PLAV: A Python 6 Degree-of-Freedom Flight Simulator with Real-Time Arduino Hardware in Loop Simulation

Adin Kojic, '25

Submitted to the
Department of Mechanical and Aerospace Engineering
Princeton University
in partial fulfillment of the requirements of
Undergraduate Independent Work.

Final Report

May 23, 2013

Ryne Beeson
Luigi Martinelli
MAE 442
70 pages
File Copy

This thesis represents my own work in accordance with University regulations.

Abstract

A 6 Degree-of-Freedom (6DoF) Flight Simulator in Python was made, leveraging open source libraries to build a simulator capable of running in real-time and offline modes. This simulator, named Python Laptop Air Vehicles (PLAV) is capable of piloted control and has a Hardware-In-The-Loop (HITL) proof-of-concept mode implemented with an Arduino-compatible micro controller. The project is open-source and prioritizes simplicity, with the intent that amateur aircraft designers can use it to test their unique designs with their own flight dynamics model and simulate their flight control with HITL simulation.

The simulator has been validated using the NASA Engineering and Safety Center's Check-cases for Verification of Six-Degree-of-Freedom Flight Vehicle Simulations, ensuring that for a good Flight Dynamics Model (FDM) the simulation gives accurate results. The relevance of implementing the a rotating ellipsoidal Earth is also analyzed in amateur contexts such as high power rocketry. The code is published at `https://github.com/adinkojic/PLAV`.

Acknowledgements

Thank you to Al Gaillard and Glenn Northey of the MAE Machine Shop, who provided expert advice, training, and assistance in the manufacture of related air vehicles.

I'd like commend my sister for her help with the BRGR hardware, as she was great moral support and an excellent metal polisher.

I am especially grateful towards NASA's Balloon Program Office, who gave me and my team incredible opportunities for research and learning with the FLOATing DRAGON challenge that BRGR emerged from.

I could not be more grateful towards Professor Daniel Marlow of Physics, who generously provided for the Balloon Research Glider Recovery project all those years. We'll get it flying soon.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Mission and Inspiration

This paper describes the process of building the 6 Degree-of-Freedom (6DoF) flight simulator Python Laptop Air Vehicles (PLAV) from the ground up. Emphasis is on simulating flights of amateur drones, where particular freedom is given to the flight dynamics implementation. Another objective of the project is to make the simulator and software as easy-to-use as possible for any amateur aircraft designer with basic programming experience. The code is published at `https://github.com/adinkojic/PLAV`.

The original purpose is for the Ballon Research Glider Recovery (BRGR) Project, which involves a glide vehicle with all-moving grid fins. The original requirements for BRGR included tight space constraints and a high-altitude drop, which resulted in a rotating stow-able wing and grid fins for high drag in the transonic regime. The wing and fuselage could be analyzed with OpenVSP[8]. Analysis of the grid fins was performed with a wind tunnel, for which the data was nondimensionalized and made into a lookup table. As such, the simulator needed to be highly customizable, leveraging a mixture of linear aerodynamics and lookup tables.

1.2 Simulation Objectives

Flight simulation is an important part of validating aircraft and subsystem performance, going beyond the classic state-space systems used for stability analysis and handling characteristics. With the increasing popularity of unmanned systems and

decreasing costs to manufacture, aircraft design is becoming more accessible at the amateur level. Compared to established organizations like those within NASA, these amateurs do not have significant experience and flight heritage. This coupled with fewer resources means that developing a custom simulator or leveraging commercial tools can be a significant obstacle, stifling innovation and learning. To help solve this problem, PLAV has been developed with amateur aircraft designers in mind.

PLAV attempts to implement a customizable flight simulator in Python with a focus on simplicity for the user without sacrificing precision. This allows users to implement experimental vehicles quickly and easily. The primary goal is to provide valid results with a 6DoF simulation with real-time and Hardware-in-the-Loop (HITL) capabilities. It must display relevant state and environmental values to the user to guide aircraft analysis and autopilot development. For this reason, the UI focuses on displaying data rather than a photorealistic view, with the main script being executed from the command line and the results displayed largely in time history data.

The real-time simulation keeps in sync with the system clock and updates the plots accordingly. It is made to allow other programs or systems more closely resembling flight hardware to take control. At most realistic level is HITL simulation, which runs the flight software on the actual flight hardware. This is achieved in the proof-of-concept by integration with Arduino micro controllers to send data across a USB port. A pause/play button and on-screen joystick widget are included as well for manual control, with future plans to implement an interface to video game controllers.

The main purpose of the project is to develop a reasonably accurate 6DoF simulator for verification the flight controller providing Guidance, Navigation and Control (GNC) for the BRGR Project. This project is an example of an amateur aircraft project that requires special considerations for its mix of conventional and nonconventional systems. This vehicle is expected to fly dozens of miles after being dropped from a balloon in the stratosphere, involving significant changes in temperature, aircraft performance, and even gravity. This simulator attempts to model this to at least the World Geodetic System 1984 (WGS84) ellipsoid [20].

1.3 Existing Flight Simulators

A variety of simulation tools have been produced by various organizations, with multiple developed within NASA. Tools like the Langley Standard Real-Time Simulation in C++ (LaSRS++), Marshall Aerospace Vehicle Representation in C (MAVERIC), have been developed to provide a framework and accelerate the simulation of different

projects [15]. These are large projects with significant resources and history behind them. Unfortunately, these simulators are not available for recreational use. All of these models are closed-source and possibly governed by export controls.

As for open-source projects, JSBSim is a simulator written in C++ with interfaces to other programming languages [12]. It is used in flight simulators such as FlightGear as well as a few research applications. The C++ programming language produces highly efficient and fast code, but development is much slower and more difficult compared to a modern high-level programming language like Python. This paper attempts to explore the feasibility of a Python simulation running on a modern laptop computer and provide background on flight simulation.

1.4 Simulator Validation Methodology

This simulator was validated using the NASA Engineering and Safety Center's (NESC) check cases provided in [16]. This paper compared five NASA simulators and along with JSBSim. These scenarios verify gravity, translational and rotational Equations of Motion (EOM), inertial coupling, atmosphere models, Earth's curvature, the Coriolis effect, and air data calculations. The vehicle models used are a sphere, a brick, and an F-16 fighter jet aircraft.

Check cases 1-3 and 6-13 were implemented and tested, all of which produced acceptable results reasonably close to the provided check cases. These cases tested simulations on the World Geodetic System 1984 (WGS84) ellipsoid model of the Earth, with cases ranging from a dropped dragless sphere to autopilot commands on the F-16. Comparison plots of relevant results such as position and forces were made, which validate the PLAV simulator.

Chapter 2

6 Degree-of-Freedom Simulation

2.1 Dynamics Review

2.1.1 Quaternions and Implementation

The simulator must keep track of the orientation of the vehicle relative to some reference. Commonly used are the Euler angles, which describe the orientation as three rotations in series [17]. The most common order in aircraft is yaw-pitch-roll, where a yaw rotation around the upwards vertical is applied, followed by a pitch along the aircraft's +y (starboard) axis, and finally a roll rotation around the nose of the aircraft. However, this method suffers from a singularity called "gimbal lock", a +/-90 degree pitch angle results in an ambiguity between what is a roll rotation or a yaw rotation.

Quaternions do not have this ambiguity. Instead, rotation is represented as a vector rotation axis and a scalar rotation angle [6]. The convention for quaternion $\mathbf{q}$ adopted by this simulator is one where the first (real) component of the quaternion represents the scalar (rotation angle) component, and the next 3 components represent the vector (rotation vector) component (Equation 2.1.1). ϑ represents the rotation angle in radians and $\mathbf{e}$ is the axis of rotation in Cartesian space, as shown in 2.1.1.

$$\mathbf{q}(\mathbf{e}, \vartheta) = \begin{bmatrix} \cos(\vartheta/2) \\ \mathbf{e}\sin(\vartheta/2) \end{bmatrix} = \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{bmatrix} \tag{2.1.1}$$

Combining rotations is as simple as multiplying the quaternions according to their defined algebraic rules [11]. Quaternions can also be expressed in the form described in Equation 2.1.2. $\hat{\mathbf{i}}$ is the imaginary number defined by $\hat{\mathbf{i}}^2 = -1$. $\hat{\mathbf{j}}$ and $\hat{\mathbf{k}}$ also share a similar definition, both being equal to -1 when squared. These symbols are not equal and multiplications between them are defined by Equations 2.1.3 and 2.1.4. That is, they are not commutative. These rules are shown in Table 2.1.

$$\mathbf{q}(\mathbf{e}, \vartheta) = \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{bmatrix} = q_1 + q_2\hat{\mathbf{i}} + q_3\hat{\mathbf{j}} + q_3\hat{\mathbf{k}} \tag{2.1.2}$$

$$\hat{\mathbf{i}}\,\hat{\mathbf{j}} = \hat{\mathbf{k}} \tag{2.1.3}$$

$$\hat{\mathbf{j}}\,\hat{\mathbf{i}} = -\hat{\mathbf{k}} \tag{2.1.4}$$

y × x	$\mathbf{1}$	$\hat{\mathbf{i}}$	$\hat{\mathbf{j}}$	$\hat{\mathbf{k}}$
$\mathbf{1}$	$\mathbf{1}$	$\hat{\mathbf{i}}$	$\hat{\mathbf{j}}$	$\hat{\mathbf{k}}$
$\hat{\mathbf{i}}$	$\hat{\mathbf{i}}$	$-\mathbf{1}$	$\hat{\mathbf{k}}$	$-\hat{\mathbf{j}}$
$\hat{\mathbf{j}}$	$\hat{\mathbf{j}}$	$-\hat{\mathbf{k}}$	$-\mathbf{1}$	$\hat{\mathbf{i}}$
$\hat{\mathbf{k}}$	$\hat{\mathbf{k}}$	$\hat{\mathbf{j}}$	$-\hat{\mathbf{i}}$	$-\mathbf{1}$

Table 2.1: Quaternion Multiplication Table

Quaternions can be converted back to Euler angles using equations 2.1.5 to 2.1.7 [3]. In this context, ϕ represents the roll/bank angle of the aircraft, θ represents the pitch angle, and ψ represents the yaw/bearing angle of the aircraft. These values are more intuitive to use for control and navigation control laws. $\arctan 2(\cdot, \cdot)$ is the four quadrant arc tangent function, which returns the angle made by the 2D vector and the x-axis. The classic $\arctan(\cdot)$ function has a smaller range and does not distinguish between $(-x, -y)$ and (x, y).

$$\phi = \arctan 2\left(2(q_1q_2 + q_3q_4), (1 - 2(q_2^2q_3^2))\right) \tag{2.1.5}$$

$$\theta = 2 * \arctan 2\left(\sqrt{1 + 2(q_1q_3 - q_2q_4)}, \sqrt{1 - 2(q_1q_3 - q_2q_4}\right) - \pi/2 \tag{2.1.6}$$

$$\psi = \arctan 2 \left(2(q_1 q_4 + q_2 q_3), (1 - 2(q_3^2 q_4^2))\right) \tag{2.1.7}$$

2.1.2 Reference Frame Rotations

In addition to tracking orientation, the quaternion is used to perform a change of basis between different frames of reference. The literature describes a quaternion representation of the attitude matrix $A_N^B(\mathbf{q})$ [6], which is used to calculate the frame rotation of a vector. This simulator implements the Equation 2.1.8 to calculate the frame rotation of a vector [2], which is equivalent. $\mathbf{v}'$ is the vector in the new frame, $\mathbf{v}$ is the original vector, s, r, and m are the scalar, vector components and magnitude of the quaternion, respectively. Note that $\times$ represents the vector cross product.

$$\mathbf{v}' = \mathbf{v} + 2r \times (s\mathbf{v} + \mathbf{r} \times \mathbf{v})/m \tag{2.1.8}$$

The quaternion is integrated over time using body rate angles using Equation 2.1.9. $\Xi(\mathbf{q})$ is defined in 2.1.10 [3]. Note that some literature uses q_4 as the scalar component rather than q_1. $\Xi(\mathbf{q})$ must be adjusted accordingly.

$$\dot{\mathbf{q}} = \frac{1}{2}\Xi(\mathbf{q})\omega \tag{2.1.9}$$

$$\Xi(\mathbf{q}) = \frac{1}{2}\begin{bmatrix} -q_2 & -q_3 & -q_4 \\ q_1 & -q_4 & q_3 \\ q_4 & q_1 & -q_2 \\ -q_3 & q_2 & q_1 \end{bmatrix} \tag{2.1.10}$$

2.1.3 Forces and Moments in 3D

Modeling aerodynamic forces and moments is critical to an accurate simulation. The simulation must consider the frame in which they are calculated and how to rotate frames to be usable in the equations of motion. The flight dynamics model describes forces in the body frame of the aircraft, which rotates relative to the local North-East-Down (NED) frame. Accelerations in the body frame are calculated by Newton's second law using the forces in the body frame ${}^B\mathbf{F}$ and vehicle mass m (Equation 2.1.11), and must then be rotated to be used in Equation of Motion 2.2.11. This rotation is implemented with 2.1.8.

$$^{B}\mathbf{a} =^{B} \mathbf{F}/m \tag{2.1.11}$$

Angular acceleration calculations do not need to be rotated in this manner as the rotation rate and orientation are tracked in the body frame relative to the NED frame.

2.2 Equations of Motion

Estimating the state is a very important part of the flight control and autopilot system. To support this, the native coordinate system of the simulator is based on the Inertial Navigation System (INS) equations provided in [11] with angular acceleration from [17]. Compared to simulating in the Earth-Centered Earth-Fixed (ECEF) Cartesian coordinates, where the center of the Earth is the origin, this system requires fewer change of basis and produces much more intuitive results. The state vector is defined as $\mathbf{x}$ in Equation 2.2.1.

$$\mathbf{x} \equiv \begin{bmatrix} \mathbf{q} \\ \omega_{B/I}^{B} \\ \lambda \\ \phi \\ h \\ \mathbf{V}_{NED} \end{bmatrix} \tag{2.2.1}$$

λ, ϕ are the vehicle longitude and latitude, respectively. h represents the altitude and $\mathbf{V}_{NED}$ is the velocity vector in the North-East-Down reference frame with components v_N, v_E, v_D. $\mathbf{q}$ is the quaternion that describes the rotation from the body frame to the NED frame, where the identity quaternion describes a level aircraft oriented facing north. It should be noted that the change in $\dot{v}_N$, $\dot{v}_E$, and $\dot{v}_D$ includes the effects of the Earth's roundness, rotation, and Coriolis effects in addition to the acceleration and gravity terms. Accounting for these phenomena are required for precise implementation of an INS system and for modeling flights over long distances or high altitudes. An analysis of the importance of these terms for amateur rocketry is explored in a later section.

$$\dot{\mathbf{q}} = \frac{1}{2}\Xi(\mathbf{q})\omega_{B}^{B/N} \tag{2.2.2}$$

$$\omega_{B/N}^{B} = \omega_{B/I}^{B} - A_{N}^{B}(\mathbf{q})\omega_{N/I}^{N} \tag{2.2.3}$$

$$\dot{\omega}_{B/I}^{B} = \mathbb{I}^{-1}(\mathbf{M}_B - \omega_{B/I}^{B} \times \mathbb{I} \times \omega_{B/I}^{B}) \tag{2.2.4}$$

$$\dot{\phi} = \frac{v_N}{R_\phi + h} \tag{2.2.5}$$

$$\dot{\lambda} = \frac{v_E}{(R_\lambda + h)\cos\phi} \tag{2.2.6}$$

$$\dot{h} = -v_D \tag{2.2.7}$$

$$\dot{v}_N = -\left[\frac{v_E}{(R_\lambda + h)\cos\phi} + 2\omega_e\right] v_E \sin\phi + \frac{v_N v_D}{R_\phi + h} + a_N \tag{2.2.8}$$

$$\dot{v}_E = \left[\frac{v_E}{(R_\lambda + h)\cos\phi} + 2\omega_e\right] v_N \sin\phi + \frac{v_E v_D}{R_\phi + h} + 2\omega_e v_D \cos\phi + a_E \tag{2.2.9}$$

$$\dot{v}_D = -\frac{v_E^2}{R_\lambda + h} - \frac{v_N^2}{R_\phi + h} - 2\omega_e v_E \cos\phi + g + a_D \tag{2.2.10}$$

$${}^{N}\mathbf{a} = \begin{bmatrix} a_N \\ a_E \\ a_D \end{bmatrix} = A_{B}^{N}(\mathbf{q})\,{}^{B}\mathbf{a} \tag{2.2.11}$$

Reading longitude, latitude, and altitude measurements is intuitive compared to ECEF Cartesian coordinates. Quaternions are used because they do not have singularity/gimbal lock effects at +/-90 degrees of pitch. Although Euler angles are considered more intuitive by some, quaternions are more straightforward to use with regard to implementing sensor data. Euler angles are still calculated for autopilot and display purposes. The values $\omega_{B/I}^{B}$ and ${}^{B}\mathbf{a}$, representing the body rotation rates and body acceleration, can be used for software and hardware simulations of Kalman filters [26], with the option to apply noise to simulate a realistic Inertial Measurement Unit (IMU).

The terms $\mathbf{M}_B$and $\mathbb{I}$ refer to the moments and moment of inertia of the vehicle.

The term $\omega^B_{B/N}$ describes the rotation rate of the aircraft relative to the non-inertial frame of the Earth, which must have the $\omega^N_{N/I}$ term subtracted from it (Equation 2.2.12). $\omega^N_{N/I}$ describes the rotation of the NED frame to an inertial frame. ω_e describes the Earth's rotation rate, defined as $7.292\,115 \times 10^{-5}\,\text{rad/s}$ [20]. $\omega^B_{B/I}$ describes the body rate relative to an inertial frame and is what an idealized on-board gyroscope would measure.

$$\boldsymbol{\omega}^N_{N/I} = \omega_e \begin{bmatrix} \cos\phi \\ 0 \\ -\sin\phi \end{bmatrix} + \begin{bmatrix} \dfrac{v_E}{R_\lambda + h} \\ -\dfrac{v_N}{R_\phi + h} \\ -\dfrac{v_E \tan\phi}{R_\lambda + h} \end{bmatrix} \tag{2.2.12}$$

Gravity, in meters per second per second, is calculated using the J_2 harmonic model [20] at the given latitude. The commonly used value near $9.806\,65\,\text{m/s}^2$ is not completely accurate everywhere [20]. An analysis of simulations modeling and not modeling this term properly is shown later.

$$\begin{aligned} g = {} & 9.780327\left(1 + 5.3024 \times 10^{-3}\sin^2\phi - 5.8 \times 10^{-6}\sin^2 2\phi\right) \\ & - \left(3.0877 \times 10^{-6} - 4.4 \times 10^{-9}\sin^2\phi\right)h + 7.2 \times 10^{-14}h^2 \end{aligned} \tag{2.2.13}$$

The radius of curvature R_ϕ and length normal to the ellipsoid R_λ are defined in Equations 2.2.14, 2.2.15 [20].

$$R_\phi = \frac{a(1-e^2)}{(1 - e^2\sin^2\phi)^{3/2}} \tag{2.2.14}$$

$$R_\lambda = \frac{a}{(1 - e^2\sin^2\phi)^{1/2}} \tag{2.2.15}$$

Additionally, WGS84 defines the following constants in Table 2.2 [20]:

Constant	Meaning	Definition	Units
e	Earth Eccentricity	0.0818	n.d.
a	Earth Semi-major Axis	6,378,137.0	m
ω_e	Earth Rotation Rate	7.292115e-5	rad/s

Table 2.2: Selected WGS84 Constant Definitions

2.3 Atmosphere Model

An atmosphere model is required to provide the values for air density, temperature, and pressure for the simulation across varying altitudes. The US Standard Atmosphere 1976 (USSA1976) is most commonly used for research, as it provides a consistent environment to run tests and simulations while being reasonable for evaluating vehicle performance [19]. It is implemented in all NESC check cases and PLAV.

USSA1976 defines a piecewise linear interpolation of temperature over altitude (Table 2.3) based on empirical data. Pressure is then solved for using closed-form solutions of the hydrostatic pressure equation (2.3.1) depending on the temperature lapse (change over altitude). If there is a nonzero temperature lapse locally, then Equation 2.3.2 is used, and if there is no temperature lapse then Equation 2.3.3 is used. Note that gravity is held constant despite changes in altitude.

b	Altitude [km] H_b	Temperature [K] $T_{M,b}$	Temperature Lapse [K/km] $L_{M,b}$
0	0	288.15	-6.5
1	11	216.65	0.0
2	20	216.65	+1.0
3	32	228.65	+2.8
4	47	270.65	0.0
5	51	270.65	-2.8
6	71	214.65	-2.0
7	84.8520	-	-

Table 2.3: USSA1976 Temperature Table.

The model defines P_0 to be 101.325 kPa and R_s to be 287.052 874 J/kg/K. P_b is calculated from the appropriate equation when $H = H_{b-1}$. Actual implementations such as this simulator use tabulated values for P_b and $T_{M,b}$ and determine the appropriate value of b, while others interpolate a table of the entire atmosphere.

$$\frac{\partial P}{\partial z} = -g\rho \tag{2.3.1}$$

$$P = P_b \left[\frac{T_{M,b}}{T_{M,b} + L_{M,b}(H - H_b)} \right]^{\frac{g_0}{R_s L_{M,b}}} \tag{2.3.2}$$

$$P = P_b \exp\left[\frac{-g_0(H - H_b)}{R_s \; T_{M,b}} \right] \tag{2.3.3}$$

Air density is calculated from the ideal gas Equation 2.3.4.

$$\rho = \frac{P}{R_s T} \tag{2.3.4}$$

Dynamic viscosity is defined using 2.3.5. β is $1.458 \times 10^{-6}\,\mathrm{kg/s/m/K^{0.5}}$ and S is $110.4\,\mathrm{K}$ [19].

$$\mu = \frac{\beta T^{3/2}}{T + S} \tag{2.3.5}$$

The USSA1976 model does not represent any one given day and does not include wind. For the purposes of passing the NESC check cases a static linear wind model piecewise linear model is added. Future work on the atmosphere model includes the ability to input weather balloon data to interpolate wind, pressure, and temperature at a given altitude.

2.4 Aircraft Flight Dynamics Model

An aircraft Flight Dynamics Model (FDM) describes the forces and moments an aircraft experiences at a given flight condition and control deflection. In this simulator, the flight dynamics model is an object which receives environmental data, flight conditions, and control inputs to calculate the moments and forces on the aircraft. It also can output the flight conditions and intermediate values it calculates, such as Mach number and angle-of-attack.

In PLAV, an "AircraftConfig" object is made, which represents the "plant" of the system. It has methods to update the vehicle conditions for altitude, velocity, body rate, air density, temperature, and speed of sound. These values are used to calculate dynamic pressure, angle of attack, sideslip, dimensionless body rate, Reynolds number, and Mach number. These values are used to determine the appropriate coefficients and forces which are returned in the body frame. It has inputs in terms of trimmed control values and deflection values, which can be optionally updated.

2.4.1 Intermediate Values

Airspeed is the magnitude of the wind velocity vector. The wind velocity vector is the velocity through the air, including both wind and ground speed. Its components are described in 2.4.1, where positive u is wind in the pilots face, positive v is wind in the pilots right ear, and positive w is wind blowing up the pilot's skirt.

The angle of attack α represents the angle the incoming wind makes with the

horizontal datum plane of the aircraft (Equation 2.4.3). A positive angle of attack feels like wind is hitting under the pilot's chin. Similarly, the angle of slide slip β is the angle the incoming wind makes with the plane's plane of symmetry (Equation 2.4.4) [25]. A positive β is feels like wind in the pilot's left ear. Note that if airspeed is zero the β calculation does not make sense.

$$\mathbf{V} = \begin{bmatrix} u \\ v \\ w \end{bmatrix} \tag{2.4.1}$$

$$V = |\mathbf{V}| \tag{2.4.2}$$

$$\alpha = \arctan 2(w, u) \tag{2.4.3}$$

$$\beta = \arcsin(v/V) \tag{2.4.4}$$

Aircraft dynamics also include rotation rate damping terms. Rotation rates about the x, y, and z axis (p, q, r) are nondimensionalized as shown in Equations 2.4.5 to 2.4.7. Note that these values do not make sense when airspeed is zero.

$$\hat{p} = \frac{bp}{2V} \tag{2.4.5}$$

$$\hat{q} = \frac{\bar{c}q}{2V} \tag{2.4.6}$$

$$\hat{r} = \frac{br}{2V} \tag{2.4.7}$$

2.4.2 Basic Aerodynamics

The generic aerodynamic model depends on calculating the dimensionless coefficients for the given flight regime and then using those to determine the forces and moments on the aircraft. A review of basic aerodynamics is provided, with further details in [3] and [24].

For a 3 Degree-of-Freedom simulation, the relevant aerodynamic equations for small angles of attack are described in 2.4.9 to 2.4.10. The values of C_{L_0}, C_{L_α}, C_{D_0}, ε, and C_{m_0} are calculated from other software and treated as inputs. C_{m_α} is a consequence of the lift and drag vectors and the torque from the displacement of the

center of pressure to the center of mass.

$$C_L = C_{L_0} + \alpha C_{L_\alpha} \tag{2.4.8}$$

$$C_D = C_{D_0} + \varepsilon C_L^2 \tag{2.4.9}$$

$$C_m = C_{m_0} + C_{m_\alpha}\alpha + C_{m_{\hat{q}}}\hat{q} \tag{2.4.10}$$

Lift and Drag are aligned with the wind axis, where Drag points opposite the wind vector and lift points up perpendicular to it and the starboard axis. For this reason a rotation is performed to get the forces in the direction of the aircraft X and Y axis. For a 6DoF simulation, this rotation must also take into account sideslip. This is achieved by rotating along the vertical axis and then the pitch axis. A_W^B describes the frame rotation from the wind axis to body axis.

The moment is a sum of the pure couples and torque resulting from the distance between the aerodynamic center and center of mass. The relation between forces in the body frame and wind frame is shown, where wind frame values are reported as Lift and Drag. The wind axis has its x-axis defined by the incoming wind vector. $\mathbf{F}_{other}$ describes other forces implemented by the user, such as the unconventional grid fins in the BRGR scenario as a control surface.

The calculation for the forces and moments from the coefficients is shown below. The lift and drag coefficients are calculated with dimensions and any additional force (such as thrust) is added. Pure couples are calculated separately and then the torque from forces is added (Equation 2.4.11). $\mathbf{x}_{cp}$ represents the displacement of the center of pressure from the center of mass (Equation 2.4.12). Derivatives such as C_{m_α}, which describe the change in pitching moment due to angle of attack α, are not made redundant by this calculation. This is covered more generally by using the torque instead. This matches the NESC F-16 simulation models and check cases accurately, for which an effective $C_{m\alpha}$ is a result.

$$\mathbf{F}_B = \begin{bmatrix} X_B \\ Y_B \\ Z_B \end{bmatrix} = \bar{q}S \begin{bmatrix} C_X \\ C_Y \\ C_Z \end{bmatrix} + \mathbf{F}_{other} = A_W^B \begin{bmatrix} Lift \\ Drag \\ Side \end{bmatrix} + \mathbf{F}_{other} = A_W^B \bar{q}S \begin{bmatrix} C_{Lift} \\ C_{Drag} \\ C_{Side} \end{bmatrix} + \mathbf{F}_{other} \tag{2.4.11}$$

$$\mathbf{M}_B = \begin{bmatrix} L_B \\ M_B \\ N_B \end{bmatrix} = \bar{q}S \begin{bmatrix} C_L \\ C_M \\ C_N \end{bmatrix} + \mathbf{x}_{cp} \times \mathbf{F}_B \tag{2.4.12}$$

2.4.3 A Note on Nonlinear Aerodynamics

More accurate models, such as the F-16 aircraft in [1], are not entirely implemented with linear relations. Use of lookup tables is common, where for an input value of α or control surface deflection tabulated values from simulations or wind tunnels are used. These are still provided in non-dimensional coefficient form and might resemble linear relations, but may be implemented in body coordinates rather than wind axis.

Chapter 3

Python Implementation

3.1 Software Used

3.1.1 Python

Python is an increasingly popular programming language favored for its rapid development speed. It is an interpreted high-level language that is open-source [23]. The ease with which programs can be written combined with an incredibly set of easy to install and use open source libraries makes Python the first choice for any project.

However, this enhanced development speed comes at the cost of program speed. Python is usually 3-5 times slower than Java and 5-10 times slower than C++ for an equivalent program [7]. Both the development and program execution speed vary by application and implementation. The software developed for PLAV is all Python, although different techniques and software are used to provide significant speed improvements.

3.1.2 Numba

Numba is a library for Python that can take the interpreted code of some Python functions and classes and make it into just-in-time (JIT) compiled code [14]. With a simple `@jit` decorator applied to the beginning of a function, Numba automatically attempts to convert the Python code to JIT code, which after compilation runs significantly faster. Take, for example, the body frame velocity to airspeed, angle of attack (alpha), and side slip (beta) angle calculation function.

```
@jit(float64[:](float64[:]))
def velocity_to_alpha_beta(velocity_body):
    """Gets velocity to alpha beta, assumes x direction is datum
    Reference: Fundamentals of Helicopter Dyanmics 10.42,10.43"""
    airspeed = math.sqrt(velocity_body[0]**2+velocity_body[1]**2\
        +velocity_body[2]**2)

    if abs(airspeed) > 0.01:
        beta = math.asin(velocity_body[1]/airspeed)
    else:
        beta = 0.0
    alpha = math.atan2(velocity_body[2], velocity_body[0])
    return np.array([airspeed, alpha, beta], 'd')
```

For a fair comparison, the function is run with a 10000 random values, and the execution time and compile time are recorded in Table 3.1. Running many times is required to reduce effects such as cache misses and other nonideal realities of a modern computer. The difference in pure Python NumPy and the default Math library implementation is likely due to NumPy also being able to handle complex values, generating overhead. These differences are not present once compiled with Numba.

"Lazy" and "Eager" compilation refer to telling the Numba which data types to expect as inputs and outputs of the function. "Lazy" compilation is only using the `@jit` decorator, which does not affect execution time but does dramatically affect compilation time. "Eager" compilation tells the compiler what data types to expect but requires more effort from the programmer. This leads to dramatically faster compilation time, approaching the first time run of pure Python.

A nearly two-fold improvement in runtime is seen versus the standard math library. This required nothing more than placing the `@jit` decorator above the function declaration after importing Numba. Compared to integrating with other languages with Python (such as Boost.Python), this library is simple and easy to use.

However, Numba is not fully mature. Lazy compilation times can become relatively long while the improvement does not bring it all the way to the speeds of Java or C++. Improving compilation time means telling the compiler the data types, approaching the effort required to write a Java or C++ program. In particular, the `@jitclass` decorator, which can turn classes into compiled code, results in much longer compilation times and very strange and difficult to debug errors. Compiled code and pure Python code behave differently, with bugs and errors that only occur in one version. In particular, an array out-of-bounds indexing for compiled code often returns an incorrect result instead causing an exception.

Test	First Run Time [μs]	Execute Time[μs]
Pure Python (NumPy)	101.6	2.9015
Pure Python (Math)	104.6	1.3663
Numba (Lazy)	493,456	0.8184
Numba (Eager)	228.5	0.8319

Table 3.1: Numba Use Example.

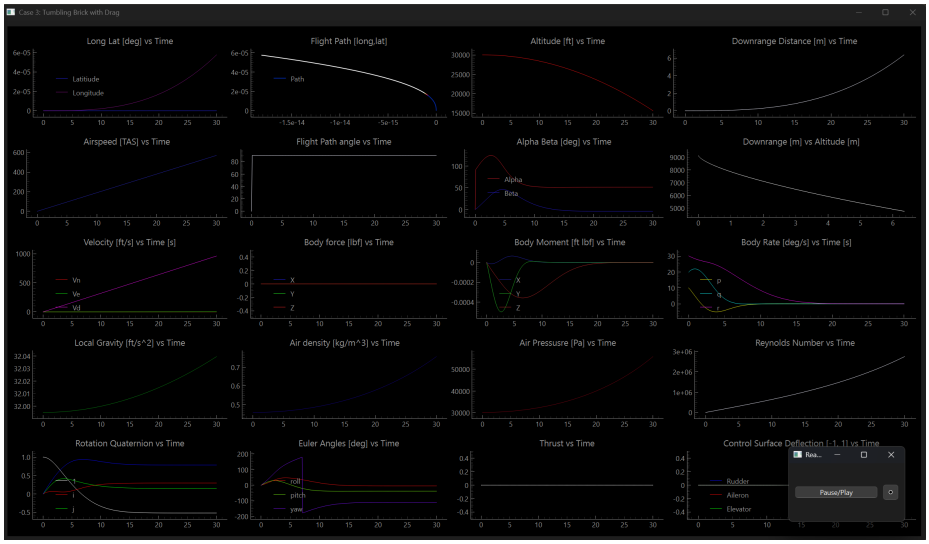


Figure 3.1: PLAV Software Interface

Libraries

Additional libraries were used to provide other functionality, with core libraries being entirely open source. NumPy was particularly useful for it's array computation abilities [9], with PyQtGraph and PyQt6 used for plotting the data [4]. Compared to MatPlotLib [10], PyQtGraph is faster and allows for easier updating of data for real-time simulations. The PyQtGraph-based interface is shown in Figure 3.1

3.2 Software Implementation

3.2.1 Architecture

PLAV is meant to be easily user-modifiable and run in real-time simulation, so the aircraft flight dynamics model control input is directly accessible by the pilot/flight hardware. To provide a single canonical source of data, a logger object is the source of all data used by the different components. The Simulator is made to update the

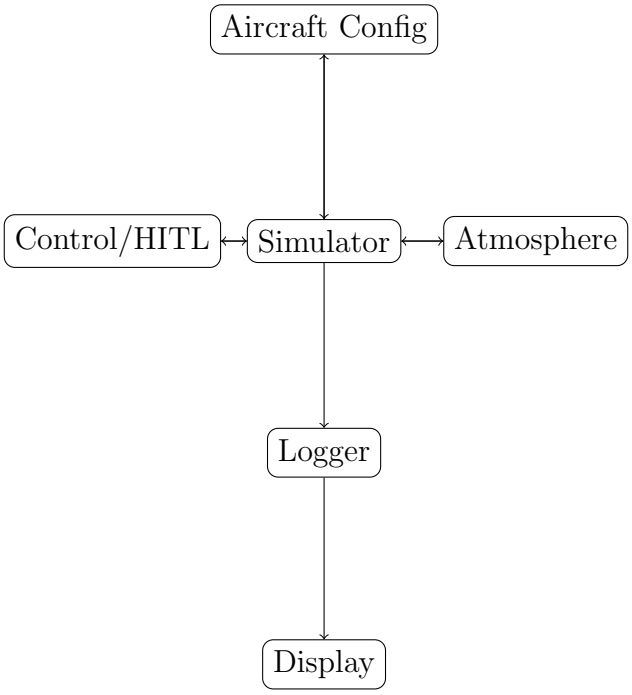


Figure 3.2: PLAV Architecure

flight environment and conditions of the AircraftConfig object with every iteration. The display node represents the time history plotting features that display the data stored by the logger. A graphical description of the architecture is shown in Figure 3.2.

The Simulator object handles all control of the simulation, as it passes along information about the state to the control system, then the control input into the aircraft plant, solves the timestep, and saves the data to the logger object. It has methods to return the logger object, which is then used for the time history display component.

PLAV in its current iteration is meant to support only a single air vehicle on a single thread, and only a single state vector is saved in the Simulator class. However, there can be multiple Simulator classes instantiated, each with a different (or not) aircraft configuration. There is no protocol for interaction at this time. The AircraftConfig object does not store the aircraft's position, but it does store the last set value of the control surfaces and trim settings. Future iterations may see this moved to the Simulator class to facilitate multiple in-flight aircraft at a time.

3.2.2 Aircraft Implementation

The FDM is made to be easy for the user to make dramatic changes to, and so the simulator itself allows any file with the required methods to be used. As such, the exact implementation can vary dramatically. This affords flexibly to implement any model, such as the F-16 fighter jet model described in [1].

Implementing aircraft can be done in two ways: either by using the AircraftConfig class with a JSON file or implementing a custom class with methods compatible with the simulator. Both examples are implemented, one for the F-16 included in the check cases and a linear aerodynamics class.

AircraftConfig Class

The AircraftConfig class is meant to allow rapid and easy implementations of custom aircraft from a JSON file containing various aerodynamic derivatives, from which the actual aerodynamic coefficients are calculated from linear approximations. This class also saves the certain properties of the aircraft on instantiation, namely mass, the inertia tensor, geometric properties like the reference area, mean chord length, and span, as well as the displacement of the aerodynamic center with respect to the center of mass. It also includes an initial control vector to be used as trim.

As part computing coefficients, this class calculates the following values, which can be returned with included methods: angle of attack, angle of sideslip, mach number, dynamic pressure, true airspeed, and Reynolds number. Note that these values were calculated from the last use of the `update_conditions()` method and may contain stale or unrealized data. Proper instructions will be available in the Github repository.

Custom Classes

The AircraftConfig class may not model nonlinear dynamics or model all required phenomena. The simulator is designed with custom or modded implementations in mind, and so allows for fully custom flight dynamics models. One such model implemented is the F-16 model from [1]. This model includes nonlinear aerodynamics, lookup tables, and other interesting behavior. These values are hard-coded in this implementation as it was only meant to be used with the check cases in [16]. There is a soft requirement that the model be compatible as a Numba `@jitclass`, otherwise the speed improvements from compilation will be canceled.

Any aircraft class must include an `update_conditions()` method to be called before the `get_forces()` and `calculate_thrust()` methods. If the aircraft includes control surfaces or engines, the `update_control()` method is used to change those values. Control is not updated between or during intermediate timesteps. In addition, it must have `get_mass()` and `get_inertia_matrix()` to return the relevant values in SI units. Additional methods are used for logging purposes and return intermediate calculations, such as the angle of attack.

The files provided in [1] are shown in the DAVE-ML standard, which encodes aircraft functions and lookup tables in an XML format [18]. The advantage of this setup is that it enables easy cross-compatibility of the aircraft model between simulators. However, it is tedious to hand code into this format and it is expected that amateur aircraft designers would be more comfortable with a straightforward Python program. For the purposes of building this flight simulator the F16_aero, -_inertia, -_prop, and -_control models were hand coded into a Python class (`f16_model.py`), complete with lookup tables and linear functions. Future work may include a DAVE-ML translator script to automatically make the dynamics model. Unfortunately, the Numba library does not yet properly support inheritance for compiled classes, so for model developers will need to make sure the method names match.

3.2.3 Control Systems Implementation

Unlike, for example, a continuous state-space system with no delay, real digital control systems include a signal processing and control computation delay. For this reason, control is fixed during a Runge-Kutta timestep and is only updated between steps. The control communications were implemented with HITL system in mind, and so the control system is completely separate from the aircraft class and is not directly involved in the `x_dot()` function. At the end of a timestep the `control_sys_update()` method is immediately called, which gives the control system the very latest data. For a HITL arrangement, this gives the hardware under test the time to make computations before `control_sys_request_response()` is called, at which point the computations must be complete. During that time, the simulator may update plots before the next timestep is solved. This is otherwise inconsequential for the single-thread simulated control.

This arrangement is designed to allow the hardware or software under test to respond to the system but inherently limits the response rate to the simulation frame rate. For fast frame rates this becomes irrelevant as the frame time approaches the

sensor read time, although unlike an offline simulation the real-time simulation must keep up with the hardware. Currently, for a timestep of 0.1 s the offline simulation runs at 18x speed, suggesting that the real-time simulation could be run at as high as 1800 frames per second. At this speed the latency of the USB protocol to the hardware under test will likely become a limiting factor.

3.3 Implementing Atmosphere Models

The atmospheric model implemented is the USSA1976 model with linear piecewise winds. Environmental parameters are derived from calls to USSA1976 functions, which calculate the parameters from the nonlinear equation starting tabulated values from lowest change in temperature lapse below the current altitude. Winds are read from the JSON file and were implemented for the NASA check cases. It also includes an `update_conditions()` method followed by getter functions. The Atmosphere class was designed such that in future iterations other models can be implemented, such as one constructed from (possibly live) NOAA weather balloon data.

3.3.1 Integration Method

PLAV uses the fixed-timestep Runge-Kutta 4th order method to solve the equations of motion. This allowed a very direct implementation of the equations of motion, which are contained in a large `x_dot()` function. The function takes the current time, current state, aircraft object, and atmosphere model as inputs. Optionally, it can take in a logger object, although this should not be input when using the Runge-Kutta solver. A fixed timestep is required due to fast changes in state near the poles and longitudinal wraparound.

Due to the nature of the Runge-Kutta method, `x_dot()` is called many times with non-realized inputs. For this reason, data is not logged in the Runge-Kutta method or from the aircraft directly, but by calling `x_dot()` at the current timestep with the aircraft, atmosphere model, and logger object. Admittedly, this extra function call is wasteful and redundant with the first calculation in the Runge-Kutta method. Future work includes eliminating this extra call. Note that unrealized states are held onto by the aircraft object, although the control is input separately and is the same for all Runge-Kutta steps.

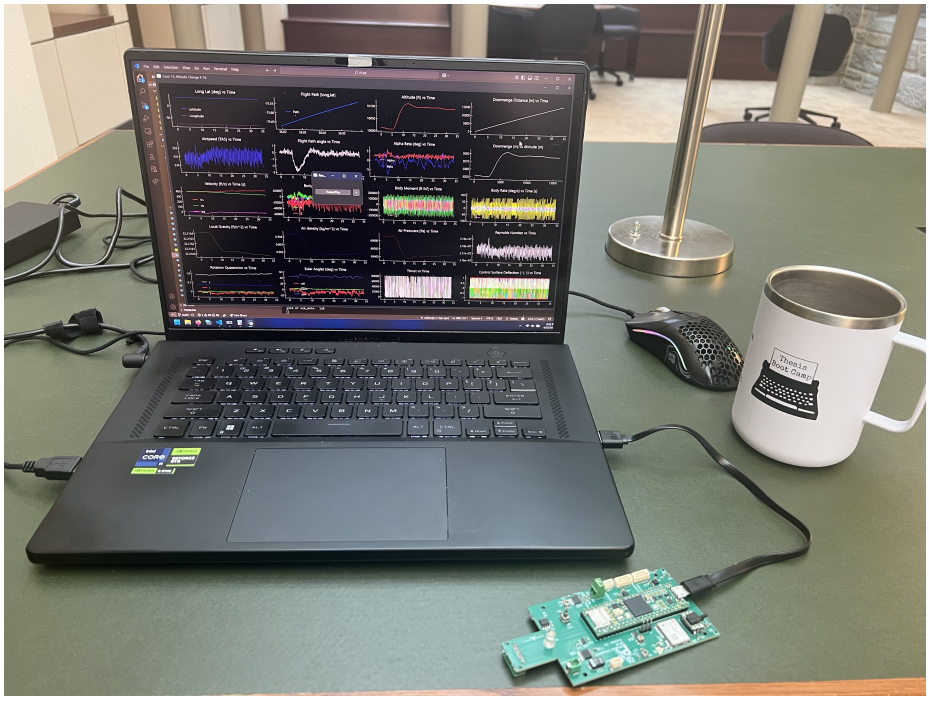


Figure 3.3: HITL Arrangement with Teensy 4.1

3.4 Hardware

This simulator is meant to run on easily accessible compute hardware, namely laptops. The libraries used to run the simulator are fully compatible with Windows, Linux, and Windows. For this paper, the simulation hardware used is a laptop running Windows 11 24H2 with an Intel x84-64 i9-13900H CPU. It is expected that the simulator will function fine on lower-end hardware.

For the Hardware-in-the-Loop (HITL) component, an Arduino-compatible micro controller is used, in this case a Teensy 4.1 [22]. This device runs at a 600 MHz core clock speed and uses the high-speed USB 2.0 mode (up to 480 Mbit/s). This is a somewhat high-end micro controller with a true USB implementation, unlike the Arduino Uno which uses a serial converter at a much slower speed. Controllers like the ESP32 do offer a true USB implementation while being very low cost, which may see a future implementation. The arrangement is shown in Figure 3.3. The HITL feature is in the proof-of-concept stage at this time and exhibits jittery behavior.

3.5 Porting Code to Hardware

The F16_control.dml file was implemented in Python to run locally and C++ to run on the micro controller for HITL testing. The C++ coding experience was much less enjoyable and was slower than the Python effort, referring back to the purpose of implementing a simulator in Python. Both arrangements used a Python object to be passed into the Simulator, although the HITL variant updated the micro controller and requested responses over the USB port. Note that sensors or actuation is not simulated.

The HITL simulation feature is in the proof-of-concept phase, where a quick and dirty communication protocol was made between the host and hardware. In particular, the host could send a packet containing a and data or it could issue a header that would induce a response from the hardware. The header was a sequence of three + characters followed by one specific character to specify the packet type. Available packets to send were environmental update packets and control update packets. Available responses to receive are the control actuation and a debug-only state dump. 32 bits/4 bytes were allocated to each floating point value, which was sent as $(2^{31}/10000)^{-1} = 4.656 \cdot 10^{-5}$ units per Least Significant Bit (LSB). This inherently limits any value to around 21474. This was done to make the packets as small as possible, as the default Arduino Serial receive buffer is 64 bytes. Future work includes dramatically increasing packet size and including error checking. Implementing these features might require handshaking or a fleshed-out communication protocol.

Chapter 4

Simulation Comparison Results

4.1 Comparison Methods

The output of the PLAV simulator was compared with the check cases in [16]. The difference and variation between PLAV and each simulator was plotted for values of interest, interpolating where necessary. Variation plots were made according to Equation 4.1.1, where $x_{s,i}$ is the check case simulation and $x_{p,i}$ are the results from PLAV. Note that variation plot can be misleading around zero crossings (appears as a vertical asymptote) and very large nominal values.

$$\delta_i = \frac{x_{s,i} - x_{p,i}}{x_{p,i}} \tag{4.1.1}$$

4.2 Selected Data

The original paper describing the check cases is almost 400 pages of figures [16]. This paper also contains many pages of figures in the appendix, although certain plots of note are included in this section for discussion.

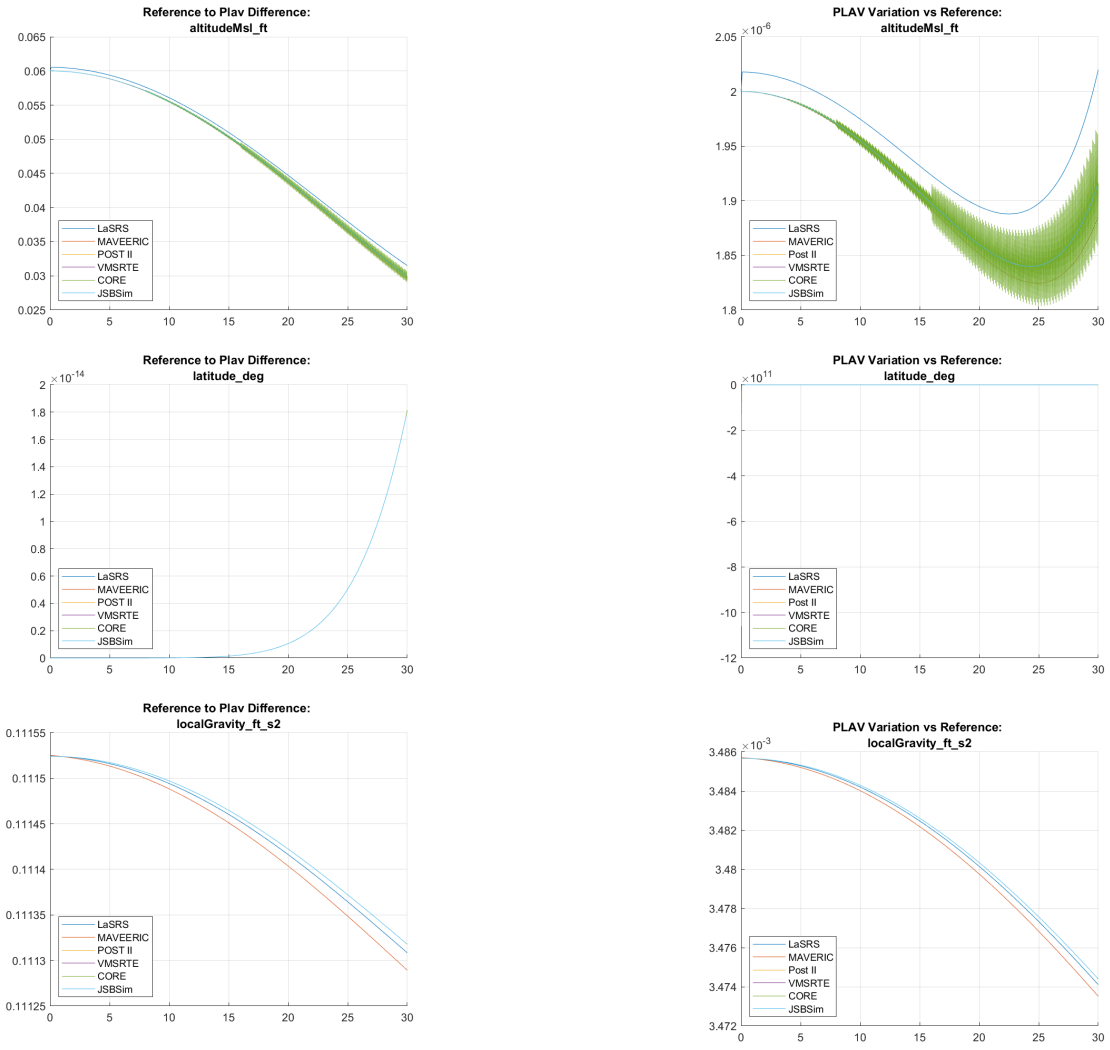


Figure 4.1: Case 1 Selected Results

4.2.1 Check Case 1: Dragless Sphere

This check case attempts to verify the translational equations of motion without any forces. It includes effects from Earth's rotation and the J_2 gravity model. The variation of PLAV with all the simulators as shown is minimal, suggesting accurate results in Figure 4.1. It is noted that the initial conditions are slightly off, possibly due to a conversion error. This is likely responsible for the gravity error as well.

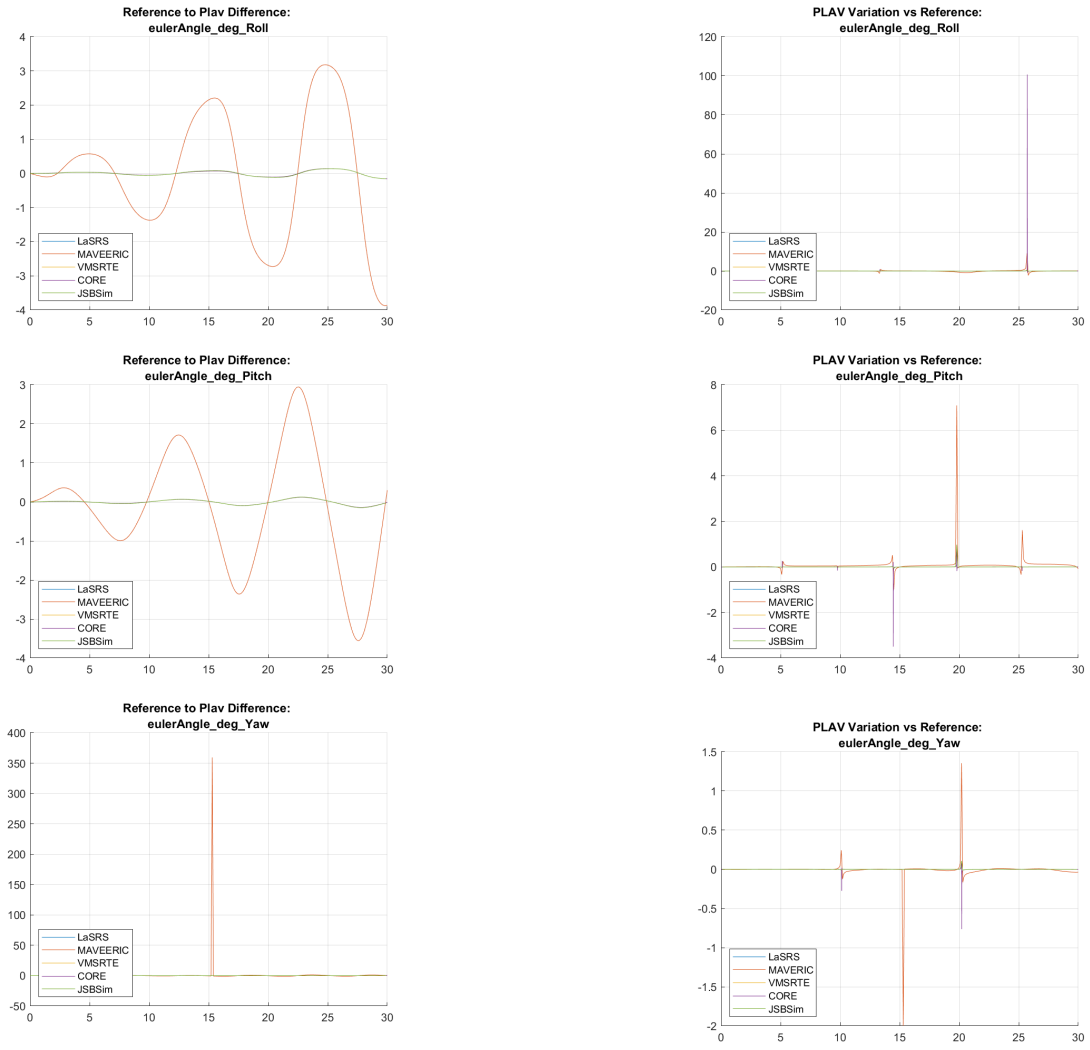


Figure 4.2: Case 2 Selected Results

4.2.2 Check Case 2: Dragless Brick

This check case attempts to verify the rotational equations of motion without any forces. For this reason, euler angles are shown in Figure 4.2. It is otherwise the same as the previous model. It includes all the effects of the previous model. The variation of PLAV with all the simulators as shown is minimal, suggesting accurate results. In fact, PLAV's error's are smaller than the NASA MAVERIC reference model. Note that the spikes in the variation plots are a consequence of zero crossings.

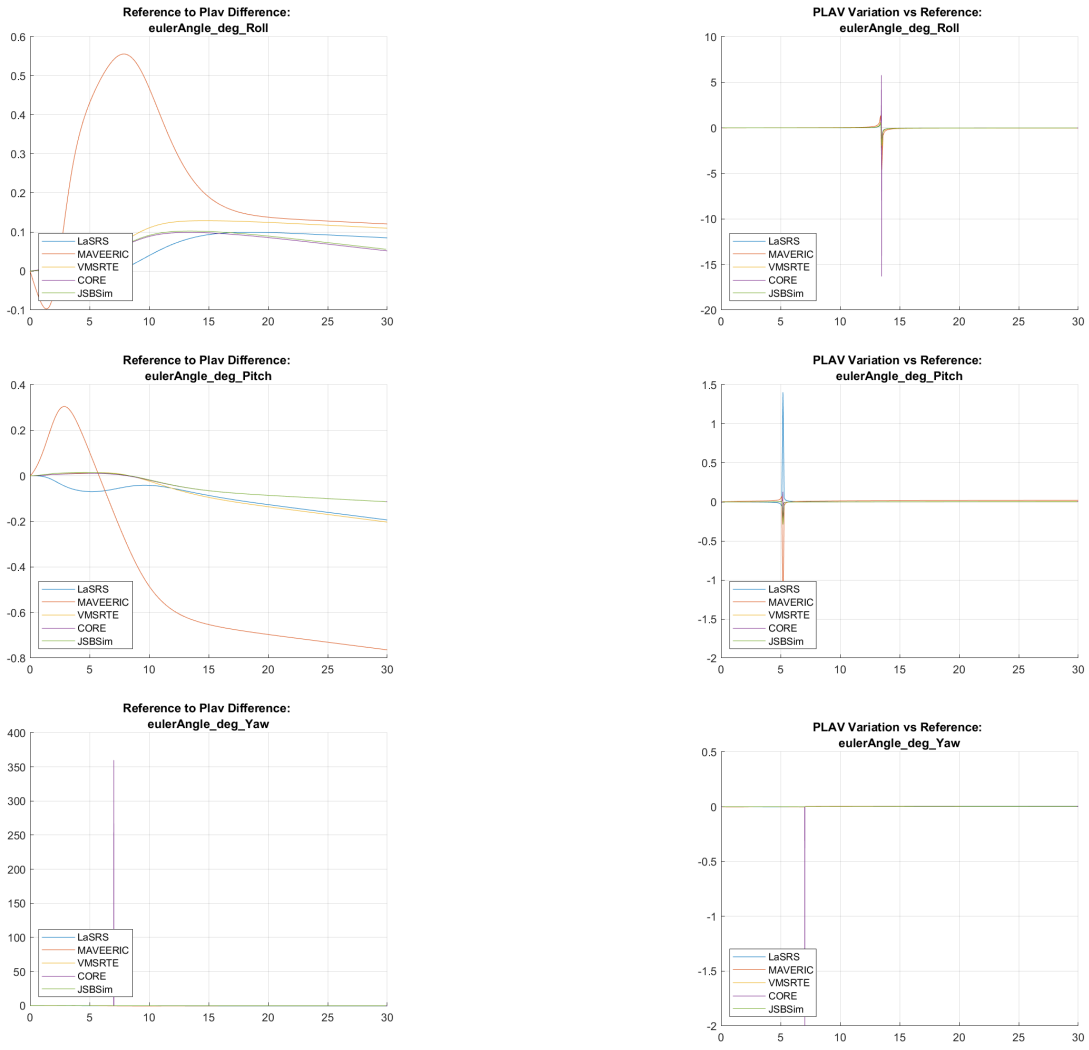


Figure 4.3: Case 3 Selected Results

4.2.3 Check Case 3: Brick with Aerodynamic Damping

This check case attempts to verify moments and simple aerodynamic calculations such as nondimensionalized rotation rates. For this reason, euler angles are shown in Figure 4.3. It is otherwise the same as the previous model. The difference across all simulators small but approaching integer degrees in error. Still, the variation is small enough that results are likely acceptable. Note the spikes in the variation plots are a consequence of zero crossings.

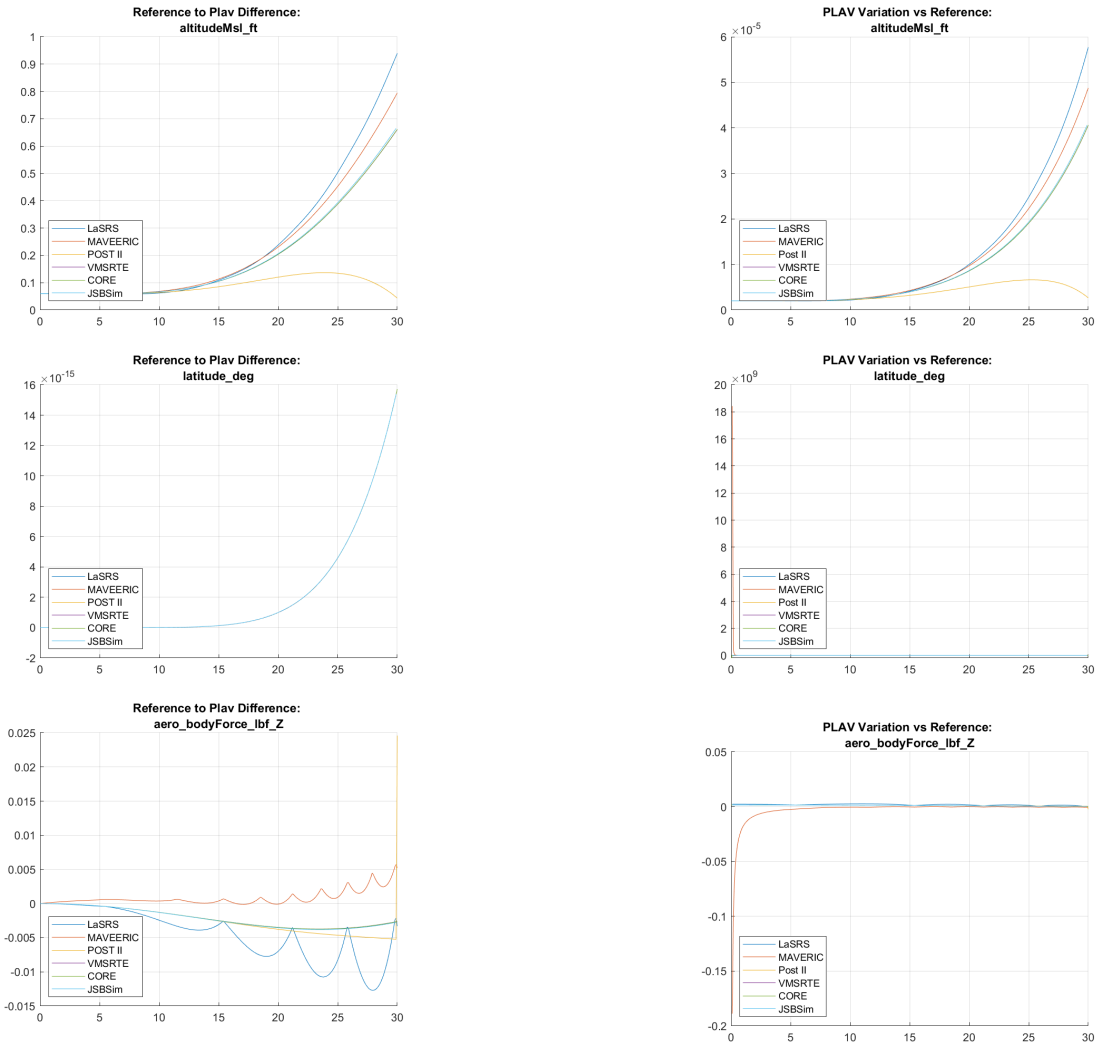


Figure 4.4: Case 6 Selected Results

4.2.4 Check Case 6: Sphere with Drag

This check case attempts to verify forces and simple aerodynamic calculations. For this reason, altitude, latitude, and downward force is shown in Figure 4.4. The difference across all simulators is noticeable but PLAV is not far from the pack. The results are acceptable.

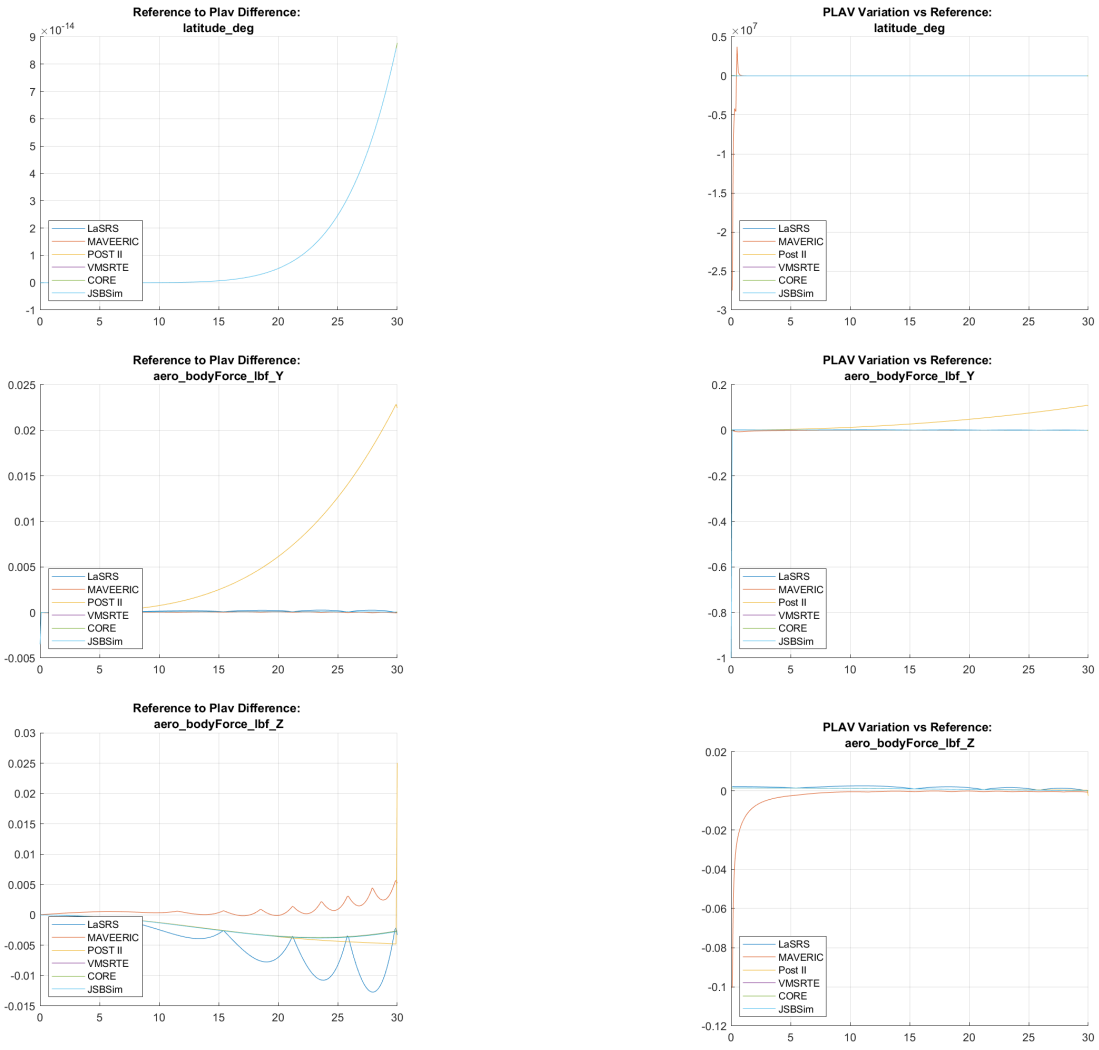


Figure 4.5: Case 7 Selected Results

4.2.5 Check Case 7: Sphere with Drag and Wind

This check case attempts check wind modeling. For this reason, the latitude and Z (down) and Y (facing wind) forces are shown in Figure 4.5. The differences are small across most simulations, verifying PLAV's ability to model constant wind.

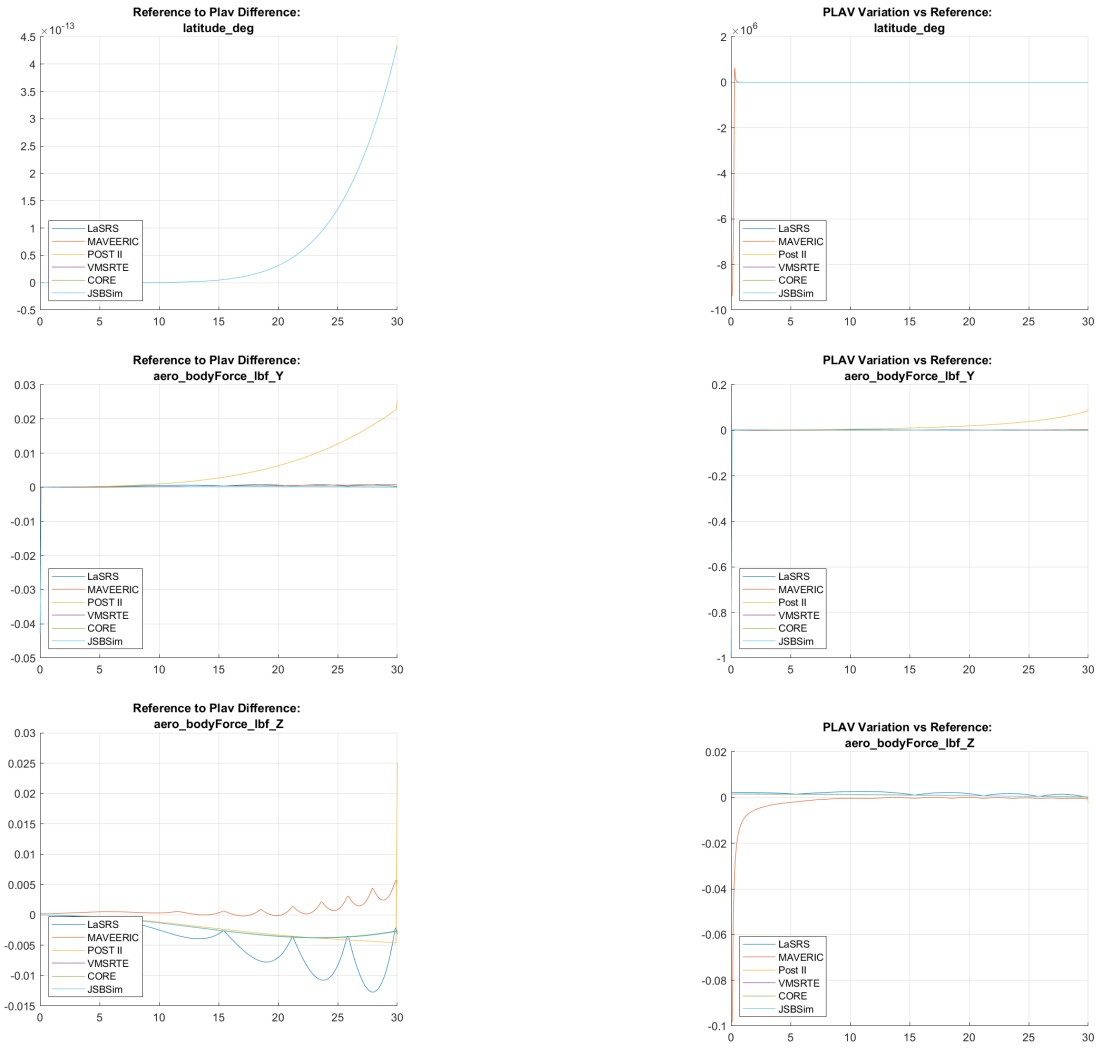


Figure 4.6: Case 8 Selected Results

4.2.6 Check Case 8: Sphere with Drag and Varying

This check case attempts check wind shear modeling. For this reason, the latitude and Z (down) and Y (facing wind) forces are shown in Figure 4.6. The differences are small across most simulations, verifying PLAV's ability to model linearly varying wind.

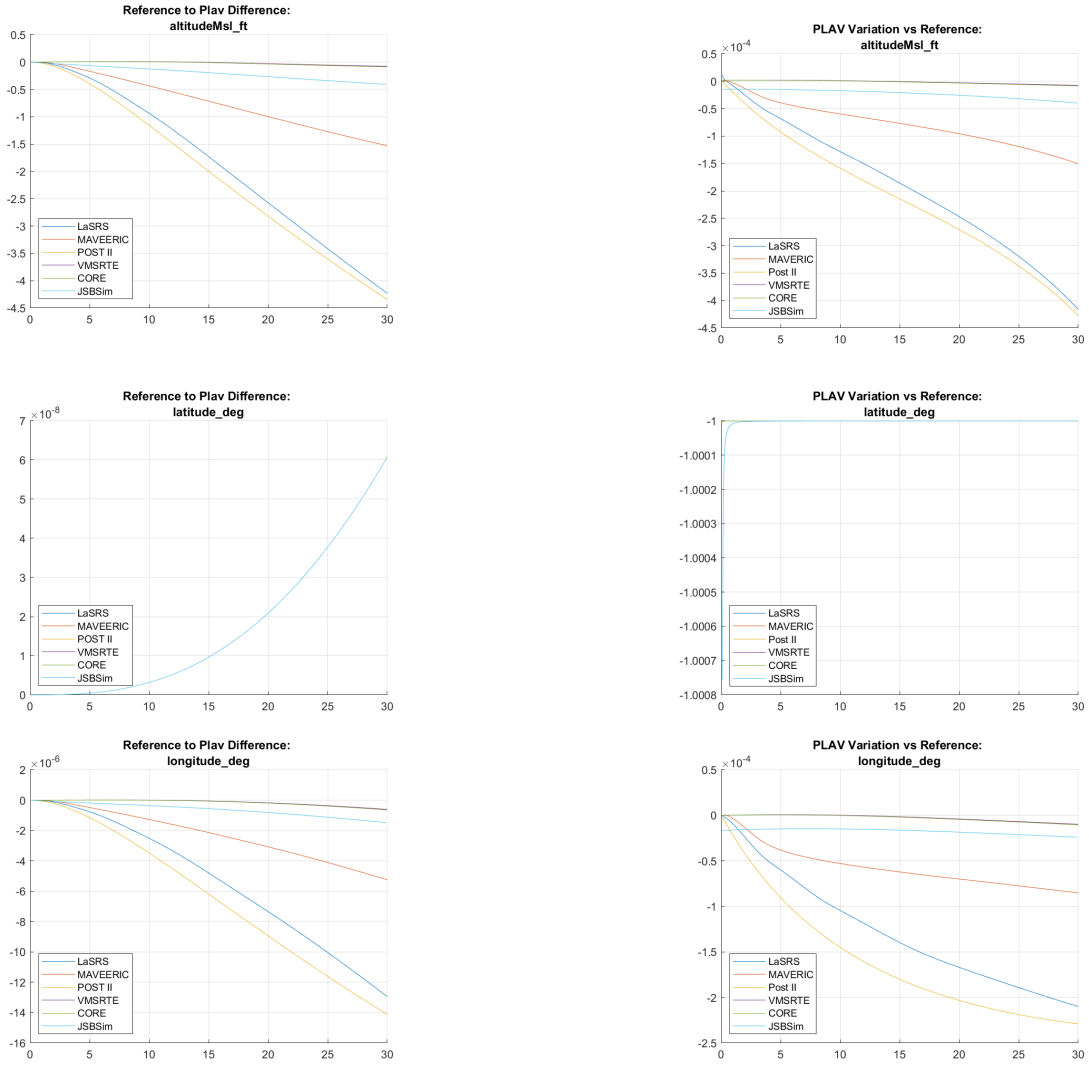


Figure 4.7: Case 9 Selected Results

4.2.7 Check Case 9: Eastward Cannonball

This check case attempts to verify translational motion without the Corilois effect. This is the first check case with significant downrange distance. It has the same small altitude difference as most cases, but small error in longitude and latitude. For this reason PLAV's downrange modeling is considered verified.

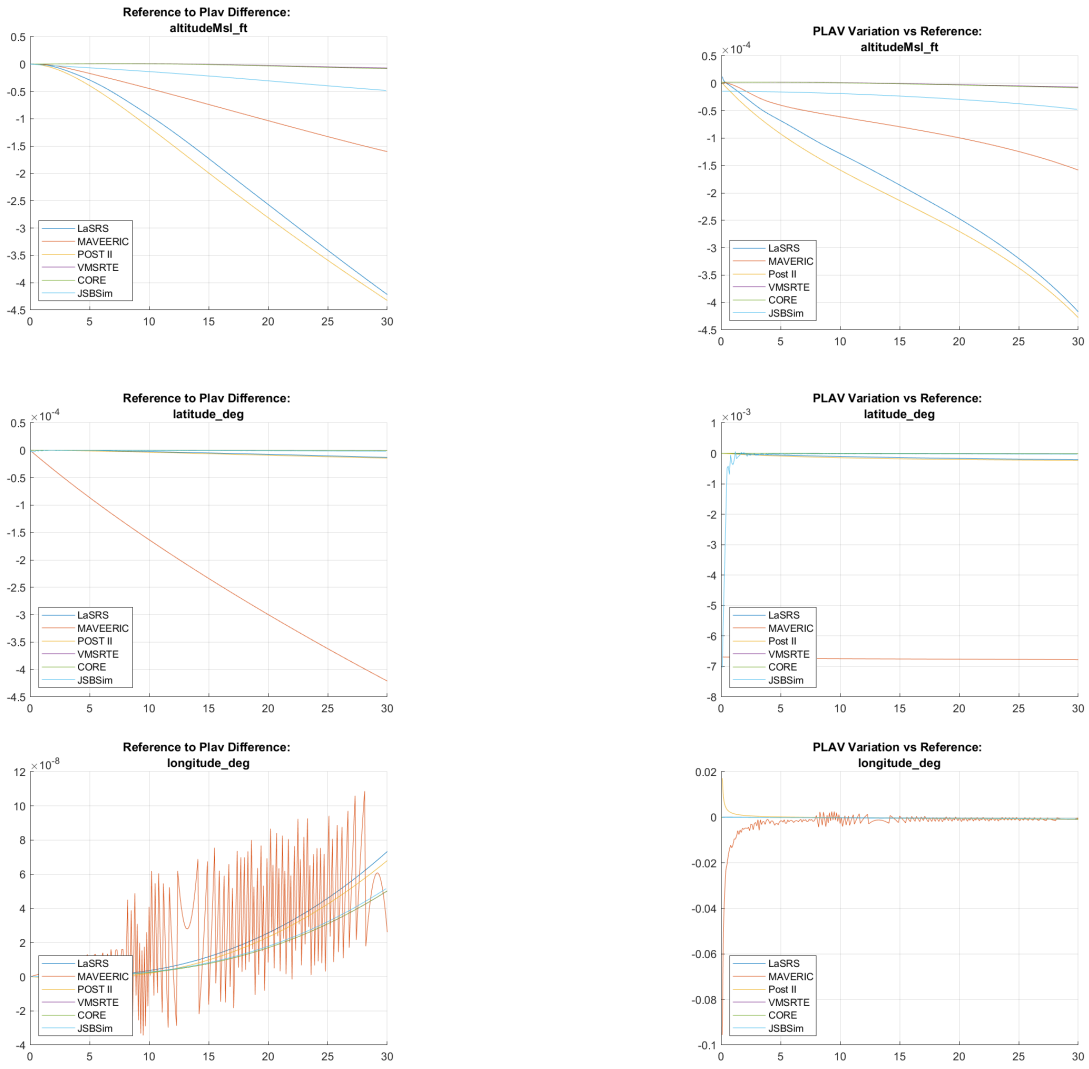


Figure 4.8: Case 10 Selected Results

4.2.8 Check Case 10: Northward Cannonball

This check case attempts to verify translational motion with the Coriolis effect. This is the first check case with significant northward distance. It has the same small altitude difference as most cases, but small error in longitude and latitude. For this reason PLAV's Coriolis modeling is considered verified.

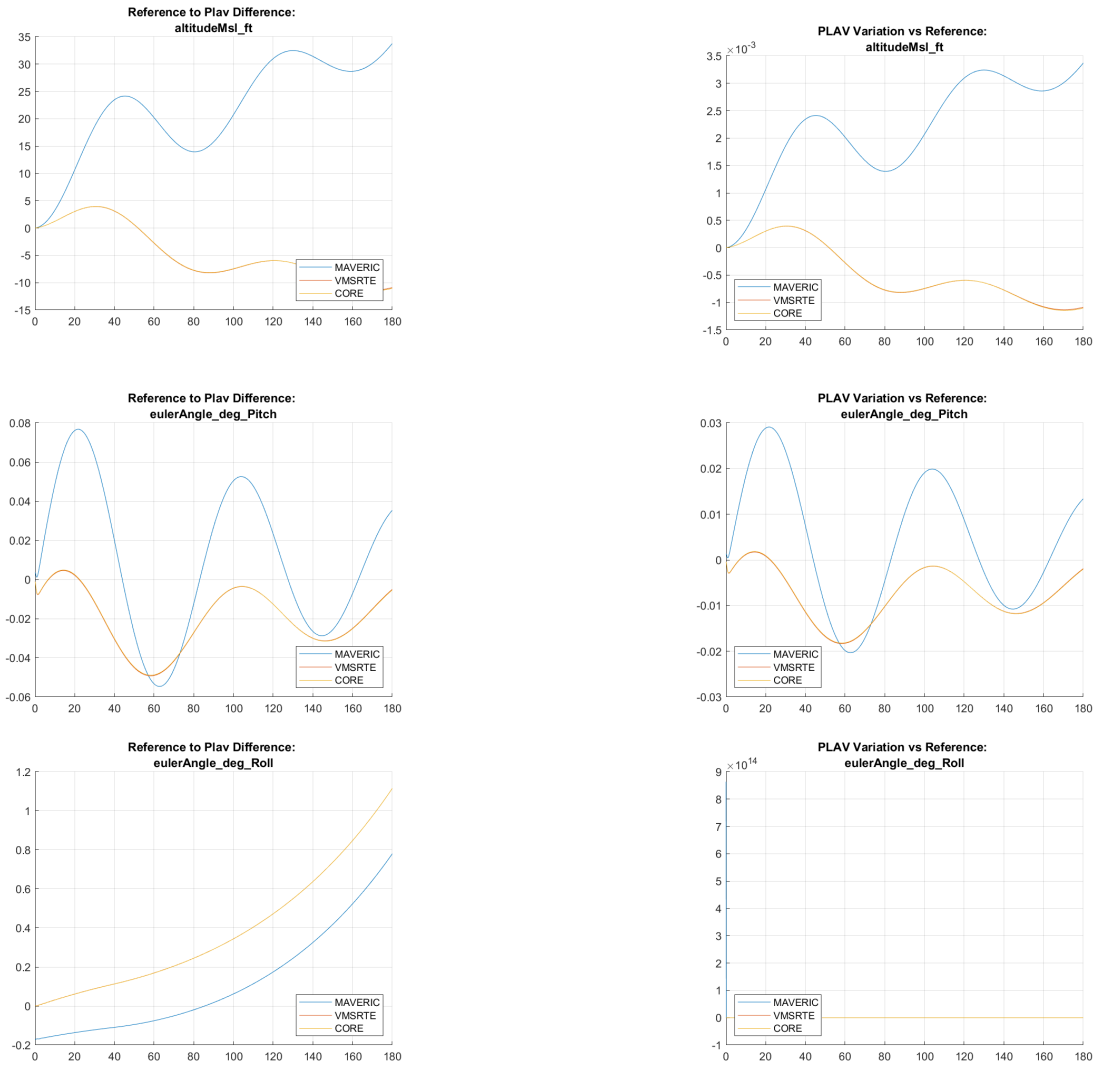


Figure 4.9: Case 11 Selected Results

4.2.9 Check Case 11: Open Loop F-16

This check case is the first to implement [1]. Unfortunately, only half of the simulators have results provided in the check case. This case also served to test the trim solver of the relevant simulator. This is not yet implemented in PLAV, so the solutions were manually trimmed to resemble the result. In addition to pitch and throttle settings it also had to trim the ailerons and elevators, for which PLAV made no attempt. Regarding the results, there is good agreement with PLAV given the trim variation and the simulator is considered validated.

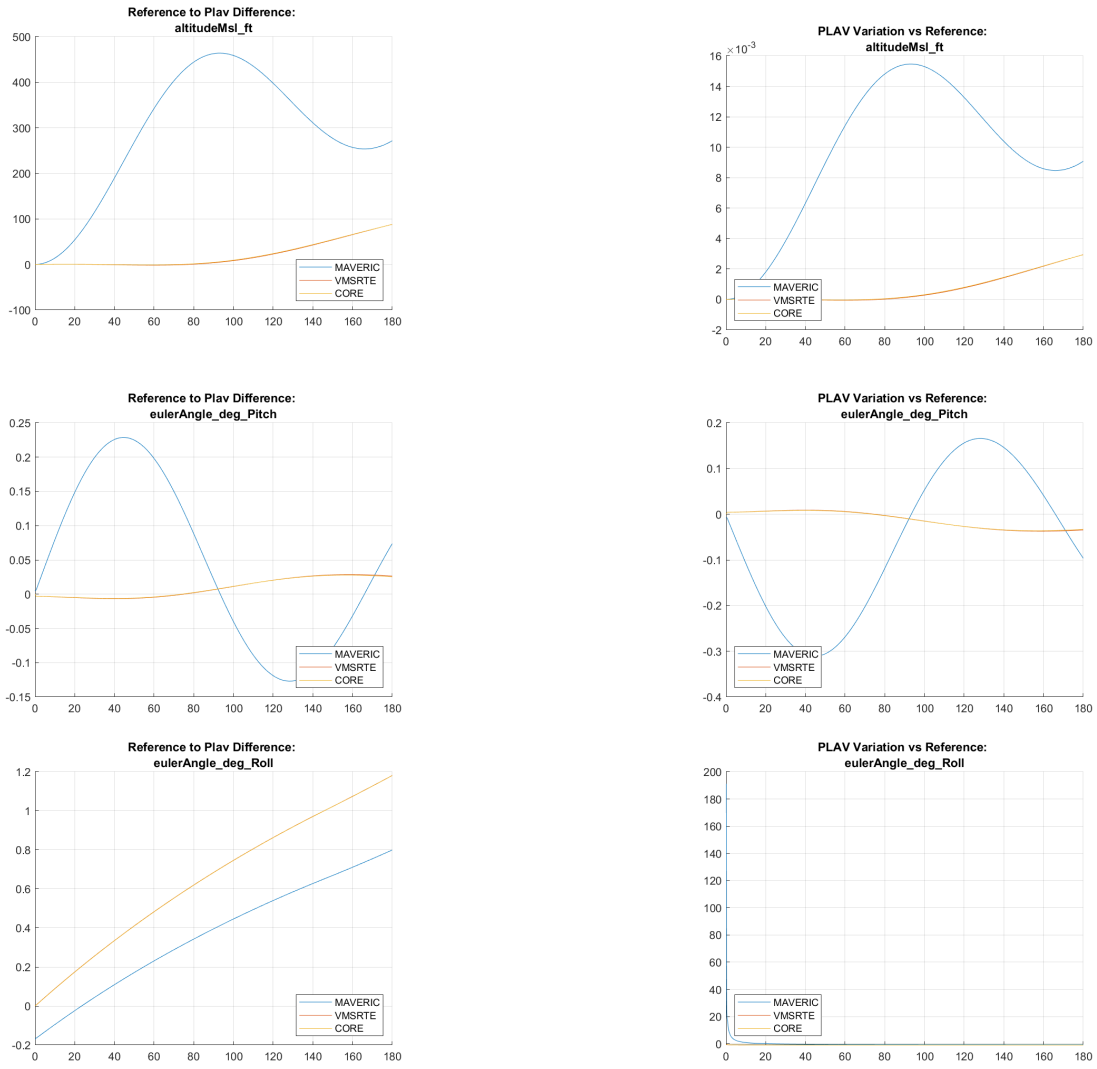


Figure 4.10: Case 12 Selected Results

4.2.10 Check Case 12: Supersonic F-16

This check case uses [1] at supersonic speeds. Once again, the trim was manually solved. Regarding the results, PLAV is in the middle of the pack and considered validated given the trim variation.

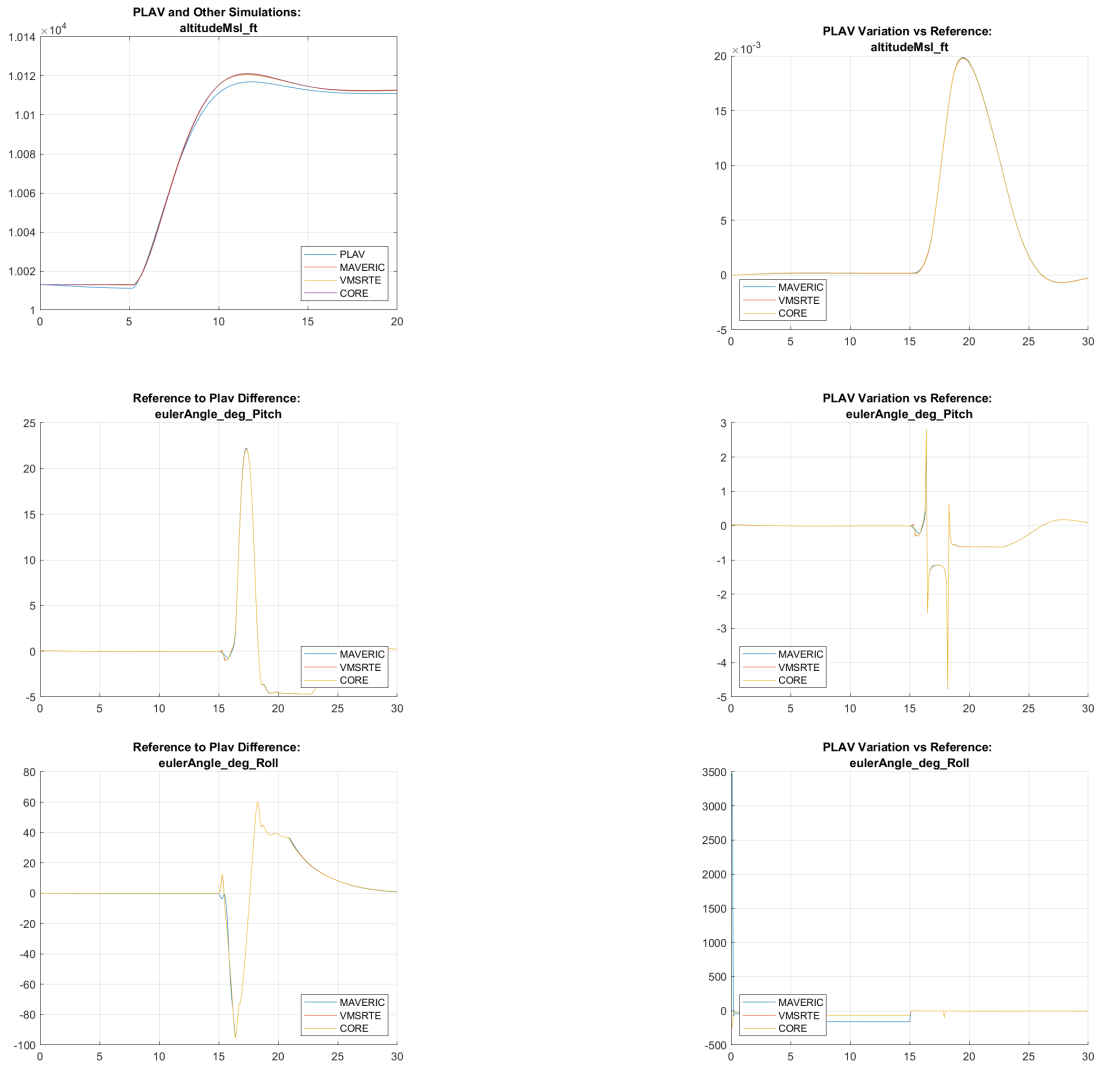


Figure 4.11: Case 13A Selected Results

4.2.11 Check Case 13A: Autopilot F-16 Altitude Increase

This check case uses the [1] with the autopilot active. At $t = 5$ seconds the autopilot is commanded to increase the altitude by 100 feet. Once again, the trim was manually solved. Regarding the results there is noticeable difference in autopilot behavior. As shown in the altitude plot of Figure 4.11, the altitude seems to be marginally less than the other simulations. This may have to do with the trim solution or air data calculations. Given the variables involved this is still considered an acceptable result for PLAV.

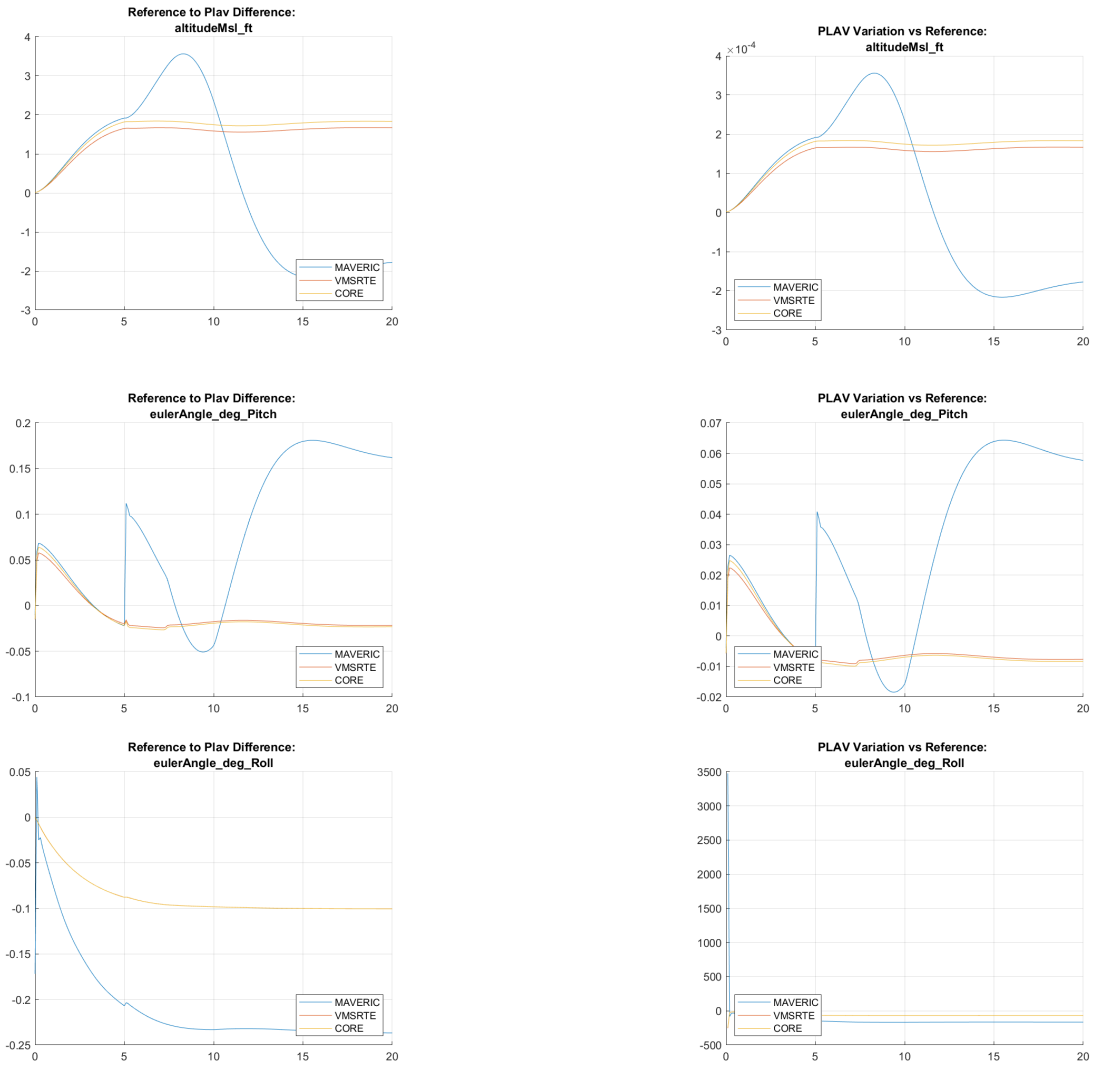


Figure 4.12: Case 13B Selected Results

4.2.12 Check Case 13B: Autopilot F-16 Speed Decrease

This check case uses [1] with the autopilot active. At $t = 5$ seconds the autopilot is commanded to decrease the speed by 5 knots. Once again, the trim was manually solved. There is a much smaller difference in autopilot behavior. As shown in the altitude plot of Figure 4.12, the values agree much more closely. This is considered a good result for PLAV.

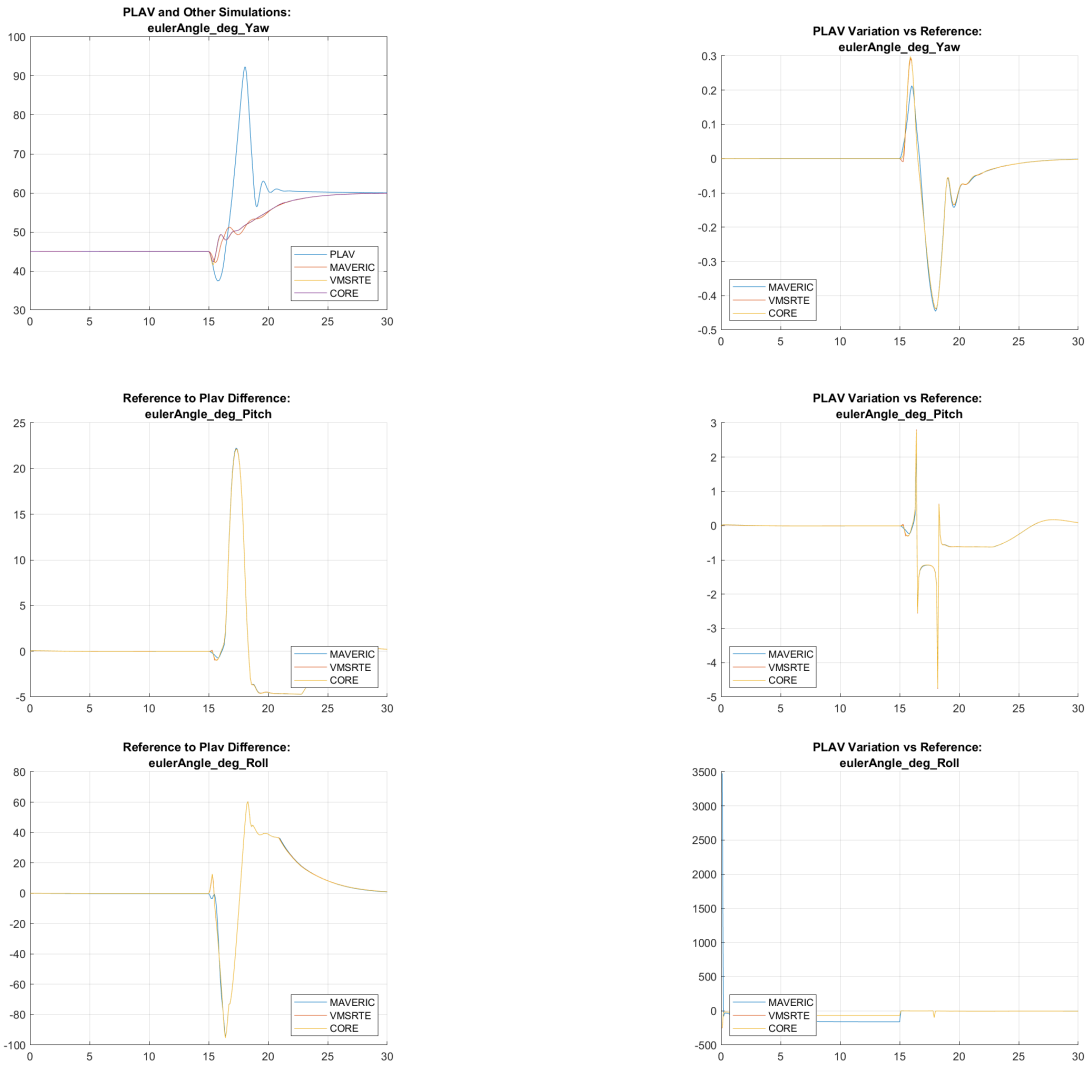


Figure 4.13: Case 13C Selected Results

4.2.13 Check Case 13C: Autopilot F-16 Course Change

This check case uses [1] with the autopilot active. At $t = 15$ seconds the autopilot is commanded to change the course by 15 degrees. Once again, the trim was manually solved. PLAV dramatically overshoots and steers more aggressively compared to the other simulators, as shown in the absolute yaw plot in Figure 4.12. This is because there was no lead logic implemented in PLAV, which may be necessary for an actual air vehicle.

Chapter 5

Assessing Practical Accuracy Requirements

5.1 Objective

This section attempts to analyze the consequences and importance of fully implementing Equations 2.2.5 to 2.2.10 to describe Coriolis effects and 2.2.13 to describe gravity effects with sufficient precision. In particular, NESC check case 10 of [16] is used and extended to 50 seconds. This vaguely resembles a large amateur rocket launch, for which there is increasing interest in developing active drag or control systems. This section seeks to evaluate whether terms beyond acceleration and constant gravity should be considered for simulations prepared for the analysis of these systems using numerical experiments.

5.2 Methodology

Four simulations were performed, one "full" reference version, one with no Coriolis effects and a flat Earth but J_2 gravity, one with constant gravity and an ellipsoidal Earth, and one with both no Coriolis effects, a flat Earth, and constant gravity. [15] describes simulations with no Earth rotation effects and a spherical Earth but still uses inverse square gravity. This section is designed to respond directly to the simplest simulation efforts.

Relative difference and absolute difference are compared for results most relevant

to rocketry. These are the maximum altitude, maximum speed, local gravity, and downrange distance. Deviations greater than one meter in altitude are considered significant, as they are within the resolution of a basic barometer (e.g. inside a cell phone) at sea level. Local gravity and speed deviations are relevant towards the goal of precisely predicting altitude. Downrange distance deviation is included as well.

5.3 Results

The maximum relative difference between the constant gravity and J_2 harmonic model (2.2.13) was 0.37%, corresponding to $0.036\,\mathrm{m/s^2}$ (Figure 5.2). This resulted in a difference in the apogee of 10.5 m for a reference altitude of 3252 m. The downrange distance and airspeed values were different by a noticeable but marginal amount considering the magnitude of the value (Figures 5.4, 5.6).

5.4 Experiment Conclusion

A 10.5 m altitude deviation at 3252 m seems small, however in the context of collegiate rocketry this different is significant. At the 2024 Spaceport America Cup rocketry challenge, 4 teams were within 12 m of their target altitude, with two teams only a couple of feet from each other [5]. For the maximum possible accuracy and best chance of winning, it is highly recommended to use a simulation with accurate gravity modeling, given the need for such accuracy and relative simplicity of implementation. Proper rendering of the WGS84 ellipsoid is more difficult and less important for a short flight, and so no strong recommendation is made as to its implementation for these purposes.

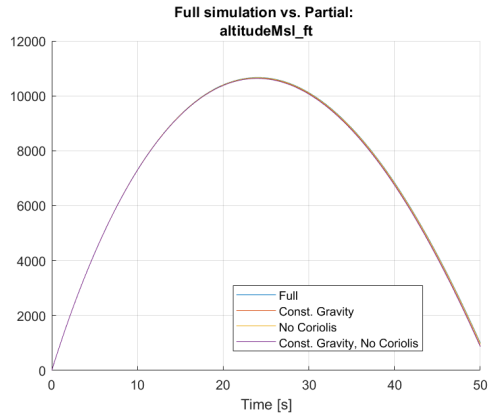


Figure 5.1: Altitude Plot for Modified Simulators

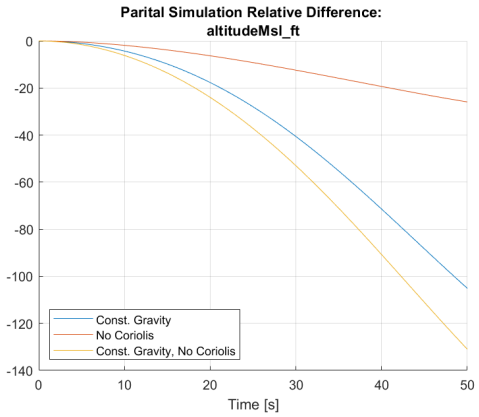


Figure 5.2: Altitude Difference Plot

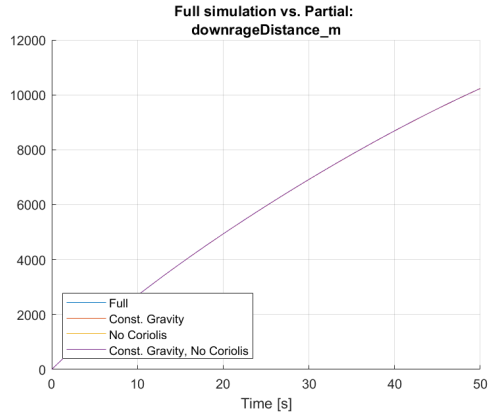


Figure 5.3: Downrange Distance Plot for Modified Simulators

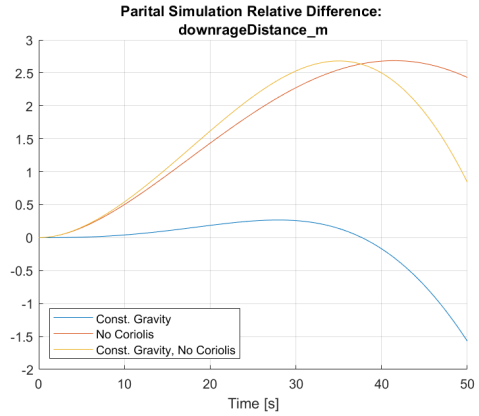


Figure 5.4: Downrange Distance Difference Plot

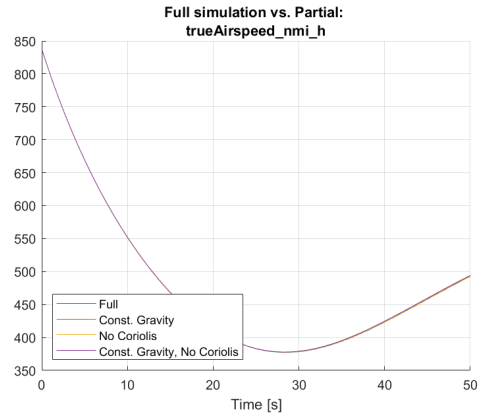


Figure 5.5: Airspeed Plot for Modified Simulators

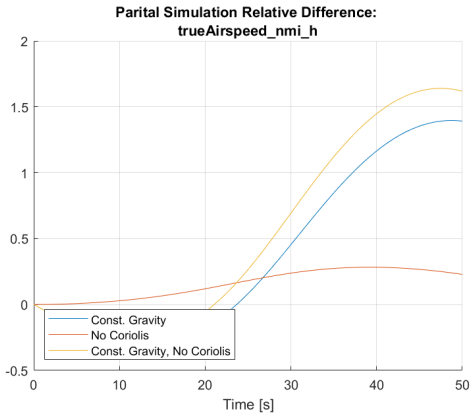


Figure 5.6: Airspeed Difference Plot

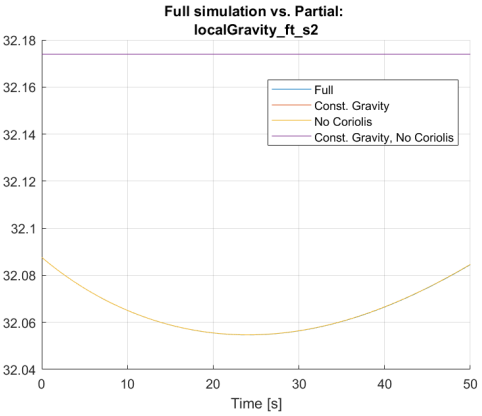


Figure 5.7: Gravity Plot for Modified Simulators

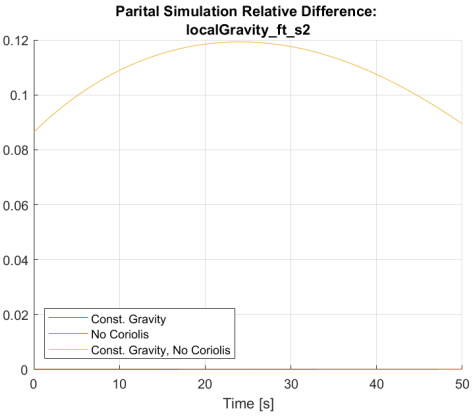


Figure 5.8: Gravity Difference Plot

Chapter 6

Discussion

6.1 Quality of Results

Overall, PLAV exhibited excellent agreement with the atmospheric check cases and can be considered validated for simulation use. It handled the open loop check cases correctly while closely approximating the F-16 check cases. Most differences can be explained by PLAV not fully implementing a trim solver or fully implementing the autopilot.

The comparison between the fully modeled WGS84 ellipsoid with J_2 harmonic gravity compared to a flat Earth with constant gravity demonstrates a valid case at the amateur level for fully modeling vehicle motion. At the most competitive collegiate rocketry level, fully modeling the Earth is just about required.

6.2 Lessons Learned

In the process of implementing a simulator in code, a few lessons about actual aircraft implementation were learned. Some are important clarifications while others are details an amateur aircraft designer might miss.

Perhaps the least obvious is relevance of α in the pitching moment calculation. As discussed in the aerodynamics section, this is actually a consequence of the level arm between the center of pressure to center of mass, which results in the expected torque. This was noticed when F-16 model was implemented, which makes no reference to a C_{m_α}.

Regarding the HITL implementation, care must be taken to how Python handles

integers compared to C++. Issues arise when signing the byte values received from the micro controller, as Python does not fix the integer size like C++ does. Therefore, the result must be manually signed from the sign bit.

The F-16 control system provided in [1] as well as the unimplemented GNC system demonstrate interesting lessons in autopilot design. In particular, it shows how the LQR control system must have its inputs carefully controlled to avoid over saturating the control and behaving erratically. This is a usable basis for implementing control for a drone vehicle.

Multiple conventions exist for defining the meaning of a quaternion. In works such as [6] q_4 is the scalar component, while in [3] q_1 is the scalar component. It is important to check which convention the source is using before implementing any formulas, otherwise the simulation will produce non-physical results.

Another important reminder for the aspiring aerodynamicist is to remember the difference between the dimensional and non-dimensional values of body roll rates p, q, and r. $\hat{p}$, $\hat{q}$, and $\hat{r}$, the non-dimensional values, are actually used to calculate the damping coefficient. It can be difficult to see the little hat in some environments, such as in $C_{\hat{q}}$.

6.3 Future Work

The simulator as is performs the most basic functions associated with aircraft simulation. For practical use, it requires a user interface and the ability to restart simulations without stopping the whole program. This also involves major refactoring to untangle the code from the check cases it was build to pass. Also, the Numba compilation time can be improved by caching compiled code.

The HITL system is still very basic and does not yet implement all the functions of a flight computer. Future work includes feeding it simulated sensor data with noise and modeling actuator dynamics. This will require improvements to the data transfer protocol and software beyond what was provided for the check cases. This way, the flight software used for simulation could about identical to the software used in flight, providing the highest assurance for success.

Bibliography

[1] Anon. *Standard: Flight Dynamics Model Exchange Standard (ANSI/AIAA S-119-2011(2016))*. American Institute of Aeronautics and Astronautics, 2016.

[2] Michael Boyle. *Quaternions in NumPy*, 2025. Version 2024.0.8, released April 1, 2025.

[3] Eric N. Johnson Brian L. Stevens, Frank L. Lewis. *Aircraft Control and Simulation: Dynamics, Controls Design, and Autonomous Systems*. Wiley-Blackwell, 3 edition, 2015.

[4] PyQtGraph developers. PyQtGraph: Pure-python graphics and gui library. `https://www.pyqtgraph.org`. Accessed: 2025-04-25.

[5] Experimental Sounding Rocket Association (ESRA). 2024 spaceport america cup. `https://www.soundingrocket.org/2024-sa-cup.html`, 2024. Accessed: 2025-04-25.

[6] John L. Crassidis F. Landis Markley. *Fundamentals of Spacecraft Attitude Determination and Control*. Springer Nature Link, 2014.

[7] Python Software Foundation. Comparing python to other languages.

[8] Andrew Hahn. *Vehicle Sketch Pad: A Parametric Geometry Modeler for Conceptual Aircraft Design*. American Institute of Aeronautics and Astronautics, 2012.

[9] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren

Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[10] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[11] John L. Junkins John L. Crassidis. *Optimal Estimation of Dynamic Systems*. CRC Press, 2nd edition, 2012.

[12] JSBSim Team. *JSBSim: Open Source Flight Dynamics & Control Software Library*. SourceForge, 2025. Version 1.1.11, released February 13, 2025; accessed April 22, 2025.

[13] Adin Kojic. PLAV: Python linear air vehicle flight simulator, 2025. GitHub repository.

[14] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: a llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, New York, NY, USA, 2015. Association for Computing Machinery.

[15] Jackson E. Bruce Shelton Robert O. Murri, Daniel G. Check-cases for verification of 6-degree-of-freedom flight vehicle simulations. Technical Memorandum NASA/TM-2015-218675/Vol I, NASA, Langley Research Center, Hampton VA 23681-2199, USA, January 2015.

[16] Jackson E. Bruce Shelton Robert O. Murri, Daniel G. Check-cases for verification of 6-degree-of-freedom flight vehicle simulations: Appendices. Technical Memorandum NASA/TM-2015-218675/VOL2, NASA, Langley Research Center, Hampton VA 23681-2199, USA, January 2015.

[17] Derek A. Paley N. Jeremy Kasdin. *Engineering Dyanmics: A Comprehensive Introduction*. Princeton University Press, 2011.

[18] NASA Langley Research Center and AIAA Modeling and Simulation Technical Committee. *DAVE-ML: Flight Dynamic Model Exchange using XML*. DAVE-ML Project, 2021. Last modified January 21, 2021; accessed April 22, 2025.

[19] National Oceanic and Atmospheric Administration, National Aeronautics and Space Administration, and United States Air Force. U.s. standard atmosphere,

1976. Technical Report NOAA-S/T 76-1562, NOAA, NASA, USAF, Washington, D.C., 1976.

[20] Office of Geomatics. Department of defense world geodetic system 1984: Its definition and relationships with local geodetic systems. Standardization Document NGA.STND.0036 1.0.0 WGS84, National Geospatial-Intelligence Agency, National Geospatial-Intelligence Agency 3838 Vogel Road Arnold, MO 63010-6205, July 2014.

[21] The pandas development team. pandas-dev/pandas: Pandas, February 2020.

[22] Robin C. Coon Paul J. Stoffregen. *Teensy® 4.1 Development Board*. PJRC, 2025. Accessed April 22, 2025.

[23] Python Software Foundation. *Python Language Reference, version 3.x*. Python Software Foundation, 2025.

[24] Daniel P. Raymer. *Aircraft Design: A Conceptual Approach*. AAIA, 6 edition, 2018.

[25] C. Venkatesan. *Fundamentals of Helicopter Dyanmics*. CRC Press, 2015.

[26] Matthew P. Whittaker and John L. Crassidis. Inertial navigation employing a common-frame error model. *Journal of Guidance, Control, and Dynamics*, 48(1):5–19, 2025.

Appendix A

Simulation Result Comparison Plots

More plots comparing PLAV with the other simulators are shown here for posterity.

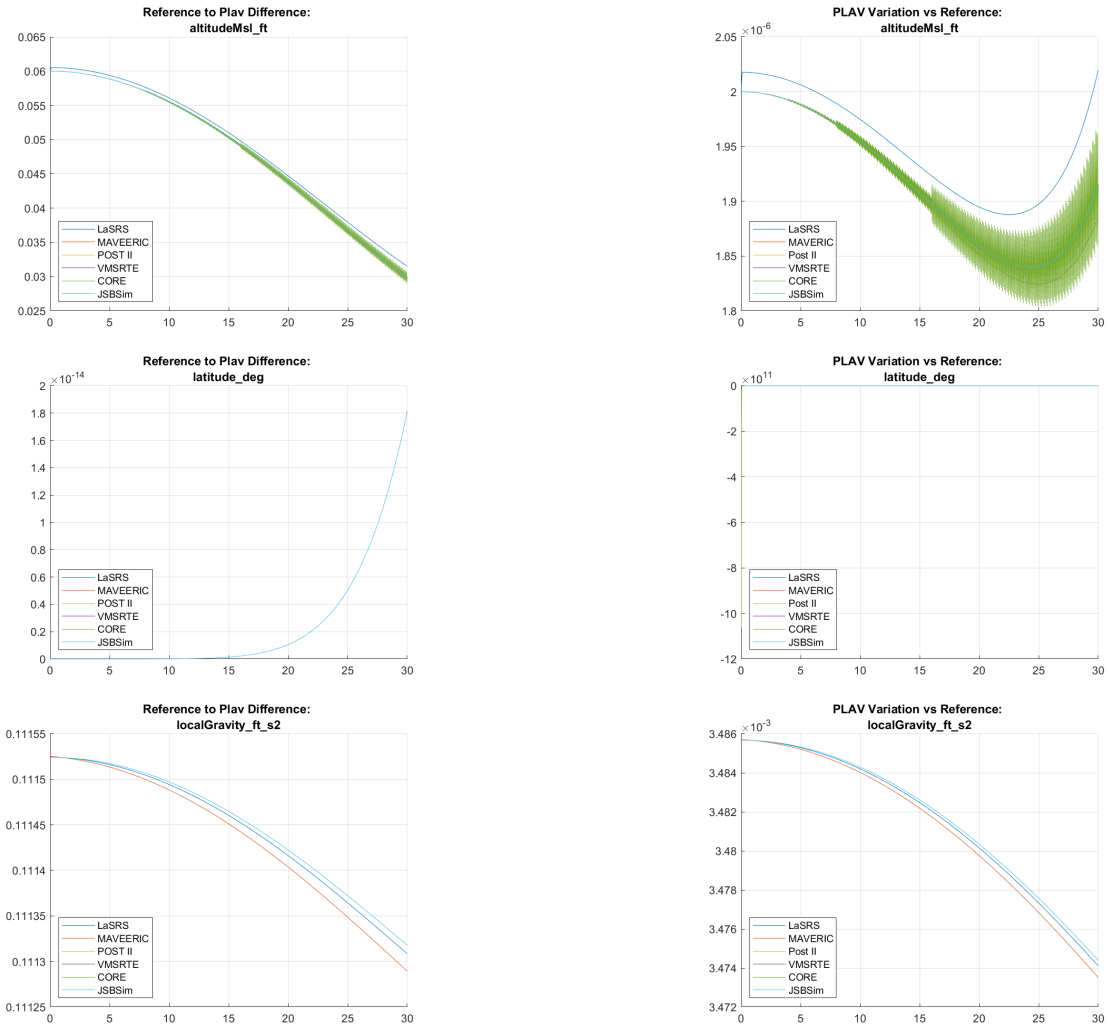


Figure A.1: Case 1 Results

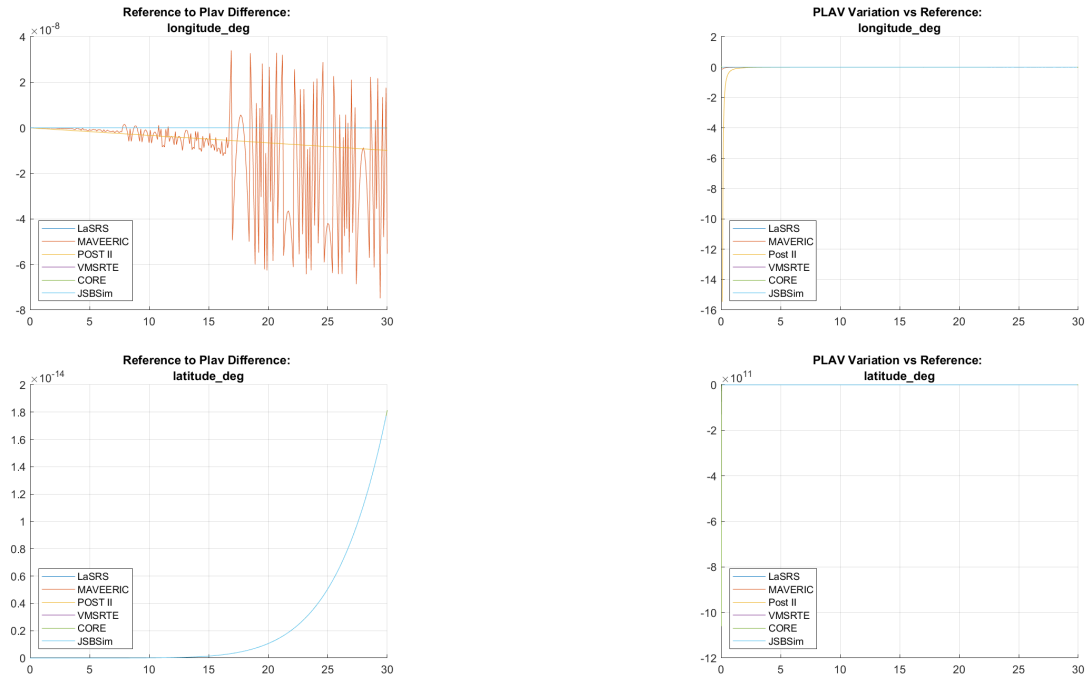


Figure A.2: Case 1 Results

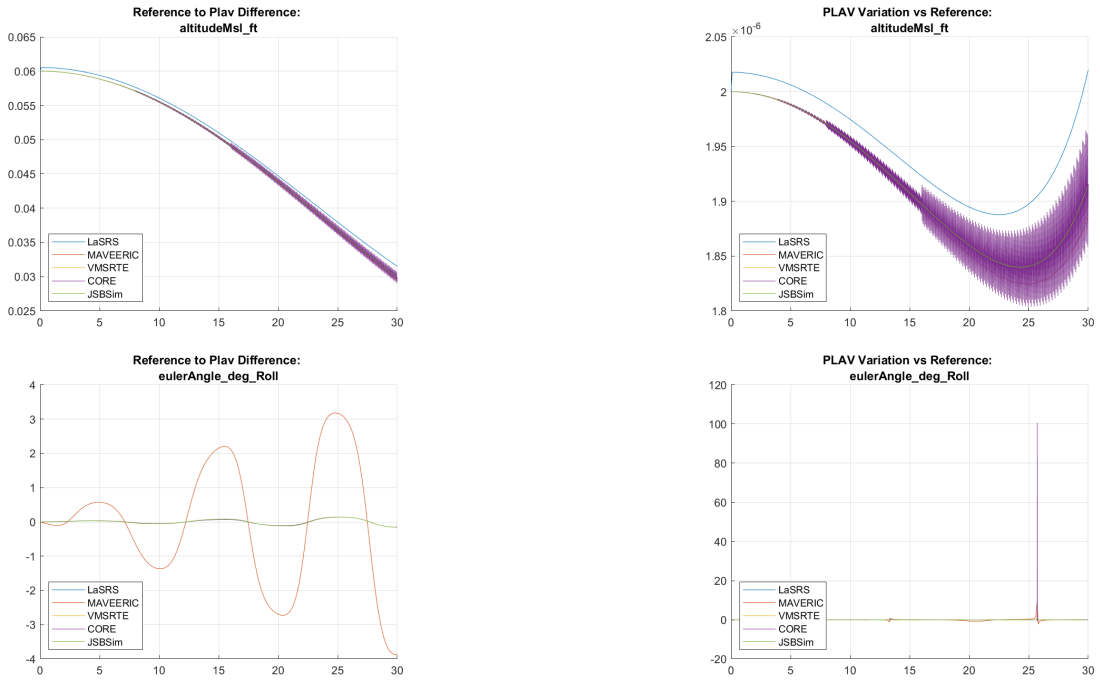


Figure A.3: Case 2 Results

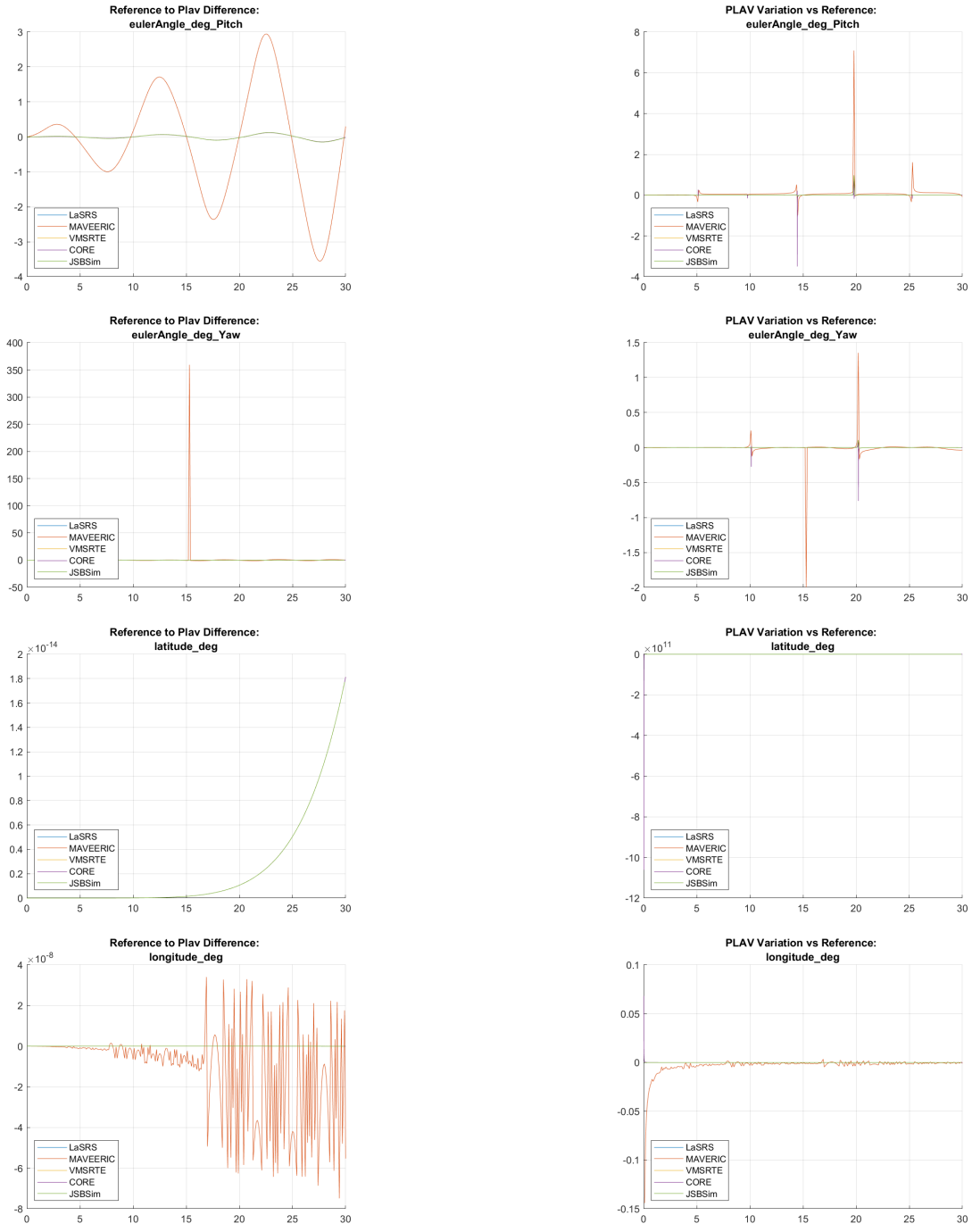


Figure A.4: Case 2 Results

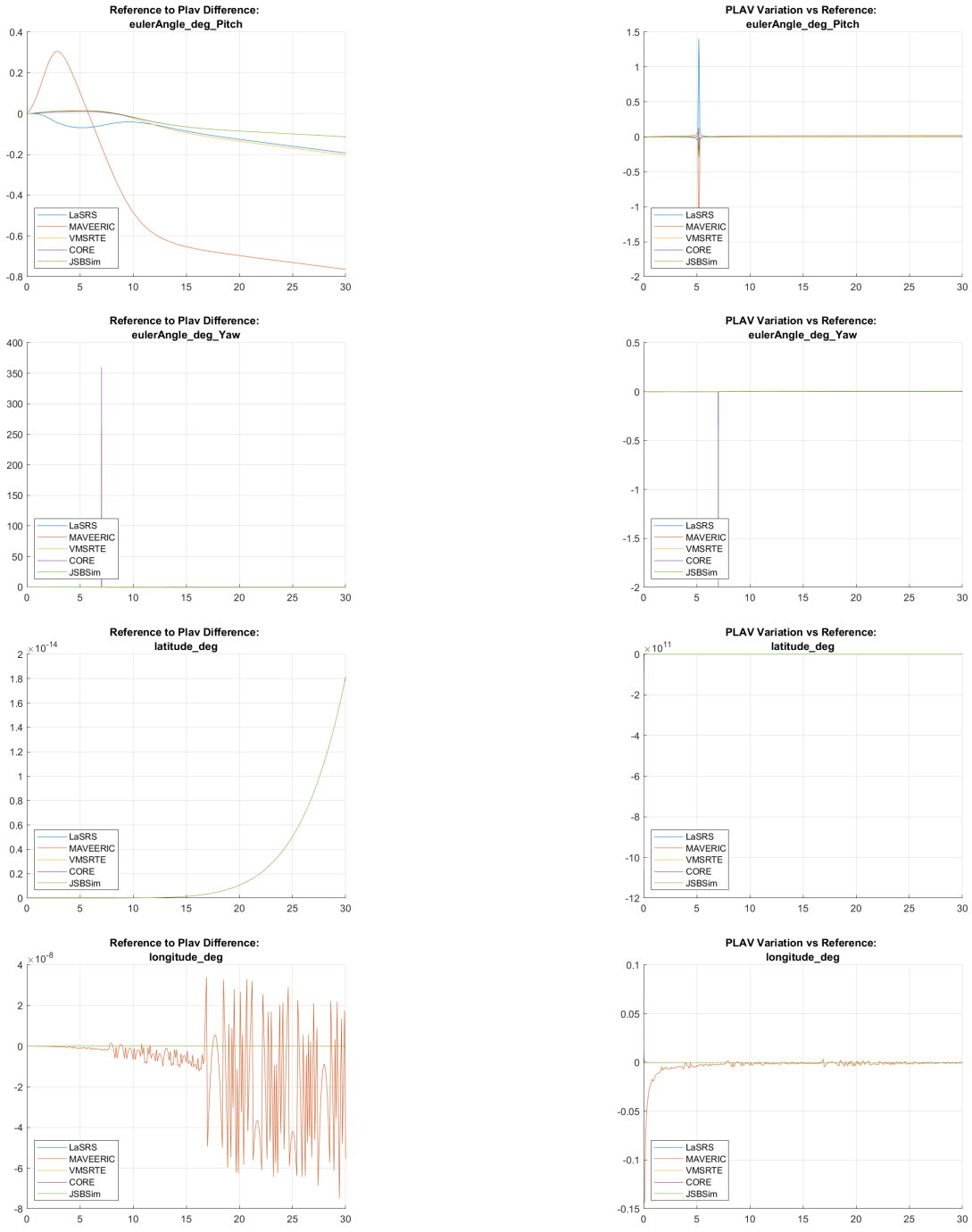


Figure A.5: Case 3 Results

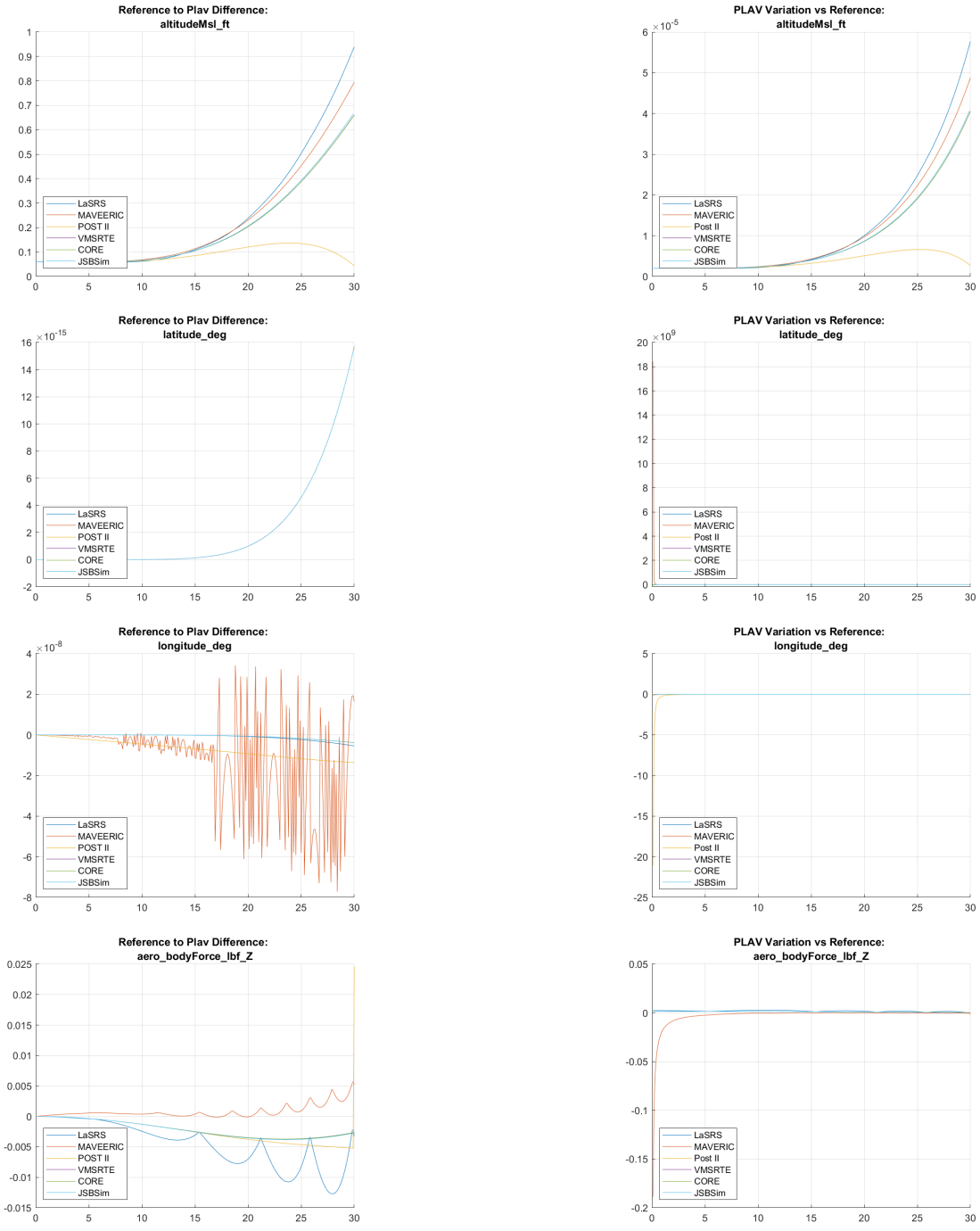


Figure A.6: Case 6 Results

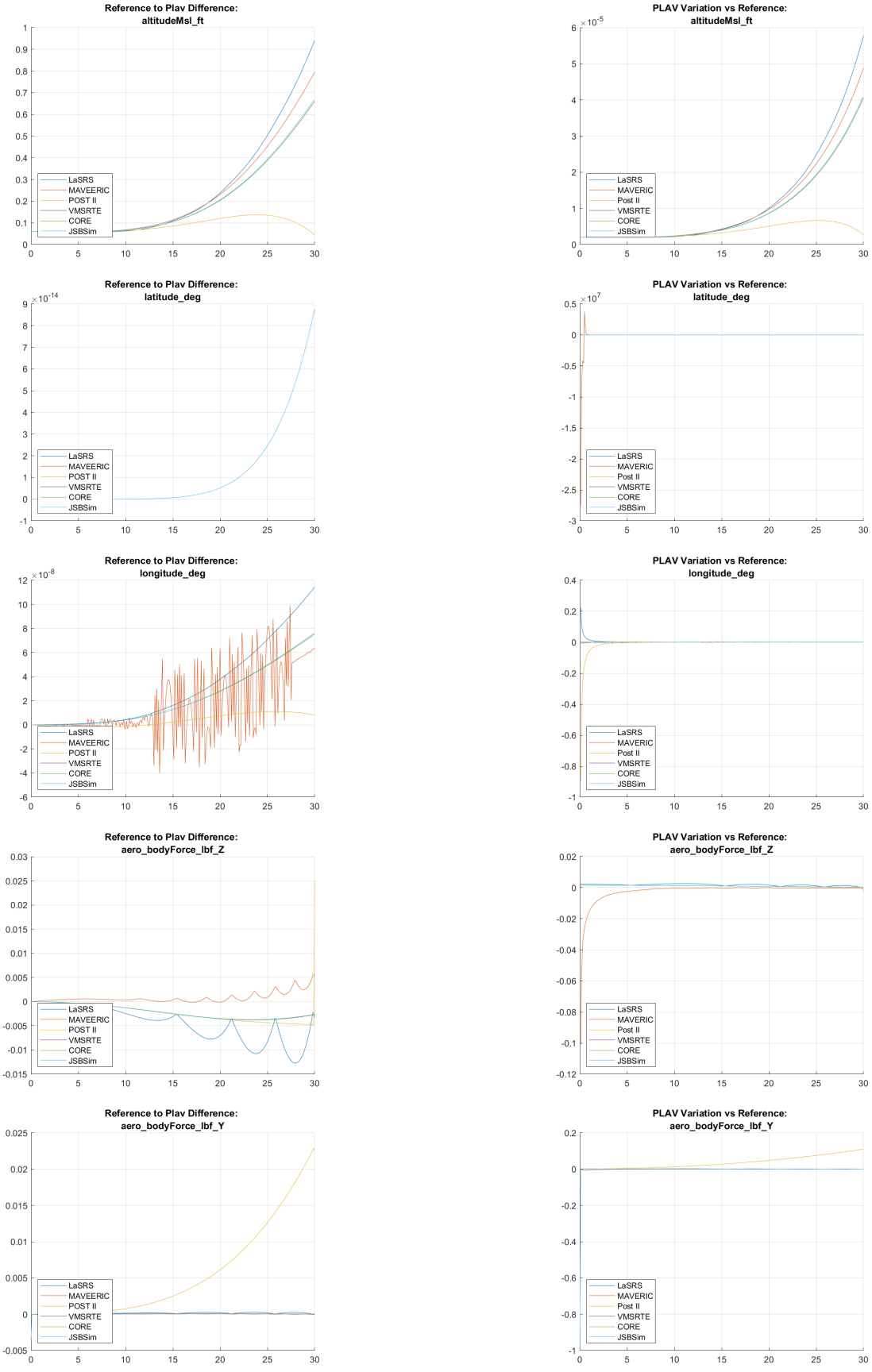


Figure A.7: Case 7 Results

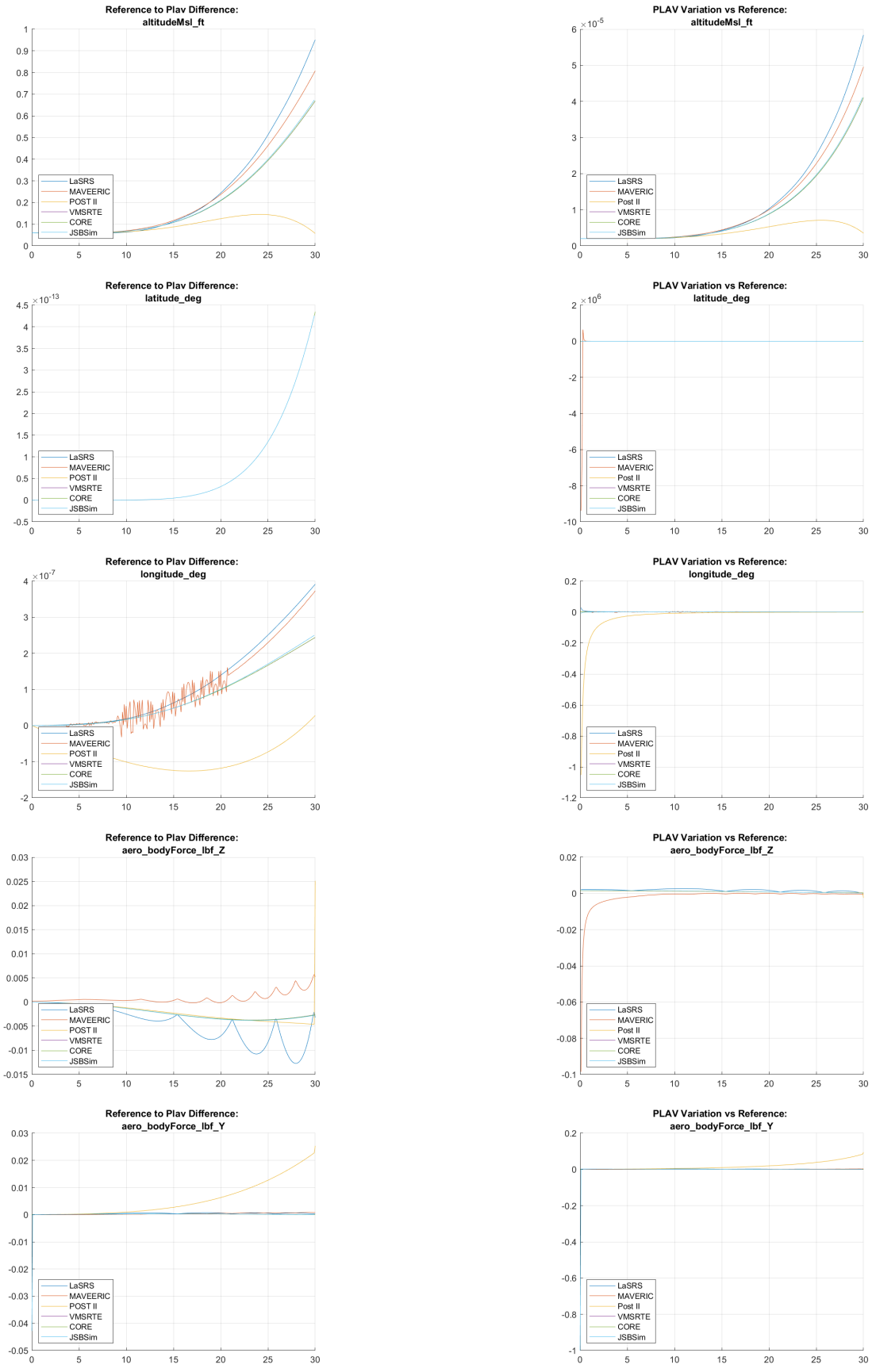
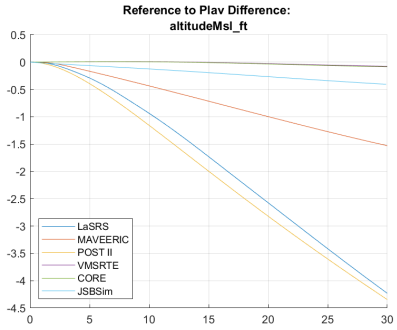
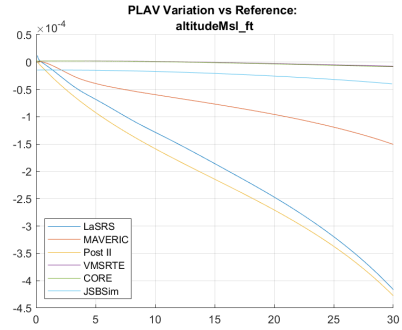
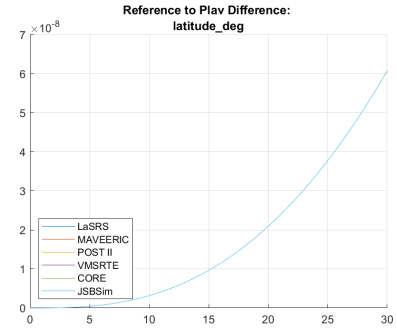
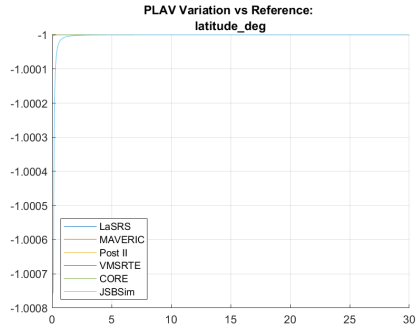


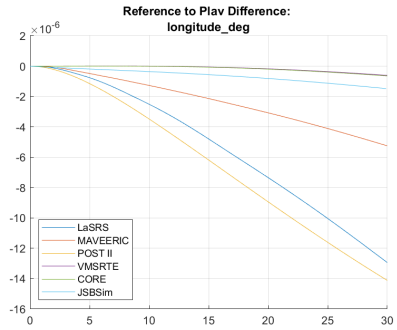
Figure A.8: Case 8 Results

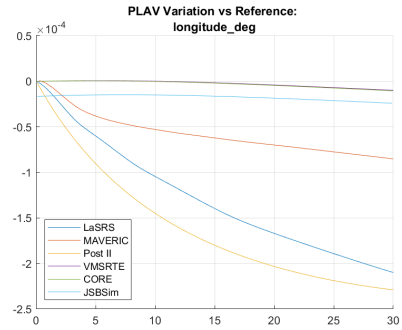


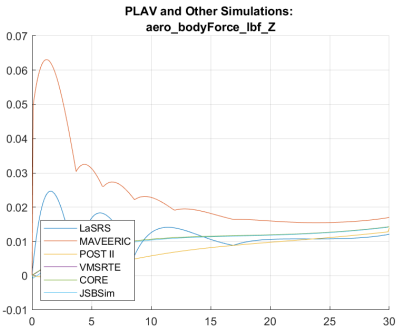


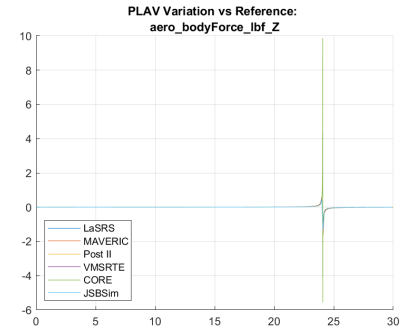


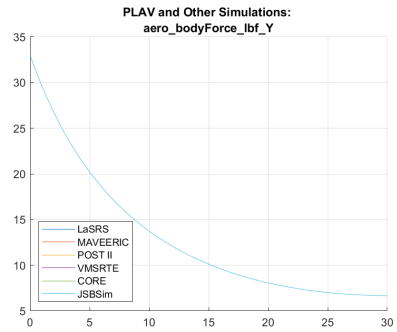












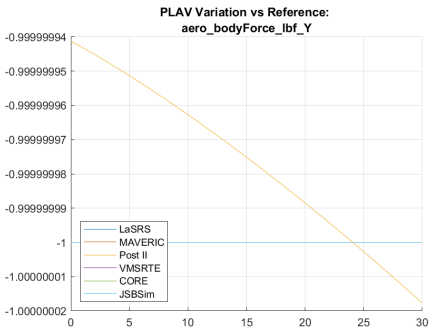


Figure A.9: Case 9 Results

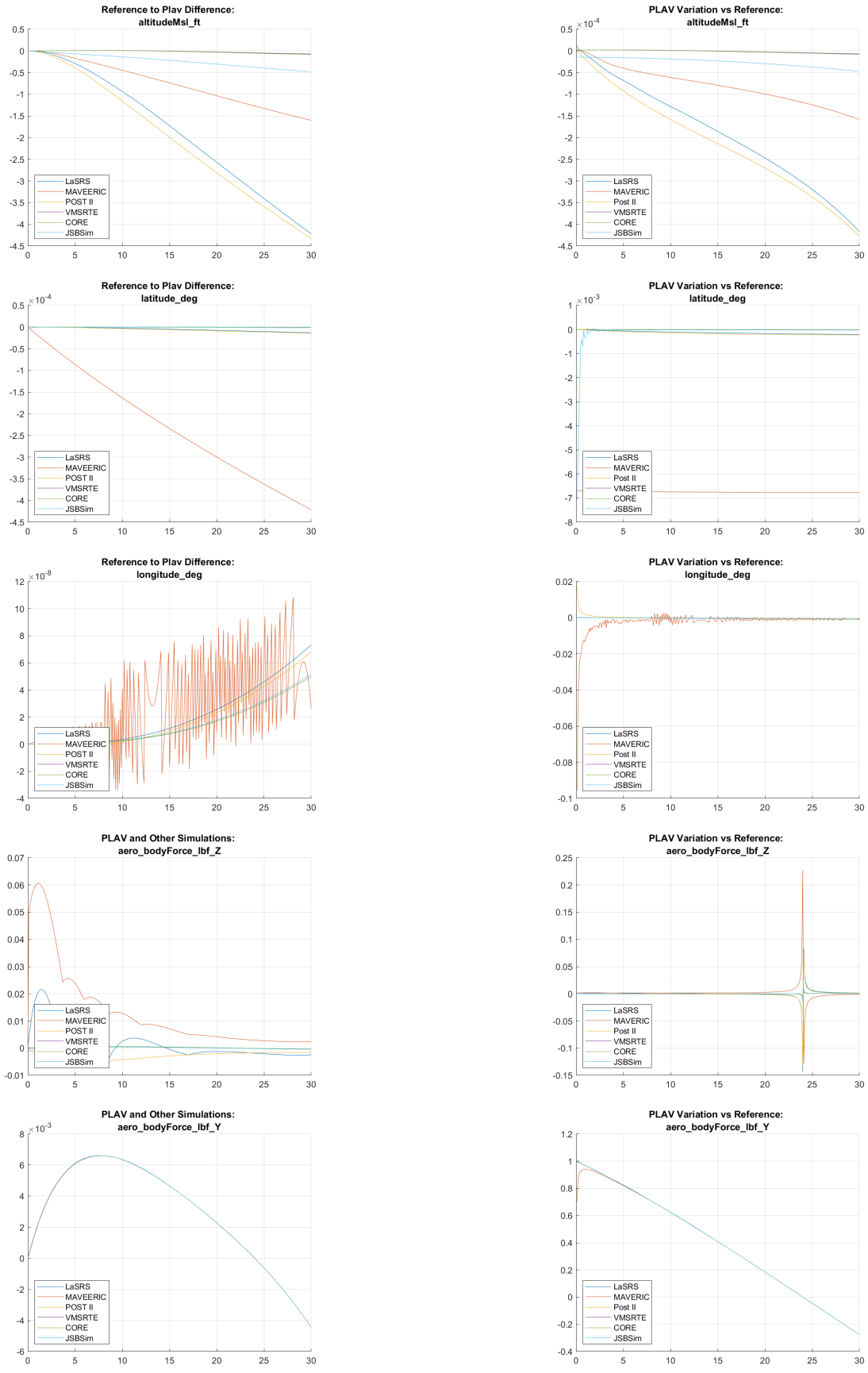


Figure A.10: Case 10 Results

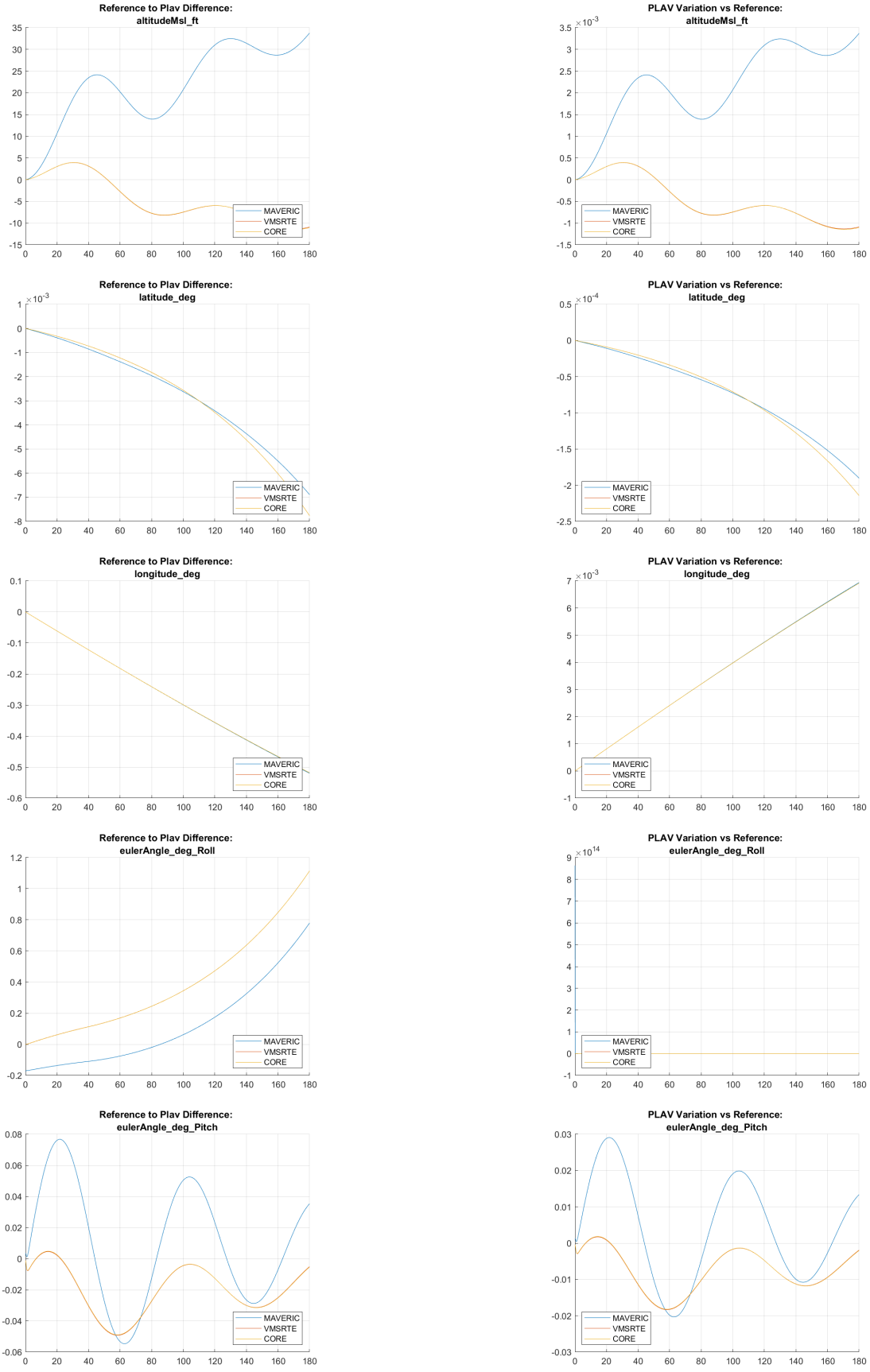


Figure A.11: Case 11 Results

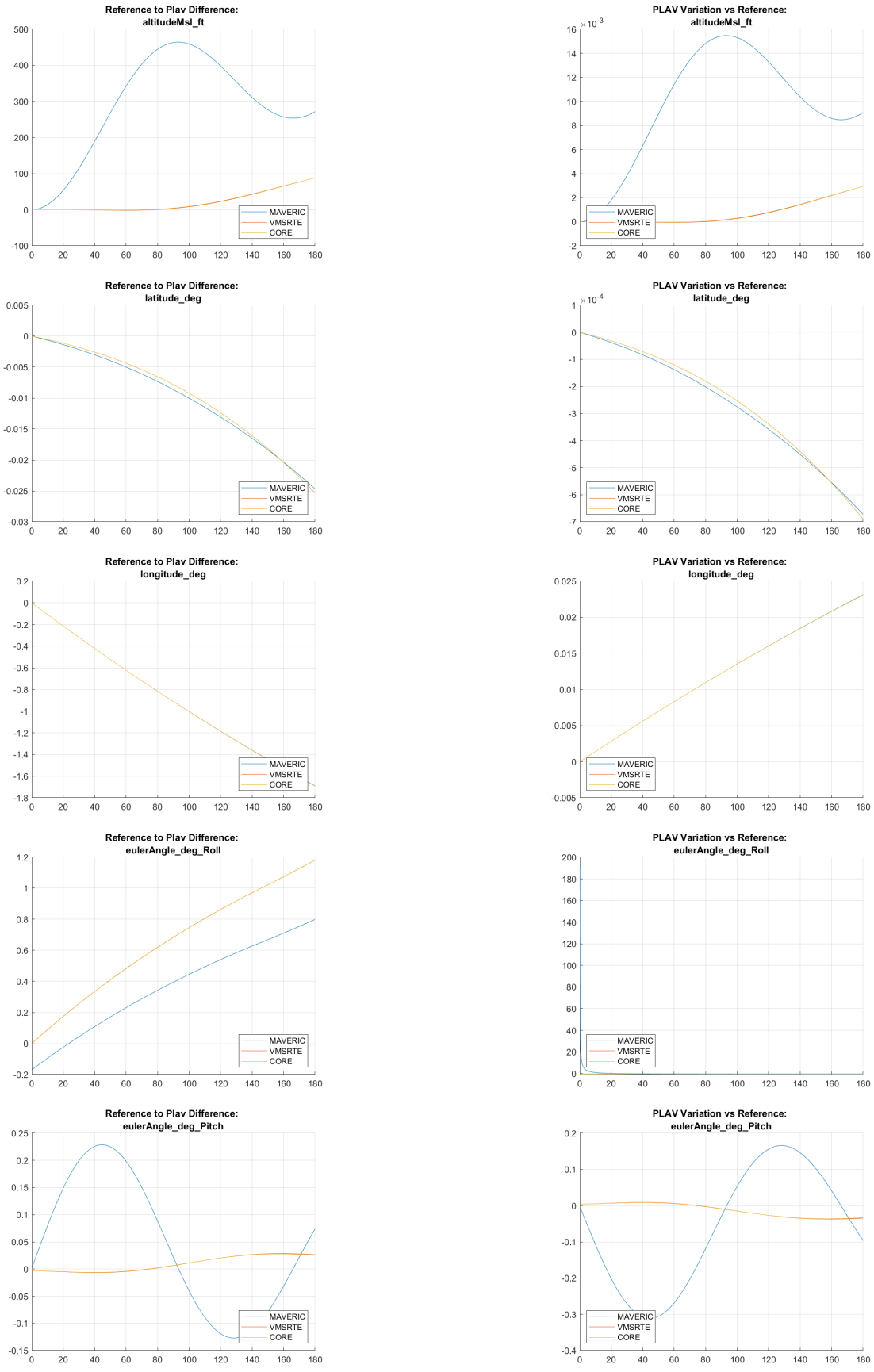


Figure A.12: Case 12 Results

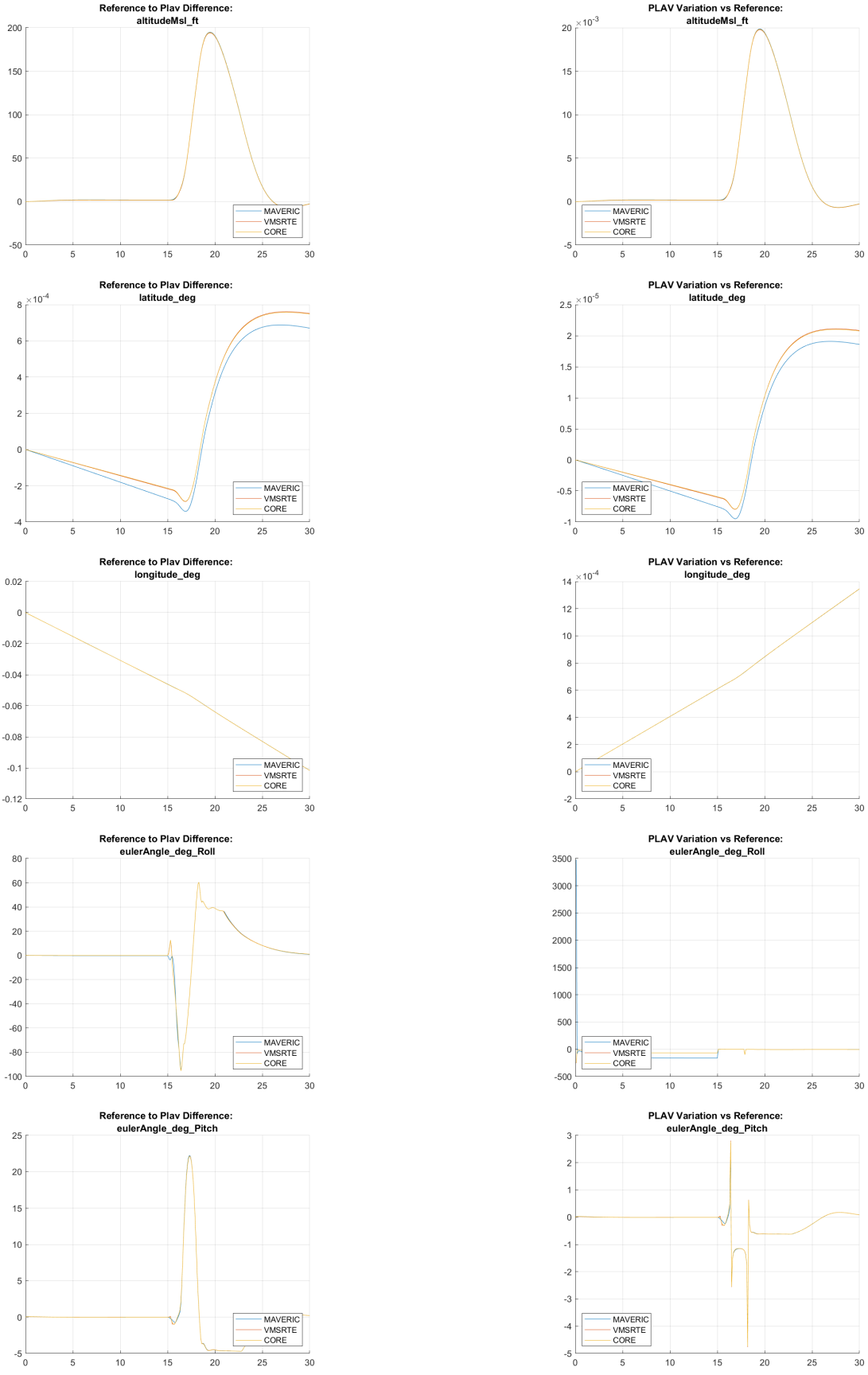


Figure A.13: Case 13A Results

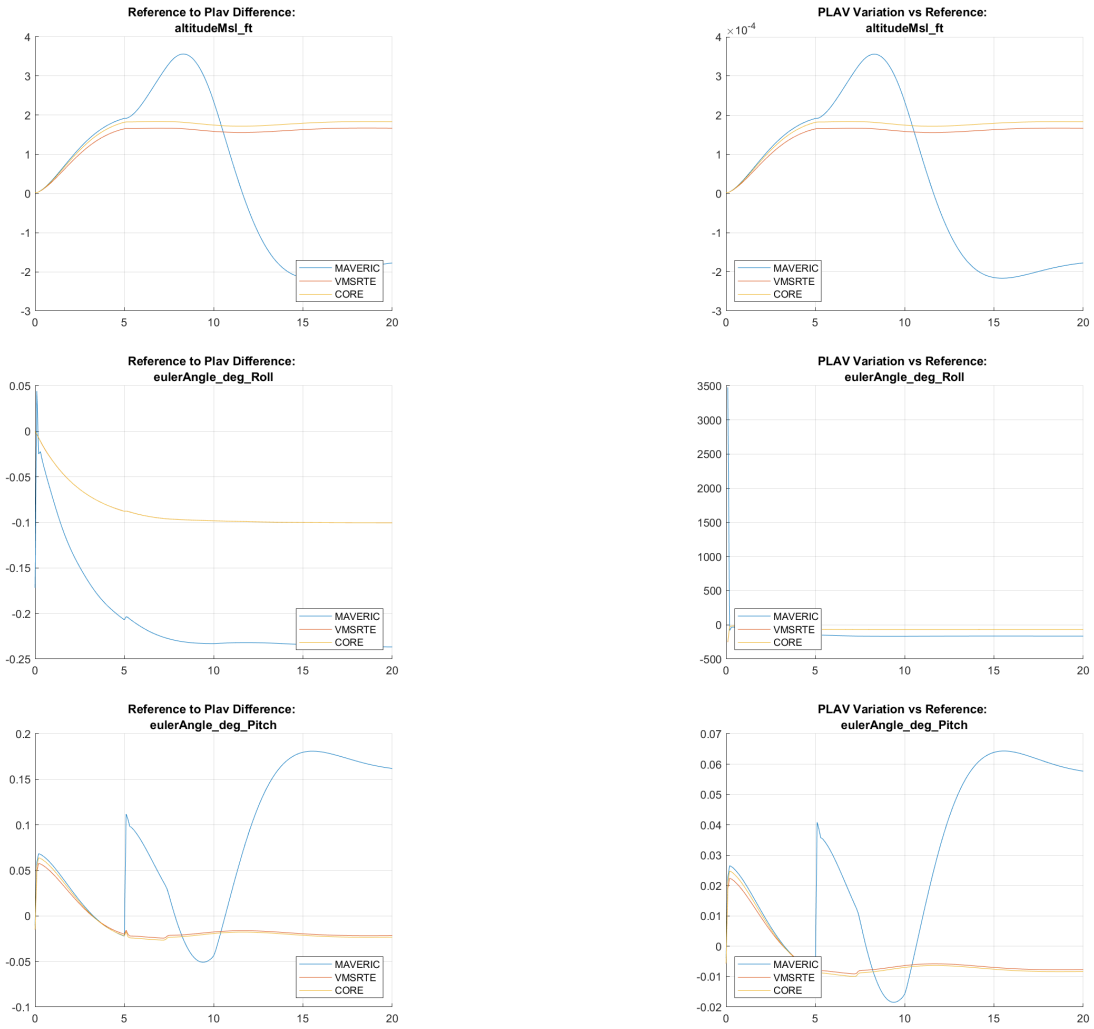


Figure A.14: Case 13B Results

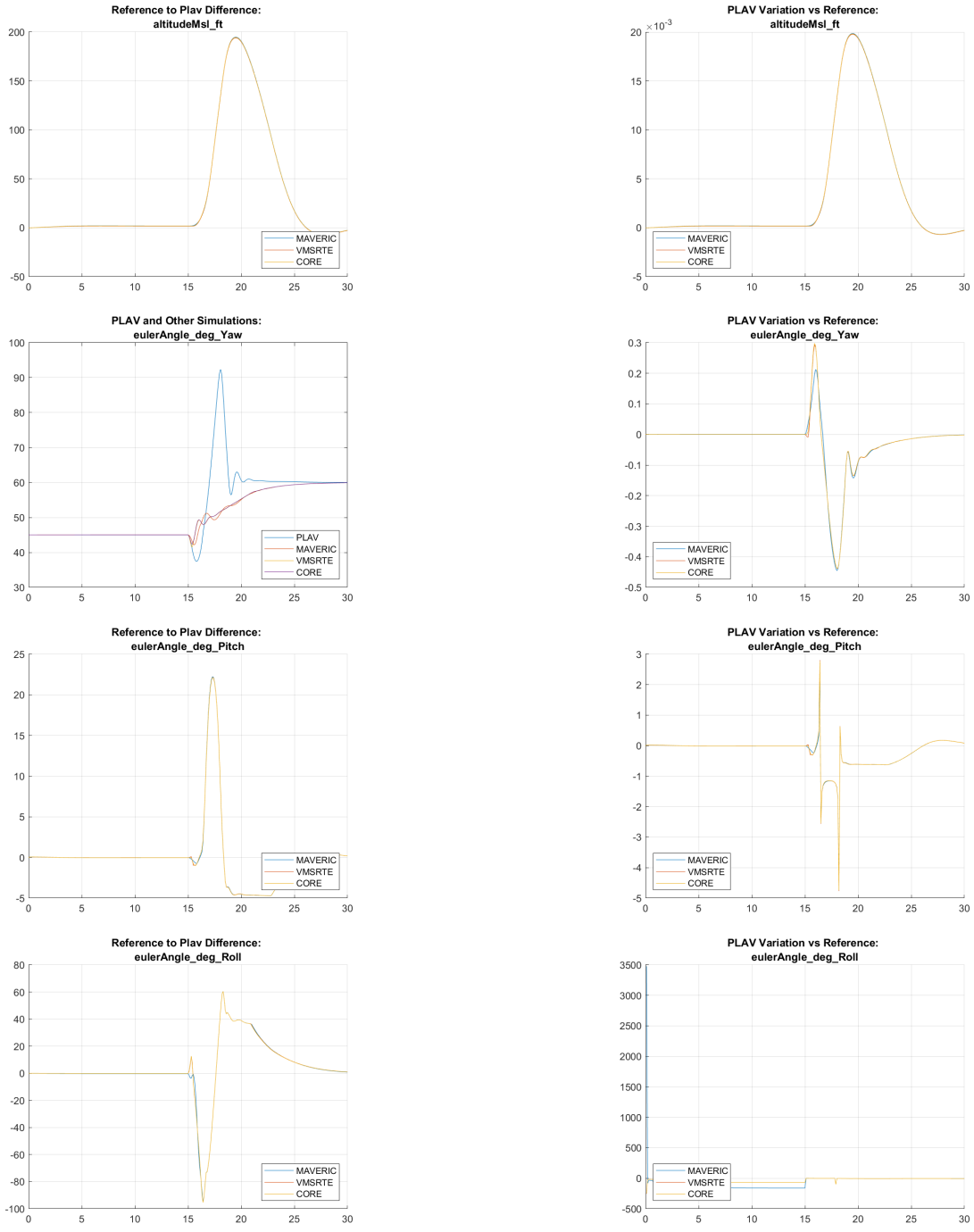


Figure A.15: Case 13C Results