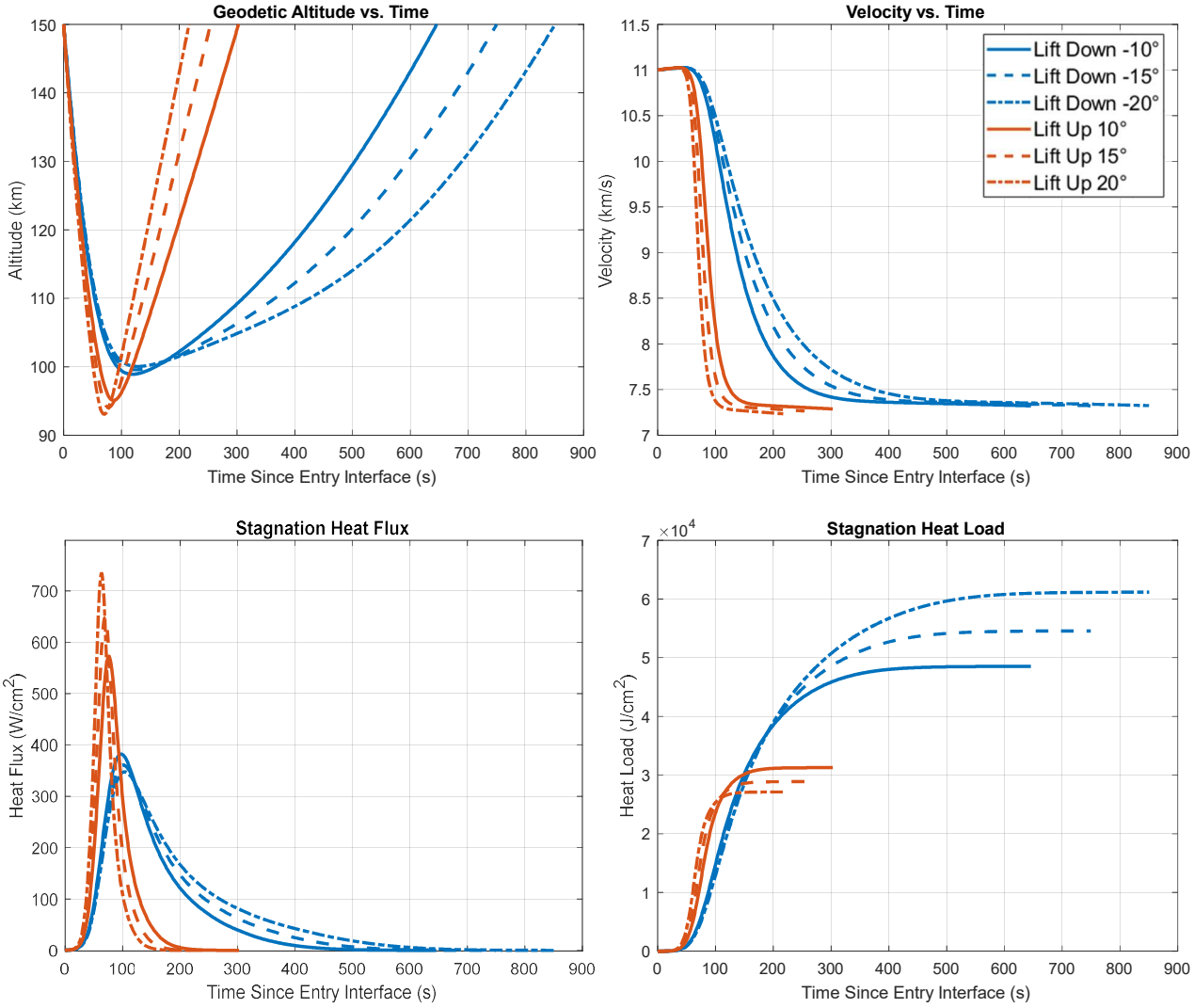


Figure 5.9 Lift Up /Lift Down MATLAB Results

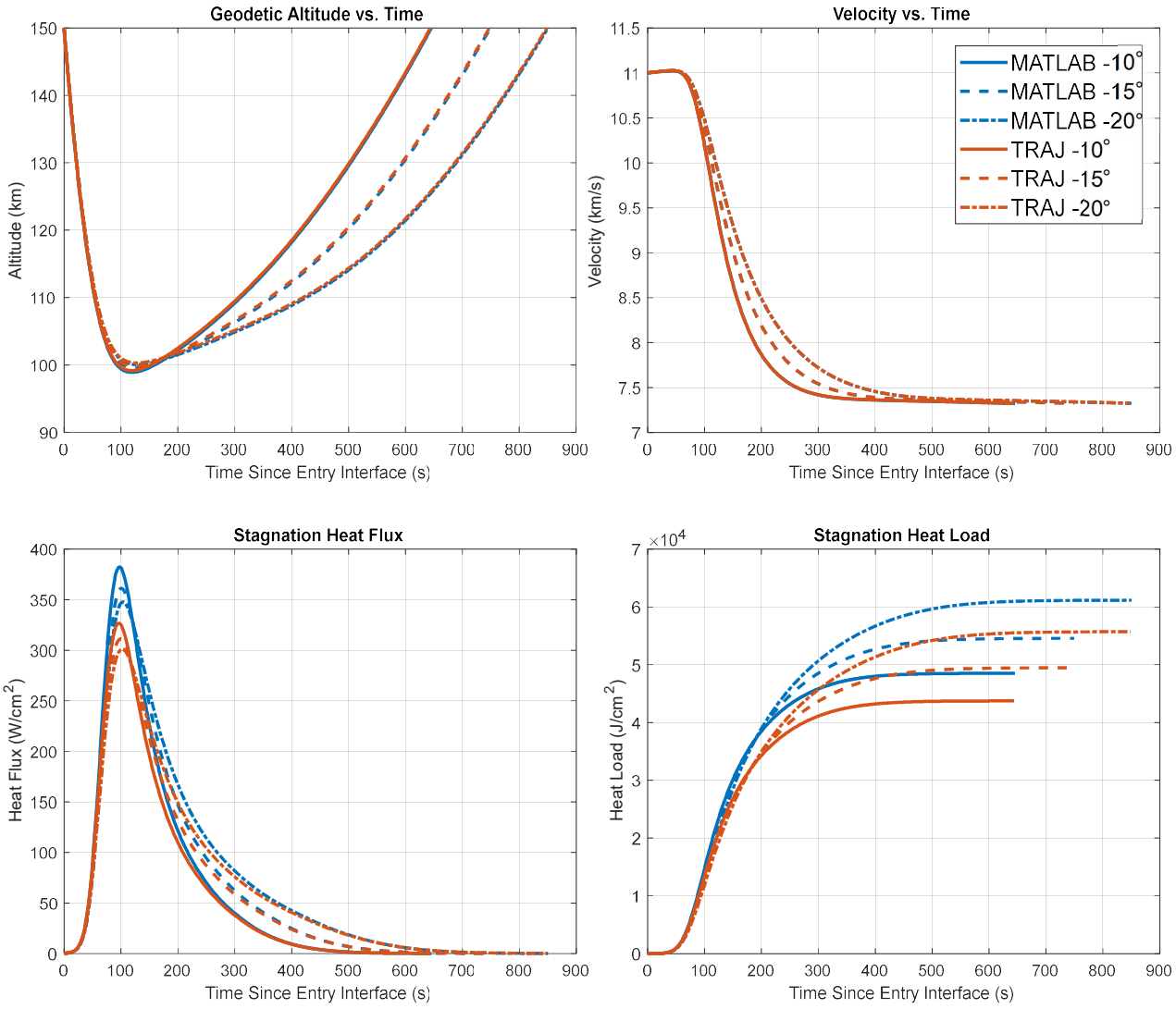


Figure 5.10 MATLAB-TRAJ Comparison Lift Down

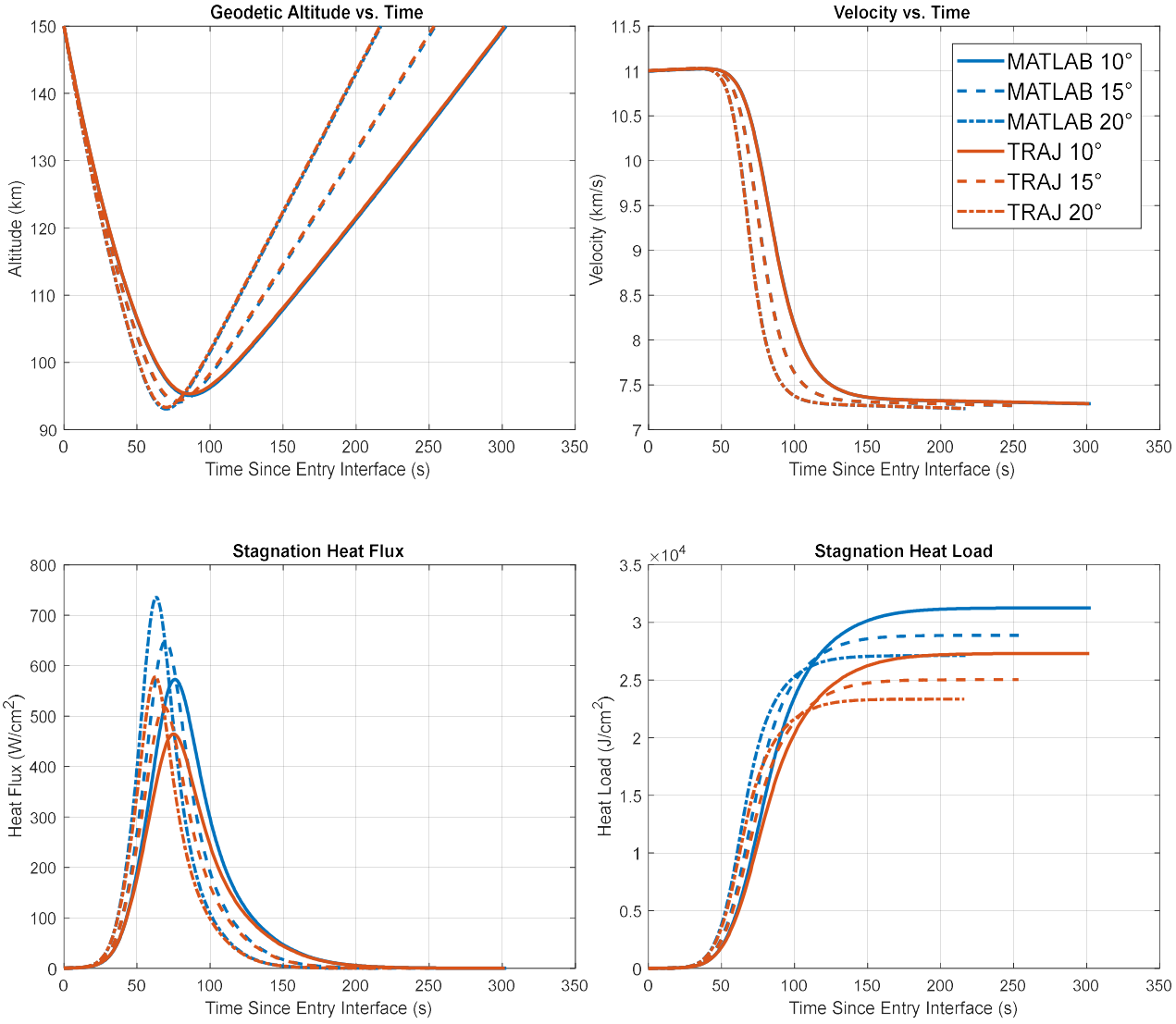


Figure 5.11 MATLAB-TRAJ Comparison Lift Up

Table 5.4 MATLAB-TRAJ Lift Up, Lift Down Comparison

Lift Down / Lift Up	Resulting EFPA (deg)	Peak Convective Heat Flux (W/cm^2)	Peak Radiative Heat Flux (W/cm^2)	Total Heat Load (Conv. J/cm^2)
MATLAB (-10°)	-5.424	382.25	-	48521.66
TRAJ (-10°)	-5.410	326.70	11.16	43108.42
MATLAB (-15°)	-5.368	361.27	-	54560.16
TRAJ (-15°)	-5.354	302.84	9.320	48836.38
MATLAB (-20°)	-5.329	347.80	-	61161.99
TRAJ (-20°)	-5.315	301.17	9.590	55053.46
MATLAB (10°)	-5.976	572.89	-	31264.45
TRAJ (10°)	-5.962	464.67	24.49	26578.55
MATLAB (15°)	-6.248	648.38	-	28883.92
TRAJ (15°)	-6.234	518.04	32.14	24242.16
MATLAB (20°)	-6.592	735.47	-	27123.57
TRAJ (20°)	-6.576	578.46	66.14	22462.20

6. Preliminary Results

6.1. Preliminary Multi-Pass Venus Aerocapture Trajectory

A test multi pass trajectory was implemented to achieve the same initial state as the validation run at atmospheric interface (Figure 5.1). The orbit was backed out to an altitude of ~300,000 km to allow for a small trajectory correction maneuver to adjust the perigee to target a 500,000 km apoapsis after the initial aero-pass. The spacecraft state class can readily convert between classical orbital elements, a position and velocity state vector, and the topocentric coordinates (Table 4.1). The mission plan object produced 4 intermediate braking orbits before the final 500 km science orbit was reached. These were automatically scaled to match the delta V lost on the first orbital insertion pass. The spacecraft was given the same initial mass and configuration as the validation run. Propulsion system parameters were estimated, the system needed a high thrust propulsion system for the perigee raise maneuver and low thrust maneuvering system for the high accuracy TCM's.

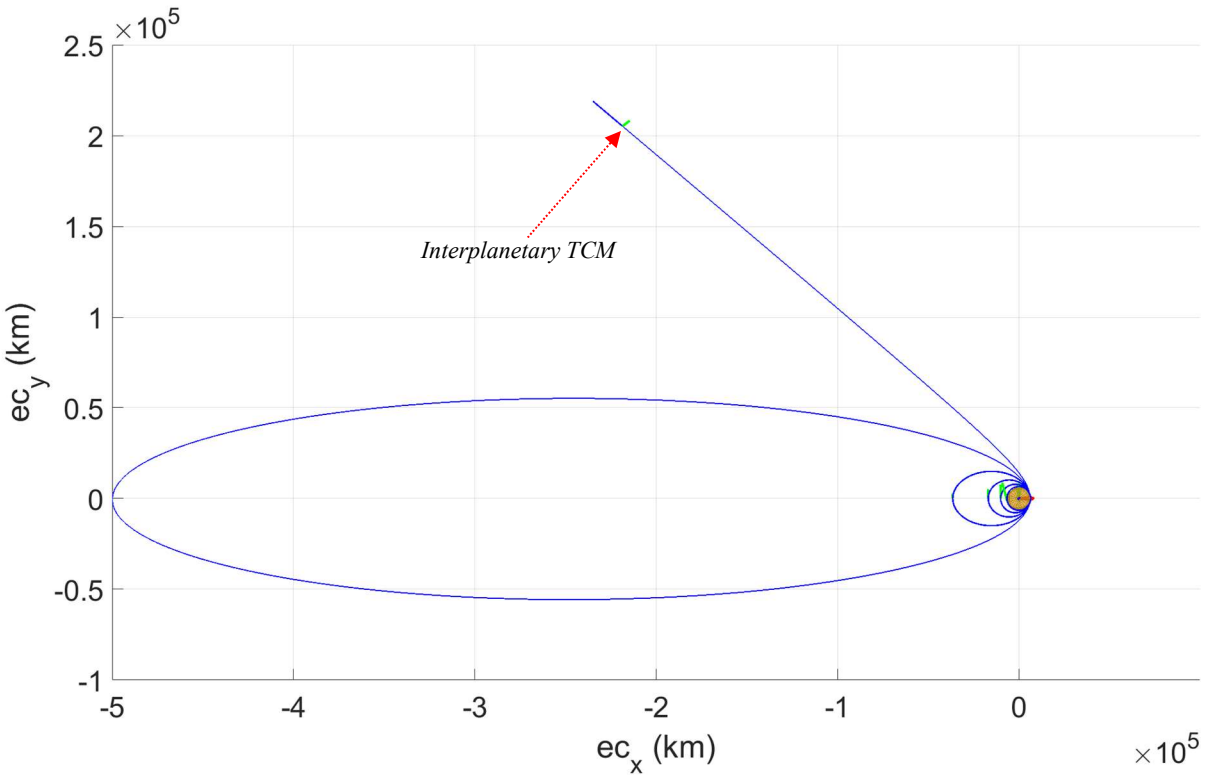


Figure 6.1 Full Venus Multi-Pass Trajectory

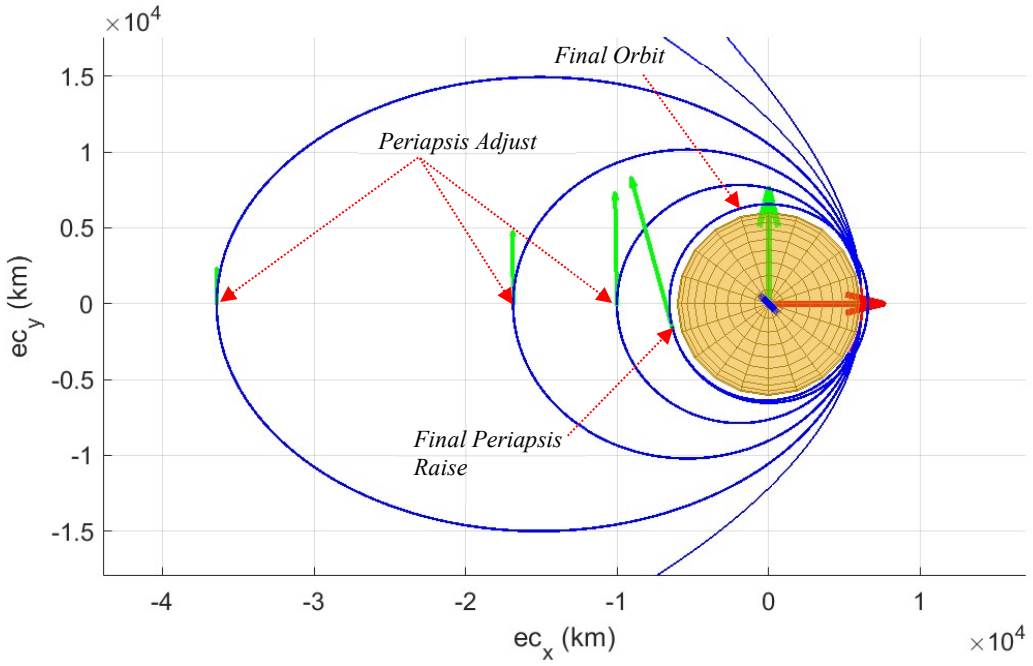


Figure 6.2 Venus Multi Pass Trajectory Planet Centered

Table 6.1 Spacecraft Inputs

Spacecraft Input Parameters	
Parameter	**Value**
Initial Mass	150 kg
High Thrust System	300 N
Low Thrust System	10 N
ISP (both systems)	300 s
Drag Coefficient (α=0°)	1.393
Diameter	1 m
Nose Radius	0.25 m
Sphere Cone Angle	60°

Table 6.2 Con-Ops Summary

Maneuvers Summary		
Maneuver	**Duration (s)**	**ΔV (m/s)**
Interplanetary TCM	2.27	0.15
1st Periapsis Adjust	0.14	0.01
2nd Periapsis Adjust	0.85	0.06
3rd Periapsis Adjust	2.85	0.19
4th Periapsis Adjust	9.11	0.61
Final Perigee Raise	55.21	112.20
Totals		
Mission Duration (days)		17.78
ΔV Expenditure (m/s)		113.20
Propellant Expenditure (kg)		5.67

Table 6.3 Initial State

Interplanetary Orbital Elements	
Parameter	**Value**
Eccentricity (e)	1.307
Semi Major Axis (a)	-2.00e4 km
Inclination (i)	0°
Argument of Periapsis (ω)	0°
Long. of Ascending Node (Ω)	0°
True Anomaly (Θ)	137°
Hyperbolic Excess Velocity	4.030 km/s
Julian Date	2455504

The final science orbit was slightly off from the target of 500 km at ~494 x 506 km, the implementation of the final trajectory raise burn could be improved. Overall the mission performance is satisfactory with only a small percentage of the initial spacecraft mass being expended chemical propellant. The mission duration is also manageable and is a small fraction of the interplanetary cruise phase. The duration figure is measured from the initial interplanetary state all the way to one completion of the final science orbit. Disadvantages to this architecture include the need for orbital maneuvers and navigation measurements while the spacecraft is still contained within the TPS aeroshell.

6.1.1. Aerothermal Results: Preliminary

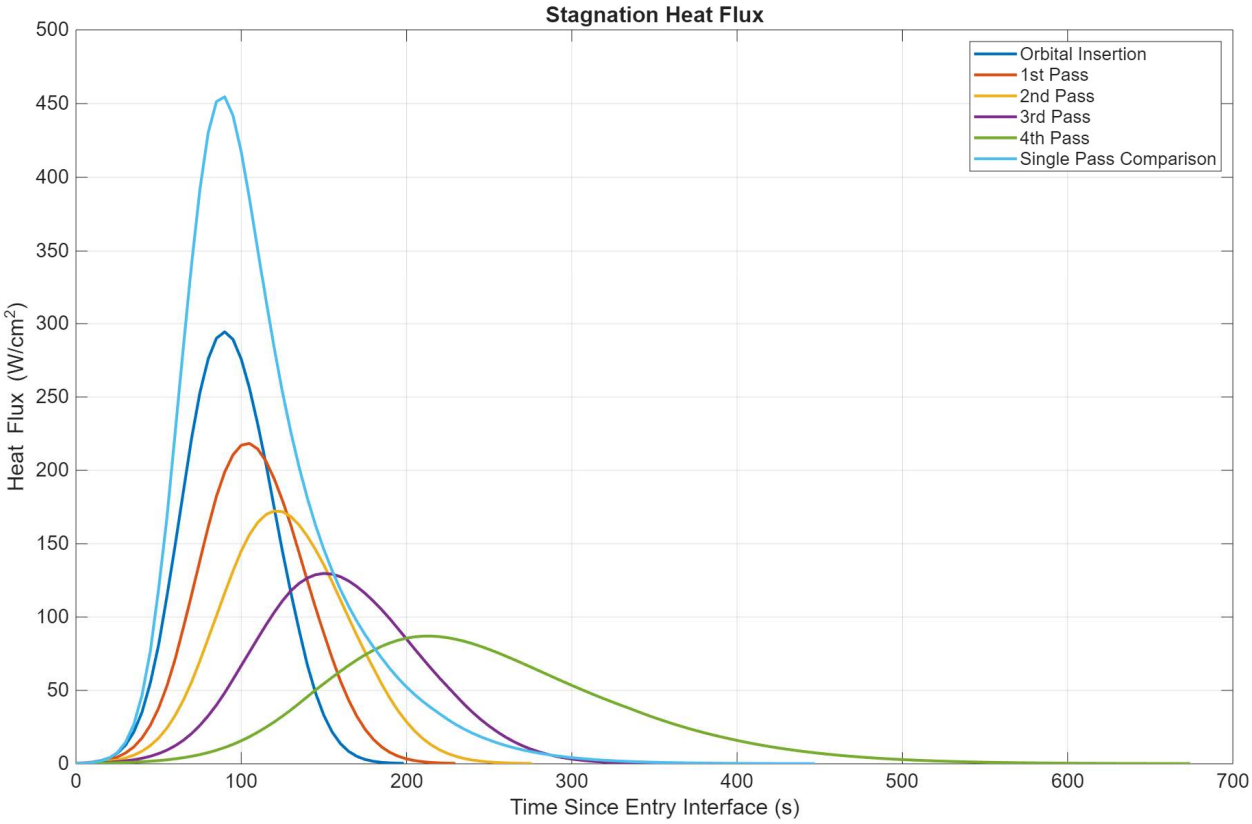


Figure 6.3 Multi Pass Heat Flux Results

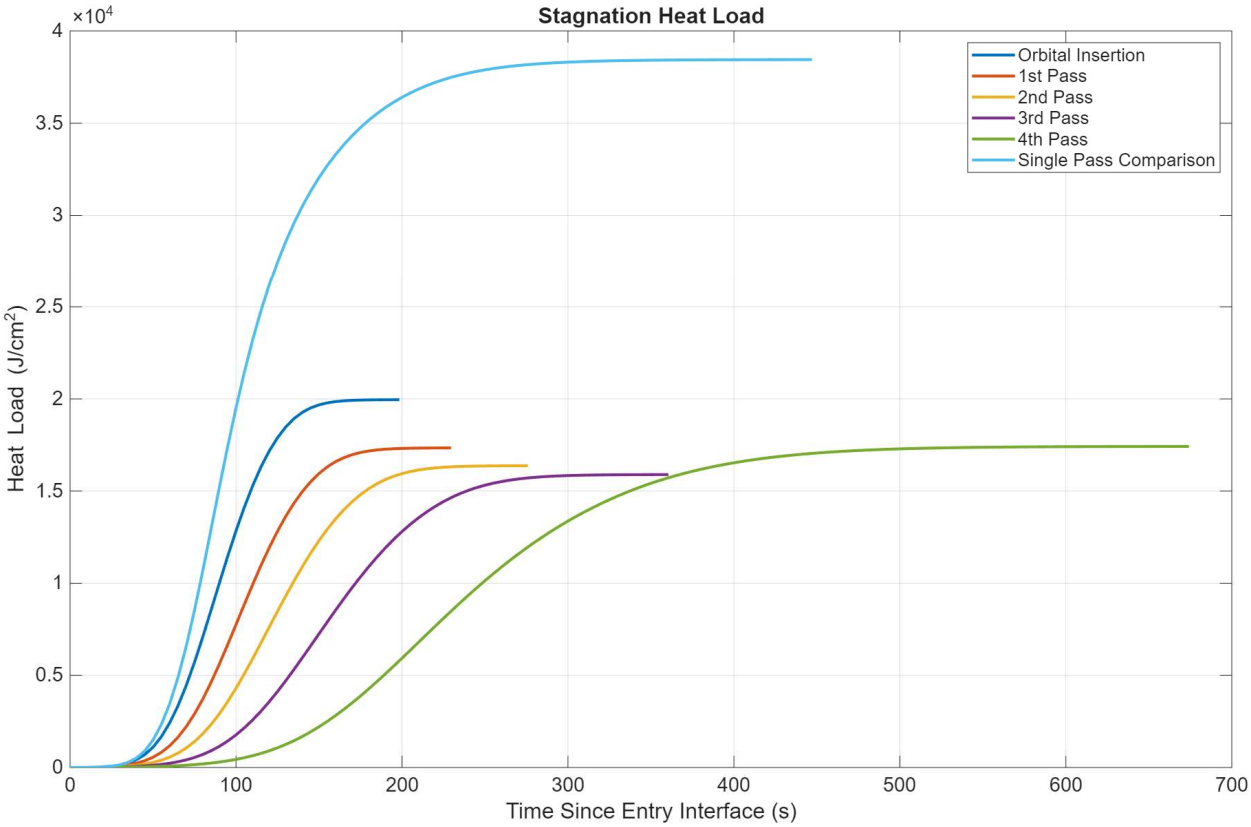


Figure 6.4 Multi Pass Total Heat Load Results

Preliminary results comparing the aerothermal environments of multi and single-pass aerocapture missions indicate a significant reduction in total heat load for each atmospheric entry. The target apogee of 500,000 km appeared to be a point of diminishing returns of reduced heating vs. mission duration, this value is further optimized in section 6.2. While the environments of each pass are more benign, the sum of the total energy absorbed by the TPS throughout the multiple passes is higher than the single pass. Given the orbital periods are on the order of several days, the heatshield would have sufficient time to cool down, though with an ablative TPS there would be a finite and compounding amount of material lost on each pass. This approach could be enabling for re-usable TPS such as flexible carbon weaves that have a lower maximum heat flux tolerance but can survive multiple insertions. A thorough TPS sizing effort is required to fully assess any mass savings (if any) with the multi-pass method.

6.2. Aerocapture Sensitivity Analysis

From previous results it is apparent that the first atmospheric entry from an interplanetary state is the driving case in terms of the heating environment and reduction in velocity. The spacecraft must become captured on this pass which puts a lower bound on the overall intensity of the heating environment. A study was conducted to examine the design and trajectory space of amulti-pass aerocapture by looking at the initial pass bounding case in terms of the target apoapsis, vehicle ballistic coefficient, and entry velocity. These are the primary driving factors for the maximum heat load which determines the type of TPS material required. The target apoapsis of 500,000 km used in previous cases was a qualitative estimate based on engineering judgement. An optimal value requires a balance between the orbital period of the post-exit orbit, which drives the mission duration and the peak heat flux which dictates the TPS material.

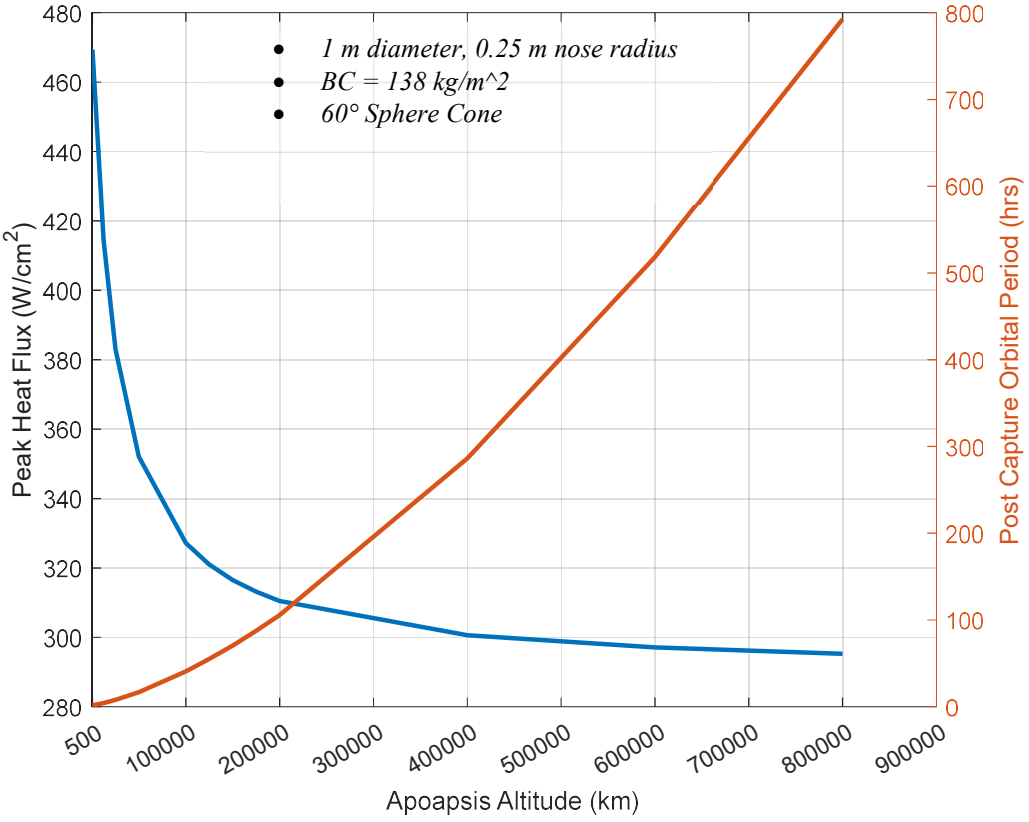


Figure 6.5 Peak Stag. Heat Flux to Orbital Period Tradeoff

Figure 6.5 shows a series of trajectories run for different post-exit apoapsis targets from an interplanetary trajectory, the vehicle configuration and initial state are the same as Table 6.1 and Table 6.3. Figure 6.5 shows significant diminishing returns in terms of heating reduction after around 200,000 km while the orbital period continues to rise nonlinearly. The first post capture orbit is also dominant in terms of the total multi-pass con-ops duration. An obvious visual choice for the ideal post-capture apoapsis would be the intersection of the two curves but a more quantitative approach would be to construct a weight function between normalized values of peak heat flux (q_N) and orbital period (τ_N).

$$q_N(r_a) = \frac{q(r_a) - q_{min}}{q_{max} - q_{min}} \tag{6.1}$$

$$\tau_N(r_a) = \frac{\tau(r_a) - \tau_{min}}{\tau_{max} - \tau_{min}} \tag{6.2}$$

$$W(r_a) = \sqrt{\tau_N^2 + q_N^2} \tag{6.3}$$

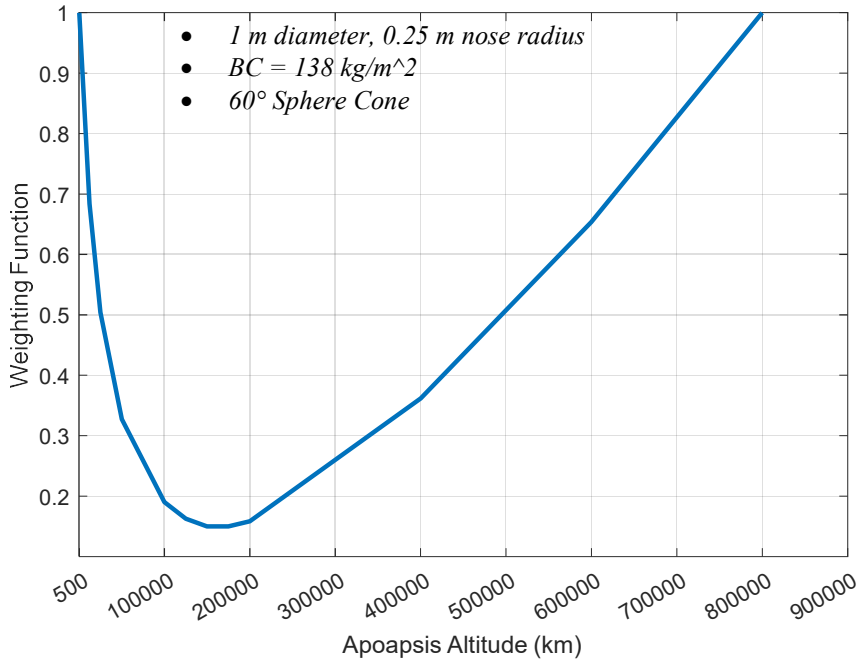


Figure 6.6 q_N-τ_N Weight Function

The weight function shown in eqn. 6.3 is based on the peak heat flux and orbital period at chosen apoapsis limits $\tau_{min,max}$ and $q_{min,max}$. The values are normalized over the entire range. The lower apoapsis limit was set to 500 km, as that is a standard low science orbit described in [23], the upper limit was set to 800000 km, which is near the limit of a bounded Venusian orbit. The minimum of this weighting function is the point where peak heat flux is minimized without the expense of a significant increase in mission duration. Figure 6.6 shows the weight function plotted with the same x axis values as Figure 6.5. The minimum occurs at ~175000 km with a peak stagnation heat flux of ~313 W/cm^2 and orbital period of ~88 hrs. The weight function limits can be adjusted depending on mission requirements.

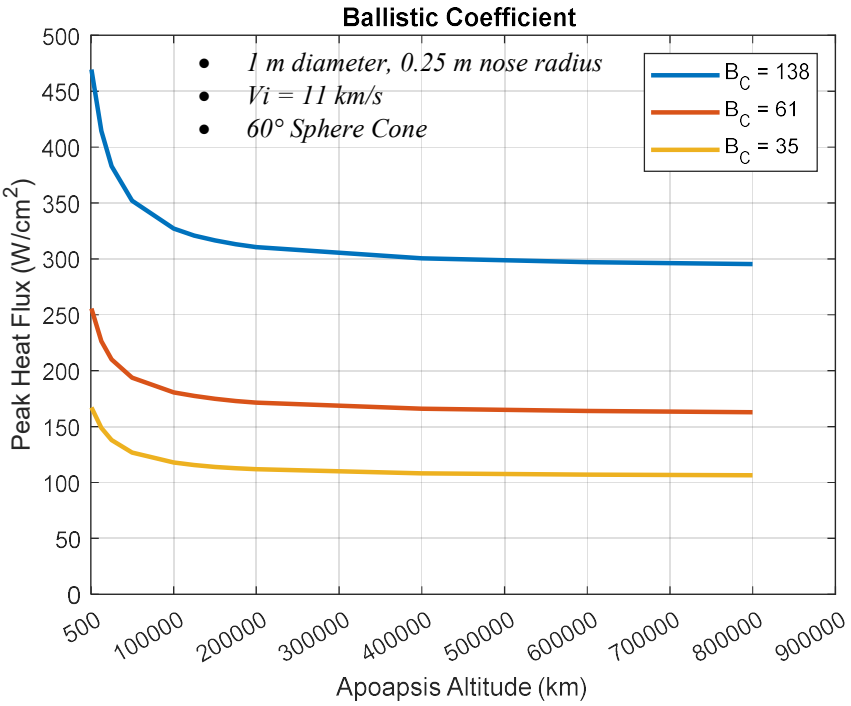


Figure 6.7 Ballistic Coefficient Effect on Peak Heating

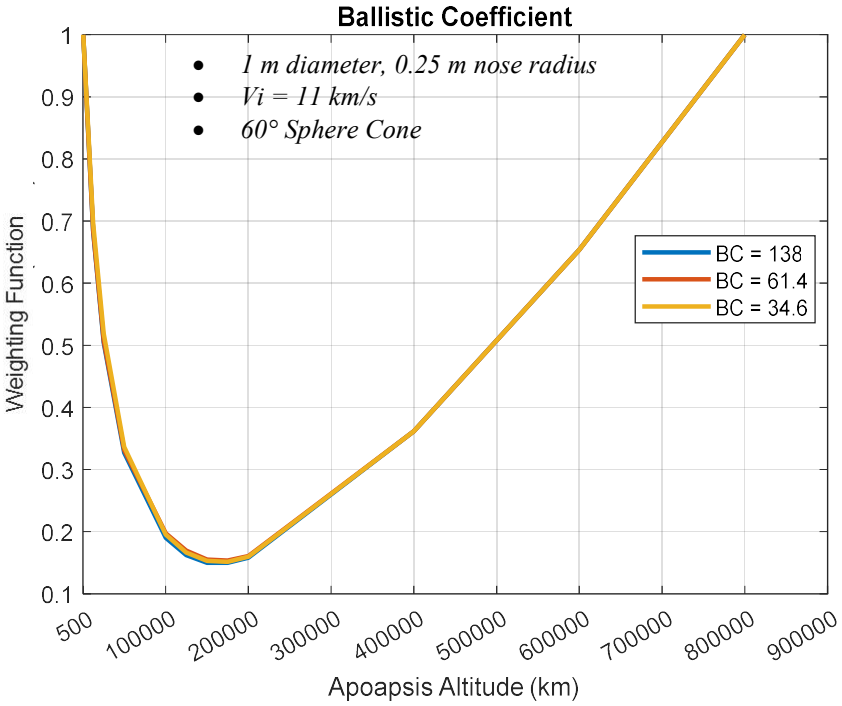


Figure 6.8 Ballistic Coefficient (kg/m^2) Effect on Weight Function

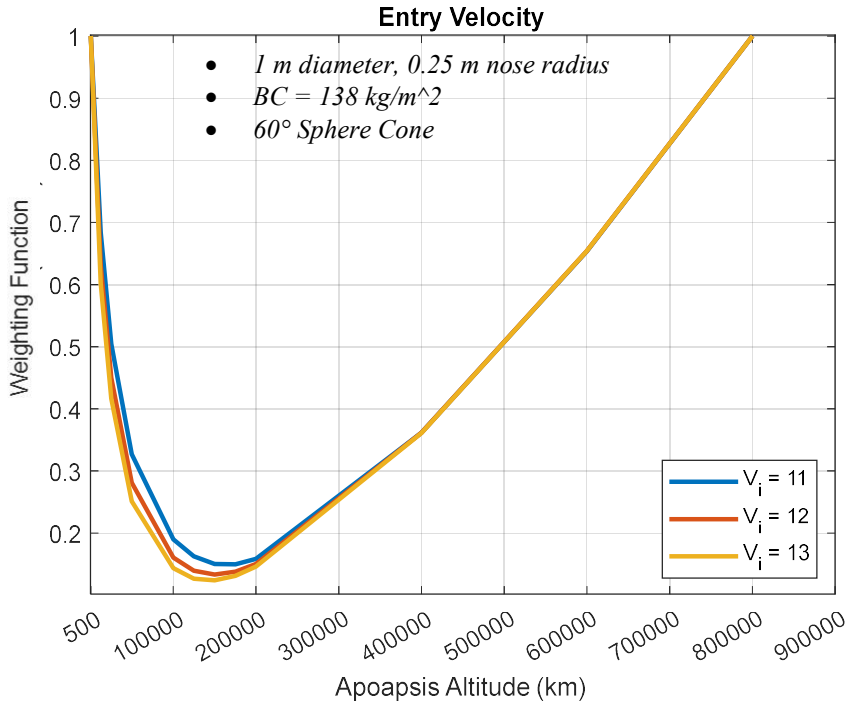


Figure 6.9 Entry Velocity (km/s) Effect on Weight Function

The ballistic coefficient, shown in eqn. 6.4 is a way to normalize several aerodynamic characteristics of the entry vehicle into one parameter. Figure 6.8 and Figure 6.9 illustrate low sensitivity of the weight function minimum to ballistic coefficient and entry velocity. This "sweet spot" of apoapsis altitudes is almost entirely dependent on the planetary destination, with the obvious condition that it is above the target science orbit for the mission. It should be noted that the sensitivity analysis trajectories run in this section were ballistic entries at $\alpha = \beta = 0$.

$$B_C = \frac{m}{C_D A} \tag{6.4}$$

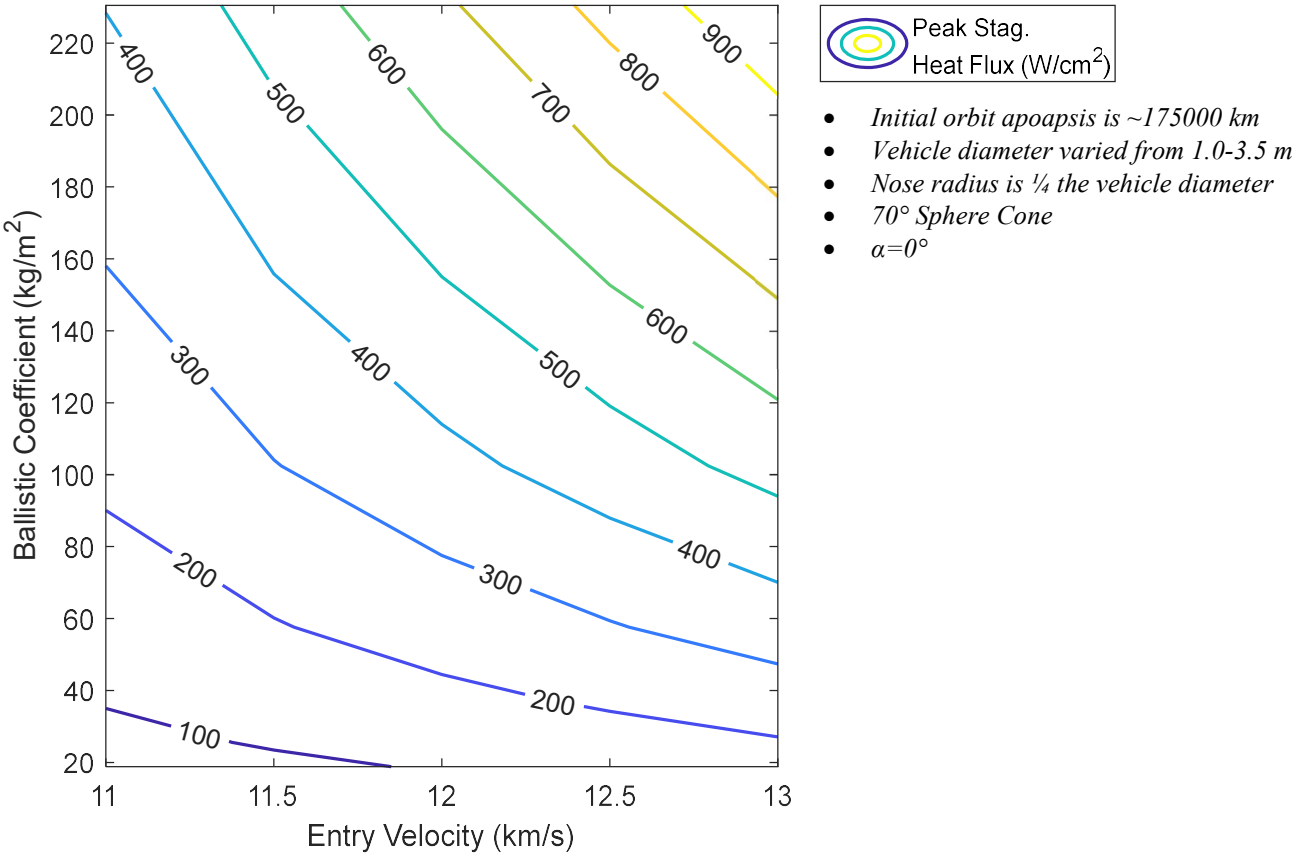


Figure 6.10 Venus Aerocapture Design Space

An optimal target apoapsis altitude is tied to a particular trajectory and resulting heating environment. This allows for an aerocapture mission design space to be calculated where the peak heat flux can be visualized as a function of ballistic coefficient and entry velocity. The weight function analysis shows that additional constraints are not necessary as the apoapsis range is only strongly dependent on the destination. One exception is the entry vehicle effective nose radius which factors into the Sutton Graves heating correlation but does not have a significant impact on the vehicle ballistic coefficient. A vehicle with a smaller nose radius will produce higher stagnation heat fluxes for the same ballistic coefficient and all else being equal. With this

in mind new design spaces should be generated for different classes of entry vehicles or new planetary destinations.

The design space in Figure 6.10 indicates vehicle configuration (B_C) and arrival velocity can have a significant influence on the heating environment. The values are valid for any target apoapsis above the weight function predictions, Entry vehicles like MSL and Mars 2020 have ballistic coefficients ~150 kg/m^2, if such a vehicle was used for a Venus aerocapture, ablative TPS would be necessary for peak heating rates >300 W/cm^2.

6.2.1. Uncertainty Modeling

So far only nominal values have been used in the analysis, though uncertainty propagation is necessary to gauge the various sensitivities of the design space. GRAM supports monte carlo runs and perturbations on all atmospheric properties. Assessing how the aerodynamic coefficients respond to a normal distribution of angles of attack, and free stream conditions could allow a 3σ envelope of trajectories to be plotted. One of the potential advantages to multi-pass aerocapture is the higher initial target apoapsis leaves more room for trajectory or guidance dispersions. Directly targeting a low 500 km orbit after an aerocapture requires a much tighter flight corridor than a 200,000 km orbit. The flexible object oriented nature of the simulation and perturbation ready parameters of GRAM would make the addition of uncertainties straight forward. However, quantifying the uncertainties and utilizing them for mission design is more involved and is out of the scope of this current project.

6.3. Mission Concept: SmallSat Venus Aerocapture with Deployable TPS

NASA's Adaptable, Deployable Entry Placement Technology (ADEPT) [25] is an attractive candidate for a low ballistic coefficient entry vehicle that can withstand multiple atmospheric entries. The technology utilizes a flexible carbon weave as a TPS that can be folded to allow for efficient packing in launch vehicle fairings. This TPS was tested in the NASA ARC Arc Jet facility at conditions of over 150 W/cm^2 of stagnation point convective heating though 50 W/cm^2 is a more realistic re-usable limit [25].

To home in on a basic entry vehicle configuration, a second batch run was conducted to narrow down the design window to the lower left corner of Figure 6.10. This focused design space is shown in Figure 6.11. In the spirit of choosing round numbers, a 3 m diameter aeroshell with a mass of 200 kg would have a ballistic coefficient of 17.55 kg/m^2. This is almost 8 times lower than the ballistic coefficient of the MSL entry vehicle and is likely close to the limit of materials and mass constraints for a flexible, deployable entry system. Additional system design work would be necessary to determine the feasibility of vehicles in this range. Assuming a TPS mass fraction of 30-40% would allow for 120-140 kg science payload orbiter. A trim angle of attack of 10° was chosen arbitrarily as optimizing for angle of attack requires stability, mass properties, and other vehicle characteristics that aren't modeled by the simulation. If this vehicle targeted an optimized post capture apoapsis of 200,000 km it would encounter a peak stagnation heat flux of ~59.5 W/cm^2 for the lift down case and ~70 W/cm^2 for the lift up case which is in the ballpark for a re-usable, deployable entry system.

A multi-pass architecture analyzing this configuration was setup with values shown in Table 6.4 through 6.7. The entry state (Table 6.6) is identical to the mission studied in section 6.3. The mission planner solution with a 200,000 km initial apoapsis yields only 3 passes instead of 4, bringing the mission duration down to ~5.5 days. Note that the drag and lift coefficients shown in Table 6.4 are nominal values and can vary slightly due to the effects discussed in section 4.2.4. A small 100 kg satellite would likely consist of a pressure fed monopropellant system with space for only a few kgs of propellant. A large cruise stage would be necessary for a purely propulsive orbital insertion. The advantage of a deployable system is the back shell can be opened to the space environment to allow for easier communication and maneuvers. The aerothermal plots shown in Figure 6.16 illustrate the flight corridor between the extreme lift up and lift down cases. The aerothermal heating metrics trend as expected, with heat flux dominating the lift-up trajectory and total heat load the lift-down trajectory. The delta between the lift-up and lift down stagnation heat flux and total heat load is significantly higher for the single pass trajectory. The multi-pass configuration offers a smaller range of heating environments that the TPS needs to be sized for, which can be attractive to mission designers. The nominal entry trajectory will fall somewhere between the two cases depending on the bank angle profile. Provided that the carbon weave TPS can cool down between passes this may be enabling for such a system that only comprises of a few layers of fabric to maintain flexibility and cannot absorb excessive amounts of energy. An additional multi pass trajectory with an even lower ballistic coefficient vehicle is presented in Appendix 9.3.

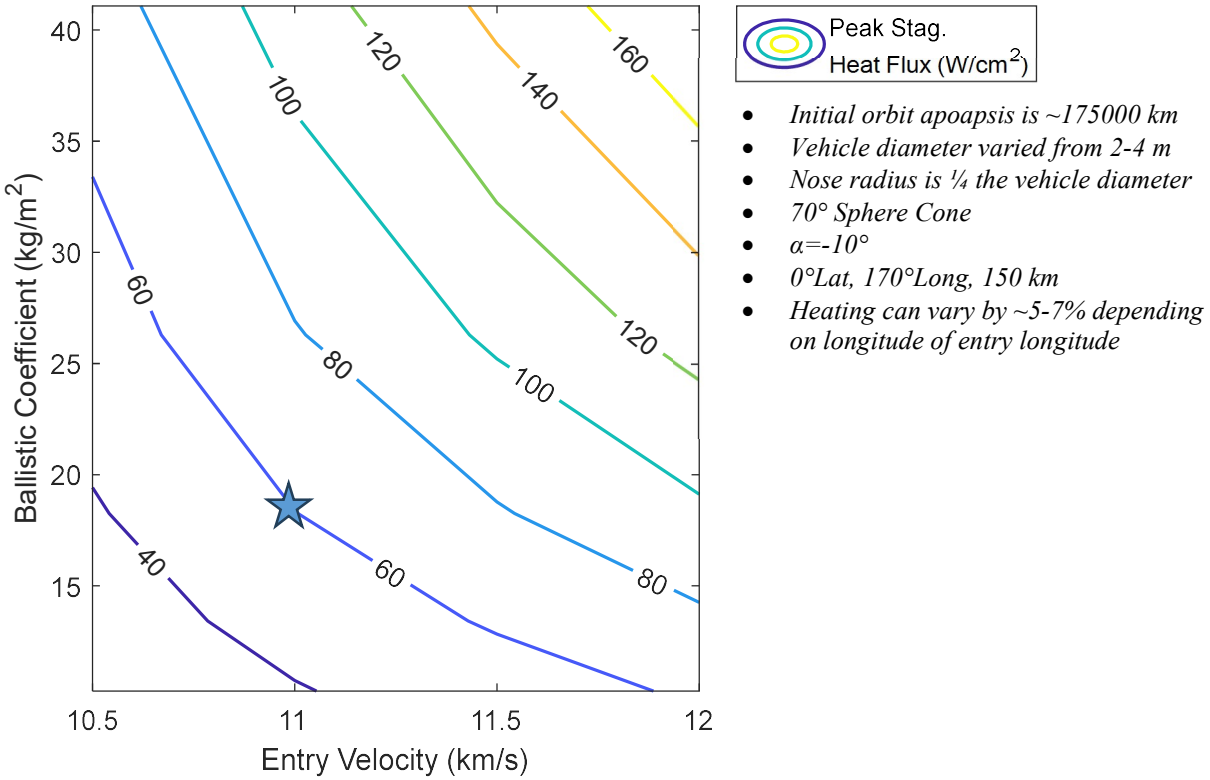


Figure 6.11 Venus Aerocapture Design Space for Low B_C Vehicles (Lift Down)

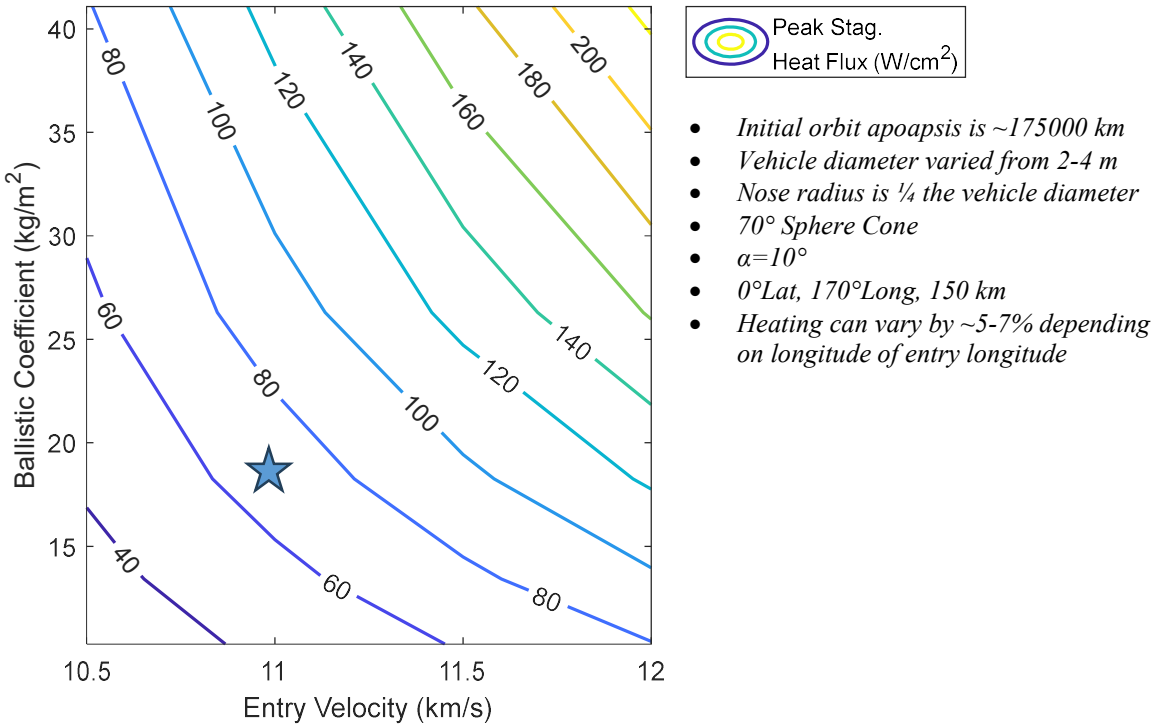


Figure 6.12 Venus Aerocapture Design Space for Low B_C Vehicles (Lift Up)

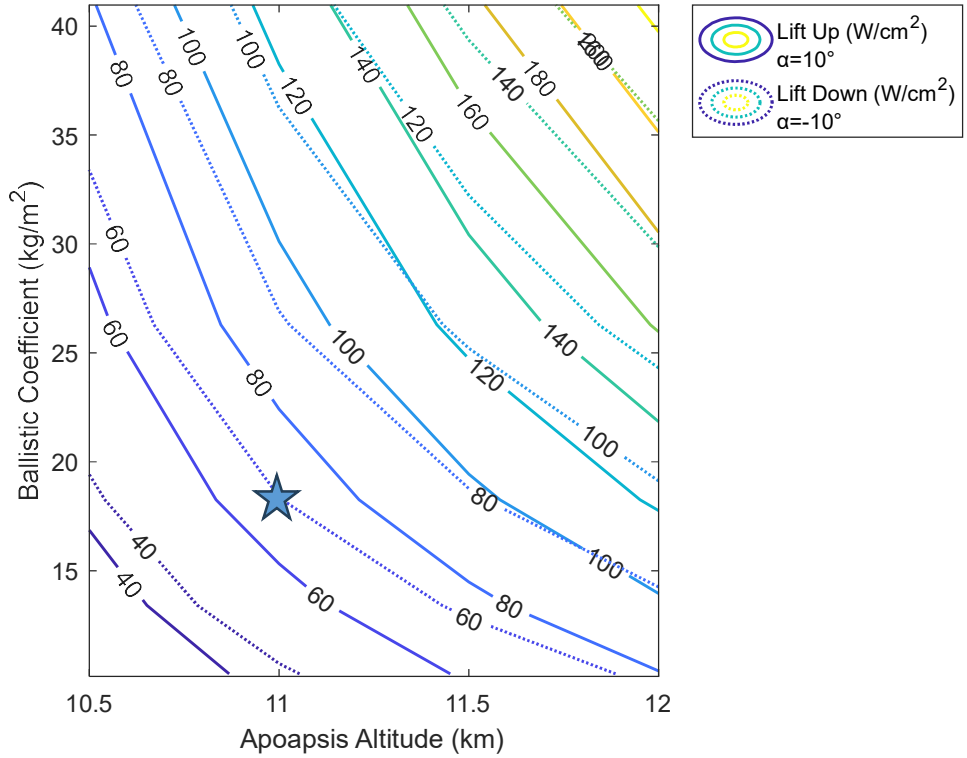


Figure 6.13 Venus Aerocapture Design Space for Low B_C Vehicles (Overlay)

Table 6.4 Spacecraft Inputs

Spacecraft Input Parameters	
Parameter	**Value**
Initial Mass	200 kg
High Thrust System	120 N
Low Thrust System	10 N
ISP (both systems)	300 s
Trim Angle of Attack (α)	10°/-10°
C_D at α=10°/-10°	1.550
C_L at α=10°/-10°	$\pm$0.237
Diameter	3 m
Nose Radius	0.75 m
Sphere Cone Angle	70°
B_C at α=0°	17.55 kg/m^2
B_C at α=10°/-10°	18.26 kg/m^2

Table 6.5 Con-Ops Summary

Maneuvers Summary		
Maneuver	**Duration (s)**	**ΔV (m/s)**
Lift Up		
Interplanetary TCM	10.49	0.525
1st Periapsis Adjust	0.128	0.006
2nd Periapsis Adjust	2.000	0.100
3rd Periapsis Adjust	10.15	0.507
Final Perigee Raise	190.0	116.3
Lift Down		
Interplanetary TCM	12.17	0.610
1st Periapsis Adjust	0.194	0.010
2nd Periapsis Adjust	2.141	0.110
3rd Periapsis Adjust	8.137	0.407
Final Perigee Raise	177.1	108.2
Totals	**Lift Up**	**Lift Down**
Mission Duration (days)	5.461	5.468
Maneuvers ΔV (m/s)	117.4	109.4
Propellant Usage (kg)	7.832	7.304

Table 6.6 Initial State

Interplanetary Orbital Elements	
Parameter	**Value**
Eccentricity (e)	1.3074
Semi Major Axis (a)	-2.0005e4 km
Inclination (i)	0°
Argument of Periapsis (ω)	0°
Long. of Ascending Node (Ω)	0°
True Anomaly (Θ)	137°
Hyperbolic Excess Velocity	4.0298 km/s
Julian Date	2455504
Initial Target Apoapsis (km)	200000

Table 6.7 Atmospheric Entries

Aero-Pass Summary			
Pass	q_{max} **(W/cm^2)**	J_s **(J/cm^2)**	**ΔV (km/s)**
Lift Up			
Insertion	70.05	4022	0.916
1st Pass	53.74	3645	0.914
2nd Pass	39.55	3350	0.912
3rd Pass	26.58	3340	0.892
Lift Down			
Insertion	61.66	4416	0.916
1st Pass	46.05	4085	0.915
2nd Pass	32.30	3903	0.914
3rd Pass	19.22	4445	0.912

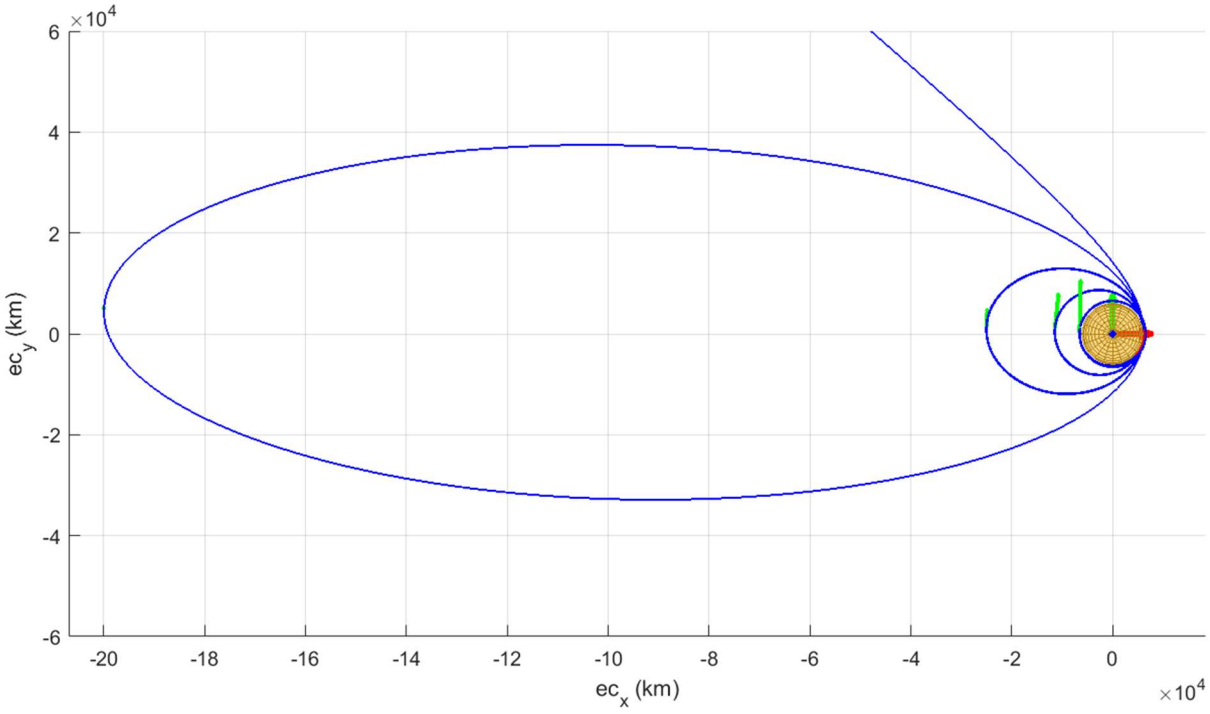


Figure 6.14 Venus Deployable TPS Multi-Pass Trajectory

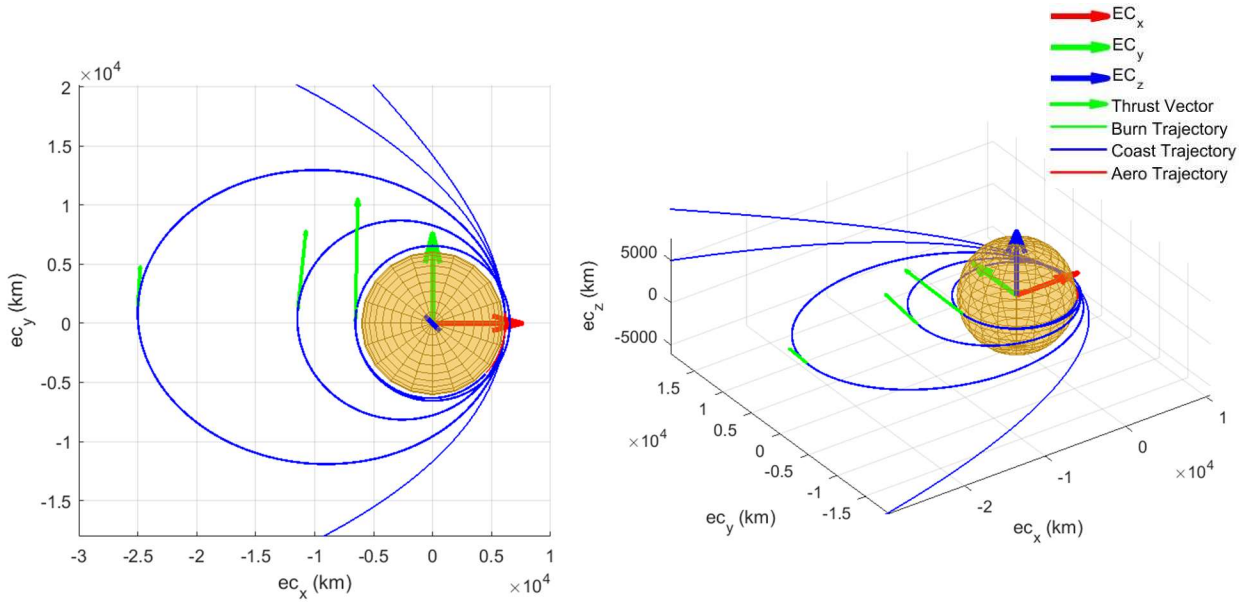


Figure 6.15 Venus Deployable TPS Additional Trajectory Views

6.3.1. Aerothermal Results: Deployable TPS

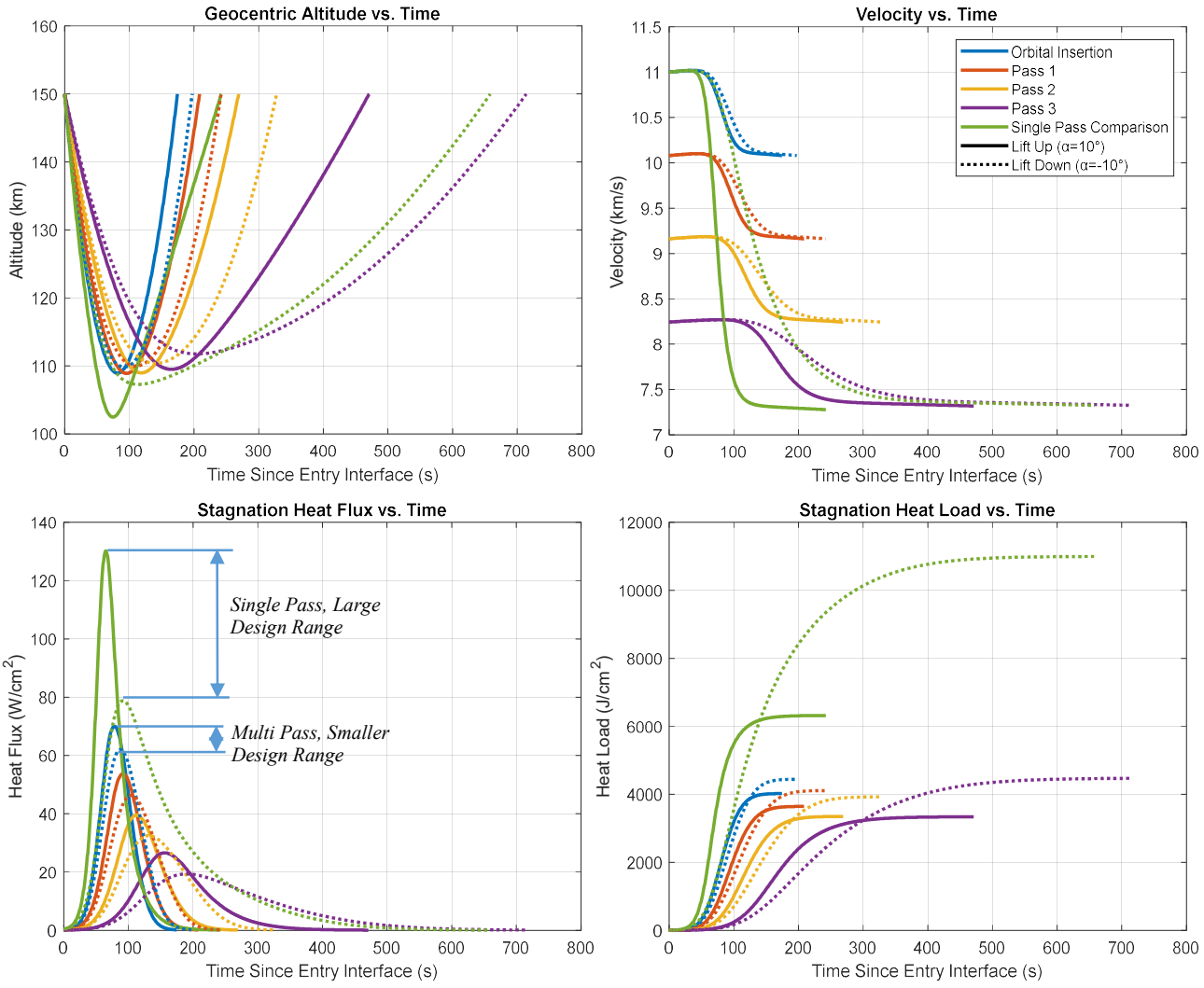


Figure 6.16 Venus Multi-Pass Deployable TPS Aerothermal Results

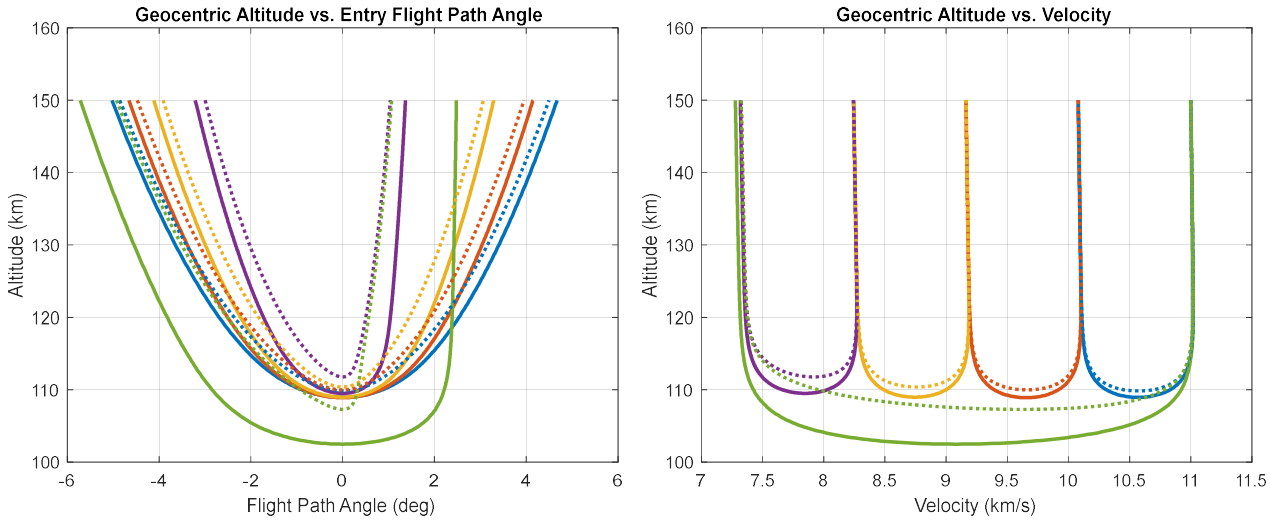


Figure 6.17 Venus Multi-Pass Deployable TPS Additional Results

7. Next Steps

In the first phase of this project an extensive MATLAB object oriented simulation framework was developed to study multiple atmospheric insertions to achieve a target orbit. The simulation utilizes the NASA Global Atmospheric Reference Model and is based on validated physics and prediction methods. In the second half of the project some improvements were made to the overall architecture and robustness to different inputs. The code was compared to the NASA TRAJ flight validated trajectory program and other results from literature with results that are in-family.

The aerodynamic database was significantly expanded to model aerodynamic forces in 3 dimensions and at angles of attack and sideslip. Additional validations on the modified Newtonian model have been performed and lift and drag cases have been compared with TRAJ and indicate strong agreement. The rarefied flow modeling has been implemented and tested. GRAM has been tested for all 7 atmospheric destinations. Uncertainty modeling is a major next step that would need to be addressed for this modeling tool be used in real conceptual design.

On the Aerothermal side the chemistry model methodology described in 4.3 needs to be expanded to include all constituents from each planetary atmosphere while utilizing the thermodynamic data from [3]. This would add additional properties to compare like temperature and pressure behind the shock at the stagnation point and flank. The ultimate longer term goal is to size TPS for a conceptual mission based on the predicted aerothermal environments. The NASA developed tool FIAT is an industry standard for TPS sizing. Additional longer term goals involve expanding the simulation reference frames to include SPICE kernals for modeling time dependent planetary effects such nutation.

A simulation architecture utilizing multi-body mechanics of the entire solar system would allow for a macro level of mission planning and generation of initial states for atmospheric entries. This would allow direct coupling between elements such as launch, transfer windows, and gravity assists to aerothermal environments to assess the feasibility of various orbital insertion methods. While numerous mission planning software tools exist, EDL focused tools with an emphasis on preliminary design and optimization are less common. So far, the modeling and simulation skills learned during this project have been invaluable.

8. References

[1] Wright, M., and Dec, J., "Thermal and Fluids Analysis Workshop, 2012," Aerothermodynamic and Thermal Protection System Aspects of Entry System Design Course, Aug. 13-17, 2012, Pasadena, CA:.

[2] Origins, Worlds, and Life: A Decadal Strategy for Planetary Science and Astrobiology 2023-2032, National Academies Press, 500 5th St. NW, Washington, DC 20418, 2023.

[3] McBride, B. J., Zehe, M. J., and Gordon, S., "NASA Glenn coefficients for calculating thermodynamic properties of individual species," NASA/TP—2002-211556, John H. Glenn Research Center at Lewis Field, 2002.

[4] McBride, B. J., "Thermodynamic properties to 6000° K for 210 substances involving the first 18 elements, by Bonnie J. McBride and others," NASA SP-3001, 1963.

[5] Gordon, S., and McBride, B. J., "Computer program for calculation of complex chemical equilibrium compositions and applications," NASA Reference Publication 1311, 1994.

[6] Justh, H., Cianciolo, A. D., Hoffman, J., and Allen Jr, G., "Uranus Global Reference Atmospheric Model (Uranus-GRAM): User Guide," NASA/TM - 20240011228, Marshall Space Flight Center Huntsville, AL. 35812, 2021.

[7] Justh, H., Cianciolo, A. D., Hoffman, J., and Allen Jr, G., "Venus Global Reference Atmospheric Model (Venus-GRAM): User Guide," NASA/TM-20210022168, Marshall Space Flight Center Huntsville, AL. 35812, 2024.

[8] Moses J. I., Cavalié T., Fletcher L. N. and Roman M. T., 2020 Atmospheric chemistry on Uranus and Neptune. Phil. Trans. R. Soc. A. 378: 20190477. http://doi.org/10.1098/rsta.2019.0477

[9] Dutta, S., Shellabarger, E., Scoggins, J. B., Gomez-Delrio, A., Lugo, R., Deshmukh, R., Tackett, B., Williams, J., Johnson, B., Matz, D., Geiser, J., Morgan, J., Restrepo, R., and Mages, D., "Uranus Flagship-Class Orbiter and Probe Using Aerocapture," AIAA SCITECH 2024 Forum, Jan. 8-12, 2024, Orlando, FL. https://doi.org/10.2514/6.2024-0714

[10] Deshmukh, R., Dutta, S., Lugo, R., Restrepo, R., Mages, D., Johnson, B., Matz, D., Geiser, J., Scoggins, J. B., Shellabarger, E., Gomez-Delrio, A., and Williams, J., "Performance Analysis of Aerocapture Systems for Uranus Orbiter," AIAA SCITECH 2024 Forum, Jan. 8-12, 2024, Orlando, FL. https://doi.org/10.2514/6.2024-0716

[11] Restrepo, R., Mages, D., Smith, M., Deshmukh, R., Dutta, S., and Benhacine, L., "Mission Design and Navigation Solutions for Uranus Aerocapture," AIAA SCITECH 2024 Forum, Jan. 8-12, 2024, Orlando, FL. https://doi.org/10.2514/6.2024-0715

[12] Morgan, J., Williams, J.D., Venkatapathy, E., Gasch, M., Deshmukh, R.G., Shellabarger, E., Scoggins, J.B., Gomez-Delrio, A., Tackett, B., Dutta, S., "Thermal Protection System Design of Aerocapture Systems for Uranus Orbiters," AIAA SCITECH 2024 Forum, Jan. 8-12, 2024, Orlando, FL. https://doi.org/10.2514/6.2024-0952

[13] Shellabarger, E., Scoggins, J. B., Hinkle, A. D., Dutta, S., Deshmukh, R., Patel, M., and Agam, S., "Aerodynamic Implications of Aerocapture Systems for Uranus Orbiters," AIAA SCITECH 2024 Forum, Jan. 8-12, 2024, Orlando, FL. https://doi.org/10.2514/6.2024-0950

[14] Scoggins, J. B., Hinkle, A. D., and Shellabarger, E., "Aeroheating Environment of Aerocapture Systems for Uranus Orbiters," AIAA SCITECH 2024 Forum, Jan. 8-12, 2024, Orlando, FL. https://doi.org/10.2514/6.2024-0951

[15] Gomez-Delrio, A. J., and Dutta, S., "Design Implications for Aerocapture Systems Placing Flagship-Class Uranus Orbiters," AIAA SCITECH 2024 Forum, Jan. 8-12, 2024, Orlando, FL. https://doi.org/10.2514/6.2024-0953

[16] L. L. Perini, "Compilation And Correlation Of Experimental, Hypersonic, Stagnation Point Convective Heating Rates," ANSP-M-4, United States. Dept. of Energy. Technical Information Center, Oak Ridge, Tenn. 1972.

[17] Anderson, John D. Hypersonic and High Temperature Gas Dynamics. 2nd ed., American Institute of Aeronautics and Astronautics, 1801 Alexander Bell Drive, Reston, VA. 2006.

[18] Murri, D. et al., "Improvements to the Flight Analysis and Simulation Tool (FAST) and Initial Development of the Genesis Flight Mechanics Simulation for Ascent, Aerocapture, Entry, Descent, and Landing (A2EDL) Trajectory Design," NASA TM-20210014622, 2021.

[19] Noca, M., and Bailey, R., "Mission trades for aerocapture at Neptune," 40th AIAA/ASME/SAE/ASEE Joint Propulsion Conference and Exhibit, Jul. 11-14, 2004, Fort Lauderdale, FL. https://doi.org/10.2514/6.2004-3843

[20] Rymer, A., Clyde B., Runyon, K. "Neptune Odyssey: Neptune Odyssey: A Flagship Concept for the Exploration of the Neptune–Triton System, The Planetary Science Journal, 2:184 (15pp), 2021 October. Published online by the American Astronomical Society. https://doi.org/10.3847/PSJ/abf654

[21] Simon, A., Nimmo, F., Anderson, C. R. Journey to an Ice Giant System, Uranus Orbiter & Probe, PLANETARY MISSION CONCEPT STUDY FOR THE 2023–2032 DECADAL SURVEY, Johns Hopkins Applied Physics Lab, 11100 Johns Hopkins Rd., Laurel, MD. 20723, 7 June 2021. https://smd-cms.nasa.gov/wp-content/uploads/2023/10/uranus-orbiter-and-probe.pdf

[22] Curtis, H. D., Orbital Mechanics for Engineering Students, 3rd ed., Butterworth-Heinemann, An imprint of Elsevier, The Boulevard, Langford Lane, Kidlington, Oxford, OX5 1GB, UK, 2014.

[23] Lugo, R. A., Dutta, S., Matz, D., Johnson, B. J., Pensado, A. R., Roelke, E., Aguirre, J. T., and Powell, R., "Performance Analysis of SMALLSAT Aerocapture at Venus," AIAA SCITECH 2023 Forum, Jan. 23–27, 2023, National Harbor, MD. https://doi.org/10.2514/6.2023-0878

[24] Kumar, M., and Tewari, A., "Trajectory and attitude simulation for aerocapture and aerobraking," Journal of Spacecraft and Rockets, vol. 42, Jul. 2005, pp. 684–693. https://doi.org/10.2514/1.7117

[25] Morgan, J., Gokcen, T., and Wercinski, P., "Arc-jet testing of continuously woven aeroshells –Spiderweave– for adaptable deployable entry placement technology," AIAA AVIATION 2022 Forum, Jun. 27- Jul. 1, 2022, Chicago, IL. https://doi.org/10.2514/6.2022-3503

[26] Matz, D. A., and Cerimele, C. J., "Development of a Numeric Predictor-Corrector Aerocapture Guidance for Direct Force Control," AIAA Scitech 2020 Forum, Jan. 6-10, 2020, Orlando, FL. https://doi.org/10.2514/6.2020-0847

[27] Lu, P., Cerimele, C. J., Tigges, M. A., and Matz, D. A., "Optimal Aerocapture Guidance," Journal of Guidance, Control, and Dynamics, Vol. 38, No. 4, 2015, pp 553–565. http://dx.doi.org/10.2514/1.g000713.

[28] McBrayer, K. T., and Edwards, S. J., "Keplerian analysis for versatile evaluation of arbitrary trajectories," AIAA 2020-4031. *ASCEND 2020*. Nov. 16-18, Virtual. https://doi.org/10.2514/6.2020-4031

[29] Grant, M., and Braun, R., "Analytic hypersonic aerodynamics for conceptual design of entry vehicles," 48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition, Jan. 4-7, 2010, Orlando, FL. https://doi.org/10.2514/6.2010-1212

[30] Archinal, B.A., Acton, C.H., A'Hearn, M.F. *et al.* Report of the IAU Working Group on Cartographic Coordinates and Rotational Elements: 2015. *Celest Mech Dyn Astr* **130**, 22 (2018). https://doi.org/10.1007/s10569-017-9805-5

[31] Kenneth A. Hart, Soumyo Dutta, Kyle Simonis, Bradley A. Steinfeldt and Robert D. Braun. "Analytically-derived Aerodynamic Force and Moment Coefficients of Resident Space Objects in Free-Molecular Flow," AIAA 2014-0728. AIAA Atmospheric Flight Mechanics Conference, January 13-17, National Harbor, Maryland, USA 2014. https://doi.org/10.2514/6.2014-0728

[32] Kenneth A. Hart, Kyle R. Simonis, Bradley A. Steinfeldt, and Robert D. Braun. "Analytic Free-Molecular Aerodynamics for Rapid Propagation of Resident Space Objects," Journal of Spacecraft and Rockets 2018 55:1, 27-36 https://doi.org/10.2514/1.A33606

[33] Storch, Joel. (2002). Aerodynamic Disturbances on Spacecraft in Free-Molecular Flow. 70. 10.1061/40722(153)60.

[34] Jannuel V. Cabrera, Rohan Deshmukh and David Spencer. "Aerodynamic Database Development using a Bridging Function for a Conceptual Morphable Entry System," AIAA 2021-0935. AIAA Scitech 2021 Forum. January 11-21, 2021, virtual. https://doi.org/10.2514/6.2021-0935

[35] R. G. Wilmoth, R. C. Blanchard, J. N. Moss, “Rarefied Transitional Bridging of Blunt Body Aerodynamics” 21st International Symposium on Rarefied Gas Dynamics,” 21st International Symposium on Rarefied Gas Dynamics, July 26-31, Marseille, France, 1998.

[36] Richard G. Wilmoth, Robert A. Mitcheltree, and James N. Moss, “Low-Density Aerodynamics of the Stardust Sample Return Capsule, ” Journal of Spacecraft and Rockets 1999 36:3, 436-44. https://doi.org/10.2514/2.3464

[37] Alex T. Carroll and Aaron M. Brandis. "Stagnation Point Convective Heating Correlations for Entry into H2/He Atmospheres," AIAA 2023-0208. AIAA SCITECH 2023 Forum. January 2023. Jan. 23–27, 2023, National Harbor, MD. https://doi.org/10.2514/6.2023-0208

[38] Thomas K. West and Aaron M. Brandis. "Updated Stagnation Point Aeroheating Correlations for Mars Entry," AIAA 2018-3767. 2018 Joint Thermophysics and Heat Transfer Conference. June 2018, Atlanta, Georgia, USA. https://doi.org/10.2514/6.2018-3767

[39] Aaron M. Brandis and Christopher O. Johnston. "Characterization of Stagnation-Point Heat Flux for Earth Entry," AIAA 2014-2374. 45th AIAA Plasmadynamics and Lasers Conference. June 2014, Atlanta, Georgia, USA. https://doi.org/10.2514/6.2014-2374

9. Appendix

9.1. Modified Newtonian Aerodynamics: Additional Validation and Convergence

9.1.1. Additional validation with [29], cone and spherical segment

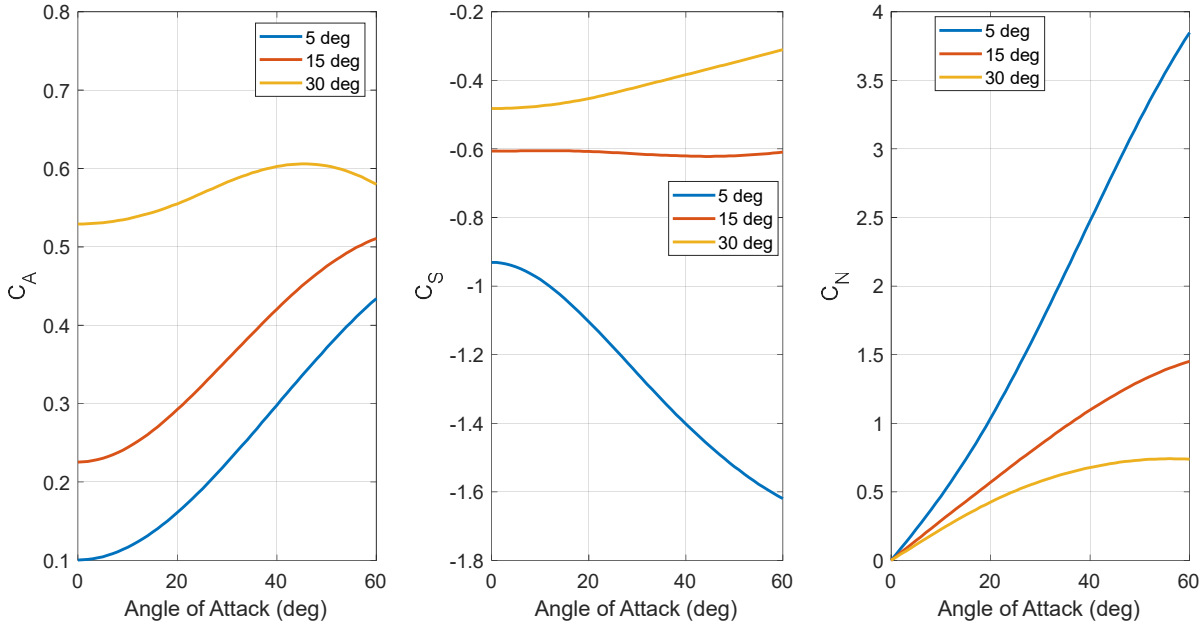


Figure 9.1 Panel Method for Cone (20 radial panels)

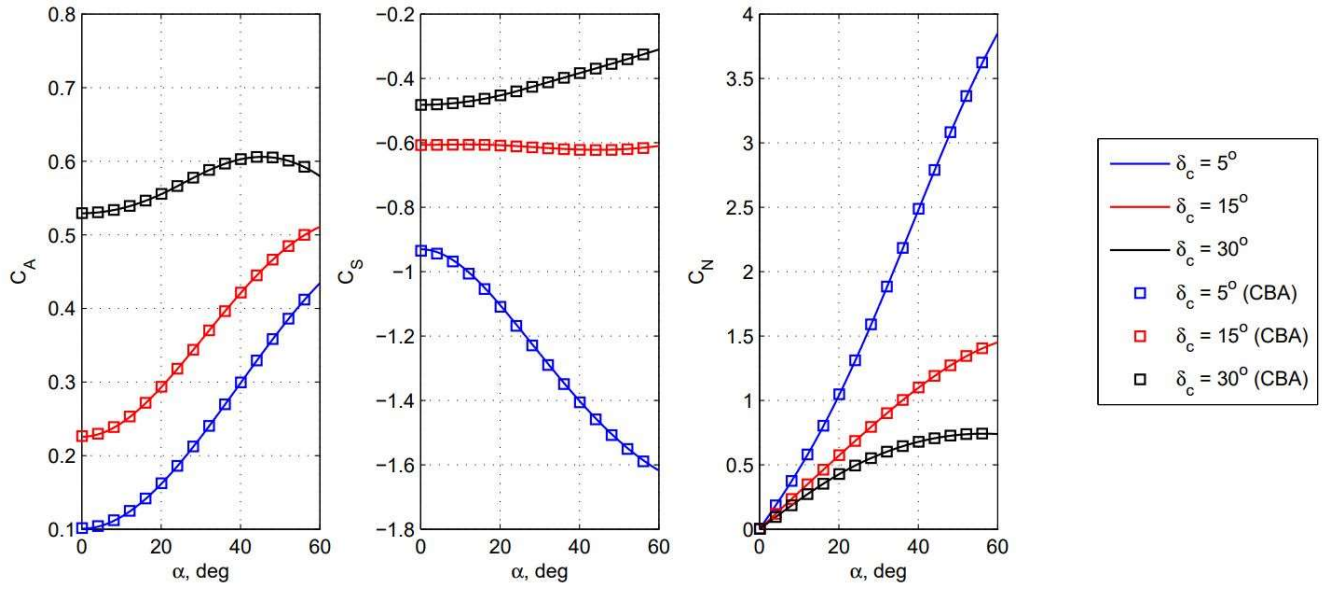


Figure 6. Sharp Cone Force Coefficient Validation, $\beta = 20^o$.

Figure 9.2 Ref [29] Data for Cone

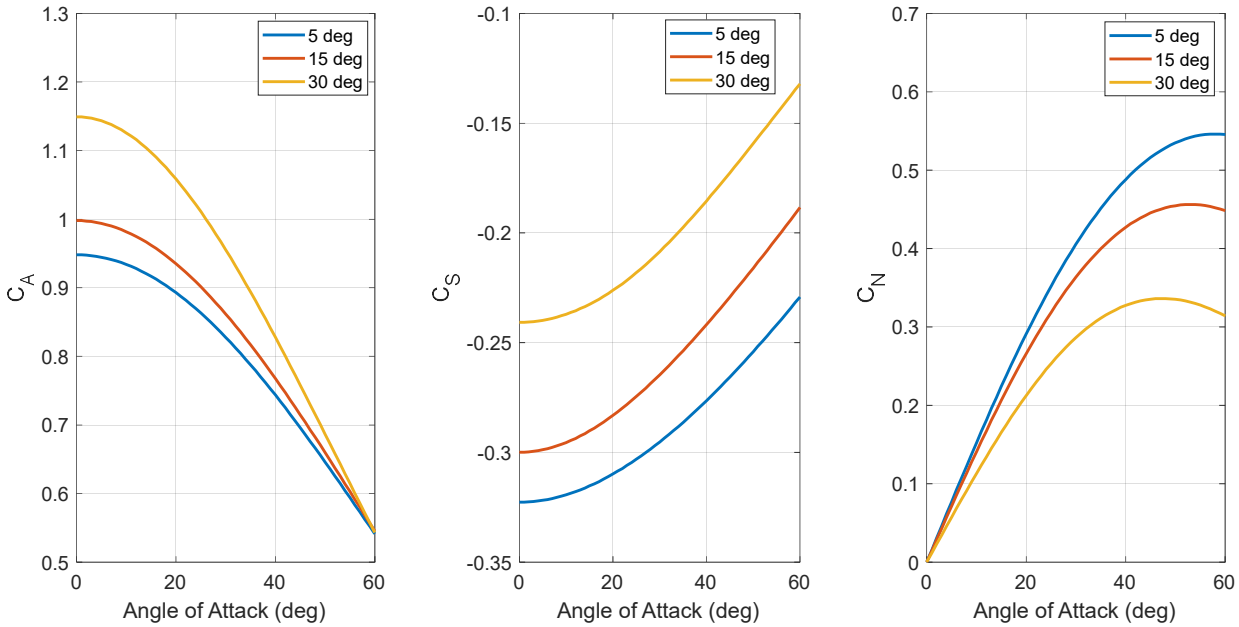


Figure 9.4 Panel Method for Sphere (20 radial panels)

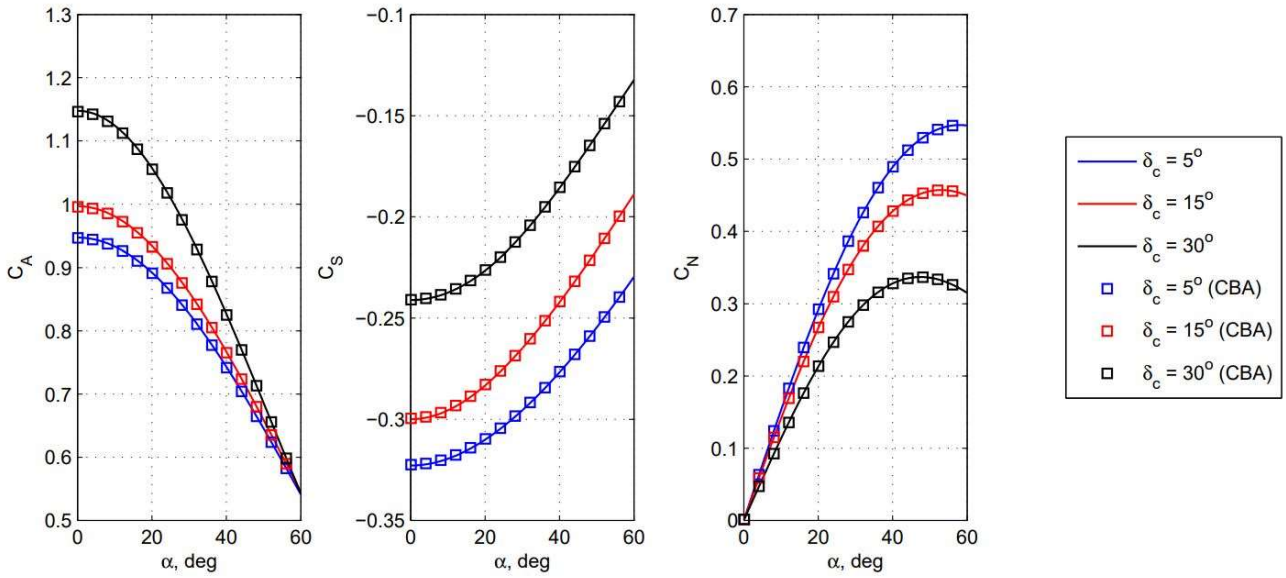


Figure 10. Spherical Segment Force Coefficient Validation, $\beta = 20^{\circ}$.

Figure 9.3 Ref [29] Data for Cone

9.1.2. Panel Method Convergence

A convergence study was conducted on the modified Newtonian panel method discussed in 4.2.2 to assess performance and optimize the number of divisions. Recall lengthwise divisions are applied to curved sections along the x-axis (Figure 4.5) and radial divisions are revolved around the x axis (Figure 4.6). Straight frustrum sections are represented by only one lengthwise division. The solver was setup with a 60° sphere cone and a nose to body radius ratio of 0.5 at $\alpha = \beta = 15°$. The model shows strong convergence performance with the aero coefficients settling within 3 decimal places after just 5 divisions. In this configuration the normal and axial forces depend on both divisions while the side force is dominated by the number of radial divisions. The analysis was repeated for a 25° sphere cone, the results Figure 9.8 illustrate that the performance is slightly worse for more slender bodies but still robust. 8-10 divisions are likely more than sufficient for blunt bodies at low angles of attack. Overall this convergence study adds additional confidence to the aerodynamic model.

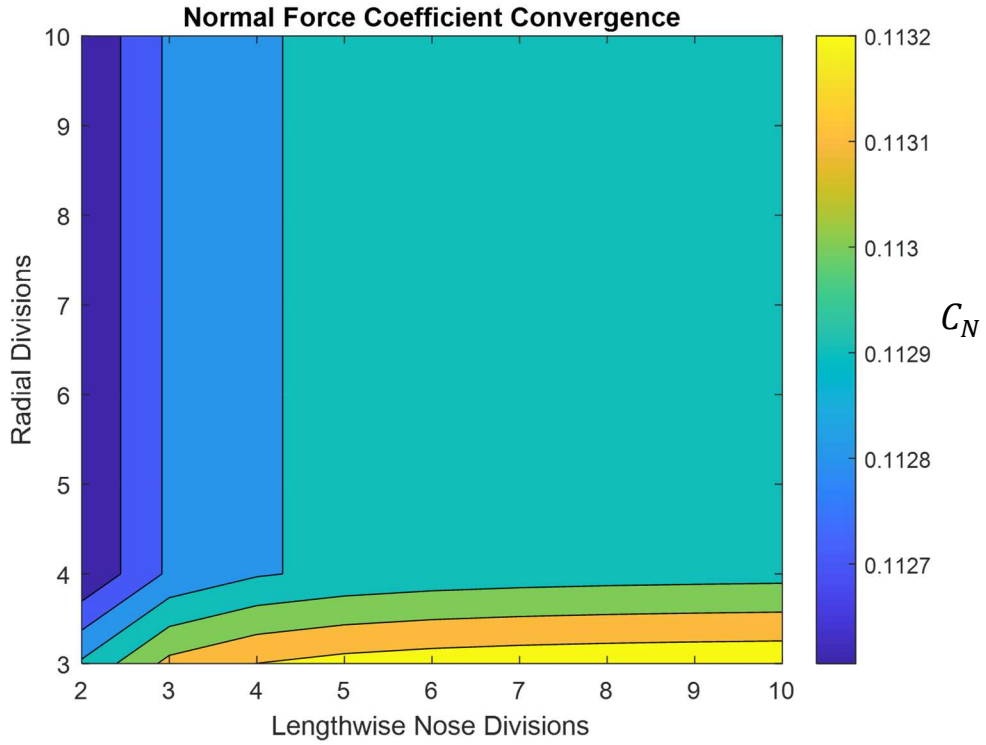


Figure 9.5 Normal Force vs. Number of Divisions (60° sphere cone)

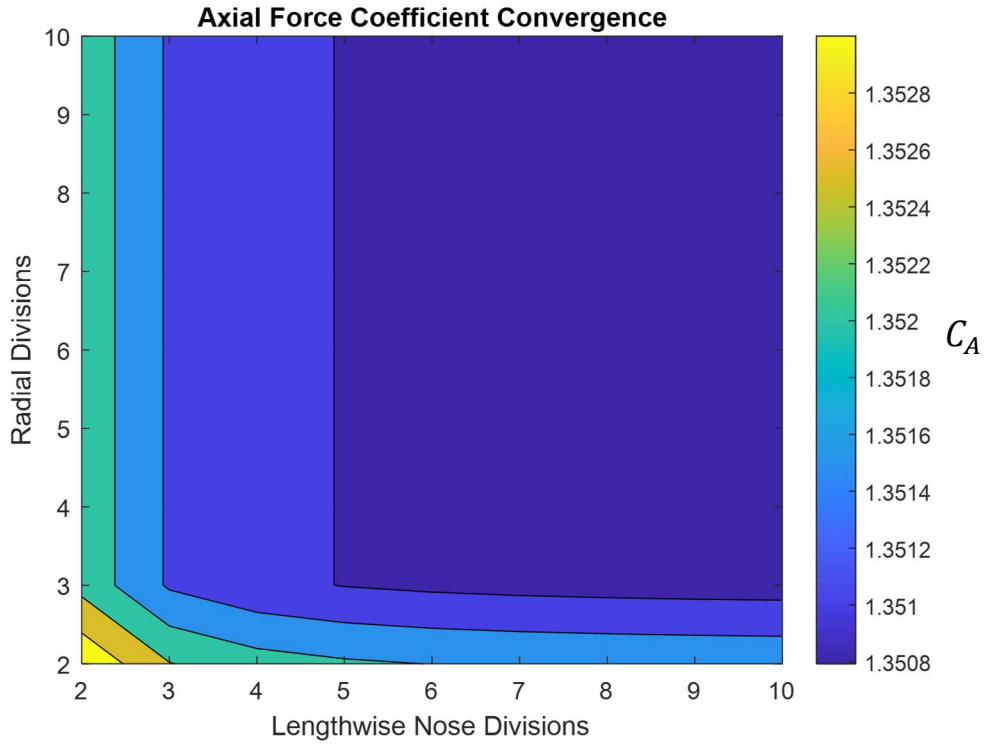


Figure 9.6 Axial Force vs. Number of Divisions (60° sphere cone)

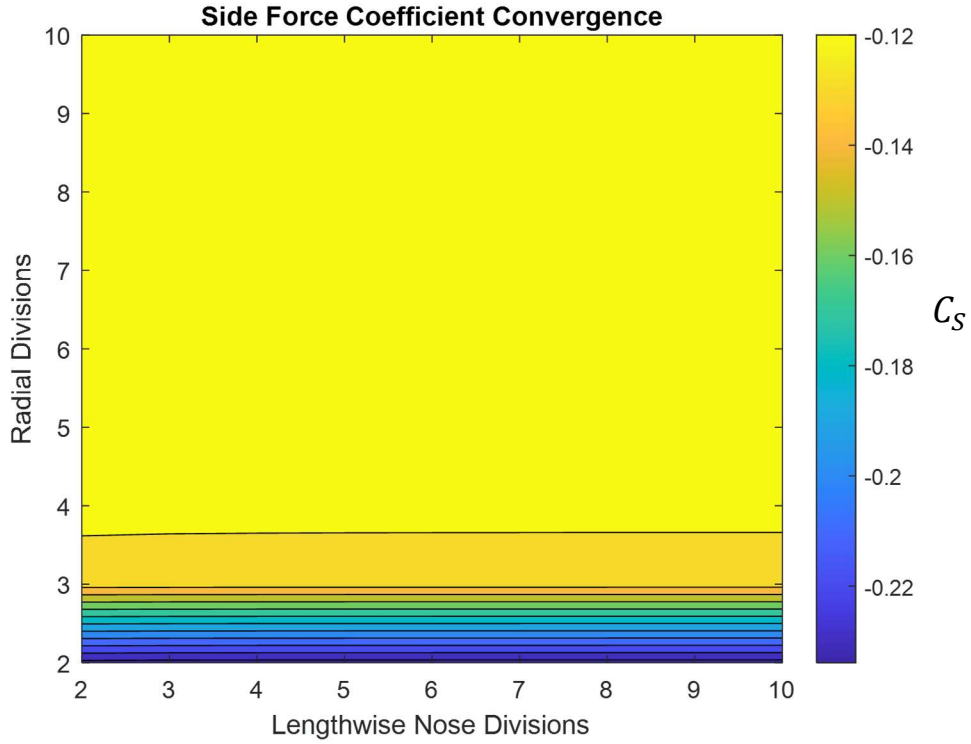


Figure 9.7 Side Force Convergence (60° sphere cone)

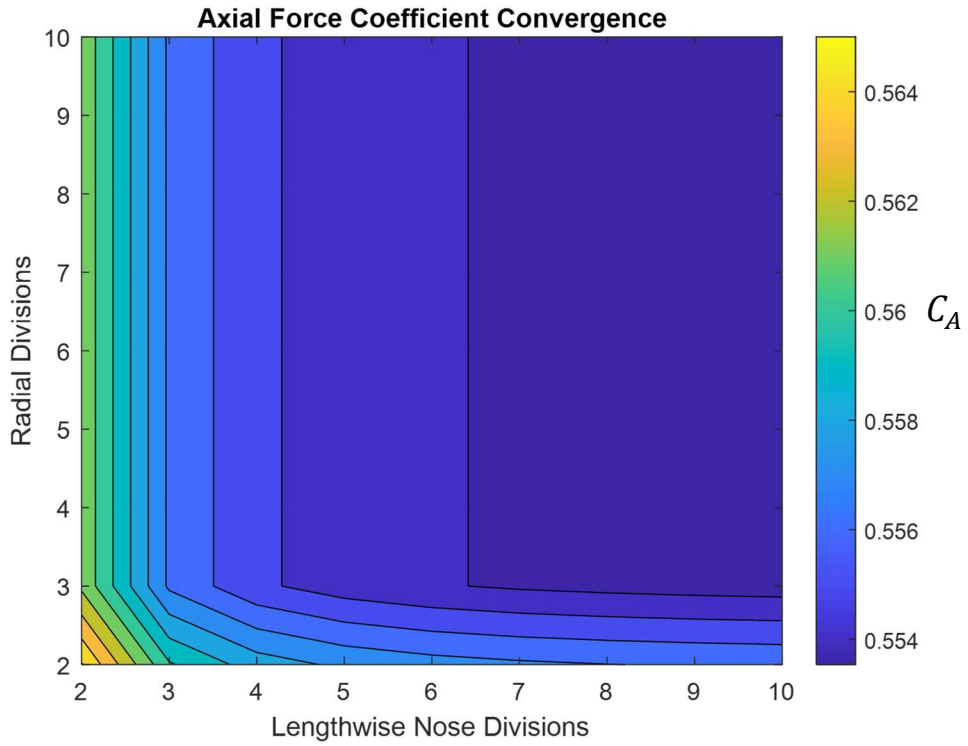


Figure 9.8 Axial Force vs. Number of Divisions (25° sphere cone)

9.2. Free Molecular and Rarefied Flow Aerodynamics

An entry vehicle flying through a planetary atmosphere encounters various flow regimes that vary with altitude. The two primary bounding flow regimes are continuum and free molecular. In the continuum regime, the distance between individual molecules is much smaller than defining features of the flow field allowing it treated as a continuous medium. Continuum flow is described by the Navier Stokes equations and disturbances and direction changes propagate smoothly through the flow field. Free molecular flow is defined by large distances between individual particles where collisions do not propagate or affect adjacent particles. The Knudsen number is a defining nondimensional parameter for free molecular, transitional, or continuum flow where λ is the mean free path and L is a physical length scale, often the entry vehicle diameter.

$$K_n = \frac{\lambda}{L} \tag{9.1}$$

$$\lambda = \frac{k_b T}{\sqrt{2}\pi d^2 p} \tag{9.2}$$

The mean free path can be defined from the Boltzmann constant k_b, temperature, pressure, and the particle kinetic diameter, d, which is available in literature for common substances [17]. There are no strict Knudsen number limits to free molecular and continuum flow but a general guideline is a $K_n < 0.01$ defines continuum flow and $K_n > 10$ defines free molecular flow [17]. These limits often require refinement based on flow geometry and atmospheric characteristics, comparison with empirical data is usually required. The large region between free molecular and continuum flow is known as the transitional or rarefied flow regime and can be difficult to model. Direct simulation monte carlo (DSMC) is a widely used but computationally expensive way of modeling free molecular and rarefied flow by simulating the kinetic collisions and interactions between individual particles.

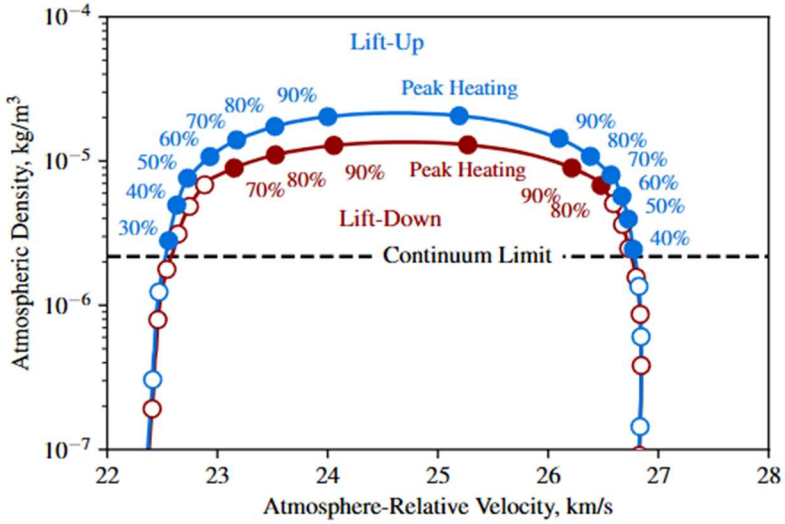


Figure 9.9 Continuum Limit of Uranus Aerocapture Trajectory from [14]

Aerocapture and skip-out trajectories remain at high altitudes and often a large portion of the trajectory is in the rarefied and free molecular flow regimes. As discussed in 4.2, the modified Newtonian method is most accurate for continuum flow. In the search of higher fidelity trajectory modeling an analytical scheme for predicting free molecular and rarefied flow needed to be implemented. [31][32], and [33] describe an approach to modeling aerodynamic coefficients in free molecular flow based on a Maxwellian distribution of spectral and diffuse particle collisions with the vehicle surface. The pressure and shear coefficients are calculated and integrated over the surface. This allows the same paneling algorithm to be reused with the free molecular pressure and shear coefficients as shown in 9.6. The normal vector is the same as is shown in Figure 4.5 and the tangential vector is calculated through 9.7.

$$C_p = \frac{1}{s^2}\left[\left(\frac{2-\sigma_N}{\sqrt{\pi}}\, s\sin(\theta) + \frac{\sigma_N}{2}\sqrt{\frac{T_W}{T_\infty}}\right) e^{-(s\sin(\theta))^2} + \left\{(2-\sigma_N)(s\sin(\theta))^2 + \frac{1}{2} + \frac{\sigma_N}{2}\sqrt{\frac{\pi T_W}{T_\infty}}\, s\sin(\theta)\right\}(1+\operatorname{erf}(s\sin(\theta)))\right] \tag{9.3}$$

$$C_\tau = \frac{\sigma_N \cos(\theta)}{s\sqrt{\pi}}\left[e^{-(s\sin(\theta))^2} + \sqrt{\pi} s\sin(\theta)(1+\operatorname{erf}(s\sin(\theta)))\right] \tag{9.4}$$

$$s = \frac{V_\infty}{\sqrt{2RT_\infty}} \tag{9.5}$$

$$\begin{bmatrix} C_A \\ C_S \\ C_N \end{bmatrix} = \frac{1}{A_{ref}} \iint_S (C_p \hat{\boldsymbol{n}} + C_\tau \hat{\boldsymbol{t}})\, dA \tag{9.6}$$

$$\hat{\boldsymbol{t}} = \frac{\hat{\boldsymbol{n}}(\hat{\boldsymbol{V}}_\infty \cdot \hat{\boldsymbol{n}}) - \hat{\boldsymbol{V}}_\infty}{\sqrt{1 - (\hat{\boldsymbol{V}}_\infty \cdot \hat{\boldsymbol{n}})^2}} \tag{9.7}$$

The Maxwellian distribution model described above applies to free molecular flow and was validated with DSMC in [32] and [33], however this isn't necessarily accurate in the transitional region which is a large part of any entry trajectory. [34] and [35] describe various methods of "blending" the two flow regimes to predict aerodynamic coefficients at any trajectory point as a function of K_n. The theory states that a local or global aerodynamic coefficient can be described as a weighted average of that coefficient at the free molecular and continuum limits. A commonly used bridging function based on the sine squared law is shown in 9.8-9.10, $P_b = 1$ at the free molecular limit and 0 at the continuum limt. The two constants a_1 and a_2 are related to the free molecular and continuum K_n values and are calculated with 9.11. More accurate bridging functions exist though they require a DSMC anchor point at the middle of the transitional regime. This sine squared function was used for the MATLAB trajectory code aerodynamics database.

$$C_{tr} = P_b C_{fm} + (1 - P_b) * C_{cont} \tag{9.8}$$

$$P_b = \sin^2 \psi \tag{9.9}$$

$$\psi = \pi(a_1 + a_2 log10 K_{n_\infty}) \tag{9.10}$$

$$\begin{bmatrix} 0.5 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & log10 K_{n_{cont}} \\ 1 & log10 K_{n_{fm}} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} \tag{9.11}$$

9.2.1. Validation of Rarefied Flow Mechanics

With the relationships established for free molecular and transitional flow mechanics, test cases can be run and the results can be compared/validated with literature. The existing paneling method for Newtonian flow is easily utilized for the free molecular conditions and allows for a variety of vehicle shapes to be quickly tested. [32] and [33] present the exact equations 9.3-9.7 and an initial simple test of the drag coefficient of a sphere while varying the velocity, wall temperature, and specular and diffuse ratio to assess sensitivity. With each comparison, parameters were set to the same values as the reference literature where possible. Comparison of Figure 9.10Figure 9.11indicates the results are in family, however [32] does not present values used for the specific gas constant which is required for 9.5, so a value of 287.058 J/(kg-K) for air was assumed.

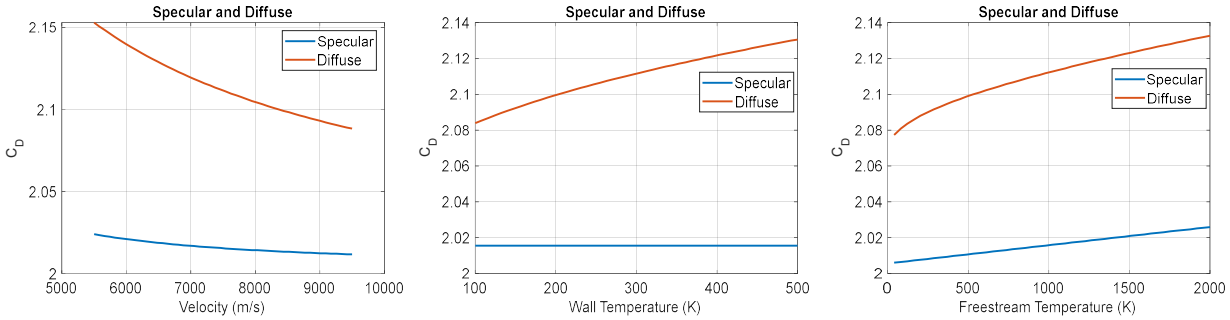


Figure 9.11 Sphere Drag Coefficient Validation Results

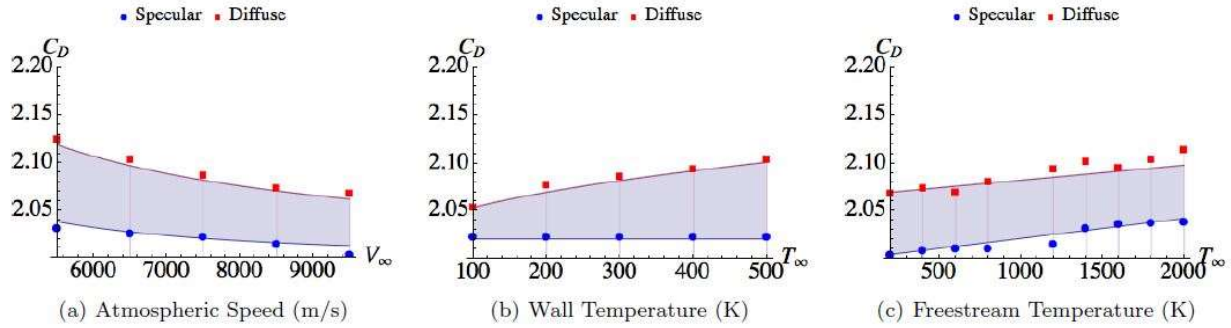


Figure 9.10 Sphere Drag Coefficient Results from [32]

Ref. [33] was written by some of the same SME's and they go into greater detail on plotting more complex geometries. The results for the decreasing bi-conic were replicated in the internally developed simulation and the results indicate excellent agreement with [33]. While T_∞, T_W, and V_∞ were provided, R was not and was again set to the standard value for air. [33] also presents results from the sphere-cone geometry of the mars microprobe and the results agree up to around 45° where the spherical backshell becomes exposed to the flow. This backshell is not modeled in this comparison case as the panel solver currently only supports a single spherical segment on the nose of the body followed by straight frustrum segments. Angles of attack were only run out to 90° due to this limitation. Development is underway to allow for any number of frustrum or radiused segments to be superimposed to allow a much wider range of axisymmetric bodies to be modeled. Pre-generated X and Y lengthwise coordinates can also be provided to the solver to create a revolved grid.

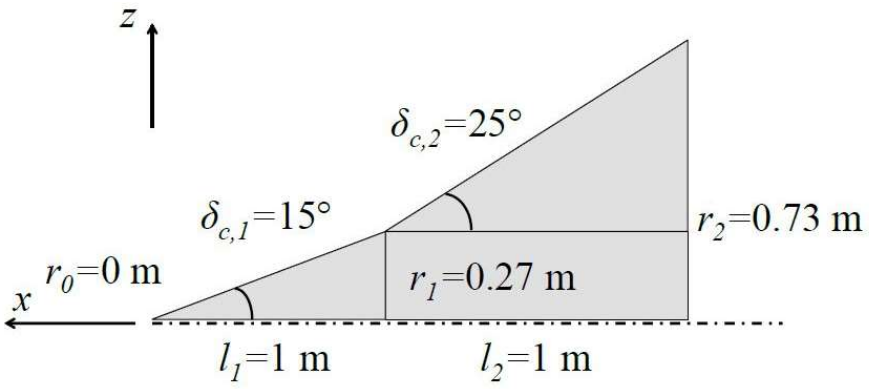


Figure 9.12 Increasing Bi-conic Geometry from [33]

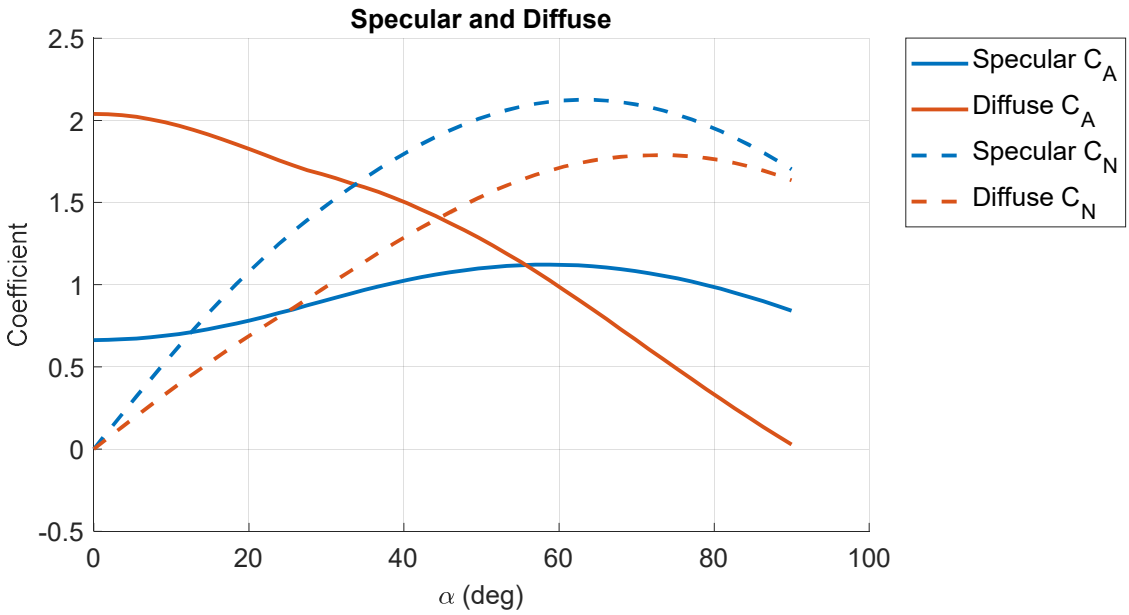


Figure 9.13 Increasing Bi-Conic Validation Results

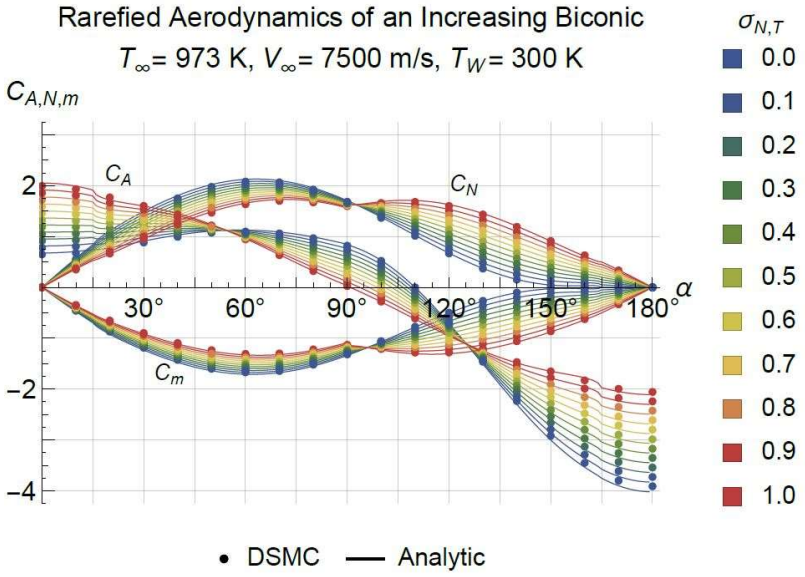


Figure 9.14 Increasing Bi-conic Results from [33]

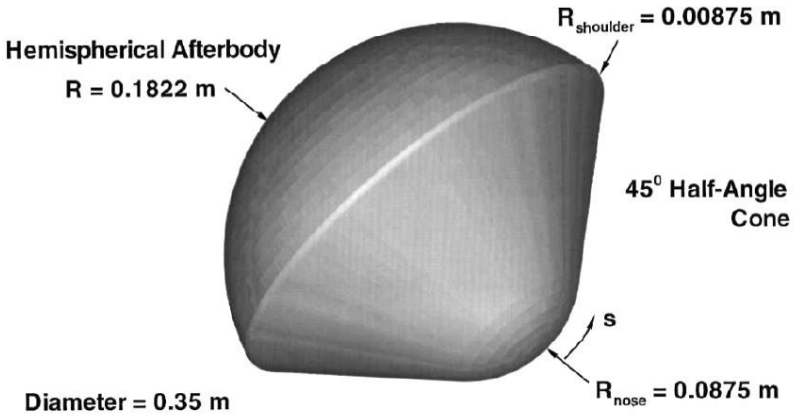


Figure 9.17 Mars Microprobe Geometry from [33]

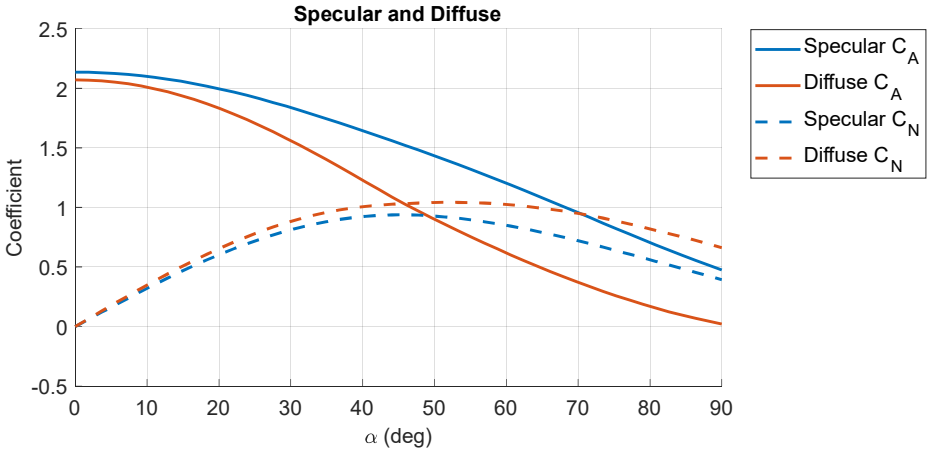


Figure 9.16 Mars Microprobe Validation Results

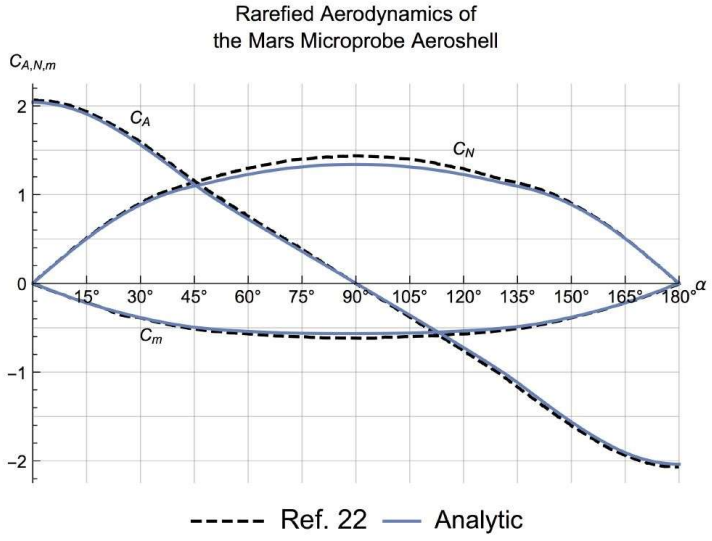


Figure 9.15 Mars Microprobe Results from [33]

Prediction of the Knudsen number vs. altitude was necessary for implementation of the bridging function which integrates the continuum and free molecular flow mechanisms. [36] studies the low density aerodynamics of the stardust sample return vehicle and was used to validate the rarefied flow aerodynamics. For gas mixtures with multiple species, higher fidelity tools like DSMC will often use complex collision mechanics like the variable soft sphere model (VSS) to determine the mean free path which is required for the Knudsen number. In the aerodynamics modeling for this project the assumption is made that the mean free path is based on a weighted average of the species particle kinetic diameters (d_i) and mole fractions (X_i). An expansion of 9.2 is for gas mixtures can be simplified as GRAM can output the particle number density directly. More accurate methods of calculating the mean free path exist but are more computationally expensive. 9.12 – 9.14 were tested against the results in [36]. EarthGRAM was setup with the same time and position of the stardust landing of January 15th, 2006 and an approximate longitude and latitude of the Utah desert landing site. Comparison of Figure 9.18 and Table 9.1 show some non-negligible differences in the atmospheric properties from [36] and the GRAM output. The Knudsen number was re-calculated based on the conditions from [36] to get a true comparison of the mean free path calculation methods. There is variation in the absolute values of the Knudsen numbers between the two methods with the weighted average calculation trending high by up to 50%. Considering the multiple orders of magnitude of the Knudsen number scale within the transitional regime, the differences seen here are acceptable. Validation of the sine squared bridging function is shown in Figure 9.19 and Figure 9.20 with free molecular and continuum Kn bounds of 10 and 0.001. Comparison with the results from [36] indicates strong agreement of the aero-coefficients vs. Kn. This adds confidence to the rarefied flow techniques developed thus far, especially given the high-fidelity tools used in [36].

$$n = \frac{p}{k_b T} \tag{9.12}$$

$$\sigma = \sum_{i=1}^{m} d_i^2 X_i \tag{9.13}$$

$$\lambda = \frac{1}{\sqrt{2}\pi\sigma} \tag{9.14}$$

Altitude, km	Velocity, m/s	Number Density, $1/m^3$	Mole Fractions O_2	N_2	O	T_∞, K	T_w, K	Knudsen Number	Mach Number
134.75	12,597.1	1.48832×10^{17}	0.0659	0.6716	0.2625	577.23	400	12.8	23.96
120.45	12,607.7	5.93941×10^{17}	0.0845	0.7327	0.1828	381.15	500	2.92	30.27
100.90	12,620.2	1.09988×10^{19}	0.1768	0.7844	0.0388	199.37	1,000	0.136	43.94
92.00	12,618.5	4.98474×10^{19}	0.2056	0.7873	0.0071	202.05	1,200	0.0301	44.10
83.68	12,591.5	1.77978×10^{20}	0.2385	0.7615	0.0000	216.54	1,500	0.00857	42.68
75.98	12,486.8	5.73854×10^{20}	0.2385	0.7615	0.0000	218.14	1,800	0.00266	42.17

Figure 9.18 Flight Conditions from [36]

Table 9.1 Knudsen Number Validation with [36]

Altitude (km)	Number Density $(1/m^3)$	O_2	N_2	O	T_∞ (K)	K_N	K_N [36] conditions
134.75	1.133222e+17	0.0434	0.6175	0.3376	648.63	19.2	14.58
120.45	3.840013e+17	0.0715	0.6866	0.2393	409.33	5.6	3.63
100.90	1.027121e+19	0.1537	0.7847	0.0543	188.14	0.208	0.194
92.00	4.976260e+19	0.1890	0.7967	0.0057	197.25	0.0428	0.0428
83.68	1.807716e+20	0.2038	0.7861	2.5e-4	213.11	0.0118	0.0120
75.98	5.926458e+20	0.2094	0.7808	2.5e-5	216.49	0.0036	0.0037

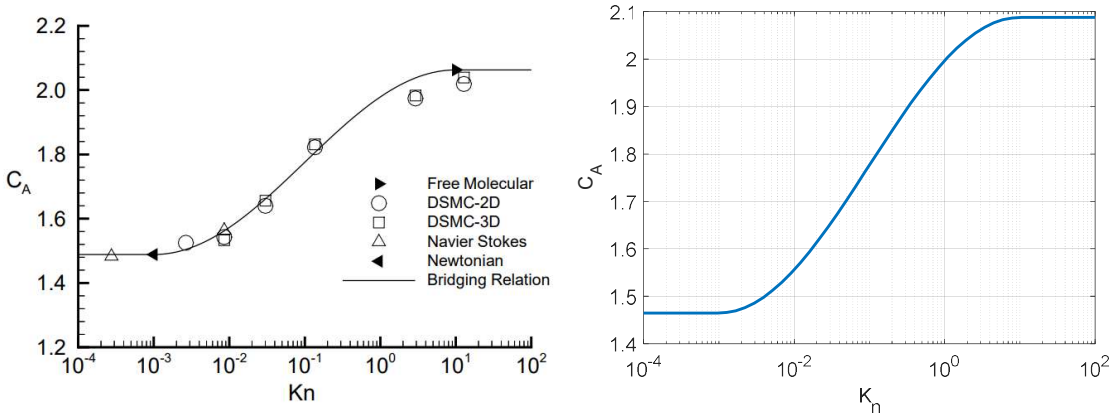
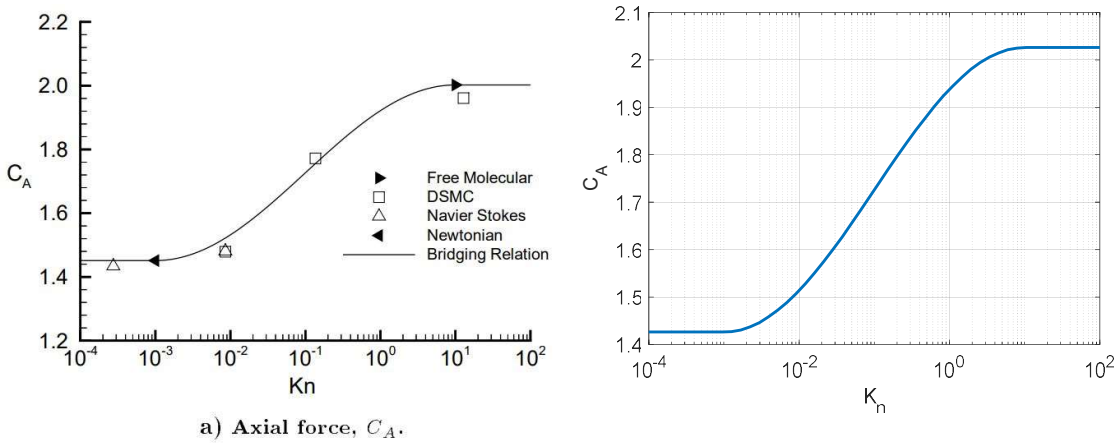
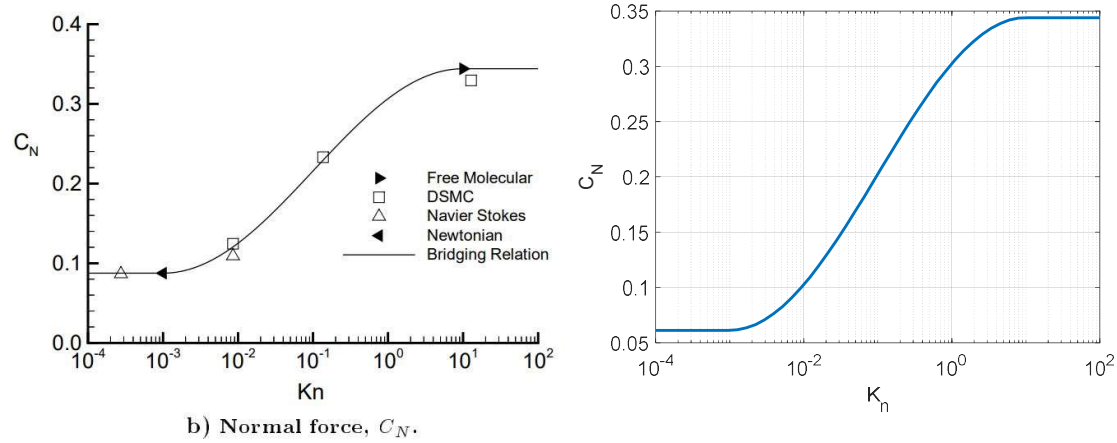


Figure 9.20 Validation with [36] at α=0°



a) Axial force, C_A.



b) Normal force, C_N.

Figure 9.19 Validation with [36] at α=10°

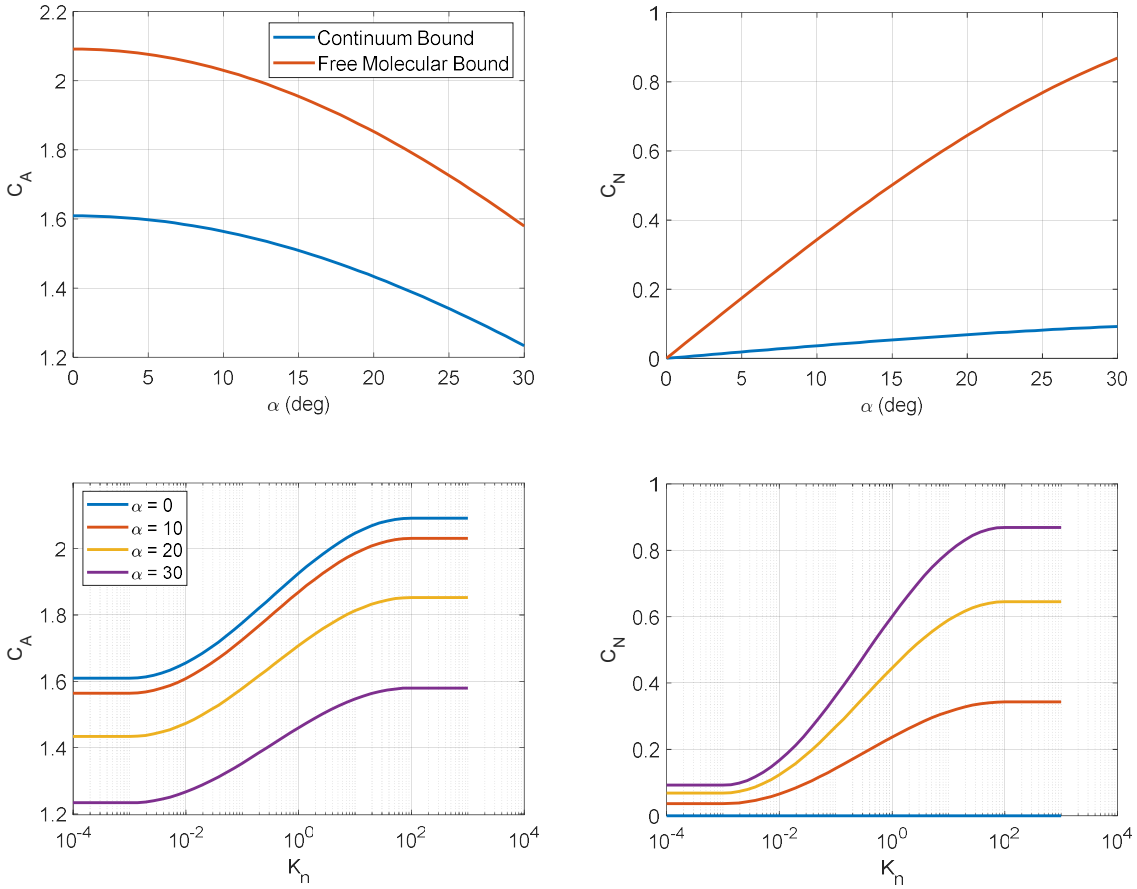


Figure 9.22 Aero-database Validation with [13]

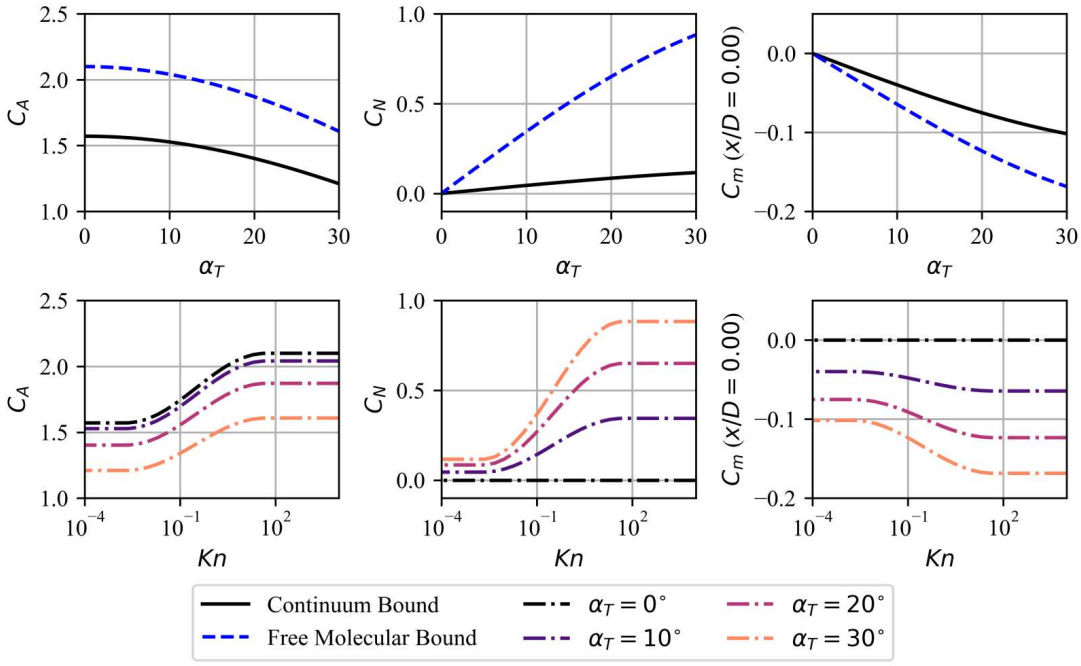


Figure 9.21 Aero-database from [13]

The final validation of the rarefied flow aerodynamics database is with the NASA ECI work studying Uranus aerocapture outlined in [9]-[15]. The study utilizes a MSL like vehicle with a 70° sphere cone and a combination of modified Newtonian and empirical flight data for aerodynamics. Reference [13] lays out the aerodynamics implications for Uranus aerocapture and presents results on the aero-coefficients vs. angle of attack and Knudsen number. These figures were replicated to generate a reasonable comparison. In [13], the transitional flow regime is extended to free molecular and continuum bounds of 100 and 0.001. While the exact numbers from [13] were not obtained, visual comparison of the plots in Figure 9.21 and Figure 9.22 indicate strong agreement. The free molecular flow characteristics modeled in equations 9.3-9.7 match up well with a variety of other data sources including DSMC and flight data. Moment coefficients have not been modeled as part of this study as the focus is strictly on 3DOF however adding them in the future is straightforward.

Figure 9.23 shows a comparison trajectory aerocapture trajectory targeting 500 km with rarefied flow effects enabled and disabled, the vehicle and entry state are the same as in Table 9.2 and Table 9.4 but with α=0°. The trajectory dispersion effects are amplified with decreasing vehicle ballistic coefficients though they are still small. The slightly higher drag from the free molecular flow effects results in a slightly shallower entry angle to achieve the same target apoapsis altitude of 500 km.

α=0°	q_{max} (W/cm^2)	J_s (J/cm^2)	EFPA (deg)
Rarefied flow on	96.567	7860	-5.168
Rarefied flow off	97.631	7941	-5.172

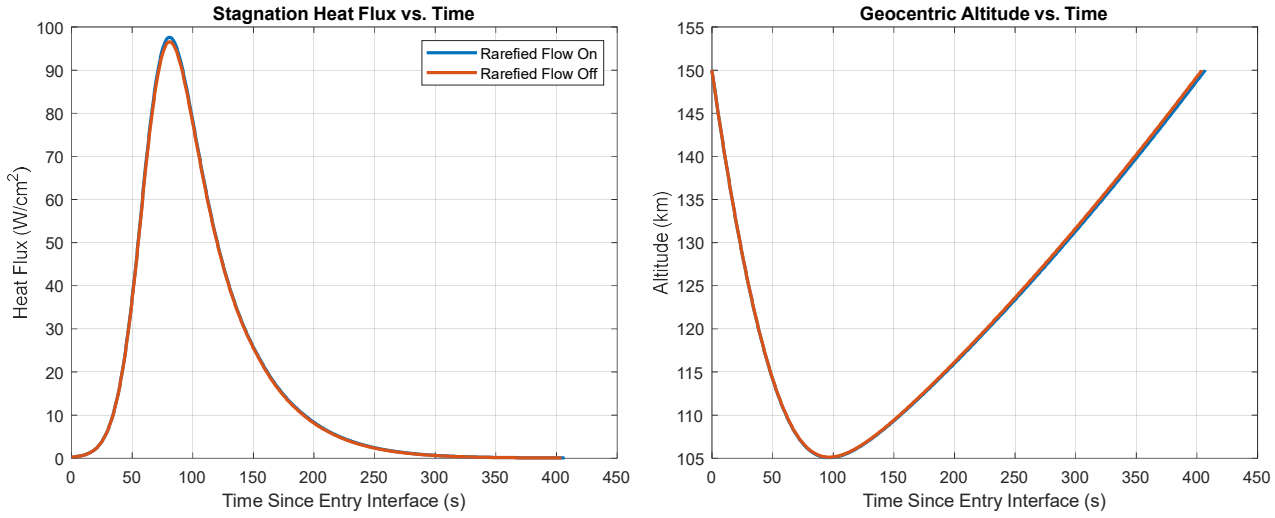


Figure 9.23 Rarefied Flow Effects Comparison

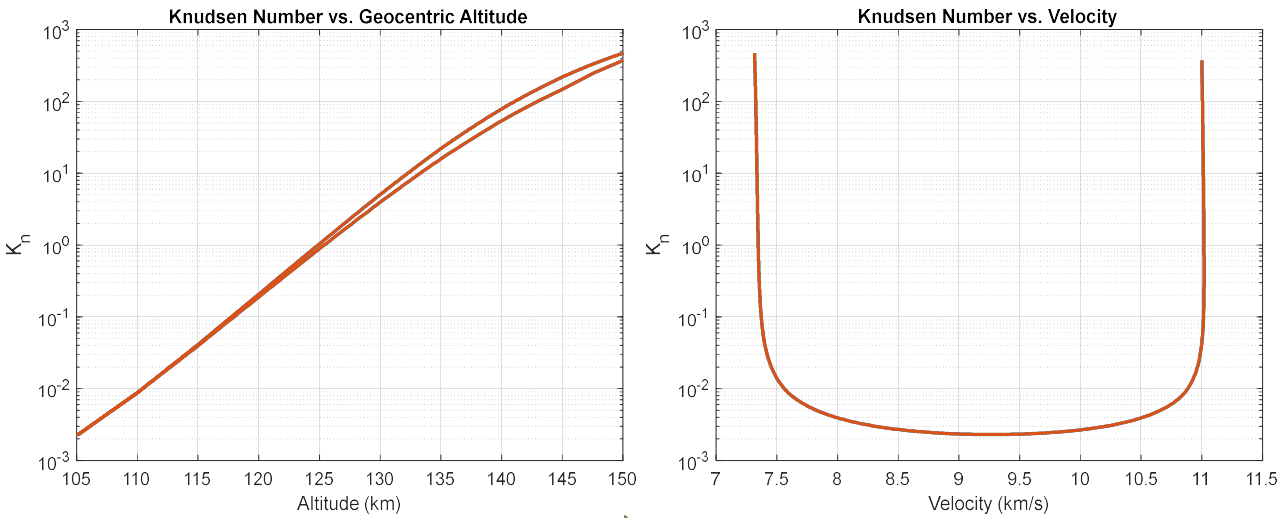


Figure 9.24 Knudsen Number Trajectory Space

9.3. Ultra Low Ballistic Coefficient Venus Multi-Pass Trajectory

Table 9.2 Spacecraft Inputs (Ultra Low B_C)

Spacecraft Input Parameters	
Parameter	**Value**
Initial Mass	50 kg
High Thrust System	100 N
Low Thrust System	4 N
ISP (both systems)	300 s
Drag Coefficient α=-10	1.5332
Lift Coefficient α=-10	-0.2337
Diameter	3 m
Nose Radius	0.75 m
Sphere Cone Angle	70°

Table 9.3 Con-Ops Summary (Ultra Low B_C)

Maneuvers Summary		
Maneuver	**Duration (s)**	**ΔV (m/s)**
Interplanetary TCM	6.576	0.526
1st Perigee Adjust	0.124	0.01
2nd Perigee Adjust	0.675	0.054
3rd Perigee Adjust	2.038	0.163
4th Perigee Adjust	5.817	0.466
Final Perigee Raise	52.34	106.64
Totals		
Mission Duration (days)		17.79
ΔV Expenditure (m/s)		107.86
Propellant Expenditure (kg)		1.8

Table 9.4 Initial State

Interplanetary Orbital Elements	
Parameter	**Value**
Eccentricity (e)	1.3074
Semi Major Axis (a)	-2.0005e4 km
Inclination (i)	0°
Argument of Periapsis (ω)	0°
Long. of Ascending Node (Ω)	0°
True Anomaly (θ)	137°
Hyperbolic Excess Velocity	4.0298 km/s
Julian Date	2455504
Post Capture Apoapsis (km)	500000

Table 9.5 Atmospheric Entries

Aero-Pass Summary			
Pass	**q_{max} (W/cm^2)**	**J_s (J/cm^2)**	**ΔV (km/s)**
Insertion	61.66	4416	0.916
1st Pass	46.05	4085	0.915
2nd Pass	32.30	3903	0.914
3rd Pass	19.22	4445	0.912

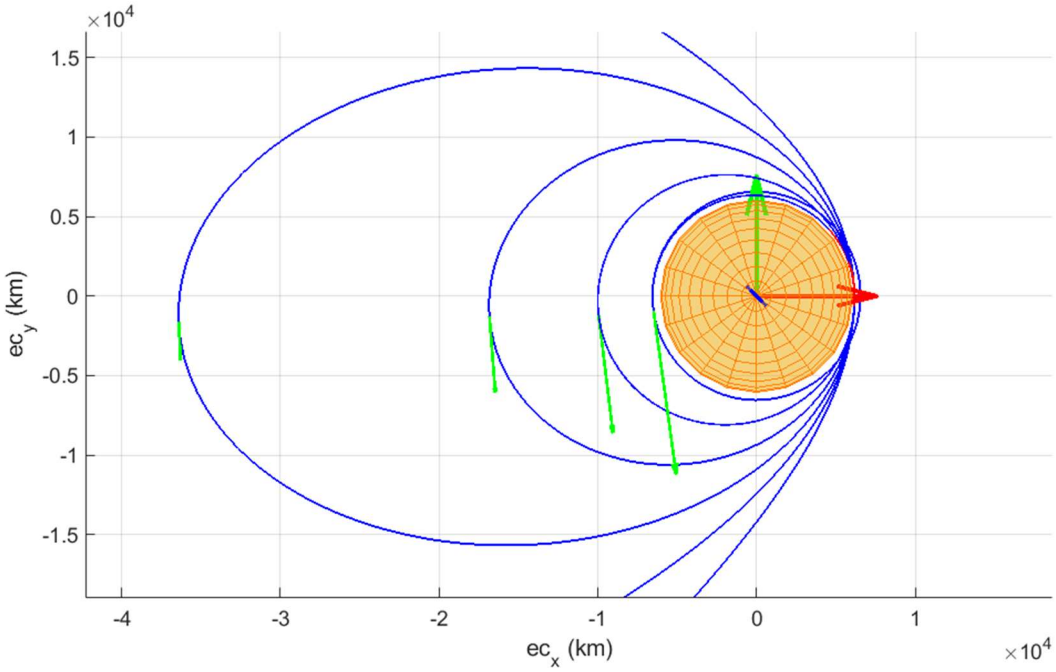


Figure 9.25 Multi-Pass Trajectory, Venus Deployable TPS, Ultra Low B_C

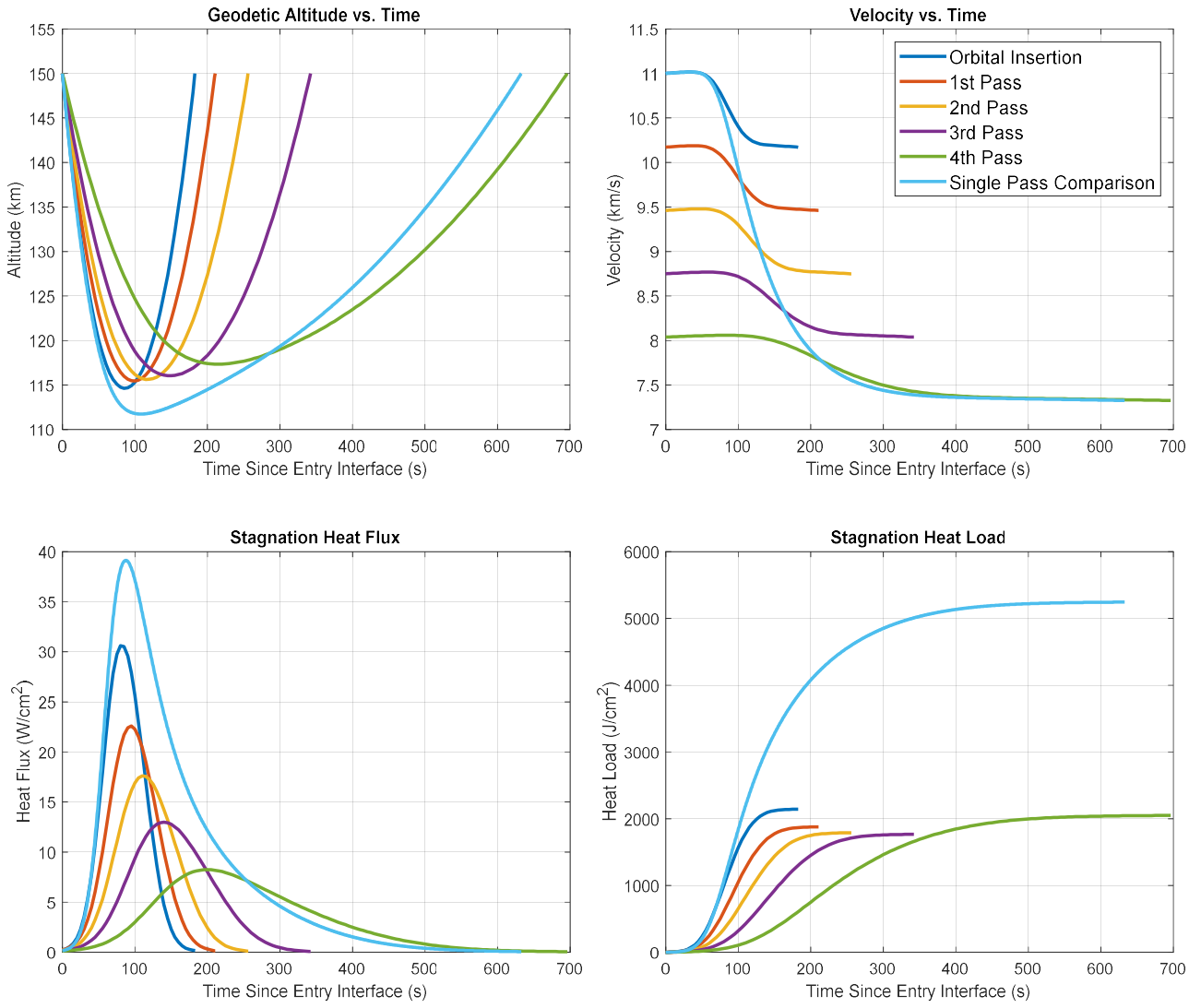


Figure 9.26 Aerothermal Results: Deployable TPS, Ultra Low B_C (Lift Down Only)

9.4. Uranus Aerocapture Analysis

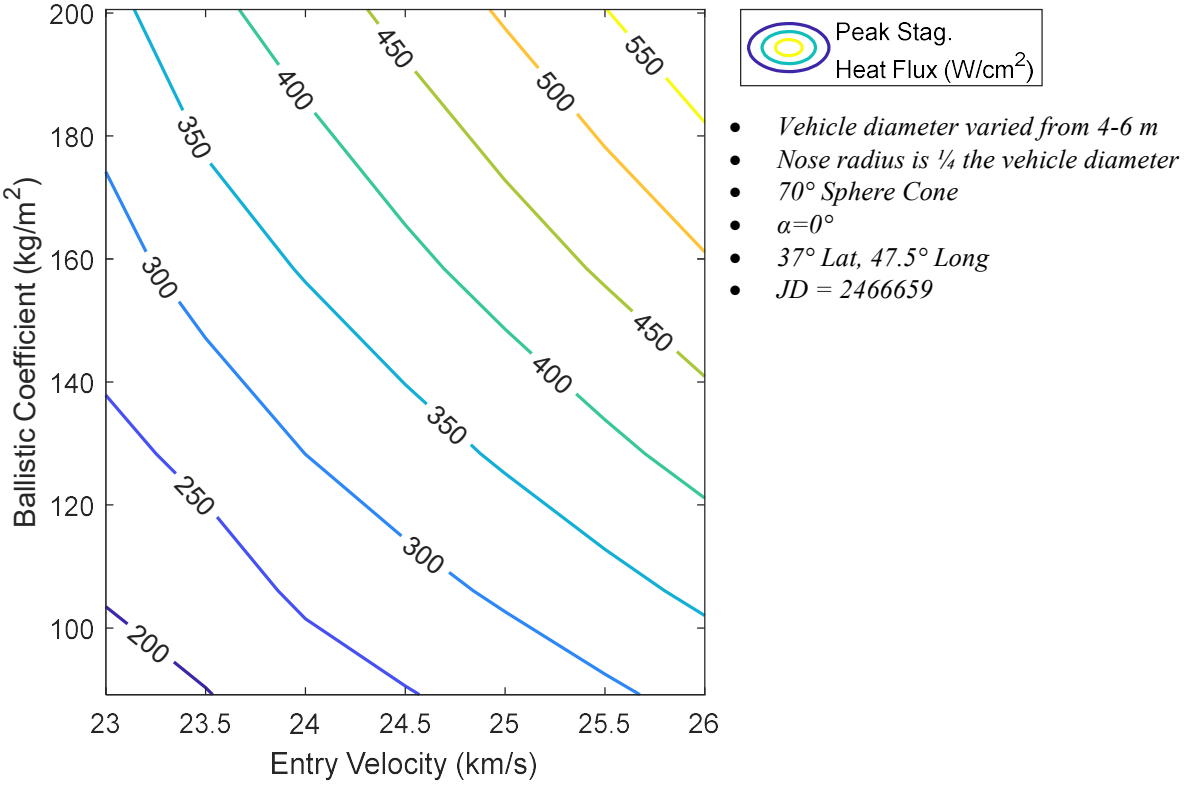


Figure 9.27 Uranus Aerocapture Design Space

The species to be modeled for the Uranus entry are H_2, He, H, $H+$, and e-, $He+$ was initially considered though initial CEA testing showed mole concentrations well below 1e-5 up to 12000 K so it was dropped for simplicity. The effects of chemical reactions on the vehicle aerodynamics was evaluated at one trajectory point close to the time of peak heating, the results are shown in Table 9.7. While the effect on the aero-coefficients is non-negligible, there is still a computation time penalty despite the optimized symbolic math expressions. The iterative method utilizing equations 4.32 and 4.33 requires a high number of thermodynamic property evaluations through equations 4.48 and 4.49. The object oriented simulation architecture allows for numerous physics effects to be turned on and off and tolerance properties for the various iterative methods to be adjusted. The most efficient trajectory design method involves turning most of the higher fidelity effects off for batch runs and trajectory optimization and then re-enabling them for the final trajectory design and aerothermal analysis. The rarefied flow and chemical equilibrium effects have the highest impact on the computation time of one trajectory point.

Table 9.6 Uranus Test Trajectory Inputs

Spacecraft Input Parameters		**Entry State**	
Initial Mass	4064 kg	Inertial Velocity	24.5 km/s
Trim Angle of Attack (α)	17°/-17°	Lift up EFPA	-23.806°
C_D at α=17°/-17°	1.436	Lift down EFPA	-23.509°
C_L at α=17°/-17°	$\pm$0.378	Geocentric Altitude	4000 km
Diameter	5 m	Longitude (deg)	47.5°
Nose Radius	1.25 m	Azimuth (deg)	-90°
Sphere Cone Angle	70°	Geocentric Latitude	37°
B_C at α=0°	128.4 kg/m^2	Target Apoapsis Altitude	500000 km
B_C at α=17°/-17°	144.1 kg/m^2	Atmosphere Cutoff	4000 km

Table 9.7 Chemically Reacting Flow Effects on Aerodynamics for Uranus Aerocapture Trajectory

α=17°, alt = 310 km, Mach=26.93	Calorically Perfect (γ=1.45)	Chemical Equilibrium (H_2/He)
C_A	1.4836	1.5840
C_N	0.0587	0.0627
C_D	1.4360	1.5331
C_L	0.3776	0.4032

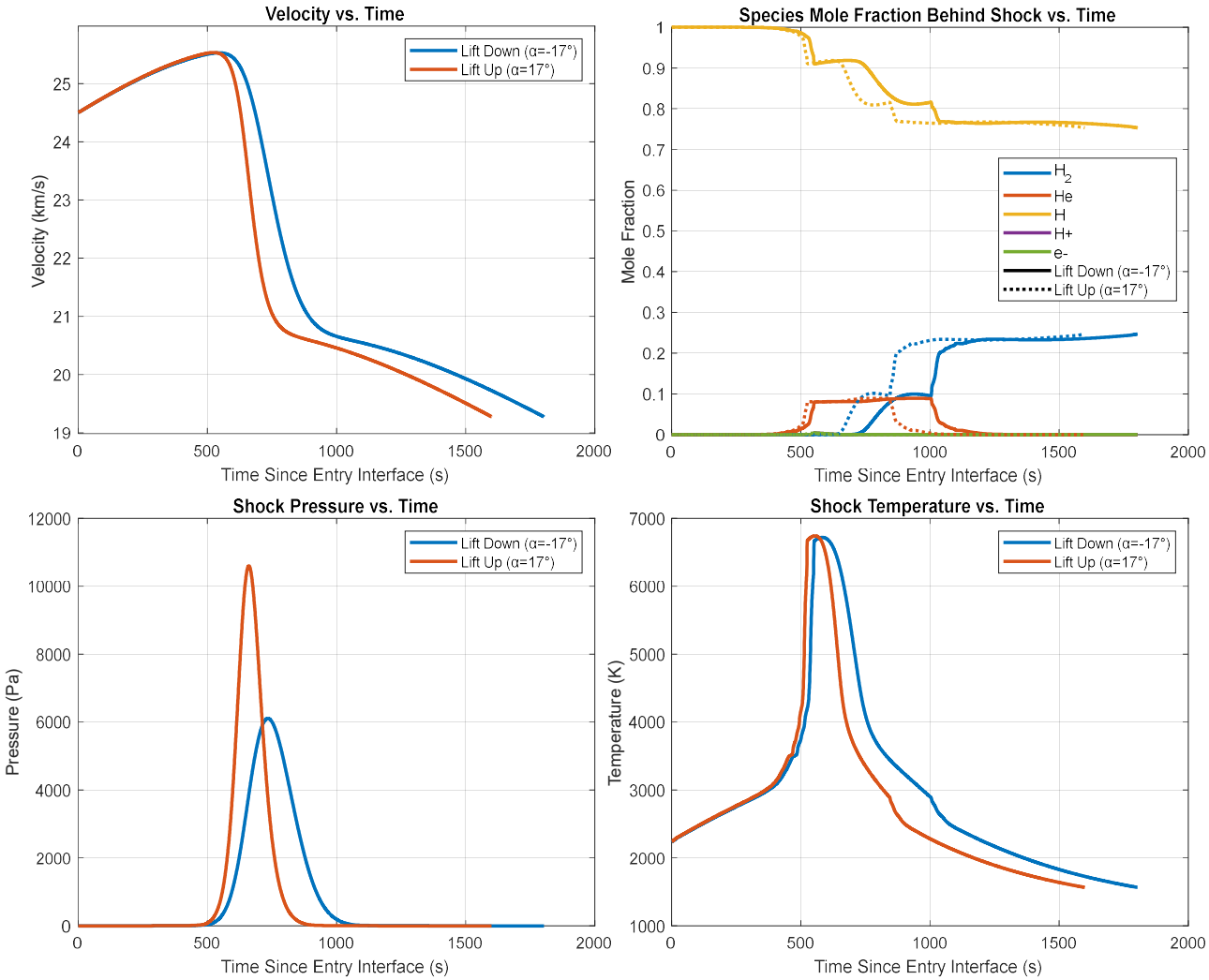


Figure 9.28 Uranus Test Trajectory Results

Figure 9.28 illustrates several abrupt changes in the species mole fraction and temperature behind the stagnation shock front. This appears unusual at first but after inspection of the nominal mole fractions of H_2 and He vs. altitude from UranusGRAM, much of the abrupt changes correspond with the onset of higher Helium concentrations around 500km. There is a small amount of atomic hydrogen ionization that occurs around 550 s for the lift down trajectory and 525 s for the lift up with peak mole fractions of e- and H+ of ~0.005. At the onset of entry interface, virtually all the H_2 dissociates though at the lower velocities close to atmospheric exit only around 75% of the hydrogen is dissociated. For the aerothermal heating results, modern stagnation heating correlations for H_2/He atmospheres [37] were employed utilizing the equilibrium chemistry results and compared to the standard Sutton Graves correlation (eq. 4.31). The results match up surprisingly well though additional evaluation is necessary over a wider trajectory space (Figure 9.31). For the analysis in sections 5 and 6, only the Sutton Graves correlation was used.

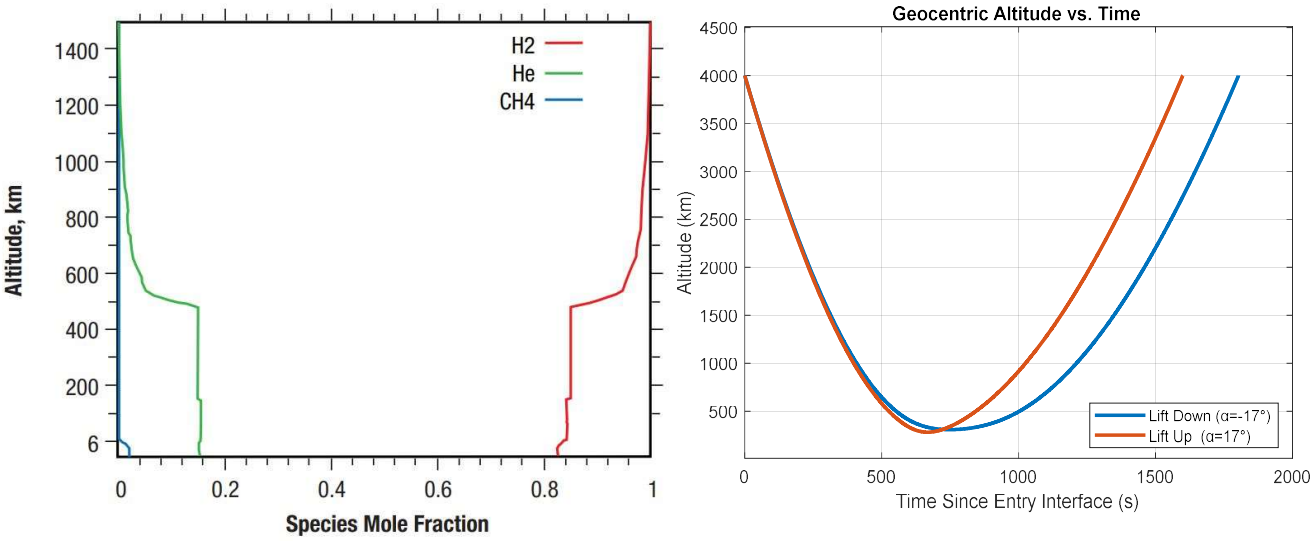


Figure 9.29 UranusGRAM [6] Species Mole Fractions vs. Altitude

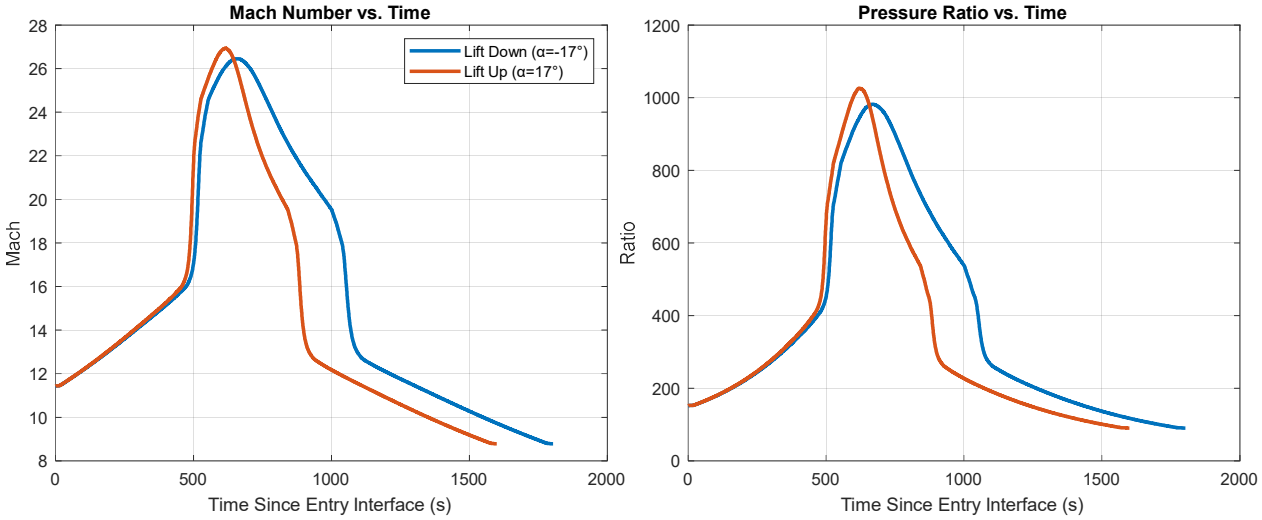


Figure 9.30 Uranus Test Trajectory Normal Shock Effects

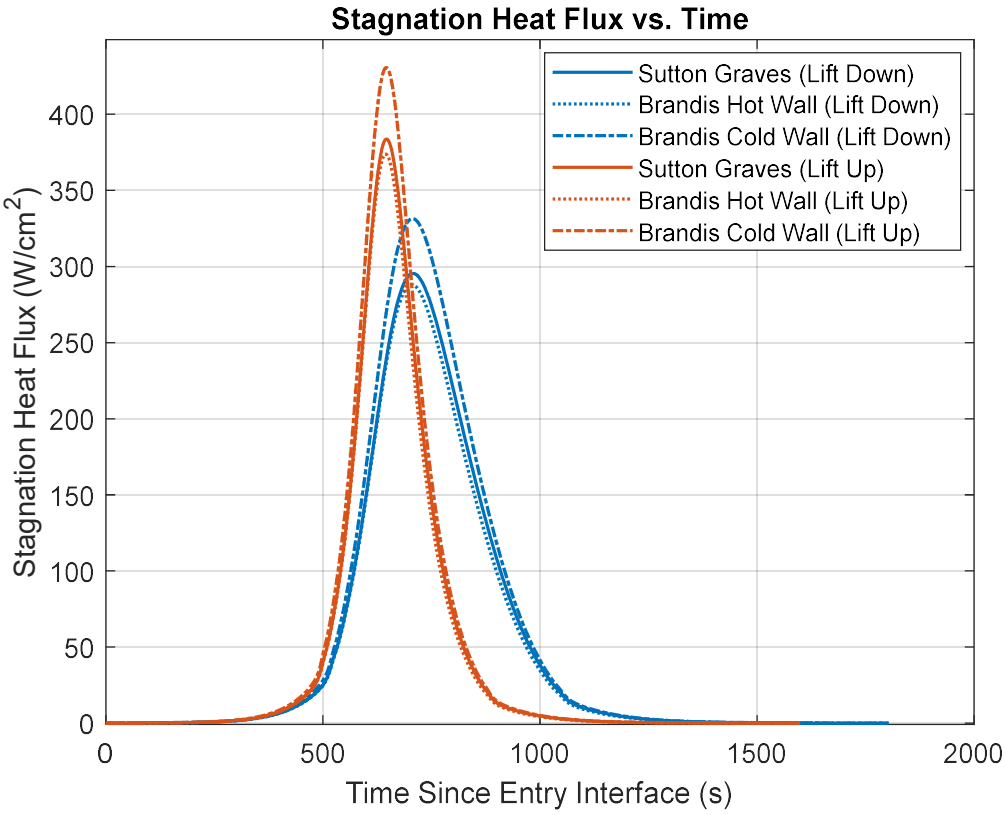


Figure 9.31 Convective Heating Correlation Comparison

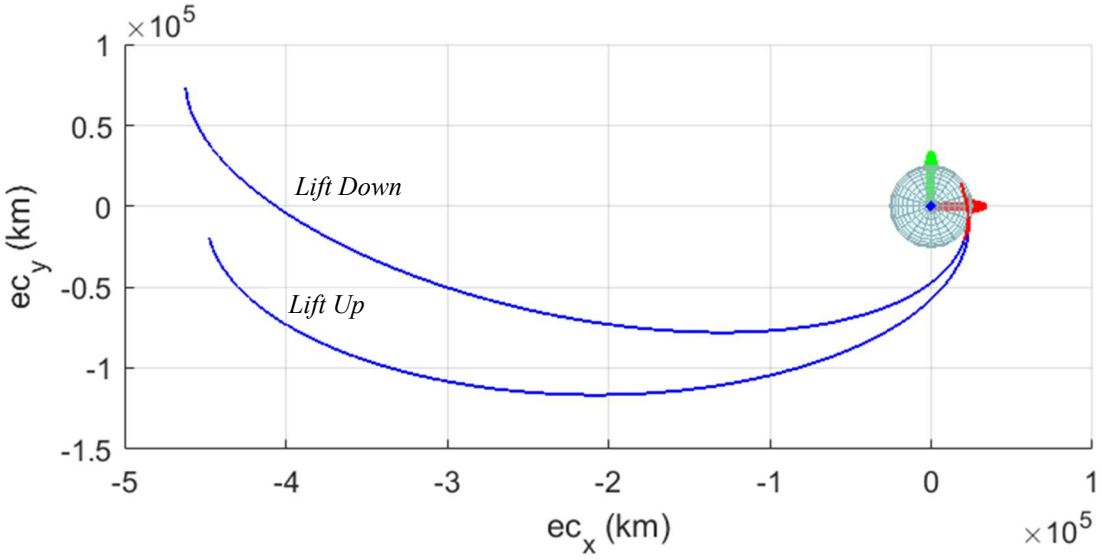


Figure 9.32 Uranus Aerocapture Test Trajectory

9.5. Basic Equations for Orbital Mechanics and Rocket Propulsion

$$V = \sqrt{\frac{2\mu}{r_a} - \frac{\mu}{a}} \tag{9.15}$$

9.12 is fundamental and can be used to calculate the required delta V for basic propulsive maneuvers like a Hohmann transfer.

Hyperbolic Excess Velocity

$$V_\infty = \sqrt{-\frac{\mu}{a}} \tag{9.16}$$

Orbital Period

$$T = \frac{2\pi}{\sqrt{\frac{\mu}{a^3}}} \tag{9.17}$$

eccentricity

$$e = \frac{r_a - r_p}{r_a + r_p} \tag{9.18}$$

9.5.1. Conversion of Keplerian Orbital Elements to Position and Velocity Vector

Angular Momentum

$$h = \sqrt{\mu a(1 - e^2)} \tag{9.19}$$

Perifocal Frame Transformation

$$\boldsymbol{R_{pf}} = \frac{h^2}{\mu(1 + e\cos(\theta))}\begin{bmatrix}\cos(\theta)\\ \sin(\theta)\\ 0\end{bmatrix} \tag{9.20}$$

$$\boldsymbol{V_{pf}} = \frac{\mu}{h}\begin{bmatrix}-\sin(\theta)\\ \mathrm{e} + \cos(\theta)\\ 0\end{bmatrix} \tag{9.21}$$

$$Q_1 = \begin{bmatrix}\cos\omega & \sin\omega & 0\\ -\sin\omega & \cos\omega & 0\\ 0 & 0 & 1\end{bmatrix} \tag{9.22}$$

Formulate matrix for ECI frame transformation

$$Q_2 = \begin{bmatrix}1 & 0 & 0\\ 0 & \cos i & \sin i\\ 0 & -\sin i & \cos i\end{bmatrix} \tag{9.23}$$

$$Q_3 = \begin{bmatrix}\cos\Omega & \sin\Omega & 0\\ -\sin\Omega & \cos\Omega & 0\\ 0 & 0 & 1\end{bmatrix} \tag{9.24}$$

$$Q_{ECI} = Q_1 Q_2 Q_3 \tag{9.25}$$

$$\boldsymbol{R_{ECI}} = {Q_{ECI}}^T \boldsymbol{R_{pf}} \tag{9.26}$$

$$\boldsymbol{V_{ECI}} = {Q_{ECI}}^T \boldsymbol{V_{pf}} \tag{9.27}$$

9.5.2. Conversion of Position and Velocity Vector to Keplerian Elements

Radial Velocity

$$V_{rad} = \frac{\boldsymbol{R_{ECI}} \cdot \boldsymbol{V_{ECI}}}{|\boldsymbol{R_{ECI}}|} \tag{9.28}$$

Momentum Vector

$$\boldsymbol{H} = \boldsymbol{R_{ECI}} \times \boldsymbol{V_{ECI}} \tag{9.29}$$

$$h = \sqrt{\cos^{-1}(\boldsymbol{H} \cdot \boldsymbol{H})} \tag{9.30}$$

$$i = \cos^{-1}\left(\frac{H\hat{\boldsymbol{k}}}{h}\right) \tag{9.31}$$

Nodal Vector

$$\boldsymbol{K} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \qquad \boldsymbol{N} = \boldsymbol{K} \times \boldsymbol{H} \qquad n = |\boldsymbol{N}| \tag{9.32}$$

Ascending node with quadrant ambiguity

$$\Omega = \begin{cases} \cos^{-1}\left(\dfrac{N\hat{\boldsymbol{i}}}{n}\right), & N\hat{\boldsymbol{j}} \geq \boldsymbol{0} \\ 360 - \cos^{-1}\left(\dfrac{N\hat{\boldsymbol{i}}}{n}\right), & N\hat{\boldsymbol{j}} < \boldsymbol{0} \end{cases} \tag{9.33}$$

Eccentricity Vector

$$\boldsymbol{E} = \frac{\boldsymbol{V_{ECI}} \times \boldsymbol{H}}{\mu - \dfrac{\boldsymbol{R_{ECI}}}{r}} \qquad e = |\boldsymbol{E}| \tag{9.34}$$

Argument of Periapsis with quadrant ambiguity

$$\omega = \begin{cases} \cos^{-1}\left(\dfrac{\boldsymbol{N} \cdot \boldsymbol{E}}{ne}\right), & E\hat{\boldsymbol{k}} \geq \boldsymbol{0} \\ 360 - \cos^{-1}\left(\dfrac{\boldsymbol{N} \cdot \boldsymbol{E}}{ne}\right), & E\hat{\boldsymbol{k}} < \boldsymbol{0} \end{cases} \tag{9.35}$$

True Anomaly with quadrant ambiguity

$$\theta = \begin{cases} \cos^{-1}\left(\dfrac{\boldsymbol{E}}{e} \cdot \dfrac{\boldsymbol{R_{ECI}}}{r}\right), & V_{rad} \geq \boldsymbol{0} \\ 360 - \cos^{-1}\left(\dfrac{\boldsymbol{E}}{e} \cdot \dfrac{\boldsymbol{R_{ECI}}}{r}\right), & V_{rad} < \boldsymbol{0} \end{cases} \tag{9.36}$$

9.5.3. Spherical Harmonics and Oblateness Effects

Oblateness effects are incorporated into the gravity model of the trajectory program. The model only considers the 2nd zonal harmonic, J2 as it has the greatest effect by several orders of magnitude. Expanded calculations and derivations to obtain 9.39 are shown in pp. 660-664 of [22]. Note that ϕ here refers to the angle between the position vector and polar axis, not to be confused with the geometric angle used in section 4.2.2.

Gravitational perturbation based on J2 term only.

$$\Phi(r,\phi) = \frac{J_2}{2}\frac{\mu}{r}\left(\frac{R}{r}\right)^2 (3\cos^2\phi - 1) \tag{9.37}$$

Perturbing Acceleration Vector

$$\mathbf{p} = -\nabla\Phi = -\frac{\partial\Phi}{\partial x}\hat{\mathbf{i}} - \frac{\partial\Phi}{\partial y}\hat{\mathbf{j}} - \frac{\partial\Phi}{\partial z}\hat{\mathbf{k}} \tag{9.38}$$

Final perturbation vector in the ECI frame

$$\mathbf{p} = \frac{3}{2}\frac{J_2\mu R^2}{r^4}\left[\frac{x}{r}\left(5\frac{z^2}{r^2} - 1\right)\hat{\mathbf{i}} + \frac{y}{r}\left(5\frac{z^2}{r^2} - 1\right)\hat{\mathbf{j}} + \frac{z}{r}\left(5\frac{z^2}{r^2} - 3\right)\hat{\mathbf{k}}\right] \tag{9.39}$$

9.5.4. Basic Elements of Rocket Propulsion

Tsiolkovsky rocket equation: mf is the initial mass and me is the final or dry mass

$$\Delta V = I_{sp}\, g_o \ln\left(\frac{m_f}{m_e}\right) \tag{9.40}$$

Calculate propellant mass usage Mp

$$M_p = m_f - m_f e^{-\frac{\Delta V}{I_{sp} g_o}} \tag{9.41}$$

Calculate burn time for a specified ΔV, 9.36 is used in conjunction with 9.12 to calculate the perigee raise maneuver.

$$t_b = m_f - m_f e^{-\frac{\Delta V}{I_{sp} g_o}} \tag{9.42}$$

9.6. Source Code

The trajectory analysis program that was developed to support this project has been rigorously validated and tested and works for all GRAM supported planets with an atmosphere. Rudimentary property validation and error checking has been implemented but it is still at a development level and is not intended for redistribution or re-use. Below are the core trajectory propagation and shared handle object class definition files. There are numerous development scripts and supporting functions that are not included to limit excessive page length.

```
classdef AeroDB < matlab.System
    % Aerodynamics Database Object. Contains a modified newtonian and free
    % molecular panel codes as well as a sine squared bridging function for
    % the transitional regime.
    %
    % Paneling algorithm supports basic
    % axisymmetric bodies with radiused or straight frustrum lengthwise
    % segments
    %
    % Written by: Bohdan Wesely, MSAE at SJSU, NASA ARC Entry Systems
    % and Technology Division, May 2025, MATLAB 2024b.

    % Public, tunable properties
    properties
        g = 1.45; % Specific heat ratio
        Mi = 30; % freestream mach
        Pinf = 10; % (Pa) freestream pressure
        R = 2.5; % Aeroshell Radius
        R2 = 3; % Biconic Radius
        RN = 1.25; % (m) span
        tc1 = 70; % deg
        tc2 = 5.2; % deg
        seg = 8; % nose segments
        rseg = 8; % radial axisymmetric divisions
        pan = 1; % straight panel segments
        trimBeta = 0; % trim sideslip angle
        trimAlpha = 0; % trim angle of attack

        % Rarefied gas parameters
        Vi = 9000; % freestream velocity (m/s)
        Tw = 1000; % (K) Wall temperature
        Ti = 300; % (K) Freestream temperature
        Rspec = 287.058; % J/kg K
        sigN = 1; % normal momentum accommodation coefficient (0 for specular 1 for↙
diffuse)
        sigT = 1; % tangential momentum accommodation coefficient (0 for specular 1 for↙
diffuse)
        kn = 1; % free stream knudsen number
        rarefiedGasEffects = 'off';
        a1 % Bridging Function Constant
        a2 % Bridging Function Constant
        Xc % Lengthwise X coodinates
        Yc % Lengthwise Y coodinates
        plotX = ["Xc", "alph", "alph", "kn"]; % X axis properties to plot
        plotY = ["Yc", "CA", "CN", "kn"]; % Y axis properties to plot
    end

    properties (SetObservable, AbortSet)
        knFm = 10; % knudsen number free molecular bound
        knCont = 0.001; % knudsen number continuum bound
```

```
    end

    properties
        CD % Drag Coefficient
        CL % Lift Coefficient
        CZ % Side Force Coefficient
        CA % Axial Force Coefficient
        CN % Normal Force Coefficient
        CS % Side Slip Coefficient
        alph % Angle of attack
        beta % Angle of sideslip
    end

    methods (Access = protected)

        function setupImpl(obj)
            % Convert Angles to radians
            radConvert(obj)

            % Set up transitional flow regime bridging function
            bridgeSetup(obj)
        end

        function stepImpl(obj)

            % create vectors
            Tc = [obj.tc1 obj.tc2];
            H = [obj.R obj.R2];

            % find angle of one segment
            tseg = (pi/2-obj.tc1)/obj.seg;

            % Freestream molecular Speed Ratio
            s = obj.Vi/sqrt(2*obj.Rspec*obj.Ti);

            % Isentropic Pressure Ratio
            Pratio = ((obj.g+1)^2*obj.Mi^2/(4*obj.g*obj.Mi^2-2*(obj.g-1)))^(obj.g/(obj.↙
g-1))*((1-obj.g+2*obj.g*obj.Mi^2)/(obj.g+1));

            % Subsonic Error Check
            if ~isreal(Pratio)
                error(['Imaginary Number detected for isentropic Pressure ratio, ' ...
                    'Potentially means vehicle is subsonic, increase velocity↙
termination cutoff']);
            end

            % Modified Newtonian Multipliers (CpMax)
            Pratio_chem = 515.9397;
            CpMax(1) = 2/(obj.g*obj.Mi^2)*(Pratio-1);
            CpMax(2) = 2/(obj.g*obj.Mi^2)*(Pratio_chem-1);
```

```
            % Disable any transitional effects if flow is fully free
            % molecular or continuum
            if strcmp(obj.rarefiedGasEffects,'on')
                if obj.kn > obj.knFm
                    aeroTyp = 'FM';
                elseif obj.kn < obj.knCont
                    aeroTyp = 'Cont';
                else
                    aeroTyp = 'both';
                end
            else
                aeroTyp = 'Cont';
            end

            % **Initialize body length parameterization**
            % x coordinate, y coordinate, and total distance along body surface
            xb = zeros(1,obj.seg+2*obj.pan); yb = xb; db = xb;

            % angle of each panel and Cp for each panel, CpN is for regular newtonian,
            thetai = zeros(1,obj.seg+2*obj.pan);

            % **Initialize Coefficients **
            CVFi_fm = zeros(3,obj.seg+2*obj.pan); CVFi_cont = CVFi_fm;

            for i2 = 1:obj.seg
                % because nose is circular, an angle index can define panels of equal
                % length
                thetai(i2) = pi/2-tseg*(2*i2-1)/2;

                % X and Y parameterization
                xb(i2+1) = obj.RN-obj.RN*cos(tseg*i2); yb(i2+1) = obj.RN*sin(tseg*i2);↙
db(i2+1) = obj.RN*tseg*i2;

                % Call paneling function to calculate coefficients
                [CVFi_fm(:,i2), CVFi_cont(:,i2)] = panelCalc(obj.alph,obj.beta,obj.↙
rseg,thetai(i2) ...
                    ,yb(i2),yb(i2+1),obj.RN*tseg,s,obj.Tw,obj.Ti,obj.sigT,obj.sigN,↙
aeroTyp);
            end

            % set conditions for straight segments
            j1 = 1;
            for i2 = obj.seg+1:2:2*obj.pan+obj.seg

                % set conditions the same at the beginning and end of each panel
                thetai(i2) = Tc(j1);

                % X and Y parameterization
                [xb(i2+1), yb(i2+1), dnew] = PointSlope(xb(i2), yb(i2), H(j1), thetai↙
```

```
(i2));

                % Call paneling function to calculate coefficients
                [CVFi_fm(:,i2), CVFi_cont(:,i2)] = panelCalc(obj.alph,obj.beta,obj.↙
rseg,thetai(i2),yb(i2),yb(i2+1),dnew,s,obj.Tw,obj.Ti,obj.sigT,obj.sigN,aeroTyp);

                % Distance between nodes
                db(i2+1) = dnew + db(i2);

                % avoids creating duplicate point at end of panel
                if i2 <  2*obj.pan+obj.seg-1
                    xb(i2+2) = xb(i2+1);  yb(i2+2) = yb(i2+1);
                    db(i2+2) = db(i2+1);
                end
                j1 = j1 + 1;
            end

            % Aero coefficients in velocity vector frame (first term is newtonian
            % multiplier, 2 for standard and CpMax for modified)
            switch aeroTyp
                case 'FM' % Free molecular
                    CVF = sum(CVFi_fm,2)/(pi*max(yb)^2);
                case 'Cont' % Continuum
                    CVF = CpMax(1)*sum(CVFi_cont,2)/(pi*max(yb)^2);
                case 'both' % Transitional Region
                    CVFfm = sum(CVFi_fm,2)/(pi*max(yb)^2);
                    CVFcont = CpMax(1)*sum(CVFi_cont,2)/(pi*max(yb)^2);

                    % Bridging function
                    Pb = sin(pi*(obj.a1+obj.a2*log10(obj.kn))).^2;
                    CVF = Pb*CVFfm+(1-Pb)*CVFcont;
            end

            % Lengthwise coordinates
            obj.Xc = xb; obj.Yc = yb;

            % Aero coefficients in body frame
            CBF = RZZ(obj.alph)'*RYY(obj.beta)'*CVF;

            % Extract body frame coefficients
            obj.CA = CBF(1); obj.CN = CBF(2); obj.CS = CBF(3);

            % Extravt Velocity frame coefficients
            obj.CD = CVF(1); obj.CL = -CVF(2); obj.CZ = CVF(3);


            function [CVFi_fm, CVFi_cont] = panelCalc(alpha,beta,rseg,theta,y1,y2,d,s,↙
Tw,Ti,sigT,sigN,aeroTyp)

                % Calculate area of one panel
```

```
            Atot = pi*d*(y2+y1);
            Arseg = Atot/rseg;

            % create series of normal angles for rseg number of panels
            inc = 2*pi/rseg;
            Rang = linspace(0.5*inc,2*pi-0.5*inc,rseg);

            % Local Cone angle
            Norm = pi/2-theta;

            % Initial normal vector to vehicle surfrace
            V = [cos(Norm)
                sin(Norm)
                0];

            % Create 3D matrix to revolve normal vector around axisymmetric body
            Rx = [ones(1,rseg); zeros(1,rseg); zeros(1,rseg);
                zeros(1,rseg); cos(Rang); sin(Rang);
                zeros(1,rseg); -sin(Rang);  cos(Rang); ];
            Rx = reshape(Rx,3,3,[]);

            % Initialize series of normal vectors
            Z = zeros(3,rseg); T = Z;
            Vinf = [-1 0 0]';

            for i = 1:rseg

                % Revolve around body
                Z(:,i) = Rx(:,:,i)*V;

                % Rotate normal vector with respect to velocity vector frame
                Z(:,i) = RYY(beta)*RZZ(alpha)*Z(:,i);

                % if vehicle surface is in the shadow area, no aero forces are
                % applied per newtonian mechanics
                if Z(1,i) < 0
                    Z(:,i) = [0;0;0];
                end

                % calc tangential vector
                if strcmp(aeroTyp,'FM') || strcmp(aeroTyp,'both')
                    T(:,i) = (Z(:,i)*dot(Vinf,Z(:,i))-Vinf)/sqrt(1-dot(Vinf,Z(:,i))↙
^2);
                end
            end

            % Calculate velocity frame coefficients
            switch aeroTyp
                case 'FM'
                    CVFi_fm = getCoeFm;
```

```
                    CVFi_cont = [0;0;0];
                case 'Cont'
                    CVFi_fm = [0;0;0];
                    CVFi_cont = getCoeCont;
                case 'both'
                    CVFi_fm = getCoeFm;
                    CVFi_cont = getCoeCont;
            end

            % Free Molecular
            function cOut = getCoeFm

                % Pressure Coefficient
                Cp = (2-sigN)*1/s^2*(s*Z(1,:)/sqrt(pi).*exp(-(s*Z(1,:)).^2)+(0.5+↙
(s*Z(1,:)).^2).*(1+erf(s*Z(1,:))))...
                    +sigN/(2*s^2)*sqrt(Tw/Ti)*(s*Z(1,:).*sqrt(pi).*(1+erf(s*Z↙
(1,:)))+exp(-(s*Z(1,:)).^2))-1/s^2;

                % Shear Coefficient
                Ct = sigT*cos(asin(Z(1,:)))/s.*(1/sqrt(pi)*exp(-(s*Z(1,:)).^2)+s*Z↙
(1,:).*(1+erf(s*Z(1,:))));

                cOut = sum((Cp.*Z+Ct.*T)*Arseg,2);
            end

            % Continuum (Newtonian)
            function cOut = getCoeCont
                cOut = sum(Z(1,:).^2.*Z*Arseg,2);
            end

            % Free molecular aero coefficients are calculated using a Maxwellian
            % distribution of specular or diffuse particle collisions

            % Kenneth A. Hart, Kyle R. Simonis, Bradley A. Steinfeldt, and Robert↙
D. Braun.
            % "Analytic Free-Molecular Aerodynamics for Rapid Propagation of↙
Resident Space Objects,
            % " Journal of Spacecraft and Rockets 2018 55:1, 27-36
            %%
            % <https://doi.org/10.2514/1.A33606>
        end

        % Z axis rotation matrix
        function Rz = RZZ(ang)
            % for a sphere cone EV, lift up is CCW rotation of alpha (neg sign
            % swap)
            Rz = [cos(ang) sin(ang) 0
                -sin(ang) cos(ang) 0
                0        0     1];
        end
```

```
        % Y axis rotation matrix
        function Ry = RYY(ang)
            Ry = [cos(ang) 0 -sin(ang)
                0     1     0
                sin(ang) 0  cos(ang)];
        end

        % Create connecting points for straight segments
        function [x2, y2, d] = PointSlope(x1, y1, H, theta)
            y2 = H;
            x2 = (y2-y1)/tan(theta)+x1;
            d = sqrt((x2-x1)^2+(y2-y1)^2);
        end
    end

end

methods
    % convert to radians
    function radConvert(obj)
        obj.beta = obj.trimBeta*pi/180;
        obj.alph = obj.trimAlpha*pi/180;
        obj.trimBeta = obj.trimBeta*pi/180;
        obj.trimAlpha = obj.trimAlpha*pi/180;
        obj.tc1 = obj.tc1*pi/180;
        obj.tc2 = obj.tc2*pi/180;
    end

    % setup bridging function
    function bridgeSetup(obj)
        A = [1 log10(obj.knFm);1 log10(obj.knCont)];
        B = [0.5;0];
        A12 = A\B;
        obj.a1 = A12(1); obj.a2 = A12(2);
    end

    % Generate Sample plots of Aerodatabase
    function plotAero(obj)

        % initialize, save current properties in temp variables
        alphTemp = obj.alph; KNtemp = obj.kn;
        setTemp = obj.rarefiedGasEffects;
        viTemp = obj.Vi; sigTtemp = obj.sigT; sigNtemp = obj.sigN;
        tWtemp = obj.Tw;

        % Preallocate
        m = 50; alphaSet = 10*pi/180;
        xRange = linspace(0,30,m);
        kNrange = logspace(log10(obj.knCont/10),log10(obj.knFm*10),m);
```

```
CAfm = zeros(1,m); CAcont = CAfm;
CNfm = CAfm; CNcont = CAfm; CAkn = CAfm; CNkn = CAfm;

for i1 = 1:m
    obj.(obj.plotX(2)) = xRange(i1)*pi/180;

    % Continuum Bound
    obj.rarefiedGasEffects = 'off';
    step(obj);
    CAcont(i1) = obj.CA;
    CNcont(i1) = obj.CN;

    % Free Molecular Bound
    obj.rarefiedGasEffects = 'on';
    step(obj);
    CAfm(i1) = obj.CA;
    CNfm(i1) = obj.CN;
end

obj.rarefiedGasEffects = 'on';
obj.(obj.plotX(2)) = alphaSet;

% Vary Knudsen Number
for i1 = 1:m
    obj.kn = kNrange(i1);
    step(obj);
    CAkn(i1) = obj.CA;
    CNkn(i1) = obj.CN;
end

hold on
% plot geometry
subplot(2,2,1)
plot(obj.Xc,obj.Yc,'LineWidth',2);
daspect([1 1 1])
grid on
xlabel('X (m)');
ylabel('Y (m)');
title('Geometry');
grid on

% Plot CA
subplot(2,2,2)
plot(xRange,CAcont,xRange,CAfm,'LineWidth',2)
title('Axial Force Coefficient')
xlabel('\alpha (deg)')
ylabel('C_A')
legend('Continuum Bound','Free Molecular Bound')
grid on
```

```
            % Plot CN
            subplot(2,2,3)
            plot(xRange,CNcont,xRange,CNfm,'LineWidth',2)
            title('Normal Force Coefficient')
            xlabel('\alpha (deg)')
            ylabel('C_N')
            legend('Continuum Bound','Free Molecular Bound')
            grid on

            % Plot Coefficients vs. Kn
            subplot(2,2,4)
            semilogx(kNrange,CAkn,kNrange,CNkn,'LineWidth',2)
            title(['Aero Coefficients vs. K_n at 10',char(176),'\alpha'])
            xlabel('K_n');
            ylabel('Coefficient');
            legend('C_A ','C_N');
            grid on

            % Restore coefficients
            obj.alph = alphTemp; obj.kn = KNtemp;
            obj.rarefiedGasEffects = setTemp;
            obj.Vi = viTemp; obj.sigT = sigTtemp; obj.sigN = sigNtemp;
            obj.Tw = tWtemp;
            step(obj);
        end
    end
end
```

```
classdef AeroPass < OrbitProp
    % Atmospheric Flight Trajectory Propagation Object (subclass of
    % OrbitProp)

    % Written by: Bohdan Wesely, MSAE at SJSU, NASA ARC Entry Systems
    % and Technology Division, May 2025, MATLAB 2024b.

    properties
        Tmax = 2000 % (s) Maximum time for atmospheric flight
        AeroSpd = 'slow' % fast: fixed L/D, slow: CL,CD,CA, etc. are reclaculated at↙
each time step
        AeroStep = 0.5; % Default time step for atmospheric flight
    end

    methods (Access = protected)
        function P = Perturb(obj,t,P,R)

            % Convert and extract topo coordinates from state object
            obj.Time.elTime = t;
            obj.State.Reci = R(1:3);
            obj.State.Veci = R(4:6);
            obj.State.ECItoLLA;

            % Extract Vars from shared objects
            A = obj.SC_DB.A; m = obj.State.ScM; W = obj.Body_DB.W;
            alt = obj.State.Alt; lat = obj.State.Lat; long = obj.State.Long;
            fpa = obj.State.FPA; Az = obj.State.Az; Qmat = obj.State.Qmat;

            % Set position/time in GRAM
            obj.GRAM.position.height = alt;
            obj.GRAM.position.latitude = lat;
            obj.GRAM.position.longitude = long;
            obj.GRAM.position.elapsedTime = t;
            obj.GRAM.body.setPosition(obj.GRAM.position);

            % update GRAM model
            obj.GRAM.body.update();

            % Extract Winds from GRAM
            nsw = obj.GRAM.atmos.nsWind; % north south wind
            eww = obj.GRAM.atmos.ewWind; % east west wind
            vw = obj.GRAM.atmos.verticalWind; % vertical wind

            % Velocity vector in the ENZ frame (north south winds are
            % negative as north to south and east to west expressed as a positive
            % values in GRAM, also convert to km/s
            Venz = [-eww/1000
                    -nsw/1000
                    vw/1000];
            Wvec = Qmat*Venz;
```

```
% Formulate relative velocity vector
Vrel = R(4:6) - cross(W',R(1:3)); % account for planet rotation
Vrel = Vrel - Wvec; % account for winds
vrel = norm(Vrel); % magnitude
Uv = Vrel/vrel; % unit vector
% Uvinr = R(4:6)/norm(R(4:6));

% Relative and Inertial Velocity vectors in Velocity frame
UVF = ROTY(fpa)*ROTZ(90-Az)'*Qmat'*Uv;
% UVFinr = ROTY(fpa)*ROTZ(90-Az)'*Qmat'*Uvinr;

% Calculate effective angle of attack and sideslip angle based
% on winds and relative velocity
dBeta = real(asin(UVF(2))); dAlpha = real(acos(UVF(1)/cos(dBeta)));

% Update Aero coefficients at each timestep
if strcmp(obj.AeroSpd,'slow')
    % Update aerodatabase with effective angles of attack and
    % sideslip
    obj.SC_DB.Aero_DB.beta = obj.SC_DB.Aero_DB.trimBeta + dBeta;
    obj.SC_DB.Aero_DB.alph = obj.SC_DB.Aero_DB.trimAlpha + dAlpha;

    % Update Aero data needed for coefficients
    obj.SC_DB.Aero_DB.g = obj.GRAM.atmos.specificHeatRatio;
    aCurr = obj.GRAM.atmos.speedOfSound;
    obj.SC_DB.Aero_DB.Mi = vrel*1000/aCurr;

    % Rarefied Gas Effects
    if strcmp(obj.SC_DB.Aero_DB.rarefiedGasEffects,'on')
        % Query additional properties from GRAM
        obj.SC_DB.Aero_DB.Vi = vrel*1000;
        obj.SC_DB.Aero_DB.Rspec = obj.GRAM.atmos.specificGasConstant;
        obj.SC_DB.Aero_DB.Ti = obj.GRAM.atmos.temperature;

        % Calculate Knudsen number
        obj.chemObj.GRAMatmos = obj.GRAM.atmos;
        lamda = obj.chemObj.getMeanFreePath;
        Kn = lamda/obj.SC_DB.D;
        obj.SC_DB.Aero_DB.kn = Kn;
    end

    % Calculate aero coefficients
    obj.SC_DB.Aero_DB.step;
end

% Aero Coefficients
CL = obj.SC_DB.Aero_DB.CL;
CZ = obj.SC_DB.Aero_DB.CZ;
CD = obj.SC_DB.Aero_DB.CD;
```

```
            % Extract density from GRAM
            rho = obj.GRAM.atmos.density;

            % Perturbation vector in the relative velocity frame with aero
            % coefficients
            PVF = [-CD
                   CZ
                   CL]*A/m*0.5*rho*(1000*vrel)^2/1000;

            % Convert from relative velocity to inertial velocity frame
            PVFinr = ROTZ(dBeta*180/pi)'*ROTY(dAlpha*180/pi)*PVF;

            % Convert to ECI frame
            PECI = Qmat*ROTZ(90-Az)*ROTY(fpa)'*PVFinr;

            % Net Perturbation vector
            P = P + PECI;
        end

    end

    methods
        % Setup function, initialize chemistry object
        function setupfun(obj)
            obj.PhysTyp = 'Aero';
            obj.tstep = obj.AeroStep;
            obj.chemObj.GRAMatmos = obj.GRAM.atmos;
            obj.chemObj.setupInds;
        end
    end
end
```

```
classdef Aerothermal < matlab.System
    % Primary trajectory post-processer and plotter, re-runs trajectory
    % position and time data through GRAM to calculate aerothermal and any
    % other time history results

    % Written by: Bohdan Wesely, MSAE at SJSU, NASA ARC Entry Systems
    % and Technology Division, May 2025, MATLAB 2024b.

    % Public, tunable properties
    properties
        TrajPoint % subclass to perform aerotherm calculations at one trajectory point
        plotYN = true % switch to turn on or off plotting
    end

    % Shared handle objects
    properties
        State % handle object for current spacecraft state
        Results % handle object for trajectory results/outputs
        Body_DB % handle object for planetary body parameters database
        SC_DB % handle object for spacecraft parameters database
        GRAM % handle object for the GRAM interface
        Time % handle object for tracking elapsed time and time dependent planet↙
orientation
        chemObj % handle object for atmospheric chemistry calculations
        plotData % handle object for managing plotting options
    end

    methods
        % Constructor: Pass State and Result Handle objects
        function obj = Aerothermal(varargin)
            % No inputs case, creates default reference objects internally
            if nargin == 0
                % Provide values for superclass constructor
                % and initialize other inputs
                obj.State = SCState;
                obj.Results = TrajResults;
                obj.Body_DB = BodyInputs;
                obj.SC_DB = SCInputs;
                obj.GRAM = gramMgr;
                obj.Time = timeMgr;
                obj.chemObj = chemMgr;
                obj.plotData = plotProps;

                % Individaul reference objects passed to constructor as inputs args
            elseif nargin == 8
                % When nargin ~= 0, assign to cell array,
                % which is passed to supclass constructor
                for i1 = 1:8
                    if isa(varargin{i1},'SCState'); obj.State = varargin{i1};
                    elseif isa(varargin{i1},'TrajResults'); obj.Results = varargin{i1};
```

```
                elseif isa(varargin{i1},'BodyInputs'); obj.Body_DB = varargin{i1};
                elseif isa(varargin{i1},'SCInputs'); obj.SC_DB = varargin{i1};
                elseif isa(varargin{i1},'gramMgr'); obj.GRAM = varargin{i1};
                elseif isa(varargin{i1},'timeMgr'); obj.Time = varargin{i1};
                elseif isa(varargin{i1},'chemMgr'); obj.chemObj = varargin{i1};
                elseif isa(varargin{i1},'plotProps'); obj.plotData = varargin{i1};
                else; error('Invalid shared object inputs');
                end
            end

        % Reference objects passed as a masterHand encapsulating object
        elseif nargin == 1 && isa(varargin{1},'masterHand')
            obj.State = varargin{1}.State;
            obj.Results = varargin{1}.Results;
            obj.Body_DB = varargin{1}.Body;
            obj.SC_DB = varargin{1}.S_C;
            obj.GRAM = varargin{1}.GRAM;
            obj.Time = varargin{1}.Time;
            obj.chemObj = varargin{1}.chemData;
            obj.plotData = varargin{1}.plotData;
        else
            error('Invalid Constructor Inputs')
        end
    end
end

methods (Access = protected)
    function setupImpl(obj)
        % Perform one-time calculations, such as computing constants
        setupfun(obj);
    end

    function stepImpl(obj)

        % Will only plot results for atmospheric flight
        if strcmp(obj.Results.Type,'Aero')

            % Save current vehicle state to reset to after aerothermal
            % calculations
            obj.State.saveState;

            % Loop through generated trajectory to post process
            % aerothermal and other results
            t = obj.Results.t; n = length(t);
            Js = zeros(n,1);
            out(n) = obj.TrajPoint.step(1);
            for i1 = 1:length(t)
                out(i1) = obj.TrajPoint.step(i1);
                qsI = out(i1).qs;
                if i1 > 1
```

```
                    Js(i1) = Js(i1-1) + qsI*(t(i1)-t(i1-1)); % total heat load
                end
            end

            % Reset State
            obj.State.reset;

            % Reset fallback state to beginning of simulation
            obj.State.revertState;

            % Populate results
            obj.Results.alt = [out.alt];
            obj.Results.qs = [out.qs];
            obj.Results.qsMax = max([out.qs]);
            obj.Results.Js = Js;
            obj.Results.jsMax = Js(end);
            obj.Results.fpa = [out.fpa];
            obj.Results.fpaI = obj.Results.fpa(1);
            obj.Results.rho = [out.rho];
            obj.Results.Kn = [out.kn];
            obj.Results.BC = obj.SC_DB.BC;

            % Calculate delta V lost with each pass (change in velocity
            % at periapsis)
            mu = obj.Body_DB.mu;
            [aPre,ePre] = ECItoKep(obj.Results.Rt(1,1:3)',obj.Results.Rt(1,4:6)',↙
mu);
            [aPost,ePost] = ECItoKep(obj.Results.Rt(end,1:3)',obj.Results.Rt(end,4:↙
6)',mu);
            rpPre = aPre*(1-ePre);
            rpPost = aPost*(1-ePost);
            Vpre = sqrt(2*mu/rpPre-mu/aPre);
            Vpost = sqrt(2*mu/rpPost-mu/aPost);
            obj.Results.dVaero = Vpre-Vpost;
            obj.Results.tPost = 2*pi/sqrt(mu/aPost^3);

            % Plot results
            if obj.plotYN
                obj.plotData.step
            end
        end
    end
end

methods

    function setupfun(obj)
        obj.TrajPoint = AerothermStep(obj);
    end
end
```

```
end
```

```
classdef AerothermStep < Aerothermal
    % Sub class of aerothermal which calls GRAM for one trajectory point
    % and generates aerothermal and other time history results

    % Written by: Bohdan Wesely, MSAE at SJSU, NASA ARC Entry Systems
    % and Technology Division, May 2025, MATLAB 2024b.

    methods
        % Constructor
        function obj = AerothermStep(varargin)
            if nargin == 1 && isa(varargin{1},'Aerothermal')
                obj.State = varargin{1}.State;
                obj.Results = varargin{1}.Results;
                obj.Body_DB = varargin{1}.Body_DB;
                obj.SC_DB = varargin{1}.SC_DB;
                obj.GRAM = varargin{1}.GRAM;
                obj.Time = varargin{1}.Time;
                obj.chemObj = varargin{1}.chemObj;
            else
                error('Invalid Constructor Inputs')
            end
        end
    end

    methods (Access = protected)

        function outStrct = stepImpl(obj,inc)
            % Convert and extract topo coordinates from state object
            obj.Time.elTime = obj.Results.t(inc);
            obj.State.Reci = obj.Results.Rt(inc,1:3)';
            obj.State.Veci = obj.Results.Rt(inc,4:6)';
            obj.State.ECItoLLA;

            % Extract Vars from shared objects
            W = obj.Body_DB.W; k = obj.Body_DB.k; Qmat = obj.State.Qmat;
            alt = obj.State.Alt; lat = obj.State.Lat; long = obj.State.Long;
            t = obj.Results.t(inc); Reci = obj.State.Reci;
            Veci = obj.State.Veci; Rn = obj.SC_DB.RN;

            % Set position/time in GRAM
            obj.GRAM.position.height = alt;
            obj.GRAM.position.latitude = lat;
            obj.GRAM.position.longitude = long;
            obj.GRAM.position.elapsedTime = t;
            obj.GRAM.body.setPosition(obj.GRAM.position);

            % update GRAM model
            obj.GRAM.body.update();

            % Calculate Winds
```

```
            nsw = obj.GRAM.atmos.nsWind; % north south wind
            eww = obj.GRAM.atmos.ewWind; % east west wind
            vw = obj.GRAM.atmos.verticalWind; % vertical wind

            % Velocity vector in the ENZ frame (north south winds are
            % negative as north to south and east to west expressed as a positive
            % values in GRAM, also convert to km/s
            Venz = [-eww/1000
                    -nsw/1000
                    vw/1000];
            Wvec = Qmat*Venz;

            % Formulate relative velocity vector
            Vrel = Veci - cross(W',Reci); % account for planet rotation
            Vrel = Vrel - Wvec; % account for winds
            vrel = norm(Vrel); % magnitude

            % Extract Knudsen Number
            obj.chemObj.GRAMatmos = obj.GRAM.atmos;
            lamda = obj.chemObj.getMeanFreePath;
            Kn = lamda/obj.SC_DB.D;

            % Extract density from GRAM
            rho = obj.GRAM.atmos.density;

            % calculate stagnation heat flux (sutton graves correlation)
            qs = k*sqrt(rho/Rn)*(vrel*1000)^3/1e4;

            % Struct can be populated with more time resolved properties (long, lat,
            % stag pressure, temperature, etc.
            outStrct.qs = qs;
            outStrct.alt = obj.State.Alt;
            outStrct.fpa = obj.State.FPA;
            outStrct.rho = rho;
            outStrct.kn = Kn;
        end
    end

    methods
        % Setup Chemistry solver
        function setupfun(obj)
            obj.chemObj.GRAMatmos = obj.GRAM.atmos;
            obj.chemObj.setupInds;
        end
    end
end
```

```
classdef BodyInputs < matlab.System
    % Class that handles planetary constants and associated properties

    % Written by: Bohdan Wesely, MSAE at SJSU, NASA ARC Entry Systems
    % and Technology Division, May 2025, MATLAB 2024b.

    % Public, tunable properties
    properties
        mu  %km^3/s^2 G*M so specific grav constant
        Re  % (km) primary body equatorial radius
        Rp  % (km) primary body polar radius
        W   % Planet angular velocity vector (rad/s)
        J2  % 2nd Zonal Harmonic
        k  % Aerothermal Constant

        AltThr = 150; % (km) altitude threshold where drag comes into effect
        termSpd = 1; % km/s if velocity falls below this value (in aeropass) simulation
is terminated
        planModel = 'ellipse' % planet shape model (sphere or ellipse) a
                              % spherical model assumes the equatorial
                              % radius Rp is the spherical radius

        RGB % Planet Display Color
    end

    % Observable property so other objects can update when planet is
    % changed
    properties (SetObservable, AbortSet, Dependent)
        planet
    end

    properties (Access = private)
        storePlanet
    end

    methods (Access = protected)

        % Output properties as struct if necessary
        function BodyDB = stepImpl(obj)
            BodyDB = struct;
            publicProperties = properties(obj);
            for fi = 1:numel(publicProperties)
                BodyDB.(publicProperties{fi}) = obj.(publicProperties{fi});
            end
        end

        % Save Object
        function s = saveObjectImpl(obj)
            s = saveObjectImpl@matlab.System(obj);
            s.storePlanet = obj.storePlanet;
```

```
        end

        % Load Object
        function loadObjectImpl(obj,s,isInUse)
            loadObjectImpl@matlab.System(obj,s,isInUse);
            obj.storePlanet = s.storePlanet;
        end
    end

    methods

        % Planet Shape Model
        function set.planModel(obj,val)
            if strcmp(val,'ellipse') || strcmp(val,'sphere')
                obj.planModel = val;
            else
                error('Planet shape model must either be "ellipse" or "sphere"↙
lowercase')
            end
        end

        % Pull planet string from privated non-dependent property
        function planetOut = get.planet(obj)
            planetOut = obj.storePlanet;
        end

        % Planet set method containing all planetary constants
        % Source: NASA Planetary Fact Sheet
        % https://nssdc.gsfc.nasa.gov/planetary/factsheet/
        function set.planet(obj,body)
            obj.storePlanet = body;
            switch body
                case 'Venus'
                    obj.mu = 324858.592; %km^3/s^2 G*M so specific grav constant
                    obj.Re = 6051.8; % (km) primary body radius
                    obj.Rp = 6051.8; % (km) primary body polar radius
                    obj.W = [ 0 0 -2.9924e-07]; % Planet angular velocity (rad/s)
                    obj.J2 = 4.458E-06; % 2nd Zonal Harmonic
                    obj.k = 0.00019; % Aerothermal Constant
                    obj.RGB = [0.9290 0.6940 0.1250];
                case 'Uranus'
                    obj.mu = 5.7940e6; %km^3/s^2 G*M so specific grav constant
                    obj.Re = 25559; % (km) primary body equatorial radius
                    obj.Rp = 24973; % (km) primary body polar radius
                    obj.W = [ 0 0 -1.0124e-04]; % Planet angular velocity (rad/s)
                    obj.J2 = 3343.43E-06; % 2nd Zonal Harmonic
                    obj.k = 8.645E-5; % Aerothermal Constant
                    obj.RGB = [192 236 240]/255;
                case 'Neptune'
                    obj.mu = 6.8351e6; %km^3/s^2 G*M so specific grav constant
```

```
                    obj.Re = 24764; % (km) primary body equatorial radius
                    obj.Rp = 24341; % (km) primary body polar radius
                    obj.W = [ 0 0 1.0834e-04]; % Planet angular velocity (rad/s)
                    obj.J2 = 3411E-06; % 2nd Zonal Harmonic
                    obj.k = 8.645E-5; % Aerothermal Constant *unverified for neptune,↙
set the same as Uranus
                    obj.RGB = [90 145 226]/255;
                case 'Jupiter'
                    obj.mu = 126.687e6; %km^3/s^2 G*M so specific grav constant
                    obj.Re = 71492; % (km) primary body equatorial radius
                    obj.Rp = 66854; % (km) primary body polar radius
                    obj.W = [ 0 0 1.7584e-04]; % Planet angular velocity (rad/s)
                    obj.J2 = 14736E-06; % 2nd Zonal Harmonic
                    obj.k = 8.645E-5; % Aerothermal Constant *unverified for neptune,↙
set the same as Uranus
                    obj.RGB = [220 174 66]/255;
                case 'Earth'
                    obj.mu = 0.39860e6; %km^3/s^2 G*M so specific grav constant
                    obj.Re = 6378.137; % (km) primary body equatorial radius
                    obj.Rp = 6356.752; % (km) primary body polar radius
                    obj.W = [ 0 0 7.2921e-05]; % Planet angular velocity (rad/s)
                    obj.J2 =  1082.63E-06; % 2nd Zonal Harmonic
                    obj.k = 1.7415e-4; % Aerothermal Constant
                    obj.RGB = [58 218 250]/255;
                case 'Mars'
                    obj.mu = 0.042828e6; %km^3/s^2 G*M so specific grav constant
                    obj.Re = 3396.2; % (km) primary body equatorial radius
                    obj.Rp = 3376.2; % (km) primary body polar radius
                    obj.W = [ 0 0 7.0882e-05]; % Planet angular velocity (rad/s)
                    obj.J2 =  1960.45E-06; % 2nd Zonal Harmonic
                    obj.k = 1.9027e-4; % Aerothermal Constant
                    obj.RGB = [240 118 47]/255;
                case 'Titan'
                    obj.mu = 0.0089781384e6; %km^3/s^2 G*M so specific grav constant
                    obj.Re = 2.5747e+03; % (km) primary body equatorial radius
                    obj.Rp = 2.5747e+03; % (km) primary body polar radius
                    obj.W = [0 0 4.5607e-06]; % Planet angular velocity (rad/s)
                    obj.J2 =  0.315E-06; % 2nd Zonal Harmonic
                    obj.k = 1.9e-4; % Aerothermal Constant
                    obj.RGB = [250 199 58]/255;
            end
        end
    end
end
```

```
classdef Burn < OrbitProp
    % Propulsive Maneuvers Trajectory Propagation Object (subclass of
    % OrbitProp)

    % Written by: Bohdan Wesely, MSAE at SJSU, NASA ARC Entry Systems
    % and Technology Division, May 2025, MATLAB 2024b.

    methods
        function setupfun(obj)
            obj.PhysTyp = 'Burn';
        end
    end

    methods (Access = protected)

        % Main Integration replaces superclass to allow for finite time burn
        function mainInt(obj)

            % Extract Vars from shared objects
            Reci = obj.State.Reci; Veci = obj.State.Veci;
            t_curr = obj.State.elTime; Tb = obj.State.Tb;
            m_o = obj.State.ScM;

            % Warn user if burn time is set to 0 and skip segment
            if Tb == 0
                warning('Burn time (Tb) must be greater than zero, skipping burn↙
segment');
                return;
            end

            % establish timeframe accounting for possibility of burn time being
            % less than 1 time step
            tsteb_b = Tb*obj.tstep_f; % rounding creates potential inaccuracy

            % Setup main trajectory segment timespan
            if strcmp(obj.calcSpd, 'continuous')
                tspan = t_curr:tsteb_b:t_curr+Tb;
            elseif strcmp(obj.calcSpd, 'jump')
                tspan = [t_curr t_curr + Tb];
            else
                error('property "calcSpd" set incorrectly')
            end

            % Main trajectory integrator call
            [t,Rt] = obj.ODEfun(@(t,R) TwoBody(obj,t,R),tspan,[Reci; Veci; m_o],obj.↙
opts1);

            % Populate Results if necessary
            if strcmp(obj.resultPop,'on')
                obj.Results.t = t; obj.Results.Rt = Rt; obj.Results.te = 0; obj.↙
```

```
Results.ye = 0; obj.Results.ie = 0;
                obj.Results.Tb = Tb; obj.Results.dVec = obj.State.dVec;
                % calculate delta V of burn
                ISP = obj.SC_DB.ISP; go = obj.SC_DB.go; me = Rt(end,7); mf = Rt(1,7);
                obj.Results.dV = ISP*go*log(mf/me);
            end

            % Populate shared handle objects with new trajectory data
            obj.State.Reci = Rt(end,1:3)';
            obj.State.Veci = Rt(end,4:6)';
            obj.State.ScM = Rt(end,7);
            obj.State.Tb = 0;
            obj.Time.elTime = t(end);

            % Update State
            obj.State.step;

        end

        function P = Perturb(obj,~,P,R)

            % Extract Vars from state object
            dVec = obj.State.dVec;
            v = norm(R(4:6));

            % Set thrust value
            if strcmp(obj.Thruster, 'Low')
                T = obj.SC_DB.LowThr;
            elseif strcmp(obj.Thruster, 'High')
                T = obj.SC_DB.Thr;
            else
                error('property "Thruster" set incorrectly')
            end

            % Purturbation Vector
            Tvec = T/(1000*R(7)*v);
            P = P + [Tvec*R(4)*dVec(1)
                     Tvec*R(5)*dVec(2)
                     Tvec*R(6)*dVec(3)];
        end
    end
end
```

```
classdef chemMgr < matlab.System
    % Manages all atmospheric chemistry calcuations,

    % Written by: Bohdan Wesely, MSAE at SJSU, NASA ARC Entry Systems
    % and Technology Division, May 2025, MATLAB 2024b.

    % Public, tunable properties
    properties
        eqChem = 'off'
        GRAMatmos % GRAM atmospheric state object
        inputSpecies % Property database for species supported by GRAM
        speciesInd logical % Filters GRAM atmos state for consituient gases
        speciesProps % Names of species present in the current atmosphere
        len % Number of species present
    end

    methods
        % Function to calculate the mean free path
        function lamda = getMeanFreePath(obj)
            spNames = obj.speciesProps;
            sigTot = 0;

            for i1 = 1:obj.len

                % Get Species data from GRAM
                species = obj.GRAMatmos.(spNames{i1});

                % Compute a weighted average kinetic cross section based on
                % the species mole fractions
                dK = obj.inputSpecies.(spNames{i1}).kinDia; % kinetic diameter
                sigTot = (dK*1e-12)^2*species.moleFraction + sigTot;

            end

            % Number density from GRAM
            n = obj.GRAMatmos.totalNumberDensity;

            % Final mean free path
            lamda = 1/(sqrt(2)*pi*sigTot*n);
        end
    end

    methods

        % Import species properties
        function obj = chemMgr
            obj.inputSpecies = inputSpecies;
        end

        % Initialize the chemistry calculations by predetermining which
```

```
        % species are present in the current planetary atmosphere, ignores
        % species that are not present
        function setupInds(obj)
            props = properties(obj.GRAMatmos);

            % Filter by GRAM consituent gas objects
            for i1 = 1:length(props)
                if isa(obj.GRAMatmos.(props{i1}),'clib.GRAMmi.GRAM.ConstituentGas') &&↙
obj.GRAMatmos.(props{i1}).isPresent
                    obj.speciesInd(i1) = true;
                else
                    obj.speciesInd(i1) = false;
                end
            end

            % Names and number of species
            obj.speciesProps = props(obj.speciesInd);
            obj.len = length(obj.speciesProps);
        end
    end
end
```

```
% Simple example script of a lift up and lift down simulation of a Venus
% aerocapture to 500000 km

clear
close all

% Load file and set all handle objects to base workspace
load('VenusAerocapt.mat')
saveFile.getHands

% Set apoapsis target for optimizer
apoapsis_targ = 500000;
ra_targ = apoapsis_targ + State.Rad;

% LIFT DOWN CASE

% Optimizer input format
altOpt = optoIn('order','AO','targ',ra_targ,'objective','ra','adjust','FPA');

% Optimize trajectory
missionPlan.lookForward(altOpt);

% Atmospheric Entry Pass
aeroProp.step;

% Post process trajectory, aerothermal calcs, plots
aeroTherm.step;

% 3D trajectory plot and store results
visPlot.step

% Set coast orbit to stop at apoapsis
orbProp.EventTyp = 'Ae';

% Propagate coast orbit and plot result
orbProp.step;
visPlot.step

% Reset state
State.reset

% LIFT UP CASE

% Set angle of attack
S_C.alpha = 10;

% Optimize Trajectory
missionPlan.lookForward(altOpt);

% Propagate Atmospheric entry
```

```
aeroProp.step;
aeroTherm.step;
visPlot.step

orbProp.EventTyp = 'Ae';

% Propagate coast orbit
orbProp.step;
visPlot.step
State.reset

% Generate output summary table
visPlot.AeroTab;
```

```
classdef gramMgr < matlab.System
    % Handles the GRAM interface and all initializations

    % Written by: Bohdan Wesely, MSAE at SJSU, NASA ARC Entry Systems
    % and Technology Division, May 2025, MATLAB 2024b.

    % Public, tunable properties
    properties
        % Below parameters are GRAM interface parameters
        inputParameters % GRAM inputs
        reader % Namelist reader
        body % Planet specific object
        ttime % Start time object
        position % position object
        atmos % atmosphere object, contains all relevant atomospheric data normally in↙
output file
        Time % Time manager object
    end

    % Pre-computed constants or internal states
    properties (Dependent)
        planet
    end

    properties (Access = private)
        storePlanet
    end

    methods (Access = protected)
        function s = saveObjectImpl(obj)
            s.Time = obj.Time;
        end
    end

    methods
        function planetOut = get.planet(obj)
            planetOut = obj.storePlanet;
        end

        % Sets up GRAM based on planet entry
        function set.planet(obj,body)
            if isempty(obj.Time); error('Time handle not set in GRAM manager class');↙
end

            obj.storePlanet = body;
            switch body
                case 'Venus'
                    obj.inputParameters = clib.GRAMmi.GRAM.VenusInputParameters();
                    obj.reader = clib.GRAMmi.GRAM.VenusNamelistReader();
                    obj.body = clib.GRAMmi.GRAM.VenusAtmosphere();
```

```
                case 'Uranus'
                    obj.inputParameters = clib.GRAMmi.GRAM.UranusInputParameters();
                    obj.reader = clib.GRAMmi.GRAM.UranusNamelistReader();
                    obj.body = clib.GRAMmi.GRAM.UranusAtmosphere();
                case 'Neptune'
                    obj.inputParameters = clib.GRAMmi.GRAM.NeptuneInputParameters();
                    obj.reader = clib.GRAMmi.GRAM.NeptuneNamelistReader();
                    obj.body = clib.GRAMmi.GRAM.NeptuneAtmosphere();
                case 'Jupiter'
                    obj.inputParameters = clib.GRAMmi.GRAM.JupiterInputParameters();
                    obj.reader = clib.GRAMmi.GRAM.JupiterNamelistReader();
                    obj.body = clib.GRAMmi.GRAM.JupiterAtmosphere();
                case 'Earth'
                    obj.inputParameters = clib.GRAMmi.GRAM.EarthInputParameters();
                    obj.reader = clib.GRAMmi.GRAM.EarthNamelistReader();
                    obj.body = clib.GRAMmi.GRAM.EarthAtmosphere();
                    obj.inputParameters.useNCEP = true;
                    % inputParameters.NCEPPath = '../NCEPdata/FixedBin';
                    obj.inputParameters.dataPath = 'C:↙
\Users\bohda\OneDrive\Desktop\SJSU\AE295\Simulation Draft V05\GRAM Matlab\GRAMmi';
                case 'Mars'
                    obj.inputParameters = clib.GRAMmi.GRAM.MarsInputParameters();
                    obj.reader = clib.GRAMmi.GRAM.MarsNamelistReader();
                    obj.body = clib.GRAMmi.GRAM.MarsAtmosphere();
                    obj.inputParameters.dataPath = "C:↙
\Users\bohda\OneDrive\Desktop\SJSU\AE295\Simulation Draft V05\GRAM Matlab\GRAMmi\data";
                case 'Titan'
                    obj.inputParameters = clib.GRAMmi.GRAM.TitanInputParameters();
                    obj.reader = clib.GRAMmi.GRAM.TitanNamelistReader();
                    obj.body = clib.GRAMmi.GRAM.TitanAtmosphere();
            end

            obj.reader.tryGetSpicePath(obj.inputParameters);
            % Create a venus atmosphere object
            obj.body.setInputParameters(obj.inputParameters);

            % Set the start time of the trajectory
            obj.ttime = clib.GRAMmi.GRAM.GramTime();
            obj.ttime.setStartTime(obj.Time.JD0, clib.GRAMmi.GRAM.GRAM_TIME_SCALE.UTC,↙
clib.GRAMmi.GRAM.GRAM_TIME_FRAME.ERT);
            obj.body.setStartTime(obj.ttime);

            % create position and atmosphere output objects
            obj.position = clib.GRAMmi.GRAM.Position();
            obj.atmos = obj.body.getAtmosphereState();

            % Display version and confirmation message
            fprintf('GRAM Initialized Sucessfully\n')
            disp(obj.body.getVersionString());
        end
```

```
    end
end
```

```
classdef InitState < matlab.System
    % Initial state class for handling default values when a new simulation
    % configuration is created

    % Written by: Bohdan Wesely, MSAE at SJSU, NASA ARC Entry Systems
    % and Technology Division, May 2025, MATLAB 2024b.

    % Public, tunable properties
    properties
        planet = 'Venus';

        % TOPOCENTRIC COORDINATES
        FPA = -5; % Flight Path Angle
        V = 11; % Velocity Magnitude
        Alt = 150; % Altitude
        Lat = 5; % Latitude
        Long = 47; % Longitude
        Az =  -90; % Azimuth

        % ORBITAL ELEMENTS
        e = 1.3074; % eccentricity
        minalt = 97.6157; % altitude at periapsis Change between 120 and 150 km
        inc = 0; % inclination
        Arg = 0; % arguement of periapsis
        Asc = 0; % ascension of ascending node
        theta = 137; % true anamoly

        % POSITION AND VELOCITY VECTORS
        Reci  % Vehicle Position Vector
        Veci  % Vehicle Velocity Vector

        % START TIME
        startTime = datetime(2041,05,20,9,3,8);
        JD = 0; % Julian Date

        % COORDINATE SELECTION
        % Topo for topocentric, Kepl for keplerial/orbital elements, ECIv for positon↙
and velocity vector
        Opt = 'Topo'

    end

    methods (Access = protected)

        function stepImpl(obj,Body,State,Time,GRAM)

            % Set planet
            Body.planet = obj.planet;
            Time.planet = obj.planet;
```

```
            % Set initial Julian date
            if isempty(obj.JD) || obj.JD == 0
                Time.JD0 = juliandate(obj.startTime);
                Time.startTime = obj.startTime;
            else
                Time.JD0 = obj.JD;
            end

            % Update GRAM after planet and Julian date have been set
            GRAM.planet = obj.planet;

            % Intialize state
            State.step;

            % Populate the rest of state based on coordinate selection
            switch obj.Opt
                case 'Topo'
                    State.FPA = obj.FPA;
                    State.V = obj.V;
                    State.Alt = obj.Alt;
                    State.Lat = obj.Lat;
                    State.Long = obj.Long;
                    State.Az = obj.Az;
                case 'Kepl'
                    rp = obj.minalt+Body.Re; % periapsis
                    State.a = rp/(1-obj.e);
                    State.e = obj.e;
                    State.inc = obj.inc;
                    State.Arg = obj.Arg;
                    State.Asc = obj.Asc;
                    State.theta = obj.theta;
                case 'ECIv'
                    State.Reci = obj.Reci;
                    State.Veci = obj.Veci;
                otherwise
                    error('Invalid Input Coordinate Option')
            end
        end
    end
end
```

```
classdef inputSpecies < matlab.System
    % Database for molecular weights and kinetic diameters of all species
    % listed within GRAM.
    %
    % Reference: https://cccbdb.nist.gov/introx.asp
    %
    % Future Work: Merge properties with those provided in GRAM or
    % in McBride CEA coefficients Database

    % Written by: Bohdan Wesely, MSAE at SJSU, NASA ARC Entry Systems
    % and Technology Division, May 2025, MATLAB 2024b.

    % Public, tunable properties
    properties
        species
    end

    properties (Access=private)
        % Reduction in kinetic diameter for free atomic species vs. their
        % diatomic molecules, initially used the spacing between atoms as
        % an offset however using the same diameter showed better agreement
        % with literature on knudsen number calculations

        % hOffset = 74.14
        % oOffset = 120.75
        % nOffset = 109.77
        hOffset = 0
        oOffset = 0
        nOffset = 0
    end

    methods
        function f1 = argon(~)
            f1.molWeight = 39.948;
            f1.kinDia = 340; % picometer (pm)
        end

        function f1 = carbonDioxide(~)
            f1.molWeight = 44.0095;
            f1.kinDia = 330; % picometer (pm)
        end

        function f1 = carbonMonoxide(~)
            f1.molWeight = 28.0101;
            f1.kinDia = 376; % picometer (pm)
        end

        function f1 = dihydrogen(~)
            f1.molWeight = 2.01588;
            f1.kinDia = 289; % picometer (pm)
```

```
end

function f1 = dinitrogen(~)
    f1.molWeight = 28.0134;
    f1.kinDia = 364; % picometer (pm)
end

function f1 = dioxygen(~)
    f1.molWeight = 31.9988;
    f1.kinDia = 346; % picometer (pm)
end

function f1 = helium(~)
    f1.molWeight = 4.002602;
    f1.kinDia = 260; % picometer (pm)
end

function f1 = hydrogen(obj)
    f1.molWeight = 1.00794;
    f1.kinDia = 289-obj.hOffset; % picometer (pm)
    % ref: https://cccbdb.nist.gov/exp2x.asp
    % dN2 - distance between atoms
end

function f1 = methane(~)
    f1.molWeight = 16.0425;
    f1.kinDia = 380; % picometer (pm)
end

function f1 = nitrogen(obj)
    f1.molWeight = 14.0067;
    f1.kinDia = 364-obj.nOffset; % picometer (pm)
end

function f1 = oxygen(obj)
    f1.molWeight = 15.9994;
    f1.kinDia = 346-obj.oOffset; % picometer (pm)
end

function f1 = ozone(~)
    f1.molWeight = 47.9982;
    f1.kinDia = 344; % picometer (pm)
end

function f1 = nitrousOxide(~)
    f1.molWeight = 44.0128;
    f1.kinDia = 330; % picometer (pm)
end

function f1 = water(~)
```

```
            f1.molWeight = 18.0153;
            f1.kinDia = 265; % picometer (pm)
        end
    end
end
```

```
classdef masterHand < matlab.System
    % All encompassing system object that sets up and contains all handle
    % objects necessary to run the simulation.

    % Written by: Bohdan Wesely, MSAE at SJSU, NASA ARC Entry Systems
    % and Technology Division, May 2025, MATLAB 2024b.

    % Public, tunable properties
    properties
        S_C % spacecraft inputs class
        Body % body inputs
        State % current spacecraft state
        Results % Results object
        GRAM % GRAM interface object
        Time % handle object for tracking elapsed time and time dependent planet↙
orientation
        Inputs % Sets up default values when setting up a new configuration
        orbProp % Coast trajectory
        aeroProp % Atmospheric flight trajectory
        burnProp % Burn/Maneuver Trajectory
        aeroTherm % Aerothermal postprocessor
        visPlot % 3D Trajectory plotter and results storage
        missionPlan % Trajectory optomization and multi-pass mission planner
        plotData % Handle object containing all 2D trajectory plot options
        chemData % handle object for atmospheric chemistry calculations
    end

    % Pre-computed constants or internal states
    properties (Access = private)
        lis event.proplistener % listeners for S_C property changes
        configWindow % object for configuration editor window
    end

    methods
        function obj = masterHand
            % Create low level shared handles
            obj.S_C = SCInputs;
            obj.Body = BodyInputs;
            obj.State = SCState;
            obj.Results = TrajResults;
            obj.Time = timeMgr;
            obj.Inputs = InitState;
            obj.GRAM = gramMgr;
            obj.chemData = chemMgr;
            obj.plotData = plotProps;

            % Initialize low level shared handles
            stateSetup(obj)
            obj.GRAM.Time = obj.Time;
            obj.S_C.step;
```

```
        % Create orbit propagation shared handles
        obj.orbProp = OrbitProp(obj);
        obj.aeroProp = AeroPass(obj);
        obj.burnProp = Burn(obj);
        obj.aeroTherm = Aerothermal(obj);
        obj.visPlot = TrajPlot(obj);

        % Create mission planner
        obj.missionPlan = MissionPlan(obj);

        % populate shared objects with input parameters
        processInputs(obj);

        % create listeners to automatically update
        listenScInputs(obj);
    end

    % Setup state object
    function stateSetup(obj)
        obj.State.ScM = obj.S_C.m_o; % Initialize S/C mass
        obj.State.setBody(obj.Body);
        obj.State.setTime(obj.Time);
    end

    % Automatically flattens class structure and populates workspace
    % with shared objects
    function getHands(obj)
        publicProperties = properties(obj);
        for fi = 1:numel(publicProperties)
            assignin('base',publicProperties{fi},obj.(publicProperties{fi}))
        end
    end

    % Setup Default Values
    function processInputs(obj)
        % Initialize body, state and time shared handle objects
        obj.Inputs.step(obj.Body,obj.State,obj.Time,obj.GRAM);

        % Uodate and save initial state
        obj.State.step;
        obj.State.saveState;
        obj.Time.step;
        obj.plotData.Results = obj.Results;
    end

    % Opens a configuration editor window and populates with current
    % setup
    function configEditor(obj)
        obj.configWindow = configurationEditor;
```

```
            obj.configWindow.getData(obj);
        end

        % Save a default values class
        function saveInputs(obj)
            saveInputs = obj.Inputs;
            save('SimInputs','saveInputs');
        end

        % Load a default values class
        function loadInputs(obj)
            inputsIn = load('SimInputs','saveInputs');
            obj.Inputs = inputsIn.saveInputs;
            processInputs(obj);
        end

        % Set the integration tolerance for all three trajectory
        % propagation objects
        function setIntTol(obj,relTol,absTol)
            obj.orbProp.RelTol = relTol; obj.orbProp.AbsTol = absTol;
            obj.aeroProp.RelTol = relTol; obj.aeroProp.AbsTol = absTol;
            obj.burnProp.RelTol = relTol; obj.burnProp.AbsTol = absTol;
        end

        % Change the ODE solver function for all three trajectory
        % propagation objects
        function setODEfun(obj,func)
            obj.orbProp.ODEfun = func;
            obj.aeroProp.ODEfun = func;
            obj.burnProp.ODEfun = func;
        end

        function listenScInputs(obj)

            % Create listeners for all Spacecraft object observable
            % properties
            propsSC = properties(obj.S_C);
            for i1 = 1:length(propsSC)
                obj.lis(i1) = addlistener(obj.S_C,propsSC{i1},'PostSet',@(src,evnt)obj.↙
scEvents(src,evnt,obj));
            end

            % Upate the aerodynamics bridging function if the knudsen
            % number bounds are changed
            obj.lis(i1+1) = addlistener(obj.S_C.Aero_DB,'knFm','PostSet',@(src,evnt)↙
obj.scEvents(src,evnt,obj));
            obj.lis(i1+2) = addlistener(obj.S_C.Aero_DB,'knCont','PostSet',@(src,evnt)↙
obj.scEvents(src,evnt,obj));

            % Update time and GRAM objects if planet is changed
```

```
            obj.lis(i1+3) = addlistener(obj.Body,'planet','PostSet',@(src,evnt)obj.↙
scEvents(src,evnt,obj));
        end
    end

    methods (Static)
      function scEvents(src,evnt,master)
         % terminology
         rocket = ["ISP","Thr","LowThr"];
         geometry = ["D","halfAng","RN","alpha","beta"];
         KN = ["knFm","knCont"];
         mass = "m_o"; planSet = "planet";

         % Object triggering event
         inOBJ = evnt.AffectedObject;

         % update s/c rocket engine parameters if a property is changed
         if contains(src.Name,rocket)
             inOBJ.updateEnginePerf;

         % update s/c geometry and aero parameters if a property is changed
         elseif contains(src.Name,geometry)
             inOBJ.updateGeometry;
             inOBJ.Aero_DB.radConvert;
             inOBJ.Aero_DB.step;
             inOBJ.updateBC;

         % update ballistic coefficient and mass in state object
         elseif contains(src.Name,mass)
             inOBJ.updateBC;
             master.State.ScM = inOBJ.m_o;

         % update bridging function if kn limits are changed
         elseif contains(src.Name,KN)
             inOBJ.bridgeSetup;

         % update planet in GRAM and time objects if body is updated
         elseif strcmp(src.Name,planSet)
             master.Time.planet = inOBJ.planet;
             master.GRAM.planet = inOBJ.planet;
         end
      end
   end

    methods (Access = protected)

        % Save a master handle object which effectively saves the entire
        % configuration and current state of the simulation
        function s = saveObjectImpl(obj)
            s = saveObjectImpl@matlab.System(obj);
```

```
        end

        % Load a master handle object
        function loadObjectImpl(obj,s,isInUse)
            loadObjectImpl@matlab.System(obj,s,isInUse);

            % Re-initialize GRAM since GRAM C++ objects are not saved
            obj.GRAM = gramMgr;
            obj.GRAM.Time = obj.Time;
            obj.GRAM.planet = obj.Body.planet;

            obj.orbProp.GRAM = obj.GRAM;
            obj.aeroProp.GRAM = obj.GRAM;
            obj.burnProp.GRAM = obj.GRAM;
            obj.aeroTherm.GRAM = obj.GRAM;

            % Re-populate listeners
            listenScInputs(obj);
        end
    end
end
```

```
classdef MissionPlan < matlab.System
    % Contains all trajectory optomization functions and some specific
    % manuever calculations to support a multi-pass aerocapture orbital
    % insertion simulation.

    % Written by: Bohdan Wesely, MSAE at SJSU, NASA ARC Entry Systems
    % and Technology Division, May 2025, MATLAB 2024b.

    % Mission parameters geared towards a multi-pass
    % orbital insertion
    properties
        raPost = 500000; % Apoapsis of initial post capture orbit (km)
        TCMdelay = 2500; % (s) initial cruise prior to TCM
        % Set calc speed to jump for faster calculation of final states for
        % trajectory optomization (accuracy may suffer)
        opts1 = struct↙
('calcSpd','continuous','EventTyp','Aero','output','off','resultPop','off');
        opts2 = struct↙
('calcSpd','continuous','EventTyp','Ae','output','off','resultPop','off');
        psiInt = [-180 180]; % initial attidude guess for burn attitude optomizer
        raTarg % vector of apogee targets for each aeropass
        tolX = 1e-8; % Convergence tolerance for traj optomization
        fzOpts % fzero options struct
    end

    % Optomization Inputs
    properties
        check % inputs to look forward and assess current state
        intTCM % inputs for interplanetary perigee trim TCM
        peOpt % inputs for perigee trim between aero passes
    end

    % Shared handle objects
    properties
        State % handle object for current spacecraft state
        SC_DB % handle object for spacecraft parameters database
        Body_DB % handle object for planetary body parameters database
        orbProp % handle object for coast trajectory
        aeroProp % handle object for atmospheric flight trajectory
        burnProp % handle object for simulating maneuvers
    end

    % Pre-computed constants or internal states
    properties (Access = private)
        currPass = 1 % Current pass number
    end

    methods
        % Constructor: Pass State and Result Handle objects
        function obj = MissionPlan(varargin)
```

```
        if nargin == 0
            % Provide values for superclass constructor
            % and initialize other inputs
            obj.State = SCState;
            obj.Body_DB = BodyInputs;
            obj.SC_DB = SCInputs;
            obj.orbProp = OrbitProp;
            obj.aeroProp = AeroPass;
            obj.burnProp = Burn;

        % Individaul reference objects passed to constructor as inputs args
        elseif nargin == 6
            % When nargin ~= 0, assign to cell array,
            % which is passed to supclass constructor
            for i1 = 1:6
                if isa(varargin{i1},'SCState'); obj.State = varargin{i1};
                elseif isa(varargin{i1},'BodyInputs'); obj.Body_DB = varargin{i1};
                elseif isa(varargin{i1},'SCInputs'); obj.SC_DB = varargin{i1};
                elseif isa(varargin{i1},'OrbitProp'); obj.orbProp = varargin{i1};
                elseif isa(varargin{i1},'AeroPass'); obj.aeroProp = varargin{i1};
                elseif isa(varargin{i1},'Burn'); obj.burnProp = varargin{i1};
                else; error('Invalid shared object inputs');
                end
            end

        % Reference objects passed as a masterHand encapsulating object
        elseif nargin == 1 && isa(varargin{1},'masterHand')
            obj.State = varargin{1}.State;
            obj.Body_DB = varargin{1}.Body;
            obj.SC_DB = varargin{1}.S_C;
            obj.orbProp = varargin{1}.orbProp;
            obj.aeroProp = varargin{1}.aeroProp;
            obj.burnProp = varargin{1}.burnProp;

        else
            error('Invalid Constructor Inputs')
        end
        obj.fzOpts = optimset('Tolx',obj.tolX);
        optionSet(obj);
    end
end

methods
    %% MISSION PLANNING FUNCTIONS

    function [Ntp, dVtp] = numPassCalc(obj, stateI,stateF)
        % calculate current orbit properties
        raF = stateF.ra;  rpI = stateI.rp; aI = stateI.a; mu = obj.Body_DB.mu;

        aPost = (rpI+obj.raPost)/2; % Calculate semi major axis of post capture↙
```

```
orbit
            aFinal = (raF+rpI)/2; % Calculate semi major axis of final target orbit

            VpI = sqrt(mu*(2/rpI-1/aI)); % Velocity at perigee for hyperbolic trajectory
            VpPost = sqrt(mu*(2/rpI-1/aPost)); % Velocity at perigee of post capture orbit
            VpFinal = sqrt(mu*(2/rpI-1/aFinal)); % Velocity at perigee of final aero pass orbit

            dVCapt = VpI-VpPost; % delta V required to get into initial captured orbit
            dVRest = VpPost-VpFinal; % delta V required to get from initial captured orbit to final target science apogee

            Ntp = ceil(dVRest/dVCapt); % Number of Aeropasses
            dVtp = dVRest/Ntp; % dV per aeropass
            ratp = zeros(1,Ntp); % apoapsis of each aeropass

            fprintf('Mission Planner found a solution with %d aeropasses, ~%0.3f km/s per pass\n',Ntp,dVtp);
            fprintf('**********************************************************************\n');
            fprintf('Post capture apoapsis altitude: %0.3f km\n',obj.raPost-obj.Body_DB.Re);

            for i1 = 1:Ntp

                Vptp = VpPost-dVtp*i1;
                atp = (2/rpI-Vptp^2/mu)^(-1);
                ratp(i1) = 2*atp-rpI;
                fprintf('Pass %d apoapsis altitude: %0.3f km\n',i1,ratp(i1)-obj.Body_DB.Re);

            end

            obj.raTarg = ratp;
            obj.currPass = 1;

            % set thruster to low for TCM and correction maneuvers
            obj.burnProp.Thruster = 'Low';
            optionSet(obj);

        end

        %% ORBITAL MANEUVER CALCULATION FUNCTIONS

        function Pe_Raise(obj, stateF)

            % Break out some variables from shared objects
            ra = obj.State.ra; a = obj.State.a;
```

```
            rp2 = stateF.rp; a2 = stateF.a;
            mu = obj.Body_DB.mu; m_o = obj.State.ScM;
            ISP = obj.SC_DB.ISP; go = obj.SC_DB.go; m_dot = obj.SC_DB.m_dot;

            % transfer orbit properties
            at = (ra+rp2)/2;

            % Calculate delta V requirements
            Va = sqrt(2*mu/ra-mu/a); % initial orbit velocity at apoapsis (where we↙
want initial burn)
            Vpt = sqrt(2*mu/ra-mu/at); % transfer orbit velocity at periapsis

            Vat = sqrt(2*mu/rp2-mu/at); % transfer orbit velocity at apoapsis
            Vp2 = sqrt(2*mu/rp2-mu/a2); % Final orbit velocity at periapsis

            dV1 = Vpt-Va; % Sum delta V for first burn
            dV2 = Vp2-Vat; % Sum delta V for second burn

            % calculate propellant masses and burn times
            Mp1 = m_o-m_o*exp(-dV1*1000/(ISP*go));
            Tb1 = -Mp1/m_dot; % m_dot expressed as negative value
            m1 = m_o-Mp1;
            Mp2 = m1-m1*exp(-dV2*1000/(ISP*go));
            Tb2 = -Mp2/m_dot;

            % array for two circularization burn times
            % obj.State.Tb = [Tb1 Tb2 0];
            obj.State.Tb = Tb1;
            obj.State.dVec = [1 1 1]';

            % Print periapsis raise data
            fprintf('\nFinal perigee raise solution found: %0.1f s burn for %0.2f↙
km/s\n',Tb1,dV1)


        end

        function Intpl_TCM(obj)
            % Save current state
            obj.State.saveState;

            % Determine optimal attitude to execute TCM maneuver
            [psiOpt, delta] = fminbnd(@(psi) -dVecOpt(obj,psi), obj.psiInt(1), obj.↙
psiInt(2));

            Reci = obj.State.Reci;
            Veci = obj.State.Veci;
            v = norm(Veci);
            Uv = [Veci(1)/v; Veci(2)/v; Veci(3)/v];
```

```
            ROT = [cosd(psiOpt) -sind(psiOpt) 0
                   sind(psiOpt) cosd(psiOpt) 0
                   0    0   1];
            dVec = ROT*Uv;
            obj.State.dVec = dVec;

            % Reset Burn Object
            obj.burnProp.calcSpd = 'continuous'; obj.burnProp.output = 'on';

            % Determine current unadjusted trajectory
            obj.lookForward(obj.check);

            % Is current trajectory leading us above or below the target
            % apoapsis
            if obj.State.ra < obj.raPost && obj.State.e < 1 % verify spacecraft isn't↙
interplanetary
                obj.State.reset;

                % Optomize prograde burn time
                out = lookForward(obj,obj.intTCM);
                fprintf('\nInterplanetary periapsis raise TCM necessary, %0.1f s burn↙
converged after %d iterations\n',obj.State.Tb,out);
            else
                obj.State.reset;

                % Retrograde burn necessary to lower periapsis
                obj.State.dVec = -obj.State.dVec;

                % Optomize retrograde burn time
                out = lookForward(obj,obj.intTCM);
                fprintf('\nInterplanetary periapsis lower TCM necessary, %0.1f s burn↙
converged after %d iterations\n',obj.State.Tb,out);
            end

            % burn attitude optimization (optomization of all three euler
            % angles needs to be added)
            function delta = dVecOpt(obj,psi)

                % Rotate abount velocity vector
                Vopt = obj.State.Veci;
                vopt = norm(Vopt);
                Uvopt = [Vopt(1)/vopt; Vopt(2)/vopt; Vopt(3)/vopt];

                ROTopt = [cosd(psi) -sind(psi) 0
                          sind(psi)  cosd(psi) 0
                          0          0         1];

                UvN = ROTopt*Uvopt;
                % quiver3(Reci(1),Reci(2),Reci(3),UvN(1),UvN(2),UvN(3),↙
750,'g','LineWidth',1);
```

```
            % create test burn to determine optimal burn vector
            obj.State.dVec = UvN;
            obj.State.Tb = 5;
            rp_old = obj.State.rp;

            % Set a few options, propage maneuver trajectory
            obj.burnProp.calcSpd = 'jump'; obj.burnProp.output = 'off';
            obj.burnProp.step;

            % Reset for next iteration
            rp_new = obj.State.rp;
            delta = rp_new-rp_old;
            obj.State.reset;
        end
    end

    function Aeropass_TCM(obj)

        % burn will be inline with velocity vector
        obj.State.dVec = [1 1 1]';

        % determine if periapsis altitude must be raised or decreased
        obj.lookForward(obj.check);

        % set the optomizer to target a predetermined altitude after
        % each pass
        obj.peOpt.targ = obj.raTarg(obj.currPass);

        % Is current trajectory leading us above or below the target
        % apoapsis
        if obj.State.ra < obj.raTarg(obj.currPass)
            obj.State.reset;

            % Optomize prograde burn time
            out = lookForward(obj,obj.peOpt);
            fprintf('\nPass %d periapsis raise TCM necessary, %0.1f s burn↙
converged after %d iterations\n',obj.currPass,obj.State.Tb,out);
        else
            obj.State.reset;

            % Retrograde burn necessary to lower periapsis
            obj.State.dVec = -obj.State.dVec;

            % Optomize retrograde burn time
            out = lookForward(obj,obj.peOpt);
            fprintf('\nPass %d periapsis lower TCM necessary, %0.1f s burn↙
converged after %d iterations\n',obj.currPass,obj.State.Tb,out);
        end
        obj.currPass = obj.currPass + 1;
```

```
        end

        %% TRAJECTORY OPTOMIZATION FUNCTION

        function out = lookForward(obj,optIn)

            % Save current state before any optomizations
            obj.State.saveState;

            % break out variables from options object
            order = optIn.order;
            objective = optIn.objective;
            targ = optIn.targ;
            adjust = optIn.adjust;

            % No optomization, just a forward propagation of current
            % trajectory
            if ~optIn.opt
                propForward;
                out = [];

            % Perform optomization
            elseif optIn.opt

                % Shooting method trajectory optomization optomize the
                % "adjust" property until the "objective" property equals
                % the "target" property
                [Opt,~,~,output] = fzero(@(adj) propForward(adj), optIn.range,obj.↙
fzOpts);

                % Number of iterations required
                out = output.iterations;

                % Update state with the optomized parameter
                obj.State.(adjust) = Opt;
                obj.State.step;

                % Delete the pre-optomization saved state and revert back to the
                % saved state at the start of the simulation
                obj.State.revertState;
            end

            function min = propForward(adj)
                n = length(order);

                % Will optomization be performed
                if optIn.opt
                    obj.State.(adjust) = adj;
                    obj.State.step;
```

```
                end

                for i1 = 1:n
                    % Bake in a few special cases, i.e. if a burn segment is set
                    % to follow a coast segment, stop at apoapsis
                    if i1 < n && order(i1) == 'O' && order(i1+1) == 'B'
                        opts = obj.opts2;
                    elseif i1 == n && order(i1) == 'O'
                        opts = obj.opts2;
                    else
                        opts = obj.opts1;
                    end

                    % opts.output = 'on';
                    % obj.opts1.output = 'on';
                    switch(order(i1))
                        case 'O' % Coast segment
                            obj.orbProp.step(opts);
                        case 'A' % Atmospheric flight segment
                            obj.aeroProp.step(opts);
                        case 'B' % Propulsive maneuver segment
                            obj.burnProp.step(obj.opts1);
                    end
                end

                % Generate outputs
                if optIn.opt
                    switch adjust
                        case 'FPA' % Flight path angle option
                            fprintf('Apoapsis Altitude: %.3f km at %.3f deg\n',obj.↙
State.(objective)-obj.Body_DB.Re,adj)
                        case 'Tb' % Burn time option
                            fprintf('Apoapsis Altitude: %.3f km at %.3f s burn\n',obj.↙
State.(objective)-obj.Body_DB.Re,adj)
                    end

                    % Primary objective function
                    min = obj.State.(objective)-targ;

                    % covers special interplanetary case
                    if strcmp(objective,'ra') && obj.State.e >= 1
                        min = 5e6;
                    end

                    % Reset State for next iteration
                    obj.State.reset;

                % No optomization case
                else
                    min = [];
```

```
                end
            end
        end

        function optionSet(obj)
            % Forward propagation of current trajectory
            obj.check = optoIn('order','OA');
            % interplanetary periapsis trim TCM
            obj.intTCM = optoIn('order','BOA','targ',obj.↙
raPost,'objective','ra','adjust','Tb');
            % periapsis trim between aero passes
            obj.peOpt = optoIn('order','OBOA','targ',obj.↙
raPost,'objective','ra','adjust','Tb');
        end
    end
end
```

```
classdef optoIn < matlab.System
    % Format for inputs to trajectory optomization function

    % Written by: Bohdan Wesely, MSAE at SJSU, NASA ARC Entry Systems
    % and Technology Division, May 2025, MATLAB 2024b.

    properties
        order % Order of trajectory segments to perform A: Atmospheric flight, B:↙
Burn/Maneuver, O: Coast Orbit
        targ % Target or "Objective" of optomization
    end

    properties (Dependent)
        range % Initial values for optomizer
        objective % Objective Function
        adjust % Input parameter to optomize to the objective
        opt % false (0) only one propagation will be performed true (1) optomization↙
will be performed
    end

    % Private properties to store values for dependent properties
    properties (Access = private)
        stateProps
        objectiveStore
        adjustStore
        fpaRgdef = [-4 -25] % Starting range of flight path angles
        TbRgdef = [0.01 200]
    end

    methods (Access = protected)
        % Save Object
        function s = saveObjectImpl(obj)
            s = saveObjectImpl@matlab.System(obj);
            s.stateProps = obj.stateProps;
            s.objectiveStore = obj.objectiveStore;
            s.adjustStore = obj.adjustStore;
            s.fpaRgdef = obj.fpaRgdef;
            s.TbRgdef = obj.TbRgdef;
        end

        % Load Object
        function loadObjectImpl(obj,s,isInUse)
            obj.stateProps = s.stateProps;
            obj.objectiveStore = s.objectiveStore;
            obj.adjustStore = s.adjustStore;
            obj.fpaRgdef = s.fpaRgdef;
            obj.TbRgdef = s.TbRgdef;
            loadObjectImpl@matlab.System(obj,s,isInUse);
        end
    end
```

```
    methods
        % Create names of state properties that can be used as objectives
        % for optomization
        function obj = optoIn(varargin)
            state = SCState;
            obj.stateProps = properties(state);
            setProperties(obj,nargin,varargin{:});
        end

        % Verify that order inputs are only either 'B','O', or 'A'
        function set.order(obj,val)
            if ischar(val)
                BB = ismember(val,'B');
                OO = ismember(val,'O');
                AA = ismember(val,'A');
                idx = AA | BB | OO;
                if ~any(~idx)
                    obj.order = val;
                else
                    error('First arguement must only contain O, A, or B: O:↙
cruise/orbit, A: Aerodynamic pass, B: Maneuver')
                end
            else
                error('First arguement must be a character vector: O: cruise/orbit, A:↙
Aerodynamic pass, B: Maneuver')
            end
        end

        % Verify that objective is a char and is a state property
        function set.objective(obj,val)
            if ischar(val) && ismember(val,obj.stateProps)
                obj.objectiveStore = val;
            else
                error('Objective must be a state object property character vector')
            end
        end

        % Validate target as a numeric double
        function set.targ(obj,val)
            if isa(val,"double")
                obj.targ = val;
            else
                error('Target must be a double')
            end
        end

        % Validate adjust as a char and a state property
        function set.adjust(obj,val)
            if ischar(val) && ismember(val,obj.stateProps)
```

```
                obj.adjustStore = val;
            else
                error('adjust must be a state object property character vector')
            end
        end

        % Get private stored property for objective
        function val = get.objective(obj)
            val = obj.objectiveStore;
        end

        % Get private stored property for adjust
        function val = get.adjust(obj)
            val = obj.adjustStore;
        end

        % Currently default initial values are only supported for flight
        % path angle and burn time
        function val = get.range(obj)
            if ~isempty(obj.adjust)
                switch obj.adjust
                    case 'FPA'
                        val = obj.fpaRgdef;
                    case 'Tb'
                        val = obj.TbRgdef;
                end
            else
                val = [];
            end
        end

        % Option to either optomize or simply propagate a trajectory
        function val = get.opt(obj)
            optProps = [{obj.targ} {obj.adjust} {obj.objective}];
            valid = cellfun(@isempty,optProps);

            if ~any(~valid)
                val = false;
            elseif ~any(valid)
                val = true;
            else
                % error('Three optomization properties must be provided, if no↙
optomization is desired, only supply the order input')
            end
        end
    end
end
```

```
classdef OrbitProp < matlab.System
    % This object propogates a trajectory segment, the superclass contains
    % physics for a coast under gravity only. Thrust or atmospheric flight
    % propagations are subclasses of the upper OrbitProp superclass Class

    % Written by: Bohdan Wesely, MSAE at SJSU, NASA ARC Entry Systems
    % and Technology Division, May 2025, MATLAB 2024b.

    % Public, tunable properties
    properties
        Thruster = 'Low'
        PhysTyp = 'Cruise'
        EventTyp = 'Aero'; % Location to stop integration
        laps = 5; % time to move past event to prevent false trigger
        tstep = 60; % (s) timestep
        tstep_f = 0.05; % factor to divide burn time by to create custom timestep for
burn integrations to ensure start and end points are accurate
        calcSpd = 'continuous'; % 'continuous' specifies a tspan to the ODE integrator
as a vector, 'jump' specifies only a start and end point
        output = 'on'; % print output option (typically off for optomization runs)
        resultPop = 'on'; % populate results option (typically off for optomization
runs)
        opts1 % options struct for ODE integrator
        opts2 % 2nd options struct that contains the ODE event function
        ODEfun = @ode45 % ODE integrator function
        warnFlg = true; % Altitude warning flag (prevents repeat warning messages)
        RelTol = 1e-7; % Relative Tolerance
        AbsTol = 1e-8; % Absoluate Tolerance

        % it was found with a relative and absolute tolerance of 1e-5
        % and 1e-7 respectively that the error between the first and
        % 4th orbit is on the order of a few hundred meters.
        % 1e-7 and 1e-8 tols did fix some integration issues

    end

    properties
        State % handle object for current spacecraft state
        Results % handle object for trajectory results/outputs
        Body_DB % handle object for planetary body parameters database
        SC_DB % handle object for spacecraft parameters database
        GRAM % handle object for the GRAM interface
        Time % handle object for tracking elapsed time and time dependent planet
orientation
        chemObj % handle object for atmospheric chemistry calculations
    end

    % Pre-computed constants or internal states
    properties (Access = private)
        stop % stopping criteria vector
```

```
        saveStrct % struct to save overwrite options
    end

    methods
        % Constructor: Pass State and Result Handle objects
        function obj = OrbitProp(varargin)
            % No inputs case, creates default reference objects internally
            if nargin == 0
                % Provide values for superclass constructor
                % and initialize other inputs
                obj.State = SCState;
                obj.Results = TrajResults;
                obj.Body_DB = BodyInputs;
                obj.SC_DB = SCInputs;
                obj.GRAM = gramMgr;
                obj.Time = timeMgr;
                obj.chemObj = chemMgr;

            % Individaul reference objects passed to constructor as inputs args
            elseif nargin == 7
                % When nargin ~= 0, assign to cell array,
                % which is passed to supclass constructor
                for i1 = 1:7
                    if isa(varargin{i1},'SCState'); obj.State = varargin{i1};
                    elseif isa(varargin{i1},'TrajResults'); obj.Results = varargin{i1};
                    elseif isa(varargin{i1},'BodyInputs'); obj.Body_DB = varargin{i1};
                    elseif isa(varargin{i1},'SCInputs'); obj.SC_DB = varargin{i1};
                    elseif isa(varargin{i1},'gramMgr'); obj.GRAM = varargin{i1};
                    elseif isa(varargin{i1},'timeMgr'); obj.Time = varargin{i1};
                    elseif isa(varargin{i1},'chemMgr'); obj.chemObj = varargin{i1};
                    else; error('Invalid shared object inputs');
                    end
                end

            % Reference objects passed as a masterHand encapsulating object
            elseif nargin == 1 && isa(varargin{1},'masterHand')
                obj.State = varargin{1}.State;
                obj.Results = varargin{1}.Results;
                obj.Body_DB = varargin{1}.Body;
                obj.SC_DB = varargin{1}.S_C;
                obj.GRAM = varargin{1}.GRAM;
                obj.Time = varargin{1}.Time;
                obj.chemObj = varargin{1}.chemData;
            else
                error('Invalid Constructor Inputs')
            end
        end
    end

    methods (Access = protected)
```

```
        function setupImpl(obj)
            % Setup solver options and events
            obj.opts1 = odeset('RelTol',obj.RelTol,'AbsTol',obj.AbsTol);
            obj.opts2 = obj.opts1;
            obj.opts2.Events = @(t,R) Event_fcn(t,R,obj.stop,obj.Body_DB,obj.State,obj.↙
Time);

            % placeholder function for subclasses
            setupfun(obj);
        end

        function stepImpl(obj,varargin)

            % Skips atmospheric flight integration if altitude is over
            % threshold
            if strcmp(obj.PhysTyp, 'Aero') && obj.State.Alt > obj.Body_DB.AltThr+1
                if strcmp(obj.output,'on')
                    warning('Attempting to enter atmospheric flight phase while above↙
threshold, skipping propagation segment')
                end
                return;
            end

            % populate properties based on options struct
            if nargin > 0 && ~isempty(varargin) && isa(varargin{1},'struct')
                 optsGen(obj,varargin{1});
            end

            % Generate stopping criteria for event functions
            switch obj.EventTyp
                case 'Pe' % periapsis
                    obj.stop = [1 0 0 0 0 0 1 1 0 0];
                case 'Ae' % apoapsis
                    obj.stop = [0 1 0 0 0 0 1 1 0 0];
                case 'Asc' % ascending node
                    obj.stop = [0 0 1 0 0 0 1 1 0 0];
                case 'Dsc' % descending node
                    obj.stop = [0 0 0 1 0 0 1 1 0 0];
                case 'Aero' % aerobrake (stops at certain altitude)
                    % Enable velocity stop if vehicle is in atmosphere
                    if strcmp(obj.PhysTyp,'Aero')
                        obj.stop = [0 0 0 0 1 1 1 1 1 1];
                    else
                        obj.stop = [0 0 0 0 1 1 1 1 0 0];
                    end
                case 'Crz'  % cruise option, do not stop at any orbital nodes
                    obj.stop = [0 0 0 0 0 0 1 1 0 0];
                case 'Trans' % Transfer Orbit, waits 1/2 period and accounts for burn↙
times
```

```
                obj.stop = [0 0 0 0 0 0 1 1 0 0];
            otherwise
                error('Invalid BrnTyp Value')
        end

        % Main orbit propagation integration
        mainInt(obj);

        % Populate shared results object if needed
        if strcmp(obj.resultPop,'on')
            obj.Results.Type = obj.PhysTyp;
            obj.Results.Vi = norm(obj.Results.Rt(1,4:6));
            obj.Results.raAlt = obj.State.ra-obj.Body_DB.Re; % Warning: result is↙
approximation if in ellipsoid planet mode
        end

        % Print outputs
        if strcmp(obj.output,'on')
            Outputs(obj) % print
        end

        % Reset options properties to defaults
        if nargin > 0 && ~isempty(varargin) && isa(varargin{1},'struct')
             optsReset(obj,varargin{1});
        end

        % Reset Altitude warning flag
        obj.warnFlg = true;
    end

    % Save object to MAT file
    function s = saveObjectImpl(obj)
        s = saveObjectImpl@matlab.System(obj);
        s.stop = matlab.System.saveObject(obj.stop);
        s.saveStrct = matlab.System.saveObject(obj.saveStrct);
    end

    % Load object from MAT file
    function loadObjectImpl(obj,s,isInUse)
        obj.stop = s.stop;
        obj.saveStrct = s.saveStrct;
        loadObjectImpl@matlab.System(obj,s,isInUse);
    end

end

methods (Access = protected)

    function mainInt(obj)
```

```
            % Break out a few state object properties
            T = obj.State.T; Reci = obj.State.Reci; Veci = obj.State.Veci;
            t_curr = obj.Time.elTime;

            % Perform a short integration to bring vehicle slightly past critical
            % point to prevent double triggering of event fcn
            m_o = obj.State.ScM;
            tspanJ = [t_curr t_curr+obj.laps/2 t_curr+obj.laps];
            [t_jump,Rt_jump] = obj.ODEfun(@(t,R) TwoBody(obj,t,R),tspanJ,[Reci; Veci;↙
m_o],obj.opts1);

            % Reset initial conditions after short step
            Reci = Rt_jump(end,1:3)';
            Veci = Rt_jump(end,4:6)';
            t_curr = t_jump(end);

            % Setup main trajectory segment timespan
            if strcmp(obj.calcSpd, 'continuous')
                if strcmp(obj.PhysTyp, 'Aero'); T = obj.Tmax; end
                tspan = t_curr:obj.tstep:t_curr + T;
            elseif strcmp(obj.calcSpd, 'jump')
                if strcmp(obj.PhysTyp, 'Aero'); T = obj.Tmax; end
                tspan = [t_curr t_curr + T];
            else
                error('property "calcSpd" set incorrectly')
            end

            % Main trajectory integrator call
            [t,Rt,te,ye,ie] = obj.ODEfun(@(t,R) TwoBody(obj,t,R),tspan,[Reci; Veci;↙
m_o],obj.opts2);

            % Stitch together short integration and main integration
            t = [t_jump; t]; Rt = [Rt_jump; Rt];

            % If next orbital segment is a burn
            if obj.State.Tb > 0
                % go back and find conditions 1/2 the burn time back in orbit to ensure↙
burn splits event
                [~,ix] = min(abs(t-(t(end)-obj.State.Tb/2)));

                % populate total array up until burn start
                Rt = Rt(1:ix,:);
                t = t(1:ix,:);
            end

            % Populate Results if necessary
            if strcmp(obj.resultPop,'on')
                obj.Results.t = t; obj.Results.Rt = Rt; obj.Results.te = te; obj.↙
Results.ye = ye; obj.Results.ie = ie;
                obj.Results.Tb = 0; obj.Results.dVec = 0; obj.Results.dV = 0;
```

```
            end

            % Populate shared handle objects with new trajectory data
            obj.State.Reci = Rt(end,1:3)';
            obj.State.Veci = Rt(end,4:6)';
            obj.State.ScM = Rt(end,7);
            obj.Time.elTime = t(end);

            % Update State
            obj.State.step;

        end

        function drdt = TwoBody(obj,t,R)

            % Extract Vars from struct inputs
            mu = obj.Body_DB.mu; Re = obj.Body_DB.Re;

            % Set the massflow rate to zero for cruise and aero orbits
            % when rocket engine isn't burning and S/C isn't loosing mass
            if strcmp(obj.PhysTyp, 'Cruise') || strcmp(obj.PhysTyp, 'Aero')
                m_dot = 0;
            elseif strcmp(obj.Thruster, 'Low')
                m_dot = obj.SC_DB.m_dot_low;
            elseif strcmp(obj.Thruster, 'High')
                m_dot = obj.SC_DB.m_dot;
            else
                error('property "Thruster" set incorrectly')
            end

            % Warns user if crossing below altitude threshold while in
            % cruise or burn phase
            if (strcmp(obj.PhysTyp, 'Burn') || strcmp(obj.PhysTyp, 'Cruise')) && obj.↙
State.Alt < obj.Body_DB.AltThr-1 && strcmp(obj.output,'on') && obj.warnFlg
                warning('Below atmospheric threshold while in cruise or maneuver↙
physics (vacuum only) check stopping criteria')
                obj.warnFlg = false;
            end

            % Initialization
            r = norm(R(1:3)); % calculate magnitude
            v = norm(R(4:6));

            % OBLATENESS EFFECTS
            J2 = obj.Body_DB.J2; % second zonal harmonic

            x = R(1); y = R(2); z = R(3);
            p1 = x/r*(5*z^2/r^2-1);
            p2 = y/r*(5*z^2/r^2-1);
            p3 = z/r*(5*z^2/r^2-3);
```

```
            % Perturbation vector
            P = 3/2*J2*mu*Re^2/r^4.*[p1
                                     p2
                                     p3];

            % Disable J2 perturbations
            % P = [0;0;0];

            P = Perturb(obj,t,P,R);

            % Kepler ODE in state space form
            drdt = [R(4)
                    R(5)
                    R(6)
                    -mu/r^3*R(1)+P(1)
                    -mu/r^3*R(2)+P(2)
                    -mu/r^3*R(3)+P(3)
                    m_dot];

        end

        function P = Perturb(~,~,P,~)
            % Default superclass is for a cruise orbit so unperturbed
        end
    end

    methods
        function setupfun(~)
            % placeholder for subclasses
        end

        % Print termination criteria
        function termfun(~,ie)
            if ~isempty(ie)
                switch ie(end)
                    case 1 % periapsis
                        fprintf('Integration Stopped: Periapsis Reached\n')
                    case 2 % apoapsis
                        fprintf('Integration Stopped: Apoapsis Reached\n')
                    case 3 % ascending node
                        fprintf('Integration Stopped: Ascending Node Reached\n')
                    case 4 % descending node
                        fprintf('Integration Stopped: Descending Node Reached\n')
                    case 5 % aerobrake (stops at certain altitude)
                        fprintf('Integration Stopped: Atmospheric Cutoff Altitude↙
Reached (Descending) \n')
                    case 6  % aerobrake (stops at certain altitude)
                        fprintf('Integration Stopped: Atmospheric Cutoff Altitude↙
Reached (Ascending) \n')
```

```
                    case 7 % spacecraft has impacted surface
                        fprintf('Integration Stopped: Spacecraft Impacted Surface↙
(Descending) \n')
                    case 8 % spacecraft has impacted surface from below (If you see↙
this, something is really screwed up)
                        fprintf('Integration Stopped: Spacecraft Impacted Surface↙
(Ascending) \n')
                    case 9 % Termination Velocity Reached
                        fprintf('Integration Stopped: Termination Velocity Reached \n')
                    case 10 % Termination Velocity Reached
                        fprintf('Integration Stopped: Termination Velocity Reached \n')
                    otherwise % Termination Velocity Reached
                        fprintf('Integration Stopped: Maximum time step or number or↙
orbits reached \n')
                end
            end
        end

        % Prints the current state after completing a trajectory segment
        function Outputs(obj)
            fmt = "%-20s %-10.3f %-5s\n";
            fprintf('\nIntegrator completed %s trajectory segment\n', obj.PhysTyp)
            fprintf("Current Vehicle State:\n");
            fprintf(fmt,"Altitude:",obj.State.Alt, "km")
            fprintf(fmt,"Velocity:",obj.State.V, "km/s")
            fprintf(fmt,"Flight Path Angle:",obj.State.FPA, "deg")
            fprintf(fmt,"Azimuth:",obj.State.Az, "deg")
            fprintf(fmt,"Elapsed Time:",obj.State.elTime, "s")

            if strcmp(obj.PhysTyp,'Burn')
                fprintf(fmt,"Burn Time:",obj.Results.Tb, "s")
                fprintf(fmt,"Delta V:",obj.Results.dV, "m/s")
            end
            termfun(obj,obj.Results.ie)
        end

        % populates properties based on a struct input
        function optsGen(obj,options)
            optFields = fieldnames(options);
            for fi = 1:numel(optFields)
                obj.saveStrct.(optFields{fi}) = obj.(optFields{fi});
                obj.(optFields{fi}) = options.(optFields{fi});
            end
        end

        % Reset properties to previous values after run.
        function optsReset(obj,options)
            optFields = fieldnames(options);
            for fi = 1:numel(optFields)
                 obj.(optFields{fi}) = obj.saveStrct.(optFields{fi});
```

```
            end
        end
    end
end
```

```
classdef plotProps < matlab.System
    % Contains all colors and plotting poperties for 2D aerothermal and
    % trajectory plots

    % Written by: Bohdan Wesely, MSAE at SJSU, NASA ARC Entry Systems
    % and Technology Division, May 2025, MATLAB 2024b.

    % Public, tunable properties
    properties
        Axes % Axis handle
        numPlots = 6; % Number of aerothermal plots
        aeroYax = ["qs","Js","V","alt","alt","alt"]; % x axis data
        aeroXax = ["t", "t", "t", "t","fpa","V"]; % y axis data
        resultYax = ["qsMax"]; % Batch plot run X axis data
        resultXax = ["raAlt"]; % Batch plot run Y axis data
        resultLeg = ["BC","Vi"]; % Legend Properties for batch run plot
        weightFactor = ["tPost"]; % Weight function property
        titles % Title array for plots
        pAxes % Axis handle for batch run plot
        pXaxes % X axis handle for batch run plot
        pYaxes % Y axis handle for batch run plot
        pLegend % Legend for batch run plot
        pLegCt % Legend index
        linewidth = 2; % Line width
        coloFun = @hsv % color array function (jet, hsv, parula, etc)
        pallette = "gem" % color pallete string (see MATLAB documentation)
        colorOpt = "pal" % grad (hsv, parula, jet, etc.) or pal (gem, reef, etc.)
        numColors = 4; % Number of colors before switching line style
        linestyles = ["-","--","-.",":"]; % Line styles
        styleOrder = 'aftercolor'; % aftercolor runs through colors before linestyles,↙
beforecolor runs through linestyles first
        col % Matrix of RGB triplets for color
    end

    properties
        Results % trajectory results shared object
    end

    % Pre-computed constants or internal states
    properties (Access = private)

    end

    methods (Access = protected)

        % Generate plots for initial setup
        function setupImpl(obj)
            plotQuery(obj);
        end
```

```
        function stepImpl(obj)

            % regen plots if deleted
            if ~isvalid(obj.Axes)
                plotQuery(obj)
            end

            % Plot results
            for i2 = 1:obj.numPlots

                % X axis
                Xx = obj.Results.(obj.aeroXax(i2));

                % crop to time since entry interface
                if strcmp(obj.aeroXax(i2),"t")
                    Xx = Xx-Xx(1);
                end

                % Y axis
                Yx = obj.Results.(obj.aeroYax(i2));

                % Get title and x/y label strings from results object
                xLab = obj.Results.plotLabels.(obj.aeroXax(i2)).label;
                yLab = obj.Results.plotLabels.(obj.aeroYax(i2)).label;
                Title = strcat(obj.Results.plotLabels.(obj.aeroYax(i2)).title," vs. ",↙
obj.Results.plotLabels.(obj.aeroXax(i2)).title);

                % query colors and styles
                linestyleorder(obj.Axes(i2),obj.linestyles)
                linestyleorder(obj.Axes(i2),obj.linestyles,obj.styleOrder);
                colororder(obj.Axes(i2),obj.col)

                % generate plots, allow for log plots with knudsen number
                if strcmp(obj.aeroXax(i2),'Kn')
                    semilogx(obj.Axes(i2),Xx,Yx,'LineWidth',obj.linewidth)
                elseif strcmp(obj.aeroYax(i2),'Kn')
                    semilogy(obj.Axes(i2),Xx,Yx,'LineWidth',obj.linewidth)
                else
                    plot(obj.Axes(i2),Xx,Yx,'LineWidth',obj.linewidth)
                end

                % Plot formatting
                title(obj.Axes(i2),Title)
                xlabel(obj.Axes(i2),xLab)
                ylabel(obj.Axes(i2),yLab)
                grid(obj.Axes(i2),"on")
                hold(obj.Axes(i2),"on")
            end
        end
    end
```

```
    methods

        % Plot select properties from batch runs
        function plotResults(obj,xX,yY)

            % Re-create figure if it has been deleted
            if isempty(obj.pAxes) || ~isvalid(obj.pAxes)
                figure
                obj.pAxes = axes(); obj.pLegCt = 1;
            end

            % Get title and x/y label strings from results object
            xLab = obj.Results.plotLabels.(obj.resultXax).label;
            yLab = obj.Results.plotLabels.(obj.resultYax).label;
            Title = strcat(obj.Results.plotLabels.(obj.resultYax).title," vs. ",obj.↙
Results.plotLabels.(obj.resultXax).title);

            % query colors and styles
            linestyleorder(obj.pAxes,obj.linestyles)
            linestyleorder(obj.pAxes,obj.linestyles,obj.styleOrder);
            colororder(obj.pAxes,obj.col)

            % generate plots
            plot(obj.pAxes,xX,yY,'LineWidth',obj.linewidth)
            title(obj.pAxes,Title)
            xlabel(obj.pAxes,xLab)
            ylabel(obj.pAxes,yLab)
            grid(obj.pAxes,"on")
            hold(obj.pAxes,"on")
        end

        % Setup plot color and style options
        function colOut = get.col(obj)
            switch obj.colorOpt
                case 'grad'
                    colOut = obj.coloFun(obj.numColors);
                case 'pal'
                    colOut = orderedcolors(obj.pallette);
                otherwise
                    colOut = orderedcolors("gem");
                    colOut = colOut(1:obj.numColors,:);
            end

            % Crop color array if needed
            if obj.numColors < height(colOut)
                colOut = colOut(1:obj.numColors,:);
            end
        end
```

```
        % Iteratively populate legend labels
        function setLegend(obj,str)
            obj.pLegend{obj.pLegCt} = str;
            obj.pLegCt = obj.pLegCt + 1;
        end

        % Generate legend
        function createLegend(obj)
            % legend(obj.pAxes,obj.pLegend);

            legend(obj.pAxes,obj.pLegend(1:3))
            % plot(1,1,'Color','k','LineStyle','-','LineWidth',2)
            % plot(1,1,'Color','k','LineStyle','--','LineWidth',2)
            % plot(1,1,'Color','k','LineStyle','-.','LineWidth',2)
        end

        % Generate and setup figures and axes
        function plotQuery(obj)
            figure
            obj.Axes = axes();
            for i1 = 1:obj.numPlots
                if i1 ~= 1
                    figure
                    obj.Axes(i1) = axes();
                end
            end
        end
    end
end
```

```
classdef SCInputs < matlab.System
    % Manages the spacecraft and entry vehicle geometry, contains an
    % aerodatabase object to perform all aerodynamic calculations

    % Written by: Bohdan Wesely, MSAE at SJSU, NASA ARC Entry Systems
    % and Technology Division, May 2025, MATLAB 2024b.

    % Public, tunable properties
    properties (SetObservable, AbortSet)
        % Rocket Engine and satellite Parameters
        Thr = 100; % N Thrust
        LowThr = 4; % N Low thrust output for small maneuvers
        m_o = 300; % kg Initial spacecraft mass
        ISP = 300; % seconds specific impulse
        go = 9.8; % gravity m/s^2
        D = 1; % Diameter (m)
        halfAng = 70; % Cone angle (deg)
        biCon = 30; % 2nd biconic angle (deg) Parameter must be set in aeroDB to use
        alpha = 0; % Trim Angle of Attack (deg) % potentially add conversion to↙
bank/roll angle
        beta = 0; % Trim Angle of Sideslip (deg)
        RN = .25; % Nose radius
        m_dot  % mass flow rate (kg/s)
        m_dot_low % low thruster mass flow rate (kg/s)
        A % Frontal area (m^2)
        BC % Ballistic Coefficient (kg/m^2)
        Aero_DB % Aero-database object
    end

    % Pre-computed constants or internal states
    properties (Access = private)
        SCDB
    end

    methods (Access = protected)

        function setupImpl(obj)
            % Initialize rocket engine parameters
            updateEnginePerf(obj)

            % Update Aero-database based on geometry inputs
            obj.Aero_DB = AeroDB;
            updateGeometry(obj)
            obj.Aero_DB.step;

            % Update ballistic coefficient
            updateBC(obj)
        end

        function stepImpl(~)
```

```
        end

        % Save object
        function s = saveObjectImpl(obj)
            s = saveObjectImpl@matlab.System(obj);
        end

        % Load object
        function loadObjectImpl(obj,s,isInUse)
            loadObjectImpl@matlab.System(obj,s,isInUse);
        end
    end

    methods
        % Export current object properties as a struct
        function strctOut = get.SCDB(obj)
            strctOut = struct;
            publicProperties = properties(obj);
            for fi = 1:numel(publicProperties)
                strctOut.(publicProperties{fi}) = obj.(publicProperties{fi});
            end
        end

        % Update rocket engine parameters
        function updateEnginePerf(obj)
            obj.m_dot = -obj.Thr/(obj.ISP*obj.go); % kg/s
            obj.m_dot_low = -obj.LowThr/(obj.ISP*obj.go); % kg/s
        end

        % Update geometry and aerodatabase
        function updateGeometry(obj)
            obj.A = pi/4*(obj.D^2); %Frontal area (m^2)
            obj.Aero_DB.R = obj.D/2; % Diameter (m)
            obj.Aero_DB.RN = obj.RN; % Nose Radius (m)
            obj.Aero_DB.tc1 = obj.halfAng; % Sphere cone half angle (deg)
            obj.Aero_DB.tc2 = obj.biCon; % Bi-conic half angle (deg)
            obj.Aero_DB.trimAlpha = obj.alpha; % trim angle of attack (deg)
            obj.Aero_DB.trimBeta = obj.beta; % trim angle of sideslip (deg)
        end

        % Update Ballistic Coefficient
        function updateBC(obj)
            obj.BC = obj.m_o/(obj.Aero_DB.CD*obj.A); %kg/m^2
        end
    end
end
```

```
classdef SCState < matlab.System
    % This object represents all orbital elements, state vectors and
    % properties needed to pass from one orbit propagation to another. The
    % methods are used to calculate other properties given a few

    % Written by: Bohdan Wesely, MSAE at SJSU, NASA ARC Entry Systems
    % and Technology Division, May 2025, MATLAB 2024b.

    % Public, tunable properties
    properties
        %% TOPOCENTRIC COORDINATES
        FPA = 0; % Flight Path Angle
        V = 0; % Velocity Magnitude
        Alt = 0; % Altitude
        Lat = 0; % Latitude
        Long = 0; % Longitude
        Az = 0; % Azimuth

        %% Keplerian Elements
        e  = 0; % eccentricity
        a = 0; % Semi Major Axis
        inc = 0; % inclination
        Arg = 0; % arguement of periapsis
        Asc = 0; % ascension of ascending node
        theta = 0; % true anamoly

        %% ADDITIONAL ORBITAL PARAMETERS
        minalt = 0; % altitude at periapsis Change between 120 and 150 km
        rp = 0; % Radius at perigee
        ra = 0; % Radius at apogee
        h = 0; % Angular Momentum
        T = 0; % Orbital Period
        v_inf = 0; % Escape Velocity

        %% POSITION AND VELOCITY VECTORS
        Reci = zeros(3,1); % Vehicle Position Vector
        Veci = zeros(3,1); % Vehicle Velocity Vector
        QECI = zeros(3,3); % ECI to Perifocal Coordinate Transformation Matrix
        Qmat = zeros(3,3); % ECI to ENZ Coordinate Transformation Matrix
        R = 0; % Position Magnitude

        %% TIME PARAMETERS
        elTime = 0; % Elapsed time

        %% MASS AND MANEUVERING PARAMETERS
        ScM = 0; % Spacecraft Mass
        Tb = 0; % Burn Time
        dVec = zeros(3,1); % Burn Vector
    end
```

```
    properties (Dependent)
        Rad % local body radius dependent on whether spherical or ellipsoid
    end

    %% Shared Handle Objects and Internal States
    properties (Access = private)
        radSave % Stored value for local body radius
        prevState % Previous saved states
        StStrct % Struct Representing all elements in state
        Body_DB % Planetary parameters shared handle object
        timeObj % Time dependent property shared handle object
        resetFlag = false; % Flag indicating wheter object has just reset
    end

    methods
        % Constructor: Pass State and Result Handle objects
        function obj = SCState(InState)
            % Create a state based on struct input
            if nargin ~= 0
                popState(obj,InState)
            end
        end
    end

    %% Protected System Object Methods
    % System object specific methods like stepImpl, resetImpl,
    % processTunedPropertiesImpl, saveObjectImpl, and loadObjectImpl

    methods (Access = protected)

        function processTunedPropertiesImpl(obj)
            if obj.resetFlag
                obj.resetFlag = false;
            else

                % Check if position and velocity vectors have changed
                rChg = isChangedProperty(obj,'Reci');
                vChg = isChangedProperty(obj,'Veci');
                if rChg && vChg
                    ECItoKep(obj)
                    ECItoLLA(obj)
                else

                    % Check if keplerian elements have changed
                    aChg = isChangedProperty(obj,'a');
                    eChg = isChangedProperty(obj,'e');
                    incChg = isChangedProperty(obj,'inc');
                    ArgChg = isChangedProperty(obj,'Arg');
                    AscChg = isChangedProperty(obj,'Asc');
                    ThetaChg = isChangedProperty(obj,'theta');
```

```
                    minaltChg = isChangedProperty(obj,'minalt');
                    if aChg || eChg || incChg || ArgChg || AscChg || ThetaChg ||↙
minaltChg
                        % allows adjustment of minimum altitude
                        if minaltChg
                            obj.rp = obj.minalt+obj.Rad; % periapsis
                            obj.a = obj.rp/(1-obj.e);
                            obj.e = obj.e;
                        end
                        KeptoECI(obj)
                        ECItoLLA(obj)
                    else

                        % Check if topocentric coordinates have changed
                        VChg = isChangedProperty(obj,'V');
                        AltChg = isChangedProperty(obj,'Alt');
                        LongChg = isChangedProperty(obj,'Long');
                        LatChg = isChangedProperty(obj,'Lat');
                        fpaChg = isChangedProperty(obj,'FPA');
                        AzChg = isChangedProperty(obj,'Az');

                        if VChg || AltChg || LongChg || LatChg || fpaChg || AzChg
                            LLAtoECI(obj)
                            ECItoKep(obj)
                        end
                    end
                end
                ExtraProps(obj)
            end
        end

        function stepImpl(~)

        end

        % reset supports only two saved states, one as an initial starting
        % state, and one prior to any optomization or look-forward
        % functions
        function resetImpl(obj)
            if length(obj.prevState) == 2
                popState(obj,obj.prevState(2))
                obj.timeObj.elTime = obj.prevState(2).elTime;

                % update state to reinitialize properties
                obj.resetFlag = true; obj.step;
            elseif isscalar(obj.prevState)
                popState(obj,obj.prevState)
                obj.timeObj.elTime = obj.prevState.elTime;

                % update state to reinitialize properties
```

```
            obj.resetFlag = true; obj.step;
        end
    end

    % Save state object
    function s = saveObjectImpl(obj)
        s = saveObjectImpl@matlab.System(obj);
        s.Body_DB = obj.Body_DB;
        s.timeObj = obj.timeObj;
        s.prevState = obj.prevState;
        s.resetFlag = obj.resetFlag;
    end

    % Load state object
    function loadObjectImpl(obj,s,isInUse)
        obj.Body_DB = s.Body_DB;
        obj.timeObj = s.timeObj;
        obj.prevState = s.prevState;
        obj.resetFlag = s.resetFlag;
        loadObjectImpl@matlab.System(obj,s,isInUse);
    end
end

methods
    %% SAVED STATE MANAGEMENT
    % State object supports up to two saved states one at the start of
    % the trajectory, and a 2nd prior to any optomizations

    % Clears previous states and sets the current state to the initial
    % state
    function newState(obj)
        obj.prevState = [];
        obj.saveState;
    end

    % Stores previous states in structs
    function saveState(obj)
        % supports storing of two previous states
        if isempty(obj.prevState)
            obj.prevState = obj.StStrct;
        else
            obj.prevState = [obj.prevState(1) obj.StStrct];
        end
    end

    % reduces the number of saved states from 2 to 1
    function revertState(obj)
        if length(obj.prevState) == 2
            obj.prevState = obj.prevState(1);
        end
```

```
        end

        % Populates properties from struct
        function popState(obj,inStruct)
            publicProperties = properties(obj);
            if  length(fieldnames(inStruct)) == numel(publicProperties)
                for fi = 1:numel(publicProperties)
                    obj.(publicProperties{fi}) = inStruct.(publicProperties{fi});
                end
            end
        end

        %% COORDINATE TRANSFORMATION FUNCTIONS

        % Keplerian Elements to ECI position and velocity Vector
        function KeptoECI(obj)
            % Extract Gravitational Parameter
            mu = obj.Body_DB.mu;

            % Angular momentum
            obj.h = sqrt(obj.a*mu)*sqrt(1-obj.e^2); % km^2/s

            % Calculate r and v in perifocal frame
            RpF = obj.h^2/(mu*(1+obj.e*cosd(obj.theta))).*[cosd(obj.theta);sind(obj.↙
theta);0];
            VpF = mu/obj.h.*[-sind(obj.theta); obj.e+cosd(obj.theta); 0];

            % formulate perifocal to ECI transform matrix
            Qeci1 = [cosd(obj.Arg) sind(obj.Arg) 0
                -sind(obj.Arg) cosd(obj.Arg) 0
                0 0 1];

            Qeci2 = [1 0 0
                0 cosd(obj.inc) sind(obj.inc)
                0 -sind(obj.inc) cosd(obj.inc)];

            Qeci3 = [cosd(obj.Asc) sind(obj.Asc) 0
                -sind(obj.Asc) cosd(obj.Asc) 0
                0 0 1];

            % Formulate matrix
            obj.QECI = Qeci1*Qeci2*Qeci3;

            % compute transpose
            obj.QECI = obj.QECI';

            % ECI Position and Velocity vector
            obj.Reci = obj.QECI*RpF;
            obj.Veci = obj.QECI*VpF;
            obj.R = norm(obj.Reci);
```

```
    end

    % ECI position and velocity Vector to keplerian Elements
    function ECItoKep(obj)
        % Extract Gravitational Parameter
        mu = obj.Body_DB.mu;

        % Total Radius
        obj.R = norm(obj.Reci); % km

        % radial velocity
        vrad = dot(obj.Reci,obj.Veci)/obj.R;

        % calculating angular momentum
        H = cross(obj.Reci,obj.Veci); % km^2/s
        obj.h = sqrt(dot(H,H)); % km^2/s

        % inclination
        obj.inc = acosd(H(3)/obj.h);
        if obj.inc == 180 % establish that 0 and 180 inclinationa are the same
            obj.inc = 0;
        end

        % Nodal Vector
        K = [0 0 1];
        N = cross(K,H);
        n = norm(N);

        % Calculate ascending node with quadrant ambiguity
        if N(2) >= 0
            obj.Asc = acosd(N(1)/n);
        else
            obj.Asc = 360-acosd(N(1)/n);
        end

        % calculating eccentricity vector and magnitude
        E = cross(obj.Veci,H)./mu-obj.Reci./obj.R;
        obj.e = norm(E);

        % calculate argument of periapsis with quadrant ambiguity
        if E(3) >= 0
            obj.Arg = acosd(dot(N,E)/(n*obj.e));
        else
            obj.Arg = 360-acosd(dot(N,E)/(n*obj.e));
        end

        % calculate true anamoly with quadrant ambiguity (use radial velocity)
        if vrad >= 0
            obj.theta = acosd(dot(E/obj.e,obj.Reci/obj.R));
```

```
        else
            obj.theta = 360-acosd(dot(E/obj.e,obj.Reci/obj.R));
        end

        % Semi major axis
        obj.a = obj.h^2/(mu*(1-obj.e^2));

        % Formulate Matrix
        Qeci1 = [cosd(obj.Arg) sind(obj.Arg) 0
            -sind(obj.Arg) cosd(obj.Arg) 0
            0 0 1];

        Qeci2 = [1 0 0
            0 cosd(obj.inc) sind(obj.inc)
            0 -sind(obj.inc) cosd(obj.inc)];

        Qeci3 = [cosd(obj.Asc) sind(obj.Asc) 0
            -sind(obj.Asc) cosd(obj.Asc) 0
            0 0 1];

        obj.QECI = Qeci1*Qeci2*Qeci3;

        % compute transpose
        obj.QECI = obj.QECI';

    end

    % Topocentric coordinates to ECI position and velocity Vector
    function LLAtoECI(obj)

        % Get current sidereal time from time object
        W = obj.timeObj.Wcurr;

        % longitude (long) must be expressed in 0-360 scale for this calculation
        sid = W + obj.Long;

        % Rotation matrix from the ENZ (East, North, Zenith) frame to the ECI
        % (Equator Centered Inertial) Frame (switch away from cosd/sind
        % to radians for slight performance boost)
        Q = [-sind(sid) -sind(obj.Lat)*cosd(sid) cosd(obj.Lat)*cosd(sid)
             cosd(sid) -sind(obj.Lat)*sind(sid) cosd(obj.Lat)*sind(sid)
             0          cosd(obj.Lat)           sind(obj.Lat)          ];

        % Position vector in the ENZ frame
        Renz = [0; 0; obj.Rad+obj.Alt];

        % Create ECI position Vector
        obj.Reci = Q*Renz;

        % Velocity vector in the ENZ frame
```

```
        Venz = [cosd(obj.FPA)*sind(obj.Az)
                cosd(obj.FPA)*cosd(obj.Az)
                sind(obj.FPA)];

        % Create ECI velocity Vector
        obj.Veci = obj.V*Q*Venz;
    end

    % ECI position and velocity Vector to topocentric coordinates
    function ECItoLLA(obj)
        % Get current sidereal time from time object
        W = obj.timeObj.Wcurr;

        % Enforce longitude convention is 0 to 360
        sid = atan2d(obj.Reci(2),obj.Reci(1));
        if sid < 0
            sid = 360 + sid;
        end

        % Calculate Longitude
        obj.Long = sid-W;
        if obj.Long < 0 % Longitude convention is 0 to 360
            obj.Long = 360 + obj.Long;
        end

        % Calculate Geocentric Latitude
        obj.Lat = 90-atan2d(sqrt(obj.Reci(1)^2+obj.Reci(2)^2),obj.Reci(3));

        % Geocentric Altitude
        obj.Alt = norm(obj.Reci)-obj.Rad;

        % Rotation matrix from the ENZ (East, North, Zenith) frame to the ECI
        % (Equator Centered Inertial) Frame (switch away from cosd/sind
        % to radians for slight performance boost)
        Q = [-sind(sid) -sind(obj.Lat)*cosd(sid) cosd(obj.Lat)*cosd(sid)
            cosd(sid) -sind(obj.Lat)*sind(sid) cosd(obj.Lat)*sind(sid)
            0          cosd(obj.Lat)           sind(obj.Lat)           ];

        % Inertial Velocity
        obj.V = norm(obj.Veci);
        Venz = Q'*obj.Veci/obj.V;

        % Flight Path Angle
        obj.FPA = asind(Venz(3));

        % Necessary to force a real value here as roundoff errors can produce a
        % value slightly above 1 (on order of 1+1e-16) and acos will spit out
        % an imaginary value (often occurs near north pole)
        % Azimuth
        obj.Az = real(acosd(Venz(2)/cosd(obj.FPA)));
```

```
            % Quadrant Check (supports -180 to 180 convention)
            if Venz(1) < 0
                obj.Az = -obj.Az;
            end
            obj.Qmat = Q;
        end

        % Additional orbital parameters dependent on keplerian elements
        function ExtraProps(obj)
            % Bounded Orbit
            if obj.e < 1
                obj.T = 2*pi/sqrt(obj.Body_DB.mu/obj.a^3);

            % Interplanetary Hyperbolic Orbit
            else % bad practice hardcoded limit, need additional property to limit↙
maximum time
                obj.T = 2*pi/sqrt(obj.Body_DB.mu/900000^3);
                obj.v_inf = sqrt(-obj.Body_DB.mu/obj.a);
            end
            obj.rp = obj.a*(1-obj.e);
            obj.ra = obj.a*(1+obj.e);
            obj.minalt = obj.rp - obj.Body_DB.Re;
        end

        %% GET/SET METHODS

        % Create a struct of all current state properties
        function Out = get.StStrct(obj)
            publicProperties = properties(obj);
            for fi = 1:numel(publicProperties)
                Out.(publicProperties{fi}) = obj.(publicProperties{fi});
            end
        end

        % Extract struct private property
        function Out = passStrct(obj)
            Out = obj.StStrct;
        end

        % Extract Elapsed time from time manager object
        function timeOut = get.elTime(obj)
            if isempty(obj.timeObj)
                timeOut = [];
            elseif isa(obj.timeObj,'timeMgr')
                timeOut = obj.timeObj.elTime;
            else
                error('Invalid State inputs, time manager object may be set↙
incorrectly')
            end
```

```
        end

        % Position dependent body radius, variable if ellipsoid planet
        % model is used
        function radOut = get.Rad(obj)
            if isempty(obj.Body_DB)
                radOut = [];
            elseif strcmp(obj.Body_DB.planModel,'sphere')
                radOut = obj.Body_DB.Re; obj.radSave = radOut;
            elseif strcmp(obj.Body_DB.planModel,'ellipse')
                Re = obj.Body_DB.Re; Rp = obj.Body_DB.Rp;
                radOut = Re*Rp/(sqrt((Rp*cosd(obj.Lat))^2+(Re*sind(obj.Lat))^2));
                obj.radSave = radOut;
            else
                error('Planet shape model must be either "ellipse" or "sphere" ')
            end
        end

        % Set private property to store value
        function set.Rad(obj,val)
            obj.radSave = val;
        end

        % Shared handle object for planetary body inputs
        function setBody(obj,bodIn)
            obj.Body_DB = bodIn;
        end

        % Shared handle object time dependent properties
        function setTime(obj,timeIn)
            obj.timeObj = timeIn;
        end
    end
end
```

```
classdef timeMgr < matlab.System
    % Handles time dependent planetary properties

    % Written by: Bohdan Wesely, MSAE at SJSU, NASA ARC Entry Systems
    % and Technology Division, May 2025, MATLAB 2024b.

    % Public, tunable properties
    properties
        planet % Planet
        a0 % Polar axis orientation angle
        d0 % Polar axis orientation angle
        Wcurr % Current sidereal time
        W0 % Initial sidereal time
        JDcurr % Current Julian date
        JDelapse % Julian date since standard epoch
        elTime = 0; % elapsed time
        Epoch = 2451545; % Standard Epoch 2000 January 1 12 h TDB
        wNut = 'off' % model nutation of planetary body rotation rate
        ICRF = 'off' % Model trajectory propagations in the high accuracy
                     % ICRF inertial frame, Z axis points normal to the ICRF
                     % equator and the tilt and precession of the planetary
                     % rotation axis is modeled with the a0 and d0
                     % parameters
        setFlag = false
    end

    properties (SetObservable, AbortSet)
        JD0
        startTime = datetime(2025,3,25,12,0,0);
    end

    methods (Access = protected)
        function setupImpl(obj)
            setW0(obj);
        end

        function stepImpl(~)

        end
    end

    methods

        function obj = timeMgr
            % allows the julian date or UTC time to automatically update
            % when the other is set
            addlistener(obj,'JD0','PostSet',@obj.setJD0);
            addlistener(obj,'startTime','PostSet',@obj.setStartTime);
        end
```

```
        function setW0(obj)
            obj.W0 = obj.Wcurr; % initialize sidereal time
        end

        function jOut = get.JDcurr(obj)
            jOut = obj.JD0 + obj.elTime/86400;
        end

        function jOut = get.JDelapse(obj)
            jOut = obj.JDcurr-obj.Epoch;
        end

        function wOut = get.Wcurr(obj)
            if isempty(obj.planet); error('planet not set in time object'); end
            switch obj.planet
                case 'Venus'
                    wOut = 160.20 - 1.4813688*obj.JDelapse;
                case 'Uranus'
                    wOut = 203.81 - 501.1600928*obj.JDelapse;
                case 'Neptune'
                    N = 357.85 + 52.316*obj.JDelapse/36525;
                    wOut = 249.978+541.1397757*obj.JDelapse-0.48*sin(N);
                case 'Jupiter'
                    wOut = 284.95+870.536*obj.JDelapse;
                case 'Earth'
                    wOut = 360.9852*obj.JDelapse; % Warning, Earth rotation model↙
should refer to IERS data
                case 'Mars'
                    wOut = 176.049863 + 350.891982443297*obj.JDelapse + 0.555;
                    if strcmp(wOut,'on')
                        T = obj.JDelapse/36525;
                        wOut = 176.049863 + 350.891982443297*obj.JDelapse...
                            +  0.000145*sin(129.071773 + 19140.0328244*T )...
                            + 0.000157*sin(36.352167 + 38281.0473591*T )...
                            + 0.000040*sin(56.668646 + 57420.9295360*T )...
                            + 0.000001*sin(67.364003 + 76560.2552215*T )...
                            + 0.000001*sin(104.792680 + 95700.4387578*T )...
                            + 0.584542*sin(95.391654 + 0.5042615*T );
                    end
                case 'Titan'
                    wOut = 186.5855 + 22.5769768*obj.JDelapse;
                otherwise
                    error('Invalid Planet Entry')
            end

            % sidereal time in degrees from 0 to 360
            wOut = wOut/360;
            wOut = (wOut-floor(wOut))*360;

            % Archinal, B.A., Acton, C.H., A'Hearn, M.F. et al. Report of the IAU↙
```

```
Working Group on Cartographic Coordinates and
            % Rotational Elements: 2015. Celest Mech Dyn Astr 130, 22 (2018). https:↙
//doi.org/10.1007/s10569-017-9805-5

        end
    end

    methods (Static)
        function setJD0(~,evnt)
            if ~evnt.AffectedObject.setFlag
                jd = evnt.AffectedObject.JD0;
                evnt.AffectedObject.setFlag = true;
                evnt.AffectedObject.startTime = datetime↙
(jd,'convertfrom','juliandate');
                evnt.AffectedObject.setFlag = false;
                setW0(evnt.AffectedObject);
            end
        end

        function setStartTime(~,evnt)
            if  isa(evnt.AffectedObject.startTime,'datetime')
                if ~evnt.AffectedObject.setFlag
                    evnt.AffectedObject.setFlag = true;
                    evnt.AffectedObject.JD0 = juliandate(evnt.AffectedObject.↙
startTime);
                    evnt.AffectedObject.setFlag = false;
                    setW0(evnt.AffectedObject);
                end
            else
                warning('Attempted to set start time in time manager with a non↙
datetime formatted object')
            end
        end
    end
end
```

```
classdef TrajPlot < matlab.System
    % Handles 3D plotting of trajectory results. Tabulates and store
    % results across multiple trajectory segments

    % Written by: Bohdan Wesely, MSAE at SJSU, NASA ARC Entry Systems
    % and Technology Division, May 2025, MATLAB 2024b.

    % Pre-computed constants or internal states
    properties
        resultCell % Run history cell array of all trajectory segments
        stateCell % Run history cell array of state values at nodes between trajectory segments
        ResultsStr % output struct array of entire trajectory history
        StateStr % output struct array of all vehicle states at trajectory nodes
        seg = 1; % Number of trajectory segments
        Axes % 3D plot axes handle
        profileResults % Cell array for batch run results
        profileAxes % Axis handle for batch run results plot
    end

    properties
        State % Object Handle for Current Vehicle State Between Segments
        Results % Object handle for Trajectory Outputs of one orbital segment
        Body_DB % handle object for planetary body parameters database
        SC_DB % handle object for spacecraft parameters database
        plotData % handle object for managing plotting options
    end

    methods
        % Constructor: Pass State and Result Handle objects
        function obj = TrajPlot(varargin)
            % No inputs case, creates default reference objects internally
            if nargin == 0
                % Provide values for superclass constructor
                % and initialize other inputs
                obj.State = SCState;
                obj.Results = TrajResults;
                obj.Body_DB = BodyInputs;
                obj.SC_DB = SCInputs;
                obj.plotData = plotProps;

            % Individaul reference objects passed to constructor as inputs args
            elseif nargin == 5
                % When nargin ~= 0, assign to cell array,
                % which is passed to supclass constructor
                for i1 = 1:5
                    if isa(varargin{i1},'SCState'); obj.State = varargin{i1};
                    elseif isa(varargin{i1},'TrajResults'); obj.Results = varargin{i1};
                    elseif isa(varargin{i1},'BodyInputs'); obj.Body_DB = varargin{i1};
                    elseif isa(varargin{i1},'SCInputs'); obj.SC_DB = varargin{i1};
```

```
                elseif isa(varargin{i1},'plotProps'); obj.plotData = varargin{i1};
                else; error('Invalid shared object inputs');
                end
            end

        % Reference objects passed as a masterHand encapsulating object
        elseif nargin == 1 && isa(varargin{1},'masterHand')
            obj.State = varargin{1}.State;
            obj.Results = varargin{1}.Results;
            obj.Body_DB = varargin{1}.Body;
            obj.SC_DB = varargin{1}.S_C;
            obj.plotData = varargin{1}.plotData;
        else
            error('Invalid Constructor Inputs')
        end
    end
end

methods (Access = protected)

    % Setup, called only once upon first step
    function setupImpl(obj)
        initPlot(obj)
    end

    % Step: plots the trajectory contained within the results shared
    % handle object
    function stepImpl(obj)

        % Skip plot if results are empty
        if ~isempty(obj.Results.Rt)
            Rt = obj.Results.Rt;

            % Red for aero segment, blue for coast, and green for burn
            % (Make adjustable in future)
            switch obj.Results.Type
                case 'Aero'
                    C = 'r';
                case 'Cruise'
                    C = 'b';
                case 'Burn'
                    C = 'g';
                otherwise
            end

            % If plot has been closed or axes deleted, create a new
            % figure window and plot
            if ~isvalid(obj.Axes)
                initPlot(obj)
            end
```

```
            % Plot trajectory
            plot3(obj.Axes,Rt(:,1),Rt(:,2),Rt(:,3),'LineWidth',1.25,'Color',C);
            hold on

        end

        % Populate results and current state into a run history cell
        % array
        obj.resultCell{obj.seg} = obj.Results.step;
        obj.stateCell{obj.seg} = obj.State.passStrct;
        obj.seg = obj.seg + 1;
    end

    % Clear run history results and state data
    function resetImpl(obj)
        obj.resultCell = [];
        obj.stateCell = [];
        obj.seg = 1;
    end
end

methods

    % Setup function for 3D trajectory plot
    function initPlot(obj)
        Re = obj.Body_DB.Re;
        figure
        obj.Axes = axes();

        % plot primary body location
        % scatter(0,0,'r');
        hold on

        % Label formatting
        xlabel('ec_{x} (km)'); ylabel('ec_{y} (km)'); zlabel('ec_{z} (km)');

        % % plot event locations
        % scatter3(Ye(:,1),Ye(:,2),Ye(:,3));
        % hold on

        % Add orgin and ECI reference frame
        quiver3(0,0,0,1,0,0,Re+0.25*Re,'r','LineWidth',3,'MaxHeadSize',1)
        quiver3(0,0,0,0,1,0,Re+0.25*Re,'g','LineWidth',3,'MaxHeadSize',1)
        quiver3(0,0,0,0,0,1,Re+0.25*Re,'b','LineWidth',3,'MaxHeadSize',1)

        % Add a wiremesh around planet
        [X,Y,Z] = sphere;
        r = Re;
        X2 = X * r;
```

```
            Y2 = Y * r;
            Z2 = Z * r;

            % Create planet visualization and format the 3D plot
            mesh(X2+5,Y2-5,Z2,'FaceAlpha',0.1,'EdgeColor',0.75*obj.Body_DB.RGB);
            surf(X2+5,Y2-5,Z2,'FaceAlpha',0.35,'EdgeColor','none','FaceColor',obj.↙
Body_DB.RGB,'LineWidth',.1);
            ax = gca;
            ax.ClippingStyle = "rectangle";
            daspect([1 1 1]) % fix aspect ratio
            grid on
        end

        % Function to export all trajectory results and node states as structs
        function [ResultsStr, StateStr] = getResults(obj)
            ResultsStr = [obj.resultCell{:}];
            StateStr = [obj.stateCell{:}];
        end

        % Plots results from batch runs (warning: experimental)
        function optOut = plotResult(obj,n)
            % n is the amount of profiles in each x-y plot step
            m = length(obj.resultCell);
            resultProfile = [obj.resultCell{m-n+1:m}];

            xDat = obj.plotData.resultXax;
            xX = [resultProfile.(xDat)];

            yDat = obj.plotData.resultYax;
            yWeight = obj.plotData.weightFactor;
            yW = [resultProfile.(yWeight)];
            yP = [resultProfile.(yDat)];
            % yY = [resultProfile.(yDat)];
            % % Create a weighted profile between two constraints
            wRange = [yW(end)  yW(1)];
            yRange = [yP(1)  yP(end)];
            %
            yY = sqrt(((yW-wRange(2))./(wRange(1)-wRange(2))).^2 + ((yP-yRange(2))./↙
(yRange(1)-yRange(2))).^2);
            [~,I] = min(yY);
            optOut = yP(I);
            % obj.plotData.plotResults(xX,yY);


            obj.plotData.plotResults(xX,yP);
            % yyaxis right
            % obj.plotData.plotResults(xX,yW/3600);

            ticks = [500 100000:100000:900000];
            xticks(ticks);
```

```
            xticklabels(string(ticks));
            xlabel('Apoapsis Altitude (km)');

            % Create Legend Labels
            var1 = obj.plotData.resultLeg(1);
            var2 = obj.plotData.resultLeg(2);
            label = join([var1,'=',resultProfile(n).(var1),',',var2,'=',resultProfile↙
(n).(var2)]);
            obj.plotData.setLegend(label)
        end

        % Creates an output summary table of all atmospheric entries
        function AeroTab(obj)

            % Number of trajectory segments
            n = length(obj.resultCell);

            % Column labels
            aeroLabel = {'Pass';'Peak Conv. Heat Flux (W/cm^2)';'Conv. Heat Load↙
(J/cm^2)';'Delta V Lost (km/s)';'EFPA (deg)'};

            k1 = 1;
            for i1 = 1:n

                % Filter by atmospheric flight segments only
                if ~isempty(obj.resultCell{i1}) && strcmp(obj.resultCell{i1}.↙
Type,'Aero')

                    % Heat Rate and Heat Load
                    Qs(k1) = obj.resultCell{i1}.qsMax;
                    Js(k1) = obj.resultCell{i1}.jsMax;

                    % Delta V lost with each pass
                    dV(k1) = obj.resultCell{i1}.dVaero;

                    % Flight Path angle at entry interface
                    EFPA(k1) = obj.resultCell{i1}.fpaI;

                    % Pass number
                    pass(k1) = append("Pass ",num2str(k1));
                    k1 = k1 + 1;
                end
            end

            % Generate and display table
            fprintf('*****Atmospheric Entries Summary*****\n')
            aeroTab = table(pass',Qs',Js',dV',EFPA','VariableNames',aeroLabel);
            disp(aeroTab);
        end
```

```
        % Creates an output summary table of all propulsive maneuvers
        function BurnTab(obj)

            % Number of trajectory segments
            n = length(obj.resultCell);

            % Column labels
            burnLabel = {'Maneuver';'Delta V (m/s)';'Burn Time (s)';'Propellant (kg)'};

            j1 = 1;
            for i1 = 1:n

                % Filter by atmospheric flight segments only
                if strcmp(obj.resultCell{i1}.Type,'Burn')

                    % Index
                    burn(j1) = string(j1);

                    % Delta V
                    DV(j1) = obj.resultCell{i1}.dV;

                    % Burn Time
                    TB(j1) = obj.resultCell{i1}.Tb;

                    % Propellant mass usage
                    Prop(j1) = obj.stateCell{i1-1}.ScM - obj.stateCell{i1}.ScM;
                    j1 = j1 + 1;
                end
            end

            % Calculate totals
            burn(j1) = 'Totals'; DV(j1) = sum(DV); TB(j1) = sum(TB); Prop(j1) = sum↙
(Prop);

            % Generate and display table
            fprintf('\n*****Maneuver Con-Ops Summary*****\n')
            burnTab = table(burn',DV',TB',Prop','VariableNames',burnLabel);
            disp(burnTab);
        end
    end
end
```

```
classdef TrajResults < matlab.System
    % Stores and manages trajectory time history results

    % Written by: Bohdan Wesely, MSAE at SJSU, NASA ARC Entry Systems
    % and Technology Division, May 2025, MATLAB 2024b.

    % Public, tunable properties
    properties
        Type % Obital Segment Type (Cruise, Aero, or Burn)
        Rt % Trajectory position and velocity array (km, km/s)
        t % Trajectory time array (s)
        ye % Position and velocity at event (km, km/s)
        ie % Event type indication
        te % Time of Event (s)
        qs % Stagnation heat flux (W/cm^2)
        Js % stagnation heat load (J/cm^2)
        qsMax % Maximum Stag heat flux (W/cm^2)
        jsMax % Maximum Stag heat load (J/cm^2)
        dVec % Attitude vector for burns (km/s)
        Tb % Burn time for burns (s)
        dV % Total delta V for burns (m/s)
        dVaero % Delta V lost from drag (km/s)
        alt % Geocentric Altitude (km)
        V % Inertial Velocity (km/s)
        Vi % Initial Velocity (km/s)
        fpa % Flight Path Angle (deg)
        fpaI % Initial Flight Path Angle (deg)
        tPost % Orbital period of post exit trajectory (s)
        rho % Density (kg/m^3)
        Kn % Knudsen Number
        BC % Ballistic Coefficient (kg/m^2)
        raAlt % Apoapsis Altitude (km)
    end

    properties
        plotLabels % Property storing all labels and units for plotting
        StStrct % Property for outputting Results in struct format
    end

    methods
        % Class Constructor
        function obj = TrajResults
            populateLabels(obj);
        end
    end

    methods (Access = protected)
        % Create a Struct of all time history Properties
        function Out = stepImpl(obj)
            publicProperties = properties(obj);
```

```
            for fi = 1:numel(publicProperties)
                if ~strcmp(publicProperties{fi},'plotLabels') && ~strcmp↙
(publicProperties{fi},'StStrct')
                    obj.StStrct.(publicProperties{fi}) = obj.(publicProperties{fi});
                end
            end
            Out = obj.StStrct;
        end
    end

    methods
        % Label database for all results (currently only a fraction of all
        % possible time history results)
        function populateLabels(obj)
            obj.plotLabels = struct();
            obj.plotLabels.t.label = "Time Since Entry Interface (s)";
            obj.plotLabels.t.title = "Time";
            obj.plotLabels.qs.label = "Heat Flux (W/cm^2)";
            obj.plotLabels.qs.title = "Stagnation Heat Flux";
            obj.plotLabels.Js.label = "Heat Load (J/cm^2)";
            obj.plotLabels.Js.title = "Stagnation Heat Load";
            obj.plotLabels.alt.label = "Altitude (km)";
            obj.plotLabels.alt.title = "Geocentric Altitude";
            obj.plotLabels.V.label = "Velocity (km/s)";
            obj.plotLabels.V.title = "Velocity";
            obj.plotLabels.fpa.label = "Flight Path Angle (deg)";
            obj.plotLabels.fpa.title = "Entry Flight Path Angle";
            obj.plotLabels.rho.label = "Density (kg/m^3)";
            obj.plotLabels.rho.title = "Density";
            obj.plotLabels.Kn.label = "K_n";
            obj.plotLabels.Kn.title = "Knudsen Number";
            obj.plotLabels.Vi.label = "V_i";
            obj.plotLabels.Vi.title = "Entry Velocity (km/s)";
            obj.plotLabels.BC.label = "B_C";
            obj.plotLabels.BC.title = "Ballistic Coefficient (kg/m^2)";
            obj.plotLabels.raAlt.label = "R_a (km)";
            obj.plotLabels.raAlt.title = "Apoapsis Altitude (km)";
            obj.plotLabels.qsMax.label = "Heat Flux (W/cm^2)";
            obj.plotLabels.qsMax.title = "Peak Conv. Heat Flux";
            obj.plotLabels.jsMax.label = "Heat Load (J/cm^2)";
            obj.plotLabels.jsMax.title = "Conv. Heat Load";
            obj.plotLabels.dVaero.label = "Delta V (km/s)";
            obj.plotLabels.dVaero.title = "Delta V due to Drag";
            obj.plotLabels.fpaI.label = "Flight Path Angle (deg)";
            obj.plotLabels.fpaI.title = "Initial Flight Path Angle";
            obj.plotLabels.tPost.label = "Orbital Period (hr)";
            obj.plotLabels.tPost.title = "Post-Exit Orbital Period";
        end

        % calculates inertial velocity from trajectory data
```

```
        function vOut = get.V(obj)
            if isempty(obj.Rt)
                vOut = [];
            else
                vOut = vecnorm(obj.Rt(:,4:6),2,2);
            end
        end
    end
end
```