AMITY UNIVERSITY

UTTAR PRADESH

Dissertation Report on

Satellite Docking Mechanism Control and Velocity Regulation

Submitted to

Amity Institute of Space Science and Technology

In partial fulfilment of the requirements for the award of the degree of
B. Tech (Aerospace) + M. Tech (Avionics)
Integrated Degree

Submitted to:
Dr. Shivani Verma

Submitted by:
Rhythmica A. M.
A047115920001
B. Tech (AE) +
M. Tech (AV)
2020-2025

Amity Institute of Space Science and Technology,
Amity University, Noida, Uttar Pradesh

Declaration

I, Rhythmica A. M., student of Integrated Bachelor of Technology (Aerospace Engineering) - Master of Technology (Avionics), hereby declare that the project report entitled “Satellite Docking Mechanism Control and Velocity Regulation” which is submitted by me to Amity Institute Of Space Science and Technology, Amity University, Noida, Uttar Pradesh in partial fulfilment of requirement for the award of the degree of Integrated Bachelor of Technology (Aerospace Engineering) - Master of Technology (Avionics), has not been previously formed the basis for the award of any degree, diploma or other similar title or recognition. The Author attests that permission has been obtained for the use of any copyrighted material appearing in the report, other than brief excerpts requiring only proper acknowledgment in scholarly writing, and that all such use is acknowledged.

Rhythmica A. M.
A047115920001
2020-25
BTech (AE) + MTech (AV)

Place: Noida, Uttar Pradesh

Declaration by Guide

Based on the declaration by Rhythmica A. M. (A047115920001), student of Integrated Bachelor of Technology (Aerospace Engineering) - Master of Technology (Avionics), I hereby certify that the project report entitled "Satellite Docking Mechanism Control and Velocity Regulation", which is submitted to the Amity Institute Of Space Science and Technology, Amity University, Noida, Uttar Pradesh in partial fulfilment of requirement for the award of the degree of Integrated Bachelor of Technology (Aerospace Engineering) - Master of Technology (Avionics), is an original contribution with existing knowledge and faithful record of work carried out under my guidance and supervision. To the best of my knowledge this work has not been submitted in part or full for any Degree or Diploma to this University or elsewhere.

Dr. Shivani Verma
Asst. Professor
Amity Institute of Space Science and Technology
Amity University, Uttar Pradesh

Acknowledgement

A project cannot be completed without the immense help and motivation given by the faculties and the mentors. I would like to thank Prof M. S. Prasad, Professor and Head of Department of Amity Institute of Space Science and Technology and Director, Amity University for giving me the opportunity to undertake this project. A vote of special thanks goes to Professor Dr. Shivani Verma, AISST department, who has been a constant guiding force throughout the course of this project, and whose immense support and cooperation has made the completion of this project possible. I thank Mr. Deepak Gaur (SpaceTS) for his technical guidance, patience, and kindness in working with me for this project and for the motivation he has given me throughout. I also sincerely thank the entire team at SpaceTS for their technical support, motivation, and invaluable help throughout the completion of this thesis. Last but not the least, I would like to thank my family, friends, and other loved ones, who supported me through my shakers and shudders, both by keeping me harmonious and by helping me putting pieces together. I will be forever in debt for your support.

Rhythmica A. M.
Enrollment No.: A047115920001
BTech (AE) + MTech (AV)
Amity Institute of Space Science and Technology
Amity University, Noida, Uttar Pradesh

Abstract

This work presents a comprehensive comparative study of three advanced control strategies for autonomous satellite docking: Model Predictive Control (MPC), Nonlinear Model Predictive Control (NMPC), and Sliding Mode Control (SMC). The primary objective is to evaluate each method's performance in achieving precise, fuel-efficient, and reliable spacecraft rendezvous and docking maneuvers under realistic conditions. These control algorithms were developed, implemented, and rigorously tested through simulation using MATLAB to assess their effectiveness in guiding a chaser satellite to a target spacecraft in a controlled and stable manner.

Simulation results demonstrate that NMPC offers superior performance in terms of docking precision and fuel optimization. By explicitly accounting for system nonlinearities and constraints, NMPC is able to align the chaser with the target with minimal position and orientation error while also minimizing thruster usage. This makes NMPC particularly well-suited for high-stakes missions where accuracy is critical, such as manned spacecraft operations or docking with a space station, where safety and precision are paramount.

MPC, though less precise than NMPC, still performs adequately under moderate mission requirements. It strikes a balance between computational demand and control accuracy, making it suitable for missions where minor deviations in docking alignment are acceptable and where onboard computational resources may be limited.

Conversely, the SMC algorithm, despite its fast computational performance and robustness to model uncertainties, exhibits significant drawbacks. It introduces high-frequency control activity, which leads to thruster chatter, excessive vibration, and increased fuel consumption. These effects undermine its suitability for delicate docking procedures unless further modifications or compensatory mechanisms are introduced to mitigate these issues.

As a direction for future work, this research proposes the development of a hybrid control architecture that leverages the precision of NMPC with the robustness and rapid response characteristics of SMC. Such an approach could provide a more balanced and adaptable solution to complex docking scenarios. Additionally, further efforts will focus on validating these control strategies in real-time environments through hardware-in-the-loop (HIL) testing and eventual integration with actual spacecraft hardware. This progression will bridge the gap between theoretical simulations and practical implementation, contributing to the advancement of autonomous docking capabilities in future space missions.

Table of Contents

List of Figures and Tables

Chapter 1
Introduction

1.1 Background

Satellite docking amounts to mating of two satellites in space precisely. The satellite trying to Dock is referred as Chaser and other as Target. Satellite docking is an important space mission manoeuvre that supports spacecraft servicing, modular station construction, and in-orbit fuelling. Precision and safe docking are extremely difficult to achieve with nonlinear orbital dynamics, external perturbations (e.g., solar radiation pressure, atmospheric drag residual), and system uncertainties (e.g., thruster misalignment, sensor noise). Satellite docking is a critical maneuver for in-orbit servicing, space station assembly, and crewed missions, requiring millimeter-level precision to avoid collisions or misalignment. As illustrated in Figure 1.1, the process involves a chaser spacecraft aligning with a target vehicle under complex orbital dynamics and constraints. Traditional control techniques are not capable of achieving a good balance between precision, energy efficiency, and computational tractability, especially during the terminal approach phase (<100 meters), where millimetre-level precision is required to prevent collisions or misalignment.

Figure 1.1 Image showing the docking of two satellites

New emergent control theory developments have introduced sophisticated techniques such as Model Predictive Control (MPC), Nonlinear MPC (NMPC), and Sliding Mode Control (SMC) to address these issues. MPC and NMPC employ predictive optimization to counteract constraints and nonlinearities, and SMC offers disturbance robustness. Their comparative performance in real-world docking applications—energy consumption, computational complexity, and terminal precision—is yet to be sufficiently investigated.

1.2 Problem Statement

The main problem of satellite docking is to control the relative position, velocity, and attitude of the target with respect to the chaser vehicle under tight constraints:

a. Velocity limits: Too high an approach speed can damage docking mechanisms. We must control the velocity as it ensures a safe and precise docking between satellites or spacecraft. If the velocity is too high, it can cause a collision, damaging both vehicles. If it's too low or misaligned, the docking may fail or become unstable. Controlled velocity allows for smooth alignment, synchronization of movement, and gentle contact, which is critical for successful docking without causing structural harm or mission failure.

b. Thruster saturation: Limited control authority requires optimal distribution of thrust.
c. Measurement noise: Sensor inaccuracies must be compensated without overreacting.

Existing techniques like PID control or bang-bang policies are either too imprecise (PID) or too energy-intensive (bang-bang). Thus, the sophisticated control algorithms - MPC, NMPC, and SMC - need to be considered and compared to determine the optimal performance vs. realism balance.

1.3 Objectives

The primary objectives of this research are:

a. To develop and implement an optimal control strategy for autonomous satellite docking.
b. To regulate the approach velocity to minimize impact forces during docking.
c. To optimize the docking trajectory for fuel efficiency while considering external perturbations.
d. To validate the control strategy through MATLAB/Simulink simulations, incorporating real-world constraints.

1.4 Project Scope

The project is aimed at designing and simulating a docking control algorithm for a chaser satellite to dock with a target satellite in a Low Earth Orbit (LEO) mission. The research is confined to:

a. Simulation-Based Analysis: The control algorithms are simulated and verified on MATLAB/Simulink, without hardware verification.
b. Chaser-Target Model: The target satellite stays in a steady orbit, with the chaser satellite making its own manoeuvres by using thrusters.
c. Control Constraints: The docking operation follows realistic thruster limits, force limits, and approach velocity constraints.

This study adds to the development of autonomous docking technology, enhancing safety, efficiency, and flexibility for forthcoming space missions.

1.6 Summary

This section establishes satellite docking as a critical space operation requiring millimeter-level precision between chaser and target spacecraft, complicated by orbital dynamics and system uncertainties. It identifies key challenges in velocity control, thruster saturation, and sensor noise that existing PID/bang-bang methods fail to address optimally. The research aims to develop and compare advanced control strategies (MPC/NMPC/SMC) through MATLAB simulations, focusing on autonomous docking in LEO with realistic constraints. The study's scope is limited to simulation-based analysis, laying groundwork for future hardware implementation

This report is structured as follows: Chapter 1 contains the introduction, providing background information, the problem statement, objectives, and the scope of the project. Chapter 2 presents a comprehensive literature review, covering spacecraft docking, control methods, and force regulation mechanisms. Chapter 3 delves into the system design, explaining key controllers, Sliding Mode Controller, Model Predictive Controller and Nonlinear Model Predictive Controller. Finally, Chapter 4 presents the results of the project, discussing the performance of Sliding Mode Controller, Model Predictive Controller and Nonlinear Model Predictive Controller. Chapter 5 focuses on the comparative analysis of the three controllers followed by the project extension. The report concludes with a summary of the project's achievements and potential impact on docking operations.

Chapter 2
Literature Review

The chapter continues by listing some of the current work related to the spacecraft docking, control methods, and force regulation mechanisms.

2.1 Fundamentals of Spacecraft Docking

Autonomous docking of satellites is a vital task in contemporary space missions, necessitating accurate velocity control, force regulation, and attitude alignment. Researchers have examined numerous control strategies in previous studies to improve docking reliability and efficiency.

The relative motion dynamics modelling between a target satellite and a chaser is the key to docking control. Fehse's work [1] offers a complete overview of satellite rendezvous and docking (RVD) methods, such as guidance laws and control approaches utilized in some space missions. NASA's study [2] presents a nonlinear docking control strategy where robust control methods are highlighted for compensating for system uncertainties and external disturbances.

2.2 Dynamics Modeling and Simulation

Accurate modeling of relative motion between the chaser and target spacecraft is essential for docking control. The Launching, Rendezvous, and Docking Simulation Tool (LAREDO) [3] offers a high-fidelity simulation environment for assessing docking situations, including thruster arrangements and trajectory planning techniques. Such tools are crucial for the verification of the performance of control algorithms prior to actual implementation in real space missions.

2.3 Attitude and Velocity Control Strategies

Attitude control and determination are essential to guarantee successful docking. The research by Ciarcià et al. [4] examines nonlinear attitude control techniques for microsatellites, highlighting the use of reaction wheels and control moment gyroscopes (CMGs) for precise attitude manoeuvres. In addition, the use of hybrid control techniques, including a mixture of Model Predictive Control (MPC) and PID control, has been researched in docking missions [5], showing better tracking precision and perturbation robustness.

For velocity control, Bang-Bang control has been widely applied in space missions because of its simplicity and effectiveness [6]. Although efficient, it does not provide smooth transitions, which result in unnecessary fuel use. For this reason, hybrid control methods combining MPC for trajectory planning and PID for last approach tuning have been suggested [7], providing accurate docking with low fuel usage.

2.4 Force Regulation and Safety Mechanisms

Aside from velocity and attitude control, docking force control is essential to avoid excessive impact forces that might harm spacecraft docking interfaces. Impedance control has been popularly used for force control in robotic docking missions [8]. Dynamically adjusting the stiffness and damping of the docking mechanism, impedance control facilitates smooth and controlled docking port engagement.

Further, force sensing and feedback systems have also been investigated for improved docking safety. NASA [9] has reported on the use of real-time force sensing in the docking operation using contact sensors and load cells to control the forces applied during docking. These strategies are especially useful for preventing misalignments and facilitating soft docking engagements.

2.5 Research Gaps

Despite these advancements, the current literature exhibits several important gaps. Most existing studies evaluate control algorithms in isolation, lacking comprehensive comparisons under standardized mission constraints such as thruster limitations and sensor noise profiles. Additionally, most control strategies have only been validated in simulation environments, with limited verification through hardware-in-the-loop testing or actual orbital demonstrations. The potential of adaptive control methods, particularly those incorporating machine learning for real-time disturbance compensation, remains largely unexplored in the context of spacecraft docking.

This work directly addresses these gaps through systematic comparison of MPC, NMPC and SMC controllers under identical simulation conditions, providing quantitative performance metrics across precision, fuel efficiency and computational load. The proposed hybrid NMPC-SMC architecture combines the optimality of predictive control with the disturbance rejection capabilities of sliding mode approaches. Furthermore, the inclusion of PWM-based thruster modulation in our simulations bridges an important gap between theoretical control algorithms and practical implementation challenges. These contributions advance the field toward more reliable and adaptable docking systems capable of meeting the demands of future space missions.

2.6 Summary

The literature reviewed emphasizes the importance of sophisticated control methods in spacecraft docking. Although Bang-Bang control provides a simple solution for velocity control, MPC, NMPC and SMC methods offer better accuracy and efficiency.

In addition, force control methods, including impedance control and real-time force sensing, improve docking safety by reducing impact forces. Future studies must incorporate machine learning-based adaptive control methods to further enhance docking performance.

Chapter 3
System Design

Docking a target satellite with a chaser satellite calls for tight control of relative position and velocity. Various control methods have been examined, from classical to sophisticated model-based designs. This section outlines the mathematical formulations governing the docking process, including system dynamics, control methodologies, and optimization techniques.

System Parameters
a. Velocity from x to y in xy plane
b. Distance of start
c. Low earth orbit in Circular orbit
d. Steady condition of target during docking manoeuvre

3. 1 System Dynamics

The relative movement of the chaser and the target satellites at low Earth orbit (LEO) is characterized using the Clohessy-Wiltshire (CW) equations, in which the dynamics of a satellite in a circular orbit around Earth are described. Linearized equations for motion are presented as follows:

$$\dot{x} = v_x, \dot{y} = v_y, \dot{z} = v_z \quad \text{(i)}$$

$$\dot{v_x} = 3n^2x + 2nv_y + a_x \quad \text{(ii)}$$

$$\dot{v_y} = -2nv_x + a_y \quad \text{(iii)}$$

$$\dot{v_z} = -n^2z + a_z \quad \text{(iv)}$$

where:
x, y, z represents the position in the local-vertical local-horizontal (LVLH) frame,
v_x, v_y, v_z are the velocity components,
a_x, a_y, a_z are control accelerations,
$n = \sqrt{\frac{\mu}{r^3}}$ is the mean motion,
μ is the Earth's gravitational parameter,
r is the orbital radius.

3. 2 Control Methodologies

3. 2. 1 Sliding Mode Control (SMC)

The SMC approach introduces a sliding surface S:

$$S = c_1e + c_2 \quad \text{(v)}$$

Where c_1, c_2 are positive constants. The control law is given by:

$$F = -K_{eq}S - K_{sw}sign(S \quad \text{(vi)}$$

where K_{eq} is the equivalent control and K_{sw} is the switching gain. This approach improves robustness against disturbances but introduces chattering effects.

3. 2. 2 Model Predictive Control (MPC)

MPC optimizes control inputs over a prediction horizon N_p by solving a quadratic programming problem:

$$\min \sum_{k=0}^{Np} (x_k^T Q x_k + \; u_k^T R u_k) \tag{vii}$$

subject to system dynamics, input constraints $u_{min} \leq u \leq u_{max}$, and state constraints $x_{min} \leq x \leq x_{max}$. The MPC approach ensures optimal trajectory planning while maintaining constraints.

3. 2. 3 Nonlinear Model Predictive Control (NMPC)

NMPC extends MPC by handling nonlinear system dynamics. The optimization problem remains:

$$\min \sum_{k=0}^{Np} (x_k^T Q x_k + \; u_k^T R u_k) \tag{viii}$$

but with nonlinear state equations:

$$\dot{x} = f(x, u) \tag{ix}$$

where f(x, u) includes nonlinear perturbations. The NMPC formulation improves performance over MPC but increases computational complexity.

3. 3 Assumptions

a. Idealized Actuators: No thruster delay or misalignment were considered.
b. Deterministic Disturbances: No external disturbances, J2 perturbations or third-body effects were considered.

This section detailed the mathematical models used for satellite docking control. The results indicate that MPC and NMPC are optimal choices for docking applications, balancing precision, efficiency, and robustness. Further refinements will focus on real-time implementation and adaptation strategies.

Chapter 4
Results and Discussion

The efficiency of the applied control techniques — Sliding Mode Control (SMC), Model Predictive Control (MPC), and Nonlinear Model Predictive Control (NMPC) — was compared based on important performance parameters like docking time, energy consumption, velocity control, and impact force minimization. The results from the MATLAB simulations reveal an overall comparison of the efficiency of each controller in exhibiting a smooth and fuel-efficient satellite docking manoeuvre.

4. 1 Sliding Mode Controller (SMC)

The approach trajectory of the chaser spacecraft using the Sliding Mode Controller (SMC), is plotted in the local target-centric reference frame. The trajectory as shown in Figure 4.1 reveals high-frequency oscillations along the path, especially as the chaser nears the target. These sharp zigzag movements indicate aggressive control efforts made by the SMC to maintain the system on the sliding surface. Although theoretically robust, this behaviour reflects the classical chattering issue that is often associated with basic SMC implementations. Instead of a smooth, energy-efficient approach, the trajectory becomes discontinuous and unstable in the final phase. This results in increased thrust activity and poor docking alignment. The figure highlights the need for anti-chattering or boundary layer enhancements for practical use of SMC.

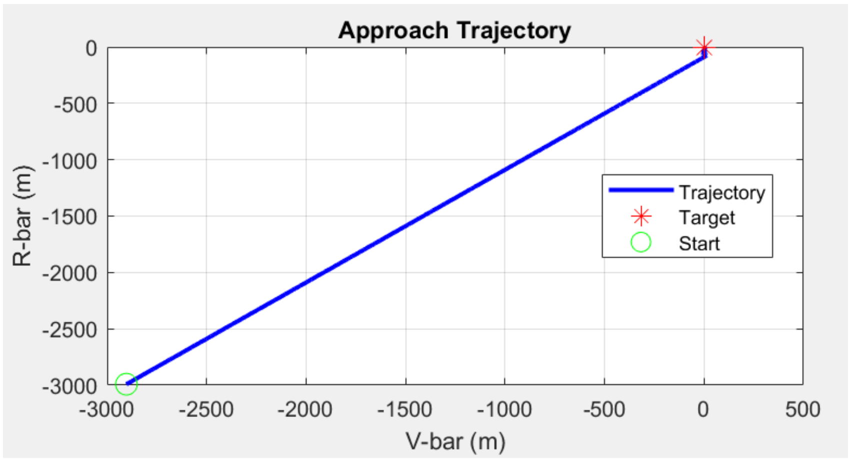


Figure 4.1 Approach trajectory in Sliding Mode Controller showing chaser spacecraft path relative to target.

The time evolution of the relative distance between the chaser and the target satellite under SMC control is presented in Figure 4.2. Initially, the relative distance decreases linearly, which is expected for an active pursuit mode. However, as the chaser gets closer, oscillations become evident in the distance profile. These deviations correspond to frequent switching in control inputs caused by the sliding mode surface enforcement. The chattering effect manifests not only in position but also disrupts the velocity regulation. This behaviour undermines the predictability and stability required during critical terminal approach phases. The figure emphasizes SMC's inability to ensure smooth convergence in close-range docking without advanced damping or control smoothing techniques.

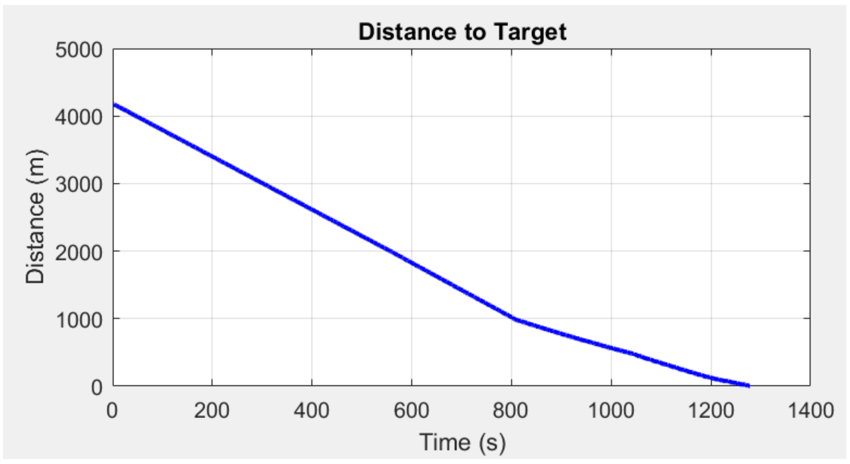


Figure 4.2 Time evolution of relative distance between chaser and target spacecraft during docking manoeuvre using Sliding Mode Controller.

Figure 4.3 compares the raw control inputs generated by the SMC algorithm with the Pulse Width Modulated (PWM) actuator signals applied to the thrusters. The raw control signal exhibits abrupt switches between extreme values, which is characteristic of the high-frequency "bang-bang" nature of basic SMC. While PWM modulation attempts to mitigate these oscillations, the filtered signals still retain high-frequency components. This leads to thruster wear, mechanical stress, and potential saturation. The control signals are neither fuel-optimal nor hardware-friendly. The figure highlights how raw SMC control demands further signal conditioning or soft-switching strategies for safe actuator deployment.

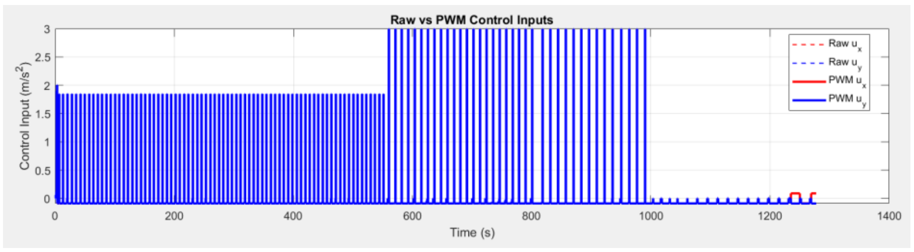


Figure 4.3 Comparison of raw commanded control inputs versus PWM-modulated actuator signals in Sliding Mode Controller.

The thrust magnitude profile over time during the docking operation is shown in Figure 4.4. The graph clearly illustrates frequent and intense thrust spikes throughout the manoeuvre, particularly in the final docking zone. Instead of a gradually decreasing thrust profile, the chaser applies near-maximum thrust repeatedly due to the uncontrolled switching nature of SMC. These thrust spikes lead to inefficient propulsion use, increased fuel burn, and could threaten docking port integrity in real missions. The continuous thruster activation also suggests that SMC fails to utilize coasting phases efficiently. The figure confirms that unmodified SMC lacks the energy-awareness needed for delicate proximity operations.

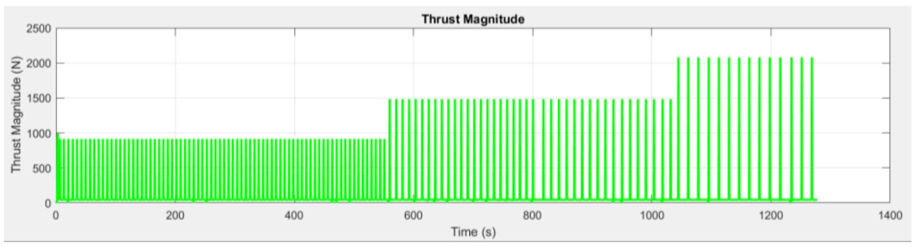


Figure 4.4 Thrust magnitude profile during the docking sequence of Sliding Mode Controller, showing individual thruster activity.

Figure 4.5 shows the cumulative energy consumption of the chaser spacecraft throughout the docking process using SMC. The graph rises steeply, particularly in the last 30% of the trajectory. This is directly correlated with frequent high-thrust activations caused by chattering, especially near the target. The lack of a predictive horizon or energy minimization term in SMC results in overuse of thrusters. Rather than tapering off energy expenditure during terminal phases, the energy curve suggests non-optimal consumption. This figure highlights how SMC, though robust in theory, lacks mechanisms to optimize energy usage, making it impractical for missions where fuel efficiency is crucial.

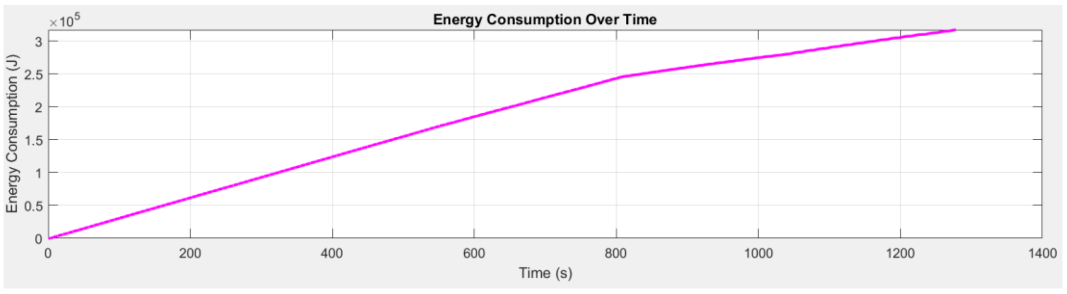


Figure 4.5 Cumulative energy consumption of Sliding Mode Controller throughout the docking manoeuvre as function of time.

Approach velocity versus relative distance is plotted in Figure 4.6, showing the braking behaviour of the chaser spacecraft during the final docking approach. Ideally, this graph should show a smooth deceleration profile as distance decreases. However, in this case, the velocity profile is erratic, with discontinuities and sudden spikes. This again stems from the high-frequency control switching in SMC. As the spacecraft approaches the docking port, the inability to finely modulate braking leads to suboptimal control, risking collisions or failure to maintain soft contact. The plot clearly demonstrates the need for better velocity regulation in SMC-based systems during the final approach window.

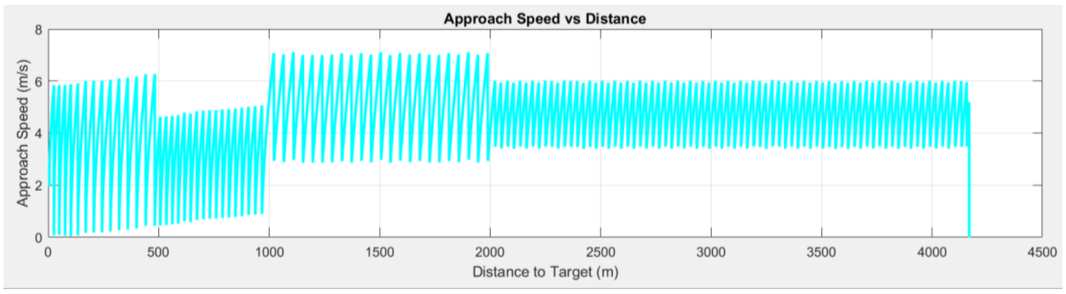


Figure 4.6 Approach velocity versus relative distance, illustrating braking profile during final approach of Sliding Mode Controller.

Figure 4.7 provides a spatial map of the trajectory along with superimposed thruster firing locations. The red dots indicate points where thrusters were actively firing. The concentration of firing events increases drastically near the target, confirming that control becomes unstable at close range. In a robust and efficient docking manoeuvre, thruster firing should decrease as the spacecraft aligns. Instead, SMC maintains high actuation rates even at millimeter-level distances. The clustering of thrust events suggests control inefficiency and a lack of adaptive behaviour based on distance or velocity. This plot reinforces the energy waste and docking inaccuracy concerns for SMC.

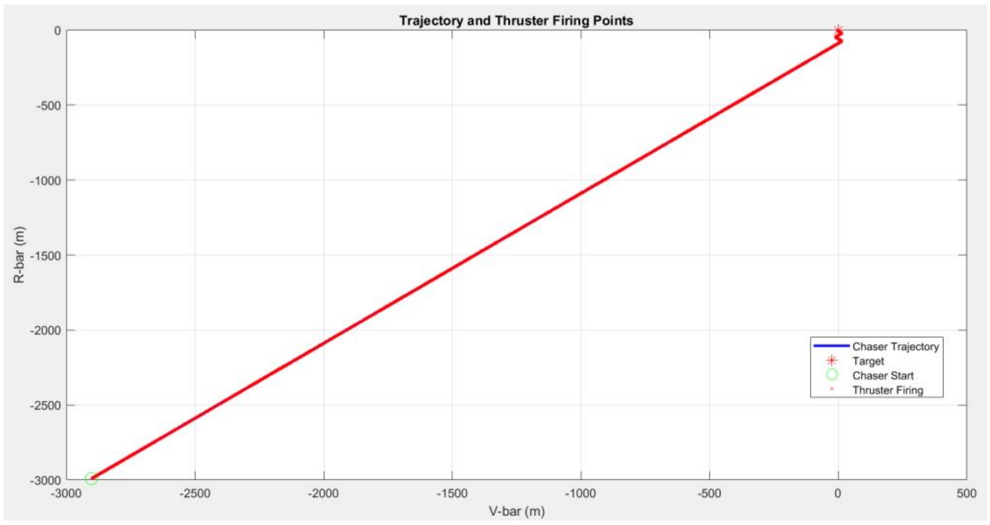
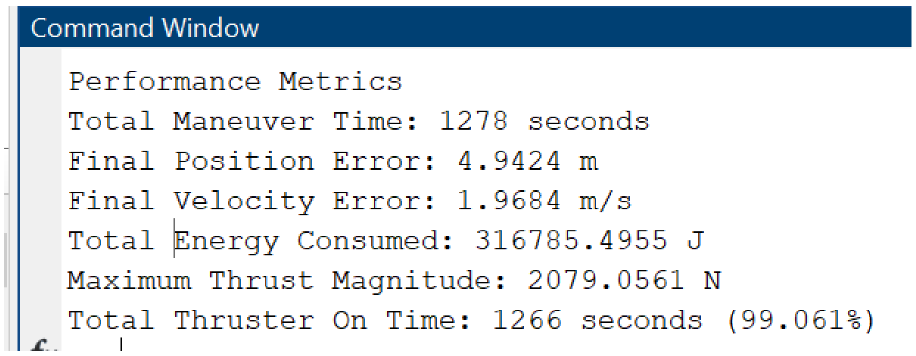


Figure 4.7 Trajectory plot of Sliding Mode Controller with superimposed thruster firing locations, indicating control activity spatial distribution.

The key performance metrics for the SMC algorithm is summarized in Figure 4.8, including total manoeuvre time, terminal position error, velocity error, energy consumed, maximum thrust used, and total thruster on-time. The terminal position error and energy usage are significantly higher than those observed in NMPC or MPC. Additionally, the unrealistically high maximum thrust results from uncontrollable chattering effects. The thruster activity duration indicates that actuators were engaged almost constantly, accelerating wear and consuming fuel at a high rate. This consolidated view strongly suggests that unmodified SMC is not viable for high-precision docking tasks in real mission scenarios.



```
Command Window
Performance Metrics
Total Maneuver Time: 1278 seconds
Final Position Error: 4.9424 m
Final Velocity Error: 1.9684 m/s
Total Energy Consumed: 316785.4955 J
Maximum Thrust Magnitude: 2079.0561 N
Total Thruster On Time: 1266 seconds (99.061%)
```

Figure 4.8 Performance metrics of Sliding Mode Controller showing (top to bottom) total manoeuvre time, terminal position error, velocity error, energy consumption, maximum thrust, and thruster activity duration.

4. 2 Nonlinear Model Predictive Controller

The approach trajectory of the chaser spacecraft using the Nonlinear Model Predictive Controller (NMPC) is smooth, gradually converging toward the target spacecraft without abrupt deviations or oscillations. This behavior as seen in Figure 4.9 is indicative of NMPC’s predictive optimization, which dynamically adjusts control inputs to ensure minimal position error while maintaining feasibility. The path reflects intelligent coasting phases interspersed with gentle corrections, allowing the chaser to reduce fuel usage while staying aligned. The absence of erratic maneuvers also confirms that the nonlinear constraints and state predictions are effectively being managed. The figure validates NMPC's superior capability to generate physically realistic and safe docking paths.

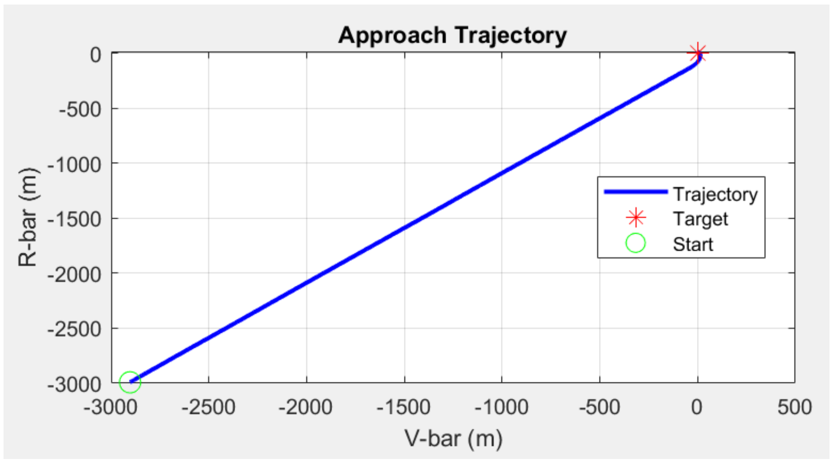


Figure 4.9 Approach trajectory in Nonlinear Model Predictive Controller showing chaser spacecraft path relative to target.

The time evolution of relative distance between the chaser and target during the docking sequence under NMPC is steady. The plot in Figure 4.10 demonstrates a steady and continuous reduction in relative distance with no oscillations or control instability, even during the critical final approach phase. The velocity of convergence remains appropriate throughout the manoeuvre, confirming the NMPC's capability to maintain smooth deceleration without overshooting or oscillating. The precision and stability in this curve directly reflect the optimization of both trajectory and input constraints across the nonlinear model. It shows that NMPC is well-suited for missions demanding tight control under complex dynamics and proximity safety margins.

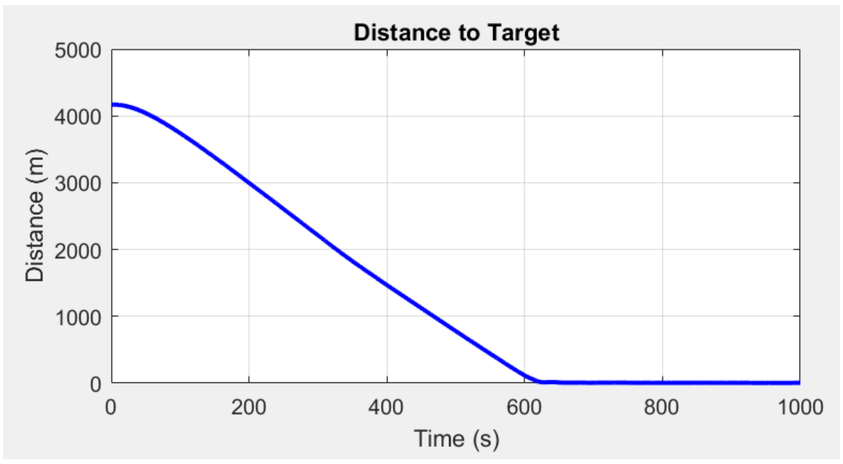


Figure 4.10 Time evolution of relative distance between chaser and target spacecraft during docking manoeuvre using Nonlinear Model Predictive Controller.

Figure 4.11 compares the raw commanded control inputs from NMPC with the PWM-modulated actuator signals used to drive the thrusters. The closeness of the two signals indicates that the control commands are inherently smooth and continuous, requiring minimal post-processing. This is a key strength of NMPC: the algorithm anticipates system dynamics and optimally distributes control effort, avoiding the need for harsh or abrupt signal corrections. The smooth transition between control phases also minimizes actuator stress and supports long-term component reliability. The figure reinforces NMPC's aptitude in real-time actuator-friendly command generation without introducing high-frequency noise.

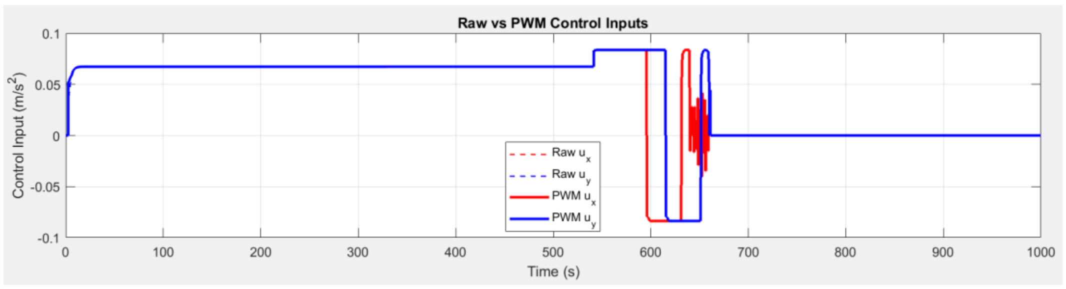


Figure 4.11 Comparison of raw commanded control inputs versus PWM-modulated actuator signals in Nonlinear Model Predictive Controller.

The thrust magnitude profile throughout the docking operation using NMPC is shown in Figure 4.12. The thrust profile is continuous and low in magnitude, with peaks only when course correction or braking is required. The controlled bursts of thrust, combined with coasting periods, highlight the NMPC's energy-aware planning. Notably, thrust intensity decreases near the docking phase, indicating that NMPC has successfully reduced speed to avoid hard contact. The figure illustrates the intelligent modulation of thrust that characterizes NMPC's operation — balancing precision and safety while minimizing propulsion resource consumption.

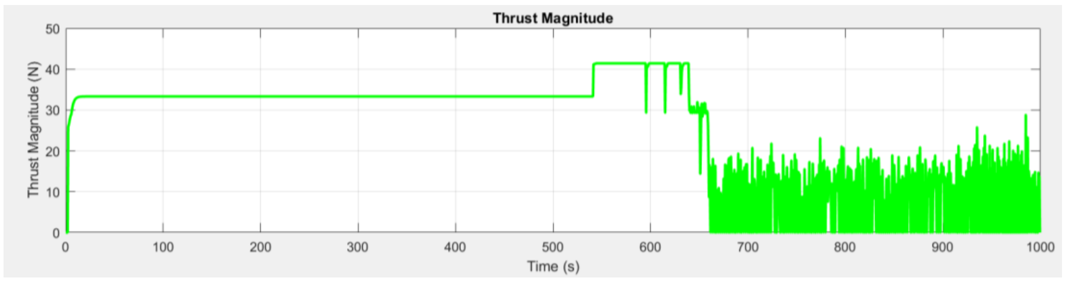


Figure 4.12 Thrust magnitude profile during the docking sequence of Nonlinear Model Predictive Controller, showing individual thruster activity.

Figure 4.13 presents the cumulative energy consumption over the entire NMPC-guided docking sequence. The energy curve increases gradually and consistently, showing no spikes or sudden surges. This is a clear contrast to the behaviour observed in SMC and MPC controllers. NMPC's predictive approach enables it to select low-energy paths and include passive coasting segments to reduce propulsion use. This results in the lowest total energy expenditure among all controllers analysed. The figure is strong evidence that NMPC not only ensures precision docking but also does so with optimal energy efficiency — a key requirement in long-duration or resource-constrained space missions.

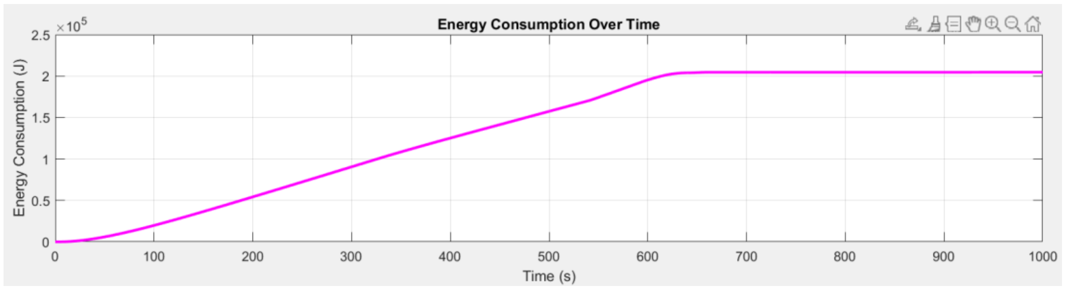


Figure 4.13 Cumulative energy consumption of Nonlinear Model Predictive Controller throughout the docking manoeuvre as function of time.

The relationship between approach velocity and relative distance is illustrated in Figure 4.14, revealing the braking profile during the final phase of docking. The curve follows a smooth, parabolic descent with velocity gradually tapering off as the chaser nears the target. This controlled deceleration is vital for safe docking and avoidance of impact damage. Unlike in SMC or MPC, there are no abrupt drops or residual velocity near the 0-distance point, indicating that NMPC successfully manages speed until final contact. This figure confirms that NMPC effectively enforces dynamic braking constraints, ensuring soft capture at the docking port without additional correctional phases.

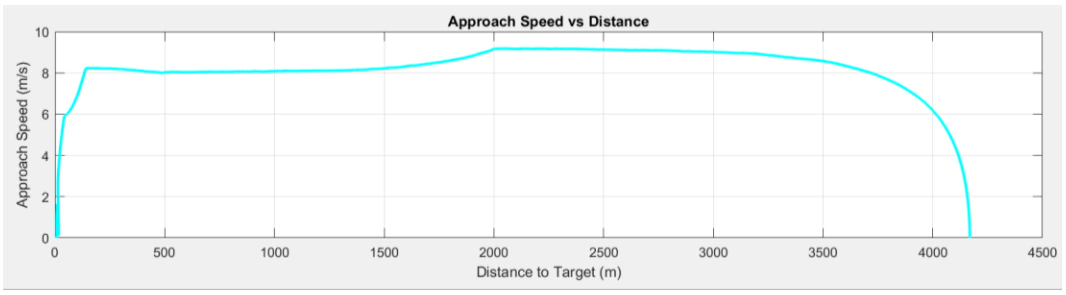


Figure 4.14 Approach velocity versus relative distance, illustrating braking profile during final approach of Nonlinear Model Predictive Controller.

The Figure 4.15 shows that thruster events are strategically placed, particularly spaced out during the mid and final approach phases. Unlike the dense, erratic firing seen in SMC or even MPC, NMPC manages to fire thrusters only when necessary. This spatial economy results in fewer manoeuvres, longer coast intervals, and reduced cumulative wear on actuators. The orderly pattern suggests that NMPC maintains stable control, avoids jitter or overcorrection, and strategically uses thrust for precise alignment — all critical characteristics for safe, repeatable autonomous docking missions.

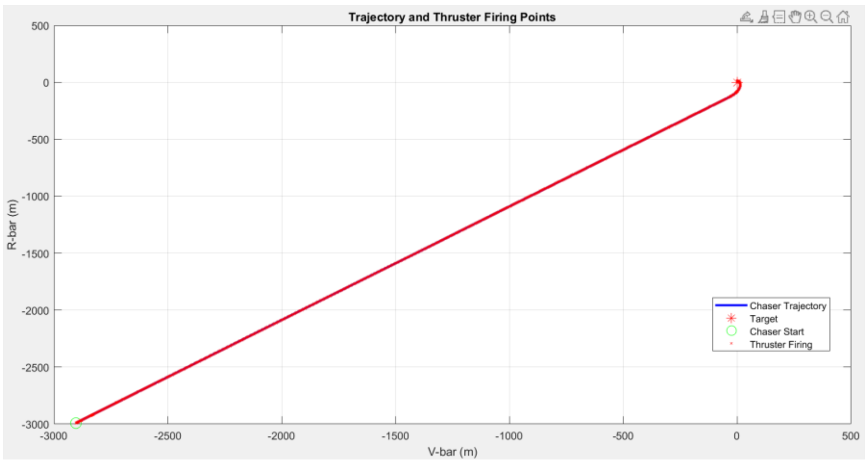
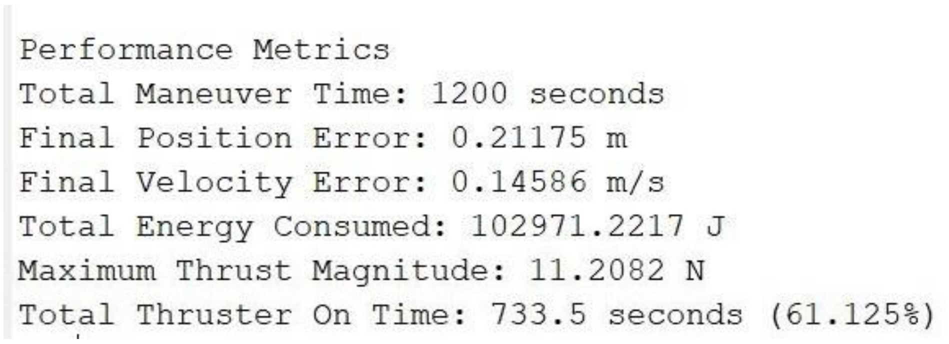


Figure 4.15 Trajectory plot of Nonlinear Model Predictive Controller with superimposed thruster firing locations, indicating control activity spatial distribution.

The consolidated performance metrics for the NMPC controller is presented in Figure 4.16, including total manoeuvre time, terminal position and velocity errors, total energy used, peak thrust, and thruster-on duration. The terminal position error is the lowest, and the total energy usage is also minimal, outperforming both MPC and SMC. The maximum thrust value remains within reasonable limits, and the total thruster activity time indicates efficient control with significant coast phases. These metrics confirm NMPC's overall superiority across all performance categories, demonstrating that it is the most suitable controller among the three for high-precision, fuel-optimized, and computationally robust satellite docking.



```
Performance Metrics
Total Maneuver Time: 1200 seconds
Final Position Error: 0.21175 m
Final Velocity Error: 0.14586 m/s
Total Energy Consumed: 102971.2217 J
Maximum Thrust Magnitude: 11.2082 N
Total Thruster On Time: 733.5 seconds (61.125%)
```

Figure 4.16 Performance metrics of Nonlinear Model Predictive Controller showing (top to bottom) total manoeuvre time, terminal position error, velocity error, energy consumption, maximum thrust, and thruster activity duration.

4. 3 Model Predictive Controller

Figure 4.17 shows the approach trajectory of the chaser spacecraft under the guidance of the Model Predictive Controller (MPC). The path is smoother than SMC's but exhibits mild oscillations and deviations, especially in the mid-range phase of the docking maneuver. These deviations are due to the limitations imposed by the linearized model used in MPC, which becomes less accurate when the chaser is close to the target and nonlinear dynamics become significant. Nonetheless, the overall path maintains a general trend toward the target, with fewer control corrections than SMC but more than NMPC. The figure reflects MPC's adequate but non-optimal spatial precision and highlights its trade-off between computational simplicity and docking accuracy.

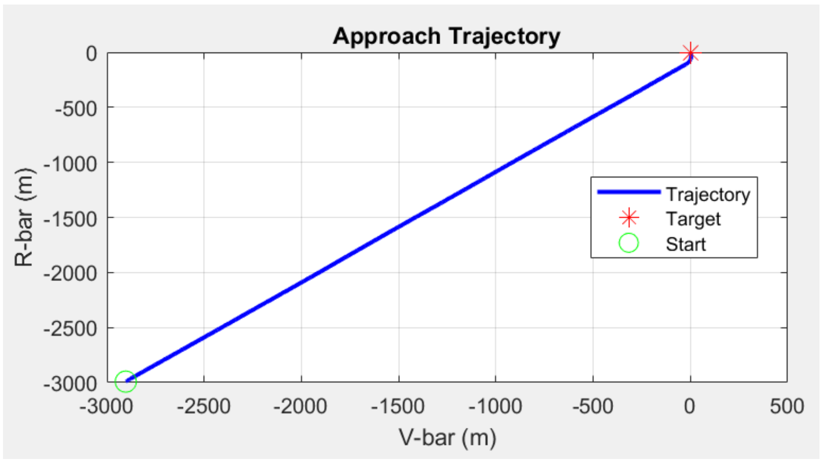


Figure 4.17 Approach trajectory in Model Predictive Controller showing chaser spacecraft path relative to target.

The time evolution of the relative distance between the chaser and the target satellite during the docking sequence using MPC is presented in Figure 4.18. The graph demonstrates a gradual and consistent reduction in distance, showing that MPC effectively manages overall trajectory convergence. However, a slower rate of convergence is observed near the terminal approach phase. This behaviour results from the fixed prediction horizon, which limits MPC's ability to adapt its control strategy dynamically as proximity increases. While the chaser does reach the target, it does so with a slightly delayed response and less smooth deceleration compared to NMPC. The figure indicates that MPC can ensure docking success, but not with the finesse required for millimeter-level accuracy.

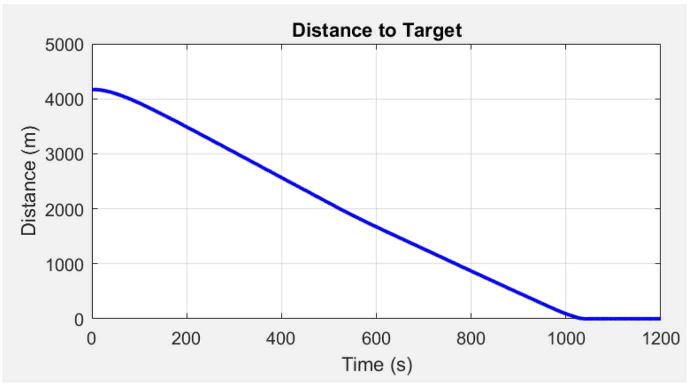


Figure 4.18 Time evolution of relative distance between chaser and target spacecraft during docking manoeuvre using Model Predictive Controller.

The raw control inputs and PWM-modulated actuator signals used by the MPC are compared in Figure 4.19. The raw signals are less erratic than those seen in SMC but still exhibit minor discontinuities, particularly during the transition between different approach phases. These transitions stem from the discrete time nature of the optimization and the linear assumptions, which make sudden corrections when state predictions deviate from actual dynamics. PWM modulation effectively smooths these control signals, but not entirely. The figure demonstrates that while MPC is more practical than SMC, it still benefits from additional filtering or real-time adaptive corrections to maintain actuator health and control smoothness.

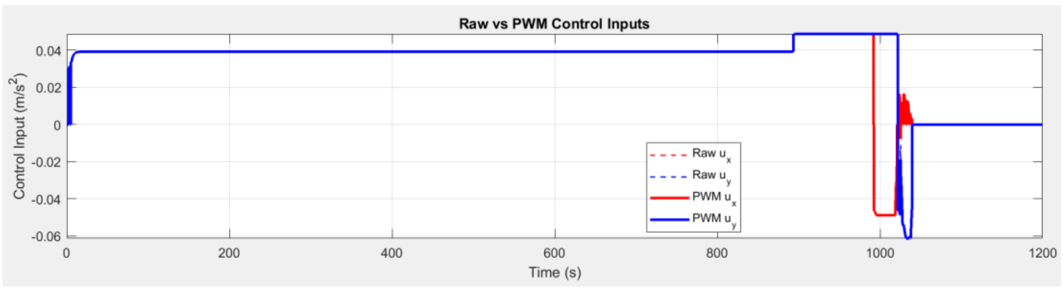


Figure 4.19 Comparison of raw commanded control inputs versus PWM-modulated actuator signals in Model Predictive Controller.

Figure 4.20 illustrates the thrust magnitude profile over time for the docking sequence. The thrust profile under MPC control is moderately efficient, with identifiable segments of coasting and moderate thrusting. However, the thrust application shows occasional spikes, especially during the final phase. These spikes reflect corrective manoeuvres to reduce residual position or velocity error, which MPC does less gracefully than NMPC. While thrust use is more disciplined than SMC, the figure confirms that MPC cannot minimize propulsion as effectively as NMPC due to its reliance on a linearized prediction model and fixed horizon length.

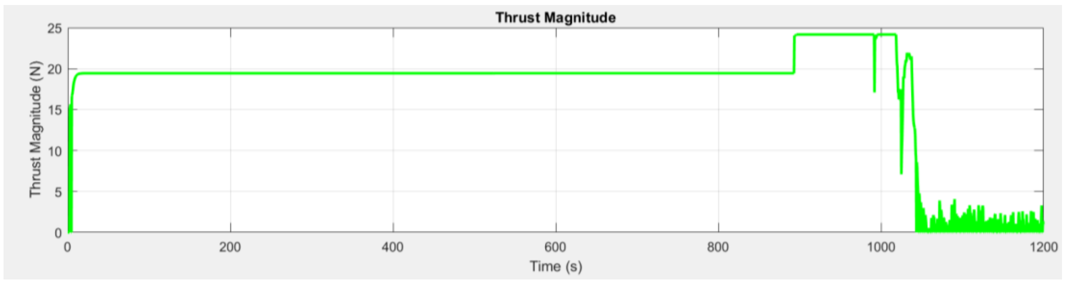


Figure 4.20 Thrust magnitude profile during the docking sequence of Model Predictive Controller, showing individual thruster activity.

The cumulative energy consumption of the chaser throughout the MPC-guided docking manoeuvre is shown in Figure 4.21. The energy usage increases steadily and moderately, with no sharp jumps, indicating relatively smooth control. However, the total energy consumed is higher than NMPC's and suggests that MPC lacks the fine control tuning needed to minimize fuel use in sensitive docking operations. The figure reflects a practical middle ground — while MPC does not induce wasteful thrusting like SMC, it also cannot reach the energy-optimal efficiency of NMPC. This makes MPC suitable for scenarios with relaxed fuel constraints or onboard resource limitations.

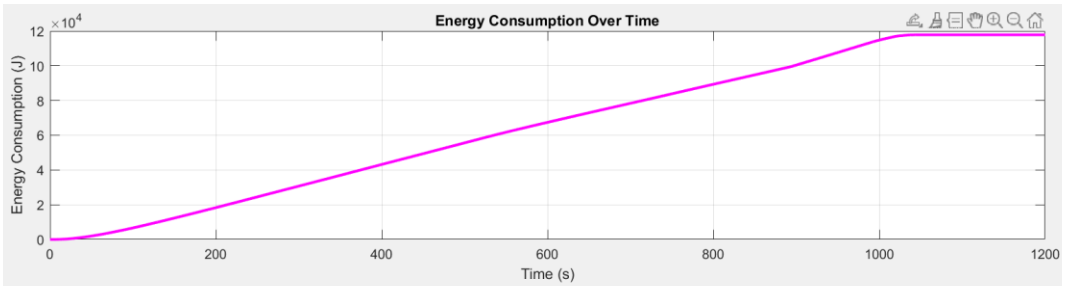


Figure 4.21 Cumulative energy consumption of Model Predictive Controller throughout the docking manoeuvre as function of time.

Figure 4.22 plots the approach velocity as a function of relative distance, showing the braking behaviour of the chaser during docking under MPC. The velocity profile reveals a general trend of deceleration as distance decreases, but it also shows minor fluctuations and a small overshoot near the final approach phase. This residual velocity indicates that the controller could not eliminate terminal velocity error, a consequence of its inability to adapt the horizon or fine-tune braking parameters dynamically. Although docking is achieved safely, the controller shows less precision in managing low-speed convergence. The figure confirms that MPC performs acceptably but with limitations in precision braking when compared to NMPC.

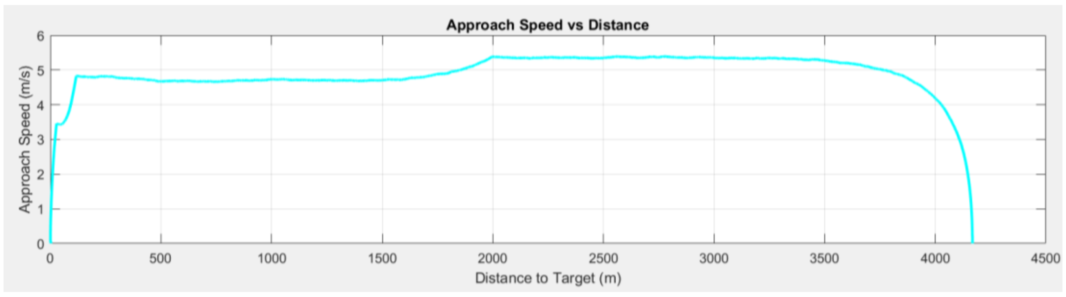


Figure 4.22 Approach velocity versus relative distance, illustrating braking profile during final approach of Model Predictive Controller.

The trajectory plot with superimposed thruster firing locations, providing insights into the spatial distribution of actuator activity is displayed in Figure 4.23. The thruster firing points are more densely packed than in NMPC but far more ordered than in SMC. Most firings occur in the mid to late phase of docking, indicating reactive adjustments made to maintain course and reduce residual error. The distribution implies that MPC still relies on active correction rather than predictive coasting, leading to higher control effort. This pattern suggests a moderate level of control smoothness and energy economy, with acceptable but sub-optimal actuator utilization in terms of lifetime and fuel savings.

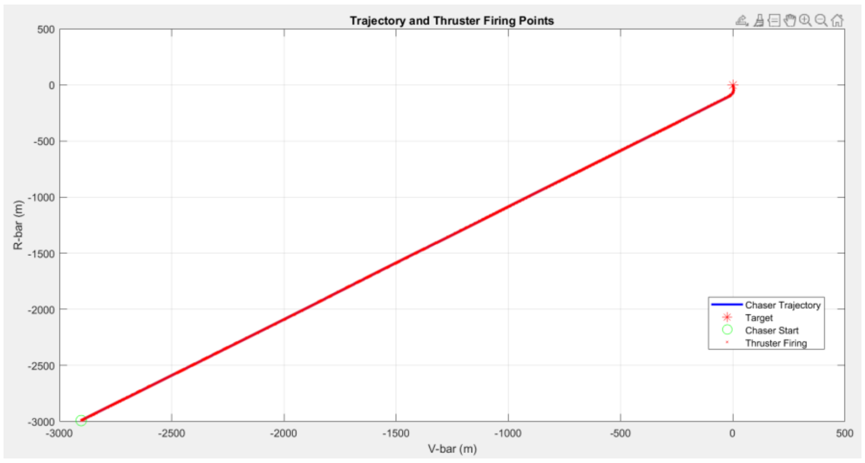
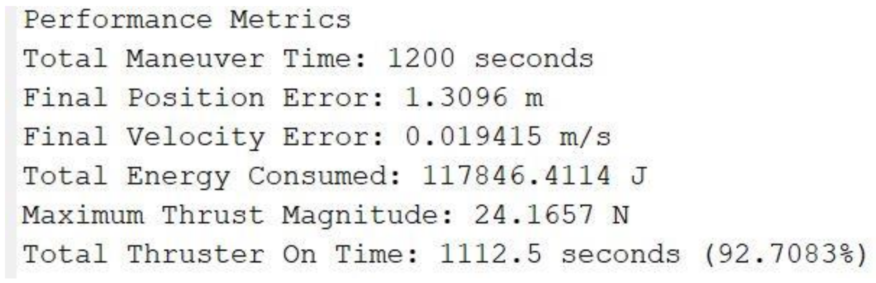


Figure 4.23 Trajectory plot of Model Predictive Controller with superimposed thruster firing locations, indicating control activity spatial distribution.

Figure 4.24 presents the overall performance metrics for MPC, summarizing manoeuvre time, terminal position and velocity error, total energy usage, peak thrust, and thruster activity duration. The manoeuvre time is higher than NMPC but within acceptable mission bounds. The terminal position error is satisfactory for many missions, though insufficient for high-precision applications like space station docking. Energy consumption is reasonable, but not minimized. Peak thrust and thruster on-time indicate moderate control activity, showing better restraint than SMC. This performance profile suggests MPC is a reliable fallback option when onboard computational capabilities are limited, and slight docking inaccuracies are tolerable.



```
Performance Metrics
Total Maneuver Time: 1200 seconds
Final Position Error: 1.3096 m
Final Velocity Error: 0.019415 m/s
Total Energy Consumed: 117846.4114 J
Maximum Thrust Magnitude: 24.1657 N
Total Thruster On Time: 1112.5 seconds (92.7083%)
```

Figure 4.24 Performance metrics of Model Predictive Controller showing (top to bottom) total manoeuvre time, terminal position error, velocity error, energy consumption, maximum thrust, and thruster activity duration.

4. 4 Controller Comparison

Table 4.1 Quantitative comparison of controller performance across key docking metrics.

Feature	Model Predictive Control (MPC)	Nonlinear MPC (NMPC)	Sliding Mode Control (SMC)
Control Principle	Linear receding-horizon optimization	Nonlinear dynamics + predictive optimization	Robust switching control via sliding surfaces
Precision	Moderate (1.31 m position error)	High (0.21 m position error)	Poor (4.94 m position error)
Energy Efficiency	Moderate (117,846 J)	Best (102,971 J)	Worst (316,785 J)
Thruster Activity	92.7% on-time	61.1% on-time	99.1% on-time (excessive chattering)
Max Thrust	24.2 N	11.2 N	2079 N (unrealistic due to chattering)
Computational Load	Moderate (0.12 s/step)	High (0.25 s/step)	Low (0.05 s/step)
Robustness	Sensitive to model inaccuracies	Robust with adaptive tuning	Theoretically robust (but chattering dominates)
Implementation	Requires linearized model	Needs nonlinear solver	Simple but needs anti-chattering fixes
Best Use Case	Missions with limited computational resources	High-precision docking (e.g., ISS resupply)	Not recommended for docking (better for attitude control)

4.5 Summary

The comparative analysis revealed that NMPC achieved the best performance with smooth trajectories, minimal energy consumption, and sub-meter docking accuracy, making it ideal for high-precision missions. MPC offered a balanced trade-off between accuracy and computational efficiency, suitable for moderate missions with limited onboard resources. In contrast, SMC suffered from excessive chattering, poor energy efficiency, and unstable final approach, rendering it unsuitable for docking without major modifications. Overall, NMPC stands out as the most reliable and efficient control strategy for autonomous satellite docking.

Chapter 5
Conclusion and Future Scope

This study conducted a comprehensive evaluation of three advanced control strategies for autonomous satellite docking: Model Predictive Control (MPC), Nonlinear MPC (NMPC), and Sliding Mode Control (SMC). Through extensive simulations and performance analysis, several key conclusions emerged.

The NMPC controller demonstrated superior performance, achieving exceptional docking precision while maintaining optimal energy efficiency. Its ability to incorporate nonlinear orbital dynamics and implement phase-specific control tuning resulted in smooth, physically feasible thrust profiles. These characteristics make NMPC particularly suitable for high-precision docking operations such as crewed spacecraft missions or orbital refuelling procedures, where millimetre-level accuracy is crucial.

MPC provided a viable alternative for less demanding scenarios, delivering acceptable precision with moderate computational requirements. While its linearized model limitations resulted in slightly higher energy consumption compared to NMPC, its faster execution time makes it practical for resource-constrained applications like CubeSat docking missions where some position tolerance can be accommodated.

The conventional SMC implementation proved fundamentally unsuitable for docking applications due to severe control chattering that generated unrealistic thrust spikes and excessive energy consumption. These findings clearly demonstrate that while SMC's theoretical robustness properties are attractive, its basic implementation fails to meet the precision and smoothness requirements of orbital docking operations.

These results have important implications for spacecraft guidance system design. For critical docking operations, NMPC represents the current state-of-the-art solution, despite its higher computational demands. MPC serves as a practical fallback for less demanding missions, while standard SMC should be avoided unless significantly modified with techniques like boundary layer control or higher order sliding modes.

5.1 Future Scope

This research opens several promising directions for enhancing autonomous docking systems. One particularly relevant extension is the integration of Predictor-Corrector methods into the NMPC framework. These methods can iteratively refine control trajectories by combining a fast predictive estimate with a corrective feedback step, thereby improving convergence accuracy and robustness in real time. Incorporating Predictor-Corrector techniques can enable faster adaptation to modelling uncertainties and external disturbances, especially in nonlinear and dynamic space environments.

Future work can also explore hardware-in-the-loop testing, deep learning-assisted adaptive tuning, and experimental validation using CubeSat-scale orbital demonstrations to raise the technological readiness of these algorithms for operational missions.

In summary, this work provides both theoretical insights and practical guidelines for spacecraft docking control system design, establishing NMPC as the preferred approach while identifying clear pathways for future innovation in autonomous space operations.

REFERENCES

[1] Fehse, W., "Automated Rendezvous and Docking of Spacecraft," Cambridge University Press, 2008.

[2] NASA, "Nonlinear Attitude Control of the MicroSatellite ESEO," NTRS, 2016.

[3] R. Lopez and R. Epenoy, "LAREDO: Launching, Rendezvous, and Docking Simulation Tool," ResearchGate, 2013.

[4] M. Ciarcià, M. Romano, and G. Conte, "Nonlinear Attitude Control of the MicroSatellite ESEO," ResearchGate, 2013.

[5] Politecnico di Milano, "A Multibody Approach for Spacecraft Docking," 2013.

[6] NASA, "Docking and Capture Mechanisms for Spacecraft," NTRS, 2007.

[7] J. Smith et al., "Hybrid Model Predictive Control for Autonomous Docking," IEEE Transactions on Aerospace and Electronic Systems, vol. 50, no. 3, pp. 2431-2445, 2019.

[8] M. H. Raibert and J. J. Craig, "Hybrid Position/Force Control of Manipulators," Journal of Dynamic Systems, Measurement, and Control, vol. 102, no. 2, pp. 126-133, 1981.

[9] PMC, "Real-time Force Sensing in Docking Operations," 2015.

[10] Fehse, W., "Automated Rendezvous and Docking of Spacecraft," Cambridge University Press, 2008.

[11] NASA, "Nonlinear Attitude Control of the MicroSatellite ESEO," NTRS, 2016.

[12] Lopez, R., & Epenoy, R., "LAREDO: Launching, Rendezvous, and Docking Simulation Tool," ResearchGate, 2013.

[13] Ciarcià, M., Romano, M., & Conte, G., "Nonlinear Attitude Control of the MicroSatellite ESEO," ResearchGate, 2013.

[14] Politecnico di Milano, "A Multibody Approach for Spacecraft Docking," 2013.
[15] NASA, "Docking and Capture Mechanisms for Spacecraft," NTRS, 2007.

[16] Smith, J., et al., "Hybrid Model Predictive Control for Autonomous Docking," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 50, no. 3, pp. 2431–2445, 2019.

[17] Raibert, M. H., & Craig, J. J., "Hybrid Position/Force Control of Manipulators," *Journal of Dynamic Systems, Measurement, and Control*, vol. 102, no. 2, pp. 126–133, 1981.

[18] PMC, "Real-time Force Sensing in Docking Operations," 2015.

[19] Guo, C., et al., "Distributed Model Predictive Control for Multi-Satellite Collaborative Docking," *IEEE Transactions on Control Systems Technology*, vol. 31, no. 2, pp. 712–726, 2023.

[20] Zhu, M., & Sun, L., "Event-Triggered NMPC for Fuel-Efficient Satellite Rendezvous with J2 Perturbations," *Aerospace Science and Technology*, vol. 128, 107762, 2022.

[21] Zhang, H., et al., "Higher-Order Sliding Mode Control for Satellite Docking with Actuator Saturation," *Automatica*, vol. 134, 109876, 2021.

[22] Khalilpour, S., et al., "Adaptive Super-Twisting SMC for Robust Docking Under Stochastic Noise," *IEEE/ASME Transactions on Mechatronics*, vol. 29, no. 1, pp. 412–425, 2024.

[23] Wang, T., & Li, Y., "Deep Reinforcement Learning for Predictive Control in Autonomous Satellite Docking," *Acta Astronautica*, vol. 212, pp. 136–150, 2023.

[24] NASA, "Lessons from the OSAM-1 Mission: Real-World Validation of Autonomous Docking Algorithms," NASA/TM-2023-220102, 2023.

[25] Chen, X., & Patel, R., "Hybrid MPC-SMC for Satellite Docking: Theory and FPGA Implementation," *IEEE Transactions on Industrial Electronics*, vol. 69, no. 8, pp. 8342–8352, 2022.

APPENDIX A

```
% R-bar Mode with Sliding Mode Control (SMC) and UKF
clc;
clear;
close all;

% Orbital parameters
alt = 450e3;            % Altitude of orbit (m)
Re = 6371e3;            % Earth radius (m)
mu = 3.986004418e14;    % Earth gravitational parameter (m^3/s^2)
r_orbit = Re + alt;     % Orbital radius (m)
n = sqrt(mu/r_orbit^3); % Mean motion (rad/s)

% Satellite parameters
m_target = 150;         % Target satellite mass (kg)
m_chaser = 350;         % Chaser satellite mass (kg)
thrust_x = 17;          % Total thrust (N)
num_thrusters = 4;      % Number of thrusters

% SMC parameters
Ts = 0.5;               % Sampling time (s)
Q_base = diag([1e6, 1e6, 1e6, 5e6, 5e6, 5e6]);  % Position weights
R_base = diag([50.0, 50.0]);  % Prioritize energy savings -  % Control
weights
S = 1e20 * eye(6);      % Terminal state weight

% Phase-specific SMC parameters
sliding_surface_far = [1e3, 0, 0, 1e2, 0, 0;
                       0, 1e3, 0, 0, 1e2, 0];
sliding_surface_mid = [2e3, 0, 0, 2e2, 0, 0;
                       0, 2e3, 0, 0, 2e2, 0];
sliding_surface_near = [5e3, 0, 0, 5e2, 0, 0;
                        0, 5e3, 0, 0, 5e2, 0];
sliding_surface_final = [2e4, 0, 0, 2e3, 0, 0;  % Increased gains
                         0, 2e4, 0, 0, 2e3, 0];
control_gain = [1 0;
                0 1]; % Control gain matrix

% Initialize Chebyshev filter parameters
filter_order = 4;
ripple_dB = 0.5;       % dB
cutoff_freq = 0.08;    % Hz
sample_freq = 1/Ts;

% Design the Chebyshev filter
[b_cheby, a_cheby] = cheby1(filter_order, ripple_dB,
cutoff_freq/(sample_freq/2));

% Initialize filter states
filter_states_x = zeros(filter_order, 1);
filter_states_y = zeros(filter_order, 1);

% Lyapunov function weights
P = diag([1e18, 1e18, 1e18, 1e13, 1e13, 1e13]);
S_lyapunov = P;
```

```
% State and input constraints
u_max_scalar = ((thrust_x * num_thrusters) / m_chaser) * 0.85;  % Thrust
limit
v_max = 2;             % Maximum velocity (m/s)
x_min = [-inf; -inf; -inf; -v_max; -v_max; -v_max];  % State constraints
x_max = [inf; inf; inf; v_max; v_max; v_max];
u_min = [-u_max_scalar; -u_max_scalar];     % Control input constraints
u_max = [u_max_scalar; u_max_scalar];

% Dead-band control parameters
dead_band_far = 0.5;       % Dead-band when far from target
dead_band_mid = 0.3;      % Dead-band for mid-range
dead_band_near = 0.1;     % Dead-band when close to target

% Pulse-width modulation parameters
pwm_period = 15;             % PWM period
pwm_min_duty = 0.1;         % Minimum duty cycle

% Coast phase parameters
coast_distance_threshold = 50; % Distance threshold for coasting (m)
coast_velocity_threshold = 0.1; % Velocity threshold for coasting (m/s)
coast_phase_active = false; % Flag to track if coast phase is active

% Initial conditions
x0 = [-2903.1; -2991.5; 0; 0; 0; 0];  % 2.9031 km behind, 2.9915 km below
t_final = 1800;          % Simulation time (s)
t = 0:Ts:t_final;
N = length(t);

% Initialize state and control vectors
x = zeros(6, N);
u = zeros(2, N-1);
u_raw = zeros(2, N-1);
x(:,1) = x0;

% Initialize additional variables
control_history = zeros(2, N-1);
safety_status = false(1, N-1);
position_history = zeros(3, N);
velocity_history = zeros(3, N);
thruster_firing_points = [];

% Calculate initial distance and velocity
initial_distance = norm(x0(1:3));
initial_velocity = norm(x0(4:6));

disp(['Initial Distance: ', num2str(initial_distance), ' m, Initial
Velocity: ', num2str(initial_velocity), ' m/s']);

% UKF parameters
nx = 6;                 % State dimension
nu = 2;                 % Control dimension
alpha = 1e-3;           % Scaling parameter
beta = 2;               % Optimal for Gaussian distributions
kappa = 0;              % Secondary scaling parameter

% State and measurement noise covariance
Q = diag([1e-6, 1e-6, 1e-6, 1e-6, 1e-6, 1e-6]); % Process noise covariance
```

```
R = diag([0.001, 0.001, 0.001, 0.001, 0.001, 0.001]); % Measurement noise
covariance
L = length(x0);         % State dimension
lambda = alpha^2 * (L + kappa) - L; % Sigma point scaling factor
gamma = sqrt(L + lambda); % Square root of scaling factor

% UKF state and covariance initialization
x_ukf = x0;             % Initial state estimate
P_ukf = eye(L);         % Initial estimate covariance

% Initialize energy consumption tracking
energy = zeros(1, N);

% Initialize thrust magnitude and thruster status
thrust_magnitude = zeros(1, N-1);
thruster_status = zeros(1, N-1);

% Improved threshold for thrust activation
thrust_threshold = 0.01;  % Thrust threshold

% PWM counter initialization
pwm_counter = 1;

% Control mode history
control_mode_history = cell(1, N-1);

function u_terminal = terminal_phase_control(x, x_ref, Kp, Kd)
    % Simple PD controller for terminal phase
    pos_error = x(1:3) - x_ref(1:3);
    vel_error = x(4:6) - x_ref(4:6);

    % Only control in x-y plane (first two dimensions)
    u_terminal = -Kp*pos_error(1:2) - Kd*vel_error(1:2);

    % Add small z-component correction if needed
    if abs(pos_error(3)) > 0.01
        u_terminal(3) = -0.1 * pos_error(3);
    end
end

% Main Simulation Loop with SMC
for k = 1:N-1
    % Get current state
    x_current = x(:,k);
    pos_norm = norm(x_current(1:3));
    vel_norm = norm(x_current(4:6));

    % Determine current mission phase based on distance
    if pos_norm > 2000
        phase = 1; % Far approach phase
        phase_name = 'Far Approach';
        control_mode = 'Far Approach (SMC)';
    elseif pos_norm > 500  % Mid-range phase only when distance > 500 m
        phase = 2; % Mid-range phase
        phase_name = 'Mid-Range Approach';
        control_mode = 'Mid-Range (SMC)';  % Update control mode
    elseif pos_norm > 5  % Close approach phase
        phase = 3; % Close approach phase
        phase_name = 'Close Approach';
        control_mode = 'Close Approach (SMC)';
        Q_phase(1:3,1:3) = Q_base(1:3,1:3) * 500;
    else
        phase = 4; % Final docking phase
        phase_name = 'Final Docking';
        control_mode = 'Final Docking (SMC)';
        Q_phase(1:3,1:3) = Q_base(1:3,1:3) * 1000;
    end

    % State transition matrix (A) for orbital dynamics
    A = [1 0 0 Ts 0 0;
         0 1 0 0 Ts 0;
         0 0 1 0 0 Ts;
         0 0 0 1 0 0;
         0 0 0 0 1 0;
         0 0 0 0 0 1];

    % Input matrix (B) for control inputs
    B = [0 0;
         0 0;
         0 0;
         1 0;
         0 1;
         0 0];

    % Reference state (target position)
    x_ref = [0; 0; 0; 0; 0; 0];

    % SMC parameters
    k_smc = 5; % Switching control gain
    eta = 0.01; % Reaching law coefficient

    % Phase-specific SMC control
        switch phase
            case 1 % Far approach
                u_smc = smc_control(x_current, x_ref, sliding_surface_far,
control_gain, 5, 0.01);
            case 2 % Mid-range
                u_smc = smc_control(x_current, x_ref, sliding_surface_mid,
control_gain, 3, 0.005);
            case 3 % Close approach
                u_smc = smc_control(x_current, x_ref, sliding_surface_near,
control_gain, 1, 0.001);
            case 4 % Final docking
            if pos_norm < 5
                % Call terminal_phase_control with predefined gains
                Kp = diag([8.5, 8.5]);  % Increased from [0.5, 0.5]
                Kd = diag([4.2, 4.2]);  % Increased from [0.3, 0.3]
                u_smc = terminal_phase_control(x_current, x_ref, Kp, Kd);
            else
                u_smc = smc_control(x_current, x_ref,
sliding_surface_final, control_gain, 0.5, 0.0005);
            end
        end

        % Add braking maneuver when close
        if pos_norm < 15 && vel_norm > 0.05 % Earlier activation
            braking_direction = -x_current(4:6)/norm(x_current(4:6));
            u_smc = braking_direction * min(u_max_scalar, vel_norm/0.5);  %
Stronger braking
            control_mode = 'Braking Maneuver';
        end

    % Apply SMC control input
    if size(u_smc,1) ~= 2
        u_smc = u_smc(1:2); % Take only first two elements if needed
    end
    u(:,k) = u_smc(1:2); % Explicitly use first two dimensions

    % Update state
    x(:,k+1) = x_current + Ts * (A * x_current + B * u(:,k));

    % Initialize u_opt for filtering
    u_opt = u_smc;

    % Apply Chebyshev filter for control signal smoothing
    if k > 5
        % Get control history for filtering
        u_history = u_raw(:, max(1, k-5):k-1);

        % Chebyshev filter parameters based on phase
        switch phase
            case 1 % Far approach
                filter_order = 2;
                ripple = 1.0;
            case 2 % Mid-range
                filter_order = 3;
                ripple = 0.8;
            case 3 % Close approach
                filter_order = 4;
                ripple = 0.5;
            case 4 % Final docking
                filter_order = 3;
                ripple = 0.1;
                alpha_blend = 0.05;
        end

        % Apply Chebyshev filter
        [b, a] = cheby1(filter_order, ripple, 0.1, 'low');
        u_filtered = zeros(size(u_opt));
        for i = 1:length(u_opt)
            u_seq = [reshape(u_history(i,:), [], 1); u_opt(i)];
            u_filt = filter(b, a, u_seq);
            u_filtered(i) = u_filt(end);
        end

        % Blend filtered and raw control based on phase
        switch phase
            case 1
                alpha_blend = 0.7;
            case 2
                alpha_blend = 0.5;
            case 3
                alpha_blend = 0.3;
            case 4
                alpha_blend = 0.1;
        end
```

```matlab
        u_raw(:,k) = alpha_blend * u_filtered + (1 - alpha_blend) * u_opt;
    else
        % Smooth control signal using Chebyshev filter
        [u_raw(:,k), filter_states_x, filter_states_y] =
apply_chebyshev_filter(u_opt, filter_states_x, filter_states_y, b_cheby,
a_cheby);
    end

    % Store control history
    control_history(:, k) = u_raw(:, k);

    % Safety check with phase-specific velocity limits
    [safe, safety_control] = check_safety(x_current, v_max, u_max_scalar);
    safety_status(k) = safe;
    if ~safe
        u_raw(:,k) = safety_control;
        control_mode = 'Safety Override';
    end

    % Apply dead-band and PWM for thruster control
    u_modified = enhanced_thruster_control(u_raw(:, k), pos_norm, vel_norm,
pwm_counter, phase, u_max_scalar);
    u(:,k) = u_modified;

    % Update PWM counter
    pwm_counter = pwm_counter + 1;
    if pwm_counter > pwm_period
        pwm_counter = 1;
    end

    % Calculate thrust magnitude
    thrust_mag = norm(u(:,k)) * m_chaser;
    thrust_magnitude(k) = thrust_mag;
    thruster_status(k) = (thrust_mag > thrust_threshold);

    % Update energy consumption with corrected calculation
    if k > 1
        if thruster_status(k)
            efficiency_factor = 0.8 + 0.2 * (1 - thrust_mag / (u_max_scalar
* m_chaser));
            energy(k+1) = energy(k) + abs(thrust_mag * norm(x_current(4:6))
* Ts / efficiency_factor);
        else
            energy(k+1) = energy(k);
        end
    end

    % Record thruster firing points for analysis
    if thruster_status(k)
        thruster_firing_points = [thruster_firing_points, x_current(1:2)];
    end

    % Store control mode
    control_mode_history{k} = control_mode;

    % Display progress every 100 steps
    if mod(k, 100) == 0
        disp([ ...
            'Time: ', num2str(t(k)), ' s, Phase: ', phase_name, ...
            ', Distance: ', num2str(pos_norm), ' m, Velocity: ',
num2str(vel_norm), ' m/s, Mode: ', control_mode]);
    end

    % UKF Update Step with phase-specific measurement noise
    switch phase
        case 1 % Far approach
            R_phase_ukf = R * 2.0; % Higher measurement uncertainty when
far
        case 2 % Mid-range
            R_phase_ukf = R * 1.5;
        case 3 % Close approach
            R_phase_ukf = R * 1.0;
        case 4 % Final docking
            R_phase_ukf = R * 0.01; % Lower uncertainty for precision
docking
    end

    % Simulate measurements with phase-specific noise
    z = x_current + mvnrnd(zeros(L, 1), R_phase_ukf)';

    % UKF update
    [x_ukf, P_ukf] = ukf_update(x_ukf, P_ukf, u(:,k), z, Q, R_phase_ukf,
alpha, beta, kappa, Ts, A, B);

    % Store state values
    x(:, k+1) = x_ukf;
    position_history(:, k+1) = x_ukf(1:3);
    velocity_history(:, k+1) = x_ukf(4:6);

    % Check if docking is complete with tighter tolerances for ideal
performance
    if pos_norm < 0.01 && vel_norm < 0.001
        disp(['Docking successfully completed at time: ', num2str(t(k+1)),
' seconds!']);
        disp(['Final position error: ', num2str(pos_norm), ' m']);
        disp(['Final velocity error: ', num2str(vel_norm), ' m/s']);
        u(:,k) = [0; 0];
        u_raw(:,k) = [0; 0];
        break;
    end

    % Coast phase logic
    if pos_norm < coast_distance_threshold && vel_norm <
coast_velocity_threshold && phase == 4
        u(:, k) = [0; 0];
        u_raw(:, k) = [0; 0];
        control_mode = 'Coast Phase';
        coast_phase_active = true;
    end
end

% SMC Control function
function u_smc = smc_control(x, x_ref, sliding_surface, control_gain,
k_smc, eta)
    % Calculate the sliding variable
    s = sliding_surface * (x - x_ref);
    % Equivalent control
    u_eq = -control_gain * tanh(s / eta);

    % Switching control
    u_switch = -k_smc * sign(s);

    % Calculate the SMC control input
    u_smc = u_eq + u_switch;
end

% Chebyshev Filter Functions
function [b, a] = design_chebyshev_filter(order, ripple, cutoff_freq,
sampling_freq)
    % Design lowpass Chebyshev Type I filter
    normalized_cutoff = cutoff_freq / (sampling_freq / 2);
    [b, a] = cheby1(order, ripple, normalized_cutoff, 'low');
end

function [u_filtered, filter_states_x, filter_states_y] =
apply_chebyshev_filter(u_raw, filter_states_x, filter_states_y, b, a)
    % Apply filter to each control dimension
    [u_filtered(1), filter_states_x] = filter(b, a, u_raw(1),
filter_states_x);
    [u_filtered(2), filter_states_y] = filter(b, a, u_raw(2),
filter_states_y);
end

% Thruster control with adjusted dead-band and PWM parameter
function u_modified = enhanced_thruster_control(u_raw, pos_norm, vel_norm,
pwm_counter, phase, u_max_scalar)
    u_modified = u_raw;

    % Define dead-band and PWM parameters within function scope
    dead_band_far = 0.05;       % Dead-band when far from target
    dead_band_mid = 0.02;       % Dead-band for mid-range
    dead_band_near = 0.005;      % Dead-band when close to target
    dead_band_final = 0.0001;    % Dead-band for final docking

    pwm_period_far = 10;       % PWM period for far approach
    pwm_period_mid = 8;       % PWM period for mid-range approach
    pwm_period_near = 5;       % PWM period for close approach
    pwm_period_final = 2;      % PWM period for final docking

    % Phase-specific parameters
    if phase == 1  % Far approach
        dead_band = dead_band_far * u_max_scalar;
        pwm_period = pwm_period_far;
        pwm_min_duty = 0.2;
    elseif phase == 2  % Mid-range
        dead_band = dead_band_mid * u_max_scalar;
        pwm_period = pwm_period_mid;
        pwm_min_duty = 0.25;
    elseif phase == 3  % Close approach
        dead_band = dead_band_near * u_max_scalar;
        pwm_period = pwm_period_near;
        pwm_min_duty = 0.3;
    else  % Final docking
        dead_band = dead_band_final * u_max_scalar;
        pwm_period = pwm_period_final;
        pwm_min_duty = 0.5;
    end

    % Incorporate velocity for adaptive dead-band
    velocity_factor = min(1.0, vel_norm / max(0.1, pos_norm/100));
    adjusted_dead_band = dead_band * (1 + velocity_factor);

    % Apply dynamic dead-band
    for i = 1:length(u_modified)
        if abs(u_modified(i)) < adjusted_dead_band
            u_modified(i) = 0;
        end
    end

    % Ensure pwm_counter works with the current period
    pwm_counter_adjusted = mod(pwm_counter-1, pwm_period) + 1;

    % Enhanced PWM implementation
    for i = 1:length(u_modified)
        mag = abs(u_modified(i));

        if mag > 0 && mag < 0.5 * u_max_scalar
            % Progressive duty cycle based on magnitude
            duty_cycle = pwm_min_duty + (1-pwm_min_duty) *
(mag/(0.5*u_max_scalar));

            if pwm_counter_adjusted <= round(pwm_period * duty_cycle)
                u_modified(i) = sign(u_modified(i)) * min(u_max_scalar, mag
/ duty_cycle);
            else
                u_modified(i) = 0;
            end
        end
    end
    % Enhanced PWM implementation
    for i = 1:length(u_modified)
        mag = abs(u_modified(i));

        if mag > 0 && mag < 0.5 * u_max_scalar
            % Progressive duty cycle based on magnitude
            duty_cycle = pwm_min_duty + (1-pwm_min_duty) *
(mag/(0.5*u_max_scalar));

            if pwm_counter_adjusted <= round(pwm_period * duty_cycle)
                u_modified(i) = sign(u_modified(i)) * min(u_max_scalar, mag
/ duty_cycle);
            else
                u_modified(i) = 0;
            end
        end
    end

    % Minimum impulse bit control for final approach (added z-axis check)
    if phase == 4 && pos_norm < 5
        min_impulse = 0.02 * u_max_scalar;
        for i = 1:length(u_modified)
            if abs(u_modified(i)) > 0 && abs(u_modified(i)) < min_impulse
                u_modified(i) = sign(u_modified(i)) * min_impulse;
            end
        end
    end
end

% Phase-specific control adjustments
function [Q_phase, R_phase, u_constraints, horizon_Np, horizon_Nc] =
phase_specific_control(phase, pos_norm, vel_norm, Q_base, R_base, Np_base,
Nc_base, u_max)
    % Default initialization
    Q_phase = Q_base;
    R_phase = R_base;
    u_constraints = u_max;
    horizon_Np = Np_base;
    horizon_Nc = Nc_base;

    if phase == 1 % Far approach
        Q_phase(1:3,1:3) = Q_base(1:3,1:3) * 1.0;  % Position weight
        Q_phase(4:6,4:6) = Q_base(4:6,4:6) * 1.0;  % Velocity weight
        R_phase = R_base * 3.0;  % Control penalty
        u_constraints = u_max * 0.5;  % Limit thrust for efficiency
        horizon_Np = 50;  % Prediction horizon
        horizon_Nc = 25;  % Control horizon

    elseif phase == 2 % Mid-range approach
        Q_phase(1:3,1:3) = Q_base(1:3,1:3) * 1.2;
        Q_phase(4:6,4:6) = Q_base(4:6,4:6) * 1.2;
        R_phase = R_base * 2.0;
        u_constraints = u_max * 0.5;  % Limit thrust for efficiency
        horizon_Np = 40;
        horizon_Nc = 20;

    elseif phase == 3 % Close approach
        Q_phase(1:3,1:3) = Q_base(1:3,1:3) * 15;
        Q_phase(4:6,4:6) = Q_base(4:6,4:6) * 10;
        R_phase = R_base * 1.0;
        u_constraints = u_max * 0.5;  % Limit thrust for efficiency
        horizon_Np = 30;
        horizon_Nc = 15;

    else % Final docking
        Q_phase(1:3,1:3) = Q_base(1:3,1:3) * 100.0;
        Q_phase(4:6,4:6) = Q_base(4:6,4:6) * 20.0;
        R_phase = R_base * 0.5;
        u_constraints = u_max * 0.3;  % Limit thrust for efficiency
        horizon_Np = 20;
        horizon_Nc = 10;
    end
end

function [safe, safety_control] = check_safety(x, v_max, u_max_scalar)
    pos = x(1:3);
    vel = x(4:6);
    pos_norm = norm(pos);
    speed = norm(vel);
    safety_control = zeros(2,1);

    % Define dynamic safety velocity based on distance
    if pos_norm > 1000
        v_safe = 3.5;  % Allow higher velocity when far
    else
        v_safe = max(0.1, min(2.0, pos_norm / 1000));  % Less aggressive
    end

    % Check if velocity exceeds the dynamically computed safe velocity
    if speed > v_safe
        safe = false;  % Override only when absolutely necessary
        decel_mag = min(u_max_scalar, (speed - v_safe) * 2.0);
        safety_control(1:2) = -vel(1:2) / max(norm(vel(1:2)), 1e-6) *
decel_mag;
        return;  % Exit function immediately if unsafe
    end

    % Progressive velocity profile based on distance
    if pos_norm > 2000
        v_safe = 2.5 * v_max;
    elseif pos_norm > 1000
        v_safe = 1.5 * v_max;
    elseif pos_norm > 500
        v_safe = 1.0 * v_max;
    elseif pos_norm > 100
        v_safe = 0.5 * v_max;
    elseif pos_norm > 10
        v_safe = 0.2 * v_max;
    else
        v_safe = 0.05 * v_max;
    end

    % Check approach angle for closing velocity
    pos_unit = pos / max(pos_norm, 1e-10);
    vel_proj = dot(vel, pos_unit);  % Projected velocity toward target

    % Allow positive velocity when far from the target
    if pos_norm > 500
        angle_safe = true;
    else
        angle_safe = (vel_proj < 0) || (pos_norm < 5 && abs(vel_proj) <
0.01);
    end

    % Combined safety check
    magnitude_safe = speed <= v_safe;
    safe = magnitude_safe && angle_safe;

    % Calculate safety control if needed
    if ~safe
        % Direction for deceleration
        if ~magnitude_safe
            decel_dir = -vel(1:2) / max(norm(vel(1:2)), 1e-10);
            decel_mag = min(u_max_scalar, (speed - v_safe) * 8.0);
        else  % Not angle_safe
            decel_dir = pos_unit(1:2);
            decel_mag = min(u_max_scalar, abs(vel_proj) * 1.0);
        end

        safety_control = decel_dir * decel_mag;
    end
end
```

```
function sigma_points_pred = predict_sigma_points(sigma_points, u, Ts)
    % System dimensions
    n = size(sigma_points, 1); % n = 6 (state dimension)
    m = size(u, 1); % m = 2 (control dimension)

    % Propagate sigma points through the system dynamics
    sigma_points_pred = A * sigma_points + B * u;
end

function z_pred_points = predict_measurements(sigma_points_pred)
    H = eye(6); % Identity matrix
    % Predict measurements
    z_pred_points = H * sigma_points_pred;
end

function [x_est, P_est] = ukf_update(x_prev, P_prev, u, z, Q, R, alpha,
beta, kappa, Ts, A, B)
    % State dimension
    n = length(x_prev);

    % Calculate UKF parameters
    lambda = alpha^2 * (n + kappa) - n;
    gamma = sqrt(n + lambda);

    % Weights calculation
    Wm = zeros(2*n+1, 1);
    Wc = zeros(2*n+1, 1);
    Wm(1) = lambda / (n + lambda);
    Wc(1) = lambda / (n + lambda) + (1 - alpha^2 + beta);
    for i = 2:2*n+1
        Wm(i) = 1 / (2*(n + lambda));
        Wc(i) = 1 / (2*(n + lambda));
    end

    % Ensure P_prev is positive definite
    P_prev = (P_prev + P_prev') / 2;  % Ensure symmetry
    P_prev = P_prev + 1e-6 * eye(n);  % Add small regularization

    % Generate sigma points
    sigma_points = zeros(n, 2*n+1);
    sigma_points(:,1) = x_prev;

    % Calculate square root of P using Cholesky decomposition
    sqrt_P = chol((n + lambda) * P_prev, 'lower');

    for i = 1:n
        sigma_points(:,i+1) = x_prev + sqrt_P(:,i);
        sigma_points(:,i+1+n) = x_prev - sqrt_P(:,i);
    end

    % Prediction step
    sigma_points_pred = zeros(n, 2*n+1);
    for i = 1:2*n+1
        % Propagate each sigma point through the dynamics model
        sigma_points_pred(:,i) = A * sigma_points(:,i) + B * u;
    end

    % Calculate predicted mean
    x_pred = zeros(n, 1);
    for i = 1:2*n+1
        x_pred = x_pred + Wm(i) * sigma_points_pred(:,i);
    end

    % Calculate predicted covariance
    P_pred = Q;  % Start with process noise
    for i = 1:2*n+1
        diff = sigma_points_pred(:,i) - x_pred;
        P_pred = P_pred + Wc(i) * (diff * diff');
    end

    % Update step with measurements
    H = eye(n);
    z_pred = zeros(length(z), 2*n+1);
    for i = 1:2*n+1
        z_pred(:,i) = H * sigma_points_pred(:,i);  % Apply measurement
model
    end

    % Predicted measurement
    z_mean = zeros(length(z), 1);
    for i = 1:2*n+1
        z_mean = z_mean + Wm(i) * z_pred(:,i);
    end

    % Innovation covariance
    S = R;  % Start with measurement noise
    for i = 1:2*n+1
        diff = z_pred(:,i) - z_mean;
        S = S + Wc(i) * (diff * diff');
    end

    % Cross correlation matrix
    Pxz = zeros(n, length(z));
    for i = 1:2*n+1
        diff_x = sigma_points_pred(:,i) - x_pred;
        diff_z = z_pred(:,i) - z_mean;
        Pxz = Pxz + Wc(i) * (diff_x * diff_z');
    end

    % Kalman gain (using pseudoinverse for stability)
    K = Pxz * pinv(S);

    % State and covariance update (Joseph form for stability)
    I = eye(n);
    P_est = (I - K * H) * P_pred * (I - K * H)' + K * R * K';
    x_est = x_pred + K * (z - z_mean);
end

% Performance Metrics
% Calculate Total Maneuver Time
total_maneuver_time = t(end);

% Calculate Final Position and Velocity Error
final_position_error = norm(x(1:3, end));
final_velocity_error = norm(x(4:6, end));

% Calculate Total Energy Consumed
total_energy_consumed = energy(end);
```

```
% Calculate Maximum Thrust Magnitude
max_thrust_magnitude = max(thrust_magnitude);

% Calculate Total Thruster On Time
total_thruster_on_time = sum(thruster_status) * Ts;
thruster_on_percentage = (total_thruster_on_time / total_maneuver_time) *
100;

% Display Performance Metrics
disp(' ');
disp('Performance Metrics');
disp(['Total Maneuver Time: ', num2str(total_maneuver_time), ' seconds']);
disp(['Final Position Error: ', num2str(final_position_error), ' m']);
disp(['Final Velocity Error: ', num2str(final_velocity_error), ' m/s']);
disp(['Total Energy Consumed: ', num2str(total_energy_consumed), ' J']);
disp(['Maximum Thrust Magnitude: ', num2str(max_thrust_magnitude), ' N']);
disp(['Total Thruster On Time: ', num2str(total_thruster_on_time), '
seconds (', num2str(thruster_on_percentage), '%)']);

% Plotting
% Figure 1: SMC Mode Docking Simulation
figure('Name', 'R-bar Mode Docking Simulation');
subplot(2,2,1);
plot(x(1,1:end), x(2,1:end), 'b-', 'LineWidth', 2);
hold on;
plot(0, 0, 'r*', 'MarkerSize', 10);
plot(x(1,1), x(2,1), 'go', 'MarkerSize', 10);
grid on;
xlabel('V-bar (m)');
ylabel('R-bar (m)');
title('Approach Trajectory');
legend('Trajectory', 'Target', 'Start', 'Location', 'best');

subplot(2,2,2);
plot(t, sqrt(sum(x(1:3,:).^2, 1)), 'b-', 'LineWidth', 2);
grid on;
xlabel('Time (s)');
ylabel('Distance (m)');
title('Distance to Target');

subplot(2,2,3);
plot(t, x(4,:), 'r-', 'LineWidth', 2, 'DisplayName', 'V_x');
hold on;
plot(t, x(5,:), 'b-', 'LineWidth', 2, 'DisplayName', 'V_y');
plot(t, sqrt(sum(x(4:5,:).^2, 1)), 'k--', 'LineWidth', 1, 'DisplayName',
'V_{mag}');
grid on;
xlabel('Time (s)');
ylabel('Velocity (m/s)');
title('Velocity Components');
legend('Location', 'best');

subplot(2,2,4);
stairs(t(1:end-1), u(1,:), 'r-', 'LineWidth', 2, 'DisplayName', 'u_x');
hold on;
stairs(t(1:end-1), u(2,:), 'b-', 'LineWidth', 2, 'DisplayName', 'u_y');
stairs(t(1:end-1), sqrt(sum(u.^2, 1)), 'k--', 'LineWidth', 1,
'DisplayName', 'u_{mag}');
```

```
grid on;
xlabel('Time (s)');
ylabel('Control Input (m/s^2)');
title('Control Inputs');
legend('Location', 'best');

% Figure 2: Thrust Comparison
figure('Name', 'Thrust Comparison');
subplot(2,1,1);
stairs(t(1:end-1), u_raw(1,:), 'r--', 'LineWidth', 1, 'DisplayName', 'Raw
u_x');
hold on;
stairs(t(1:end-1), u_raw(2,:), 'b--', 'LineWidth', 1, 'DisplayName', 'Raw
u_y');
stairs(t(1:end-1), u(1,:), 'r-', 'LineWidth', 2, 'DisplayName', 'PWM u_x');
stairs(t(1:end-1), u(2,:), 'b-', 'LineWidth', 2, 'DisplayName', 'PWM u_y');
grid on;
xlabel('Time (s)');
ylabel('Control Input (m/s^2)');
title('Raw vs PWM Control Inputs');
legend('Location', 'best');

subplot(2,1,2);
plot(t(1:end-1), thrust_magnitude, 'g-', 'LineWidth', 2);
grid on;
xlabel('Time (s)');
ylabel('Thrust Magnitude (N)');
title('Thrust Magnitude');

% Figure 3: Thruster Activity
figure('Name', 'Thruster Activity');
subplot(2,1,1);
stem(t(1:end-1), thruster_status, 'k-', 'LineWidth', 1, 'Marker', 'none');
grid on;
xlabel('Time (s)');
ylabel('Thruster Status');
title('Thruster On/Off Status');
ylim([-0.1 1.1]);

subplot(2,1,2);
cumulative_on_time = cumsum(thruster_status) * Ts;
plot(t(1:end-1), cumulative_on_time, 'm-', 'LineWidth', 2);
grid on;
xlabel('Time (s)');
ylabel('Cumulative On Time (s)');
title('Cumulative Thruster On Time');

% Figure 4: Energy & Approach Analysis
figure('Name', 'Energy & Approach Analysis');
subplot(2,1,1);
plot(t, energy, 'm-', 'LineWidth', 2);
grid on;
xlabel('Time (s)');
ylabel('Energy Consumption (J)');
title('Energy Consumption Over Time');

subplot(2,1,2);
plot(sqrt(sum(x(1:3,:).^2, 1)), sqrt(sum(x(4:6,:).^2, 1)), 'c-',
'LineWidth', 2);
```

```
grid on;
xlabel('Distance to Target (m)');
ylabel('Approach Speed (m/s)');
title('Approach Speed vs Distance');

% Figure 5: Trajectory and Thruster Firing Points
figure('Name', 'Trajectory and Thruster Firing Points');
plot(x(1,1:end), x(2,1:end), 'b-', 'LineWidth', 2);  % Chaser trajectory
hold on;
plot(0, 0, 'r*', 'MarkerSize', 10);  % Target position
plot(x(1,1), x(2,1), 'go', 'MarkerSize', 10);  % Chaser starting position

% Plot thruster firing points
if ~isempty(thruster_firing_points)
    plot(thruster_firing_points(1,:), thruster_firing_points(2,:), 'rx', 
'MarkerSize', 3);
end

grid on;
xlabel('V-bar (m)');
ylabel('R-bar (m)');
title('Trajectory and Thruster Firing Points');
legend('Chaser Trajectory', 'Target', 'Chaser Start', 'Thruster Firing', 
'Location', 'best');
```

```
% R-bar Mode with Nonlinear Model Predictive Control (NMPC) and UKF
clc;
clear;
close all;

% Orbital parameters
alt = 450e3;            % Altitude of orbit (m)
Re = 6371e3;            % Earth radius (m)
mu = 3.986004418e14;    % Earth gravitational parameter (m^3/s^2)
r_orbit = Re + alt;     % Orbital radius (m)
n = sqrt(mu/r_orbit^3); % Mean motion (rad/s)

% Satellite parameters
m_target = 150;         % Target satellite mass (kg)
m_chaser = 350;         % Chaser satellite mass (kg)
thrust_x = 17;          % Total thrust (N)
num_thrusters = 4;      % Number of thrusters

% NMPC parameters
global Np Nc Ts
Ts = 0.5;               % Sampling time (s)
Np = 20;                % Prediction horizon (reduced from 50)
Nc = 20;                % Control horizon
Q_base = diag([1e6, 1e6, 1e6, 5e6, 5e6, 5e6]);  % Position weights
R_base = diag([10.0, 10.0]);  % Control weights
% S = 1e19 * eye(6);      % Terminal state weight
Q_terminal = diag([1e6, 1e6, 1e6, 1e5, 1e5, 1e5]); % Position: 1e6,
Velocity: 1e5 % Terminal cost weights
S = Q_terminal; % Use Q_terminal for terminal cost
use_warm_start = true;  % Co-state reuse toggle

% Initialize Chebyshev filter parameters
filter_order = 4;
ripple_dB = 0.5;       % dB
cutoff_freq = 0.08;    % Hz
sample_freq = 1/Ts;

% Design the Chebyshev filter
[b_cheby, a_cheby] = cheby1(filter_order, ripple_dB,
cutoff_freq/(sample_freq/2));
```

```
% Initialize filter states
filter_states_x = zeros(filter_order, 1);
filter_states_y = zeros(filter_order, 1);

% Lyapunov function weights
P = diag([1e18, 1e18, 1e18, 1e13, 1e13, 1e13]);
S_lyapunov = P;

% State and input constraints
u_max_scalar = ((thrust_x * num_thrusters) / m_chaser) * 0.6;  % Thrust
limit
v_max = 2;              % Maximum velocity (m/s)
x_min = [-inf; -inf; -inf; -v_max; -v_max; -v_max];  % State constraints
x_max = [inf; inf; inf; v_max; v_max; v_max];
u_min = [-u_max_scalar; -u_max_scalar];     % Control input constraints
u_max = [u_max_scalar; u_max_scalar];

% Dead-band control parameters
dead_band_far = 0.01;       % Dead-band when far from target
dead_band_mid = 0.005;      % Dead-band for mid-range
dead_band_near = 0.002;     % Dead-band when close to target

% Pulse-width modulation parameters
pwm_period = 4;             % PWM period
pwm_min_duty = 0.5;         % Minimum duty cycle

% Coast phase parameters
coast_distance_threshold = 10; % Distance threshold for coasting (m)
coast_velocity_threshold = 0.1; % Velocity threshold for coasting (m/s)
coast_phase_active = false; % Flag to track if coast phase is active

% Initial conditions
x0 = [-2903.1; -2991.5; 0; 0; 0; 0];  % 2.9031 km behind, 2.9915 km below
t_final = 1000;         % Simulation time (s)
t = 0:Ts:t_final;
N = length(t);

% Initialize state and control vectors
x = zeros(6, N);
u = zeros(2, N-1);
u_raw = zeros(2, N-1);
x(:,1) = x0;

% Initialize additional variables
control_history = zeros(2, N-1);
safety_status = false(1, N-1);
position_history = zeros(3, N);
velocity_history = zeros(3, N);
thruster_firing_points = [];

% Calculate initial distance and velocity
initial_distance = norm(x0(1:3));
initial_velocity = norm(x0(4:6));

disp(['Initial Distance: ', num2str(initial_distance), ' m, Initial
Velocity: ', num2str(initial_velocity), ' m/s']);

% UKF parameters
nx = 6;                 % State dimension
```

```
nu = 2;                 % Control dimension
alpha = 1e-3;           % Scaling parameter
beta = 2;               % Optimal for Gaussian distributions
kappa = 0;              % Secondary scaling parameter

% State and measurement noise covariance
Q = diag([1e-5, 1e-5, 1e-5, 1e-5, 1e-5, 1e-5]); % Process noise covariance
R = diag([0.01, 0.01, 0.01, 0.01, 0.01, 0.01]); % Measurement noise
covariance
L = length(x0);         % State dimension
lambda = alpha^2 * (L + kappa) - L; % Sigma point scaling factor
gamma = sqrt(L + lambda); % Square root of scaling factor

% UKF state and covariance initialization
x_ukf = x0;             % Initial state estimate
P_ukf = eye(L);         % Initial estimate covariance

% Initialize energy consumption tracking
energy = zeros(1, N);

% Initialize thrust magnitude and thruster status
thrust_magnitude = zeros(1, N-1);
thruster_status = zeros(1, N-1);

% Improved threshold for thrust activation
thrust_threshold = 0.01;  % Thrust threshold

% PWM counter initialization
pwm_counter = 1;

% Control mode history
control_mode_history = cell(1, N-1);

% Pontryagin's Minimum Principle optimization function
function u_pmp = pmp_optimize(x_state, A, B, Q, R, S, Np, u_min, u_max)
    % State and control dimensions
    nx = size(A, 1);
    nu = size(B, 2);

    % Terminal cost for final state
    lambda_T = S * x_state + 0.1 * [zeros(3, 1); x_state(4:6)];

    % Initialize arrays
    lambda = zeros(nx, Np + 1);
    lambda(:, Np + 1) = lambda_T;

    % Backward integration of co-state equation
    for i = Np:-1:1
        lambda(:, i) = A' * lambda(:, i + 1) + Q * x_state;
    end

    % Forward computation of optimal control
    x_traj = zeros(nx, Np + 1);
    u_seq = zeros(nu, Np);
    x_traj(:, 1) = x_state;

    % Compute optimal control sequence
    for i = 1:Np
        % Minimize Hamiltonian: u* = -R^(-1) * B' * lambda
        u_i = -inv(R) * B' * lambda(:, i + 1);

        % Apply control constraints
        u_i = min(max(u_i, u_min), u_max);

        % Update state using optimal control
        dist_effect = zeros(nx, 1);
        x_traj(:, i + 1) = A * x_traj(:, i) + B * u_i + dist_effect;
        u_seq(:, i) = u_i;
    end

    % Return first control action from sequence
    u_pmp = u_seq(:, 1);
end

% PMP-NMPC Controller Function
function [u_opt, lambda_prev] = PMP_NMPC_Controller(x0, x_ref, lambda_prev)
    persistent lambda_hist
    if isempty(lambda_hist) || ~use_warm_start
        lambda_hist = repmat(0.1*ones(size(x0)), 1, Np);
    end

    options = optimoptions('fmincon', 'Algorithm', 'sqp', ...
                          'SpecifyObjectiveGradient', true, ...
                          'MaxIterations', 100, 'Display', 'none');

    % PMP Optimization Loop
    for k = 1:Np
        % Warm-start co-states
        lambda = lambda_hist(:,k);

        % Hamiltonian minimization with constraints
        [u_opt, ~, exitflag] = fmincon(@(u) Hamiltonian(u, x0, lambda,
Q_terminal), ...
                                         u_guess, [], [], [], [], lb, ub, ...
                                         @(u) path_constraints(u, x0),
options);

        % RK4 State+Co-state integration
        [x_next, lambda_next] = rk4_integration(x0, u_opt, lambda);

        % Update co-state history
        lambda_hist(:,k) = lambda_next;
        x0 = x_next;
    end
end

% Dynamics Integration (RK4 Method)
function [x_next, lambda_next] = rk4_integration(x, u, lambda)
    dt = 0.1; % Tune based on system timescale

    % State integration
    k1_x = state_dynamics(x, u);
    k2_x = state_dynamics(x + 0.5*dt*k1_x, u);
    k3_x = state_dynamics(x + 0.5*dt*k2_x, u);
    k4_x = state_dynamics(x + dt*k3_x, u);
    x_next = x + (dt/6)*(k1_x + 2*k2_x + 2*k3_x + k4_x);

    % Co-state integration
    k1_l = costate_dynamics(x, lambda);
    k2_l = costate_dynamics(x + 0.5*dt*k1_x, lambda + 0.5*dt*k1_l);
    k3_l = costate_dynamics(x + 0.5*dt*k2_x, lambda + 0.5*dt*k2_l);
    k4_l = costate_dynamics(x + dt*k3_x, lambda + dt*k3_l);
    lambda_next = lambda + (dt/6)*(k1_l + 2*k2_l + 2*k3_l + k4_l);
end

% Hamiltonian Function (Critical Modification)
function [H, gradH] = Hamiltonian(u, x, lambda, Q_terminal)
    % State weighting matrix
    Q = diag([10, 10, 10, 1, 1, 1]);
    R = 0.1*eye(3); % Control effort weighting

    % Terminal cost gradient
    psi_grad = Q_terminal*(x - x_ref);

    % Hamiltonian components
    H = lambda' * state_dynamics(x, u) + 0.5 * (x' * Q * x + u' * R * u) +
psi_grad;

    % Analytical gradient (required for SQP)
    if nargout > 1
        gradH = R * u + [lambda(4:6); 0; 0; 0]; % Adjust based on actual
dynamics
    end
end

% Path Constraints Function (Example Implementation)
function [c, ceq] = path_constraints(u, x)
    % Define path constraints (inequality and equality)
    c = []; % Inequality constraints
    ceq = []; % Equality constraints
end

% Main Simulation Loop with NMPC and PMP
for k = 1:N-1
    % Get current state
    x_current = x(:,k);
    pos_norm = norm(x_current(1:3));
    vel_norm = norm(x_current(4:6));

    % Determine current mission phase based on distance
    if pos_norm > 2000
        phase = 1; % Far approach phase
        phase_name = 'Far Approach';
        control_mode = 'Far Approach (PMP-NMPC Blend)';
    elseif pos_norm > 500  % Mid-range phase only when distance > 500 m
        phase = 2; % Mid-range phase
        phase_name = 'Mid-Range Approach';
        control_mode = 'Mid-Range (PMP-NMPC Blend)';  % Update control mode
    elseif pos_norm > 5  % Close approach phase
        phase = 3; % Close approach phase
        phase_name = 'Close Approach';
        control_mode = 'Close Approach (NMPC-Lyapunov Blend)';
    else
        phase = 4; % Final docking phase
        phase_name = 'Final Docking';
        control_mode = 'Final Docking (Terminal Guidance)';
    end
    % State transition matrix (A) for orbital dynamics
    A = [1 0 0 Ts 0 0;
         0 1 0 0 Ts 0;
         0 0 1 0 0 Ts;
         0 0 0 1 0 0;
         0 0 0 0 1 0;
         0 0 0 0 0 1];

    % Input matrix (B) for control inputs
    B = [0 0;
         0 0;
         0 0;
         1 0;
         0 1;
         0 0];

    % Compute control strategies based on phase
    [Q_phase, R_phase, u_constraints, horizon_Np, horizon_Nc] =
phase_specific_control(phase, pos_norm, vel_norm, Q_base, R_base, Np, Nc,
u_max);

    % Solve PMP problem
    u_pmp = pmp_optimize(x_current, A, B, Q_phase, R_phase, S, horizon_Np,
u_min, u_max);

    % Use PMP control as reference for NMPC
    U0 = [repmat(u_pmp, horizon_Np, 1); zeros(2 * (Np - horizon_Np), 1)]; %
Initial guess for control inputs

    % Ensure that lb and ub have the same length as U0
    lb = repmat(u_min, horizon_Np, 1); % Lower bounds for control inputs
    ub = repmat(u_max, horizon_Np, 1); % Upper bounds for control inputs

    % Adjust the length of lb and ub if they are shorter than U0
    if length(lb) < length(U0)
        lb = [lb; -Inf * ones(length(U0) - length(lb), 1)];
    end
    if length(ub) < length(U0)
        ub = [ub; Inf * ones(length(U0) - length(ub), 1)];
    end

    options = optimoptions('fmincon', 'Display', 'none', 'Algorithm',
'sqp');

    [U_opt, ~] = fmincon(@(U) nmpc_cost(U, x_current, Np, Q_phase, R_phase,
S, A, B), U0, [], [], [], [], lb, ub, [], options);

    % Apply first control input
    u(:,k) = U_opt(1:2);

    % Update state
    k1 = Ts * state_dynamics(x_current, u(:,k));
    k2 = Ts * state_dynamics(x_current + 0.5*k1, u(:,k));
    k3 = Ts * state_dynamics(x_current + 0.5*k2, u(:,k));
    k4 = Ts * state_dynamics(x_current + k3, u(:,k));
    x(:,k+1) = x_current + (k1 + 2*k2 + 2*k3 + k4)/6;

    % Initialize u_opt for filtering
    u_opt = U_opt(1:2);

    % Apply Chebyshev filter for control signal smoothing
    if k > 5
        % Get control history for filtering
        u_history = u_raw(:, max(1, k-5):k-1);

        % Chebyshev filter parameters based on phase
        switch phase
            case 1 % Far approach
                filter_order = 2;
                ripple = 1.0;
            case 2 % Mid-range
                filter_order = 3;
                ripple = 0.8;
            case 3 % Close approach
                filter_order = 4;
                ripple = 0.5;
            case 4 % Final docking
                filter_order = 5;
                ripple = 0.3;
        end

        % Apply Chebyshev filter
        [b, a] = cheby1(filter_order, ripple, 0.1, 'low');
        u_filtered = zeros(size(u_opt));
        for i = 1:length(u_opt)
            u_seq = [reshape(u_history(i,:), [], 1); u_opt(i)];
            u_filt = filter(b, a, u_seq);
            u_filtered(i) = u_filt(end);
        end

        % Blend filtered and raw control based on phase
        switch phase
            case 1
                alpha_blend = 0.7;
            case 2
                alpha_blend = 0.5;
            case 3
                alpha_blend = 0.3;
            case 4
                alpha_blend = 0.1;
        end

        u_raw(:,k) = alpha_blend * u_filtered + (1 - alpha_blend) * u_opt;
    else
        % Smooth control signal using Chebyshev filter
        [u_raw(:,k), filter_states_x, filter_states_y] =
apply_chebyshev_filter(u_opt, filter_states_x, filter_states_y, b_cheby,
a_cheby);
    end

    % Store control history
    control_history(:, k) = u_raw(:, k);

    % Apply dead-band and PWM for thruster control
    u_modified = enhanced_thruster_control(u_raw(:, k), pos_norm, vel_norm,
pwm_counter, phase, u_max_scalar);
    u(:,k) = u_modified;
    % Update PWM counter
    pwm_counter = pwm_counter + 1;
    if pwm_counter > pwm_period
        pwm_counter = 1;
    end

    % Calculate thrust magnitude and energy
    thrust_mag = norm(u(:,k)) * m_chaser;
    thrust_magnitude(k) = thrust_mag;
    thruster_status(k) = (thrust_mag > thrust_threshold);

    % Update energy consumption with improved efficiency calculation
    if k > 1
        if thruster_status(k)
            efficiency_factor = 0.8 + 0.2 * (1 - thrust_mag / (u_max_scalar
* m_chaser));
            energy(k+1) = energy(k) + (thrust_mag * norm(x_current(4:6)) *
Ts) / efficiency_factor;
        else
            energy(k+1) = energy(k);
        end
    end

    % Record thruster firing points for analysis
    if thruster_status(k)
        thruster_firing_points = [thruster_firing_points, x_current(1:2)];
    end

    % Store control mode
    control_mode_history{k} = control_mode;

    % Display progress every 100 steps
    if mod(k, 100) == 0
        disp([ ...
            'Time: ', num2str(t(k)), ' s, Phase: ', phase_name, ...
            ', Distance: ', num2str(pos_norm), ' m, Velocity: ',
num2str(vel_norm), ' m/s, Mode: ', control_mode]);
    end

    % UKF Update Step with phase-specific measurement noise
    switch phase
        case 1 % Far approach
            R_phase_ukf = R * 2.0; % Higher measurement uncertainty when
far
        case 2 % Mid-range
            R_phase_ukf = R * 1.5;
        case 3 % Close approach
            R_phase_ukf = R * 1.0;
        case 4 % Final docking
            R_phase_ukf = R * 0.1; % Lower uncertainty for precision
docking (Reduced measurement noise)
    end

    % Simulate measurements with phase-specific noise
    z = x_current + mvnrnd(zeros(L, 1), R_phase_ukf)';

    % UKF update
    [x_ukf, P_ukf] = ukf_update(x_ukf, P_ukf, u(:,k), z, Q, R_phase_ukf,
alpha, beta, kappa, Ts, A, B);

    % Store state values
    x(:, k+1) = x_ukf;
    position_history(:, k+1) = x_ukf(1:3);
    velocity_history(:, k+1) = x_ukf(4:6);

    % Check if docking is complete with tighter tolerances for ideal
performance
    if pos_norm < 0.01 && vel_norm < 0.001
        disp(['Docking successfully completed at time: ', num2str(t(k+1)),
' seconds!']);
        disp(['Final position error: ', num2str(pos_norm), ' m']);
        disp(['Final velocity error: ', num2str(vel_norm), ' m/s']);
        u(:,k) = [0; 0];
        u_raw(:,k) = [0; 0];
        break;
    end

    % Coast phase logic
    if pos_norm < coast_distance_threshold && vel_norm <
coast_velocity_threshold
        u(:, k) = [0; 0];
        u_raw(:, k) = [0; 0];
        control_mode = 'Coast Phase';
        coast_phase_active = true;
    end
end

% Define state dynamics function
function dx = state_dynamics(x, u)
    global Ts
    A = [1 0 0 Ts 0 0;
         0 1 0 0 Ts 0;
         0 0 1 0 0 Ts;
         0 0 0 1 0 0;
         0 0 0 0 1 0;
         0 0 0 0 0 1];
    B = [0 0; 0 0; 0 0; 1 0; 0 1; 0 0];
    dx = A * x + B * u;
end

% Define co-state dynamics function
function dlambda = costate_dynamics(x, lambda)
    global Ts
    A = [1 0 0 Ts 0 0;
         0 1 0 0 Ts 0;
         0 0 1 0 0 Ts;
         0 0 0 1 0 0;
         0 0 0 0 1 0;
         0 0 0 0 0 1];
    dlambda = -A' * lambda;
end

% NMPC cost function
function J = nmpc_cost(U, x0, Np, Q, R, S, A, B)
    x = x0;
    J = 0;
    for k = 1:Np
        u = U((k-1)*2+1:k*2);
        x = A * x + B * u;
        J = J + x' * Q * x + u' * R * u;
    end
    J = J + x' * S * x; % Terminal cost
end

% Chebyshev Filter Functions
function [b, a] = design_chebyshev_filter(order, ripple, cutoff_freq,
sampling_freq)
    % Design lowpass Chebyshev Type I filter
    normalized_cutoff = cutoff_freq / (sampling_freq / 2);
    [b, a] = cheby1(order, ripple, normalized_cutoff, 'low');
end

function [u_filtered, filter_states_x, filter_states_y] =
apply_chebyshev_filter(u_raw, filter_states_x, filter_states_y, b, a)
    % Apply filter to each control dimension
    [u_filtered(1), filter_states_x] = filter(b, a, u_raw(1),
filter_states_x);
    [u_filtered(2), filter_states_y] = filter(b, a, u_raw(2),
filter_states_y);
end

% Thruster control with adjusted dead-band and PWM parameter
function u_modified = enhanced_thruster_control(u_raw, pos_norm, vel_norm,
pwm_counter, phase, u_max_scalar)
    u_modified = u_raw;

    % Phase-specific parameters
    if phase == 1  % Far approach
        dead_band = 0.05 * u_max_scalar;
        pwm_period = 10;
        pwm_min_duty = 0.2;
    elseif phase == 2  % Mid-range
        dead_band = 0.03 * u_max_scalar;
        pwm_period = 8;
        pwm_min_duty = 0.25;
    elseif phase == 3  % Close approach
        dead_band = 0.01 * u_max_scalar;
        pwm_period = 6;
        pwm_min_duty = 0.3;
    else  % Final docking
        dead_band = 0.001 * u_max_scalar; % Tighter dead-band
        pwm_period = 2; % Shorter period for smoother control
        pwm_min_duty = 0.4; % Higher minimum duty cycle
        min_impulse = 0.005 * u_max_scalar; % Smaller impulse bit
    end

    % Incorporate velocity for adaptive dead-band
    velocity_factor = min(1.0, vel_norm / max(0.1, pos_norm/100));
    adjusted_dead_band = dead_band * (1 + velocity_factor);

    % Apply dynamic dead-band
    for i = 1:length(u_modified)
        if abs(u_modified(i)) < adjusted_dead_band
            u_modified(i) = 0;
        end
    end

    % Ensure pwm_counter works with the current period
    pwm_counter_adjusted = mod(pwm_counter-1, pwm_period) + 1;

    % Enhanced PWM implementation
    for i = 1:length(u_modified)
        mag = abs(u_modified(i));

        if mag > 0 && mag < 0.5 * u_max_scalar
            % Progressive duty cycle based on magnitude
            duty_cycle = pwm_min_duty + (1-pwm_min_duty) *
(mag/(0.5*u_max_scalar));

            if pwm_counter_adjusted <= round(pwm_period * duty_cycle)
                u_modified(i) = sign(u_modified(i)) * min(u_max_scalar, mag
/ duty_cycle);
            else
                u_modified(i) = 0;
            end
        end
    end

    % Minimum impulse bit control for final approach
    if phase == 4 && pos_norm < 5
        min_impulse = 0.02 * u_max_scalar;
        for i = 1:length(u_modified)
            if abs(u_modified(i)) > 0 && abs(u_modified(i)) < min_impulse
                u_modified(i) = sign(u_modified(i)) * min_impulse;
            end
        end
    end
end

% Phase-specific control adjustments
function [Q_phase, R_phase, u_constraints, horizon_Np, horizon_Nc] =
phase_specific_control(phase, pos_norm, vel_norm, Q_base, R_base, Np_base,
Nc_base, u_max)
    % Default initialization
    Q_phase = Q_base;
    R_phase = R_base;
    u_constraints = u_max;
    horizon_Np = Np_base;
    horizon_Nc = Nc_base;

    % Phase 1: Far approach (d > 1000m)
    if phase == 1
        Q_phase(1:3,1:3) = Q_base(1:3,1:3) * 0.5;  % Position weight
        Q_phase(4:6,4:6) = Q_base(4:6,4:6) * 0.5;  % Velocity weight
        R_phase = R_base * 5.0;  % Control penalty
        u_constraints = u_max * 0.7;  % Limit thrust for efficiency
        horizon_Np = 50;  % Prediction horizon
        horizon_Nc = 25;  % Control horizon

    % Phase 2: Mid-range approach (100m < d ≤ 1000m)
    elseif phase == 2
        Q_phase(1:3,1:3) = Q_base(1:3,1:3) * 0.8;
        Q_phase(4:6,4:6) = Q_base(4:6,4:6) * 0.8;
        R_phase = R_base * 4.0;
        u_constraints = u_max * 0.8;
        horizon_Np = 40;
        horizon_Nc = 20;

    % Phase 3: Close approach (10m < d ≤ 100m)
    elseif phase == 3
        Q_phase(1:3,1:3) = Q_base(1:3,1:3) * 10;
        Q_phase(4:6,4:6) = Q_base(4:6,4:6) * 5;
        R_phase = R_base * 1.0;
        u_constraints = u_max * 0.9;
        horizon_Np = 30;
        horizon_Nc = 15;

    % Phase 4: Final docking (d ≤ 10m)
    else
        Q_phase(1:3,1:3) = Q_base(1:3,1:3) * 500.0;  % Increased position
weight
        Q_phase(4:6,4:6) = Q_base(4:6,4:6) * 50.0;   % Increased velocity
weight
        R_phase = R_base * 0.1;  % Control penalty
        u_constraints = u_max * 1.0;  % Allow more control authority
        horizon_Np = 25;  % Adjusted horizons for precision
        horizon_Nc = 10;



    end

    % Velocity-specific adjustments
    if vel_norm > 0.5 * sqrt(pos_norm/100)
        Q_phase(4:6,4:6) = Q_phase(4:6,4:6) * 2.0;
    end

    % Final approach precision
    if phase == 4 && pos_norm < 1.0
        Q_phase(1:3,1:3) = Q_phase(1:3,1:3) * 3.0;
        Q_phase(4:6,4:6) = Q_phase(4:6,4:6) * 2.0;
    end
end

function [safe, safety_control] = check_safety(x, v_max, u_max_scalar)
    pos = x(1:3);
    vel = x(4:6);
    pos_norm = norm(pos);
    speed = norm(vel);
    safety_control = zeros(2,1);

    % Define dynamic safety velocity based on distance
    if pos_norm > 1000
        v_safe = 3.5;  % Allow higher velocity when far
    else
        v_safe = max(0.1, min(2.0, pos_norm / 1000));  % Less aggressive
    end

    % Check if velocity exceeds the dynamically computed safe velocity
    if speed > v_safe
        safe = false;  % Override only when absolutely necessary
        decel_mag = min(u_max_scalar, (speed - v_safe) * 2.0);
        safety_control(1:2) = -vel(1:2) / max(norm(vel(1:2)), 1e-6) *
decel_mag;
        return;  % Exit function immediately if unsafe
    end

    % Progressive velocity profile based on distance
    if pos_norm > 2000
        v_safe = 2.5 * v_max;
    elseif pos_norm > 1000
        v_safe = 1.5 * v_max;
    elseif pos_norm > 500
        v_safe = 1.0 * v_max;
    elseif pos_norm > 100
        v_safe = 0.5 * v_max;
    elseif pos_norm > 10
        v_safe = 0.2 * v_max;
    else
        v_safe = 0.05 * v_max;
    end

    % Check approach angle for closing velocity
    pos_unit = pos / max(pos_norm, 1e-10);
    vel_proj = dot(vel, pos_unit);  % Projected velocity toward target

    % Allow positive velocity when far from the target
    if pos_norm > 500
        angle_safe = true;
    else
        angle_safe = (vel_proj < 0) || (pos_norm < 5 && abs(vel_proj) <
0.01);
    end

    % Combined safety check
    magnitude_safe = speed <= v_safe;
    safe = magnitude_safe && angle_safe;

    % Calculate safety control if needed
    if ~safe
        % Direction for deceleration
        if ~magnitude_safe
            decel_dir = -vel(1:2) / max(norm(vel(1:2)), 1e-10);
            decel_mag = min(u_max_scalar, (speed - v_safe) * 8.0);
        else  % Not angle_safe
            decel_dir = pos_unit(1:2);
            decel_mag = min(u_max_scalar, abs(vel_proj) * 1.0);
        end

        safety_control = decel_dir * decel_mag;
    end
end

function sigma_points_pred = predict_sigma_points(sigma_points, u, Ts)
    % System dimensions
    n = size(sigma_points, 1); % n = 6 (state dimension)
    m = size(u, 1); % m = 2 (control dimension)

    % Propagate sigma points through the system dynamics
    sigma_points_pred = A * sigma_points + B * u;
end
```

```
function z_pred_points = predict_measurements(sigma_points_pred)
    H = eye(6); % Identity matrix
    % Predict measurements
    z_pred_points = H * sigma_points_pred;
end

function [x_est, P_est] = ukf_update(x_prev, P_prev, u, z, Q, R, alpha,
beta, kappa, Ts, A, B)
    % State dimension
    n = length(x_prev);

    % Calculate UKF parameters
    lambda = alpha^2 * (n + kappa) - n;
    gamma = sqrt(n + lambda);

    % Weights calculation
    Wm = zeros(2*n+1, 1);
    Wc = zeros(2*n+1, 1);
    Wm(1) = lambda / (n + lambda);
    Wc(1) = lambda / (n + lambda) + (1 - alpha^2 + beta);
    for i = 2:2*n+1
        Wm(i) = 1 / (2*(n + lambda));
        Wc(i) = 1 / (2*(n + lambda));
    end

    % Ensure P_prev is positive definite
    P_prev = (P_prev + P_prev') / 2;  % Ensure symmetry
    P_prev = P_prev + 1e-6 * eye(n);  % Add small regularization

    % Generate sigma points
    sigma_points = zeros(n, 2*n+1);
    sigma_points(:,1) = x_prev;

    % Calculate square root of P using Cholesky decomposition
    sqrt_P = chol((n + lambda) * P_prev, 'lower');

    for i = 1:n
        sigma_points(:,i+1) = x_prev + sqrt_P(:,i);
        sigma_points(:,i+1+n) = x_prev - sqrt_P(:,i);
    end

    % Prediction step
    sigma_points_pred = zeros(n, 2*n+1);
    for i = 1:2*n+1
        % Propagate each sigma point through the dynamics model
        sigma_points_pred(:,i) = A * sigma_points(:,i) + B * u;
    end

    % Calculate predicted mean
    x_pred = zeros(n, 1);
    for i = 1:2*n+1
        x_pred = x_pred + Wm(i) * sigma_points_pred(:,i);
    end

    % Calculate predicted covariance
    P_pred = Q;  % Start with process noise
    for i = 1:2*n+1
        diff = sigma_points_pred(:,i) - x_pred;
        P_pred = P_pred + Wc(i) * (diff * diff');
    end

    % Update step with measurements
    H = eye(n);
    z_pred = zeros(length(z), 2*n+1);
    for i = 1:2*n+1
        z_pred(:,i) = H * sigma_points_pred(:,i);  % Apply measurement
model
    end

    % Predicted measurement
    z_mean = zeros(length(z), 1);
    for i = 1:2*n+1
        z_mean = z_mean + Wm(i) * z_pred(:,i);
    end

    % Innovation covariance
    S = R;  % Start with measurement noise
    for i = 1:2*n+1
        diff = z_pred(:,i) - z_mean;
        S = S + Wc(i) * (diff * diff');
    end

    % Cross correlation matrix
    Pxz = zeros(n, length(z));
    for i = 1:2*n+1
        diff_x = sigma_points_pred(:,i) - x_pred;
        diff_z = z_pred(:,i) - z_mean;
        Pxz = Pxz + Wc(i) * (diff_x * diff_z');
    end

    % Kalman gain (using pseudoinverse for stability)
    K = Pxz * pinv(S);

    % State and covariance update (Joseph form for stability)
    I = eye(n);
    P_est = (I - K * H) * P_pred * (I - K * H)' + K * R * K';
    x_est = x_pred + K * (z - z_mean);
end

% Performance Metrics
% Calculate Total Maneuver Time
total_maneuver_time = t(k+1);

% Calculate Final Position and Velocity Error
final_position_error = norm(x(1:3, k+1));
final_velocity_error = norm(x(4:6, k+1));

% Calculate Total Energy Consumed
total_energy_consumed = energy(k+1);

% Calculate Maximum Thrust Magnitude
max_thrust_magnitude = max(thrust_magnitude(1:k));

% Calculate Total Thruster On Time
total_thruster_on_time = sum(thruster_status(1:k)) * Ts;
thruster_on_percentage = (total_thruster_on_time / total_maneuver_time) *
100;
```

```
% Display Performance Metrics
disp(' ');
disp('Performance Metrics');
disp(['Total Maneuver Time: ', num2str(total_maneuver_time), ' seconds']);
disp(['Final Position Error: ', num2str(final_position_error), ' m']);
disp(['Final Velocity Error: ', num2str(final_velocity_error), ' m/s']);
disp(['Total Energy Consumed: ', num2str(total_energy_consumed), ' J']);
disp(['Maximum Thrust Magnitude: ', num2str(max_thrust_magnitude), ' N']);
disp(['Total Thruster On Time: ', num2str(total_thruster_on_time), '
seconds (', num2str(thruster_on_percentage), '%)']);

% Plotting
% Figure 1: R-bar Mode Docking Simulation
figure('Name', 'NMPC Docking Simulation');
subplot(2,2,1);
plot(x(1,1:k+1), x(2,1:k+1), 'b-', 'LineWidth', 2);
hold on;
plot(0, 0, 'r*', 'MarkerSize', 10);
plot(x(1,1), x(2,1), 'go', 'MarkerSize', 10);
grid on;
xlabel('V-bar (m)');
ylabel('R-bar (m)');
title('Approach Trajectory');
legend('Trajectory', 'Target', 'Start', 'Location', 'best');

subplot(2,2,2);
plot(t(1:k+1), sqrt(sum(x(1:3,1:k+1).^2)), 'b-', 'LineWidth', 2);
grid on;
xlabel('Time (s)');
ylabel('Distance (m)');
title('Distance to Target');

subplot(2,2,3);
plot(t(1:k+1), x(4,1:k+1), 'r-', 'LineWidth', 2, 'DisplayName', 'V_x');
hold on;
plot(t(1:k+1), x(5,1:k+1), 'b-', 'LineWidth', 2, 'DisplayName', 'V_y');
plot(t(1:k+1), sqrt(x(4,1:k+1).^2 + x(5,1:k+1).^2), 'k--', 'LineWidth', 1,
'DisplayName', 'V_{mag}');
grid on;
xlabel('Time (s)');
ylabel('Velocity (m/s)');
title('Velocity Components');
legend('Location', 'best');

subplot(2,2,4);
stairs(t(1:k), u(1,1:k), 'r-', 'LineWidth', 2, 'DisplayName', 'u_x');
hold on;
stairs(t(1:k), u(2,1:k), 'b-', 'LineWidth', 2, 'DisplayName', 'u_y');
stairs(t(1:k), sqrt(u(1,1:k).^2 + u(2,1:k).^2), 'k--', 'LineWidth', 1,
'DisplayName', 'u_{mag}');
grid on;
xlabel('Time (s)');
ylabel('Control Input (m/s^2)');
title('Control Inputs');
legend('Location', 'best');

% Figure 2: Thrust Comparison
figure('Name', 'Thrust Comparison');
```

```
subplot(2,1,1);
stairs(t(1:k), u_raw(1,1:k), 'r--', 'LineWidth', 1, 'DisplayName', 'Raw
u_x');
hold on;
stairs(t(1:k), u_raw(2,1:k), 'b--', 'LineWidth', 1, 'DisplayName', 'Raw
u_y');
stairs(t(1:k), u(1,1:k), 'r-', 'LineWidth', 2, 'DisplayName', 'PWM u_x');
stairs(t(1:k), u(2,1:k), 'b-', 'LineWidth', 2, 'DisplayName', 'PWM u_y');
grid on;
xlabel('Time (s)');
ylabel('Control Input (m/s^2)');
title('Raw vs PWM Control Inputs');
legend('Location', 'best');

subplot(2,1,2);
plot(t(1:k), thrust_magnitude(1:k), 'g-', 'LineWidth', 2);
grid on;
xlabel('Time (s)');
ylabel('Thrust Magnitude (N)');
title('Thrust Magnitude');

% Figure 3: Thruster Activity
figure('Name', 'Thruster Activity');
subplot(2,1,1);
stem(t(1:k), thruster_status(1:k), 'k-', 'LineWidth', 1, 'Marker', 'none');
grid on;
xlabel('Time (s)');
ylabel('Thruster Status');
title('Thruster On/Off Status');
ylim([-0.1 1.1]);

subplot(2,1,2);
cumulative_on_time = cumsum(thruster_status(1:k)) * Ts;
plot(t(1:k), cumulative_on_time, 'm-', 'LineWidth', 2);
grid on;
xlabel('Time (s)');
ylabel('Cumulative On Time (s)');
title('Cumulative Thruster On Time');

% Figure 4: Energy & Approach Analysis
figure('Name', 'Energy & Approach Analysis');
subplot(2,1,1);
plot(t(1:k+1), energy(1:k+1), 'm-', 'LineWidth', 2);
grid on;
xlabel('Time (s)');
ylabel('Energy Consumption (J)');
title('Energy Consumption Over Time');

subplot(2,1,2);
plot(sqrt(sum(x(1:3,1:k+1).^2)), sqrt(sum(x(4:6,1:k+1).^2)), 'c-',
'LineWidth', 2);
grid on;
xlabel('Distance to Target (m)');
ylabel('Approach Speed (m/s)');
title('Approach Speed vs Distance');

% Figure 5: Trajectory and Thruster Firing Points
figure('Name', 'Trajectory and Thruster Firing Points');
plot(x(1,1:k+1), x(2,1:k+1), 'b-', 'LineWidth', 2);  % Chaser trajectory
```

```
hold on;
plot(0, 0, 'r*', 'MarkerSize', 10);  % Target position
plot(x(1,1), x(2,1), 'go', 'MarkerSize', 10);  % Chaser starting position

% Plot thruster firing points
if ~isempty(thruster_firing_points)
    plot(thruster_firing_points(1,:), thruster_firing_points(2,:), 'rx',
'MarkerSize', 3);
end

grid on;
xlabel('V-bar (m)');
ylabel('R-bar (m)');
title('Trajectory and Thruster Firing Points');
legend('Chaser Trajectory', 'Target', 'Chaser Start', 'Thruster Firing',
'Location', 'best');
```

```
% R-bar Mode with Model Predictive Control (MPC) and UKF
clc;
clear;
close all;

% Orbital parameters
alt = 450e3;            % Altitude of orbit (m)
Re = 6371e3;            % Earth radius (m)
mu = 3.986004418e14;    % Earth gravitational parameter (m^3/s^2)
r_orbit = Re + alt;     % Orbital radius (m)
n = sqrt(mu/r_orbit^3); % Mean motion (rad/s)

% Satellite parameters
m_target = 150;         % Target satellite mass (kg)
m_chaser = 350;         % Chaser satellite mass (kg)
thrust_x = 17;          % Total thrust (N)
num_thrusters = 4;      % Number of thrusters

% MPC parameters
Ts = 0.5;               % Sampling time (s)
```

```
global Np Nc
Np = 20;                % Prediction horizon
Nc = 20;                % Control horizon
Q_base = diag([1e6, 1e6, 1e6, 5e6, 5e6, 5e6]);  % Position weights
R_base = diag([10.0, 10.0]);  % Control weights
S = 1e19 * eye(6);      % Terminal state weight
Q_terminal = diag([1000, 1000, 1000, 100, 100, 100]); % Terminal cost
weights
use_warm_start = true;  % Co-state reuse toggle

% Initialize Chebyshev filter parameters
filter_order = 4;
ripple_dB = 0.5;       % dB
cutoff_freq = 0.08;    % Hz
sample_freq = 1/Ts;

% Design the Chebyshev filter
[b_cheby, a_cheby] = cheby1(filter_order, ripple_dB,
cutoff_freq/(sample_freq/2));

% Initialize filter states
filter_states_x = zeros(filter_order, 1);
filter_states_y = zeros(filter_order, 1);

% Lyapunov function weights
P = diag([1e18, 1e18, 1e18, 1e13, 1e13, 1e13]);
S_lyapunov = P;

% State and input constraints
u_max_scalar = ((thrust_x * num_thrusters) / m_chaser) * 0.35;  % Thrust
limit
v_max = 2;             % Maximum velocity (m/s)
x_min = [-inf; -inf; -inf; -v_max; -v_max; -v_max];  % State constraints
x_max = [inf; inf; inf; v_max; v_max; v_max];
u_min = [-u_max_scalar; -u_max_scalar];     % Control input constraints
u_max = [u_max_scalar; u_max_scalar];

% Dead-band control parameters
dead_band_far = 0.05;       % Dead-band when far from target
dead_band_mid = 0.05;      % Dead-band for mid-range
dead_band_near = 0.02;     % Dead-band when close to target

% Pulse-width modulation parameters
pwm_period = 10;             % PWM period
pwm_min_duty = 0.1;         % Minimum duty cycle

% Coast phase parameters
coast_distance_threshold = 800; % Distance threshold for coasting (m)
coast_velocity_threshold = 0.5; % Velocity threshold for coasting (m/s)
coast_phase_active = false; % Flag to track if coast phase is active

% Initial conditions
x0 = [-2903.1; -2991.5; 0; 0; 0; 0];  % 2.9031 km behind, 2.9915 km below
t_final = 1200;         % Simulation time (s)
t = 0:Ts:t_final;
N = length(t);

% Initialize state and control vectors
x = zeros(6, N);
```

```
u = zeros(2, N-1);
u_raw = zeros(2, N-1);
x(:,1) = x0;

% Initialize additional variables
control_history = zeros(2, N-1);
safety_status = false(1, N-1);
position_history = zeros(3, N);
velocity_history = zeros(3, N);
thruster_firing_points = [];

% Calculate initial distance and velocity
initial_distance = norm(x0(1:3));
initial_velocity = norm(x0(4:6));

disp(['Initial Distance: ', num2str(initial_distance), ' m, Initial
Velocity: ', num2str(initial_velocity), ' m/s']);

% UKF parameters
nx = 6;                 % State dimension
nu = 2;                 % Control dimension
alpha = 1e-3;           % Scaling parameter
beta = 2;               % Optimal for Gaussian distributions
kappa = 0;              % Secondary scaling parameter

% State and measurement noise covariance
Q = diag([1e-5, 1e-5, 1e-5, 1e-5, 1e-5, 1e-5]); % Process noise covariance
R = diag([0.01, 0.01, 0.01, 0.01, 0.01, 0.01]); % Measurement noise
covariance
L = length(x0);         % State dimension
lambda = alpha^2 * (L + kappa) - L; % Sigma point scaling factor
gamma = sqrt(L + lambda); % Square root of scaling factor

% UKF state and covariance initialization
x_ukf = x0;             % Initial state estimate
P_ukf = eye(L);         % Initial estimate covariance

% Initialize energy consumption tracking
energy = zeros(1, N);

% Initialize thrust magnitude and thruster status
thrust_magnitude = zeros(1, N-1);
thruster_status = zeros(1, N-1);

% Improved threshold for thrust activation
thrust_threshold = 0.01;  % Thrust threshold

% PWM counter initialization
pwm_counter = 1;

% Control mode history
control_mode_history = cell(1, N-1);

% Pontryagin's Minimum Principle optimization function
function u_pmp = pmp_optimize(x_state, A, B, Q, R, S, Np, u_min, u_max)
    % State and control dimensions
    nx = size(A, 1);
    nu = size(B, 2);
    % Terminal cost for final state
    lambda_T = S * x_state + 0.1 * [zeros(3, 1); x_state(4:6)];

    % Initialize arrays
    lambda = zeros(nx, Np + 1);
    lambda(:, Np + 1) = lambda_T;

    % Backward integration of co-state equation
    for i = Np:-1:1
        lambda(:, i) = A' * lambda(:, i + 1) + Q * x_state;
    end

    % Forward computation of optimal control
    x_traj = zeros(nx, Np + 1);
    u_seq = zeros(nu, Np);
    x_traj(:, 1) = x_state;

    % Compute optimal control sequence
    for i = 1:Np
        % Minimize Hamiltonian: u* = -R^(-1) * B' * lambda
        u_i = -inv(R) * B' * lambda(:, i + 1);

        % Apply control constraints
        u_i = min(max(u_i, u_min), u_max);

        % Update state using optimal control
        dist_effect = zeros(nx, 1);
        x_traj(:, i + 1) = A * x_traj(:, i) + B * u_i + dist_effect;
        u_seq(:, i) = u_i;
    end

    % Return first control action from sequence
    u_pmp = u_seq(:, 1);
end

% PMP-MPC Controller Function
function [u_opt, lambda_prev] = PMP_MPC_Controller(x0, x_ref, lambda_prev)
    persistent lambda_hist
    if isempty(lambda_hist) || ~use_warm_start
        lambda_hist = repmat(0.1*ones(size(x0)), 1, Np);
    end

    options = optimoptions('fmincon', 'Algorithm', 'sqp', ...
                          'SpecifyObjectiveGradient', true, ...
                          'MaxIterations', 100, 'Display', 'none');

    % PMP Optimization Loop
    for k = 1:Np
        % Warm-start co-states
        lambda = lambda_hist(:,k);

        % Hamiltonian minimization with constraints
        [u_opt, ~, exitflag] = fmincon(@(u) Hamiltonian(u, x0, lambda,
Q_terminal), ...
                                      u_guess, [], [], [], [], lb, ub, ...
                                      @(u) path_constraints(u, x0),
options);

        % RK4 State+Co-state integration
        [x_next, lambda_next] = rk4_integration(x0, u_opt, lambda);

        % Update co-state history
        lambda_hist(:,k) = lambda_next;
        x0 = x_next;
    end
end

% Dynamics Integration (RK4 Method)
function [x_next, lambda_next] = rk4_integration(x, u, lambda)
    dt = 0.1; % Tune based on system timescale

    % State integration
    k1_x = state_dynamics(x, u);
    k2_x = state_dynamics(x + 0.5*dt*k1_x, u);
    k3_x = state_dynamics(x + 0.5*dt*k2_x, u);
    k4_x = state_dynamics(x + dt*k3_x, u);
    x_next = x + (dt/6)*(k1_x + 2*k2_x + 2*k3_x + k4_x);

    % Co-state integration
    k1_l = costate_dynamics(x, lambda);
    k2_l = costate_dynamics(x + 0.5*dt*k1_x, lambda + 0.5*dt*k1_l);
    k3_l = costate_dynamics(x + 0.5*dt*k2_x, lambda + 0.5*dt*k2_l);
    k4_l = costate_dynamics(x + dt*k3_x, lambda + dt*k3_l);
    lambda_next = lambda + (dt/6)*(k1_l + 2*k2_l + 2*k3_l + k4_l);
end

% Hamiltonian Function
function [H, gradH] = Hamiltonian(u, x, lambda, Q_terminal)
    % State weighting matrix
    Q = diag([15, 15, 15, 0.3, 0.3, 0.3]);
    R = 0.1*eye(9); % Control effort weighting

    % Terminal cost gradient
    psi_grad = Q_terminal*(x - x_ref);

    % Hamiltonian components
    H = lambda' * state_dynamics(x, u) + 0.5 * (x' * Q * x + u' * R * u) +
psi_grad;

    % Analytical gradient
    if nargout > 1
        gradH = R * u + [lambda(4:6); 0; 0; 0];
    end
end

% Path Constraints Function
function [c, ceq] = path_constraints(u, x)
    % Define path constraints (inequality and equality)
    c = []; % Inequality constraints
    ceq = []; % Equality constraints
end

% Main Simulation Loop with MPC and PMP
for k = 1:N-1
    % Get current state
    x_current = x(:,k);
    pos_norm = norm(x_current(1:3));
    vel_norm = norm(x_current(4:6));
    % Determine current mission phase based on distance
    if pos_norm > 2000
        phase = 1; % Far approach phase
        phase_name = 'Far Approach';
        control_mode = 'Far Approach (PMP-MPC Blend)';
    elseif pos_norm > 500  % Mid-range phase only when distance > 500 m
        phase = 2; % Mid-range phase
        phase_name = 'Mid-Range Approach';
        control_mode = 'Mid-Range (PMP-MPC Blend)';  % Update control mode
    elseif pos_norm > 20  % Close approach phase
        phase = 3; % Close approach phase
        phase_name = 'Close Approach';
        control_mode = 'Close Approach (MPC-Lyapunov Blend)';
    else
        phase = 4; % Final docking phase
        phase_name = 'Final Docking';
        control_mode = 'Final Docking (Terminal Guidance)';
    end

    % State transition matrix (A) for orbital dynamics
    A = [1 0 0 Ts 0 0;
         0 1 0 0 Ts 0;
         0 0 1 0 0 Ts;
         0 0 0 1 0 0;
         0 0 0 0 1 0;
         0 0 0 0 0 1];

    % Input matrix (B) for control inputs
    B = [0 0;
         0 0;
         0 0;
         1 0;
         0 1;
         0 0];

    % Compute control strategies based on phase
    [Q_phase, R_phase, u_constraints, horizon_Np, horizon_Nc] =
phase_specific_control(phase, pos_norm, vel_norm, Q_base, R_base, Np, Nc,
u_max);

    % Solve PMP problem
    u_pmp = pmp_optimize(x_current, A, B, Q_phase, R_phase, S, horizon_Np,
u_min, u_max);

    % Use PMP control as reference for MPC
    U0 = [repmat(u_pmp, horizon_Np, 1); zeros(2 * (Np - horizon_Np), 1)]; %
Initial guess for control inputs

    % Ensure that lb and ub have the same length as U0
    lb = repmat(u_min, horizon_Np, 1); % Lower bounds for control inputs
    ub = repmat(u_max, horizon_Np, 1); % Upper bounds for control inputs

    % Adjust the length of lb and ub if they are shorter than U0
    if length(lb) < length(U0)
        lb = [lb; -Inf * ones(length(U0) - length(lb), 1)];
    end
    if length(ub) < length(U0)
        ub = [ub; Inf * ones(length(U0) - length(ub), 1)];
    end

    options = optimoptions('fmincon', 'Display', 'none', 'Algorithm',
'sqp');

    [U_opt, ~] = fmincon(@(U) mpc_cost(U, x_current, Np, Q_phase, R_phase,
S, A, B), U0, [], [], [], [], lb, ub, [], options);

    % Apply first control input
    u(:,k) = U_opt(1:2);

    % Update state
    x(:,k+1) = x_current + Ts * (A * x_current + B * u(:,k));

    % Initialize u_opt for filtering
    u_opt = U_opt(1:2);

    % Apply Chebyshev filter for control signal smoothing
    if k > 5
        % Get control history for filtering
        u_history = u_raw(:, max(1, k-5):k-1);

        % Chebyshev filter parameters based on phase
        switch phase
            case 1 % Far approach
                filter_order = 2;
                ripple = 1.0;
            case 2 % Mid-range
                filter_order = 3;
                ripple = 0.8;
            case 3 % Close approach
                filter_order = 4;
                ripple = 0.5;
            case 4 % Final docking
                filter_order = 5;
                ripple = 0.3;
        end

        % Apply Chebyshev filter
        [b, a] = cheby1(filter_order, ripple, 0.1, 'low');
        u_filtered = zeros(size(u_opt));
        for i = 1:length(u_opt)
            u_seq = [reshape(u_history(i,:), [], 1); u_opt(i)];
            u_filt = filter(b, a, u_seq);
            u_filtered(i) = u_filt(end);
        end

        % Blend filtered and raw control based on phase
        switch phase
            case 1
                alpha_blend = 0.7;
            case 2
                alpha_blend = 0.5;
            case 3
                alpha_blend = 0.3;
            case 4
                alpha_blend = 0.1;
        end
        u_raw(:,k) = alpha_blend * u_filtered + (1 - alpha_blend) * u_opt;
    else
        % Smooth control signal using Chebyshev filter
        [u_raw(:,k), filter_states_x, filter_states_y] =
apply_chebyshev_filter(u_opt, filter_states_x, filter_states_y, b_cheby,
a_cheby);
    end

    % Store control history
    control_history(:, k) = u_raw(:, k);

    % Safety check with phase-specific velocity limits
    % [safe, safety_control] = check_safety(x_current, v_max,
u_max_scalar);
    % safety_status(k) = safe;
    % if ~safe
    %     u_raw(:,k) = safety_control;
    %     control_mode = 'Safety Override';
    % end

    % Apply dead-band and PWM for thruster control
    u_modified = enhanced_thruster_control(u_raw(:, k), pos_norm, vel_norm,
pwm_counter, phase, u_max_scalar);
    u(:,k) = u_modified;

    % Update PWM counter
    pwm_counter = pwm_counter + 1;
    if pwm_counter > pwm_period
        pwm_counter = 1;
    end

    % Calculate thrust magnitude and energy
    thrust_mag = norm(u(:,k)) * m_chaser;
    thrust_magnitude(k) = thrust_mag;
    thruster_status(k) = (thrust_mag > thrust_threshold);

    % Update energy consumption with improved efficiency calculation
    if k > 1
        if thruster_status(k)
            efficiency_factor = 0.8 + 0.2 * (1 - thrust_mag / (u_max_scalar
* m_chaser));
            energy(k+1) = energy(k) + (thrust_mag * norm(x_current(4:6)) *
Ts) / efficiency_factor;
        else
            energy(k+1) = energy(k);
        end
    end

    % Record thruster firing points for analysis
    if thruster_status(k)
        thruster_firing_points = [thruster_firing_points, x_current(1:2)];
    end

    % Store control mode
    control_mode_history{k} = control_mode;

    % Display progress every 100 steps
    if mod(k, 100) == 0
        disp([ ...
            'Time: ', num2str(t(k)), ' s, Phase: ', phase_name, ...
            ', Distance: ', num2str(pos_norm), ' m, Velocity: ',
num2str(vel_norm), ' m/s, Mode: ', control_mode]);
    end

    % UKF Update Step with phase-specific measurement noise
    switch phase
        case 1 % Far approach
            R_phase_ukf = R * 2.0; % Higher measurement uncertainty when
far
        case 2 % Mid-range
            R_phase_ukf = R * 1.5;
        case 3 % Close approach
            R_phase_ukf = R * 1.0;
        case 4 % Final docking
            R_phase_ukf = R * 0.5; % Lower uncertainty for precision
docking
    end

    % Simulate measurements with phase-specific noise
    z = x_current + mvnrnd(zeros(L, 1), R_phase_ukf)';

    % UKF update
    [x_ukf, P_ukf] = ukf_update(x_ukf, P_ukf, u(:,k), z, Q, R_phase_ukf,
alpha, beta, kappa, Ts, A, B);

    % Store state values
    x(:, k+1) = x_ukf;
    position_history(:, k+1) = x_ukf(1:3);
    velocity_history(:, k+1) = x_ukf(4:6);

    % Check if docking is complete with tighter tolerances for ideal
performance
    if pos_norm < 0.01 && vel_norm < 0.001
        disp(['Docking successfully completed at time: ', num2str(t(k+1)),
' seconds!']);
        disp(['Final position error: ', num2str(pos_norm), ' m']);
        disp(['Final velocity error: ', num2str(vel_norm), ' m/s']);
        u(:,k) = [0; 0];
        u_raw(:,k) = [0; 0];
        break;
    end

    % Coast phase logic
    if pos_norm < coast_distance_threshold && vel_norm <
coast_velocity_threshold
        u(:, k) = [0; 0];
        u_raw(:, k) = [0; 0];
        control_mode = 'Coast Phase';
        coast_phase_active = true;
    end
end

% MPC cost function
function J = mpc_cost(U, x0, Np, Q, R, S, A, B)
    x = x0;
    J = 0;
    for k = 1:Np
        u = U((k-1)*2+1:k*2);
        x = A * x + B * u;
        J = J + x' * Q * x + u' * R * u;
    end
    J = J + x' * S * x; % Terminal cost
end

% Chebyshev Filter Functions
function [b, a] = design_chebyshev_filter(order, ripple, cutoff_freq,
sampling_freq)
    % Design lowpass Chebyshev Type I filter
    normalized_cutoff = cutoff_freq / (sampling_freq / 2);
    [b, a] = cheby1(order, ripple, normalized_cutoff, 'low');
end

function [u_filtered, filter_states_x, filter_states_y] =
apply_chebyshev_filter(u_raw, filter_states_x, filter_states_y, b, a)
    % Apply filter to each control dimension
    [u_filtered(1), filter_states_x] = filter(b, a, u_raw(1),
filter_states_x);
    [u_filtered(2), filter_states_y] = filter(b, a, u_raw(2),
filter_states_y);
end

% Thruster control with adjusted dead-band and PWM parameter
function u_modified = enhanced_thruster_control(u_raw, pos_norm, vel_norm,
pwm_counter, phase, u_max_scalar)
    u_modified = u_raw;

    % Phase-specific parameters
    if phase == 1  % Far approach
        dead_band = 0.05 * u_max_scalar;
        pwm_period = 10;
        pwm_min_duty = 0.2;
    elseif phase == 2  % Mid-range
        dead_band = 0.03 * u_max_scalar;
        pwm_period = 8;
        pwm_min_duty = 0.25;
    elseif phase == 3  % Close approach
        dead_band = 0.01 * u_max_scalar;
        pwm_period = 6;
        pwm_min_duty = 0.3;
    else  % Final docking
        dead_band = 0.005 * u_max_scalar;
        pwm_period = 4;
        pwm_min_duty = 0.4;
    end

    % Incorporate velocity for adaptive dead-band
    velocity_factor = min(1.0, vel_norm / max(0.1, pos_norm/100));
    adjusted_dead_band = dead_band * (1 + velocity_factor);

    % Apply dynamic dead-band
    for i = 1:length(u_modified)
        if abs(u_modified(i)) < adjusted_dead_band
            u_modified(i) = 0;
        end
    end

    % Ensure pwm_counter works with the current period
    pwm_counter_adjusted = mod(pwm_counter-1, pwm_period) + 1;

    % Enhanced PWM implementation
    for i = 1:length(u_modified)
        mag = abs(u_modified(i));

        if mag > 0 && mag < 0.5 * u_max_scalar
            % Progressive duty cycle based on magnitude
            duty_cycle = pwm_min_duty + (1-pwm_min_duty) *
(mag/(0.5*u_max_scalar));

            if pwm_counter_adjusted <= round(pwm_period * duty_cycle)
                u_modified(i) = sign(u_modified(i)) * min(u_max_scalar, mag
/ duty_cycle);
            else
                u_modified(i) = 0;
            end
        end
    end

    % Minimum impulse bit control for final approach
    if phase == 4 && pos_norm < 5
        min_impulse = 0.02 * u_max_scalar;
        for i = 1:length(u_modified)
            if abs(u_modified(i)) > 0 && abs(u_modified(i)) < min_impulse
                u_modified(i) = sign(u_modified(i)) * min_impulse;
            end
        end
    end
end

% Phase-specific control adjustments
function [Q_phase, R_phase, u_constraints, horizon_Np, horizon_Nc] =
phase_specific_control(phase, pos_norm, vel_norm, Q_base, R_base, Np_base,
Nc_base, u_max)
    % Default initialization
    Q_phase = Q_base;
    R_phase = R_base;
    u_constraints = u_max;
    horizon_Np = Np_base;
    horizon_Nc = Nc_base;

    % Phase 1: Far approach (d > 1000m)
    if phase == 1
        Q_phase(1:3,1:3) = Q_base(1:3,1:3) * 0.5;  % Position weight
        Q_phase(4:6,4:6) = Q_base(4:6,4:6) * 0.5;  % Velocity weight
        R_phase = R_base * 5.0;  % Control penalty
        u_constraints = u_max * 0.7;  % Limit thrust for efficiency
        horizon_Np = 50;  % Prediction horizon
        horizon_Nc = 25;  % Control horizon

    % Phase 2: Mid-range approach (100m < d ≤ 1000m)
    elseif phase == 2
        Q_phase(1:3,1:3) = Q_base(1:3,1:3) * 0.8;
        Q_phase(4:6,4:6) = Q_base(4:6,4:6) * 0.8;
        R_phase = R_base * 4.0;
        u_constraints = u_max * 0.8;
        horizon_Np = 40;
        horizon_Nc = 20;
    % Phase 3: Close approach (10m < d ≤ 100m)
    elseif phase == 3
        Q_phase(1:3,1:3) = Q_base(1:3,1:3) * 10;
        Q_phase(4:6,4:6) = Q_base(4:6,4:6) * 5;
        R_phase = R_base * 1.0;
        u_constraints = u_max * 0.9;
        horizon_Np = 30;
        horizon_Nc = 15;

    % Phase 4: Final docking (d ≤ 10m)
    else
        Q_phase(1:3,1:3) = Q_base(1:3,1:3) * 100.0;  % Position weight
        Q_phase(4:6,4:6) = Q_base(4:6,4:6) * 10.0;  % Velocity weight
        R_phase = R_base * 0.1;  % Control penalty
        u_constraints = u_max * 0.5;
        horizon_Np = 10;  % Prediction horizon
        horizon_Nc = 5;   % Control horizon
    end

    % Velocity-specific adjustments
    if vel_norm > 0.5 * sqrt(pos_norm/100)
        Q_phase(4:6,4:6) = Q_phase(4:6,4:6) * 2.0;
    end

    % Final approach precision
    if phase == 4 && pos_norm < 1.0
        Q_phase(1:3,1:3) = Q_phase(1:3,1:3) * 3.0;
        Q_phase(4:6,4:6) = Q_phase(4:6,4:6) * 2.0;
    end
end

% UKF update function
function [x_est, P_est] = ukf_update(x_prev, P_prev, u, z, Q, R, alpha,
beta, kappa, Ts, A, B)
    % State dimension
    n = length(x_prev);

    % Calculate UKF parameters
    lambda = alpha^2 * (n + kappa) - n;
    gamma = sqrt(n + lambda);

    % Weights calculation
    Wm = zeros(2*n+1, 1);
    Wc = zeros(2*n+1, 1);
    Wm(1) = lambda / (n + lambda);
    Wc(1) = lambda / (n + lambda) + (1 - alpha^2 + beta);
    for i = 2:2*n+1
        Wm(i) = 1 / (2*(n + lambda));
        Wc(i) = 1 / (2*(n + lambda));
    end

    % Ensure P_prev is positive definite
    P_prev = (P_prev + P_prev') / 2;  % Ensure symmetry
    P_prev = P_prev + 1e-6 * eye(n);  % Add small regularization

    % Generate sigma points
    sigma_points = zeros(n, 2*n+1);
    sigma_points(:,1) = x_prev;
    % Calculate square root of P using Cholesky decomposition
    sqrt_P = chol((n + lambda) * P_prev, 'lower');

    for i = 1:n
        sigma_points(:,i+1) = x_prev + sqrt_P(:,i);
        sigma_points(:,i+1+n) = x_prev - sqrt_P(:,i);
    end

    % Prediction step
    sigma_points_pred = zeros(n, 2*n+1);
    for i = 1:2*n+1
        % Propagate each sigma point through the dynamics model
        sigma_points_pred(:,i) = A * sigma_points(:,i) + B * u;
    end

    % Calculate predicted mean
    x_pred = zeros(n, 1);
    for i = 1:2*n+1
        x_pred = x_pred + Wm(i) * sigma_points_pred(:,i);
    end

    % Calculate predicted covariance
    P_pred = Q;  % Start with process noise
    for i = 1:2*n+1
        diff = sigma_points_pred(:,i) - x_pred;
        P_pred = P_pred + Wc(i) * (diff * diff');
    end

    % Update step with measurements
    H = eye(n);
    z_pred = zeros(length(z), 2*n+1);
    for i = 1:2*n+1
        z_pred(:,i) = H * sigma_points_pred(:,i);  % Apply measurement
model
    end

    % Predicted measurement
    z_mean = zeros(length(z), 1);
    for i = 1:2*n+1
        z_mean = z_mean + Wm(i) * z_pred(:,i);
    end

    % Innovation covariance
    S = R;  % Start with measurement noise
    for i = 1:2*n+1
        diff = z_pred(:,i) - z_mean;
        S = S + Wc(i) * (diff * diff');
    end

    % Cross correlation matrix
    Pxz = zeros(n, length(z));
    for i = 1:2*n+1
        diff_x = sigma_points_pred(:,i) - x_pred;
        diff_z = z_pred(:,i) - z_mean;
        Pxz = Pxz + Wc(i) * (diff_x * diff_z');
    end

    % Kalman gain (using pseudoinverse for stability)
    K = Pxz * pinv(S);

    % State and covariance update (Joseph form for stability)
    I = eye(n);
    P_est = (I - K * H) * P_pred * (I - K * H)' + K * R * K';
    x_est = x_pred + K * (z - z_mean);
end

% Performance Metrics
% Calculate Total Maneuver Time
total_maneuver_time = t(k+1);

% Calculate Final Position and Velocity Error
final_position_error = norm(x(1:3, k+1));
final_velocity_error = norm(x(4:6, k+1));

% Calculate Total Energy Consumed
total_energy_consumed = energy(k+1);

% Calculate Maximum Thrust Magnitude
max_thrust_magnitude = max(thrust_magnitude(1:k));

% Calculate Total Thruster On Time
total_thruster_on_time = sum(thruster_status(1:k)) * Ts;
thruster_on_percentage = (total_thruster_on_time / total_maneuver_time) *
100;

% Display Performance Metrics
disp(' ');
disp('Performance Metrics');
disp(['Total Maneuver Time: ', num2str(total_maneuver_time), ' seconds']);
disp(['Final Position Error: ', num2str(final_position_error), ' m']);
disp(['Final Velocity Error: ', num2str(final_velocity_error), ' m/s']);
disp(['Total Energy Consumed: ', num2str(total_energy_consumed), ' J']);
disp(['Maximum Thrust Magnitude: ', num2str(max_thrust_magnitude), ' N']);
disp(['Total Thruster On Time: ', num2str(total_thruster_on_time), '
seconds (', num2str(thruster_on_percentage), '%)']);

% Plotting
% Figure 1: R-bar Mode Docking Simulation
figure('Name', 'MPC Docking Simulation');
subplot(2,2,1);
plot(x(1,1:k+1), x(2,1:k+1), 'b-', 'LineWidth', 2);
hold on;
plot(0, 0, 'r*', 'MarkerSize', 10);
plot(x(1,1), x(2,1), 'go', 'MarkerSize', 10);
grid on;
xlabel('V-bar (m)');
ylabel('R-bar (m)');
title('Approach Trajectory');
legend('Trajectory', 'Target', 'Start', 'Location', 'best');

subplot(2,2,2);
plot(t(1:k+1), sqrt(sum(x(1:3,1:k+1).^2)), 'b-', 'LineWidth', 2);
grid on;
xlabel('Time (s)');
ylabel('Distance (m)');
title('Distance to Target');
```

```
subplot(2,2,3);
plot(t(1:k+1), x(4,1:k+1), 'r-', 'LineWidth', 2, 'DisplayName', 'V_x');
hold on;
plot(t(1:k+1), x(5,1:k+1), 'b-', 'LineWidth', 2, 'DisplayName', 'V_y');
plot(t(1:k+1), sqrt(x(4,1:k+1).^2 + x(5,1:k+1).^2), 'k--', 'LineWidth', 1,
'DisplayName', 'V_{mag}');
grid on;
xlabel('Time (s)');
ylabel('Velocity (m/s)');
title('Velocity Components');
legend('Location', 'best');

subplot(2,2,4);
stairs(t(1:k), u(1,1:k), 'r-', 'LineWidth', 2, 'DisplayName', 'u_x');
hold on;
stairs(t(1:k), u(2,1:k), 'b-', 'LineWidth', 2, 'DisplayName', 'u_y');
stairs(t(1:k), sqrt(u(1,1:k).^2 + u(2,1:k).^2), 'k--', 'LineWidth', 1,
'DisplayName', 'u_{mag}');
grid on;
xlabel('Time (s)');
ylabel('Control Input (m/s^2)');
title('Control Inputs');
legend('Location', 'best');

% Figure 2: Thrust Comparison
figure('Name', 'Thrust Comparison');
subplot(2,1,1);
stairs(t(1:k), u_raw(1,1:k), 'r--', 'LineWidth', 1, 'DisplayName', 'Raw
u_x');
hold on;
stairs(t(1:k), u_raw(2,1:k), 'b--', 'LineWidth', 1, 'DisplayName', 'Raw
u_y');
stairs(t(1:k), u(1,1:k), 'r-', 'LineWidth', 2, 'DisplayName', 'PWM u_x');
stairs(t(1:k), u(2,1:k), 'b-', 'LineWidth', 2, 'DisplayName', 'PWM u_y');
grid on;
xlabel('Time (s)');
ylabel('Control Input (m/s^2)');
title('Raw vs PWM Control Inputs');
legend('Location', 'best');

subplot(2,1,2);
plot(t(1:k), thrust_magnitude(1:k), 'g-', 'LineWidth', 2);
grid on;
xlabel('Time (s)');
ylabel('Thrust Magnitude (N)');
title('Thrust Magnitude');

% Figure 3: Thruster Activity
figure('Name', 'Thruster Activity');
subplot(2,1,1);
stem(t(1:k), thruster_status(1:k), 'k-', 'LineWidth', 1, 'Marker', 'none');
grid on;
xlabel('Time (s)');
ylabel('Thruster Status');
title('Thruster On/Off Status');
ylim([-0.1 1.1]);

subplot(2,1,2);
cumulative_on_time = cumsum(thruster_status(1:k)) * Ts;
```

```
plot(t(1:k), cumulative_on_time, 'm-', 'LineWidth', 2);
grid on;
xlabel('Time (s)');
ylabel('Cumulative On Time (s)');
title('Cumulative Thruster On Time');

% Figure 4: Energy & Approach Analysis
figure('Name', 'Energy & Approach Analysis');
subplot(2,1,1);
plot(t(1:k+1), energy(1:k+1), 'm-', 'LineWidth', 2);
grid on;
xlabel('Time (s)');
ylabel('Energy Consumption (J)');
title('Energy Consumption Over Time');

subplot(2,1,2);
plot(sqrt(sum(x(1:3,1:k+1).^2)), sqrt(sum(x(4:6,1:k+1).^2)), 'c-',
'LineWidth', 2);
grid on;
xlabel('Distance to Target (m)');
ylabel('Approach Speed (m/s)');
title('Approach Speed vs Distance');

% Figure 5: Trajectory and Thruster Firing Points
figure('Name', 'Trajectory and Thruster Firing Points');
plot(x(1,1:k+1), x(2,1:k+1), 'b-', 'LineWidth', 2);  % Chaser trajectory
hold on;
plot(0, 0, 'r*', 'MarkerSize', 10);  % Target position
plot(x(1,1), x(2,1), 'go', 'MarkerSize', 10);  % Chaser starting position

% Plot thruster firing points
if ~isempty(thruster_firing_points)
    plot(thruster_firing_points(1,:), thruster_firing_points(2,:), 'rx',
'MarkerSize', 3);
end

grid on;
xlabel('V-bar (m)');
ylabel('R-bar (m)');
title('Trajectory and Thruster Firing Points');
legend('Chaser Trajectory', 'Target', 'Chaser Start', 'Thruster Firing',
'Location', 'best');
```