

Politecnico di Torino

ThalesAlenia Space
a Thales / Leonardo company

Politecnico di Torino

Master's Degree in Mechatronic Engineering

A.a. 2025/2026

Graduation Session October 2026

Model-in-the-Loop Design and Test of a Lunar Lander Body Dynamics

Supervisors:
Marcello Chiaberge
Genny Scalise

Candidate:
Antonino Gaetano Biondo

Acknowledgements

As I conclude this important chapter of my life, there are a few special people I need to thank. First and foremost, my deepest gratitude goes to my family for their unwavering support and endless sacrifices. You never let me lack anything, and this milestone is as much yours as it is mine.

To my girlfriend, Erika: thank you for being my rock. Even with the distance between us, and despite the difficult moments you were going through yourself, you were always close to me when I felt overwhelmed, giving me the courage to persevere and never give up. I hope that in the future, I will be able to reciprocate all the love and affection you have given me.

Lastly, a huge thank you to my amazing friends. You were my safe haven during the toughest and darkest moments, always stepping in to cheer me up, make me laugh, and remind me to keep going.

Thank you for being an essential part of this journey.

Abstract

The objective of this thesis is the development of a multibody simulation model for analyzing the landing dynamics of a lunar lander, with the ultimate goal of evaluating its stability, impact absorption capacity, and rollover resistance under varying operational conditions. The core of the work focuses on the physical and mathematical modeling of the vehicle within the ROS2 and Gazebo simulation environment. Initially, the rigid body dynamics of the lander were defined, accurately modeling mass distributions and inertia matrices by also accounting for the remaining propellant at the start of the maneuver. Subsequently, the study focused on the advanced modeling of the landing gear's kinematics and dynamics. Concurrently, the mechanical interaction between the lander's footpads and the lunar regolith was parameterized to realistically reproduce friction, slipping, and ground reaction forces. To complete the model, the propulsion system component was integrated, simulating the residual kinematics during the terminal moments of the descent. The completed model was then packaged into an automated and parametric simulation framework. This architecture was specifically designed to facilitate the efficient execution of extensive simulation campaigns involving a high number of runs. Finally, to test the simulator, a global sensitivity analysis campaign was conducted to identify which initial parameters most significantly drive the lander's rollover risk.

Table of Contents

List of Figures VIII

1 Introduction 1
- 1.1 Context and Motivation . . . 1
  - 1.1.1 Returning to the Moon: The Next Generation of Space Exploration . . . 1
  - 1.1.2 Heavy-Class Lunar Landers and Logistic Platforms . . . 1
  - 1.1.3 The Criticality of the Terminal Descent Phase . . . 2
- 1.2 Problem Statement . . . 3
  - 1.2.1 Overcoming terrestrial testing limitations via Model-in-the-Loop architecture . . . 3
  - 1.2.2 Thesis Objectives . . . 5
- 1.3 Thesis Outline . . . 5

2 State of the Art and Theoretical Background 7
- 2.1 Lunar Lander Dynamics . . . 7
- 2.2 Spacecraft Landing Gear Systems . . . 8
- 2.3 Simulation Environments in Aerospace . . . 9
- 2.4 Global Sensitivity Analysis for Soft Landing . . . 10
  - 2.4.1 The Morris Method (Method of Elementary Effects) . . . 11
  - 2.4.2 Variance-Based Sensitivity Analysis . . . 11

3 Mathematical and Physical Modeling 14
- 3.1 Lander Modeling . . . 15
  - 3.1.1 Propellant Ripartition . . . 15
- 3.2 Tank and Propellant Properties . . . 15
  - 3.2.1 Geometric Parameterization of the Tank . . . 16
  - 3.2.2 Radius of the Generating Spherical Cap . . . 16
  - 3.2.3 Local Radius Profile Function . . . 17
  - 3.2.4 Iterative Determination of Fluid Level (h_{fill}) . . . 17
  - 3.2.5 Discretization and Calculation of Local Inertial Properties . 18

3.3 Body Link Assembly and Tensor Transport 18
3.3.1 Body Center of Gravity Calculation 18
3.3.2 Inertia Tensor Transport via the Parallel Axis Theorem . . . 19
3.4 System Disturbances: Plume Surface Interaction 20
3.4.1 Disturbance Force Magnitude formulation 20
3.4.2 Spatial and Temporal Distribution 20
3.5 Kinematics and Dynamics of Landing Legs 21
3.5.1 Coordinate Systems and Reference Frames 21
3.5.2 Topological Description of the Landing Gear 22
3.5.3 Kinematics of Fixed Trusses 23
3.5.4 Central and Lateral Shock Absorbers 24
3.5.5 Inertial Properties Modeling 25
3.5.6 Mathematical Model of the Crushable Shock Absorber . . . 25
3.6 Touchdown Dynamics and Contact Mechanics 27
3.6.1 Contact Mechanics 27
3.6.2 Dynamic Dual-Layer Lunar Regolith Modeling 28

4 Software Architecture and Simulation (Model-in-the-Loop) 30
4.1 System Architecture Overview 30
4.1.1 Workspace Topology and Package Organization 30
4.1.2 Simulation Orchestration via ROS 2 Launch System 31
4.1.3 Network Topology and Inter-Process Communication 32
4.2 Centralized Parametric Configuration 33
4.2.1 Configuration Domains and Data Structures 34
4.3 Body and Payload Generation (`generate_body.py`) 35
4.3.1 Mass Sampling and Scenario Selection 35
4.3.2 Numerical Integration of Fluid Properties 36
4.3.3 Global Tensor Assembly and SDF Templating 37
4.4 Scenario Generation (`generate_scenario.py`) 38
4.4.1 Safe Altitude Ray-Casting and 3D Attitude Resolution . . . 39
4.4.2 Dynamic Mesh Cropping for Computational Efficiency . . . 41
4.4.3 The InitialVelocity C++ Plugin 42
4.5 Landing Gear Procedural Assembly and Shock Absorber Modeling . 43
4.5.1 Legs Generation (`generate_legs.py`) 43
4.5.2 C++ Implementation of the Crushable Shock Absorber . . . 45
4.6 Actuation System (Thrusters) 48
4.6.1 The Custom C++ Propulsion Plugin 48
4.6.2 Main Engine Actuation and MECO Logic 49
4.7 Plume Surface Interaction (PSI) Disturbance Generation 53
4.7.1 Dynamic True Altitude Tracking 53
4.7.2 Disturbance Calculation and Thrust Synchronization 54

4.7.3 Spatial Modulation and ROS 2 Publishing 55
4.8 Dynamic Contact Mechanics 55
4.8.1 Plugin Initialization and Parameter Parsing 56
4.8.2 Intercepting the Gazebo Contact Manager 56
4.8.3 Real-Time Parameter Interpolation and Injection 57
4.9 Automated Data Logging and Evaluation Criteria 58
4.9.1 Node Architecture and Middleware Subscriptions 58
4.9.2 Kinematic Telemetry via Gazebo ROS State Plugin 59
4.9.3 Contact Mechanics via Gazebo ROS Bumper Plugin 61
4.9.4 Post-Flight Data Processing 64

5 Sensitivity Analysis and Conclusion 70
5.1 Rationale for the Sequential Approach in High-Fidelity Simulations 70
5.2 First Simulation Campaign 71
5.2.1 Phase I: Global Parameter Screening 71
5.2.2 Phase II: Variance-Based Analysis (Sobol Method) 73
5.2.3 Design of Experiments and Computational Execution 73
5.3 Fixed-Topography Analysis 75
5.3.1 Topographic Discretization and Target Spot Selection 76
5.3.2 Nominal Landing Scenario: 10° Slope Analysis 78
5.3.3 Operational Limit Scenario: 15° Slope Analysis 82
5.3.4 Off-Nominal Scenario: 20° Slope Analysis 85
5.3.5 Synthesis of the Sensitivity Campaign 87
5.4 Conclusion 89

A ROS 2 `rqt_graph` 91

Bibliography 93

List of Figures

1.1 V-Model architecture. Source: Wikipedia [1]. 4

3.1 Visual representation of the full lander (Body + Passenger + legs) within the Gazebo simulation environment. 14
3.2 Kinematic topology of the landing gear: visual representation of the structural nodes and joints within the Gazebo simulation environment. 22

4.1 Partial ROS 2 computational graph of the lunar lander simulation architecture. 33
4.2 Individual motor thrust plot. 53
4.3 Kinematic telemetry profile over the course of the simulation. . . . 59
4.4 Temporal evolution of the 3D contact forces (F_x, F_y, F_z). 63

5.1 Morris Sensitivity Analysis: Scatter plot of Global Importance (μ^*) versus Non-Linearity/Interactions (σ) for the Maximum Dynamic Tilt. 72
5.2 First-Order (S_1) and Total-Order (S_{T_i}) Sobol Indices illustrating the parameter impact on the maximum dynamic tilt angle. 74
5.3 Slope map of the lunar regolith mesh (spatial derivative). 77
5.4 Morris Sensitivity Analysis (10° Slope): Scatter plot of Global Importance (μ^*) versus Non-Linearity/Interactions (σ) for the Maximum Dynamic Tilt. 78
5.5 First-Order (S_1) and Total-Order (S_{T_i}) Sobol Indices illustrating the parameter impact on the maximum dynamic tilt angle on a 10° slope. 79
5.6 Sobol Second-Order Indices (S_2) Heatmap detailing the interaction strength between parameter pairs. 81
5.7 Morris Sensitivity Analysis (15° Slope): Scatter plot of Global Importance (μ^*) versus Non-Linearity/Interactions (σ) for the Maximum Dynamic Tilt. 82
5.8 First-Order (S_1) and Total-Order (S_{T_i}) Sobol Indices illustrating the parameter impact on the maximum dynamic tilt angle on a 15° slope. 83

5.9 Sobol Second-Order Indices (S_2) Heatmap detailing the interaction strength between parameter pairs for the 15° slope. 84
5.10 Morris Sensitivity Analysis (20° Slope): Scatter plot of Global Importance (μ^*) versus Non-Linearity/Interactions (σ) for the Maximum Dynamic Tilt. 85
5.11 First-Order (S_1) and Total-Order (S_{T_i}) Sobol Indices illustrating the parameter impact on the maximum dynamic tilt angle on a 20° slope. 86
5.12 Sobol Second-Order Indices (S_2) Heatmap detailing the interaction strength between parameter pairs for the 20° slope. 87
5.13 Analysis of dynamic rollover events across simulated topographical inclinations. (a) Probability of catastrophic failure. (b) Distribution of final tilt angles exclusively for the failed runs. 89

A.1 Full ROS 2 computational graph of the lunar lander simulation architecture generated via `rqt_graph`. 92

Chapter 1

Introduction

Space exploration is currently navigating a profound paradigm shift. Accordingly, this introductory chapter delineates the contemporary landscape and establishes the context driving this thesis.

1.1 Context and Motivation

1.1.1 Returning to the Moon: The Next Generation of Space Exploration

In recent years, the primary focus of major global space agencies has shifted from Low Earth Orbit (LEO) operations toward a renewed exploration of the Moon. Unlike the historic Apollo program, which was characterized by short-duration stays, contemporary initiatives such as the Artemis program aim to establish a permanent and sustainable human presence on the lunar surface. This renewed lunar race is driven by the recognition of our satellite as an unparalleled proving ground for testing technologies necessary for human habitation on other planetary bodies, thereby holding profound scientific and economic significance. Consequently, this endeavor has the potential to heavily impact global energy and raw material sectors, fostering the development of a robust space economy. Within this context, the development of a highly reliable logistical infrastructure, capable of efficiently and safely delivering payloads to the lunar surface, becomes paramount.

1.1.2 Heavy-Class Lunar Landers and Logistic Platforms

Driven by international cooperation and strategic competition, global space agencies are investing in the development of heavy-class lunar landers and large logistic

platforms. These landers represent the cornerstone of future strategies for independent and autonomous access to the lunar surface, serving as versatile heavy landing systems fully integrated into the international logistical architecture.

The primary objective of these vehicles is to provide a highly adaptable, end-to-end transport capability. They are designed to be launched via heavy-lift rockets and autonomously inject themselves into a lunar transfer trajectory. The uniqueness of these architectures lies in their modular nature: the same baseline landing platform (*Body Element*) can accommodate a wide variety of payloads on its upper deck (*Passenger Element*).

Anticipated configurations for these platforms include:

- Logistical resupply missions to support human crews, delivering habitation modules, Life Support Systems, food, and equipment;
- The deployment of heavy scientific rovers for geological exploration and volatile mapping;
- The landing of stationary laboratories dedicated to testing local resource extraction and utilization technologies.

With a net payload capacity often ranging between 1500 kg and 2000 kg or more depending on the operational scenario, these vehicles fall into a heavy mass category. This introduces unprecedented engineering challenges for Guidance, Navigation, and Control (GNC) systems, as well as for structural impact and dynamics analysis.

1.1.3 The Criticality of the Terminal Descent Phase

Within the mission profile of a planetary lander, the final descent phase is unequivocally the highest-risk segment with the lowest margin of error. Despite historical successes and modern computational tools, a controlled lunar landing (Commonly referred to as a *soft landing*) remains a complex engineering maneuvers.

The primary environmental severity is dictated by the near-total absence of an atmosphere on the satellite. This condition strictly precludes the use of any passive aerodynamic deceleration systems, such as heat shields, lifting surfaces, or parachutes, which on Earth or Mars dissipate the vast majority of the vehicle's kinetic energy essentially for free. In the lunar scenario, braking must rely solely and entirely on the retro-propulsive action of the main rocket engines. The vehicle must actively consume its propellant to counteract the lunar gravitational acceleration and nullify its relative velocity, both vertically and horizontally.

The culmination of this retro-propulsive process is the final dynamic transition: the *touchdown* phase, defined as the exact moment the lower extremities of the landing gear establish physical contact with the irregular lunar terrain. This phase introduces extreme multidisciplinary criticalities:

1. **Contact Mechanics and Structural Loads:** Upon impact, the residual kinetic energy must be absorbed and dissipated by the landing gear mechanisms. Poorly distributed impact forces or vertical velocities exceeding design limits can cause catastrophic structural failures, compromising the integrity of sensitive payloads.

2. **Stability and Tip-Over Risk:** The real lunar topography is inherently uncooperative, featuring local slopes, scattered boulders, and craters. A touchdown executed with a non-zero residual horizontal velocity or a highly inclined attitude (high Roll/Pitch) can trigger a tipping moment due to a single footpad catching on the terrain. Exceeding the critical tip-over angle results in total mission failure.

3. **Terminal Descent Cut-Off:** Terminating propulsion too early forces the vehicle into a prolonged free-fall. This acts as a severe energy amplifier, drastically increasing vertical impact velocity and allowing minor attitude errors to compound before contact. Consequently, precisely timing the engine shutdown is critical to prevent overwhelming the vehicle's structural and stability margins.

A profound understanding of these phenomena requires an extensive testing campaign.

1.2 Problem Statement

Since replicating the combined lunar environmental conditions at full scale on Earth is physically impossible, rigorous mathematical modeling and advanced numerical simulation emerge as the primary tools available to engineers for validating design requirements and ensuring a safe, successful landing.

1.2.1 Overcoming terrestrial testing limitations via Model-in-the-Loop architecture

Historically, the validation of landing dynamics relied heavily on physical mock-ups and terrestrial drop tests. However, evaluating the touchdown dynamics of a generic heavy-class lander exclusively through physical Hardware-in-the-Loop (HiL) testing or real-world physical prototypes introduces insurmountable logistical and physical limitations:

- **Gravitational Discrepancy:** Earth's gravitational acceleration is approximately six times greater than that of the Moon ($1/6g \approx 1.62\,\mathrm{m/s^2}$). Drop tests conducted on Earth subject the landing gear to strain rates and structural

loading profiles that do not linearly scale to the lunar environment. While gravity-offload systems are used to simulate lunar gravity, they inherently introduce unmodeled parasitic mass, friction, and pendulum-like dynamic artifacts that corrupt the free-flight multi-body dynamics.

- **Atmospheric Damping:** Terrestrial tests occur within an atmosphere, which introduces aerodynamic drag. In the lunar vacuum, the lander experiences zero aerodynamic damping, meaning that any rotational rate induced by asymmetric thruster firing or an uneven touchdown must be absorbed entirely by the structural joints and the attitude control system.
- **Cost and Destructive Constraints:** Testing the ultimate limits of a landing gear system requires exploring failure boundaries, such as maximum horizontal velocity or worst-case pitch angles. Conducting these tests physically is destructive by nature, requiring multiple expensive mock-ups and making stochastic, large-scale Monte Carlo campaigns economically and practically impossible.

To bridge the gap between pure mathematical theory and physical hardware integration, modern aerospace Systems Engineering relies heavily on the V-Model development cycle.

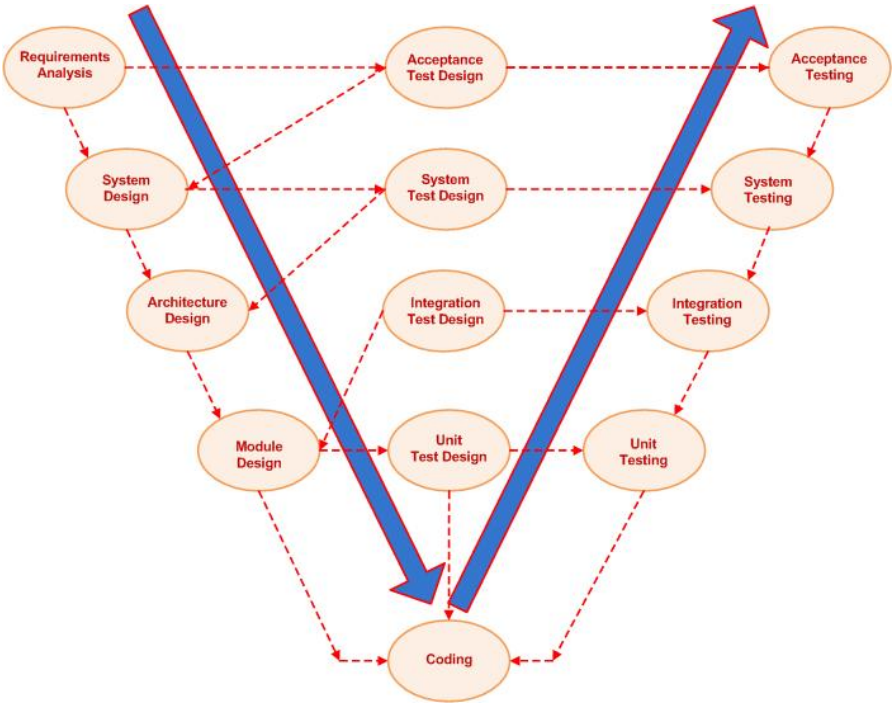


Figure 1.1: V-Model architecture. Source: Wikipedia [1].

At the very foundation of the left side of this V-Model lies the Model-in-the-Loop (MiL) paradigm. A MiL architecture provides a fully virtual, deterministic, and highly observable environment where control algorithms, kinematic systems, and environmental physics are mathematically modeled and coupled in software. For a lunar lander, this framework is critical to dynamically evaluate highly non-linear phenomena safely and cost-effectively before any physical prototype is built.

1.2.2 Thesis Objectives

The goal of this thesis is the design and implementation of an End-to-End (E2E) Model-in-the-Loop (MiL) simulator dedicated to the descent and touchdown dynamics of a generic heavy-class lunar lander. Once developed, the digital twin is rigorously tested through a global sensitivity analysis aiming to identify the critical parameters that govern lander behavior.

In order to achieve the goal, five objectives are defined:

- **Objective 1: Initial State Modeling.** Calculating the vehicle's center of gravity and inertia tensor based on residual propellant loads.
- **Objective 2: Kinematics and Contact Mechanics.** Simulating the multi-link landing gear's kinetic energy dissipation and the contact forces between the footpads and the lunar terrain.
- **Objective 3: Propulsion Integration.** Coupling main engine thrust profiles with the multibody physics to evaluate the direct effect of retro-propulsion on terminal descent rates and attitude.
- **Objective 4: E2E Software Architecture.** Developing an autonomous and scalable simulation framework capable of supporting large-scale parametric campaigns, from the procedural generation of the lander's body and landing gear to the extraction of key performance metrics for touchdown evaluation.
- **Objective 5: Global Sensitivity Analysis.** Systematically sampling the input parameter space to evaluate thousands of simulated landings, thereby quantifying the variables that most critically dictate landing success.

1.3 Thesis Outline

To systematically address the multidisciplinary objectives outlined above, this thesis is structured into five logically sequential chapters, transitioning from theoretical mathematical formulations to the software engineering of the simulation architecture, and concluding with the final simulation campaign.

The content of the manuscript is organized as follows:

Chapter 2 **State of the Art and Theoretical Background:** Reviews lunar descent dynamics, spacecraft landing gear systems, simulation environments, and the foundations of Global Sensitivity Analysis (Morris and Sobol methods).

Chapter 3 **Mathematical and Physical Modelling:** Formalizes dynamic mass and inertia tracking for propellant tanks, plume surface interactions (PSI), landing leg kinematics with crushable shock absorbers, and dual-layer regolith contact mechanics.

Chapter 4 **Software Architecture and Simulation (MiL):** Details the ROS 2/Gazebo framework, including procedural generation of the lander and lunar mesh, custom C++ plugins and automated telemetry logging.

Chapter 5 **Sensitivity Analysis and Conclusion:** Presents the execution of the simulation campaigns, utilizing sequential Morris screening and Sobol variance-based methods to evaluate parameter influence across global and fixed-topography scenarios.

Chapter 2

State of the Art and Theoretical Background

This chapter provides the technical and scientific background necessary to contextualize the development of the simulator. First, the fundamentals of descent dynamics in the lunar environment will be reviewed, and current technologies for spacecraft landing systems will be analyzed. Subsequently, the choice of the software architecture will be justified by comparing modern aerospace simulation environments, with a particular focus on ROS2 and Gazebo. Finally, the principles of Global Sensitivity Analysis will be introduced.

2.1 Lunar Lander Dynamics

As established in Section 1.2, the lunar environment imposes unique constraints on spacecraft dynamics. The absolute vacuum dictates that aerodynamic forces are non-existent, so the vehicle relies entirely on its active propulsion system to arrest its orbital velocity and manage its attitude. Simultaneously, the reduced lunar gravity ($1/6g$) significantly alters the system's response to impact forces, exacerbating the risk of rebounds and tip-over instabilities upon touchdown.

To accurately capture these phenomena during the terminal descent phase prior to mechanical contact, the lander is classically modeled as a 6-Degree-of-Freedom (6-DoF) rigid body. Its translational and rotational motions are governed by the standard Newton-Euler equations:

$$\sum \mathbf{F} = m\ddot{\mathbf{r}} = \mathbf{F}_{thrust} + m\mathbf{g}_{moon}$$

$$\sum \boldsymbol{\tau} = \mathbf{I}\dot{\boldsymbol{\omega}} + \boldsymbol{\omega} \times (\mathbf{I}\boldsymbol{\omega}) = \boldsymbol{\tau}_{thrust}$$

Where m is the instantaneous mass of the vehicle, $\mathbf{r}$ is the position vector in the inertial frame, $\mathbf{g}_{moon}$ is the lunar gravity vector, $\mathbf{I}$ is the inertia tensor, and $\boldsymbol{\omega}$ is the angular velocity vector. A lunar lander is a highly time-variant system; the continuous consumption of propellant rapidly alters both the total mass m and the inertia tensor $\mathbf{I}$. This shifting of the center of gravity directly couples the translational and rotational dynamics, requiring continuous compensation from the GNC systems.

The final, and most critical, phase of the landing trajectory is the *terminal descent*, which culminates in the touchdown. The objective of this phase is to achieve a "soft landing". A soft landing is defined in aerospace engineering as a touchdown event where the residual kinetic energy is strictly contained within the structural limits of the vehicle, ensuring the safety of the payload and the overall integrity of the spacecraft.

To guarantee a soft landing and prevent structural failure or tip-over, strict kinematic constraints must be met at the moment of contact:

- **Vertical Velocity:** Must be minimized to avoid exceeding the energy-absorption capacity of the landing gears.
- **Horizontal Velocity:** Must be completely nullified. Residual lateral velocity generates transverse loads on the landing struts and induces overturning moments upon ground contact.
- **Attitude:** The pitch and roll angles must be closely aligned with the local vertical. This ensures that the landing legs engage the lunar terrain as symmetrically as possible, distributing the impact forces evenly across the structure.

Once the first footpad makes contact with the lunar regolith, the pure free-flight 6-DoF rigid body assumption ceases to be sufficient. The system abruptly transitions into a complex multibody dynamics problem governed by impact mechanics, friction, and kinematic deployment.

2.2 Spacecraft Landing Gear Systems

The primary function of a spacecraft landing gear system is to safely dissipate the residual kinetic energy at touchdown, ensuring that the impact loads transmitted to the vehicle's main bus remain well within acceptable structural limits. Furthermore, the lander's legs must also ensure stability after landing to prevent tipping over and to handle the irregularities of the lunar terrain.

To achieve these objectives, landing gears typically rely on specific energy absorption mechanisms installed in the primary or secondary struts. The state

of the art in aerospace energy attenuation primarily identifies two technological solutions:

- **Crushable Honeycomb Structures:** This is the most established technology, historically utilized in the Apollo Lunar Module and many subsequent robotic missions. These systems rely on the irreversible plastic deformation of aluminum honeycomb cartridges enclosed within the landing struts. While they offer an excellent energy-to-mass absorption ratio and high reliability due to the absence of moving fluid parts, they are strictly single-use. The fixed stroke and constant yield force cannot be actively adapted to varying impact conditions.
- **Oleodynamic and Oleo-pneumatic Dampers:** More common in terrestrial aviation, these systems are gaining traction for modern heavy and reusable lunar landers. They dissipate energy through the viscous friction of a fluid forced through an orifice, often combined with a compressed gas acting as an elastic spring. Unlike honeycombs, oleo-pneumatic dampers provide variable resistance proportional to the impact velocity and are fully reusable. However, they introduce significant complexity regarding mass, fluid sealing in a vacuum, and temperature management in the extreme lunar environment.

Beyond the specific damping technology, spacecraft landing gears can be broadly classified into passive and active (or semi-active) mechanisms, depending on their kinematic architecture and control paradigm.

Passive systems have a fixed deployment geometry and rely entirely on the mechanical properties of their shock absorbers to manage the impact. They are the standard for current space exploration due to their simplicity and robustness. Active landing gears, on the other side, incorporate sensors and actuators to dynamically adjust the main properties of the struts (such as: length, damping coefficients, or stiffness). This adaptability allows the lander to actively level its payload deck on steep slopes or optimize energy absorption based on the exact state vector at touchdown.

The design of a modern large logistics lander often requires evaluating these trade-offs to select the optimal combination of shock attenuation and kinematic deployment, necessitating highly accurate multi-body simulations to validate the structural response before manufacturing.

2.3 Simulation Environments in Aerospace

For decades, aerospace simulations have depended heavily on monolithic tools. MATLAB and Simulink, for instance, remain the undisputed leaders for analytical

control design and 1D dynamics, while MSC Adams represents the standard for pure multi-body kinematics. However, as the aerospace sector rapidly shifts toward software-defined systems, a major bottleneck arises when linking these legacy platforms with today's distributed flight software. Traditional tools struggle to effectively support sensor-rich and decentralized architectures. To overcome this limitation, the space robotics sector is increasingly adopting the Robot Operating System 2 (ROS 2) [2] paired with the Gazebo simulation environment [3].

ROS 2 provides a decentralized, node-based middleware architecture built upon the Data Distribution Service (DDS) standard. This modularity is particularly advantageous for spacecraft design, as it allows for the independent development and asynchronous execution of various subsystems, such as thruster actuation, state estimation, and telemetry logging. By structuring the MiL simulator in ROS 2, the developed control logic and communication interfaces closely mirror the final flight software, significantly smoothing the future transition to Hardware-in-the-Loop (HiL) testing.

While ROS 2 manages the software architecture, the physical world is simulated by Gazebo, a high-fidelity 3D environment designed to accurately replicate rigid-body dynamics, sensor data generation, and complex kinematic structures. It interfaces seamlessly with ROS 2, acting as the physical plant in the MiL architecture.

By using the combination of ROS2 and Gazebo, it is possible to create a simulation framework capable of evaluating both the logical behavior of the lander and the highly non-linear physical phenomena characterizing a lunar touchdown.

2.4 Global Sensitivity Analysis for Soft Landing

Achieving a successful lunar soft landing is not merely a matter of designing a system that survives a nominal, perfectly executed trajectory. In real-world space missions, the terminal descent phase is inherently stochastic, because it's subject to multiple sources of uncertainty. To robustly evaluate the reliability of the landing system and identify the critical parameters that govern its dynamics, Global Sensitivity Analysis (GSA) methods are employed. However, doing a GSA on highly non-linear and computationally expensive simulation model is often unaffordable. To overcome this issue, a sequential approach is recommended [4]: utilizing the Morris screening method followed by a Variance-Based Sensitivity Analysis (Sobol method) provides a comprehensive framework for uncertainty quantification.

2.4.1 The Morris Method (Method of Elementary Effects)

Unlike standard Monte Carlo approaches (which rely on pure pseudo-random sampling) the Morris [5] method operates on a structured, pre-calculated multidimensional grid. A fundamental prerequisite for this Design of Experiments is the absolute statistical independence of the input variables.

The sampling strategy implies the discretization of the input space of the k parameters into a p-level grid. Each parameter X_i is mapped from its physical bounds into the dimensionless domain $[0, 1]$. The algorithm explores this domain through a sequence of independent spatial trajectories. From a random base point, the algorithm performs a One-At-A-Time (OAT) random walk: the simulator moves to a new state by perturbing a single parameter X_i by a fixed amount Δ, while keeping all other parameters constant.

After each step, the Elementary Effect (EE) for the perturbed parameter X_i on a specific output Y is computed as a finite difference:

$$EE_i = \frac{Y(X_1, X_2, \ldots, X_i + \Delta, \ldots, X_k) - Y(X_1, X_2, \ldots, X_k)}{\Delta} \tag{2.1}$$

Once the Elementary Effects are collected across r independent trajectories, two global sensitivity indices are computed for each parameter to evaluate its influence on the system:

- **Global Importance (μ^*):** The mean of the absolute values of the Elementary Effects. It quantifies the overall direct influence of the parameter on the output.

$$\mu_j^* = \frac{1}{r} \sum_{t=1}^{r} \left| EE_j^{(t)} \right| \tag{2.2}$$

- **Non-Linearity and Interaction (σ):** The standard deviation of the Elementary Effects. A high value indicates that the parameter's influence is either highly non-linear or strongly dependent on the concurrent values of other variables.

$$\sigma_j = \sqrt{\frac{1}{r} \sum_{t=1}^{r} \left(EE_j^{(t)} - \mu_j \right)^2} \tag{2.3}$$

The Morris method acts as a qualitative screening tool to identify the most critical drivers of the touchdown dynamics, allowing for a drastic reduction of the problem's dimensionality by isolating and fixing non-influential parameters.

2.4.2 Variance-Based Sensitivity Analysis

The next step is the Variance-Based Sensitivity Analysis (VBSA) using the Sobol method [6], which provides a quantitative evaluation of the exact fraction of the

model's output variance attributable to each input parameter and their mutual interactions.

Given a highly non-linear model defined as a function $Y = f(X_1, X_2, \ldots, X_k)$ over a k-dimensional unit hypercube, the total unconditional variance of the output, $V(Y)$, can be perfectly decomposed into a sum of partial variances of increasing dimensionality:

$$V(Y) = \sum_{i=1}^{k} V_i + \sum_{i<j} V_{ij} + \cdots + V_{1,2,\ldots,k} \tag{2.4}$$

where $V_i = V_{X_i}(E_{\mathbf{X}_{\sim i}}(Y|X_i))$ represents the partial variance contributed by the i-th parameter alone, and $V_{ij} = Var_{X_{ij}}(E_{X_{\sim ij}}(Y|X_i, X_j)) - V_i - V_j$ represents the variance contributed by the interaction between X_i and X_j.

By normalizing these partial variances by the total variance $V(Y)$, two fundamental sensitivity indices can be computed:

- **First-Order Index (S_i):** Defines the main, isolated effect of a parameter. It measures the expected reduction in the output variance if the true value of X_i could be perfectly known and fixed.

$$S_i = \frac{V_i}{V(Y)} \tag{2.5}$$

- **Total-Order Index (S_{T_i}):** Represents the total effect of a parameter, including both its first-order contribution and all its interactions of any order with all other input variables.

$$S_{T_i} = 1 - \frac{V_{\mathbf{X}_{\sim i}}(E_{X_i}(Y|\mathbf{X}_{\sim i}))}{V(Y)} \tag{2.6}$$

The comparison between S_i and S_{T_i} is critical , a low S_i combined with a high S_{T_i} reveals a parameter that is harmless in isolation but triggers catastrophic failures when coupled with specific adverse conditions (such as a "trip-wire" effect where lateral velocity interacts with uneven topography).

To evaluate these multi-dimensional integrals efficiently, an advanced Monte Carlo integration technique is required. While Ilya Sobol developed the foundational mathematical theory of variance decomposition, it was Andrea Saltelli and his colleagues [7] who formalized the most computationally efficient sampling strategy to estimate these indices numerically. Saltelli evaluated different approaches for calculating the Sobol indices via Monte Carlo simulations and identified the most optimal strategy in terms of convergence rate. His approach relies on a clever manipulation of sample matrices to minimize the number of model evaluations required.

The process begins by generating two completely independent matrices of dimension $N \times k$, where N is the base sample size and k is the number of input parameters:

- **Matrix A (Sample Matrix):** A randomly generated matrix using a Low-Discrepancy Sequence (such as the Sobol sequence) to explore the input space.
- **Matrix B (Re-sample Matrix):** A second independent matrix generated using the same sequence.

From these two base matrices, a set of k additional hybrid matrices, denoted as $\mathbf{C}_i$ (for $i = 1, 2, \ldots, k$), is constructed. Each $\mathbf{C}_i$ matrix is completely identical to matrix $\mathbf{A}$, except for its i-th column, which is entirely substituted with the i-th column of matrix $\mathbf{B}$.

From an engineering perspective, this means that between matrix $\mathbf{A}$ and matrix $\mathbf{C}_i$, all parameters are kept strictly "frozen" at their $\mathbf{A}$ conditions, except for the i-th parameter, which takes a completely new, independent value from $\mathbf{B}$. Finally, the Model-in-the-Loop simulation evaluates the touchdown dynamics for each row of the $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}_i$ matrices. Using these evaluated outputs, the numerical estimators for the First-Order and Total-Order indices are computed as follows:

$$S_i = \frac{\frac{1}{N}\sum_{j=1}^{N} Y(\mathbf{B})_j Y(\mathbf{C}_i)_j - f_0^2}{V(Y)} \tag{2.7}$$

$$S_{T_i} = \frac{\frac{1}{2N}\sum_{j=1}^{N} \left(Y(\mathbf{A})_j - Y(\mathbf{C}_i)_j\right)^2}{V(Y)} \tag{2.8}$$

where $Y(\mathbf{A})_j$, $Y(\mathbf{B})_j$ and $Y(\mathbf{C}_i)_j$ represent the scalar outputs for the j-th row of their respective matrices, f_0^2 is the mean squared response of the dataset, and N is the base sample size.

Chapter 3

Mathematical and Physical Modeling

This chapter formalizes the mathematical and physical models at the core of the simulation. It begins by modeling the rigid-body dynamics of the lunar lander, alongside an outline of expected system disturbances. The focus then shifts to the landing gear, detailing both the kinematics and dynamics of the legs to ultimately establish the specific contact model used for surface interaction.

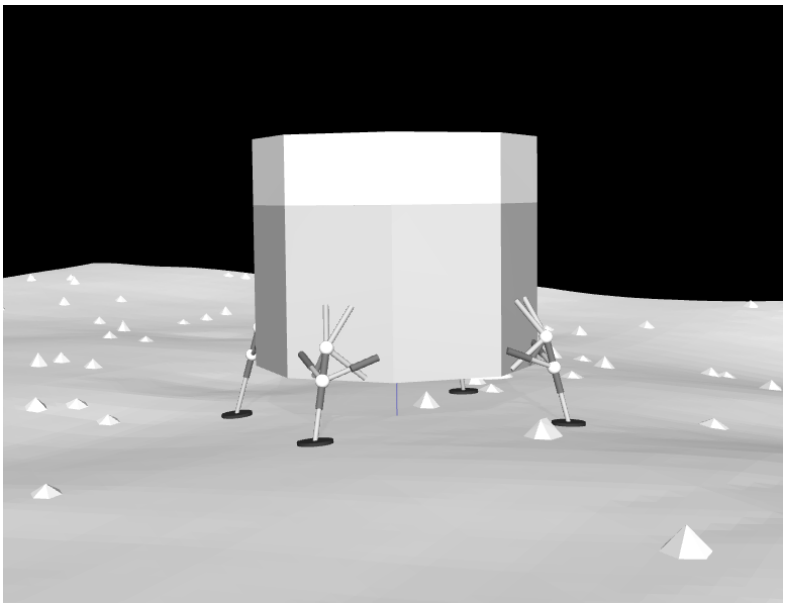


Figure 3.1: Visual representation of the full lander (Body + Passenger + legs) within the Gazebo simulation environment.

3.1 Lander Modeling

The lander is modeled as a multibody system composed of distinct macroscopic rigid elements. From the software and physical partitioning perspective within the model description files (SDF - *Simulation Description Format*), the architecture is mainly divided into the primary *Body Link* (comprising the empty structure of the body and the propellant tanks) and the *Passenger Link* (associated with the useful payload).

The total mass at the beginning of the scenario, defined as the *Landing Mass* (M_L), is expressed by the following balance equation:

$$M_L = M_{dry} + M_{passenger} + \Delta M_P \tag{3.1}$$

where M_{dry} represents the structural mass of the lander without propellant, $M_{passenger}$ is the mass of the payload, and ΔM_P defines the total mass of residual propellant available for the landing phase.

3.1.1 Propellant Ripartition

The propulsion system adopts a liquid bipropellant configuration using Monomethylhydrazine (MMH) as fuel and Nitrogen Tetroxide (MON) as oxidizer. Defining the design mixture ratio as $\alpha = m_{MON}/m_{MMH}$, the total mass fractions for each fluid are derived as:

$$x_{MON} = \frac{\alpha}{1+\alpha}, \quad x_{MMH} = 1 - x_{MON} \tag{3.2}$$

The lander allocates the propellant within four distinct tanks arranged symmetrically around the vertical axis of the body, divided into pairs:

- **Tanks 1 and 4:** Containing MMH fuel, with a mass per single tank of $m_{t,MMH} = (\Delta M_P \cdot x_{MMH})/2$;
- **Tanks 2 and 3:** Containing MON oxidizer, with a mass per single tank of $m_{t,MON} = (\Delta M_P \cdot x_{MON})/2$.

3.2 Tank and Propellant Properties

Having established the mass of propellant allocated to each unit, it is necessary to model their internal geometry to determine the spatial distribution of the fluid. The propellant tanks are modeled as geometrically identical axisymmetric solids of revolution, oriented along the vertical z-axis. The internal useful volume is bounded by three distinct geometric regions: a lower spherical dome (*bottom dome*), a central cylindrical shell, and an upper spherical dome (*top dome*). The fluid inside is considered continuous, homogeneous, incompressible, and of constant density ρ.

3.2.1 Geometric Parameterization of the Tank

To accurately model the fluid geometry, a local reference frame is defined for each individual tank. Assuming the origin $z = 0$ is at the geometric center of the central cylinder, the volume is described by the following macroscopic parameters:

- R_{cyl}: radius of the cylindrical section;
- L_{cyl}: length of the cylindrical section;
- H_{dome}: height or dome depth of the spherical end caps.

The ideal junction planes between the cylindrical body and the spherical caps are located at the vertical coordinates:

$$z_1 = -\frac{L_{cyl}}{2}, \quad z_2 = +\frac{L_{cyl}}{2} \tag{3.3}$$

The minimum and maximum geometric bounds of the tank are therefore confined to:

$$z_{bottom} = z_1 - H_{dome}, \quad z_{top} = z_2 + H_{dome} \tag{3.4}$$

3.2.2 Radius of the Generating Spherical Cap

To analytically model the curvature of the tank ends, the radius R_s of the ideal sphere generating the caps is derived. Considering the meridian section of the cap, a right triangle is identified having the spherical radius R_s as its hypotenuse, the cylindrical radius R_{cyl} as its base leg, and the vertical distance between the center of the sphere and the junction plane, equal to $(R_s - H_{dome})$, as its vertical leg. Applying the Pythagorean Theorem yields:

$$R_s^2 = R_{cyl}^2 + (R_s - H_{dome})^2 \tag{3.5}$$

Expanding the expression and isolating the term of interest, the fundamental relation is obtained:

$$R_s = \frac{R_{cyl}^2 + H_{dome}^2}{2H_{dome}} \tag{3.6}$$

The distance between the junction plane and the center of the generating sphere is defined as $d_{cap} = R_s - H_{dome}$. Consequently, the centers of the spheres describing the lower cap ($z_{c,bottom}$) and upper cap ($z_{c,top}$) are located along the z-axis at the coordinates:

$$z_{c,bottom} = z_1 + d_{cap}, \quad z_{c,top} = z_2 - d_{cap} \tag{3.7}$$

3.2.3 Local Radius Profile Function

Utilizing the analytical formulation of the Cartesian sphere translated at z_c, namely $x^2 + y^2 + (z - z_c)^2 = R_s^2$, and imposing axisymmetry such that the local radius squared equals $r(z)^2 = x^2 + y^2$, the external geometric profile of the tank is explicitly defined by the following piecewise function:

$$r(z) = \begin{cases} \sqrt{R_s^2 - (z - z_{c,bottom})^2} & \text{if } z_{bottom} \leq z < z_1 \\ R_{cyl} & \text{if } z_1 \leq z \leq z_2 \\ \sqrt{R_s^2 - (z - z_{c,top})^2} & \text{if } z_2 < z \leq z_{top} \end{cases} \tag{3.8}$$

3.2.4 Iterative Determination of Fluid Level (h_{fill})

The total analytical volume of the tank is calculated by summing the cylindrical contribution and the two terminal caps:

$$V_{tot} = \pi R_{cyl}^2 L_{cyl} + \frac{2}{3}\pi H_{dome}^2 (3R_s - H_{dome}) \tag{3.9}$$

Given a target mass of propellant confined within a single tank m_{target}, the respective volume occupied by the fluid is $V_{target} = m_{target}/\rho$. The effective height of the fluid free surface, denoted by the coordinate h_{fill}, is linked to the volume by the integral relation:

$$V_{target} = \int_{z_{bottom}}^{h_{fill}} A(z)\, dz = \int_{z_{bottom}}^{h_{fill}} \pi r(z)^2\, dz \tag{3.10}$$

Due to the piecewise nature of the local radius function $r(z)$ (3.8), direct analytical inversion of equation (3.10) to find h_{fill} is computationally impractical in real time. The problem is thus solved numerically by implementing the *Bisection Method*.

The algorithm operates by bounding the root within the physical domain of the liquid column $[z_{low}, z_{high}] = [z_{bottom}, z_{top}]$. At each iteration k, the midpoint coordinate $z_{mid} = 0.5(z_{low} + z_{high})$ is computed, and the associated partial volume $V(z_{mid})$ is evaluated via numerical integration using the trapezoidal rule over a finely discretized domain:

$$V(z_{mid}) \approx \sum_{i=1}^{N_{steps}} \frac{A(z_i) + A(z_{i+1})}{2} \Delta z \tag{3.11}$$

where $A(z) = \pi r(z)^2$ is the instantaneous circular cross-section evaluated dynamically along the piecewise profile. The residual error is computed as $e_k = |V_{target} - V(z_{mid})|$. If $V(z_{mid}) < V_{target}$, the lower bound is updated by setting $z_{low} = z_{mid}$; otherwise, $z_{high} = z_{mid}$ is set. The loop converges to the correct height $h_{fill} = z_{mid}$ when the error falls below the preset tolerance $\epsilon = 10^{-6}$.

3.2.5 Discretization and Calculation of Local Inertial Properties

Once the boundary coordinate h_{fill} is determined, it serves as the upper boundary for the physical integration domain. The actual fluid body is discretized along the vertical axis into N thin disks of thickness Δz. The elemental mass of the generic disk located at coordinate z_i is given by:

$$\Delta m_i = \rho \, \pi r(z_i)^2 \Delta z \tag{3.12}$$

The local vertical barycentric coordinate of the fluid inside the tank ($z_{cg,local}$) is derived through the weighted sum of these discrete mass elements:

$$z_{cg,local} = \frac{\sum_{i=1}^{N} z_i \, \Delta m_i}{\sum_{i=1}^{N} \Delta m_i} \tag{3.13}$$

The inertia tensor of the fluid is estimated by integrating the contributions of the individual elemental disks. The intrinsic barycentric inertia of a thin disk is:

$$\Delta I_{zz,i} = \frac{1}{2} \Delta m_i \, r(z_i)^2, \quad \Delta I_{xx,i} = \Delta I_{yy,i} = \frac{1}{4} \Delta m_i \, r(z_i)^2 \tag{3.14}$$

where $\Delta I_{xx,i}$ and $\Delta I_{yy,i}$ represent the inertia with respect to the planar orthogonal axes of rotation x and y.

Applying the local parallel axis theorem to transport the disk contributions from their height coordinate z_i to the local fluid center of mass $z_{cg,local}$, the transport distance is defined as $\Delta z_{cg,i} = z_i - z_{cg,local}$. The final inertia components of the propellant relative to its own local barycentric axis take the form:

$$I_{xx,local} = I_{yy,local} = \sum_{i=1}^{N} \left(\Delta I_{xx,i} + \Delta m_i \, \Delta z_{cg,i}^2 \right) \tag{3.15}$$

$$I_{zz,local} = \sum_{i=1}^{N} \Delta I_{zz,i} \tag{3.16}$$

The components of the off-diagonal products of inertia vanish ($I_{xy} = I_{xz} = I_{yz} = 0$) due to the perfect radial symmetry of the solid of revolution. The local barycentric inertia tensor of the fluid is thus assembled as a diagonal matrix $\mathbf{I}_{tank} = \text{diag}(I_{xx}, I_{yy}, I_{zz})$. Ultimately, both this inertial matrix and the local center of gravity coordinate ($z_{cg,local}$) are used to compute the Body Link CoG and Inertia.

3.3 Body Link Assembly and Tensor Transport

3.3.1 Body Center of Gravity Calculation

All spatial coordinates are expressed relative to the local Body Reference Frame (RF_1), whose origin is located at the geometric center of the lander's base. Each

tank j (where $j = 1, \ldots, 4$) is positioned within the structure at a fixed coordinate defined by the vector $\mathbf{r}_{tank,j} = [x_j, y_j, 0]^T$. During flight, the vertical coordinate of the fluid-only center of mass within each individual tank ($z_{cg,j}$) varies as a function of depletion, shifting the local center of gravity of the fluid to $\mathbf{r}_{cg,f,j} = [x_j, y_j, z_{cg,j}]^T$.

The global Center of Gravity of the *Body Link* alone ($M_{body} = M_{dry} + \Delta M_P$) is computed by applying the weighted average of static moments:

$$\mathbf{r}_{CoG,body} = \frac{M_{dry}\mathbf{r}_{dry} + m_{t,MMH}(\mathbf{r}_{cg,f,1} + \mathbf{r}_{cg,f,4}) + m_{t,MON}(\mathbf{r}_{cg,f,2} + \mathbf{r}_{cg,f,3})}{M_{dry} + \Delta M_P} \tag{3.17}$$

where $\mathbf{r}_{dry} = [x_{dry}, y_{dry}, z_{dry}]^T$ identifies the constant center of mass of the dry structure.

3.3.2 Inertia Tensor Transport via the Parallel Axis Theorem

To determine the global inertia tensor of the body with respect to its moving CoG, it is necessary to translate the inertia tensors of the individual constituent elements. Let us define the generalized tensor formulation of the *Parallel Axis Theorem* (or Huygens-Steiner Theorem) for translating an inertia tensor from a local barycentric system to a new reference point separated by the relative position vector $\mathbf{d} = \mathbf{r}_{old} - \mathbf{r}_{new}$:

$$\mathbf{I}_{new} = \mathbf{I}_{local} + m\left(\|\mathbf{d}\|^2\mathbf{E}_{3\times3} - \mathbf{d} \otimes \mathbf{d}\right) \tag{3.18}$$

where $\mathbf{E}_{3\times3}$ represents the identity matrix and $\otimes$ denotes the Kronecker outer product. Expanding the symmetric transport matrix (3.18) yields:

$$\mathbf{I}_{new} = \mathbf{I}_{local} + m\begin{bmatrix} (y^2+z^2) & -xy & -xz \\ -xy & (x^2+z^2) & -yz \\ -xz & -yz & (x^2+y^2) \end{bmatrix} \tag{3.19}$$

The total inertia tensor of the *Body Link* ($\mathbf{I}_{body,tot}$) is thus assembled by summing the individual translated contributions with respect to the combined global center of mass $\mathbf{r}_{CoG,body}$:

$$\mathbf{I}_{body,tot} = \mathbf{I}_{dry,trans} + \sum_{j=1}^{4} \mathbf{I}_{tank,j,trans} \tag{3.20}$$

Each term in the summation is obtained by calculating the distance vector $\mathbf{d}_j = \mathbf{r}_{cg,f,j} - \mathbf{r}_{CoG,body}$ and applying the tensor translation starting from the local fluid inertial properties contained in $\mathbf{I}_{tank,j}$.

3.4 System Disturbances: Plume Surface Interaction

During the final phase of the terminal descent, the lander enters a critical operational regime strongly influenced by the proximity to the lunar surface. The high-velocity expansion of the propulsive exhaust gases in a vacuum environment inevitably impacts the regolith, generating a complex fluid dynamic phenomenon known as Plume Surface Interaction (PSI) or lunar ground effect.

The fundamental theory governing this interaction was first extensively modeled by Leonard Roberts [8] during the Apollo program in the late 1960s. Roberts demonstrated that a supersonic jet impinging on a granular surface in a vacuum induces localized cratering, fluidization of the soil, and a severe outward flow of gas and dust. This expanding dust cloud and the associated back-pressure exert stochastic, multi-axial aerodynamic forces on the vehicle's base structure, which can introduce attitude instabilities and translation drifts just moments before touchdown.

To accurately evaluate the robustness of the landing gear system and the GNC loops within the simulation environment, an empirical mathematical model has been formulated to dynamically generate these disturbances.

3.4.1 Disturbance Force Magnitude formulation

The total magnitude of the ground effect disturbance, denoted as F_{tot}, strongly increases non-linearly as the vehicle approaches the surface. The phenomenon is considered to be negligible at altitudes strictly greater than 10 m.

For $h < 10$ m, the force is approximated using a second-order polynomial function derived from empirical data interpolation. The unconstrained disturbance profile is defined as:

$$F_{calc}(h) = ah^2 + bh + c \tag{3.21}$$

where the coefficients are set to $a = 2.587$, $b = -41.800$, and $c = 159.333$. To ensure the physical consistency of the simulation and prevent anomalous impulse peaks, the actual applied magnitude F_{tot} is mathematically saturated between a minimum of 0 N and a maximum structural threshold of 500 N:

$$F_{tot}(h) = \begin{cases} 0 & \text{if } h \geq 10 \text{ m} \\ \max(0, \min(F_{calc}(h), 500)) & \text{if } h < 10 \text{ m} \end{cases} \tag{3.22}$$

3.4.2 Spatial and Temporal Distribution

The actual PSI disturbance is rarely a purely vertical force. The chaotic nature of the back-pressure and the uneven surface of the regolith induce severe unsteadiness,

resulting in oscillating lateral forces. To model this erratic behavior, the total magnitude F_{tot} is spatially decomposed into four cardinal planar components (North, East, South, West) using a sinusoidal distribution coupled with a specific oscillation frequency $\omega = 3.0$ rad/s.

Defining t as the time elapsed since the activation of the disturbance model, the instantaneous directional forces are assigned as follows:

$$\begin{aligned} F_{North}(t) &= F_{tot}(h) \cdot \max(0, \sin(\omega t)) \\ F_{East}(t) &= F_{tot}(h) \cdot \max(0, \cos(\omega t)) \\ F_{South}(t) &= F_{tot}(h) \cdot \max(0, -\sin(\omega t)) \\ F_{West}(t) &= F_{tot}(h) \cdot \max(0, -\cos(\omega t)) \end{aligned} \tag{3.23}$$

The utilization of the maximum function acts as a half-wave rectifier, ensuring that the components alternatingly apply positive, non-overlapping pulses of force along the respective axes. This formulation successfully synthesizes a rotating, swirling force vector that contributes to modeling the actual physical behavior of the system.

3.5 Kinematics and Dynamics of Landing Legs

This section details the spatial topology, coordinate transformations, and geometric relations defining the landing legs, alongside the analytical model used to represent the shock absorbers.

3.5.1 Coordinate Systems and Reference Frames

To systematically derive the position and orientation of each link, two primary Cartesian reference frames are introduced:

- **Lander Reference Frame ($\mathcal{F}_B$):** The global coordinate system of the lander coincides with the Body Reference Frame (RF_1). Its origin $O_B = (0,0,0)_B$ is rigidly attached to the geometric center of the bottom base of the octagonal main body. The Z_B-axis points upwards along the lander's vertical axis of symmetry, while X_B and Y_B define the horizontal structural plane.
- **Leg Local Reference Frame ($\mathcal{F}_L$):** A local coordinate system defined for each of the four landing legs. To simplify the kinematic closure equations, the origin of this frame is positioned at the fixed structural node C_1, such that $C_1 = (0,0,0)_L$.

The spatial transformation from the local leg frame $\mathcal{F}_{L,i}$ (for the i-th leg) to the global body frame $\mathcal{F}_B$ is described by a homogeneous transformation matrix

$\mathbf{T}_i \in SE(3)$. Given the octagonal symmetry, the legs are radially distributed at 90-degree intervals. The position of each leg assembly is determined by an absolute radial offset (the octagon radius, R_{oct}) and a vertical offset Z_{offset}. For a generic leg rotated by an angle $\psi_i = (i-1)\frac{\pi}{2}$ around the Z_B-axis, the transformation of a generic point P from local to global coordinates is computed as:

$$\begin{bmatrix} P_{x,B} \\ P_{y,B} \\ P_{z,B} \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\psi_i) & -\sin(\psi_i) & 0 & X_{disp} \\ \sin(\psi_i) & \cos(\psi_i) & 0 & Y_{disp} \\ 0 & 0 & 1 & Z_{offset} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} P_{x,L} \\ P_{y,L} \\ P_{z,L} \\ 1 \end{bmatrix} \tag{3.24}$$

where X_{disp} and Y_{disp} represent the planar displacement derived from the structural mounting points on the octagonal bus.

3.5.2 Topological Description of the Landing Gear

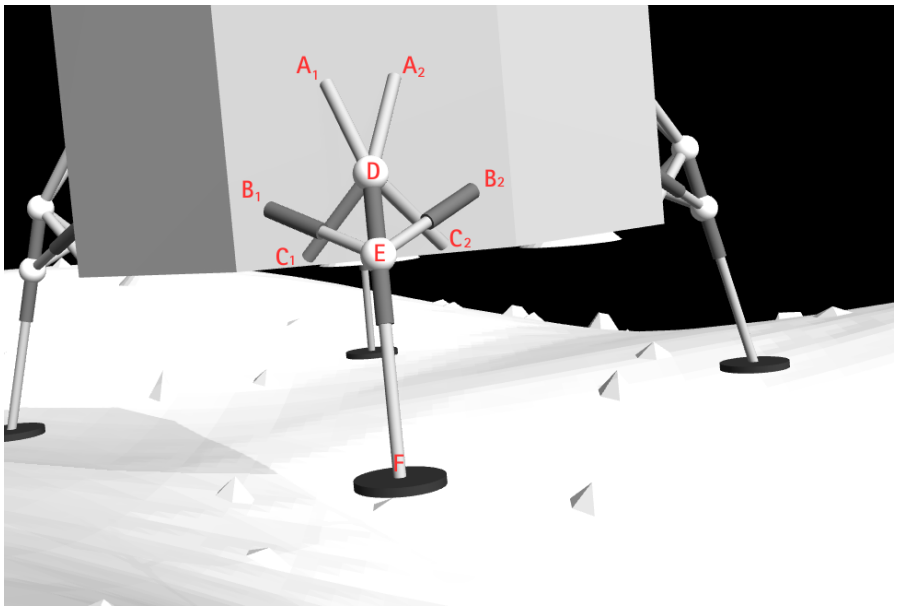


Figure 3.2: Kinematic topology of the landing gear: visual representation of the structural nodes and joints within the Gazebo simulation environment.

Each landing leg constitutes a complex spatial mechanism designed to absorb the kinetic energy during lunar impact. The system can be divided into fixed rigid links (trusses) and variable-length kinematic chains (shock absorbers). In the local frame $\mathcal{F}_L$, the topology is defined by a set of static nodes:

- **Base Nodes:** C_1, C_2, A_1, A_2. These points represent the rigid attachment joints interfacing the leg with the lander's primary structure.

- **Lateral Strut Mounts:** B_1, B_2. Hardpoints for the secondary stabilizing pistons.
- **Hip Joint (Node D):** The central convergence point of the fixed trusses, acting as the primary pivot for the main shock absorber.
- **Piston Intermediate Node (Node E):** The connection point located along the central piston axis (between nodes D and F), serving as the attachment hardpoint for the lateral stabilizing shock absorbers $B_1 - E$ and $B_2 - E$.
- **Footpad Joint (Node F):** The distal extremity of the leg where the foot is mounted.

3.5.3 Kinematics of Fixed Trusses

The fixed structure consists of four rigid cylindrical links connecting the base nodes (C_1, C_2, A_1, A_2) to the hip joint D. To instantiate these elements in the physics engine, their length, spatial centroid, and Euler orientations must be explicitly derived from the node coordinates. Let $P_{start} = (x_s, y_s, z_s)$ be a generic base node and $P_{end} = D = (x_e, y_e, z_e)$ be the target node.

The physical length L of the link is evaluated via the Euclidean norm:

$$L = \|P_{end} - P_{start}\| = \sqrt{(x_e - x_s)^2 + (y_e - y_s)^2 + (z_e - z_s)^2} \tag{3.25}$$

In standard multi-body simulation environments (such as Gazebo via SDF formatting), a primitive geometry is defined relative to its geometric center. Therefore, the positional origin P_c of the link is the midpoint of the spatial vector:

$$P_c = \left(\frac{x_s + x_e}{2}, \frac{y_s + y_e}{2}, \frac{z_s + z_e}{2} \right) \tag{3.26}$$

The orientation of the link is determined by two Euler angles: Yaw (ψ) and Pitch (θ). Assuming the default orientation of a cylinder aligns with the global Z-axis, the necessary spatial rotations to align the cylinder with the vector $\mathbf{v} = P_{end} - P_{start}$ are:

$$\psi = \text{atan2}(v_y, v_x) \tag{3.27}$$

$$\theta = \arccos\left(\frac{v_z}{L} \right) \quad (\text{for } L \neq 0) \tag{3.28}$$

This formulation ensures that the rigid spatial frame connects the lander body to the primary pivot D without introducing pre-stresses in the simulation environment.

3.5.4 Central and Lateral Shock Absorbers

The primary energy absorption mechanism of the landing gear relies on a central telescopic assembly. This core element links node D to the footpad F. Two lateral stabilizing struts provide supplementary structural integrity by connecting the base points B_1 and B_2 to an intermediate node, E.

To properly define the attachment points for these secondary pistons, the exact spatial location of the auxiliary node E must first be established. Node E lies at the intersection of the spatial segment $D - F$ and the horizontal plane $z = 0$ within the local leg frame. By applying the parametric equation of a line, $P(t) = D + t(F - D)$, we analytically extract the intersection parameter t_E evaluated at the $z = 0$ boundary:

$$t_E = \frac{0.0 - z_D}{z_F - z_D} \tag{3.29}$$

Once t_E is isolated, it is substituted back into the parametric formulation. This straightforward operation yields the precise Cartesian coordinates of the intermediate attachment point E:

$$E_x = x_D + t_E(x_F - x_D), \quad E_y = y_D + t_E(y_F - y_D), \quad E_z = 0.0 \tag{3.30}$$

Both the central and lateral shock absorber share a unified approach to compute their kinematics and mass distributions. For any generic telescopic shock absorber bounded by coordinates p_{start} and p_{end}, the total extended length L_{total} and the spatial orientation angles (ψ, θ) are computed using the exact same vector formulations previously defined in Eq. 3.25 and Eq. 3.28.

To accurately model the telescopic mechanics without inducing static compenetration, the physical geometries of the outer strut and the inner piston are partitioned into two symmetric segments. Each segment is assigned a physical half-length, defined simply as $L_{half} = L_{total}/2$.

A series of intrinsic coordinate shifts must then be applied along the local Z-axis. A visual offset (vis_z $= -L_{half}/2$) and a relative joint positioning shift (rel_z $= -L_{half}$) translate the geometric centers, seamlessly aligning the rigid boundaries with their respective pivot joints. Additionally, the mass distribution is modeled by placing the centers of gravity for the outer strut and the inner piston at specific fractional distances along the directional vector $\mathbf{v} = p_{end} - p_{start}$. These are located at 25% and 75% of the total span, respectively:

$$\mathbf{cog}_{strut} = p_{start} + 0.25\mathbf{v}, \quad \mathbf{cog}_{piston} = p_{start} + 0.75\mathbf{v} \tag{3.31}$$

By strictly enforcing these analytical lengths and precise orientation angles across the entire assembly, the closed-loop kinematic chain is fully resolved. Consequently, the simulation initializes in a state of rigorous static equilibrium, effectively preventing any spurious numerical jitter or phantom forces upon startup.

3.5.5 Inertial Properties Modeling

In multi-body dynamics, the accurate distribution of mass is as critical as the geometric topology. The physics engine computes Coriolis forces, gravitational pull, and impact restitution based on the inertia tensor of each link. Assuming uniform density distribution, the fixed trusses and the telescopic elements are approximated as solid cylinders.

Given a total mass m, a radius r, and a length L, the principal moments of inertia along the local axes are formulated as:

$$I_{xx} = \frac{1}{12} m \left(3r^2 + L^2\right) \tag{3.32}$$

$$I_{yy} = \frac{1}{12} m \left(3r^2 + L^2\right) \tag{3.33}$$

$$I_{zz} = \frac{1}{2} m r^2 \tag{3.34}$$

The cross-products of inertia (I_{xy}, I_{xz}, I_{yz}) are assumed to be zero due to the axial symmetry of the cylindrical components. This inertial modeling ensures that the mass is realistically distributed along the leg architecture, providing authentic dynamic feedback to the MiL system during the high-stress compression phase of the lunar touchdown.

3.5.6 Mathematical Model of the Crushable Shock Absorber

The central and lateral shock absorbers of the landing legs employ energy-absorbing mechanisms designed to dissipate the kinetic energy of the touchdown through irreversible plastic deformation (commonly achieved via crushable aluminum honeycomb cartridges). Within the Gazebo simulation environment, native joint implementations are inadequate for this task, as they strictly model reversible elastic-viscous spring-damper dynamics. Therefore, a phenomenological elasto-plastic model is analytically formulated, supporting three distinct operational configurations: Unidirectional Double-Stage, Unidirectional Single-Stage, and Bidirectional Single-Stage.

1. Unidirectional Double-Stage Mode

Applied primarily to the central strut (DF), this mode accounts for a progressive dual-stage crushing behavior followed by a structural bottom-out region. Let *pos* represent the joint position coordinate and *vel* its relative velocity, the effective compression is evaluated as $x = pos \cdot c_{dir}$, where c_{dir} dictates the active direction.

The dynamic yield limit $F_{limit}(x)$ is defined as a piecewise function:

$$F_{limit}(x) = \begin{cases} F_{crush,1} & \text{if } x \leq x_{limit,1} \\ F_{crush,2} & \text{if } x_{limit,1} < x \leq x_{limit,2} \\ F_{crush,2} + k_{bottom}(x - x_{limit,2}) & \text{if } x > x_{limit,2} \end{cases} \tag{3.35}$$

where $F_{crush,1}$ and $F_{crush,2}$ are the yielding thresholds of the first and second stages, $x_{limit,1}$ and $x_{limit,2}$ are their respective stroke bounds, and k_{bottom} is the penalty stiffness preventing structural interpenetration beyond the maximum allowable stroke.

The trial elastic force is computed as:

$$F_{elastic} = k\,(x - x_{max}) \tag{3.36}$$

Depending on the state, the applied force $applied_f$ is determined through three distinct regimes:

- **Plastic Flow Phase ($F_{elastic} \geq F_{limit}(x)$):** The material yields and permanently deforms. The force saturates at the limit, and the memory variable updates:
$$applied_f = F_{limit}(x), \quad x_{max} = x - \frac{F_{limit}(x)}{k} \tag{3.37}$$
- **Elastic-Viscous Phase ($0 < F_{elastic} < F_{limit}(x)$):** The system operates elastically with damping:
$$applied_f = \min\left(\max\left(F_{elastic} + c \cdot v_{comp},\, 0\right),\, F_{limit}(x)\right) \tag{3.38}$$
- **Rebound Phase ($F_{elastic} \leq 0$):** The cartridge provides no elastic restitution during extension, leaving only damping:
$$applied_f = c_{rebound} \cdot v_{comp} \tag{3.39}$$

Finally, the feedback force vector is applied as $F_{applied} = -applied_f \cdot c_{dir}$.

2. Unidirectional Single-Stage Mode

This mode mirrors the physics of the double-stage configuration but features a constant crushing threshold F_{crush} instead of a piecewise function. If $F_{elastic} \geq F_{crush}$, the force saturates at F_{crush} and x_{max} is updated similarly. Otherwise, the elastic-viscous combination or pure rebound damping is applied depending on the sign of $F_{elastic}$, maintaining zero restitution upon extension.

3. Bidirectional Single-Stage Mode

To accommodate links subject to alternating compression and tension regimes, the bidirectional configuration independently tracks both positive and negative directions relative to the neutral position.

Two separate trial forces are evaluated:

$$F_{elastic,pos} = k\,(pos - x_{max,pos}) \tag{3.40}$$

$$F_{elastic,neg} = k\,(x_{max,neg} - pos) \tag{3.41}$$

For each direction, if the elastic trial force exceeds the single crush threshold F_{crush}, plastic yielding occurs and the respective deformation memory variable ($x_{max,pos}$ or $x_{max,neg}$) is updated. If both directional forces evaluate to zero, a symmetric rebound damping opposes the motion; otherwise, the net resistive force is synthesized by combining the opposing components:

$$F_{applied} = -f_{pos} + f_{neg} \tag{3.42}$$

This comprehensive multi-mode framework accurately captures the distinct mechanical dissipation profiles of both primary and secondary energy-absorbing elements across the entire landing gear architecture.

3.6 Touchdown Dynamics and Contact Mechanics

The exact moment of touchdown marks the most discontinuous and highly nonlinear phase of the landing simulation. As the footpads of the lander impact the lunar surface, the system transitions from a purely aerodynamic/propulsive free-flight regime to a heavily constrained multi-body dynamics problem. The accurate prediction of structural loads, rebounding phenomena, and tip-over instabilities strongly depends on the mathematical modeling of the physical contact between the metallic landing gears and the lunar terrain.

3.6.1 Contact Mechanics

To mitigate the numerical instabilities inherently associated with pure rigid-body impulses, the Gazebo simulation environment resolves collisions through a compliant, continuous-contact penalty method. Rather than enforcing strict non-penetration, the underlying Open Dynamics Engine (ODE) treats physical contacts as "soft constraints" [9]. This approach allows for a controlled, natural penetration of colliding objects by modeling the interaction as an equivalent visco-elastic spring-damper system.

Let δ represent the localized normal penetration depth of the footpad collision mesh into the ground mesh, and $\dot{\delta}$ be the relative penetration velocity. The normal reaction force F_n generated by the simulator is defined as:

$$F_n = \max\left(0,\ k_p\delta + k_d\dot{\delta}\right) \tag{3.43}$$

where k_p characterizes the contact stiffness (equivalent spring constant) and k_d dictates the viscous damping coefficient. Within the Simulation Description Format (SDF), these variables map directly to the `<kp>` and `<kd>` parameters configured for the lander's collision geometries. ODE mathematically translates these physical constants into its core solver variables, the Error Reduction Parameter (ERP) and Constraint Force Mixing (CFM), scaling them according to the integration step size.

Concurrently, tangential forces F_t are bounded by ODE's approximation of the classical Coulomb dry friction model. The maximum permissible shear force before macroscopic sliding initiates is strictly proportional to the instantaneous normal force:

$$\|\mathbf{F}_t\| \leq \mu\, F_n \tag{3.44}$$

where μ is the friction coefficient. Operationally, ODE enforces this Coulomb boundary by first evaluating the normal forces under the temporary assumption of frictionless contact. Subsequently, it establishes the absolute limits for the tangential friction components and solves the overall kinematic system constraints.

3.6.2 Dynamic Dual-Layer Lunar Regolith Modeling

A standard assumption is the assignment of constant values to k_p, k_d, and the friction coefficient μ. However, the actual lunar topology has a complex geological stratigraphy: a superficial layer of highly compressible fine dust (regolith) resting upon a rigid, unyielding rocky bedrock. To bridge the gap between the simulator's native contact mechanics and this realistic environment, a dynamic *dual-layer soil model* has been analytically formulated.

This custom architecture intercepts the physics engine's collision callbacks at each integration step, allowing both the normal compliance and the tangential friction properties to transition dynamically as a function of the instantaneous penetration depth δ. The soil is characterized by two distinct boundary states:

- **Surface Layer (Dust):** Highly compliant and slippery, characterized by low stiffness $k_{p,surface}$, low damping $k_{d,surface}$, and a low friction coefficient $\mu_{surface}$ acting as a granular lubricant.
- **Bedrock Layer (Rock):** Highly rigid and abrasive, characterized by extreme stiffness $k_{p,rock}$, high damping $k_{d,rock}$, and a high friction coefficient μ_{rock} providing aggressive mechanical interlocking.

Given a defined thickness for the soft regolith layer h_{dust}, a dimensionless interpolation parameter α is defined to map the transition. By averaging the penetration depths of all active collision points across the footpad mesh, α is evaluated as:

$$\alpha = \min\left(\max\left(\frac{\delta}{h_{dust}},\, 0\right),\, 1\right) \tag{3.45}$$

This function constrains $\alpha \in [0, 1]$. The instantaneous, depth-dependent physical coefficients are subsequently calculated via linear interpolation:

$$k_p(\delta) = k_{p,surface} + \alpha\,(k_{p,rock} - k_{p,surface}) \tag{3.46}$$

$$k_d(\delta) = k_{d,surface} + \alpha\,(k_{d,rock} - k_{d,surface}) \tag{3.47}$$

$$\mu(\delta) = \mu_{surface} + \alpha\,(\mu_{rock} - \mu_{surface}) \tag{3.48}$$

Physically, as long as the footpad sinks within the shallow dust layer ($\delta < h_{dust}$), it experiences a soft, progressive deceleration and remains susceptible to lateral sliding due to the reduced surface friction. If the impact energy is severe enough to cause the footpad to pierce through the regolith and strike the bedrock ($\delta \geq h_{dust}$, yielding $\alpha = 1$), the custom control loop instantly saturates the contact parameters to the bedrock values. This generates a sharp normal force peak that thoroughly stresses the landing gear's shock absorbers, while simultaneously arresting lateral motion through maximized dry friction.

Chapter 4

Software Architecture and Simulation (Model-in-the-Loop)

The mathematical and physical models formalized in the previous chapter require a robust computational framework to be evaluated dynamically. This chapter details the software implementation of the virtual simulation environment, the general topology of the system, the procedural generation of the simulation scenarios, the integration of custom C++ physics plugins, and the automated data-logging infrastructure.

4.1 System Architecture Overview

The infrastructure developed for this thesis relies on the integration of two industry-standard platforms: ROS 2 (Robot Operating System 2), acting as the communication, scheduling, and control middleware; and Gazebo, functioning as the rigid-body physics, multibody kinematics, and collision-resolution engine.

4.1.1 Workspace Topology and Package Organization

To ensure strict modularity, high maintainability, and optimal computational performance, the ROS 2 workspace is topologically divided into two distinct packages. This separation effectively isolates the high-frequency physics interactions from the high-level operational logic and data logging:

- **C++ Package (*Gazebo Plugins*):** The physics-interaction layer is developed entirely in C++. Bypassing the ROS 2 network overhead for physics

calculations ensures maximum computational efficiency and prevents clock desynchronization within the solver. This package compiles into shared libraries (`.so`) containing custom Model and World Plugins directly injected into the Gazebo engine. The implemented modules include:

  - `GazeboRosThruster`: Applies continuous, vectorial propulsive forces to the lander's structural links.
  - `InitialVelocityPlugin`: Enforces strict initial kinematic boundary conditions without causing numerical instabilities in the solver.
  - `CrushableJointPlugin`: Implements the multi-mode elastoplastic behavior of the shock absorbers, overriding the standard elastic joint limits.
  - `DynamicTerrainPlugin`: Intercepts the `ContactManager` to dynamically interpolate regolith stiffness and friction based on penetration depth.

- **Python Package (`lander_py`):** The higher-level logic is implemented in Python to leverage its rapid prototyping capabilities, extensive data-processing libraries, and robust node orchestration. This package houses:

  - *Procedural Generators:*
    * `generate_body.py`: constructs the lander's two primary components (the body and the passenger) and configures their respective mass and inertia properties dynamically based on the specified initial conditions.
    * `generate_legs.py`: automates the spatial topology and kinematic assembly of the multi-link landing gear.
    * `generate_scenario.py`: orchestrates the complete simulation environment by integrating the lander model, the terrain and world physics.
  - *Actuation and Disturbance:* `thrusters.py` (computing the Main Engine Cut-Off transient) and `psi_disturbance.py` (generating the stochastic Plume Surface Interaction forces).
  - *Telemetry Nodes:* `data_logger.py`, `thruster_logger.py`, and `shock_absorber_logger.py`, responsible for monitoring states, detecting the touchdown event, and triggering automated shutdowns.

4.1.2 Simulation Orchestration via ROS 2 Launch System

The entire lifecycle of the simulation is governed by an automated Python-based *Launch File*. Rather than manually starting individual nodes, the launch script acts as a deterministic state machine operating in distinct sequential phases.

Phase 1: Procedural Generation. Before the physics engine is initialized, the system must adapt to the specific initial parameters selected for the current simulation run. The launch file employs the Python `subprocess` module to synchronously execute a suite of generation scripts. This process dynamically overwrites the `body.sdf` and `passenger.sdf` (updating mass and inertia tensors), generates the individual leg files (`leg1.sdf`, `leg2.sdf`, `leg3.sdf`, and `leg4.sdf`) with tailored geometric, structural, and compliance parameters, assembles the complete `lander.sdf` configuration, and updates the `landing_scenario.world` (configuring initial coordinates, attitude, and terrain topology).

Phase 2: Engine Ignition and Delayed Node Startup. Once the structural files are synthesized, Gazebo is launched in a paused state (`pause: 'true'`). A critical challenge in ROS 2-Gazebo integration is preventing race conditions, where control nodes publish commands before the physics engine has fully loaded the model. To mitigate this, a `TimerAction` is implemented to explicitly delay the instantiation of the Python nodes.

```
return LaunchDescription([
    gazebo_launch,
    # Delaying node startup to prevent race conditions with Gazebo
    TimerAction(period=2.0, actions=[thrusters_node]),
    TimerAction(period=2.0, actions=[psi_disturbance_node]),
    TimerAction(period=2.0, actions=[data_logger_node]),
    TimerAction(period=2.0, actions=[shock_absorber_logger_node]),
    TimerAction(period=2.0, actions=[thruster_logger_node])
    unpause_gazebo
])
```

Listing 4.1: Excerpt from the ROS 2 Launch File demonstrating delayed startup and automated shutdown logic.

Phase 3: Automated Graceful Shutdown. The architecture is designed to run unsupervised, which is mandatory for extensive simulation campaigns. The `data_logger` continuously evaluates the lander's kinematic state against two stopping criteria: a stabilization threshold (*Steady State*, where all velocities fall below a predefined ϵ) or a safety timeout post-touchdown. Once a criterion is met, a boolean flag is published, cleanly terminating all nodes and triggering the extraction of the telemetry plots.

4.1.3 Network Topology and Inter-Process Communication

The data flow between the modular components is achieved through the ROS 2 *Publisher/Subscriber* mechanism, creating a closed feedback loop:

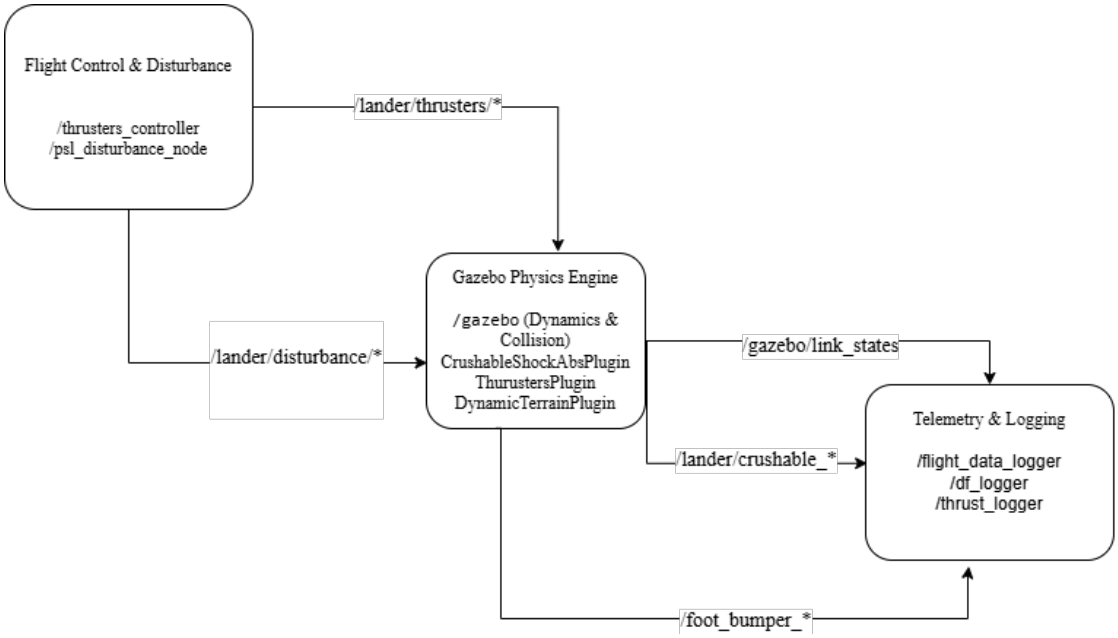


Figure 4.1: Partial ROS 2 computational graph of the lunar lander simulation architecture.

1. **Control Action:** The `thrusters` and `psi_disturbance` nodes compute the required active forces and publish `Float64` messages to their respective dedicated topics.
2. **Physical Actuation:** The C++ plugins, subscribed to these topics, read the commands and translate them into physical force vectors applied to the lander's multi-body links using Gazebo's API.
3. **State Feedback:** Gazebo calculates the resulting rigid-body dynamics and publishes the updated kinematic state on `/gazebo/model_states`.
4. **Logging and Evaluation:** The telemetry nodes subscribe to these feedback topics, recording the flight history and computing the output metrics .

The complete mapping of active nodes, topics, and message directions within this MiL architecture is illustrated in the computational graph generated via the `rqt_graph` introspection tool (Figure A.1).

4.2 Centralized Parametric Configuration

A fundamental requirement for a robust simulator, is the complete separation of the simulation logic from the physical and environmental parameters. Hardcoding

numerical values within the source code or the structural SDF files severely limits scalability and introduces human error during iterative testing.

To address this, the entire simulation framework is designed to be strictly parametric. The system is governed by a centralized configuration file named `config.yaml`. Before the Gazebo physics engine is even initialized, the Python procedural generators parse this file, extract the defined parameters, and dynamically inject them into the XML trees of the SDF models and the `.world` scenario.

The configuration file is structurally partitioned into specific functional domains (namespaces), ensuring that every aspect of the lander and the environment can be rapidly reconfigured without modifying the underlying C++ or Python source code.

4.2.1 Configuration Domains and Data Structures

The `config.yaml` file orchestrates the simulation through the following primary data structures:

- **Simulation Control (`simulation_stop`):** Defines the boundary conditions for the automatic shutdown of the nodes. It sets the kinematic steady-state tolerance (ϵ) and the safety timeouts post-touchdown, ensuring that the simulation stops gracefully once the physical transient is exhausted.
- **Mass and Inertial Scenarios (`body_generation`):** Instead of relying on a single static mass, this section provides dictionaries containing multiple discrete payload scenarios (`passenger_scenarios`) and structural states (`dry_scenarios`), complete with their respective Centers of Gravity (CoG) and inertia tensors. It also houses the geometric parameters of the tanks and the fluid densities, allowing the `generate_body.py` script to actively compute the exact fluid distribution for any requested landing mass within a continuous range.
- **Initial Kinematics and Environment (`landing_scenario`):** This section establishes the operational boundaries and the environmental conditions for the touchdown phase. The initial kinematic state vector of the lander is defined through operational ranges (`[min, max]`) that encompass absolute planar positions, altitude, Euler attitude angles, as well as linear and angular velocities. The `generate_scenario.py` script processes these boundaries to configure the precise starting state of the lander for the current simulation run. Furthermore, this section explicitly defines the contact mechanics parameters for the dual-layer soil model which are directly accessed and managed by the `DynamicTerrainPlugin`.

- **Propulsion System Constraints (`thrusters`):** This configuration block dictates the operational characteristics of the lander's active propulsion system. It specifies the architectural layout by defining the number of main engines and establishes their precise performance boundaries, including the minimum operational throttle (`t_min`) and the absolute maximum thrust capacity per engine (`t_max` and `engine_max_thrust`).
- **Landing Gear Topology and Mechanics (`leg_generation`):** This highly detailed namespace parametrizes the entire kinematic chain discussed in Chapter 3. It includes the exact spatial coordinates of the structural nodes, the mass and radius of the cylindrical links, and the specific elastoplastic parameters characterizing the `CrushableJointPlugin`. Those parameters can be systematically varied and tuned here to evaluate different geometric, structural, and compliance configurations without needing to recompile the packages.

4.3 Body and Payload Generation (`generate_body.py`)

The first phase of the procedural pipeline governs the stochastic mass distribution and inertial tracking of the lander.

4.3.1 Mass Sampling and Scenario Selection

The script begins by parsing the centralized `config.yaml` file under the `body_generation` namespace.

Instead of generating entirely random structural properties, which could lead to physically inconsistent vehicles, the script initializes the system by sampling from a discrete set of realistic design scenarios. A target total landing mass (M_L) is generated from a uniform distribution. Based on this value, the script randomly extracts two predefined configurations: one for the dry body and one for the payload. Each dictionary contains the respective baseline mass (M_{dry} and $M_{passenger}$), Center of Gravity (CoG), and principal inertia tensor.

The Passenger properties are saved directly for subsequent injection into the SDF template. Then, in order to determine the final inertial properties of the body link (with the remaining propellant), the residual propellant mass (ΔM_P) and its fractionation into MMH and MON are calculated via the mass balance and stoichiometric relations established in Section 3.1.

4.3.2 Numerical Integration of Fluid Properties

Since the tanks have a complex geometry (as detailed in Section 3.2), finding the exact CoG and Inertial properties for a given target volume (derived from the residual propellant mass) requires a robust numerical approach. To achieve this, the script implements an iterative root-finding algorithm combining numerical integration and the bisection method.

First, an internal helper function `volume_until_height(h)` is defined to compute the accumulated volume up to an arbitrary height h using trapezoidal numerical integration (`np.trapezoid`) [10]. The bisection loop iteratively refines the fluid height until the computed volume matches the target volume within a strict tolerance ϵ.

```
def volume_until_height(h):
    z = np.linspace(z_bottom, h, nz)
    area = np.zeros_like(z)
    for i, zi in enumerate(z):
        r = radius_at_z(zi)
        area[i] = np.pi * r**2
    return np.trapezoid(area, z)

max_iter = 100
iter_count = 0
error = float('inf') # Initialize the error to an infinite value

# Propellant height (bisection method)
z_low = z_bottom
z_high = z_top

while error > epsilon and iter_count < max_iter:
    z_mid = 0.5 * (z_low + z_high)
    v_mid = volume_until_height(z_mid)
    error = abs(V_target - v_mid)

    if v_mid < V_target:
        z_low = z_mid
    else:
        z_high = z_mid

    iter_count += 1

h_fill = z_mid
```

Listing 4.2: Bisection method for determining the propellant fill height.

Once h_{fill} is successfully determined, the fluid column is discretized into thousands of infinitesimal disks ($N_z = 4000$). The script computes the local Center of Gravity (CoG) and the barycentric inertia tensor of the fluid column by integrating

the elemental mass and inertia of each disk:

```
# Calculation of properties via vertical discretization
z = np.linspace(z_bottom, h_fill, nz)
dz = z[1] - z[0]
rloc = np.zeros_like(z)

for i, zi in enumerate(z):
    rloc[i] = radius_at_z(zi)

# Elemental properties
area = np.pi * rloc**2
dV = area * dz
dm = current_rho * dV
total_mass = np.sum(dm)

# Local fluid Center of Gravity
z_cg_local = np.sum(z * dm) / total_mass

# Numerical integration of the Inertia Tensor
Ixx = Iyy = Izz = 0.0

for zi, ri, dmi in zip(z, rloc, dm):
    dIzz = 0.5 * dmi * ri**2
    dIxy = 0.25 * dmi * ri**2
    dz_cg = zi - z_cg_local

    # Parallel axis theorem for the infinitesimal disk
    Ixx += dIxy + dmi * dz_cg**2
    Iyy += dIxy + dmi * dz_cg**2
    Izz += dIzz

return (z_cg_local, Ixx, Iyy, Izz)
```

Listing 4.3: Snippet from `generate_body.py` demonstrating the numerical integration of fluid mass

4.3.3 Global Tensor Assembly and SDF Templating

The Gazebo physics engine requires a single, aggregated CoG and a unified inertia tensor for the primary *Body Link* (which conceptually merges the dry structure and the remaining propellant).

To achieve this, the script calculates the global CoG of the full body, then it translates the localized inertia matrices of the dry part and the four distinct fluid columns to the newly calculated CoG. This spatial translation is mathematically executed using the generalized tensor formulation of the Huygens-Steiner (Parallel Axis) theorem, implemented by using `numpy` matrix operations.

```python
# Huygens-Steiner theorem (Parallel Axis Theorem)
def steiner_transport(I_cg, mass, old_pos, new_pos):
    d = np.array(old_pos) - np.array(new_pos)
    d_sq = np.dot(d, d)
    outer_d = np.outer(d, d)
    return np.array(I_cg) + mass * (d_sq * np.eye(3) - outer_d)
```

Listing 4.4: Implementation of the Huygens-Steiner theorem.

In the final step, the script reads the `body_template.sdf` and `passenger_template.sdf` files, replacing the predefined string placeholders (e.g., `{MASS_WET}`, `{IXX}`) with the values generated by the algorithms. The finalized files are saved directly into the active Gazebo model directories, ensuring that the simulator boots with a physically consistent mass distribution.

```python
# CoG calculation for the aggregated Body Link
M_body_link = M_dry + delta_M_P
cog_body = (M_dry * np.array(COG_DRY) +
            m_tank_MON * cog_tank2 + m_tank_MON * cog_tank3 +
            m_tank_MMH * cog_tank1 + m_tank_MMH * cog_tank4) /
    M_body_link

# Transporting all inertia tensors to the global CoG
I_dry_trans = steiner_transport(I_DRY, M_dry, COG_DRY, cog_body)
I_t1_trans = steiner_transport(I_tank_MMH, m_tank_MMH, cog_tank1,
    cog_body)
# ... [repeated for all 4 tanks] ...

I_body_totale = I_dry_trans + I_t1_trans + I_t4_trans + I_t2_trans
    + I_t3_trans
```

Listing 4.5: Global CoG and Inertia tensor assembly.

4.4 Scenario Generation (`generate_scenario.py`)

This Python script initializes the operational environment and enforces the initial kinematic state of the lander.

The execution begins by parsing the `config.yaml` file to extract the specified operational boundaries for the vehicle's initial spatial coordinates, attitude, and triaxial velocities. Concurrently, it retrieves the fundamental physical coefficients required by the dynamic terrain model, including the dual-layer contact stiffness, damping, and Coulomb friction parameters.

Once the kinematic state is sampled within the provided ranges, the script acts as a procedural generator for the simulation arena. It reads a pre-defined structural blueprint (`landing_scenario_template.world`) and dynamically injects the

computed kinematic variables and soil properties by substituting specific string placeholders. Finally, it exports the fully configured `landing_scenario.world` file, creating a tailor-made environment ready to be instantiated by the Gazebo physics engine.

4.4.1 Safe Altitude Ray-Casting and 3D Attitude Resolution

If the lander is initialized at a fixed absolute vertical coordinate (Z) over an unexpected elevated feature, the footpads could instantiate partially embedded within the ground mesh. This geometrical compenetration generates infinite repulsive penalty forces within the physics engine, leading to immediate catastrophic solver failures.

To eliminate this vulnerability, a 3D collision-avoidance algorithm was created utilizing the `trimesh` library [11]. The algorithm dynamically calculates the absolute minimum safe spawn altitude (`max_required_pos_z`) required to guarantee that no structural component intersects the terrain upon initialization, explicitly accounting for both the local topography and the vehicle's randomized initial tilt.

The true elevation under the landing gear is obtained through a downward ray-casting algorithm, structured in four sequential steps:

1. **Rotation Matrix Assembly:** The sampled Euler angles are mathematically converted into a standard 3D rotation matrix (R_{mat}).

2. **Footpad Edge Projection:** Because the footpads are physical disks rather than infinitesimal points, tilting the lander lowers one edge of the pad relative to its joint center. The script analytically computes this geometric offset (`foot_disc_drop`) based on the current pitch and roll angles.

3. **Terrain Ray-Casting:** For each of the four legs, the local base offsets are rotated by R_{mat} to determine the true global (X, Y) coordinates of the footpads. A vertical ray (direction $[0, 0, -1]$) is cast downward against the terrain mesh using `trimesh` to extract the precise local ground elevation (`z_foot_terrain`).

4. **Safe Altitude Calculation:** The script calculates the exact Z-coordinate required for the main body so that the lowest edge of each specific footpad merely grazes the terrain.

The global minimum safe altitude is defined as the mathematical maximum among the four leg-specific requirements, ensuring clearance for the entire structure. Once this absolute baseline is established, the target initial flight altitude is imposed.

The implementation of this logic, including the `trimesh` intersection query and the spatial rotation loop, is detailed in the following snippet.

```
def get_z_height(x, y, max_z=200.0):
    ray_origins = np.array([[x, y, max_z]])
    ray_directions = np.array([[0, 0, -1]])
    locations, index_ray, index_tri = mesh.ray.intersects_location
(ray_origins=ray_origins,ray_directions=ray_directions)
    if len(locations) > 0:
        return np.max(locations[:, 2])
    return None
```

Listing 4.6: Snippet from `generate_scenario.py` that shows the `get_z_height` function.

```
max_required_pos_z = -9999.0

    for dx, dy in feet_offsets_local:
        # Apply the 3D rotation
        rx, ry, rz = rotate_point(dx, dy, z_foot_joint_local,
R_mat)

        # New global coordinate of the foot
        foot_x_global = pos_x + rx
        foot_y_global = pos_y + ry

        # Find the lowest point of the foot
        z_foot_lowest_rotated = rz + foot_disc_drop

        # Find the height of the mesh under the foot
        z_foot_terrain = get_z_height(foot_x_global, foot_y_global
)
        if z_foot_terrain is None:
            z_foot_terrain = 0.0

        # Required body Z so that the foot's edge exactly touches
the ground
        required_pos_z = z_foot_terrain - z_foot_lowest_rotated

        # Update the global maximum Z requirement
        if required_pos_z > max_required_pos_z:
            max_required_pos_z = required_pos_z

    if max_required_pos_z == -9999.0:
        max_required_pos_z = 0.0
```

Listing 4.7: Snippet from `generate_scenario.py` demonstrating the 3D attitude rotation and terrain ray-casting logic to determine the safe spawn altitude.

4.4.2 Dynamic Mesh Cropping for Computational Efficiency

High-fidelity topographical maps of the lunar surface are typically composed of large-scale, highly dense polygon meshes. Loading an entire regional map into the Gazebo physics engine for every iteration of a long simulation campaign introduces a severe computational issue. Since the physical interaction between the lander and the terrain during the terminal descent and touchdown phases is strictly localized to a small geographical footprint, a dynamic mesh-cropping algorithm is executed prior to engine initialization. The script utilizes the `trimesh` Python library to extract a localized Region of Interest (ROI) from the heavy master terrain file.

Once the randomized global spawn coordinates (X, Y) are sampled, the algorithm defines a bounding box around this epicenter using a predefined safety threshold (`CROP_RADIUS`). The extraction is performed computationally through four sequential boolean `slice_plane` operations. Each operation is defined by an origin point and a normal vector, effectively slicing the 3D space and discarding all mesh geometry that falls outside the active landing zone.

```
# Extract randomized global spawn coordinates
pos_x = random.uniform(*POS_X_RANGE)
pos_y = random.uniform(*POS_Y_RANGE)

# Mesh cutting
print(f"\nCropping the mesh within a {CROP_RADIUS}m radius around X={pos_x:.2f}, Y={pos_y:.2f}...")

cropped_mesh = mesh.copy()

# Cuts
cropped_mesh = cropped_mesh.slice_plane(plane_origin=[pos_x + CROP_RADIUS, 0, 0], plane_normal=[-1, 0, 0])
cropped_mesh = cropped_mesh.slice_plane(plane_origin=[pos_x - CROP_RADIUS, 0, 0], plane_normal=[1, 0, 0])
cropped_mesh = cropped_mesh.slice_plane(plane_origin=[0, pos_y + CROP_RADIUS, 0], plane_normal=[0, -1, 0])
cropped_mesh = cropped_mesh.slice_plane(plane_origin=[0, pos_y - CROP_RADIUS, 0], plane_normal=[0, 1, 0])

# New mesh
CROPPED_MESH_PATH = os.path.join(MESH_DIR, "terrain_cropped.stl")
cropped_mesh.export(CROPPED_MESH_PATH)
```

Listing 4.8: Snippet from `generate_scenario.py` demonstrating the dynamic mesh cropping via boolean plane intersections.

The resulting lightweight mesh (`terrain_cropped.stl`) is immediately exported and referenced dynamically by the `landing_scenario.world` template. This

automated preprocessing step drastically accelerates the overall execution speed of the simulation campaign.

4.4.3 The InitialVelocity C++ Plugin

While Gazebo natively allows defining the initial spatial pose (position and orientation) of a model directly inside the XML `.world` structures, in order to enforce non-zero initial linear and angular velocities a custom C++ ModelPlugin named `InitialVelocityPlugin` was developed.

Plugin Initialization and SDF Parameter Parsing

The `InitialVelocityPlugin` is compiled as a shared C++ library (`.so`) and injected directly into the `lander.sdf` structural configuration as a Model Plugin. The initialization sequence is governed by the overridden `Load` method, which is executed synchronously by the physics engine during the model's instantiation. Upon execution, the method acquires an authoritative pointer (`physics::ModelPtr`) to the parent lander model in the simulator's memory space. Concurrently, it establishes a bridge to the ROS 2 middleware interface (`gazebo_ros::Node::Get`), granting the plugin access to the system's centralized logging framework.

The primary function of the `Load` method is to parse the initial kinematic constraints required by the specific run. It scans the parsed SDF (`sdf::ElementPtr`) for the custom XML tags `<linear_velocity>` and `<angular_velocity>`, (which were procedurally injected by the Python orchestrator) and mathematically casts their raw string arrays into native `ignition::math::Vector3d` objects.

```
1  // Read SDF parameters
2  if (_sdf->HasElement("linear_velocity")) {
3      this->linear_velocity_ = _sdf->Get<ignition::math::Vector3d>("
       linear_velocity");
4  }
5
6  if (_sdf->HasElement("angular_velocity")) {
7      this->angular_velocity_ = _sdf->Get<ignition::math::Vector3d>(
       "angular_velocity");
8  }
9
10 // Connect update callback to the main physics thread
11 this->update_connection_ = event::Events::ConnectWorldUpdateBegin(
12     std::bind(&InitialVelocityPlugin::OnUpdate, this)
13 );
```

Listing 4.9: Parsing of the initial kinematic vectors from the procedural SDF file within the `Load` method.

As a final initialization step, the plugin registers a persistent callback to the core Gazebo physics thread, binding its execution to the `ConnectWorldUpdateBegin` signal event loop.

Kinematic State Injection (`ApplyVelocities`)

Once the desired velocity vectors are securely cached in the plugin's internal memory, the state must be physically transferred to the rigid body structure. This operation is isolated within the private `ApplyVelocities` function.

This method interacts directly with the Gazebo physics API. By invoking the `SetLinearVel()` and `SetAngularVel()` methods on the parent model pointer, the plugin forcefully overrides the instantaneous velocities of the lander's center of mass.

```
1 void ApplyVelocities()
2 {
3     // Forcefully override the kinematic state array
4     this->model_->SetLinearVel(this->linear_velocity_);
5     this->model_->SetAngularVel(this->angular_velocity_);
6 }
```

Listing 4.10: Direct manipulation of the model's kinematic state via the Gazebo Physics API.

This routine is invoked immediately at the end of the `Load` method to establish the correct spatial trajectories upon the lander's initial spawn, ensuring the vehicle correctly enters the ballistic descent phase towards the lunar surface.

4.5 Landing Gear Procedural Assembly and Shock Absorber Modeling

This section details the software implementation governing both the procedural generation of the lander's leg architecture and the numerical modeling of the crushable shock absorbers. Specifically, it addresses the transition from the theoretical kinematic formulation to the functional rigid-body representation required by the Gazebo physics engine.

4.5.1 Legs Generation (`generate_legs.py`)

The `generate_legs.py` script serves as an automated, programmatic CAD assembly engine. Rather than relying on hard-coded static models, the script operates as a highly flexible procedural pipeline, specifically engineered to support the

stochastic nature of the simulation campaign. Its execution flow can be broken down into three primary software mechanisms:

Parameter Extraction

It begin with parsing the `leg_generation` block within the global `config.yaml`. To facilitate extensive testing, the script identifies also the critical contact mechanics variables that will be injected into the (`CrushableJointPlugin`), such as suspension stiffness (`df_k`), damping coefficients (`df_c`), and structural yield forces (`df_crush_force_stage1`).

SDF Templating Engine

To translate the Python dictionaries into executable physics environments, the script employs the same string-replacement templating methodology used in the previous section. It loads generic blueprint files (`leg_template.sdf`), which are entirely devoid of numerical data and populated instead with placeholder tags (e.g., `{TRUSS_MASS}`, `{POSE_C1D}`, `{IXX_FOOT}`).

In a sequential loop, the algorithm mathematically solves the kinematics and inertial properties for each link (using the equations detailed in Section 3.5), formats the resulting arrays into strictly constrained strings, and replaces the corresponding tags. This procedure generates four distinct, physics-ready XML directories (`leg1` through `leg4`).

```
1 # Example of SDF string replacement for the fixed trusses
2 for node in ["C1", "C2", "A1", "A2"]:
3     leg_model = leg_model.replace(f"{{POSE_{node}D}}", trusses[
    node]["pose_link"])
4     leg_model = leg_model.replace(f"{{L_{node}D}}", f"{trusses[
    node]['L']:.4f}")
5     leg_model = leg_model.replace(f"{{IXX_{node}D}}", f"{trusses[
    node]['ixx']:.4f}")
```

Listing 4.11: Sequential tag replacement bridging the Python math engine to the Gazebo SDF format.

Macro-Assembly and Metadata Injection

After generating the individual legs, the script calculates the comprehensive global mass and inertial tensor of the complete leg structure by applying the parallel axis theorem (Steiner's transport) to all local components.

To enable accurate post-flight telemetry analysis, these aggregated inertial parameters, along with the randomized variables, are injected directly into the end of the `leg1.sdf` file as structured XML comments (`<!- METADATA_LEG_PROPERTIES...`

->). Finally, the algorithm integrates the four legs onto the main lander body via the `lander_template.sdf`, enforcing a rigid 90° yaw rotation step ($\psi = i \times \pi/2$) for each consecutive quadrant around the octagonal base, thus finalizing the simulation-ready rigid body hierarchy.

4.5.2 C++ Implementation of the Crushable Shock Absorber

Standard robotic physics engines, including Gazebo's ODE, natively model joints as elastic-viscous spring-damper systems ($F = kx + c\dot{x}$). However, in this case, the energy dissipation relies on the irreversible, elastoplastic deformation of internal aluminum honeycomb cartridges. To bridge the architectural gap between the theoretical piecewise formulation (detailed in Section 3.5.6) and the limitations of the native physics solver, a custom C++ Plugin (`CrushableJointPlugin`) was created. The plugin dynamically overrides the native joint forces, calculating the correct elastoplastic reaction at every physics integration step based on the joint's loading history.

Initialization and Parameter Parsing

Upon instantiation, the plugin executes the overridden `Load` method. While it utilizes the same foundational architecture detailed in Section 4.4.3 (acquiring the authoritative `physics::ModelPtr` and the `sdf::ElementPtr` to establish the ROS 2 bridge), this setup phase specifically leverages the model pointer to query the physics engine for the target shock absorber. By extracting the `joint_name` from the XML, the plugin successfully binds to the corresponding Gazebo joint pointer, granting direct control over its internal mechanical states.

Following this pointer acquisition, the method parses a comprehensive set of kinematic and dynamic parameters to precisely define the shock absorber's behavior. The input parameters extracted from the SDF are detailed below:

`stiffness` (k): The base structural stiffness of the joint (default: 5×10^6 N/m).

`damping` (c): The viscous damping coefficient applied during the compression phase (default: 15000 Ns/m).

`rebound_damping`: A specific damping coefficient applied exclusively during the extension (rebound) phase, allowing for asymmetric energy dissipation (default: 1000 Ns/m).

`compression_dir`: A scalar multiplier (typically 1.0 or -1.0) used to correctly align the plugin's logic with the physical joint's positive axis of movement.

`bidirectional`: A boolean flag that, when active, allows the crushable logic to be applied in both directions of travel, rather than just during compression.

`dual_stage`: A boolean flag that determines the operational mode of the energy absorber. Depending on this flag, the method branches into two distinct parameter parsing logics:

Single-Stage Configuration (`dual_stage = false`)
If the dual-stage mode is disabled, the plugin behaves as a standard energy absorber characterized by a single force threshold:

- `crush_force`: The constant limit force at which the structure begins to crush and yield plastically (default: 35000 N).

Dual-Stage Configuration (`dual_stage = true`)
If activated, the plugin simulates a more complex, two-tiered crushing mechanism, requiring the following specific parameters:

- `crush_force_stage_1` and `stroke_limit_stage_1`: Define the yielding force (default: 20000 N) and the maximum displacement (default: 0.254 m) for the initial, softer crushing phase.
- `crush_force_stage_2`: The higher force threshold (default: 42200 N) required to continue crushing the structure once the first stage is fully depleted.
- `total_stroke_limit`: The absolute maximum displacement allowed for the entire mechanism (default: 0.8128 m).
- `bottomout_stiffness`: A highly punitive stiffness value (default: 10^7 N/m) applied to simulate a hard stop when the `total_stroke_limit` is reached.

Finally, after successfully loading the configuration, the method establishes a ROS 2 publisher that broadcasts the instantaneous joint state (publishing a `geometry_msgs::msg::Point` message containing position, force, and velocity data). The setup concludes by binding the `OnUpdate` function to the Gazebo world update event, ensuring the physics are calculated at every simulation step.

Dynamic Execution and Elastoplastic Logic: The `OnUpdate` Callback

The core of the plugin resides in its `OnUpdate` callback, which executes synchronously with the Gazebo physics engine at every simulation step. This method is responsible for querying the instantaneous joint state (position and velocity), evaluating the elastoplastic flow rules based on the selected configuration and computing the restoring force (by implementing the equations described in Section 3.5.6).

```
double current_compression = pos * this->compression_dir_;
double comp_vel = vel * this->compression_dir_;

// Trial elastic force
double elastic_force = this->stiffness_k_ * (current_compression -
    this->max_pos_deformation_);
if (elastic_force < 0.0) elastic_force = 0.0;

double applied_f = 0.0;
if (elastic_force >= this->crush_force_) {
    // Plastic yielding
    applied_f = this->crush_force_;
    this->max_pos_deformation_ = current_compression - (this->
   crush_force_ / this->stiffness_k_);
} else if (elastic_force > 0.0) {
    // Elastic-viscous loading
    applied_f = elastic_force + (this->damping_c_ * comp_vel);
    if (applied_f > this->crush_force_) applied_f = this->
   crush_force_;
    if (applied_f < 0.0) applied_f = 0.0;
} else {
    // Rebound phase
    applied_f = this->rebound_damping_ * comp_vel;
}

force_to_apply = -applied_f * this->compression_dir_;
```

Listing 4.12: Elastoplastic flow logic for the unidirectional single-stage mode.

Force Injection and Telemetry

Once the algorithm calculates `force_to_apply` based on the active mode, this value is injected directly into the physics engine using the `SetForce` API. Immediately afterward, the plugin packages the current joint displacement, the applied force, and the compression velocity into a `geometry_msgs::msg::Point` object. This message is broadcasted over the ROS 2 network, enabling high-frequency telemetry logging and post-impact analysis.

```
this->joint_->SetForce(0, force_to_apply);
geometry_msgs::msg::Point msg;
msg.x = pos;
msg.y = force_to_apply;
msg.z = vel;
force_disp_pub_->publish(msg);
```

Listing 4.13: Force injection into the physics engine and ROS 2 telemetry publishing.

4.6 Actuation System (Thrusters)

4.6.1 The Custom C++ Propulsion Plugin

The `GazeboRosThruster` plugin was developed to simulate the physical action of propulsors inside the Gazebo simulator. The plugin ingests raw thrust commands via the ROS 2 middleware network and translates them into continuous 3D force vectors applied at exact geometric coordinates relative to the lander's structure.

Initialization and Parameter Parsing: The `Load` Method

The plugin class inherits publicly from the native Gazebo `ModelPlugin` base class. The active instantiation of the subsystem occurs inside the overridden `Load` method, which is automatically triggered by the Gazebo server thread when parsing the model's structural XML file. The method receives an `sdf::ElementPtr` to instantiate the ROS 2 node bridge and parse the custom configuration tags. Concurrently, the authoritative `physics::ModelPtr` is utilized to query the physics engine and acquire a direct pointer to the specific rigid body (`physics::LinkPtr`) where the thruster is physically attached.

Once the target link is successfully resolved, the method proceeds to extract the engine's spatial offsets, thrust vectors, and topic definitions. The input parameters extracted from the SDF are described below:

`link_name`: Defines the specific rigid structural element (`body_link`) to which the engine is mechanically attached.

`offset`: Stores the local Cartesian coordinates (x, y, z) defining the exact point where the engine nozzle exit plane is located relative to the structural origin of the selected link.

`direction`: A normalized unit vector defining the line of action of the thrust.

`ros_topic`: The specific ROS 2 topic string. Dynamically allocating the topic instead of hardcoding it makes the plugin completely modular: the same compiled library can simulate multiple independent thrusters by simply declaring different topics in the XML file.

Finally, `ConnectWorldUpdateBegin` acts as the driving clock of the plugin, binding the local `OnUpdate` method directly to the simulator's primary event loop.

Continuous Physics Execution

A fundamental characteristic of physics solvers like Gazebo's ODE is that force and torque accumulators are entirely cleared at the end of every numerical integration

step. Consequently, propulsion forces cannot be applied via sporadic event triggers; they demand a continuous polling mechanism strictly synchronized with the physics clock. To achieve this, the plugin decouples data ingestion from force application. An asynchronous ROS 2 callback updates the internal thrust variable, while the `OnUpdate` method (called at every simulator time step) evaluates and applies the vector mechanics:

```
private:
    void OnThrustMsg(const std_msgs::msg::Float64::SharedPtr msg)
    {
        this->thrust_command_ = msg->data;
    }

    void OnUpdate()
    {
        if (this->thrust_command_ > 0.0) {
            // Force vector calculation (Direction * Magnitude)
            ignition::math::Vector3d force = this->direction_ *
   this->thrust_command_;

            // Apply the force at the specified nozzle coordinate
            this->link_->AddLinkForce(force, this->offset_);
        }
    }
```

Listing 4.14: Continuous force application logic synchronized with the physics step.

If the thrust command is positive, the `OnUpdate` loop performs two instantaneous operations per step:

- **Vector Modulation:** The scalar magnitude (`thrust_command_`) scales the unit direction vector, yielding the 3D thrust vector in the local frame.
- **Off-Center Application:** The `AddLinkForce(force, offset)` API applies the force at the specific nozzle offset.

4.6.2 Main Engine Actuation and MECO Logic

During the final descent phase, the engine shutdown is initiated by a critical event known as Main Engine Cut-Off (MECO). Physically, this shutdown does not occur instantaneously; instead, the engines experience a specific transient decay. The starting point of this decay sequence depends directly on the required hovering thrust, which is a function of the actual landing mass. To simulate this behaviour, a dedicated ROS 2 node named `thrusters_controller` was developed in Python (`thrusters.py`). This node operates as the active control loop, commanding the C++ `GazeboRosThruster` plugins attached to the physical links.

Dynamic Mass Parsing and Thrust Initialization

Before initiating the control loop, the node must determine the total flight mass of the vehicle. It directly parses the generated SDF files (`body.sdf`, `passenger.sdf`) using Python Regular Expressions (`re`) to extract the body mass and payload mass, and adds the analytically calculated mass of the four landing legs to compute the exact `total_mass`.

Then, the theoretical total thrust T_0 required to balance the lunar gravity ($g = 1.625$ m/s^2) is calculated as:

$$T_0 = M_{total} \cdot g_{moon} \tag{4.1}$$

This total required thrust is evenly distributed among the active main engines (defined by the `num_engines` parameter).

To guarantee physical realism, the theoretical thrust per engine is mathematically saturated between the minimum throttle boundary (`t_min`) and the absolute hardware maximum (`t_max`).

$$T_{initial} = \max(T_{min}, \min(T_{max}, \frac{T_0}{N_{engines}})) \tag{4.2}$$

```
# Initial thrust calculation
T0 = total_mass * g

# Thrust per single engine
required_thrust_per_engine = T0 / self.num_engines

# Hardware Saturation
saturated_initial_thrust = max(t_min, min(t_max,
    required_thrust_per_engine))
self.initial_throttle = saturated_initial_thrust / self.
    engine_max_thrust
```

Listing 4.15: Calculation of the theoretical thrust and application of hardware saturation limits.

Analytical Inversion of the decay Profile

The adopted decay profile maps the execution time t to a throttle percentage $P(t)$ through four distinct stages:

- **Stage 1** ($0 \leq t \leq 1.35$ **s):** Linear decay from 100% to 50% throttle.
- **Stage 2** ($1.35 < t \leq 2.0$ **s):** Linear decay from 50% to 33% throttle.
- **Stage 3** ($2.0 < t \leq 2.2$ **s):** Constant thrust hold at 33%.

- **Cut-off ($t > 2.2$ s):** Instantaneous drop to 0%.

To interact with the decay profile, the physical thrust force must first be normalized against the absolute engine capacity to obtain the dimensionless initial throttle fraction P_0:

$$P_0 = \frac{T_{initial}}{T_{max}} \tag{4.3}$$

Consequently, the system must enter the decay curve at the exact temporal coordinate t_{start} corresponding to this initial throttle (P_0).

To achieve this, the script performs an analytical inversion of the piecewise linear MECO function, solving $t_{start} = P^{-1}(P_0)$:

- If $P_0 \geq 0.50$, the throttle falls within Stage 1. The line equation is $P(t) = 1.0 - \frac{0.50}{1.35}t$. Inverting for t yields:

$$t_{start} = (1.0 - P_0) \cdot \frac{1.35}{0.50} \tag{4.4}$$

- If $0.33 \leq P_0 < 0.50$, the throttle falls within Stage 2. The line equation is $P(t) = 0.50 - \frac{0.17}{0.65}(t - 1.35)$. Inverting for t yields:

$$t_{start} = 1.35 + (0.50 - P_0) \cdot \frac{0.65}{0.17} \tag{4.5}$$

- If $P_0 < 0.33$, the thrust is already at or below the minimum holding threshold, so the system safely skips the decay phases and directly enters the hold phase at $t_{start} = 2.0$ s.

```
1  # Normalize thrust to get throttle percentage
2  self.initial_throttle = saturated_initial_thrust / self.
       engine_max_thrust
3
4  # Find t_start to seamlessly enter the MECO curve
5  if self.initial_throttle >= 0.50:
6      self.t_start = (1.0 - self.initial_throttle) * (1.35 / 0.50)
7  elif self.initial_throttle >= 0.33:
8      self.t_start = 1.35 + (0.50 - self.initial_throttle) * (0.65 /
       0.17)
9  else:
10     self.t_start = 2.0  # Safe offset directly entering the hold
       phase
```

Listing 4.16: Normalization of the initial thrust and analytical inversion of the piecewise MECO curve.

Real-time Control Loop

The core execution logic is governed by a high-frequency ROS 2 `Timer` running at 50 Hz ($\Delta t = 0.02$ s). During each iteration, the node queries the `Gazebo` clock to synchronize with the physics simulation. Upon receiving the first valid timestamp, it locks the simulation start time. In subsequent iterations, it calculates the elapsed time ($t_{elapsed}$) and computes the shifted timeline for the decay profile by adding the previously derived t_{start}:

$$t_{curve} = t_{start} + t_{elapsed} \tag{4.6}$$

Using t_{curve}, the node evaluates the MECO function to extract the instantaneous throttle fraction. This percentage is immediately converted back into a physical force (in Newtons) by multiplying it by the engine's absolute capacity, yielding the required `current_thrust` per engine.

To support various architectural configurations, the node dynamically routes this force based on the `num_engines` parameter configured in the YAML file:

- **1 Engine:** The thrust is routed exclusively to the `central` topic.
- **2 Engines:** The thrust is applied equally to the `left` and `right` topics, leaving the central node at a default 0.0 N.
- **3 Engines:** The thrust is broadcasted equally across all three topics (`left`, `right`, and `central`).

Propulsive Telemetry (`thrust_logger.py`)

While the `thrusters_controller` is responsible for simulating the propulsion, for the analysis of the landing event can be useful to know exactly how much thrust was being exerted by the engines in the moment of the impact. If the engines were still firing at a high throttle setting during touchdown, the residual thrust could contribute to structural tip-over dynamics or reduce the effective load on the shock absorbers.

To monitor the active force vectors, a dedicated node named `thrust_logger.py` was implemented. The logger initializes by subscribing to the three thrusters topics (`/lander/thrusters/left`, `right`, `central`). It records the incoming `Float64` force messages, storing them in dedicated data arrays alongside a localized relative timestamp ($t_{relative} = t_{sim} - t_{start}$).

Concurrently, it subscribes to the `/lander/t_start` topic to acquire the temporal offset injected by the MECO controller. This is necessary to correctly align the analytical phase-shift markers (e.g., t_1, t_2, MECO) on the final telemetry plots, which are generated using the Matplotlib library [12].

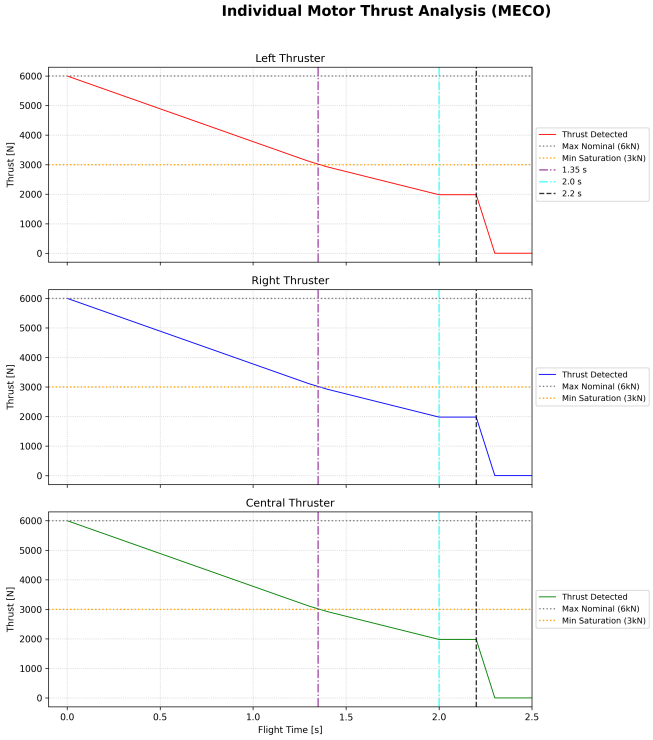


Figure 4.2: Individual motor thrust plot.

4.7 Plume Surface Interaction (PSI) Disturbance Generation

As formalized in Section 3.4, the expansion of exhaust gases against the lunar regolith generates significant stochastic aerodynamic back-pressure during the final meters of descent. To simulate this behaviour a dedicated ROS 2 node, `psi_disturbance_node`, was engineered.

4.7.1 Dynamic True Altitude Tracking

The fundamental variable driving the PSI disturbance magnitude is the *True Altitude* (h), defined as the instantaneous vertical distance between the lander's base and the highly irregular lunar terrain. Relying solely on the absolute Z coordinate from the Gazebo telemetry is insufficient, as the lander might be hovering over a crater or an elevated slope.

To solve this, the `get_terrain_elevation` function was created. The node extracts the lander's global (X, Y) position from the `/gazebo/model_states` topic.

Then, by generating a synthetic ray from an arbitrary height (`max_z`) pointing straight down towards the lunar surface, the function calculates the exact intersection geometry. The instantaneous true altitude is then dynamically evaluated by subtracting this local terrain elevation from the lander's absolute Z coordinate.

```
def get_terrain_elevation(self, x, y, max_z=200.0):
    if self.mesh is None:
        return 0.0
    # Cast a downward vertical ray from above the bounding box
    ray_origins = np.array([[x, y, max_z]])
    ray_directions = np.array([[0, 0, -1]])

    locations, index_ray, index_tri = self.mesh.ray.
    intersects_location(ray_origins=ray_origins, ray_directions=
    ray_directions)

    if len(locations) > 0:
        # Return the highest Z interception point (top surface of
    the terrain)
        return np.max(locations[:, 2])
    return 0.0
```

Listing 4.17: Python implementation of the terrain elevation raycasting using the `trimesh` library.

4.7.2 Disturbance Calculation and Thrust Synchronization

The generation of the disturbance force relies on two physical conditions: the active state of the main engines and the proximity to the lunar surface.

Because this phenomenon exists only as long as the engines are firing, the node must tightly synchronize with the propulsion system. It achieves this by subscribing directly to the thruster topics (`/lander/thrusters/*`). If the sum of the commanded thrusts drops to zero, a boolean flag `main_engine_off` is permanently toggled to `True`. From that exact microsecond, the control loop bypasses all altitude evaluations and outputs 0.0 N. As long as the engines remain active, the node evaluates the true altitude against an empirical quadratic function to determine the resulting disturbance magnitude (as detailed in Section 3.4).

```
def calculate_quadratic_disturbance(self, h):
    if h >= 10.0:
        return 0.0

    # Quadratic interpolation based on empirical data
    a = 2.587
    b = -41.800
    c = 159.333
```

```
9       force = (a * (h ** 2)) + (b * h) + c
10
11      # Safety saturation
12      return max(0.0, min(force, 500.0))
```

Listing 4.18: Empirical calculation of the total disturbance magnitude,featuring altitude-based cut-offs and hardware saturation limits.

4.7.3 Spatial Modulation and ROS 2 Publishing

Following the mathematical framework established previously (in Section 3.4), the script dynamically modulates the computed force into the four cardinal components. It calculates the elapsed time since the disturbance profile was triggered and applies the half-wave rectified sinusoidal distribution directly via the Python `math` library.

These dynamically calculated components are then packaged into standard `Float64` messages and published continuously to dedicated ROS 2 topics (e.g., `/lander/disturbance/north`). The C++ `GazeboRosThruster` plugins attached to the lander's main body subscribe to these topics, translating the incoming data into active multi-body wrenches applied directly within the physics engine.

```
1  # Elapsed time since the disturbance profile triggered
2  t_elapsed = (now - self.start_time).nanoseconds / 1e9
3  phase = self.omega * t_elapsed
4
5  # Half-wave rectified sinusoidal distribution
6  f_north = total_force * max(0.0, math.sin(phase))
7  f_east  = total_force * max(0.0, math.cos(phase))
8  f_south = total_force * max(0.0, -math.sin(phase))
9  f_west  = total_force * max(0.0, -math.cos(phase))
10
11 self.publish_forces(f_north, f_east, f_south, f_west, rel_time)
```

Listing 4.19: Temporal modulation of the disturbance vector mapping mathematical models to ROS 2 topic publishing.

4.8 Dynamic Contact Mechanics

As theoretically established in Section 3.6, the lunar surface is not a homogeneous rigid plane; it consists of a shallow layer of highly compressible regolith dust resting upon a solid rock. To model this *dual-layer soil* behavior, a custom C++ WorldPlugin named `DynamicTerrainPlugin` was designed.

4.8.1 Plugin Initialization and Parameter Parsing

Unlike standard `ModelPlugin` architectures, which attach to specific moving vehicles, a `WorldPlugin` oversees the entire global simulation environment. Within the `Load` function, the plugin first captures the global `physics::WorldPtr` to interact with the physics engine, and instantiates its internal ROS 2 node pointer (`gazebo_ros::Node`) via the provided `sdf::ElementPtr`. Subsequently, it scans the `landing_scenario.world` XML tree to register the mechanical properties for the soft layer (`kp_surface`, `kd_surface`, `mu_surface`), the hard layer (`kp_rock`, `kd_rock`, `mu_rock`), and the topological threshold (`dust_thickness`). After resolving the `terrain_name` (the specific collision mesh to monitor), it hooks its core evaluation method into the `ConnectWorldUpdateBegin` Gazebo event loop.

4.8.2 Intercepting the Gazebo Contact Manager

The plugin directly queries the Gazebo `ContactManager`, which holds an array of all active collisions occurring anywhere in the virtual world.

The plugin iterates through this array. For each contact pair (`collision1` and `collision2`), it checks if one of the interacting bodies is the lunar terrain mesh. If a terrain impact is identified, it extracts the exact penetration depth of the colliding vertices. Because a single footpad impact generates multiple simultaneous micro-contacts, the plugin calculates the average depth across the contact patch to prevent numerical jitter.

```
1  auto contacts = this->world->Physics()->GetContactManager()->
       GetContacts();
2
3  for (auto const &contact : contacts) {
4      physics::Collision *terrain_collision = nullptr;
5
6      // Identify which collision body belongs to the terrain
7      if (contact->collision1->GetModel()->GetName() == this->
       terrain_name) {
8          terrain_collision = contact->collision1;
9      } else if (contact->collision2->GetModel()->GetName() == this
       ->terrain_name) {
10         terrain_collision = contact->collision2;
11     }
12
13     if (terrain_collision != nullptr) {
14         double depth = 0.0;
15         // Average the penetration depth across all contact points
16         for (int i = 0; i < contact->count; ++i) {
17             depth += contact->depths[i];
18         }
19         if (contact->count > 0) depth /= contact->count;
```

```
20
21          // [...] (Proceed to interpolation logic)
```

Listing 4.20: Extraction and averaging of the geometric penetration depth from the Gazebo `ContactManager`.

4.8.3 Real-Time Parameter Interpolation and Injection

Using the averaged penetration `depth`, the plugin executes the mathematical dual-layer formulation. It computes (as detailed in Section 3.6.2) the dimensionless interpolation parameter α, the instantaneous contact stiffness (`current_kp`), the viscous damping (`current_kd`) and the Coulomb friction coefficient (`current_mu`) that are linearly interpolated between the soft surface state and the hard bedrock state.

The final step is injecting these new properties back into the physics engine before the forces are resolved. The plugin dynamically casts the terrain's generic surface pointer into an `ODESurfaceParams` object, allowing the modification of the spring-damper constants. Concurrently, it updates the primary and secondary axes of the `FrictionPyramid`, instantly altering the shear resistance of the soil.

```
1  // Calculate interpolation factor
2  double alpha = depth / this->dust_thickness;
3  if (alpha > 1.0) alpha = 1.0;
4  if (alpha < 0.0) alpha = 0.0;
5
6  // Interpolate mechanical properties
7  double current_kp = this->kp_surface + alpha * (this->kp_rock -
       this->kp_surface);
8  double current_kd = this->kd_surface + alpha * (this->kd_rock -
       this->kd_surface);
9  double current_mu = this->mu_surface + alpha * (this->mu_rock -
       this->mu_surface);
10
11 // Inject stiffness and damping into ODE
12 physics::ODESurfaceParams *ode_surface =
13     dynamic_cast<physics::ODESurfaceParams*>(terrain_collision->
       GetSurface().get());
14 if (ode_surface) {
15     ode_surface->kp = current_kp;
16     ode_surface->kd = current_kd;
17 }
18
19 // Inject shear friction into ODE
20 auto friction = terrain_collision->GetSurface()->FrictionPyramid()
       ;
21 if (friction) {
```

```
22      friction->SetMuPrimary(current_mu);
23      friction->SetMuSecondary(current_mu);
24  }
```

Listing 4.21: Dynamic modification of the ODE surface parameters and friction pyramid.

4.9 Automated Data Logging and Evaluation Criteria

To support an unsupervised, high-volume statistical campaign, the simulation environment cannot rely on manual data extraction. It requires a dedicated, automated node capable of tracking the high-frequency kinematics, monitoring ground impact mechanics, and terminating the simulation independently once the touchdown transient resolves.

To achieve this, the Python node `flight_data_logger.py` was developed. It acts as the central intelligence of the MiL architecture, continuously parsing data from two native Gazebo ROS 2 plugins (`libgazebo_ros_state.so` and `libgazebo_ros_bumper.so`), evaluating mission success criteria, and generating comprehensive post-flight plots.

4.9.1 Node Architecture and Middleware Subscriptions

During its initialization phase, the `FlightDataLogger` constructs its internal memory arrays to store physical variables over time and parses the `config.yaml` file which dictates the thresholds for the simulation's automatic termination logic.

To monitor the physical environment, the node acts as a passive observer, weaving a network of ROS 2 subscriptions and publishers to exchange data with the active Gazebo plugins and the lander's flight controller:

`/gazebo/model_states` (Subscriber): It receives high-frequency `ModelStates` messages containing the absolute 3D position, orientation, linear velocity, and angular velocity of all models spawned in the world. It is the primary source of kinematic telemetry.

`/lander/contact_forces` (Subscriber): Binds to the Gazebo Bumper plugin attached to the lander's footpads. It receives `ContactsState` messages to monitor ground reaction forces, determining the exact millisecond of the first touchdown and extracting peak impact loads.

`/lander/impact_thrust` (Subscriber): Listens to the flight controller to record the exact magnitude of the main engine's thrust force actively delivered at the moment of ground impact.

`/lander/stop_logging` (Publisher): A control channel used by the logger to broadcast a boolean trigger. Once the lander has reached a steady state on the lunar surface, the logger fires this flag to command other auxiliary nodes (and the simulation engine itself) to cleanly terminate the current run.

`/lander/touchdown_time` (Publisher): Broadcasts the timestamp of the first detected ground contact.

This interconnected architecture allows the logger to operate completely unsupervised, synthesizing kinematic tracking and force evaluation into a unified mission report.

4.9.2 Kinematic Telemetry via Gazebo ROS State Plugin

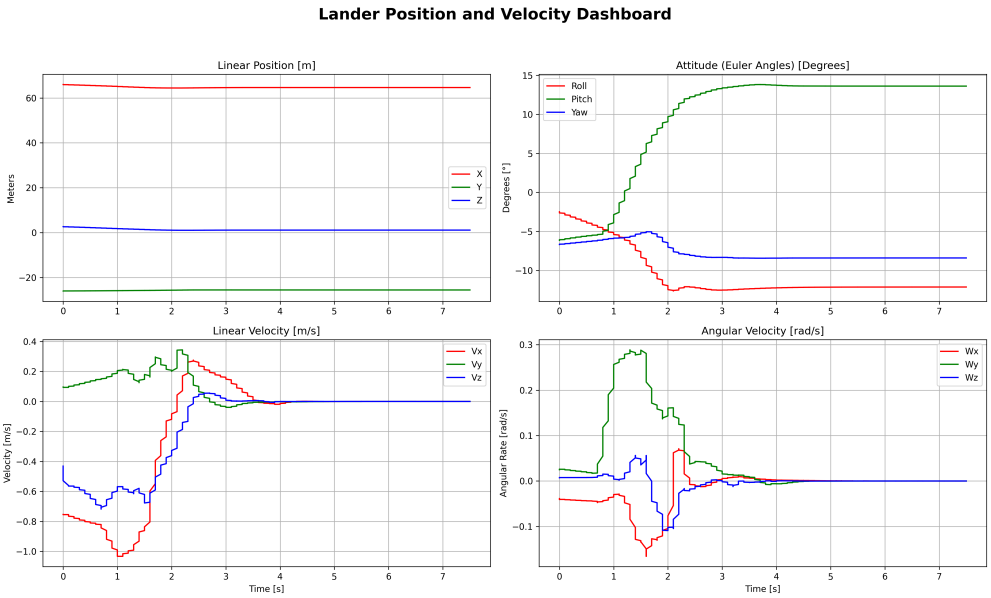


Figure 4.3: Kinematic telemetry profile over the course of the simulation.

To record the kinematic flight profile of the lander, the telemetry logger subscribes to the global `/gazebo/model_states` topic, which is continuously populated by Gazebo's native `libgazebo_ros_state` plugin. In the first place, the

`telemetry_callback` verifies the active recording status and isolates the data corresponding to the `lander_model`.

Once the correct model index is identified, the node calculates a localized relative timestamp ($t = t_{now} - t_{start}$) to ensure all data points are properly synchronized from the moment of initialization. It then sequentially extracts the instantaneous linear position (x, y, z) and linear velocity (v_x, v_y, v_z) from the corresponding `Pose` and `Twist` message structures.

Furthermore, because Gazebo inherently computes rigid-body orientations using quaternions to prevent gimbal lock, the raw angular pose data (q_x, q_y, q_z, q_w) requires mathematical transformation. The node applies a standard trigonometric conversion to extract the Roll, Pitch, and Yaw angles. These values, alongside the angular rates $(\omega_x, \omega_y, \omega_z)$, are immediately converted from radians to degrees and appended to the internal memory arrays.

Automated Simulation Termination Logic

This framework is designed to run unsupervised, the data logger is responsable to decide when a simulation run is complete. Once the first touchdown is detected (handled via the contact bumper topic), the `telemetry_callback` activates two parallel evaluation tracks to trigger a clean shutdown: the *Steady State Control* and the *Timeout Control*.

The Steady State Control works by checking if the absolute values of all linear and angular velocities fall below a strict threshold (ϵ) for a continuous predefined duration (`steady_req_duration`).

To prevent the simulation from running indefinitely in case the lander becomes physically stuck in a jittering state, the Timeout Control serves as a safety cut-off. If the time elapsed since touchdown exceeds `timeout_after_touchdown`, the simulation is forcefully terminated regardless of the kinetic state.

When the logger stops recording data, it publishes a `True` boolean to the `/lander/stop_logging` topic to signal other nodes, and initiates the final data export sequence.

```
# Timeout Control
if (now - self.touchdown_time) > self.timeout_after_touchdown:
    if not self.shutdown_initiated:
        self.shutdown_initiated = True
        self.is_recording = False
        self.get_logger().info("TIMEOUT reached after touchdown.")
        msg = Bool()
        msg.data = True
        self.pub_stop.publish(msg)
        self.create_timer(1.5, self._trigger_exit)

# Steady State Control (Speed below epsilon)
```

```
13 is_steady = (
14     abs(vx) < self.epsilon and abs(vy) < self.epsilon and abs(vz)
       < self.epsilon and
15     abs(wx) < self.epsilon and abs(wy) < self.epsilon and abs(wz)
       < self.epsilon
16 )
17
18 if is_steady:
19     if self.steady_start_time is None:
20         self.steady_start_time = now
21     elif (now - self.steady_start_time) > self.steady_req_duration
       :
22         if not self.shutdown_initiated:
23             self.shutdown_initiated = True
24             self.is_recording = False
25             self.get_logger().info(f"STEADY STATE reached (
       Variables < {self.epsilon}).")
26             msg = Bool()
27             msg.data = True
28             self.pub_stop.publish(msg)
29             self.create_timer(1.5, self._trigger_exit)
30 else:
31     self.steady_start_time = None
```

Listing 4.22: Steady state and timeout control logic for automated simulation termination.

4.9.3 Contact Mechanics via Gazebo ROS Bumper Plugin

To capture the ground impact 3D reaction forces, a `libgazebo_ros_bumper` plugin is attached to the collision geometries of each footpad link in the SDF model. This plugin streams high-frequency `ContactsState` messages to the `/lander/contact_forces` topic. The incoming contact states are processed asynchronously by the `contact_callback` function.

Micro-Contact Multiplicity and Force Normalization

When a rigid footpad makes contact with 3D terrain mesh, the physics solver generates multiple discrete contact points across the footpad's collision geometry within a single simulation step. Each reported state contains an individual force wrench vector.

Directly summing these wrenches across all active sub-contacts for a single leg would artificially multiply the true physical load proportional to the mesh contact density, leading to unrealistically high impact force spikes. To resolve this discretization artifact, the logger implements a dynamic normalization algorithm:

1. It iterates over all collision states in the message, identifying which landing leg (`leg1` through `leg4`) corresponds to the active collision name.
2. For each leg, it accumulates the total force components (F_x, F_y, F_z) and increments a contact counter.
3. If active micro-contacts are detected, the cumulative force vector is divided by the total number of contacts for that specific leg:

$$\mathbf{F}_{\text{net, leg}} = \frac{1}{N_{\text{contacts}}} \sum_{i=1}^{N_{\text{contacts}}} \mathbf{F}_i \tag{4.7}$$

This averaging procedure yields a physically consistent resultant force vector $\mathbf{F}_{\text{net}}$ representing the true structural load transmitted through each strut.

```
def contact_callback(self, msg):
    if not self.is_recording:
        return

    now = self.get_clock().now().nanoseconds / 1e9
    if self.start_time_forces is None:
        self.start_time_forces = now

    self.times_forces.append(now - self.start_time_forces)

    contact_counts = {leg: 0 for leg in self.legs}
    new_f = {leg: {'x': 0.0, 'y': 0.0, 'z': 0.0} for leg in self.
    legs}

    # Accumulate forces and count active contact points per leg
    for state in msg.states:
        col_name = state.collision1_name
        for leg in self.legs:
            if leg in col_name:
                new_f[leg]['x'] += state.total_wrench.force.x
                new_f[leg]['y'] += state.total_wrench.force.y
                new_f[leg]['z'] += state.total_wrench.force.z
                contact_counts[leg] += 1

    # Normalize accumulated forces by the number of active micro-
    contacts
    for leg in self.legs:
        if contact_counts[leg] > 0:
            self.current_f[leg]['x'] = (new_f[leg]['x'] /
    contact_counts[leg])
            self.current_f[leg]['y'] = (new_f[leg]['y'] /
    contact_counts[leg])
            self.current_f[leg]['z'] = (new_f[leg]['z'] /
    contact_counts[leg])
```

```
30
31      # Record normalized force history
32      for leg in self.legs:
33          for axis in self.axes:
34              self.history_forces[leg][axis].append(self.current_f[
    leg][axis])
```

Listing 4.23: Contact force accumulation and normalization across active micro-contacts.

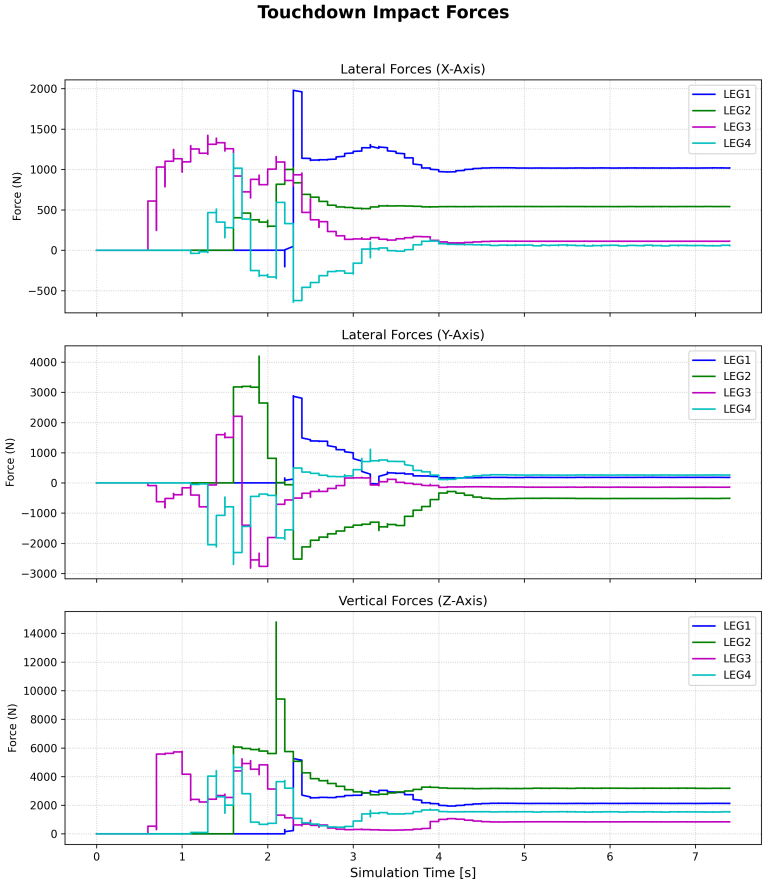


Figure 4.4: Temporal evolution of the 3D contact forces (F_x, F_y, F_z).

Touchdown Event Detection and Time Synchronization

The `contact_callback` also continuously monitors the normalized vertical reaction force (F_z) across all four legs. The exact microsecond any footpad records a vertical

force magnitude exceeding a safety threshold of $|F_z| > 5.0\,\text{N}$, the node formally registers a `TOUCHDOWN DETECTED` event. After the detection, the logger:

- Caches the current simulation timestamp (`touchdown_time = now`).
- Identifies and logs the specific leg responsible for primary impact.
- Instantiates a `Float64` message containing the exact touchdown timestamp and publishes it onto the `/lander/touchdown_time` topic.

```
    # Touchdown Detection and Timestamp Broadcasting
    if self.touchdown_time is None:
        for leg in self.legs:
            if abs(self.current_f[leg]['z']) > 5.0:
                self.touchdown_time = now
                self.first_leg_touch = leg
                self.get_logger().info(f"TOUCHDOWN DETECTED! ({leg}). Timers started.")

                # Broadcast touchdown timestamp for system-wide synchronization
                msg_td = Float64()
                msg_td.data = now
                self.pub_touchdown.publish(msg_td)
                break
```

Listing 4.24: Touchdown event detection and timestamp broadcasting logic within the callback.

4.9.4 Post-Flight Data Processing

Once the termination logic triggers the `rclpy.shutdown()` command, the node exits its active listening loop and transitions into the final data extraction and processing phase. During this stage, the buffered telemetry arrays are mathematically evaluated to compute the definitive mission metrics for the current run.

These extracted metrics are subsequently appended as a single, comprehensive row to a .csv file (`montecarlo_summary.csv`). This file is the database for the entire simulation campaign, it explicitly maps the procedurally generated initial conditions (mass properties, spatial kinematics, legs coefficient and soil mechanics) of every single landing event to its final dynamic outcome.

Initial State and True Altitude Raycasting

The first operation is to freeze the initial kinematic state of the lander at $t = 0$. To ensure the initial vertical position accurately reflects the real altitude above the

mesh, the logger utilizes the `trimesh` library. By casting an array of vertical rays downwards from the initial (x, y) coordinates to the underlying lunar mesh, the absolute Z coordinate is offset by the local terrain elevation.

Global CoG and Inertia Reconstruction

Because the lander's mass is distributed across highly modular SDF files (body, passenger payload, and four independent legs), the logger must reconstruct the global inertial properties analytically. The node parses the local mass, CoG, and Inertia tensor (I) of each sub-component. For the landing legs, the local CoG and inertia matrices are rotated by the respective yaw installation angles $(0°, 90°, 180°, 270°)$ using standard $SO(3)$ rotation matrices. Finally, the global CoG is calculated via a weighted average, and the total inertia tensor is assembled using the parallel axis theorem (Steiner's transport).

```
# Calculate Total Mass and Total CoG
total_mass = sum(p["mass"] for p in parts)
if total_mass > 0:
    tot_cog = sum(p["mass"] * p["cog_global"] for p in parts) / total_mass
else:
    tot_cog = np.zeros(3)

# Calculate Total Inertia (using Steiner's Theorem)
tot_I = np.zeros((3, 3))
for p in parts:
    tot_I += steiner_transport(p["I_global"], p["mass"], p["cog_global"], tot_cog)

# Steady-State Mass Physics Verification
final_fz = sum([f_imp[leg]['z'][-1] for leg in node.legs if len(f_imp[leg]['z']) > 0])
gazebo_mass = abs(final_fz) / 1.625 # Lunar gravity = 1.625 m/s^2
mass_error_kg = gazebo_mass - total_mass
```

Listing 4.25: Analytical reconstruction of global mass CoG and Inertia tensor.

The node also performs a steady-state physics check, comparing the analytical `total_mass` against the perceived mass derived from the final equilibrium ground reaction forces (`gazebo_mass`).

Impact Metrics and Survivability Evaluation

From this specific moment, several critical metrics are extracted:

Impact Velocity. The script isolates the Cartesian velocity components exactly at the moment of first ground contact (using the `td_index` array position). The

absolute scalar magnitude of the velocity vector is then calculated via the Euclidean norm.

```
# Impact velocity
vx_imp = vx[td_index]
vy_imp = vy[td_index]
vz_imp = vz[td_index]
v_abs_imp = math.sqrt(vx_imp**2 + vy_imp**2 + vz_imp**2)
```

Listing 4.26: Extraction of the absolute impact velocity vector.

Specific Kinetic Energy. Evaluated as $E_k/m = \frac{1}{2}|\mathbf{v}_{\text{imp}}|^2$.

```
# Specific kinetic energy (J/kg)
kin_energy_spec = 0.5 * (v_abs_imp**2)
```

Listing 4.27: Calculation of the specific kinetic energy.

Impact Verticality. This metric evaluates the deviation angle (θ) between the lander's 3D velocity vector at touchdown and the downward local gravity unit vector $[0, 0, -1]^T$. Mathematically, it is derived using the dot product, which simplifies to $\cos(\theta) = |v_z|/|\mathbf{v}_{\text{imp}}|$. To prevent numerical exceptions during the arccosine evaluation due to floating-point inaccuracies, the cosine value is strictly clamped between -1.0 and 1.0.

```
# Verticality (Dot product between: [vx, vy, vz] and [0, 0, -1])
cos_theta = abs(vz_imp) / v_abs_imp if v_abs_imp > 0 else 0.0
cos_theta = max(-1.0, min(1.0, cos_theta))
impact_angle_deg = math.degrees(math.acos(cos_theta))
```

Listing 4.28: Calculation of the impact verticality angle using the dot product.

Residual Vertical Acceleration. To assess the dynamic state of the vehicle at the exact moment of impact, the logger computes the residual vertical acceleration (a_z). This metric is derived numerically via a backward finite difference method. The algorithm calculates the slope of the vertical velocity (dv_z/dt) over a short temporal window (spanning the last five telemetry samples) immediately preceding the touchdown event (`td_index`).

```
# Residual acceleration (Derivative of velocity in the last
    moments before td)
accel_residua_z = 0.0
if td_index > 5:
    dt = t_tel[td_index] - t_tel[td_index - 5]
    dvz = vz[td_index] - vz[td_index - 5]
    accel_residua_z = dvz / dt if dt > 0 else 0.0
```

Listing 4.29: Numerical derivation of the residual vertical acceleration just before touchdown.

Primary Contact Leg. To analyze the asymmetry of the landing and predict the initial structural load distribution, the logger extracts the identity of the footpad that made first contact with the lunar surface. This identifier (e.g., `leg1`) is retrieved from the memory variable `first_leg_touch`, which is populated asynchronously by the contact callback at the exact moment the vertical reaction force threshold is exceeded.

Peak Structural Load. To evaluate the maximum stress imparted to the landing gear system, the node scans the entire high-frequency force history recorded from initial ground contact to final equilibrium. By iterating over the arrays of all four legs and extracting the absolute highest vertical reaction force (F_z) registered by any single footpad.

```
1 # Max. leg force
2 max_leg_force = 0.0
3 if len(t_forces) > 0:
4     max_leg_force = max([max(f_imp[leg]['z'], default=0.0) for leg
      in node.legs])
```

Listing 4.30: Extraction of the maximum vertical reaction force experienced by any landing leg.

Settling Time. The settling time quantifies the duration of the transient impact phase. It is calculated as the time elapsed between the primary touchdown event (when the first leg strikes the surface) and the exact moment the simulation terminates (when the vehicle satisfies the steady-state kinetic criteria or hits the safety timeout). Extended settling times are critical indicators of landing instability, often highlighting prolonged bouncing, lateral sliding across sloped terrain, or slow tip-over dynamics before the vehicle finally comes to rest.

```
1 # Settling time
2 if node.touchdown_time is not None and node.start_time_tel is not
      None:
3     td_time_relative = node.touchdown_time - node.start_time_tel
4     settling_time = t_tel[-1] - td_time_relative
5 else:
6     settling_time = 0.0
```

Listing 4.31: Calculation of the transient settling time from touchdown to steady state.

Final Resting Tilt. To evaluate the ultimate stability of the lander on the lunar surface, the script extracts the absolute values of the final attitude components (`final_roll`, `final_pitch`, and `final_yaw`) from the end of the telemetry lists. Because the yaw angle represents a planar rotation around the vertical axis, it does not contribute to the vehicle's physical inclination relative to gravity. Thus, the total tilt is computed by projecting the lander's local vertical axis onto the global Z-axis using only the roll and pitch angles converted to radians. Mathematically, this corresponds to the dot product of the two vertical unit vectors, yielding the relationship $\cos(\theta_{\text{tilt}}) = \cos(\text{pitch}) \cdot \cos(\text{roll})$. A mathematical clamping function is applied to prevent floating-point domain errors before extracting the final angle via the arccosine function.

```
# Final attitude
final_roll = abs(ro[-1])
final_pitch = abs(pi[-1])
final_yaw = abs(ya[-1])

# Tip-over (Dot product between local lander's z-axis and [0, 0, 1])
final_r_rad = math.radians(final_roll)
final_p_rad = math.radians(final_pitch)
dot_product_z = math.cos(final_p_rad) * math.cos(final_r_rad)
dot_product_z = max(-1.0, min(1.0, dot_product_z))
total_tilt_deg = math.degrees(math.acos(dot_product_z))
```

Listing 4.32: Extraction of final attitude components and calculation of the final tip-over angle at steady state.

Maximum Dynamic Tilt. A lander may eventually settle at a safe final attitude but experience severe, unstable oscillations immediately following touchdown. To capture this critical transient behavior, the logger continuously evaluates the instantaneous tip-over angle by applying the same trigonometric projection at every telemetry time step. Starting strictly from the touchdown index (`td_index`) up to the end of the simulation arrays, it iterates through the recorded roll and pitch sequences, computes the dynamic dot product, and isolates the maximum angle recorded during this window as the `max_dynamic_tilt_deg`. This provides a vital metric for identifying near-miss tip-over events or excessive rocking dynamics.

```
# Max tilt angle
max_dynamic_tilt_deg = 0.0
if td_index < len(t_tel):
    dynamic_tilts = []
    for i in range(td_index, len(t_tel)):
        r_rad = math.radians(ro[i])
        p_rad = math.radians(pi[i])
```

```
8           dot_z_dyn = math.cos(p_rad) * math.cos(r_rad)
9           dot_z_dyn = max(-1.0, min(1.0, dot_z_dyn))
10          tilt_inst_deg = math.degrees(math.acos(dot_z_dyn))
11          dynamic_tilts.append(tilt_inst_deg)
12
13      if dynamic_tilts:
14          max_dynamic_tilt_deg = max(dynamic_tilts)
```

Listing 4.33: Continuous evaluation of the instantaneous attitude to extract the maximum dynamic oscillation.

Impact Thrust. To account for the active propulsion forces acting on the spacecraft during the critical touchdown phase, the logger extracts the cumulative or snapshot impact thrust value managed by the node. This metric captures the active contribution of the descent engines at the exact moment of ground contact, which is subsequently mapped into the summary database to evaluate the engine interaction during impact.

Belly Clearance and Crash Flags. A landing is only successful if the central structure and engine nozzles do not strike the ground. The logger generates a dense grid of points representing the underside of the lander (the "belly"). At each time step after touchdown, it applies a 3D transformation matrix (`R_mat`) based on the vehicle's instantaneous Euler angles to map these points into global space. By raycasting downward from these rotated points to the terrain mesh, it calculates the minimum `belly_clearance_m`. If this distance drops below zero, a `flag_hull_crash` is raised. A secondary, tighter clearance threshold is applied specifically to points corresponding to the main engines to raise a `flag_thruster_crash`.

Chapter 5

Sensitivity Analysis and Conclusion

Building upon the theoretical foundations established in Section 2.4, this chapter outlines the practical execution of the Global Sensitivity Analysis (GSA) and the results.

5.1 Rationale for the Sequential Approach in High-Fidelity Simulations

The dynamic model of the lunar lander developed in this study relies on a highly complex, multi-dimensional input space comprising $k = 27$ independent variables. These parameters govern diverse physical domains: the initial spatial kinematics (velocities, attitudes, angular rates), the variable mass properties (dry body configurations, passenger payloads, propellant residuals), and the mechanical coefficients of the elastoplastic shock absorbers.

Applying a rigorous, variance-based analysis, such as the Sobol method, directly to a 27-dimensional space presents an insurmountable computational bottleneck, a phenomenon widely known in numerical analysis as the *Curse of Dimensionality*. Variance-based methods require a massive number of model evaluations to accurately estimate first-order and total-order interactions. In the context of the Saltelli sampling strategy, the total number of required simulations scales as $N(k + 2)$, where N is the base sample size (typically $10^3 \sim 10^4$). For 27 variables, this would easily translate to hundreds of thousands of independent iterations. While mathematically ideal, the computational cost becomes strictly prohibitive when each iteration requires spinning up a high-fidelity Gazebo physics environment, rendering a direct Sobol approach unfeasible for this specific architecture.

To overcome this constraint without compromising the statistical rigor of the study, the sequential two-step methodology introduced in Section 2.4 was strictly enforced:

1. **Screening Phase (Morris Method):** Acting as a computationally efficient filter, this qualitative global sensitivity analysis is deployed across the entire $k = 27$ parameter space. Requiring only $r(k + 1)$ simulations (where r is the number of trajectories, typically between 10 and 50), it operates within a highly constrained computational budget. Its primary objective is to evaluate both linear and interactive effects to decisively separate the "few important" parameters from the "many unimportant" ones.

2. **Variance-Based Phase (Sobol Method):** Once the Morris screening has drastically reduced the dimensionality of the problem by identifying and freezing the non-influential variables, the computationally expensive Sobol indices can be targeted exclusively at these dominating factors. This final phase yields a highly accurate, quantitative decomposition of the model's output variance, identifying exactly how much of the final uncertainty is driven by each specific input parameter and their mutual interactions.

5.2 First Simulation Campaign

This first simulation campaign was designed as an unconstrained exploration of the landing dynamics. During this phase, all $k = 27$ independent parameters were allowed to vary simultaneously across their maximum operational bounds. This included not only the internal vehicle properties but also the absolute spatial spawn coordinates (x, y), forcing the lander to interact randomly with the complex topographical features of the lunar mesh.

5.2.1 Phase I: Global Parameter Screening

Design of Experiments and Computational Budget

To achieve statistical significance and avoid local anomalies, the algorithm does not rely on a single trajectory. Instead, it generates r independent trajectories, starting from r different regions of the input space. The total number of computationally expensive physics simulations N required by the Morris campaign is purely deterministic and follows the relation:

$$N = r \times (k + 1) \tag{5.1}$$

For this specific analysis on the lunar lander, the problem comprises $k = 27$ parameters. In order to force mathematical convergence of the sensitivity indices

while maintaining a manageable computational footprint, the number of trajectories was set to $r = 30$. This configuration generated a rigorous execution matrix of:

$$N = 30 \times (27 + 1) = 840 \text{ independent runs} \tag{5.2}$$

To execute this 840-run matrix, a custom Python orchestrator was developed. The generation of the parametric grid was handled utilizing the Sensitivity Analysis Library (`SALib`) [13] , an open-source Python package widely employed for complex system evaluations. The orchestrator maps the sequences generated by `SALib` into physical variables and, for each iteration, dynamically overwrites the centralized `config.yaml` file before launching the Gazebo simulation in a headless (no-GUI) state.

Results and Physical Interpretation

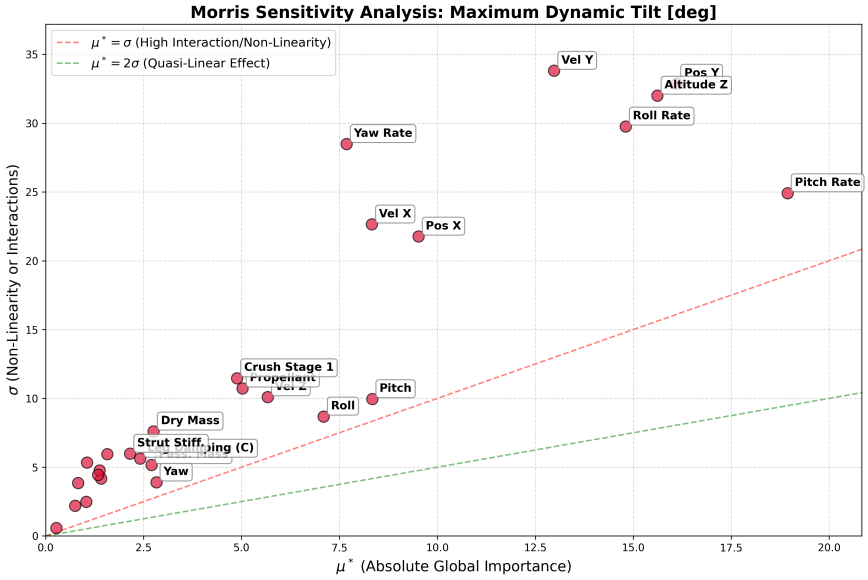


Figure 5.1: Morris Sensitivity Analysis: Scatter plot of Global Importance (μ^*) versus Non-Linearity/Interactions (σ) for the Maximum Dynamic Tilt.

The sensitivity analysis was primarily focused on the maximum dynamic tilt angle, which serves as the ultimate metric for landing stability. The resulting Morris dispersion plot (Figure 5.1) clearly clusters the 27 parameters into three distinct tiers of criticality.

1. **Kinematic and Topographic Dominance (Tier 1):** The system is overwhelmingly dominated by the residual rotational kinetic energy at touchdown

(`Pitch Rate`, `Roll Rate`, `Yaw Rate`) and the spatial interaction with the terrain (`Pos Y`, `Pos X`, `Altitude Z`). Notably, for these spatial coordinates, the interaction index σ is significantly higher than the global importance μ^*, this is the signature of the terrain topography: a 2-meter horizontal shift yields a near-zero effect on a flat plain, but triggers a catastrophic rollover if it places the lander on a crater edge.

2. **Mass and Attitude Modulators (Tier 2):** Parameters such as `Pitch`, `Roll`, and `Propellant Mass` present moderate sensitivity indices. An asymmetric initial attitude exacerbates the landing dynamics, but the vehicle is generally capable of absorbing the impact provided there is no excessive residual angular velocity.

3. **Hardware Robustness (Tier 3):** All the mechanical parameters governing the shock absorbers (`Leg Damping`, `Strut Stiffness`, `Foot Stiffness`, etc.) reside at the bottom of the ranking ($\mu^* < 2.5$). This means that variations within the operational envelope of the suspension system do not significantly trigger instability or rollover events.

The implementation of the Morris screening method successfully fulfilled its objective. By demonstrating that the mechanical parameters of the landing gear have a negligible impact on the global stability boundary compared to kinematic and topographic factors, the dimensionality of the problem can be drastically reduced.

5.2.2 Phase II: Variance-Based Analysis (Sobol Method)

5.2.3 Design of Experiments and Computational Execution

Relying on the theoretical framework detailed in Section 2.4.2, Saltelli's extension of the Sobol Low-Discrepancy Sequence was deployed to construct the execution grid. Rather than reiterating the internal construction of the base and hybrid matrices, it is critical to focus on the operational cost of this sampling scheme, which dictates a deterministic number of runs:

$$N_{runs} = N \times (k + 2) \tag{5.3}$$

where N represents the base sample size. To preserve the optimal space-filling and uniformity properties of the Sobol sequence, N must strictly be a power of two. For this evaluation, a base sample size of $N = 256$ (2^8) was selected.

This rigid mathematical requirement explicitly validates the necessity of the sequential methodology (Morris followed by Sobol) adopted in this study. If the

variance-based analysis had been naively applied to the entire original parameter space ($k = 27$), the required computational footprint would have been:

$$N_{runs_unfiltered} = 256 \times (27 + 2) = 7424 \text{ evaluations}$$

Given the physics engine's execution, this would have translated to over 80 hours of uninterrupted computation for a single campaign. Instead, by leveraging the Morris screening to isolate only the $k = 13$ genuine drivers of instability, the computational matrix was reduced:

$$N_{runs_optimized} = 256 \times (13 + 2) = 3840 \text{ evaluations} \tag{5.4}$$

To physically execute this optimized 3840-run matrix, the Python orchestrator utilized in the screening phase was adapted for variance-based sampling. The generation of the Saltelli parameter grid was once again handled via the `SALib` library. Crucially, the orchestrator's configuration updater was fundamentally restructured to enforce the dimensionality reduction: the lowest-ranking mechanical and topographic parameters were strictly hard-coded to their nominal design values, only the 13 Tier 1 and Tier 2 variables were dynamically injected into the centralized `config.yaml` file for each iteration.

This architectural decision successfully halved the execution time to approximately 42 hours.

Results and Engineering Discussion

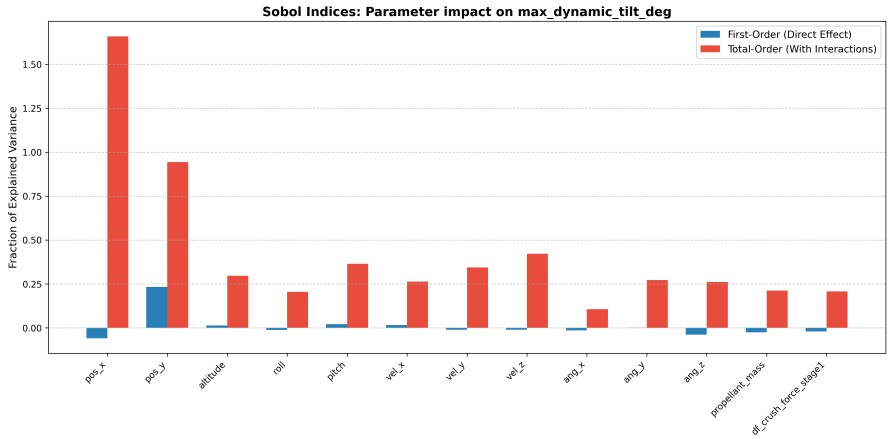


Figure 5.2: First-Order (S_1) and Total-Order (S_{T_i}) Sobol Indices illustrating the parameter impact on the maximum dynamic tilt angle.

The variance decomposition computed through these estimators reveals a system fundamentally dominated by environmental interactions rather than single-variable extremes, as illustrated in Figure 5.2.

The physical interpretation of the indices shows that the spatial coordinates act as the primary drivers of the system's variance. The longitudinal parameter `pos_x` exhibited a slightly negative First-Order index ($S_1 < 0$) alongside a staggering Total-Order index of $S_{T_i} = 1.658$. In the context of variance-based numerical estimators, both anomalies are known mathematical artifacts associated with finite sampling sizes. Since variance must mathematically be non-negative, a negative S_1 simply indicates that the true isolated contribution of the parameter is strictly zero, with the computed value dropping below zero purely due to the numerical noise of the Monte Carlo integration. Conversely, an S_{T_i} exceeding 1.0 occurs when evaluating highly discontinuous physical models, highlighting an explosive variance injection triggered by binary catastrophic events (such as the sudden transition from a stable stance to an unrecoverable rollover). Physically, this means that shifting the lander along the X-axis over a flat plain has absolutely no effect in isolation; however, if this displacement places the landing gear on a crater ridge while subjected to lateral velocity, it triggers a catastrophic failure. Similarly, `pos_y` emerged with a massive total influence ($S_{T_i} = 0.943$), indicating an intrinsic geometric sensitivity of the lander to lateral topographical gradients.

This sensitivity analysis formally demonstrates the importance of GNC algorithms designed to prioritize autonomous hazard avoidance to actively guarantee mission success and expand the safe-landing envelope.

5.3 Fixed-Topography Analysis

The Variance-Based Sensitivity Analysis discussed in Section 5.2 highlighted that topographic uncertainty (`pos_x` and `pos_y`) dominates the system's variance. Random planar spawning over a highly irregular lunar mesh inherently mixes two different sources of failure: dynamic instability (the vehicle's inability to arrest its kinetic energy) and geometric impossibility (spawning directly on a sharp crater edge or a boulder). To systematically eliminate this aleatoric uncertainty, a second analytical campaign was conducted. The objective of this phase is to evaluate the lander's stability by fixing the horizontal spawning coordinates to specific challenging spots on the lunar surface. The analysis was repeated across three pre-selected landing spots characterized by local slopes of 10°, 15°, and 20°. The selection of these specific inclinations is driven by standard GNC requirements, which typically constrain target landing zones to a maximum slope of 15°. By removing the random spatial variance and testing on known, controlled inclinations, this analysis effectively isolates the true dynamic failure.

5.3.1 Topographic Discretization and Target Spot Selection

To identify a specific landing coordinate, a dedicated Python framework was developed. The script process the 3D lunar mesh to locate regions with a specific topographical slope. The algorithm leverages the `trimesh` library to load the (`.stl`) terrain mesh, then the continuous 3D space is discretized into a 2D planar grid with a resolution of 0.2 m. To evaluate the exact ground elevation at each grid node, a ray-casting technique is employed. A dense array of simulated laser rays is projected vertically downward onto the mesh, and the precise Z-intersection coordinates are extracted to construct an elevation matrix (Heightmap).

```
import numpy as np
import trimesh

# Discretization of the 2D grid
resolution = 0.2
x_grid = np.arange(min_bounds[0], max_bounds[0], resolution)
y_grid = np.arange(min_bounds[1], max_bounds[1], resolution)
X, Y = np.meshgrid(x_grid, y_grid)

# Downward ray-casting setup
ray_origins = np.column_stack((X.flatten(), Y.flatten(), np.full(X
    .size, max_bounds[2] + 10.0)))
ray_directions = np.tile([0, 0, -1], (X.size, 1))

locations, index_ray, _ = mesh.ray.intersects_location(
    ray_origins=ray_origins,
    ray_directions=ray_directions
)
```

Listing 5.1: Ray-casting initialization and Heightmap extraction.

Once the discrete Heightmap Z is populated, the spatial gradient is computed to evaluate the local terrain slope. However, evaluating the slope of a single 0.2×0.2 m pixel is insufficient for stability analysis, as the lander's footprint spans several meters. To address this, a mathematical convolution is applied using `scipy.ndimage.uniform_filter` [14]. This operation calculates the *average* slope beneath the entire spatial footprint of the lander, preventing localized micro-anomalies (such as a small rock) from being misidentified as continuous slopes.

```
from scipy.ndimage import uniform_filter

# Calculation of the spatial gradient (dy, dx)
dy, dx = np.gradient(Z, resolution, resolution)
slope_rad = np.arctan(np.sqrt(dx**2 + dy**2))
slope_deg = np.degrees(slope_rad)

# Evaluation of the lander footprint average slope
```

```
slope_deg_safe = np.nan_to_num(slope_deg, nan=90.0)
footprint_pixels = int(lander_radius / resolution)
avg_slope_deg = uniform_filter(slope_deg_safe, size=
    footprint_pixels, mode='nearest')
```

Listing 5.2: Spatial gradient calculation and footprint convolution.

Finally, the algorithm applies a Boolean mask to isolate the target areas meeting the challenging slope criteria. To ensure diverse testing conditions and avoid selecting multiple points within the same crater, an Euclidean distance filter is applied, enforcing a minimum separation (15 m) between the identified spots.

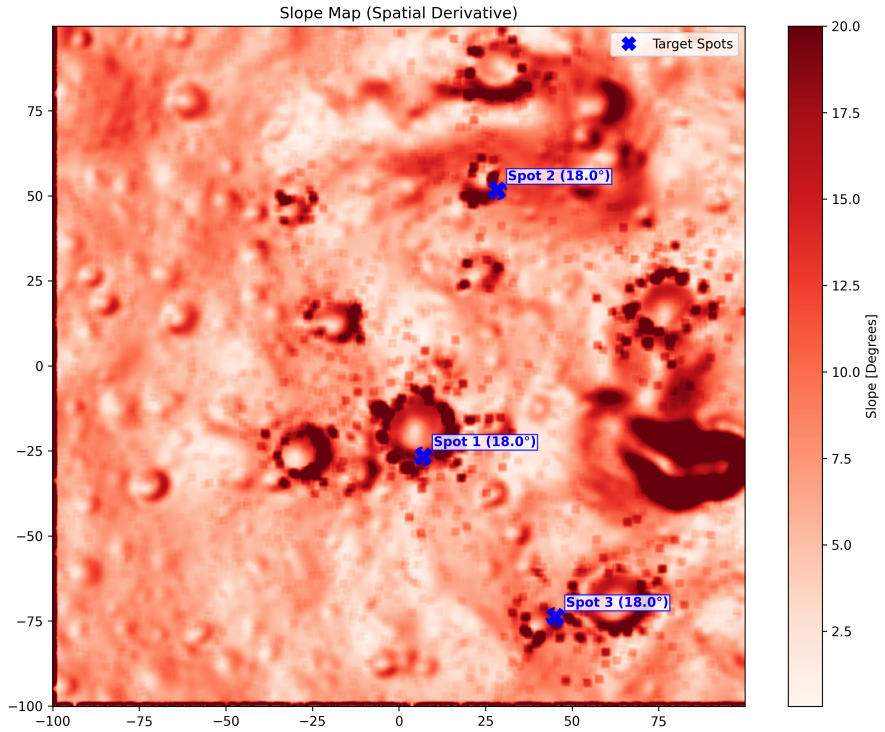


Figure 5.3: Slope map of the lunar regolith mesh (spatial derivative).

5.3.2 Nominal Landing Scenario: 10° Slope Analysis

Dimensionality Reduction (Morris Screening)

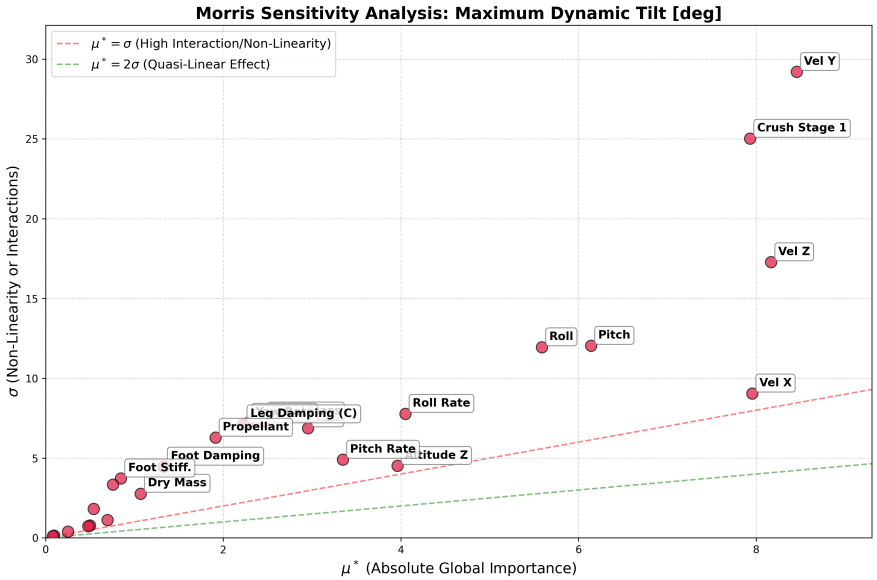


Figure 5.4: Morris Sensitivity Analysis (10° Slope): Scatter plot of Global Importance (μ^*) versus Non-Linearity/Interactions (σ) for the Maximum Dynamic Tilt.

The Morris dispersion plot for the 10° slope scenario (Figure 5.4) highlights a fundamental shift in the system's sensitivity hierarchy compared to the global random-spawning analysis.

The system is dominated by the initial linear velocities (`Vel Y`, `Vel Z`, `Vel X`) and the primary shock absorber yield limit (`Crush Stage 1`). Initial attitude angles (`Pitch`, `Roll`), altitude, and angular rates immediately follow in the ranking, exhibiting significant non-linear interactions ($\sigma > 11.0$). While the secondary mechanical properties of the landing gear (such as `Leg Damping`, `Foot Stiffness`, and `Strut Rebound`) demonstrate a negligible impact on the overall stability ($\mu^* < 2.5$). Consequently, to drastically reduce the dimensionality of the input space, only the top 13 highest-ranking variables were retained for the subsequent Variance-Based analysis, while the remaining lower-tier hardware parameters were fixed to their nominal design values.

Variance-Based Analysis (Sobol Method)

While the previous theoretical framework (Section 2.4.2) detailed the extraction of First-Order (S_i) and Total-Order (S_{T_i}) indices using $N(k+2)$ evaluations, this time also the pairwise interactions were analyzed by calculating the Second-Order index (S_{ij}). The S_{ij} metric isolates the fraction of the output variance caused by the coupled interaction of parameter i and parameter j. To compute this within the `SALib` framework, the Saltelli scheme generates an extended set of cross-matrices, doubling the required parameter variations. Consequently, the operational cost of the campaign expands to:

$$N_{runs} = N \times (2k + 2) \tag{5.5}$$

For this 10° slope evaluation, the dimensionality was reduced to $k = 13$ critical parameters thanks to the preceding Morris screening. A base sample size of $N = 128$ was selected to maintain the optimal space-filling properties of the Sobol sequence. Therefore, the orchestrated execution grid required:

$$N_{runs} = 128 \times (2(13) + 2) = 3584 \text{ independent runs} \tag{5.6}$$

The variance decomposition for the maximum dynamic tilt reveals a system heavily governed by coupled kinetic factors rather than isolated variables. Figure 5.5 illustrates the massive gap between the direct effects (S_1) and the total interaction effects (S_{T_i}).

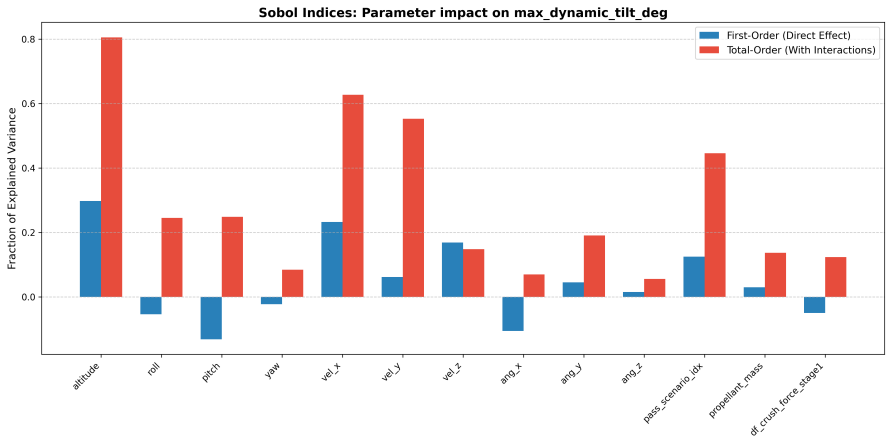


Figure 5.5: First-Order (S_1) and Total-Order (S_{T_i}) Sobol Indices illustrating the parameter impact on the maximum dynamic tilt angle on a 10° slope.

The physical interpretation highlights three main drivers:

- **Altitude Drop ($S_{T_i} \approx 0.81$ for `altitude`):** The initial altitude parameter directly represents the height at which the Main Engine Cut-Off (MECO) event is triggered. While its direct effect ($S_1 = 0.30$) is substantial, its massive total-order influence demonstrates that shutting down the thrusters too early acts as a severe energy amplifier. A higher MECO altitude prolongs the free-fall phase in lunar gravity, drastically increasing the vertical kinetic energy upon impact and synergistically exacerbating any existing lateral drift or asymmetric attitude errors before the footpads touch the regolith.

- **Lateral Velocities ($S_{T_i} \approx 0.63$ for `vel_x`):** Any residual horizontal velocity directed down the slope is highly critical. If the lander drifts laterally during a sloped touchdown, the downhill landing leg strikes the ground first. This single leg is forced to abruptly arrest both the vertical fall and the sideways drift, acting as a pivot point. This sudden deceleration at the base, while the upper mass of the vehicle continues moving by inertia, creates a severe overturning moment that trips the lander and causes a rollover.

- **Payload Mass ($S_{T_i} \approx 0.45$ for `pass_scenario_idx`):** The mass of the passenger payload, which is integrated on top of the lander's main body, exhibits a substantial total-order influence. Physically, a heavier payload significantly raises the vehicle's global Center of Gravity (CoG). On an inclined surface, a higher CoG drastically shrinks the static stability margin, making the lander top-heavy and exceptionally vulnerable to tipping over when subjected to lateral decelerations or asymmetric impacts.

- **Negative First-Order Artifacts:** Parameters like `pitch` and `roll` display small negative S_1 values. In numerical variance-based analysis, this is a standard computational artifact indicating that the parameter's independent influence is virtually zero; its entire contribution to the failure boundary exists exclusively through interactions.

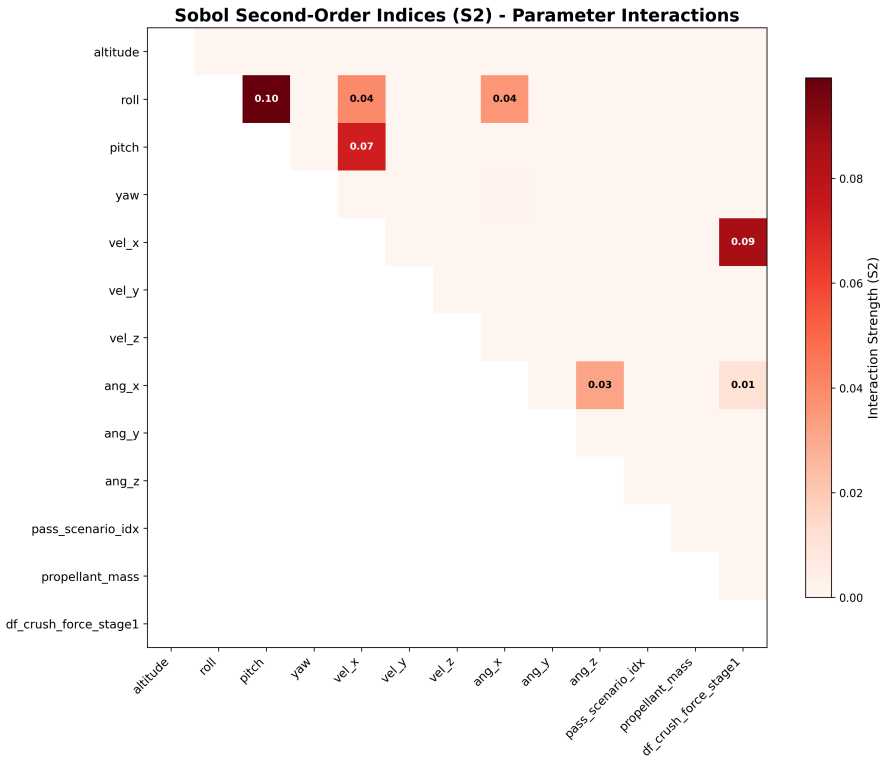


Figure 5.6: Sobol Second-Order Indices (S_2) Heatmap detailing the interaction strength between parameter pairs.

Figure 5.6 filters out numerical noise and highlights the most prominent pairwise interactions:

- **`roll` and `pitch` ($S_2 = 0.10$):** The strongest interaction occurs between the two attitude angles. A combined pitch and roll offset places the center of gravity dangerously close to the geometric boundary of a single downhill footpad, minimizing the energy required to tip the vehicle.
- **`vel_x` and `df_crush_force_stage1` ($S_2 = 0.09$):** This specific coupling proves that when the lander translates laterally down the 10° slope (`vel_x`), the survival of the vehicle is also dictated by the yield force of the primary shock absorber (`stage 1`). If the strut crushes too easily under the combined lateral-vertical load, the chassis bottoms out, acting as a pivot point for a dynamic rollover.

5.3.3 Operational Limit Scenario: 15° Slope Analysis

Dimensionality Reduction (Morris Screening)

Following the nominal 10° evaluation, the simulation framework was subjected to the 15° slope scenario.

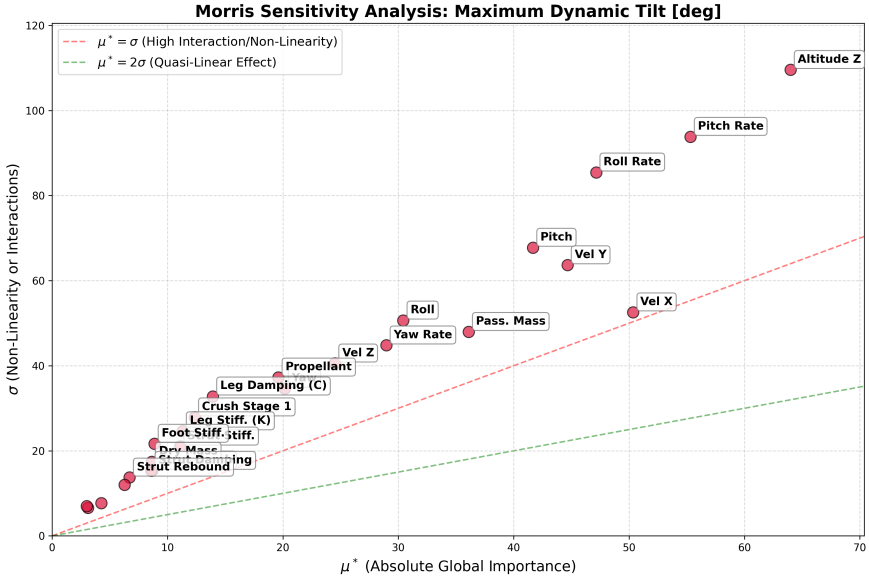


Figure 5.7: Morris Sensitivity Analysis (15° Slope): Scatter plot of Global Importance (μ^*) versus Non-Linearity/Interactions (σ) for the Maximum Dynamic Tilt.

The Morris dispersion plot for this boundary condition (Figure 5.7) illustrates a drastic escalation in both parameter importance (μ^*) and non-linear interactions (σ). Mathematically, this explosion in sensitivity values indicates that the vehicle is operating at the very edge of its stability limit, where minor kinematic variations easily trigger macroscopic, catastrophic rollover events. The system is overwhelmingly dominated by `Altitude Z` ($\mu^* \approx 64.0$) and initial angular velocities (`Pitch Rate`, `Roll Rate`), which exhibit massive interaction indices ($\sigma > 90.0$). However, despite the sharp increase in all absolute sensitivity values and the internal reshuffling of the top hierarchy, the categorical divide between critical variables and negligible hardware remains unchanged. Consequently, the exact same subset of 13 top-tier parameters was retained for the subsequent Variance-Based evaluation,

Variance-Based Analysis (Sobol Method)

Following the exact same computational architecture established for the 10° scenario, the variance decomposition for this operational boundary limit reveals a critical shift in the lander's failure mechanism. While the preceding Morris screening highlighted `Altitude Z` as having the highest local gradients due to the razor-thin stability margin, the Sobol method isolates the true physical drivers of the variance. As illustrated in Figure 5.8, the massive influence of pure altitude drop vanishes. The variance framework mathematically recognizes that altitude acts merely as a time-multiplier for other errors, attributing the actual cause of failure primarily to complex kinematic parameters that actively generate overturning moments upon ground contact.

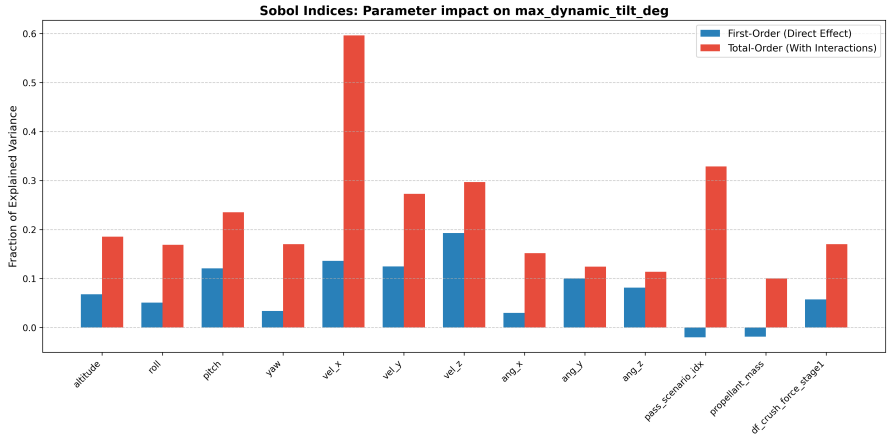


Figure 5.8: First-Order (S_1) and Total-Order (S_{T_i}) Sobol Indices illustrating the parameter impact on the maximum dynamic tilt angle on a 15° slope.

The physical interpretation of these indices highlights the absolute criticality of the GNC terminal targeting:

- **Lateral Velocity Dominance ($S_{T_i} \approx 0.60$ for `vel_x`):** At a 15° inclination, the static stability margin is so narrow that lateral translation becomes the most lethal factor. Any residual velocity moving down the slope entirely dictates the failure boundary through massive interactions with the terrain and the suspension system.
- **Vertical Impact Energy ($S_1 \approx 0.19$ for `vel_z`):** The initial vertical velocity exhibits the highest direct, First-Order effect. Because the slope is severe, a hard vertical touchdown forces asymmetrical compression on the uphill and downhill legs, directly driving the vehicle toward a tip-over regardless of other parameters.

- **Payload Mass and CoG Shift ($S_{T_i} \approx 0.33$ for `pass_scenario_idx`):** The inclusion of a top-mounted passenger payload exhibits a negative First-Order artifact but a substantial Total-Order influence. This confirms that while raising the global Center of Gravity (CoG) does not tip the lander on its own, it acts as a massive instability multiplier. An elevated CoG drastically reduces the energy required for lateral velocities or asymmetrical impacts to pivot the vehicle over its downhill footpads.

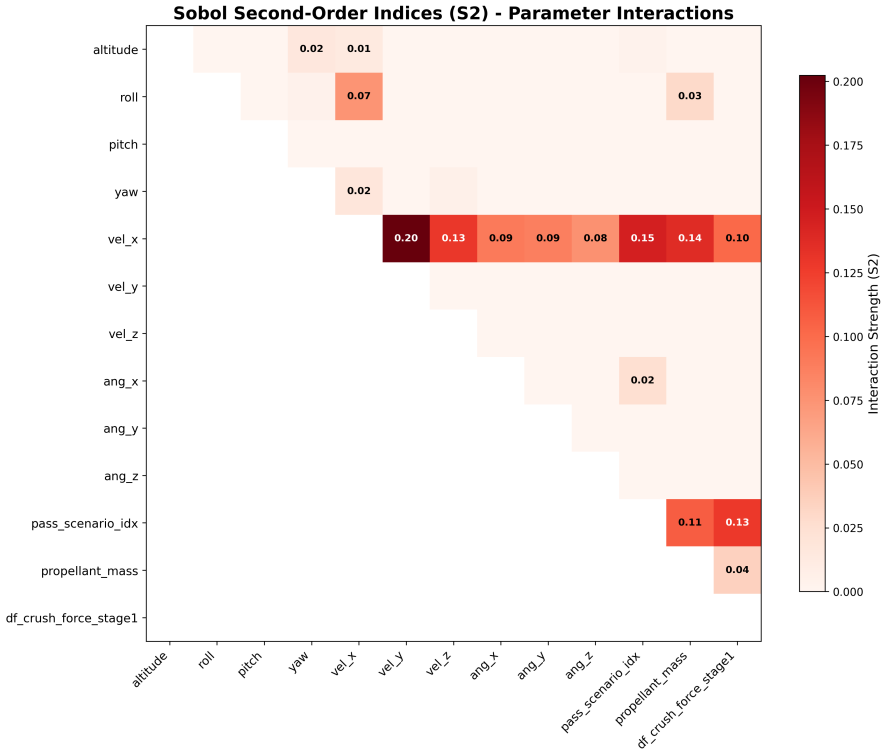


Figure 5.9: Sobol Second-Order Indices (S_2) Heatmap detailing the interaction strength between parameter pairs for the 15° slope.

The heatmap demonstrates a profound coupling driven almost entirely by the longitudinal sliding velocity:

- **Bi-Directional Lateral Drift (`vel_x` interacting with `vel_y`, $S_2 = 0.20$):** The absolute strongest interaction occurs when the lander experiences simultaneous downhill sliding (`vel_x`) and transverse cross-slope drift (`vel_y`). This creates a diagonal kinetic vector across the 15° incline. When the downhill footpad strikes the regolith, it catches at an oblique angle, generating a severe, multi-axis overturning moment that the symmetric suspension layout cannot effectively counteract.

- **Kinetic Translation and Inertial properties (`vel_x` interacting with `pass_scenario_idx` [0.15] and `propellant_mass` [0.14]):** Downhill sliding heavily couples with the vehicle's mass distribution. Both the top-mounted payload mass and the internal propellant mass dictate the lander's global Center of Gravity (CoG) and inertia tensor. When a heavily loaded lander translates down the slope, the elevated mass acts as a massive lever arm, drastically increasing the overturning moment the moment the footpads snag the terrain.

5.3.4 Off-Nominal Scenario: 20° Slope Analysis

Dimensionality Reduction (Morris Screening)

The final analytical phase evaluates the vehicle's dynamic response at a 20° inclination, representing an off-nominal touchdown scenario.

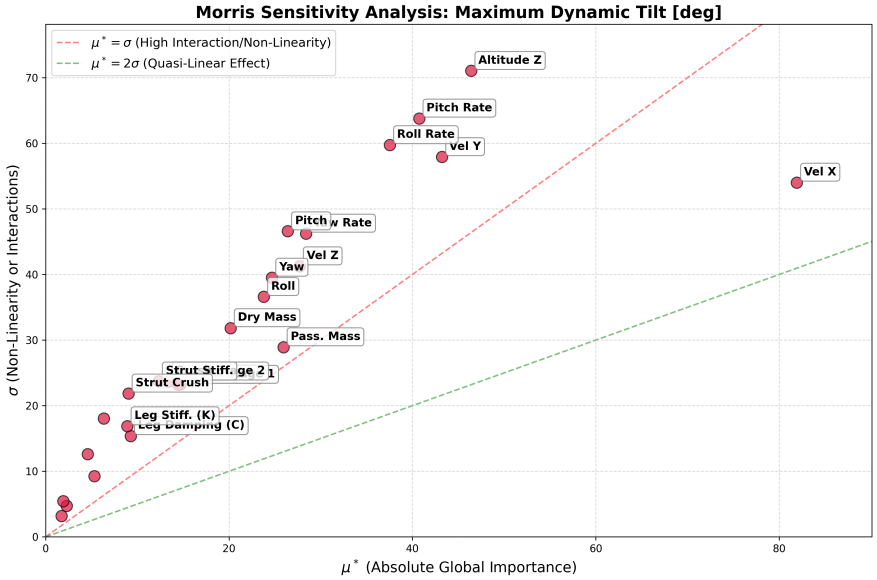


Figure 5.10: Morris Sensitivity Analysis (20° Slope): Scatter plot of Global Importance (μ^*) versus Non-Linearity/Interactions (σ) for the Maximum Dynamic Tilt.

Evaluating the parameter ranking at the extreme inclination reveals that longitudinal downhill translation completely overwhelms the system. `Vel X` severely detaches from all other variables, emerging as the absolute failure trigger ($\mu^* \approx 81.9$). Following this primary driver, `Altitude Z`, `Vel Y`, and initial angular rates form a highly coupled second tier of criticality ($\mu^* \approx 37.5 - 46.4$), with altitude exhibiting

massive non-linearity ($\sigma \approx 71.1$). While structural and mechanical variables, such as the shock absorber yield limits (`Crush Stage 1`, `Crush Stage 2`) and structural stiffness, remain entirely clustered at the bottom of the distribution ($\mu^* < 15.0$). Despite the extreme severity of this slope, the exact same subset of 13 top-tier parameters was retained for the subsequent Variance-Based evaluation.

Variance-Based Analysis (Sobol Method)

The resulting variance decomposition, illustrated in Figure 5.11, highlights a distinct evolutionary shift in the system's failure mechanics compared to shallower inclinations.

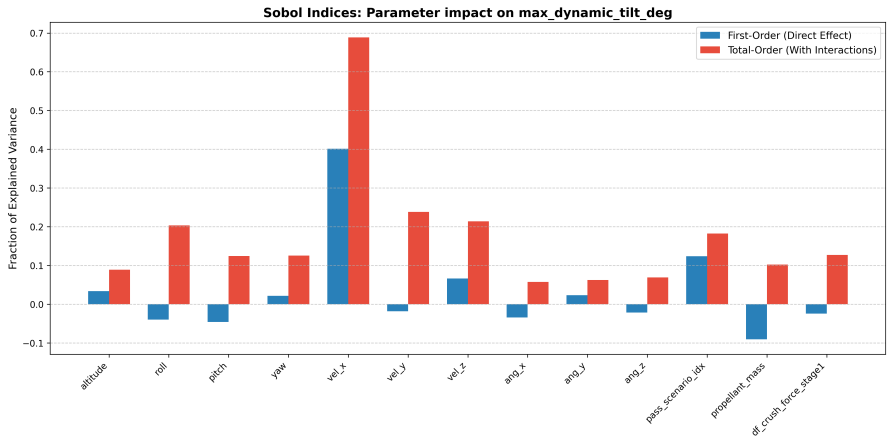


Figure 5.11: First-Order (S_1) and Total-Order (S_{T_i}) Sobol Indices illustrating the parameter impact on the maximum dynamic tilt angle on a 20° slope.

The physical interpretation of these indices reveals an environment where static stability margins are nearly non-existent:

- **Absolute Longitudinal Dominance ($S_1 \approx 0.40$, $S_{T_i} \approx 0.69$ for `vel_x`):** Unlike the 15° slope, where vertical and lateral kinetics shared the variance, the 20° slope is completely overwhelmed by longitudinal downhill translation. The massive First-Order index proves that any residual velocity directed down the slope directly causes failure, independent of all other parameters.
- **Attitude as Pure Multipliers:** Initial attitude angles (`pitch`, `roll`) and angular rates display negative S_1 numerical artifacts coupled with substantial Total-Order effects ($S_{T_i} \approx 0.12 - 0.20$). This confirms their role as pure interaction multipliers; an asymmetric posture does not act alone but severely exacerbates existing kinetic translations.

- **Diminished Altitude Influence:** Interestingly, the impact of initial altitude (`altitude`) drops significantly ($S_{T_i} \approx 0.09$), at a 20° incline, the slope is so severe that failure occurs almost immediately upon contact if lateral velocity is present, regardless of how long the vehicle was in free-fall.

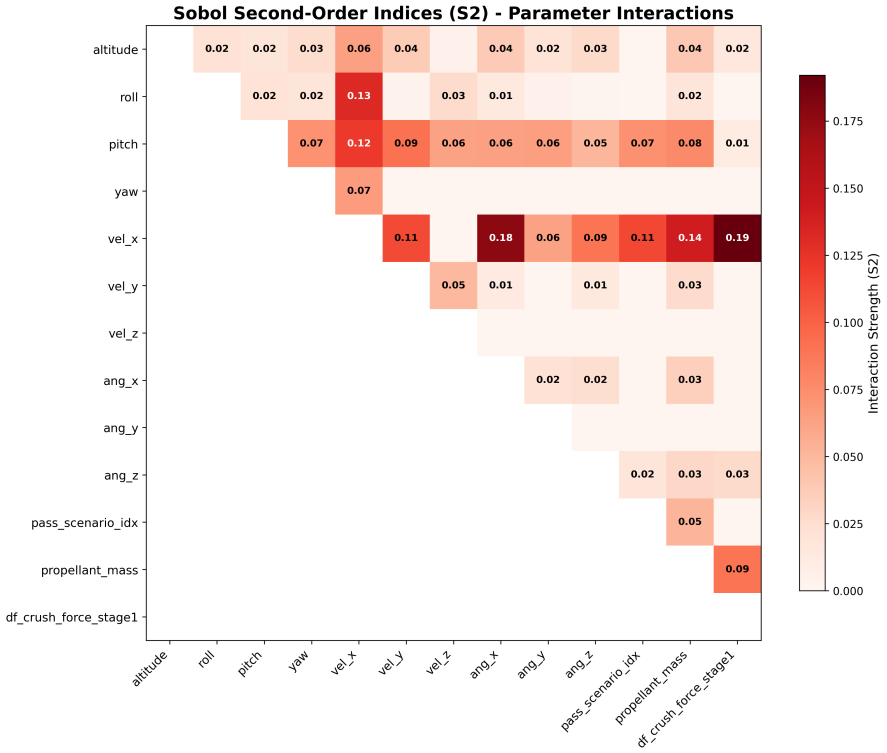


Figure 5.12: Sobol Second-Order Indices (S_2) Heatmap detailing the interaction strength between parameter pairs for the 20° slope.

The interaction matrix exposes a violent mechanical coupling upon touchdown that is completely centralized around the longitudinal downhill velocity (`vel_x`). As clearly visible in the heatmap, the absolute highest second-order indices are exclusively tied to `vel_x`, demonstrating that residual sliding momentum interacts aggressively across all other physical domains of the lander.

5.3.5 Synthesis of the Sensitivity Campaign

Fundamental Drivers of Dynamic Instability

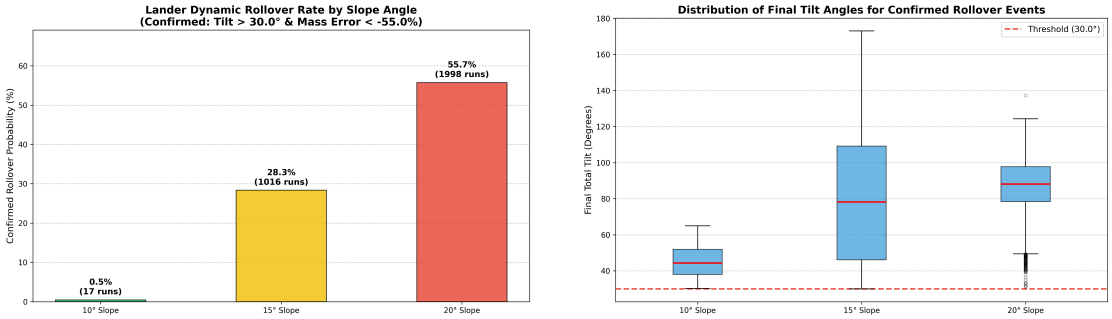
The multi-stage sensitivity analysis across the 10°, 15°, and 20° slope scenarios definitively isolates the fundamental physical drivers of lunar lander instability.

As the topographical inclination increases, the touchdown dynamics transition from a vertical-energy-dominated regime into a highly non-linear, kinematically coupled environment where static stability margins virtually vanish. The continuous evolution of the Morris and Sobol indices yields three critical insights into the system's dynamic response:

- **Kinematic Supremacy over Mechanical Tuning:** Across all tested boundary conditions, initial translational velocities (`vel_x`,`vel_y`) and initial attitude completely eclipse the structural parameters of the landing gear. This outcome highlights the critical importance of GNC systems in nullifying lateral drift and angular rates prior to the initiation of the terminal descent phase in order to achieve a successful soft landing.
- **The Lethality of Downhill Sliding:** As the slope angle increases, the downhill sliding velocity (`vel_x`) becomes the absolute primary cause of failure. By the extreme 20° scenario, it completely dominates the system's variance. If the lander has any residual velocity pushing it down the slope just before touchdown, the lowest landing leg strikes the terrain first and stops abruptly. Because the heavy upper chassis continues moving forward by inertia, this stuck leg acts as a rigid pivot. The vehicle effectively trips over its own footpad, instantly transforming the sliding momentum into a catastrophic rollover.
- **Altitude and CoG as Instability Multipliers:** While purely kinematic parameters provide the overturning force, the initial altitude and top-mounted payload mass (`pass_scenario_idx`) act as massive energy multipliers. The altitude parameter physically represents the Main Engine Cut-Off (MECO) height; triggering the shutdown too high prolongs the free-fall phase, drastically increasing vertical kinetic energy and amplifying minor rotational errors before contact. Simultaneously, a heavier payload raises the global Center of Gravity (CoG), drastically shrinking the static stability base and weaponizing even minor asymmetric impacts.

Statistical Rollover Analysis

To quantify the macroscopic effect of topographical inclination on the lander's survivability, a dual-threshold criterion for mechanically confirmed rollover was evaluated across the total 10,752 simulated touchdown events.A single failure is strictly defined by the coupling of a kinematic threshold (final total tilt $> 30°$) and a dynamic threshold (perceived mass error from the footpad $< -55\%$), which filters out false positives and confirms physical instability. The discrete failure occurrence rates highlight a severe, non-linear degradation of the vehicle's stability margin:



(a) Rollover Probabilities **(b)** Tilt Angle Distribution

Figure 5.13: Analysis of dynamic rollover events across simulated topographical inclinations. (a) Probability of catastrophic failure. (b) Distribution of final tilt angles exclusively for the failed runs.

- **Nominal Safety at** 10° **(0.5% Failure Rate):** With only 17 rollovers out of 3584 runs, this inclination is firmly within the vehicle's structural and kinematic capabilities. Instability at this slope is a statistical outlier, requiring an improbable, worst-case alignment of maximum lateral velocity, severe asymmetric attitude, and primary shock absorber bottom-out.
- **The Operational Boundary at** 15° **(28.3% Failure Rate):** The exponential jump to almost one-third of the simulations failing mathematically confirms 15° as the absolute limit of the vehicle's operational envelope. At this inclination, the mechanical stability margin is so narrow that the system transitions into a highly coupled state, relying almost entirely on the GNC system to deliver near-perfect terminal kinematics.
- **Catastrophic Degradation at** 20° **(55.7% Failure Rate):** At this extreme off-nominal incline, survival becomes the statistical exception rather than the rule. The fact that over half of all simulated landings result in an unrecoverable rollover proves that the static footprint is fundamentally incapable of counteracting the gravitational bias and longitudinal sliding momentum.

5.4 Conclusion

This thesis has successfully culminated in the development of a fully parametric and modular Model-in-the-Loop (MIL) lunar landing simulator. The primary strength of this computational framework lies in its highly adaptable architecture. Every physical, kinematic, and structural parameter is governed by centralized configuration files, allowing for rapid modifications without altering the underlying codebase. This level of configurability makes the simulator an exceptionally practical

tool during the iterative phases of spacecraft design, enabling engineers to seamlessly evaluate changing hardware specifications, varying payload masses, or updated Guidance, Navigation, and Control parameters. The established framework provides a reliable, scalable foundation that can be readily deployed in the future to conduct highly advanced, large-scale stochastic analyses.

Appendix A

ROS 2 `rqt_graph`

This appendix contains the complete node-topic topology of the simulation environment, generated during runtime via the `rqt_graph` introspection tool.

Due to the high number of active nodes, publishers, and subscribers involved in the simulation dynamics, the complete architecture is reported in Figure A.1 on the following page.

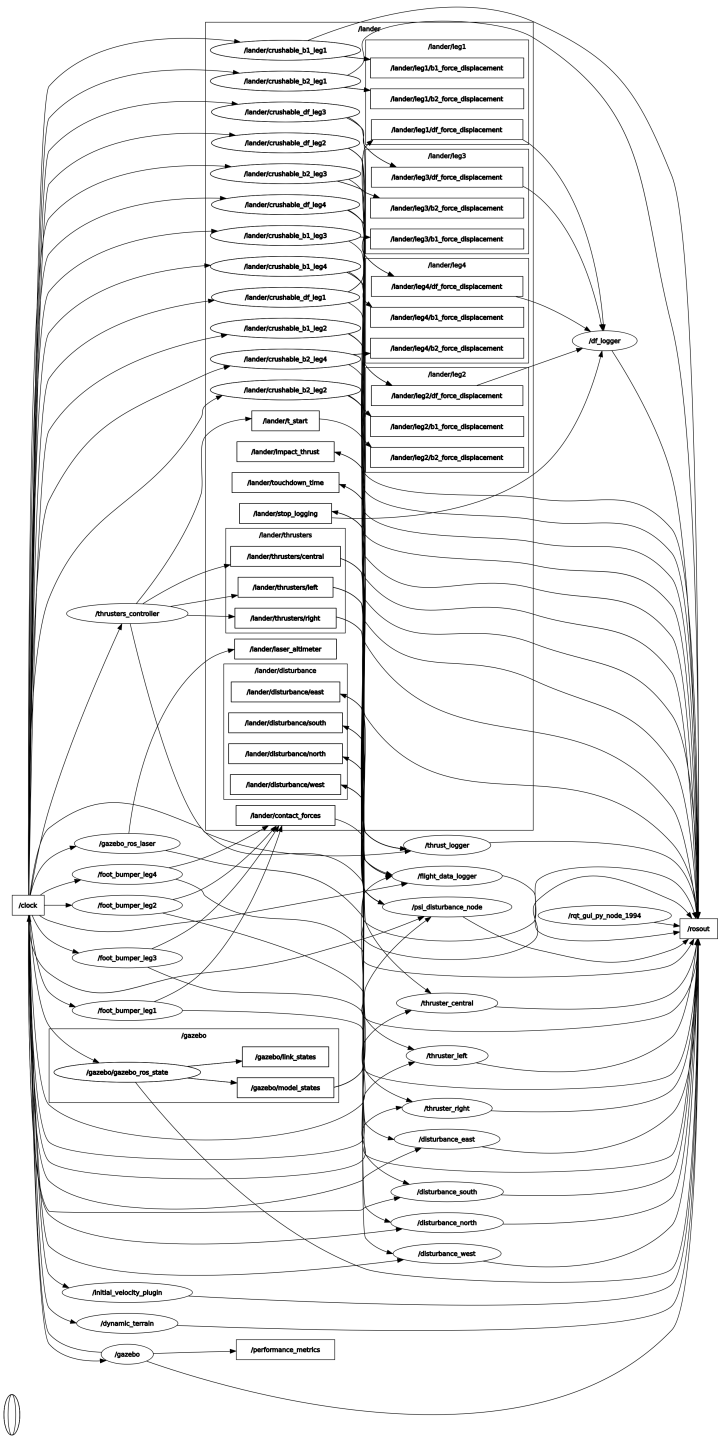


Figure A.1: Full ROS 2 computational graph of the lunar lander simulation architecture generated via `rqt_graph`.

Bibliography

[1] Wikipedia contributors. *V-Model — Wikipedia, The Free Encyclopedia*. 2026. URL: https://it.wikipedia.org/wiki/V-Model#/media/File:V-model.JPG (cit. on p. 4).

[2] Open Source Robotics Foundation. *ROS 2 Documentation*. 2026. URL: https://docs.ros.org/ (cit. on p. 10).

[3] Open Source Robotics Foundation. *Gazebo Simulator Documentation*. 2026. URL: https://gazebosim.org/ (cit. on p. 10).

[4] Dorleta Garcia, Inmaculada Arostegui, and Raúl Prellezo. «Robust combination of the Morris and Sobol methods in complex multidimensional models». In: *Environmental Modelling & Software* 122 (2019), p. 104517 (cit. on p. 10).

[5] Max D Morris. «Factorial sampling plans for preliminary computational experiments». In: *Technometrics* 33.2 (1991), pp. 161–174 (cit. on p. 11).

[6] Ilya M Sobol. «Sensitivity analysis for nonlinear mathematical models». In: *Mathematical modelling and computational experiments* 1.4 (1993), pp. 407–414 (cit. on p. 11).

[7] Andrea Saltelli, Paola Annoni, Ivano Azzini, Francesca Campolongo, Marco Ratto, and Stefano Tarantola. «Variance based sensitivity analysis of model output. Design and estimator for the total sensitivity index». In: *Computer physics communications* 181.2 (2010), pp. 259–270 (cit. on p. 12).

[8] Leonard Roberts. «The interaction of a rocket exhaust with the lunar surface». In: *The Fluid Dynamic Aspects of Space Flight (Proceedings of the AGARD-NATO Specialists Meeting)*. 1966, pp. 269–290 (cit. on p. 20).

[9] Russell Smith et al. *Open Dynamics Engine (ODE) User Guide*. 2006. URL: https://ode.org/ode-latest-userguide.html (cit. on p. 27).

[10] NumPy Developers. *NumPy: The fundamental package for scientific computing with Python*. 2026. URL: https://numpy.org/ (cit. on p. 36).

[11] Michael Dawson-Haggerty et al. *Trimesh: A Python library for loading and using triangular meshes*. 2026. URL: https://trimesh.org/ (cit. on p. 39).

[12] Matplotlib Development Team. *Matplotlib: Visualization with Python*. 2026. URL: `https://matplotlib.org/` (cit. on p. 52).

[13] Jon Herman, Will Usher, et al. *SALib: Sensitivity Analysis Library in Python*. 2026. URL: `https://salib.readthedocs.io/` (cit. on p. 72).

[14] SciPy Developers. *SciPy: Fundamental algorithms for scientific computing in Python*. 2026. URL: `https://scipy.org/` (cit. on p. 76).