Autonomous Thruster Failure Recovery for Underactuated Spacecraft

Christopher Masaru Pong, David W. Miller

September 2010  SSL #13–10
Autonomous Thruster Failure Recovery for Underactuated Spacecraft

Christopher Masaru Pong, David W. Miller

September 2010

SSL #12-11

This work is based on the unaltered text of the thesis by Christopher Masaru Pong submitted to the Department of Aeronautics and Astronautics in partial fulfillment of the requirements for the degree of Master of Science at the Massachusetts Institute of Technology.
Autonomous Thruster Failure Recovery for Underactuated
Spacecraft

by

Christopher Masaru Pong

Submitted to the Department of Aeronautics and Astronautics
on August 19, 2010, in partial fulfillment of the
requirements for the degree of
Master of Science in Aeronautics and Astronautics

Abstract

Thruster failures historically account for a large percentage of failures that have occurred on orbit. Therefore, autonomous thruster failure detection, isolation, and recovery (FDIR) is an essential component to any robust space-based system. This thesis focuses specifically on developing thruster failure recovery techniques as there exist many proven thruster FDI algorithms. Typically, thruster failures are handled through redundancy—if a thruster fails, control can be allocated to other operational thrusters. However, with the increasing push to using smaller, less expensive satellites there is a need to perform thruster failure recovery without additional hardware, which would add extra mass, volume, and complexity to the spacecraft. This means that a thruster failure may cause the spacecraft to become underactuated, requiring more advanced control techniques. Therefore, the objective of this thesis is to develop and analyze thruster failure recovery techniques for the attitude and translational control of underactuated spacecraft.

To achieve this objective, first, a model of a thruster-controlled spacecraft is developed and analyzed with linear and nonlinear controllability tests. This highlights the challenges involved with developing a control system that is able to reconfigure itself to handle thruster failures. Several control techniques are then identified as potential candidates for solving this control problem. Solutions to many issues with implementing one of the most promising techniques, Model Predictive Control (MPC), are described such as a method to compensate for the large delays caused by solving an nonlinear programming problem in real time. These control techniques were implemented and tested in simulation as well as in hardware on the SPHERES testbed. These results show that MPC provided superior performance over a simple path planning technique in terms of maneuver completion time and number of thruster failure cases handled at the cost of a larger computational load and slightly increased fuel usage. Finally, potential extensions to this work as well as alternative methods of providing thruster failure recovery are provided.
Thesis Supervisor: David W. Miller
Title: Professor of Aeronautics and Astronautics

Thesis Supervisor: Alvar Saenz-Otero
Title: Research Scientist, Aeronautics and Astronautics
Acknowledgments

This work was performed primarily under contract AFS09-1297 with Aurora Flight Sciences as part of the SPHERES Fault Detection and Reconfiguration program. The author gratefully thanks the sponsors for their generous support that enabled this research.
Contents

1 Introduction ................................................................. 17
   1.1 Motivation ........................................................... 17
   1.2 Overview of the SPHERES Testbed ................................. 21
   1.3 Approach & Thesis Overview ...................................... 23

2 Literature Review & Gap Analysis ..................................... 25
   2.1 Thruster Fault Detection and Isolation .............................. 25
   2.2 Thruster Failure Recovery .......................................... 29
   2.3 Gap Analysis .......................................................... 31

3 Spacecraft Model ......................................................... 33
   3.1 Rigid-Body Dynamics ................................................ 33
       3.1.1 Reference Frames & Rotations ................................. 34
       3.1.2 Attitude Kinematics & Kinetics ............................... 37
       3.1.3 Translational Kinematics & Kinetics ........................... 39
   3.2 Equations of Motion of a Thruster-Controlled Spacecraft ........ 39
       3.2.1 Six-Degree-of-Freedom Model ................................. 40
       3.2.2 Three-Degree-of-Freedom Model .............................. 41
   3.3 Controllability ....................................................... 42
       3.3.1 Linear Time-Invariant Controllability ....................... 43
       3.3.2 Small-Time Local Controllability ............................. 47
   3.4 Summary ............................................................... 56
4 Thruster Failure Recovery Techniques

4.1 Control System Design Challenges

4.1.1 Coupling

4.1.2 Multiplicative Nonlinearities

4.1.3 Saturation

4.1.4 Nonholonomicity

4.2 Reconfigurable Control Allocation

4.2.1 Redistributed Pseudoinverse

4.2.2 Active Set Method

4.3 Path Planning

4.3.1 Piecewise Trajectory

4.3.2 Rapidly Exploring Dense Trees

4.4 Model Predictive Control

4.4.1 Stability

4.4.2 Optimality

4.5 Summary

5 Model Predictive Control Implementation Issues

5.1 Regulation & Attitude Error

5.2 Nonlinear Programming Algorithm

5.2.1 Selection

5.2.2 Implementation

5.3 Processing Delay

5.4 Feasibility & Guaranteed Stability

5.5 Summary

6 Simulation & Hardware Testing Results

6.1 Six-Degree-of-Freedom Results

6.1.1 Baseline Translation

6.1.2 Reconfigurable Control Allocation

6.1.3 Piecewise Trajectory
6.1.4 Model Predictive Control ........................................... 105

6.2 Three-Degree-of-Freedom Results .................................. 109
  6.2.1 Baseline Translation .................................................. 110
  6.2.2 Model Predictive Control .......................................... 110

7 Conclusion ................................................................. 115
  7.1 Thesis Summary .......................................................... 115
  7.2 Contributions ............................................................ 119
  7.3 Recommendations & Future Work .................................... 121

A Small-Time Local Controllability of SPHERES .................... 123
  A.1 Three-Degree-of-Freedom SPHERES Model ....................... 124
  A.2 Six-Degree-of-Freedom SPHERES Model .......................... 127

B Optimization ............................................................... 131
  B.1 Nonlinear Programming: Sequential Quadratic Programming .. 131
  B.2 Quadratic Programming: Active Set Method ...................... 142
List of Figures

1-1 Distribution of attitude and orbit control system failures. . . . . . . . 18
1-2 Artist’s conception of DARPA’s System F6. . . . . . . . . . . . . . . 20
1-3 SPHERES satellites on the ISS. . . . . . . . . . . . . . . . . . . . . . 21
1-4 SPHERES satellite without shell. . . . . . . . . . . . . . . . . . . . . 21

2-1 NASA’s Cassini spacecraft with direct redundancy. . . . . . . . . . . 30
2-2 ESA’s Automated Transfer Vehicle with functional redundancy. . . . 30

3-1 Thruster locations, directions and physical properties for an example 3DOF spacecraft. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
3-2 Pictorial representation of flow along vector fields producing non-zero net motion. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
3-3 Differential-drive example showing a series of four motion primitives. 49
3-4 Thruster locations and directions for 3DOF SPHERES spacecraft. . . 54
3-5 Thruster locations and directions for 6DOF SPHERES spacecraft. . . 55

4-1 Decomposition of a controller into a control law and control allocator. 64

5-1 Example showing how the SQP algorithm iteratively approaches the constrained global optimum of the Rosenbrock function. . . . . . . . 88
5-2 Delay compensation method for mitigating the effects of processing delay. 90

6-1 Thruster locations and directions for 6DOF SPHERES spacecraft. . . 96
6-2 6DOF representative maneuver: PD control, all thrusters operational (simulation). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Variable definitions for the 6DOF model</td>
<td>41</td>
</tr>
<tr>
<td>3.2</td>
<td>Variable definitions for the 3DOF model</td>
<td>42</td>
</tr>
<tr>
<td>3.3</td>
<td>Small-time local controllability of SPHERES</td>
<td>55</td>
</tr>
<tr>
<td>5.1</td>
<td>Comparison of the SQP algorithm versus \texttt{fmincon} run time on an example problem</td>
<td>89</td>
</tr>
<tr>
<td>6.1</td>
<td>6DOF SPHERES parameters</td>
<td>96</td>
</tr>
<tr>
<td>6.2</td>
<td>6DOF MPC parameters</td>
<td>107</td>
</tr>
<tr>
<td>6.3</td>
<td>Comparison of the piecewise trajectory and MPC techniques</td>
<td>108</td>
</tr>
<tr>
<td>6.4</td>
<td>3DOF SPHERES parameters</td>
<td>110</td>
</tr>
<tr>
<td>6.5</td>
<td>3DOF MPC parameters</td>
<td>112</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

Failure Detection, Isolation, and Recovery (FDIR) is an integral part of any robust space-based system. If a satellite fails on orbit, currently there is little to no chance of being able to send an agent to that satellite and provide on-orbit servicing. Thus, it is paramount for a satellite to be able to tell if something is wrong (failure detection), determine what subsystem or component has failed (isolation), and appropriately change the system to handle this failure (recovery).

There are many types of spacecraft failures that can occur on orbit. Since it is impractical to create a spacecraft that can autonomously detect, isolate, and recover from all possible failures, it is important to prioritize these failures to ensure that the most likely failures with a large impact on the mission are handled. To begin, broad classifications of failures will be considered. Sarsfield has classified spacecraft failures into four distinct categories [1]:

- **Failures Caused by the Space Environment**: This includes space debris impact, degradation due to atomic oxygen, thermal fluctuations from solar and albedo radiation, charging and arcing due to solar wind, and single-event upsets, latchups, and burnouts from radiation.

- **Design Failures**: This includes all failures due to improperly designed sys-
tems such as buckling due to incorrect modeling and analysis of the spacecraft structure.

- **Failures Related to Parts and Quality**: This includes all component failures not attributed to the environment or design error. These failures normally occur randomly due to degradation over time such as the fatigue of structural components.

- **Other Types of Failures**: This includes all other types of failures such as bad commands from ground operators, errors in software, and out-of-bound conditions such as exposure of sensitive optics to direct sunlight.

While all of these types of failures are worthy of further research, this thesis will focus on random component failures, specifically thruster failures. Tafazoli has conducted an extensive review of failures that have occurred on 129 military and commercial spacecraft from 1980 to 2005 [2]. One of the many useful results of this study was determining the distribution of failures that have occurred on these spacecraft. Figure 1-1 shows that thruster failures account for 24% of all the Attitude and Orbit Control System (AOCS) failures.

![Figure 1-1: Distribution of attitude and orbit control system failures](image)

For this plot, the xenon-ion propulsion system has been included in the general thruster category.
Because thruster failures are one of the more likely failures to occur on a spacecraft, many have developed ways to mitigate the effect of these failures. One of the obvious methods of handling these failures is through the use of additional hardware. Sensors can be embedded in the thruster itself for thruster failure detection (see, e.g., [3]) and additional redundant thrusters can be used to replace the failed thrusters.

Sarsfield provides an interesting view on redundancy [1]: “Historically, redundancy has been a central method of achieving resistance to failure and has been incorporated up to the point at which the incremental costs of including it began to exceed reductions in the cost of failure.” In addition to the impact of redundancy on the cost budget, it also impacts mass, volume, and power budgets on the spacecraft.

For large, monolithic spacecraft with a cost budget on the order of billions of dollars, redundancy may well be justified. However, there is a move toward launching smaller, less expensive spacecraft. One example is a payload adapter for the Evolved Expendable Launch Vehicle (EELV) family termed the EELV Secondary Payload Adapter (ESPA) ring [4]. The ESPA ring allows up to six secondary and one primary payload to share a ride to space. By sharing this cost, launching a 180-kg spacecraft on an ESPA ring can be less than 5% of the cost of a dedicated launch vehicle. Another, more extreme, example of less expensive spacecraft are satellites that follow the CubeSat standard [5]. By adhering to this standard, these spacecraft can be launched from a Poly-Picosatellite Orbital Deployer (P-POD) [6], which can be bolted to the upper stage of many different types of launch vehicles (e.g., Rokot, Kosmos-3M, M-V, Dnepr, Minotaur, PSLV, Falcon 1). These CubeSats can be launched at a very small fraction of the launch vehicle cost. Recently, the National Aeronautics and Space Administration (NASA) has announced that it will provide subsidized CubeSat launch opportunities, enabling launches at the cost of $30,000 per CubeSat unit [7]. For these less expensive, smaller satellites, it may very well be more costly to include redundancy than risk failure.

There is also a push to replace the functionality of a monolithic spacecraft with multiple spacecraft. One such example is DARPA’s System F6 Program, shown in Figure [1-2]. The purpose of this program is to demonstrate a fractionated spacecraft
architecture, where each spacecraft has a specialized role in the cluster. As an example, there might be one spacecraft that performs most of the computations necessary for the mission and another to gather most of the power for the cluster. The main motivation is that fractionated space architectures are flexible in the face of diverse risks such as component failures, obsolescence, funding continuity, and launch failures [8]. One of many costs associated with buying this flexibility and robustness is that the cluster as a whole has a lot of redundancy. Even though there may be a single specialized “power” spacecraft, each spacecraft must have its own power subsystem. Including redundancy for each spacecraft to handle failures on top of this, may be too costly to justify the benefits. With the prospect of having multiple spacecraft in a cluster or formation, methods of performing thruster FDIR without the use of hardware redundancy has become a vital topic of research.

Figure 1-2: Artist’s conception of DARPA’s System F6 [9].
1.2 Overview of the SPHERES Testbed

To aid in the development and maturation of estimation and control algorithms for proximity operations such as formation flight, inspection, servicing, assembly, the Massachusetts Institute of Technology (MIT) Space Systems Laboratory (SSL) has developed the Synchronized Position Hold, Engage, Reorient, Experimental Satellites (SPHERES) testbed. This testbed consists of six nanosatellites: three at the MIT SSL and three on board the International Space Station (ISS). The three satellites on the ISS can be seen flying in formation in Figure 1-3 and a satellite without its shell can be seen in Figure 1-4. These satellites will serve as a representative spacecraft on which to test the developed thruster FDIR algorithms.

![Figure 1-3: SPHERES satellites on the ISS.](image)

![Figure 1-4: SPHERES satellite without shell.](image)

The SPHERES satellites contain all the necessary subsystem functionality of a typical satellite:

- **Avionics:** A Sundance SMT375 is used as the main avionics board, which includes a processor, memory, and FPGA. A Texas Instruments C6701 DSP performs all the necessary on-board computations for the satellite at a clock speed of 167 MHz. 16 MB of RAM and 512 KB of flash ROM are available as on-board memory. External analog inputs are digitized with a 12-bit D/A on the FPGA.

- **Communication:** Two RFM DR2000 transceivers are used to communicate
on two different channels (868 and 916 MHz) at an effective rate of 18 kbps between the satellites and ground station.

- **Metrology:** Inertial measurements are provided by 3 Q-Flex QA-750 accelerometers, and 3 BEI GyroChip II gyroscopes. Measurements of position and attitude relative to the surrounding laboratory frame are provided by 24 on-board ultrasound receivers. Time-of-flight data from up to five beacons placed in the laboratory frame to the on-board receivers provide global position to within a few millimeters and attitude to within 1-2 degrees [10].

- **Power:** 16 AA batteries provide the satellite with an average 15 W at 12 V.

- **Propulsion:** CO$_2$ is stored in a replenishable tank at 860 psi, fed through a regulator to be stepped down to 25 psi (set to 35 psi for ground testing), and expelled through 12 solenoid valves and nozzles, each producing around 120 mN of thrust.

Additional information on the SPHERES testbed can be found in [11, 12, 13, 14].

Testing in the space environment is typically a high-risk and costly endeavor where any anomaly or failure can easily result in the loss of a mission (see, e.g., the Demonstration of Autonomous Rendezvous Technology (DART) spacecraft [15]). The SPHERES testbed provides researchers with the ability to push the limits of new algorithms by performing testing in a low-risk, representative environment. If something goes wrong, the astronaut running the SPHERES testbed can simply stop a test and grab the satellites. Three different development environments are used: the six-degree-of-freedom (6DOF) MATLAB simulation, the three-degree-of-freedom (3DOF) ground testbed, and the 6DOF ISS testbed. These environments allow algorithms to be iteratively matured to higher Technology Readiness Levels, a measure of maturity used by NASA and the Department of Defense (DoD).
1.3  Approach & Thesis Overview

This thesis is divided into seven chapters. Chapter 2 provides a literature review and gap analysis of thruster FDIR and the focuses the thesis on thruster failure recovery techniques. Chapter 3 derives the model of a thruster-controlled spacecraft and analyzes the controllability of the model using linear and nonlinear techniques. Chapter 4 outlines the challenges associated with developing a controller that is able to handle thruster failures, analyzes techniques to handle these challenges, and selects three candidate control techniques. Chapter 5 reveals many of the implementation issues that are not addressed in the theory of Model Predictive Control (MPC), the most promising control technique for thruster failure recovery. Chapter 6 presents the results of simulation and hardware testing of the various thruster failure recovery techniques. Chapter 7 concludes the thesis with contributions and recommendations for future work.
Chapter 2

Literature Review & Gap Analysis

This chapter provides a summary of the literature on thruster FDIR and a discussion of the gap in the literature that will be addressed by this thesis. The chapter is organized into three sections. Section 2.1 reviews the methods that have been employed to detect and isolate both general actuator failures as well as methods developed specifically for thruster failures. Section 2.2 discusses how thruster failures are normally handled through control allocation as well as more advanced methods for controlling the attitude of underactuated spacecraft. Finally, Section 2.3 discusses how the literature has not addressed the issue of controlling the attitude and translational dynamics of a spacecraft that has become underactuated due to a thruster failure.

2.1 Thruster Fault Detection and Isolation

One option for detecting thruster failures is through the use of specialized pressure and temperature sensors in the nozzle of a thruster. This, however, comes at the price of extra mass, cost and complexity. This section instead provides a survey of generalizable methods of performing thruster FDI using only additional software and hardware already on board.

Kalman filters inherently have a built-in failure detection scheme: the Kalman filter innovation. This is defined as the difference in the measured output of the
system, and the estimated output of the system. In the nominal case, it is assumed that the plant model approximates the actual system reasonably well. For this nominal operating condition, there will always be some non-zero Kalman filter innovation due to process and sensor noise, from which the baseline threshold can determined. Where this threshold is placed exactly, brings up the underlying trade that must be made for all FDI systems: detection speed versus accuracy. When a failure occurs, the estimated and measured outputs begin to diverge. This happens because the estimator no longer accurately models the system—the system dynamics have changed due to the failure. When the innovation passes through the set threshold, a failure is detected. Therefore, if the threshold is set too low, false failures may be detected. However, if the threshold is set too high, the time before detection increases.

Willsky [16] has outlined numerous methods of creating failure-specific filters, where the gains of the estimator can be tweaked or the weights of the estimator can be weighted in such a way that it is more sensitive to specific failures. Thus, when one of those failures occurs, the failure is detected and isolated to a smaller subset of possible failures. The innovations can also be analyzed for specific signatures that only a particular failure would have. Because these filters are tweaked from their more optimal configuration, it is common to employ a normal filter for state estimation and have a 'failure-monitor' filter that detects failures. Recently, Chen and Speyer have developed a least-squares filter that explicitly monitors a single failure while ignoring all other failures [17]. This is done by reformulating the least-squares derivation for a Kalman filter, such that one particular failure is highlighted while other similar or 'nuisance' faults are placed in an unobservable subspace.

The next logical step to detect multiple possible failures is to run multiple Kalman filters simultaneously. Work by Willsky, Deyst and Crawford [18] develop the use of a bank of Kalman filters, based off the work of Buxbaum and Haddad [19]. There is a single Kalman filter for every expected failure type as well as the no-failure case. These filters are created with the different dynamics of the various failure modes. Therefore, when the system is operating under nominal conditions, all the Kalman filters would have high innovations except for the nominal filter. When a
failure occurs, the nominal filter innovation increases (indicating a failure) and the innovation of the filter monitoring the failure that occurred decreases (isolating the failure). While this is an intuitively simple method of detecting failures, it is extremely computationally intensive since it is running multiple Kalman filters in parallel. One possible way of reducing the computation of this bank of filters is to only run the nominal filter until a failure is detected. When this occurs, the other filters can be initialized and run until the failure is isolated. When it is isolated, the other filters can be turned off. This trades a longer isolation time with less computational power.

A different FDI technique developed specifically for thruster failures is maximum-likelihood FDI originally developed for the Shuttle reaction control subsystem jets by Deyst and Deckert [20]. It is an intuitive method that uses knowledge of the thruster geometry of the spacecraft as well as inertial measurement unit (IMU) data to determine which thruster has failed, if any. From the thruster geometry, one can calculate the expected body-fixed rotational and translational accelerations for each thruster, called influence coefficients. It must be noted here that for thrusters that are placed in relatively close proximity and similar directions, the influence coefficient are near identical. Thruster force variations along with only slightly different thrust directions and moment arms means that it is difficult to distinguish failures between these thrusters without exercising each one individually. These thruster influence coefficient vectors are stored in memory for future use. An estimator for the rotational and translational dynamics are also developed. These estimators incorporate real effects of the rate gyros and accelerometers such as quantization noise and biases. The most important part about these estimators is that they calculate estimates of disturbance accelerations, assumed to be constant between samples of the IMU.

Thruster failures can be detected through the use of the Kalman filter innovation. If the magnitude of the innovation exceeds the set threshold, a failure is detected. Once detected, the process noise covariance is increased by a specified amount. This reconfigures the estimator to trust the sensor data more than the model (which is now inaccurate due to the failed thruster). The thruster failure is then isolated through the calculation of a likelihood parameter, which is a function of the IMU sample, in-
fluence coefficients, and the covariance matrix of the estimated accelerations. These likelihood parameters of each thruster are compared against each other and set thresholds to determine which thruster has failed. This process allows for the detection and isolation of a single thruster failure, but not multiple failures.

A modification and extension of the maximum likelihood FDI algorithm has been developed by Wilson and Sutter [21]. The estimation of the disturbing acceleration is the same as the maximum likelihood case, with one useful extension. The model can incorporate properties such as moment of inertias, center of gravity, thruster blowdown (the reduction in thrust due to lower internal pressure) that are identified online or during the operation of the system. While nominal values can still be used, this allows for a more robust system because it reduces the sensitivity to system uncertainty and increases the “signal-to-noise” ratio.

Three additional steps of collection, windowing and filtering are also outlined to provide an even further increase in signal-to-noise. The collection process simply keeps track of the estimated accelerations as well as when certain thrusters are active or inactive. This specifically addresses the issue of a thruster with a hard-off failure that is infrequently commanded to thrust. The windowing parameter specifies the number of previous estimates and measurements to consider for the likelihood parameter. There is a tradeoff between fast detection of failures and reduction in noise through larger averaging, so the window size must be chosen appropriately depending on the system being considered. The filtering is performed simply by taking an average of the measured and estimated accelerations. Since the collection step separates the time when the thrusters are active versus inactive, the likelihood parameter for these two distinct time periods can be calculated. This provides more information to isolate failures faster.

This updated FDI method actually solves three problems of the original maximum likelihood FDI technique. The first is that it is more sensitive and therefore more likely to correctly isolate failed-off failures that are not frequently commanded to thrust. The second is that the separation of data for when the thruster is active or inactive allows the distinction between closely-positioned thrusters, assuming that
they are not always on and off at the same time. Even if they are, the controller could explicitly exercise one and not the other, to determine which thruster has failed. The third is that this algorithm is able to detect multiple jet failures as long as the disturbing acceleration or influence coefficient is correctly catalogued a priori.

2.2 Thruster Failure Recovery

After a thruster failure has been detected and isolated, the system can attempt to recover from this failure. Failed-on thrusters will not be explicitly considered because it is very difficult to recover from this type of failure. Without a valve to shut off this thruster, the only way to control the spacecraft with a failed-on thruster is to cancel the force and torque of the failed-on thruster, quickly depleting fuel and greatly reducing the spacecraft lifetime. Therefore, a thruster that has failed on will only be considered in the case that a valve can be closed, converting the failed-on thruster to a failed-off thruster.

General techniques for actuator failure recovery can possibly be applied to recover from these failures. Beard describes a simple way using controllability matrices to determine the minimum amount of actuators that the system needs to be controllable, thus determining the maximum amount of failures that the system can handle [22]. Beard also describes three techniques where control reconfiguration is done simply through recalculating the linear feedback gains. This process, while intuitively simple, can be done in many ways such as transforming the system such that the effect of actuators are decoupled, calculating the gains then transforming the gains back into the original state. These various techniques have advantages and disadvantages of complexity and computation time. It will be shown in Section 3.3.1 that the linearized system is not LTI controllable, therefore a simple recalculation of feedback gains is not suitable.

Thruster failure recovery has traditionally been handled through redundancy or overactuation. Overactuation means that there are more actuators than necessary to
produce any arbitrary force and torque\(^1\) This is in contrast with a fully actuated spacecraft, which has just enough actuators to produce any arbitrary force and torque. If any thruster failure occurs on a fully actuated spacecraft it becomes underactuated. NASA’s Cassini spacecraft is a simple example of an overactuated spacecraft with redundancy [23]. For example, if one of the main bi-propellant engines, shown in Figure 2-1 fails, the second can be used as a direct replacement to the failed engine. Another example is ESA’s Automated Transfer Vehicle (ATV) that uses 28 thrusters for attitude control [24], shown in Figure 2-2.

Figure 2-1: NASA’s Cassini spacecraft with direct redundancy.  
Figure 2-2: ESA’s Automated Transfer Vehicle with functional redundancy.

For overactuated spacecraft, if a thruster failure occurs, a reconfigurable control allocator is typically used to reallocate any control that would be actuated by the failed thruster to other thrusters that can provide the same control. There is a large body of literature on this topic, which is discussed further in Section 4.2. Wilson provides an interesting extension to this work which utilizes ideas from neural networks to reconfigure a control allocator in the event of a failure [25]. The neural network takes in the thruster firing commands and the resulting accelerations to perform a system identification. This information allows the neural network to “train”

\(^1\)This definition is somewhat ill-defined. There are saturation limits that make it impossible to produce an arbitrary force and torque. A more technically correct definition is that there are more actuators than necessary such that the convex hull of the set of forces and torques generated by the feasible control inputs contains the origin.
the control allocator to give better commands to produce the desired accelerations. This system was shown to be able to recover from failed-off thrusters as well as large thruster misalignments.

Thruster failure recovery for underactuated spacecraft has received some attention in the literature as well. Tsiotras and Doumtchenko provide an excellent review of work done in the area of underactuated attitude control of spacecraft [26]. Of note is work by Krishnan, Reyhanoglu, McClamroch which uses a discontinuous feedback controller to stabilize a spacecraft’s attitude using only two control torques about the principal axes [27]. Controllability and stabilizability properties are provided for the spacecraft attitude dynamics for an axially and non-axially symmetric spacecraft and it is shown that the dynamics cannot be asymptotically stabilized using continuous feedback. A discontinuous feedback controller is constructed by switching the continuous feedback controller used, following a sequence of maneuvers. This is shown to be able to arbitrarily reorient the spacecraft’s attitude. In all of these references, the control inputs are considered to be torques provided by pairs of thrusters or reaction wheels.

Failure recovery for the translational degrees of freedom have also been covered to a more limited extent in the literature. Breger developed maneuvers that are safe to thruster failures during the spacecraft docking [28]. Model predictive control is employed to ensure that there is a passive or active trajectory that the spacecraft can follow in the event of a thruster failure, to avoid a collision. This framework does not address the ability of the spacecraft to regulate its position about an equilibrium, but rather the ability to avoid a collision with the spacecraft to which it is docking. The control inputs are assumed to be forces provided by thrusters acting through the spacecraft’s center of mass.

2.3 Gap Analysis

A review of the literature for thruster FDIR has been provided. Several methods for thruster FDI have been described, including general methods applicable to any
type of failure (bank of Kalman filters) as well as methods developed specifically for thruster failures (maximum likelihood and motion-based FDI). From this review, it is concluded that thruster FDI is a solved problem. The motion-based FDI algorithm provides fast detection and isolation of thruster failures and has been proven in space on the SPHERES testbed [29].

While thruster failure recovery has also been addressed in the literature, there is a gap that has not been addressed directly. The most common technique for thruster failure recovery is through the use of a reconfigurable control allocator, typically used for overactuated spacecraft. Techniques for controlling the attitude of underactuated spacecraft have also been developed with less than three control torques. In addition, translational control techniques have been developed to avoid collisions and applied to failure scenarios during docking. However, techniques for controlling the attitude and position underactuated spacecraft have not been discussed in the literature. This is because attitude is normally independent of position. Satellites mostly control their attitude to point to targets to collect measurements, communicate to ground stations, and collect energy from the Sun. Translational control is activated infrequently and often in an open-loop fashion as it is only required during specific times such as orbit insertion and stationkeeping. Therefore, attitude and position can be treated independently. For satellites in a formation or cluster, however, attitude and position need to be controlled constantly to maintain the formation and avoid collisions. A control system that is able to stabilize the spacecraft about a certain attitude and position in the event of thruster failures has not been developed previously.
Chapter 3

Spacecraft Model

Before developing a control system that is able to handle thruster failures, a dynamic model of the spacecraft must first be developed and analyzed. This chapter is split into three main sections. Section 3.1 provides background material on rigid-body dynamics, which forms the basis for the equations of motion of a thruster-controlled spacecraft given in Section 3.2. Finally, Section 3.3 analyzes the controllability of a representative spacecraft described in Section 1.2, revealing the important characteristics about the spacecraft. Standard linear controllability analysis shows that the linearized system is not controllable in the event of thruster failures. A non-linear controllability test is applied to the spacecraft, showing that the presented representation of the system is also not small-time locally controllable (STLC) in the event of thruster failures. In addition, this analysis shows that the system, with failed thrusters, is underactuated and therefore second-order nonholonomic. Knowing these properties will become useful in the design of a reconfigurable controller for this system.

3.1 Rigid-Body Dynamics

This section provides a brief overview of rigid-body dynamics, the basic equations of motion of a spacecraft. Section 3.1.1 provides a description of reference frames and rotations, a fundamental concept for describing concepts such as spacecraft attitude.
Sections 3.1.2 and 3.1.3 derive the basic kinematic and kinetic equations of motion for the attitude and translational dynamics of a spacecraft.

### 3.1.1 Reference Frames & Rotations

Reference frames are useful for expressing vectors (a mathematical abstraction with a magnitude and direction) as concrete quantities that can be manipulated. A vector can be represented in a particular reference frame as a linear combination of the unit vectors, forming the axes, of the reference frame. The two reference frames that will be used extensively are the inertial frame, $\mathcal{F}_I$, and the body frame, $\mathcal{F}_B$.

The inertial frame is a special frame of reference in which Newton’s laws of motion apply, without modification. Many frames of reference are not truly inertial frames, but can be approximated as such when the fictitious forces due to the use of a non-inertial frame of reference are negligible. The axes of the inertial frame are denoted by three dextral (right-handed), orthonormal unit vectors $\hat{i}_1$, $\hat{i}_2$ and $\hat{i}_3$. The body frame is fixed relative to the spacecraft’s geometry with axes $\hat{b}_1$, $\hat{b}_2$ and $\hat{b}_3$. These unit vectors are referred to as basis vectors. A rigorous, mathematical definition of reference frames using vectors (useful for deriving many of the equations presented) is recommended for further reading [30].

Column matrices are used to collect the *components* of a vector expressed in a particular reference frame. In other words, a column matrix is a vector expressed in a particular reference frame. It is often necessary to convert a column matrix from one reference frame to another. Conversion of a column matrix in $\mathcal{F}_B$, $c_B \in \mathbb{R}^3$, to a column matrix in $\mathcal{F}_I$, $c_I \in \mathbb{R}^3$, is done by left multiplying it with a rotation matrix, $\Theta \in \mathbb{R}^{3 \times 3}$, that transforms column matrices from $\mathcal{F}_I$ to $\mathcal{F}_B$:

$$c_B = \Theta c_I \quad (3.1)$$

This rotation matrix is also called a direction cosine matrix because each element in $\Theta$ is $\Theta_{ij} = \cos \theta_{ij}$, where $\theta_{ij}$ is the angle between $\hat{i}_i$ and $\hat{b}_j$. A useful property of this
matrix, since it is orthonormal, is that its inverse is its transpose:

$$\mathbf{\Theta}^{-1} = \mathbf{\Theta}^T.$$  \hfill (3.2)

Thus, to transform a column matrix from $\mathcal{F}_B$ to $\mathcal{F}_I$,

$$\mathbf{c}_I = \mathbf{\Theta}^T \mathbf{c}_B.$$

(3.3)

The direction cosine matrix, since it relates two reference frames, can be used to represent the attitude of a spacecraft. However, the attitude of a spacecraft can be expressed with as few as three variables compared to the nine in a direction cosine matrix. Euler angles, a parameterization using three simple rotations, is commonly used because it is easy to visualize. However, this parameterization has geometric singularities that cause the loss of a degree of freedom or ‘gimbal lock’ at certain angles depending on the sequence of rotations. This also leads to kinematic singularities where Euler angle rates can become very large at these angles for relatively small angular rates. While it is possible to avoid these singularities, for example by switching between different rotation sequences when a singularity is approached, the complexity greatly outweighs simply using another parameterization. While there are other attitude parameterizations, the one that shall be used from here on is the unit quaternion or Euler parameters.

The unit quaternion has gained significant use in spacecrafts due to its lack of geometric and kinematic singularities, its ease of use for multiple successive rotations, and smaller computational load since it has the lowest number of parameters whose kinematic equations do not contain trigonometric functions (see Section 3.1.2 for more information on the quaternion kinematic equations). The quaternion contains four parameters,

$$\mathbf{q} \triangleq \begin{bmatrix} q_1 & q_2 & q_3 & q_4 \end{bmatrix}^T.$$  \hfill (3.4)

These parameters are derived from Euler’s Theorem, which loosely states that any rotation can be described by a rotation of angle, $\theta$, about some axis, $\vec{a}$. This axis is
invariant in both reference frames (i.e., $\mathbf{a} = \Theta \mathbf{a}$) and therefore the arrow denoting a vector can be dropped without ambiguity since it is equally valid in both reference frames. The first three parameters of a quaternion are defined as,

$$q_{13} = \begin{bmatrix} q_1 & q_2 & q_3 \end{bmatrix}^T \triangleq \mathbf{a} \sin \frac{\theta}{2}$$  \hspace{1cm} (3.5)

and the fourth is defined as,

$$q_4 \triangleq \cos \frac{\theta}{2}.$$  \hspace{1cm} (3.6)

Because four parameters are being used to parameterize three degrees of freedom, a constraint must be introduced. The constraint is that the quaternion must be unit length,

$$\sqrt{q^T q} = \sqrt{q_1^2 + q_2^2 + q_3^2 + q_4^2} = 1.$$  \hspace{1cm} (3.7)

From here on, the unit quaternion will be referred to simply as the quaternion, and the unit length constraint will be implied. Since the direction cosine matrix is still useful in transforming column matrices, it can be calculated from the quaternion by

$$\Theta(q) = \left( q_4^2 - q_{13}^T q_{13} \right) I_{3 \times 3} + 2 q_{13} q_{13}^T - 2 q_4 q_{13}^T$$  \hspace{1cm} (3.8a)

$$= \begin{bmatrix} q_1^2 - q_2^2 - q_3^2 + q_4^2 & 2(q_1 q_2 + q_3 q_4) & 2(q_1 q_3 - q_2 q_4) \\ 2(q_1 q_2 - q_3 q_4) & -q_1^2 + q_2^2 - q_3^2 + q_4^2 & 2(q_2 q_3 + q_1 q_4) \\ 2(q_1 q_3 + q_2 q_4) & 2(q_2 q_3 - q_1 q_4) & -q_1^2 - q_2^2 + q_3^2 + q_4^2 \end{bmatrix}$$  \hspace{1cm} (3.8b)

where the $\times$ superscript denotes the $3 \times 3$ the skew-symmetric cross-product matrix associated with the $3 \times 1$ column matrix. In general,

$$\mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \Rightarrow \mathbf{a}^\times \triangleq \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix}.$$  \hspace{1cm} (3.9)

For attitude estimation and control, it is useful to define an error quaternion that

\footnote{The axis of rotation is, therefore, also referred to as the eigenaxis.}
represents a rotation from one attitude to another. The rotation from a given reference quaternion, \( q_r \), to the current quaternion, \( q_c \), is defined by the error quaternion \[ e \]

\[ e = \begin{bmatrix} q_{c13} q_{r13} + q_{r4} q_{c13} - q_{c4} q_{r13} \\ q_{c13} \cdot q_{r13} + q_{c4} q_{r4} \end{bmatrix} = \begin{bmatrix} q_{r4} & q_{r3} & -q_{r2} & -q_{r1} \\ -q_{r3} & q_{r4} & q_{r1} & -q_{r2} \\ q_{r2} & -q_{r1} & q_{r4} & -q_{r3} \\ q_{r1} & q_{r2} & q_{r3} & q_{r4} \end{bmatrix} \begin{bmatrix} q_{c1} \\ q_{c2} \\ q_{c3} \\ q_{c4} \end{bmatrix}. \] \( (3.10) \)

Now that reference frames, rotations between reference frames and the quaternion parameterization have been presented, the dynamic equations that describe the motion of a rigid body will be developed as a model of a spacecraft.

### 3.1.2 Attitude Kinematics & Kinetics

With the description of attitude using direction cosine matrices and quaternions, it is now necessary to describe how attitude changes over time. This is the study of kinematics, which relates angular velocity and attitude as well as velocity and position. If \( \mathcal{F}_B \) is rotating with respect to \( \mathcal{F}_I \), then \( \mathbf{\omega}_{BI} \), is the angular velocity of \( \mathcal{F}_B \) with respect to \( \mathcal{F}_I \). It is most common to express \( \mathbf{\omega}_{BI} \) in \( \mathcal{F}_B \) since ‘strapped-down’ rate gyros can directly measure these quantities. All further references of the angular velocity column matrix, \( \mathbf{\omega} = \begin{bmatrix} \omega_1 & \omega_2 & \omega_3 \end{bmatrix}^T \), is the angular velocity of \( \mathcal{F}_B \) with respect to \( \mathcal{F}_I \), expressed in \( \mathcal{F}_B \).

Using this idea of angular velocity and a quaternion, \( q \), representing a rotation from \( \mathcal{F}_B \) to \( \mathcal{F}_I \), the kinematic equation for attitude can be expressed as \[ (30, 34) \]

\[ \dot{q} = \frac{1}{2} Q(q) \omega = \frac{1}{2} \Omega(\omega) q = \frac{1}{2} \begin{bmatrix} q_4 \omega_1 - q_3 \omega_2 + q_2 \omega_3 \\ q_3 \omega_1 + q_4 \omega_2 - q_1 \omega_3 \\ -q_2 \omega_1 + q_1 \omega_2 + q_4 \omega_3 \\ -q_1 \omega_1 - q_2 \omega_2 - q_3 \omega_3 \end{bmatrix}. \] \( (3.11) \)
where $\Omega(\omega)$ and $Q(q)$ are defined as

\begin{align}
\Omega(\omega) &= \begin{bmatrix} -\omega^x \omega^z \omega^y \omega \end{bmatrix} \\
Q(q) &= \begin{bmatrix} q^x \times q^3 + q^4 I \\
-\omega^z \end{bmatrix}.
\end{align}

When Equation 3.11 is integrated numerically, the quaternion will no longer satisfy the unit constraint of Equation 3.7 after a few integration steps due to rounding errors. The quaternion can be normalized periodically to satisfy this constraint \[35\].

Knowing the kinematics involved with quaternions, the kinetics, or how this motion arises due to forces, can now be studied. The kinetic equations can be derived from Euler's law:

\[ \vec{\dot{h}} = \vec{\tau} \]

where $\vec{\dot{h}}$ is angular momentum of the center of mass and $\vec{\tau}$ is the sum of the external torques applied to the rigid body. It is important to note that the dot represents the time derivative with respect to an inertial frame. Therefore, when the derivative is taken with $\vec{h} = J\omega$ and $\vec{\tau}$ expressed in $\mathcal{F}_B$, an extra term appears:

\[ \vec{\dot{h}} + \omega \times h = \tau \]

where $J \in \mathbb{R}^{3 \times 3}$ is the second moment of inertia or the inertia matrix about the center of mass expressed in $\mathcal{F}_B$. Assuming that $J$ is constant, this equation becomes

\[ \dot{\omega} = -J^{-1} \omega \times J \omega + J^{-1} \tau. \]

This equation, along with the Equation 3.11, can be integrated to determine the angular motion of a rigid body subject to torques.

---

\[2\]These two definitions, $\Omega$ and $Q$, are given simply to show their equality.
3.1.3 Translational Kinematics & Kinetics

In addition to the attitude dynamics, the translational dynamics of a rigid body must also be described. The position, \( \mathbf{r} \), of a rigid body is a vector from the origin of \( \mathcal{F}_I \) to the origin of \( \mathcal{F}_B \). The kinematic equations relating the position and velocity, \( \mathbf{\dot{v}} \), of a rigid body is simply

\[
\mathbf{\dot{r}} = \mathbf{\dot{v}}. 
\] (3.16)

When \( \mathbf{r} \) and \( \mathbf{\dot{v}} \) are expressed in \( \mathcal{F}_I \), this equation becomes

\[
\mathbf{\dot{r}} = \mathbf{\dot{v}}. 
\] (3.17)

The kinetic equations can be derived from Newton’s Second Law:

\[
\mathbf{\dot{p}} = \mathbf{\dot{f}} 
\] (3.18)

where \( \mathbf{\dot{p}} = m \mathbf{\dot{v}} + \mathbf{\omega} \times \mathbf{c} \) is the linear momentum of a rigid body, \( m \) is the mass of the rigid body, \( \mathbf{c} \) is the first moment of inertia with respect to the origin of \( \mathcal{F}_B \) and \( \mathbf{\dot{f}} \) is the net external force acting on the rigid body. Assuming that the origin is the center of mass (\( \mathbf{\dot{c}} = 0 \)) and \( m \) is constant, the equation of motion expressed in \( \mathcal{F}_I \) becomes

\[
\mathbf{\dot{v}} = \frac{1}{m} \mathbf{\dot{f}} 
\] (3.19)

3.2 Equations of Motion of a Thruster-Controlled Spacecraft

With the equations of motion of a rigid body given in Equations 3.11, 3.15, 3.17 and 3.19, a model of a thruster-controlled spacecraft can be developed. The full six-degree-of-freedom (6DOF) model is given in Section 3.2.1 which models the dynamics of a

---

3 This may seem trivial but it is important to note that the dot is the time derivative with respect to an inertial frame. Thus, if \( \mathbf{r} \) were expressed in \( \mathcal{F}_B \), an \( \mathbf{\omega} \times \mathbf{r} \) term would appear, similar to Equation 3.14.
thruster-controlled spacecraft with three translational and three rotational degrees of freedom. In addition, a reduced three-degree-of-freedom (3DOF) model is given in Section 3.2.2 which models the dynamics of a spacecraft restricted to a plane with two translational degrees of freedom and one rotational degree of freedom. This 3DOF model is useful for exploring concepts in a simpler setting before jumping into the full six-degree-of-freedom model.

3.2.1 Six-Degree-of-Freedom Model

Thrusters are modeled as a body-fixed force that is applied to a set location on the spacecraft. The lever arms and thrust directions of all $m$ thrusters\footnote{Because $m$ refers to mass as well as the number of thrusters, the distinction must be made by context.} can be arranged in matrices of size $3 \times m$. The thruster lever arms are given by

$$ L = \begin{bmatrix} l_1 & l_2 & \cdots & l_m \end{bmatrix} \quad (3.20) $$

where $l_i \in \mathbb{R}^3$ represents the column matrix describing the lever arm of the $i^{th}$ thruster in $\mathcal{F}_B$. The thrust directions are given by

$$ D = \begin{bmatrix} d_1 & d_2 & \cdots & d_m \end{bmatrix} \quad (3.21) $$

where $d_i \in \mathbb{R}^3$ represents the column matrix describing the direction of the $i^{th}$ thruster in $\mathcal{F}_B$. The force provided by each thruster is given by a column matrix $u \in [0, u_{max}]^m$. It is important to note that these thruster forces are \textit{unilateral}, meaning that they cannot produce negative thrust. The full equations of motion of a thruster-controlled spacecraft are summarized below.

$$ \dot{r} = v \quad (3.22a) $$

$$ \dot{v} = \frac{1}{m} \Theta^T(q) D u \quad (3.22b) $$
\[ \dot{q} = \frac{1}{2} \Omega(\omega)q \]  
\[ \dot{\omega} = -J^{-1}\omega \times J \omega + J^{-1}Lu \]

One important feature of Equation (3.22b) is that the effect of the thrusters must be rotated from \(F_B\) to \(F_I\) with \(\Theta^T\) given in Equation (3.8). The definitions of the variables of Equations (3.22a)–(3.22d) are summarized in Table 3.1. The main assumptions of these equations is that the origin of \(F_B\) is at the center of mass of the spacecraft, and that the spacecraft can be treated as a rigid body (no flexibility/slosh in the spacecraft).

<table>
<thead>
<tr>
<th>Variable</th>
<th>Size</th>
<th>Reference Frame</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(r)</td>
<td>(3 \times 1)</td>
<td>(F_I)</td>
<td>Position of the origin of (F_B) relative to (F_I)</td>
</tr>
<tr>
<td>(v)</td>
<td>(3 \times 1)</td>
<td>(F_I)</td>
<td>Velocity of the origin of (F_B) relative to (F_I)</td>
</tr>
<tr>
<td>(q)</td>
<td>(4 \times 1)</td>
<td>-</td>
<td>Rotation from (F_I) to (F_B)</td>
</tr>
<tr>
<td>(\omega)</td>
<td>(3 \times 1)</td>
<td>(F_B)</td>
<td>Angular velocity of (F_B) relative to (F_I)</td>
</tr>
<tr>
<td>(\Theta)</td>
<td>(3 \times 3)</td>
<td>(F_B)</td>
<td>Rotation matrix representation of (q)</td>
</tr>
<tr>
<td>(\Omega)</td>
<td>(4 \times 4)</td>
<td>(F_B)</td>
<td>Kinematic quaternion matrix</td>
</tr>
<tr>
<td>(m)</td>
<td>(1 \times 1)</td>
<td>-</td>
<td>Spacecraft mass</td>
</tr>
<tr>
<td>(J)</td>
<td>(3 \times 3)</td>
<td>(F_B)</td>
<td>Spacecraft inertia about the center of mass</td>
</tr>
<tr>
<td>(L)</td>
<td>(3 \times m)</td>
<td>(F_B)</td>
<td>Matrix of thruster lever arms</td>
</tr>
<tr>
<td>(D)</td>
<td>(3 \times m)</td>
<td>(F_B)</td>
<td>Matrix of thrust directions</td>
</tr>
</tbody>
</table>

### 3.2.2 Three-Degree-of-Freedom Model

The 3DOF model represents the motion of a spacecraft restricted to a plane. The spacecraft has two translational degrees of freedom, forming the plane, and one rotational degree of freedom normal to said plane. The 3DOF model has a very similar structure to the 6DOF model and the same assumptions. Position, \(r\), and velocity, \(v\), as well as the columns of the thruster direction matrix, \(D\), are reduced from the domain \(\mathbb{R}^3\) to \(\mathbb{R}^2\). Attitude, \(q\), angular rate, \(\omega\), and the columns of the thruster lever arm matrix, \(L\), and are reduced from the domain \(\mathbb{R}^3\) to \(\mathbb{R}\). Attitude is now an angle in the range \([-\pi, \pi)\) describing the rotation of from \(F_I\) to \(F_B\). The corresponding
rotation matrix is given by

\[ \Theta(q) = \begin{bmatrix} \cos(q) & \sin(q) \\ -\sin(q) & \cos(q) \end{bmatrix} \] (3.23)

With these new definitions, the 3DOF model of a thruster-controlled spacecraft is summarized below.

\[ \dot{r} = v \] (3.24a)

\[ \dot{v} = \frac{1}{m} \Theta^T(q) Du \] (3.24b)

\[ \dot{q} = \omega \] (3.24c)

\[ \dot{\omega} = \frac{1}{J} Lu \] (3.24d)

For convenience, a summary of the variables are summarized in Table 3.2.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Size</th>
<th>Reference Frame</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r )</td>
<td>( 2 \times 1 )</td>
<td>( F_I )</td>
<td>Position of the origin of ( F_B ) relative to ( F_I )</td>
</tr>
<tr>
<td>( v )</td>
<td>( 2 \times 1 )</td>
<td>( F_I )</td>
<td>Velocity of the origin of ( F_B ) relative to ( F_I )</td>
</tr>
<tr>
<td>( q )</td>
<td>( 1 \times 1 )</td>
<td>-</td>
<td>Rotation from ( F_I ) to ( F_B )</td>
</tr>
<tr>
<td>( \omega )</td>
<td>( 1 \times 1 )</td>
<td>( F_B )</td>
<td>Angular velocity of ( F_B ) relative to ( F_I )</td>
</tr>
<tr>
<td>( \Theta )</td>
<td>( 2 \times 2 )</td>
<td>-</td>
<td>Rotation matrix representation of ( q )</td>
</tr>
<tr>
<td>( m )</td>
<td>( 1 \times 1 )</td>
<td>-</td>
<td>Spacecraft mass</td>
</tr>
<tr>
<td>( J )</td>
<td>( 1 \times 1 )</td>
<td>( F_B )</td>
<td>Spacecraft inertia about the center of mass</td>
</tr>
<tr>
<td>( L )</td>
<td>( 1 \times m )</td>
<td>( F_B )</td>
<td>Matrix of thruster lever arms</td>
</tr>
<tr>
<td>( D )</td>
<td>( 2 \times m )</td>
<td>( F_B )</td>
<td>Matrix of thrust directions</td>
</tr>
</tbody>
</table>

### 3.3 Controllability

Now that the model for a thruster-controlled spacecraft has been developed, it is useful to derive the controllability of this system, which will give valuable insight into its complex nature. The idea of controllability does invoke an intuitive definition, however the exact definition must be clear from a mathematical standpoint. A system is controllable if for all states \( x_i \) and \( x_f \) there exists a finite-length control input
trajectory, \( u(t) : [0, T] \rightarrow \mathcal{U} \), such that the solution to the initial value problem

\[
\dot{x} = f(x, u), \quad x(0) = x_i
\]

has the solution \( x(T) = x_f \). Note that \( \mathcal{U} \) simply denotes an admissible set of control inputs. Put simply, a system is controllable if there is a feasible control input trajectory that can drive the system from any initial state to any final state.

Despite its fundamental role in control theory, a completely general set of conditions for controllability of nonlinear systems does not currently exist [36]. For the limited case of linear, time-invariant systems, there is a simple method of determining controllability. This will be discussed in Section 3.3.1 along with the limitations explaining why this analysis does not provide a complete controllability results. Next, a limited nonlinear controllability analysis will be discussed in Section 3.3.2 along with the derived small-time local controllability (STLC) of this system.

### 3.3.1 Linear Time-Invariant Controllability

Linear, time-invariant (LTI) systems have been studied extensively and there exists a simple test for determining the controllability of an LTI system. The dynamics of any LTI system can be expressed as

\[
\dot{x} = Ax + Bu
\]

where \( x \in \mathbb{R}^n \) is the state, \( u \in \mathbb{R}^m \) is the control inputs, \( A \in \mathbb{R}^{n \times n} \) is the dynamics matrix, and \( B \in \mathbb{R}^{n \times m} \) is the input matrix. Controllability of LTI systems can easily be determined by calculating the controllability matrix

\[
M_c = \begin{bmatrix} B & AB & A^2B & \cdots & A^{n-1}B \end{bmatrix}.
\]

If \( M_c \) is full rank, \( \text{rank}(M_c) = n \), the system is controllable [37].

This test for determining controllability cannot be used directly on the system given by Equations 3.22a through 3.22d since it is nonlinear. A system with dynamics
expressed as

\[ \dot{x} = f(x, u) \]  
(3.28)

must first be linearized about a certain state, \( x_e \), and control input, \( u_e \) by performing a Taylor series expansion of \( f \) about \( x_e \) and \( u_e \). Using only the first-order terms in the expansion, the linearized dynamics can be written as

\[
\delta \dot{x} = \begin{bmatrix}
\frac{\partial f_1}{\partial x} \big|_{x_e, u_e} & \cdots & \frac{\partial f_n}{\partial x} \big|_{x_e, u_e} \\
\vdots & \ddots & \vdots \\
\frac{\partial f_1}{\partial u} \big|_{x_e, u_e} & \cdots & \frac{\partial f_n}{\partial u} \big|_{x_e, u_e}
\end{bmatrix} \delta x + \begin{bmatrix}
\frac{\partial f_1}{\partial u} \big|_{x_e, u_e} & \cdots & \frac{\partial f_n}{\partial u} \big|_{x_e, u_e}
\end{bmatrix} \delta u
\]  
(3.29)

While the system can be linearized, there are a few issues with this approach. The first issue is that the thruster input forces are unilateral, rendering a simple rank check invalid. As a simple example, take the following double-integrator system,

\[
\dot{x} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} x + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u, \]  
(3.30)

where \( u \in [0, \infty) \) is a unilateral control input\(^5\). Physically, this system can represent a spacecraft, with a single thruster, whose motion is restricted to one translational degree of freedom. Obviously, the system is not controllable since it can only fire its thruster in a single direction and can therefore only accelerate in a single direction. To be controllable, the spacecraft would need two thrusters pointing opposite directions, which effectively eliminates the unilateral constraint on the control input. While this conclusion is relatively straightforward, determining this mathematically is more challenging. The controllability matrix for this system is

\[
M_c = \begin{bmatrix} B & AB \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}. \]  
(3.31)

If the rank condition is blindly applied to this matrix, one would incorrectly conclude that the system is controllable since the matrix is full rank. This is because the

---

\(^5\)For simplicity, the control input here is not limited to a maximum value.
unilateral constraints do not enter into this rank test. A slight modification of the rank test, however, can account for these constraints. Another way of expressing the rank condition of the controllability matrix, \( M_c = [c_1 \ c_2 \ \ldots \ c_m]^T \), is that

\[
\{\alpha_1 c_1 + \alpha_2 c_2 + \ldots + \alpha_m c_m | \alpha_1, \ldots, \alpha_m \in \mathbb{R}\} = \mathbb{R}^n. \tag{3.32}
\]

This is simply stating the condition that the span of the columns of the controllability matrix is the full \( n \)-dimensional space. To include the unilateral constraints, this condition is simply modified to a “positive” span,

\[
\{\alpha_1 c_1 + \alpha_2 c_2 + \ldots + \alpha_m c_m | \alpha_1, \ldots, \alpha_m \in [0, \infty)\} = \mathbb{R}^n. \tag{3.33}
\]

Returning to the example of a spacecraft with one translational degree of freedom, it can be seen that the positive span of the columns of the \( M_c \) is the space \([0, \infty)^2\). Therefore, one can correctly conclude that the spacecraft is uncontrollable since the positive span is not \( \mathbb{R}^2 \). The system would be controllable, however, with a second thruster that changes the controllability matrix to

\[
M_c = \begin{bmatrix}
0 & 0 & 1 & -1 \\
1 & -1 & 0 & 0
\end{bmatrix}. \tag{3.34}
\]

While this solves the issue of properly dealing with unilateral control inputs, there is another issue that makes this LTI controllability analysis insufficient for determining controllability of the full nonlinear system. The issue is that the linearization “freezes” the locations and directions of the thrusters. For the 3DOF model given by Equations 3.24a through 3.24d, if we apply the linearization given by Equation 3.29 about the origin (\( x_e = 0 \) and \( u_e = 0 \)), the dynamics remain largely unchanged except Equation 3.24b becomes

\[
\dot{v} = \frac{1}{m} \Theta^T(q_e) Du. \tag{3.35}
\]

This means that, for the linearized dynamics, the thruster directions remain constant since the rotation matrix has become constant. Therefore, the controllability results
are only valid for a single attitude.

To illustrate this further, a 3DOF spacecraft with thruster locations and directions as well as physical properties shown in Figure 3-1 will be used. This system has a controllability matrix,

\[
M_c = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & -1 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 \\
1 & -1 & -1 & 1 & 0 & 0 & 0 & 0
\end{bmatrix}, \tag{3.36}
\]

which by inspection does not have a unilateral span of \( \mathbb{R}^6 \), indicating that the linearized system is not controllable. While this is a perfectly valid result, it does not give complete insight into the full nonlinear system. Closer inspection shows that the first and third rows are all zeros, indicating that the position and velocity in the inertial x-direction cannot be controlled. In reality, a combination of thrusters 1 & 4 or 2 & 3 can rotate the spacecraft to align the thruster directions with the inertial x-axis, allowing the position and velocity in the inertial x-direction to be controlled. This possibility was neglected by this LTI controllability analysis since large-angle maneuvers are not captured in the linearized dynamics.

![Figure 3-1: Thruster locations, directions and physical properties for an example 3DOF spacecraft.](image)

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( m )</td>
<td>1 kg</td>
</tr>
<tr>
<td>( J )</td>
<td>1 kg\cdot m^2</td>
</tr>
<tr>
<td>( D )</td>
<td>\begin{bmatrix} 0 &amp; 0 &amp; 0 &amp; 0 \ 1 &amp; 1 &amp; -1 &amp; -1 \end{bmatrix} m</td>
</tr>
<tr>
<td>( L )</td>
<td>\begin{bmatrix} 1 &amp; -1 &amp; -1 &amp; 1 \end{bmatrix} m</td>
</tr>
</tbody>
</table>

It can be shown that for any (single or multiple) thruster failure on a fully actuated
spacecraft (as opposed to an overactuated spacecraft), the spacecraft becomes LTI uncontrollable. This is because any thruster failure will cause the spacecraft to lose the ability to accelerate in a certain linear and angular direction that depends on the thruster geometry. While this is a very important conclusion in itself, additional controllability tools are necessary to gain further insight into this dynamic system.

### 3.3.2 Small-Time Local Controllability

Nonlinear controllability is considerably more complex than LTI controllability. In general, nonlinear systems can have dynamics of the form $\dot{x} = f(x, u)$. However, the most developed concept of controllability of nonlinear systems is restricted to systems that can be expressed in control-affine form,

$$\dot{x} = f(x) + \sum_{i=1}^{m} g_i(x) u_i \quad (3.37)$$

where $x \in \mathbb{R}^n$ is the state, $f : \mathbb{R}^n \to \mathbb{R}^n$ is the drift vector field, $u_i$ is the $i^{th}$ control input, and $g_i : \mathbb{R}^n \to \mathbb{R}^n$ is the $i^{th}$ control vector field. In this formulation, it can easily be seen that the dynamics of the system are a linear combination of the drift vector field and control vector fields. Controllability can now be viewed in terms of adjusting the coefficients, $u_i$, to steer the system from one state to another.

An interesting property of these vector fields, arising from the nonlinear nature of the system, is that combinations of multiple vector fields can produce motion in directions of the state space that is not possible with a single control input. The flow along a vector field $g$ from state $x_0$ for time $t$, $\phi^t_g(x_0)$, is defined as the solution to the differential equation $\dot{x} = g(x)$ with initial condition $x_0$. It can be shown [38] that the composition of flows along two vector fields, $g_1$ and $g_2$ for an infinitesimally small amount of time, $\epsilon$,

$$x(4\epsilon) = \phi^{-g_2}_\epsilon (\phi^{-g_1}_\epsilon (\phi^{g_2}_\epsilon (\phi^{g_1}_\epsilon (x_0)))) \quad (3.38)$$
can be written as

\[ x(4\epsilon) = x_0 + \epsilon^2 \left( \frac{\partial g_2}{\partial x} g_1(x_0) - \frac{\partial g_1}{\partial x} g_2(x_0) \right) + \mathcal{O}(\epsilon^3) \] (3.39)

where \( \mathcal{O}(\epsilon^3) \) denotes terms on the order of \( \epsilon^3 \) or higher. This result shows that the flow along these vector fields produces a net non-zero motion on the order of \( \epsilon^2 \), ignoring the higher-order terms. This concept is shown in Figure 3-2; the dotted red line shows the net non-zero motion after following the flow of the vector fields. It is important to note that in this formulation, this net non-zero motion is valid only for states in the neighborhood of the initial state, \( x_0 \). Therefore, this concept will be used to define small-time local controllability.

This net non-zero motion can be captured in the definition of the Lie bracket operation as

\[ [g_1, g_2] = \frac{\partial g_2}{\partial x} g_1(x_0) - \frac{\partial g_1}{\partial x} g_2(x_0). \] (3.40)

This operation has a few properties \[39\] that will become useful later:

- **Bilinearity:**
  \[ [ag_1 + bg_2, g_3] = a[g_1, g_3] + b[g_2, g_3] \]
  \[ [g_1, ag_2 + bg_3] = a[g_1, g_2] + b[g_1, g_3] \]

\(^6\)The qualifier “small-time” indicates that it holds for any time greater than zero.

\(^7\)This definition of the Lie bracket operation is a specific definition for defining small-time local controllability. The Lie bracket operation can be defined in any way such that the following three properties hold: bilinearity, skew symmetry, and Jacobi identity.
• **Skew symmetry:** \([g_1, g_2] = -[g_2, g_1]\)

• **Jacobi identity:** 
  \([[[g_1, g_2], g_3] + [[[g_2, g_3], g_1] + [[[g_3, g_1], g_2] = 0]

This Lie bracket operation produces a new vector field that the system can flow along by combining multiple control inputs. To solidify this mathematical description of a Lie bracket, a simple example of a differential-drive system will be provided, originally presented in [39]. A differential-drive system is a wheeled vehicle whose wheel speeds can be controlled independently. It can therefore be thought of as a system with two control inputs: one to move forward or backward and one to turn left or right. A control sequence or trajectory of moving forward, turning left, moving backward, and turning right is shown in Figure 3-3. This control trajectory is actually equivalent to the motion produced by the Lie bracket combination of the two control inputs as the time spent applying each control input approaches zero. In addition, the resulting motion of this control trajectory is actually a sideways motion—a motion not possible with a single control input alone. This shows that the Lie bracket operation can produce an additional direction that the system can move in, when multiple control inputs are combined.

Figure 3-3: Differential drive example showing a series of four motion primitives [39].

The idea that Lie brackets can produce a non-zero vector field, is connected with the idea of nonholonomicity. When a system contains nonintegrable constraints (constraints that cannot be written in the form \(f(q, t) = 0\), where \(q\) is the configuration
space or degrees of freedom and \( t \) is time) it is considered nonholonomic. In the example differential-drive system above, there is a constraint on the system’s velocity since the system cannot slide sideways. Since it is often difficult to directly determine whether a constraint is truly nonintegrable versus one’s lack of ability to determine how to integrate it, one can employ the Frobenius theorem to determine whether a system is nonholonomic or holonomic. The Frobenius theorem states that a system is holonomic or integrable if and only if all Lie brackets are in the span of the original system vector fields \( [39] \). Inversely, a system is nonholonomic if and only if there are Lie brackets that are not spanned by the original system vector fields. Returning to the differential-drive system, since the Lie bracket combination of the two control inputs produced a direction of motion not in the span of the original control vector fields, the system is nonholonomic. Knowing whether a system is nonholonomic or not is important when considering the control system design.

Since the Lie bracket produces a new vector field, further vector fields can be defined recursively (e.g., \( g_3 = [g_1, [g_2, g_1]] \)). The degree of a vector field, \( \delta(\cdot) \), is the number of times the original system vector fields appear in the Lie bracket operation (e.g., \( \delta([g_1, [g_2, g_1]]) = 3 \)). The degree with respect to a vector field, \( \delta_{g_i}(\cdot) \), is the number of times \( g_i \) appears in the Lie bracket operation (e.g., \( \delta_{g_1}([g_1, [g_2, g_1]]) = 2 \)). Intuitively, the degree of a vector field is equivalent to a measure of “speed” of the vector field. Returning to the differential drive example, the sideways motion produced by the Lie bracket operation has a degree of two. It can be seen that this higher degree corresponds to a slower motion. The sideways motion of the differential drive system is like parallel parking a car, which is a much slower motion than just moving forward or backward.

It is also necessary to define “good” and “bad” Lie bracket terms. One minor detail that was excluded from the flow definition of a Lie bracket is that vector fields such as the drift vector field can only be followed in a single direction. For the example of the drift vector field, flow in the opposite direction can only be possible if time were reversed. Lie brackets that exhibit this asymmetry are therefore called “bad.” A Lie bracket, \( \phi \), is bad if \( \delta_f(\phi) \) is odd and \( \delta_{g_i}(\phi) \) even for all \( i \in \{1, \ldots, m\} \). The
exact reasoning behind this definition of a bad Lie bracket is beyond the scope of this thesis. The intuitive reasoning is that Lie brackets that meet these conditions have motions that contain only even exponents of the control inputs, $u_i$, and can therefore only be followed in a single direction\cite{40}. Another article\cite{41} gives an example where brackets that are seemingly bad (e.g., ones that have the drift vector field in them) can have their flows rearranged such that they are not bad. A Lie bracket that is not bad is called “good.” It will be necessary, for controllability, to have these bad Lie brackets be “neutralized” by good brackets. A bad Lie bracket can be neutralized if it can be written as a linear combination of good Lie brackets of lower degree. Intuitively, this means that the illegal motion of a bad Lie bracket can be counteracted by the faster, legal motions of a good Lie bracket.

With the definition of the Lie bracket, the Lie algebra of the system vector fields, $\mathcal{L}(\{f, g_1, \ldots, g_m\})$ can now be defined. The Lie algebra includes all the system vector fields as well as all vector fields that can be produced by a finite number of nested Lie bracket operations\cite{39}. This vector space can be determined by finding an independent basis of vector fields which spans this vector space. Since finding this basis can be a tedious task, there is an approach that can be used to help prune the search space of possible nested Lie bracket operations. This approach uses the Philip Hall basis, which is a sequence obtained through a breadth-first search that prunes all redundant vector fields arising from the skew symmetry and Jacobi identity properties. The Philip Hall basis, $\mathcal{P} = \langle \phi_1, \phi_2, \ldots \rangle$, can be generated by satisfying the following rules\cite{39}:

1. The first $m + 1$ elements of $\mathcal{P}$ are the system vector fields, $\{f, g_1, \ldots, g_m\}$.

2. If $\delta(\phi_i) < \delta(\phi_j)$, then $i < j$.

3. Each $[\phi_i, \phi_j] \in \mathcal{P}$ if and only if the following hold:

   - $\phi_i, \phi_j \in \mathcal{P}$ and $i < j$.
   - Either $\phi_j$ is a system vector field or $\phi_j = [\phi_l, \phi_r]$ for some $\phi_l, \phi_r \in \mathcal{P}$ such that $l \leq i$. 

51
These rules can be developed into an algorithm to generate a Philip Hall basis. This algorithm can terminate once $n$ independent vector fields have been found or all Lie brackets after a certain depth are zero. It is difficult to handle the cases where neither of these termination conditions are met [39].

Now that all of the necessary tools for determining controllability have been developed, the conditions for small-time local controllability (STLC) can be stated. The control-affine system given by Equation 3.37 is STLC from $x_0$ if:

1. The drift velocity field is zero at $x_0$: $f(x_0) = 0$.

2. The Lie Algebra Rank Condition is satisfied by good Lie bracket terms up to degree $i$:
   \[
   \dim(L\{f, g_1, \ldots, g_m\}) = \text{span}\{\phi_1, \ldots, \phi_n | \phi_1, \ldots, \phi_n \text{ are good}\} = n.
   \]

3. All bad Lie brackets of degree $j \leq i$ are neutralized.

4. Inputs are symmetric: $U \in U_\pm$.

These are the sufficient conditions for STLC given in [42] and summarized in [40]. The first condition arises since the drift vector field is a bad Lie bracket. Since it cannot be neutralized by good Lie brackets since it is already of the lowest possible degree, it must be equal to zero. The second and third conditions ensures that the motion produced by all possible combinations of the vector fields spans the full $n$-dimensional space, while all bad Lie brackets that are encountered can be neutralized. The final condition ensures that the inputs are symmetric, meaning the control vector fields can be followed in both directions. The set $U_\pm$ is any control set $U$ that positively spans $\mathbb{R}^m$ [8]. In contrast, the asymmetric control set $U_+ \supset U_\pm$ is any control set that spans $\mathbb{R}^m$. While this is a problem with unilateral control inputs, the conditions for STLC have been generalized, to an extent, for unilateral control inputs [41]. These slightly generalized conditions will not be presented since the two approaches are equivalent in the case when thrusters are paired [9]. This highlights an important limitation of this

---

8 An equivalent check is to see if the origin lies in the interior of the convex hull of the control set.
9 Each thruster has an opposing thruster that produces the exact opposite force and torque.
analysis; when a single thruster has failed, it's corresponding opposing thruster must also be removed from the controllability analysis for this analysis to be valid. This is necessary since if a thruster fires without some set of thrusters that can immediately counteract it, this thruster firing necessarily produces a *global* change in state.

The STLC property is important for a few reasons. First, a system that is STLC implies that it is controllable under a few mild conditions [40]. Second, it means that the system can follow a curve arbitrarily close as long as it hovers around zero-velocity states. This is a useful property for a system where obstacles need to be avoided since the necessary motion to avoid an obstacle can be achieved through a “wiggling” motion. In contrast, a system that is controllable but not STLC may need to go through large state changes to achieve a desired state and can therefore not follow any arbitrary trajectory. STLC is, therefore, a useful property for a spacecraft in a cluster to avoid collisions with other spacecrafts.

The STLC of a 3DOF spacecraft can now be analyzed in detail. While this problem has been studied in the literature [43], STLC was only proven for a specific case of three thrusters with certain geometric properties. Therefore, these results do not provide direct insight into more general cases presented here. The equations of motion for this system, can be written in control-affine form as

\[
\begin{bmatrix}
\dot{r} \\
\dot{v} \\
\dot{\omega}
\end{bmatrix} = \begin{bmatrix}
0 \\
0 \\
0
\end{bmatrix} \mathbf{u}_i + \sum_{i=1}^{m} \mathbf{g}_i(x) \mathbf{u}_i = \begin{bmatrix}
\mathbf{v} \\
\omega
\end{bmatrix} + \sum_{i=1}^{m} \begin{bmatrix}
\frac{1}{m} \Theta^T(q) d_i \\
0
\end{bmatrix} \mathbf{u}_i.
\]

(3.41)

The example spacecraft that will be used for this analysis is a model of the SPHERES spacecraft described in Section 1.2, restricted to a plane. Figure 3-4 shows the thruster locations and directions of the thrusters on the spacecraft. Note that each thruster has an opposing thruster, therefore, for example, thrusters 1 and 4 can be treated as a single thruster that produces the vector field \( g_1 \). The lever arm of thrusters 2, 3, 5 and 6 is \( l \). The first condition, the drift vector field is zero, is met for all zero-
velocity states. The second condition is determined by calculating the Lie Algebra of this system. The Lie Algebra can be created by the span of the following good Lie brackets from the Philip Hall basis:

\[ \mathcal{L} = \text{span}\left(\{g_1, g_2, g_3, [f, g_1], [f, g_2], [f, g_3]\}\right). \quad (3.42) \]

It can be shown that this Lie Algebra has full rank by arranging these vector fields into a matrix,

\[
L = \begin{bmatrix}
0 & 0 & 0 & -\frac{\cos(q)}{m} & \frac{\sin(q)}{m} & \frac{\sin(q)}{m} \\
0 & 0 & 0 & -\frac{\sin(q)}{m} & -\frac{\cos(q)}{m} & -\frac{\cos(q)}{m} \\
\frac{\cos(q)}{m} & -\frac{\sin(q)}{m} & -\frac{\sin(q)}{m} & -\frac{\omega \sin(q)}{m} & -\frac{\omega \cos(q)}{m} & -\frac{\omega \cos(q)}{m} \\
\frac{\sin(q)}{m} & \frac{\cos(q)}{m} & \frac{\cos(q)}{m} & -\frac{\omega \cos(q)}{m} & -\frac{\omega \sin(q)}{m} & -\frac{\omega \sin(q)}{m} \\
0 & 0 & 0 & 0 & -\frac{l}{J} & \frac{l}{J} \\
0 & \frac{l}{J} & -\frac{l}{J} & 0 & 0 & 0
\end{bmatrix}, \quad (3.43)
\]

and calculating the determinant, \( \det(L) = \frac{4l^2}{J^2m^4} \). Since the determinant is non-zero for any \( l \neq 0 \), the Lie Algebra is full rank and the second condition for STLC is met. The third condition is met since the only bad Lie bracket with degree \( \leq 2 \) is the drift vector field, which is zero for zero-velocity states. Since the thrusters are paired, the fourth condition is met. Since all four conditions are met, the system shown in Figure 3-4: Thruster locations and directions for 3DOF SPHERES spacecraft.

![Figure 3-4: Thruster locations and directions for 3DOF SPHERES spacecraft.](image)
3-4 with \( l \neq 0 \) is STLC from any zero-velocity state.

Appendix A shows the full calculations for STLC with various thruster failures for both the 3DOF and 6DOF models. The thruster locations and directions for the 6DOF model are shown in Figure 3-5. The results are summarized in Table 3.3. Note that STLC is always computed for zero-velocity states to satisfy the first condition for STLC.

Figure 3-5: Thruster locations and directions for 6DOF SPHERES spacecraft.

![Thruster locations and directions for 6DOF SPHERES spacecraft.](image)

Table 3.3: Small-time local controllability of SPHERES.

<table>
<thead>
<tr>
<th>3DOF Model</th>
<th>Failed Thrusters</th>
<th>STLC?</th>
<th>6DOF Model</th>
<th>Failed Thrusters</th>
<th>STLC?</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Yes</td>
<td></td>
<td>None</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>1 and/or 4</td>
<td>No</td>
<td></td>
<td>Any</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>2 and/or 5</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 and/or 6</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2 and/or 5) and (3 and/or 6)</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

These results show that for both models, a single thruster failure may cause the spacecraft to lose the STLC property. The fact that the spacecraft is not STLC, means that to steer the spacecraft to a particular position and attitude, global state changes must occur. Feasible trajectory design, especially with obstacles, becomes nontrivial.
It is important to note here that the STLC conditions presented are only *sufficient* conditions. Therefore, even if the conditions are not met, the system may still be STLC. This has been shown through counterexample. For example, [44] showed that a system that does not meet the conditions for small-time local configuration controllability (STLCC), a concept very closely related to STLC, can actually be transformed into a system that does meet the STLCC conditions. While it is possible that a transformation can be found that makes the presented system STLC even in the event of thruster failures, determining if such a transformation exists is beyond the scope of this thesis. Therefore, it has not been proven that an underactuated spacecraft is not STLC. Rather, it can be said that the underactuated spacecraft model presented (excluding any transformations) is not STLC given the set of sufficient conditions used.

Another conclusion that can be drawn from inspection of the basis vectors of the Lie Algebra, is that the spacecraft becomes second-order nonholonomic in the event of any thruster failure. This arises from the fact that when a thruster fails, the spacecraft becomes underactuated, causing a nonintegrable constraint on acceleration. Knowing that the system falls under the class of nonholonomic systems is useful to know when designing the control system.

### 3.4 Summary

This chapter has provided a derivation of the equations of motion of a 3DOF and 6DOF thruster-controlled spacecraft. This was followed by two controllability analyses to determine controllability in the event of thruster failures. First, LTI controllability, modified to handle unilateral control inputs, was used to analyze the spacecraft model. This led to the conclusion that a fully actuated spacecraft with a thruster failure is not LTI controllable. This analysis does not fully capture the system dynamics, however, due to the Jacobian linearization. Therefore, the system was analyzed with Lie brackets to determine if it was STLC. The conclusion is that the presented spacecraft model, under thruster failures, becomes nonholonomic and may lose the
STLC property with a single thruster failure. The next step is to use this information to develop a control system that can be reconfigured in the event of failures.
Chapter 4

Thruster Failure Recovery Techniques

Now that the spacecraft model has been developed and analyzed, techniques to recover from thruster failures can be discussed in the proper context. Section 4.1 will first outline the many challenges involved with designing a control system that is able to handle thruster failures. A survey of typical control system design techniques and why they fail to address these challenges will be discussed. The remaining sections describe the three ways considered in this thesis to control a spacecraft with failed thrusters.

4.1 Control System Design Challenges

There are many challenges to designing a control system for a thruster-controlled spacecraft with failed thrusters. For failed-off thrusters, this is equivalent to designing a control system for an underactuated spacecraft. The main challenges are coupling of the translational and attitude dynamics, multiplicative nonlinearities, and control input saturation. These challenges, typical ways to address these challenges, and why most of these methods are not applicable will be explained in the following sections.
4.1.1 Coupling

In the formulation of the system dynamics presented in Section 3.2.1 and 3.2.2, the control input, $u$, enters the system through both the translational and rotational degrees of freedom. The intuitive interpretation is that any thruster produces both a translational and rotational acceleration (assuming that the thruster direction does not pass directly through the center of mass of the spacecraft). This coupling increases the dimensionality of the system and makes the controller design non-intuitive.

A very common method of handling this issue is to decouple position and attitude by treating the control inputs as generalized forces and torques. This approach transforms the translational dynamics into a set of linear equations, which is easily controlled, independent of the attitude dynamics, with any linear control technique. The attitude dynamics are still nonlinear but there exist many techniques to control these dynamics. A separate module, called a control allocator, must then convert the computed control forces and torques into individual thruster forces. This module can be reconfigured to reallocate control authority from failed thrusters to other operational thrusters. This method is described further in Section 4.2 and will serve as one of the baseline thruster failure recovery techniques.

Another control algorithm that may be employed to virtually any system is called Dynamic Programming (DP) [45]. The idea behind DP is to discretize the time, state space and optionally the control input, then compute the control policy for this discretized grid that minimizes a specified cost function. However, as it was mentioned, coupling of the translational and attitude dynamics increases the dimensionality of the problem. This poses a large problem for DP since it is commonly known that DP suffers from the “curse of dimensionality,” where the size of the optimal policy grows rapidly with the number of dimensions of the problem. The size of the optimal policy for 6DOF spacecraft dynamics can easily be calculated as

$$n = n_t n_v^3 n_\theta^3 n_\omega^3,$$  \hspace{0.5cm} (4.1)

where $n_*$ denotes the number of discrete grid points for time ($t$) and the minimum
state representation of position \((r)\), velocity \((v)\), attitude \((\theta)\), and angular rate \((\omega)\).

Even with a very coarse grid of 10 points for each variable, \(n = 10^{13}\). Assuming the control policy is stored as a 13-dimensional array of single precision floating-point numbers, this corresponds to around 291 terabytes of information! This simple example illustrates why the DP algorithm, while perfectly viable in theory, is intractable in practice.

### 4.1.2 Multiplicative Nonlinearities

The equations of motion exhibit several terms with multiplicative nonlinearities. A multiplicative nonlinearity is defined here as a term with state and/or control variables multiplied together (e.g., \(\frac{1}{m} \Theta^T(q) Du\)).

As shown in Section 3.3.1, the common approach of using the Jacobian to linearize the dynamics about a certain state does not produce useful results since a linearization essentially “freezes” the thruster locations to a particular attitude. This causes the linearized spacecraft to become uncontrollable. This means that any linear control technique will not be able to control the spacecraft.

Another common method to handle nonlinearities is to employ feedback linearization [46]. This technique uses the current state to compute a control to cancel the nonlinearities. As a simple example, take the attitude kinetics,

\[
\dot{\omega} = -J^{-1} \omega \times J \omega + J^{-1} \tau. \tag{4.2}
\]

The control input torque, \(\tau\), can be computed as\(^1\)

\[
\tau = J(J^{-1} \omega \times J \omega + u). \tag{4.3}
\]

Using this control law, the closed-loop dynamics become simply

\[
\dot{\omega} = u. \tag{4.4}
\]

\(^1\)This technique is also referred to as computed-torque control.
It can be seen that through the appropriate choice of feedback, the gyroscopic non-linearities of a system can be canceled. A much simpler, linear system is the result of this process. The problem with this method is that the control input for the spacecraft is unilateral. The control necessary to transform the nonlinear system into a linear system may not be admissible due to the stringent saturations of the control input.

4.1.3 Saturation

A specific type of nonlinearity that is of special concern is the saturation nonlinearity of the control inputs. Since each thruster can only produce a positive force, saturation limits the allowable control inputs to the range \([0, u_{\text{max}}]\). One potential option is to employ the design procedures of saturation control [47]. The approach is to first characterize the null controllable region, a set of states where the system can be driven to the origin with an admissible control trajectory. Then feedback laws can be defined that are valid on a large portion of this null controllable region. There are two issues with this approach. First, characterizing the null controllable region (equivalent to determining controllability for a set of states) has been extensively studied for linear systems but is still an open area of research, as noted in Section 3.3.2. Second, many of these design procedures for deriving the feedback control laws are only valid for saturation that is symmetric about the origin [47] or asymmetric saturation that has the origin in its interior [48]. An intuitive reason why these restrictions exist is that explicit feedback control laws (i.e., \(u = Kx\)) produce both positive and negative control inputs depending on the current state. As long as the saturation allows both positive and negative values, the feedback control law would be valid at least for some subset of the state space. When the saturation disallows negative values, feedback control laws are not valid.
4.1.4 Nonholonomicity

As mentioned in Section 3.3.2, an underactuated spacecraft is second-order nonholonomic, meaning that it has non-integrable constraints on acceleration. Through the use of Lie brackets, it can be seen that motion in certain directions can be achieved only through combinations of inputs separated in time. This is an interesting challenge because linear feedback control laws produce a single command input based on the current state of the state of the satellite. They do not produce a sequence of commands that will generate motion in these “restricted” directions.

There are a few techniques that have been developed to control nonholonomic systems that are potential candidates to control an underactuated spacecraft. One of these techniques is to employ the Philip Hall basis directly in the development of the control trajectory [49]. A nonlinear control-affine system, given by Equation 3.37, can be augmented to include the independent vectors of the Philip Hall basis. Using this “extended system,” a control trajectory can be found that employs the control vector fields as well as the generated vector fields through the Lie bracket operation. This control trajectory is then converted into one that only uses the original control vector fields. This approach, however, cannot be used directly because the algorithm has only been developed for driftless control-affine systems that are STLC. While it may be possible to generalize the algorithm to be used on systems with drift, the presented model of an underactuated spacecraft is still not STLC. Therefore, other methods must be employed to control this system.

4.2 Reconfigurable Control Allocation

Now that many of the challenges with developing a control system for an underactuated, thruster-controlled spacecraft have been elucidated, potential ways to solve these challenges will be presented. As mentioned in Section 4.1.1, reconfigurable control allocation is a very common way to control a spacecraft with failed thrusters. This section will describe the decoupling of translational and attitude dynamics, separation of the control law and control allocator, and a few algorithms to implement
a control allocator.

In many mechanical systems, it is often convenient to consider the control inputs as generalized forces and torques instead of individual actuator commands. This allows the control system to be decomposed into two separate modules: a control law and a control allocator. The control law calculates the control forces and torques, which shapes the closed-loop dynamics of the system. The control allocator then determines the necessary actuator commands to produce the control forces and torques. Figure 4-1 illustrates this idea of decomposing a control system into a control law and control allocator. In the event of an actuator failure, the control allocator can be reconfigured to avoid allocating control forces and torques to the failed actuator and reallocate this control to other operational actuators, providing a means to recover from this failure.

Figure 4-1: Decomposition of a controller into a control law and control allocator.

One of the main advantages of using control allocation is that it can greatly reduce the complexity of the controller design. The controller only needs to compute control forces and torques rather than computing each individual actuator’s input. For the system given by Equation 3.22a through Equation 3.22d, assuming that the control inputs are forces expressed in $\mathcal{F}_f$ and torques expressed in $\mathcal{F}_B$, the modified equations of motion are given by Equations 3.17, 3.19, 3.11, and 3.15, repeated below

$$\dot{r} = v$$

(4.5)
\[ \dot{v} = m^{-1} f \]  
\[ \dot{q} = \frac{1}{2} \Omega(\omega)q \]  
\[ \dot{\omega} = -J^{-1} \omega \times J \omega + J^{-1} \tau. \]  

In this formulation, the translational dynamics, Equations 4.5 and 4.6, and the attitude dynamics, Equations 4.7 and 4.8, become decoupled, meaning two controllers (one for translational, one attitude) can be developed independently. In addition, the multiplicative nonlinearities of the translational kinetics have been eliminated. This transforms the translational dynamics into a linear double integrator, which can easily be controlled with a variety of linear control techniques (e.g., lead/lag, PID, LQR, etc.). A simple PD controller is given by

\[ f = -K_P r_e - K_D v_e \]  

where \( r_e \) is the position error, \( v_e \) is the velocity error, and \( K_P \in \mathbb{R}^{3\times3} \) and \( K_D \in \mathbb{R}^{3\times3} \) are specified feedback gain matrices. The gain matrices can be parameterized by the spacecraft mass, \( m \), as well as the closed-loop bandwidth, \( \omega_n \), and damping, \( \zeta \) with the following equations

\[ K_P = \omega_n^2 m I, \]  
\[ K_D = 2\zeta \omega_n m I \]  

where \( I \) is the \( 3 \times 3 \) identity matrix. The rotational equations of motion are nonlinear, however there exist simple controller designs for this system. One version of the PD control law presented in [50] is given by

\[ \tau = -K_P q_{e13} - K_D \omega_e \]  

where \( q_{e13} \) is the vector part of the error quaternion given in Equation 3.10, \( \omega_e \) is the error between the reference angular rate and the current angular rate, and \( K_P \) and \( K_D \) are specified feedback gain matrices. Similar to the translational controller,
the gain matrices can be parameterized by the spacecraft inertia, $J$, as well as the closed-loop bandwidth and damping with the following equations

$$K_P = 2\omega_n^2 J, \quad \text{(4.13)}$$

$$K_D = 2\zeta\omega_n J. \quad \text{(4.14)}$$

This shows that treating the control inputs as forces and torques allows simple, decoupled translational and rotational controllers to effectively shape the closed-loop dynamics of the system, independent of the actuators.

Another advantage of using control allocation is that the controller and control allocator can be designed independently. The control allocator is designed to command the actuators within their operational ranges to produce the requested control forces and torques—it does not change the closed-loop dynamics of the system, assuming that it is feasible to produce the requested control forces and torques. Since, in this control scheme, the control allocator is the only module that is “aware” of the actuators, only the control allocator needs to be modified in the event of an actuator failure. If an actuator fails, the control allocator can reallocate the control from the failed actuator to other operational actuators. While this works if there is some form of redundancy, it is unclear how well this control allocation scheme will work if there is no set of actuators that can directly replace the functionality of the failed actuator. Therefore, control allocation will be used as a baseline thruster failure recovery technique.

The problem of control allocation can be formulated many different ways. The underlying idea behind these methods is to find a control input, $u$, that generates the desired forces and torques, $\begin{bmatrix} f & \tau \end{bmatrix}^T$. These quantities are related through the following equation

$$Bu = \begin{bmatrix} f & \tau \end{bmatrix}^T, \quad \text{(4.15)}$$

where $B \in \mathbb{R}^{6 \times m}$ is the control effectiveness matrix that relates the effect of each control input to the generated force and torque. For the 6DOF spacecraft model
given by Equations 3.22a through 3.22d, the control effectiveness matrix is

\[
B(q) = \begin{bmatrix}
\Theta^T(q) D \\
L
\end{bmatrix}
\] (4.16)

Note that this matrix is a function of the attitude of the spacecraft. An additional complication to solving Equation 4.15 is that the actuators must satisfy the saturation constraint

\[
u_{\text{min}} \leq u \leq u_{\text{max}}. 
\] (4.17)

Formulations of the control allocation problem directly addresses the issue of saturation by explicitly stating it as a constraint in the problem formulation. While this does mean that the control system will produce thruster commands that are feasible, it has other issues. Due to the modularity of the control laws and the control allocator, the two do not necessarily act together to produce the best thruster commands. The control law produces force and torque commands without knowledge of the thruster’s capabilities, therefore these force and torque commands may actually be infeasible. The control allocator does the best it can to produce the commanded force and torque under the constraints of the thruster commands. The closed-loop stability created by the control law relies on the control allocator producing the commanded force and torque, but the control allocator does not communicate to the control law that it cannot produce the commanded force and torque. Therefore, closed-loop stability with this control system cannot be guaranteed with an underactuated spacecraft. This is because the issue of nonholonomicity is not directly addressed with this control system. It may be necessary to use combinations of control inputs separated in time to produce motion in a certain direction. Nevertheless, reconfigurable control allocation will serve as a baseline thruster failure recovery technique due to its widespread use for overactuated spacecraft.

There are many formulations and solutions to the control allocation problem. Some algorithms have even been specifically designed to handle thruster misalignments as well as failures \[25\]. A good survey of control allocation algorithms is

67
presented in [51]. Because of the large array of algorithms that exist to solve the control allocation problem, they cannot all be presented in this thesis. Instead, two algorithms for implementing a control allocator are presented. The first algorithm, described in Section [4.2.1], requires relatively little computations as it employs a pseudoinverse iteratively. The second algorithm, described in Section [4.2.2], directly solves the quadratic programming problem using an active set method.

### 4.2.1 Redistributed Pseudoinverse

One way to formulate the control allocation problem is as a constrained least-squares minimization:

\[
\begin{align*}
\min_u & \quad \frac{1}{2} u^T u \\
\text{subject to} & \quad Bu = \begin{bmatrix} f \\ \tau \end{bmatrix}^T \\
& \quad u_{\text{min}} \leq u \leq u_{\text{max}}.
\end{align*}
\]

This formulation minimizes the control effort necessary to generate the desired forces and torques while following the saturation constraints of the actuators. This, however, does not explicitly take into account that there may not be a feasible control input, \( u \), to satisfy Equation 4.18b.

The closed-form solution to the least-squares minimization given by Problem 4.18 excluding Equation 4.18c is solved by the pseudoinverse operation. Equation 4.18c is left out of the minimization since the pseudoinverse does not handle the constraints on the actuators. The optimal actuator commands are given by

\[
\begin{align*}
u = B^T(BB^T)^{-1}\begin{bmatrix} f \\ \tau \end{bmatrix}^T = B^\dagger\begin{bmatrix} f \\ \tau \end{bmatrix}^T
\end{align*}
\]

where \( \dagger \) is the Moore–Penrose pseudoinverse operation.

To account for the actuators’ limits, a simple iterative procedure was developed as described in [52] and [51]. Equation 4.19 is used to solve for the unconstrained actuator commands. If any actuator command exceed its limits, it is value is clipped,
the control forces and torques are modified take into account the saturated actuator command, the column of the control effectiveness matrix corresponding to that actuator, \( b_i \), is zeroed, and the actuator commands are resolved with Equation 4.19. This process is repeated until the calculated actuator commands are all within their specified limits. This process is summarized below, with inputs \( B, f, \tau, u_{\text{min}}, \) and \( u_{\text{max}} \).

1: Initialize the following variables.

\[
\mathbf{u}_{\text{sat}} = 0
\]

\[
\begin{bmatrix}
\mathbf{f}_c & \mathbf{\tau}_c
\end{bmatrix}^T \leftarrow \begin{bmatrix}
\mathbf{f} & \mathbf{\tau}
\end{bmatrix}^T
\]

2: repeat

3: Initialize boolean flag to true.

\[
f \leftarrow \text{true}
\]

4: Calculate the actuator commands with Equation 4.19

\[
\mathbf{u} = \mathbf{B}^\dagger \begin{bmatrix}
\mathbf{f}_c & \mathbf{\tau}_c
\end{bmatrix}^T
\]

5: for \( i = 1 \) to \( m \) do

6: if \( u_i \leq u_{i,\text{min}} \) then

7: Set boolean flag to false.

\[
f \leftarrow \text{false}
\]

8: Saturate the \( i^{th} \) actuator.

\[
u_i \leftarrow u_{i,\text{min}}
\]

9: Store the \( i^{th} \) saturated actuator command.

\[
u_{i,\text{sat}} = u_{i,\text{min}}
\]

10: Modify the commanded force and torque.

\[
\begin{bmatrix}
\mathbf{f}_c & \mathbf{\tau}_c
\end{bmatrix}^T \leftarrow \begin{bmatrix}
\mathbf{f} & \mathbf{\tau}
\end{bmatrix}^T - \mathbf{B} \mathbf{u}_{\text{sat}}
\]

11: Zero the \( i^{th} \) column of the \( B \) matrix.

\[
b_i = 0
\]

12: else if \( u_i \geq u_{i,\text{max}} \) then

13: Set boolean flag to false.

\[
f \leftarrow \text{false}
\]

14: Saturate the \( i^{th} \) actuator.
$u_i \leftarrow u_{i,max}$

15: Store the $i^{th}$ saturated actuator command.

$u_{i,sat} = u_{i,max}$

16: Modify the commanded force and torque.

$\begin{bmatrix} f_c \tau_c \end{bmatrix}^T \leftarrow \begin{bmatrix} f \tau \end{bmatrix}^T - Bu_{sat}$

17: Zero the $i^{th}$ column of the $B$ matrix.

$b_i = 0$

18: end if

19: end for

20: until $f$ is true

21: return $u + u_{sat}$

This redistributed pseudoinverse is a simple and effective method for control allocation. It is a fast process that terminates within $m$ iterations. The main drawback, however, is that it does not guarantee that all actuators’ capabilities will be used to their fullest extent. There are certain cases where this algorithm will produce actuator commands that do not exactly produce the commanded forces and torques even though there exists a set of actuator commands that will produce the exact commanded forces and torques (see [51] for an example). Since it is possible that this algorithm will not converge on the exact solution, even if the solution exists, a second, more computationally expensive algorithm will be presented.

### 4.2.2 Active Set Method

To address the issue that the redistributed pseudoinverse may not find the exact solution, a second way to form the control allocation problem as a quadratic programming problem has been developed. The problem formulation is stated as:

$$
\begin{aligned}
\min_u & \quad \frac{1}{2} \left( Bu - \begin{bmatrix} f & \tau \end{bmatrix}^T \right)^T W \left( Bu - \begin{bmatrix} f & \tau \end{bmatrix}^T \right) \\
\text{subject to} & \quad \begin{bmatrix} u_{min} & u_{max} \end{bmatrix}
\end{aligned}
$$

(4.20a)
where $W \in \mathbb{R}^{6 \times 6}$ is a positive definite weighting matrix. This method explicitly allows for the possibility that the actuator commands may not exactly produce the desired forces and torques (i.e., $Bu \neq \begin{bmatrix} f & \tau \end{bmatrix}^T$). The drawback of this formulation is that it does not attempt to minimize the use of the actuators when producing the desired forces and torques.

Since Problem 4.20 is stated as a quadratic programming problem, any quadratic programming algorithm can be used to solve it. One such algorithm is the active set method. This method keeps track of an estimate of the active constraints (equality constraints as well as inequality constraints that are exactly met). This allows the algorithm to convert the problem into a quadratic programming problem with equality constraints and no inequality constraints. This optimization is solved in the subspace defined by the active set of constraints. Inequality constraints are then added or removed from the active set and the process is repeated until the optimal point is found. The exact algorithm is given in Appendix B.2.

4.3 Path Planning

Another method to control a spacecraft with failed thrusters is through the design of a feasible trajectory. The idea is to generate a trajectory that can be tracked by a simple control system (e.g., PD control law and a control allocator) without the use of the failed thrusters.

4.3.1 Piecewise Trajectory

Perhaps the easiest way to design a trajectory is by piecing together simple motions that are known to be feasible. For example, with two body-fixed torques and one body-fixed force, a spacecraft can be rotated in two axes and translated along one axis. For the case of the spacecraft presented in Section 1.2, two torques and one force can be generated in the event of any single thruster failure and up to four thruster failures. With this reduced set of thrusters, one can maneuver a spacecraft from a zero-velocity state to any other zero-velocity state.
The path planning is split into three phases. The first phase aligns the body-fixed force vector with the vector from the initial position to the goal position. This is accomplished by piecing together the two rotation motions to align these two vectors. In the second phase, the translation motion is used to translate the spacecraft from the initial position to the goal position. The third phase uses the two rotation motions again to orient the spacecraft into its goal attitude.

This control architecture, however, has a few weaknesses. Since this technique still uses a control law and control allocator, disturbances may push the spacecraft off of its nominal path. If this happens, the control law may command infeasible forces and torques that the control allocator cannot produce. At this point, it is unclear whether stability of the spacecraft can be guaranteed and the path may need to be redesigned. Also, this technique only works for a limited set of thruster failures. Any single thruster failure may be handled, but a second thruster failure may prohibit the use of this technique since the requisite body-fixed torques and force cannot be produced. Still, it provides a useful comparison against other thruster failure recovery techniques.

4.3.2 Rapidly Exploring Dense Trees

A more general method of path planning is a sampling-based motion planning algorithm called Rapidly Exploring Dense Trees (RDTs) [39]. The basic idea is to grow a tree structure from the initial state to the goal state. This tree can be grown to take into account differential constraints such as the system dynamics such that the path from the initial state (the root vertex of the tree) to any of the vertices in the tree is feasible through some control trajectory. In addition, the tree is able to grow in a way that rapidly explores the state space. If the sampling of the state space is random, the generated tree is referred to as a rapidly exploring random tree (RRT), which is a special case of the RDT.

The algorithm for generating this tree is simple in theory. A high-level algorithm adopted from [39] is given below.
1: Add the initial state to the tree, $G$.

$$G.\text{initialize}(x_i)$$

2: Initialize the index variable.

$$i \leftarrow 0$$

3: while $(x_r \notin X_f)$ do
4: \hspace{1em} Sample the state space.

$$x_s \leftarrow \text{sample}(i)$$

5: \hspace{1em} Find vertex nearest to the sampled state.

$$x_n \leftarrow G.\text{nearest}(x_s)$$

6: \hspace{1em} Plan the control trajectory from the nearest vertex to the sampled state.

$$(u, x_r) \leftarrow \text{plan}(x_s, x_n)$$

7: \hspace{1em} Add the resulting state and control trajectory to the tree.

$$G.\text{add}(x_n, u, x_r)$$

8: \hspace{1em} Increment the index variable.

$$i \leftarrow i + 1$$

9: end while

10: return $G.\text{path}(x_r)$

Many of the algorithmic details are hidden in the six function calls ($\text{initialize}$, $\text{sample}$, $\text{nearest}$, $\text{plan}$, $\text{add}$, $\text{path}$). A few of the functions are common functions for tree structures. The $\text{initialize}$ function adds the initial state to the tree as the root vertex. The $\text{add}$ function adds a new vertex and edge to the tree. The $\text{path}$ function extracts the state and control trajectory from the initial state to the final state. The other functions are more complicated and their actual implementation can greatly impact the performance of the algorithm.

The $\text{sample}$ function picks a state from the state space. In practice, the sampling is often random with every $1/100^{th}$ sample being the final state to bias the search slightly toward the final state [39]. This provides a good balance between exploring new areas of the state space while biasing the search toward the final state without being too “greedy.” The $\text{nearest}$ function determines the vertex in the tree that is “nearest”
to the sampled state. This function hides a considerable amount of complexity for the efficient tree search used and the metric used to determine “nearness.” Some commonly used metrics to determine proximity are the Manhattan norm, Euclidean norm, and infinity norm. Finally, the `plan` function hides the most complexity. This function must find the control trajectory that connects the nearest state with the sampled state, which is essentially a two-point boundary value problem (BVP). The exact solution to this BVP is a challenging problem to solve for nonholonomic systems, therefore it is commonly avoided. Instead, one possible way to approximately solve the BVP is to select a set of motion primitives, which is a subset of the possible control trajectories that one can make from a given state. From this set of primitives, one is selected that will bring the system the closest to the sampled state. This allows the resulting state, $x_r$, to be different from the sampled state, therefore only approximately solving the BVP.

Since the resulting state from the call to the local planning method is not guaranteed to be the exact desired state, it cannot be guaranteed that the final state, $x_f$, will be exactly reached. Therefore, the termination conditions of the RDT algorithm must specify a tolerance about the final state that is acceptable. This is denoted as a set of final states, $X_f$. The size of this tolerance greatly impacts the run time of the algorithm [39]. As the tolerance decreases, the algorithm run time will increase dramatically. Sometimes a tree will also be grown from the goal state backwards in time, however this does not alleviate the issue of not exactly reaching the goal state. This is because the two trees now need to be connected through an exact solution to a BVP.

The challenges of coupling, multiplicative nonlinearities, saturation, and nonholonomicity are trivially handled in this framework. The saturation of the control inputs is explicitly handled by specifying motion primitives that obey this saturation limit. Therefore any planned motion will be produced only through feasible control inputs. If any thrusters fail, the set of motion primitives can be trimmed to exclude the use of the failed thruster. The other three challenges are handled implicitly through the `plan` function. This function has a dynamic model of the system that simulates the
system with these effects. All the states that the planner generates takes the coupling and multiplicative nonlinearities into account.

While RDTs do provide a promising way to generate a feasible state and control trajectory that steers the system from its initial state to the final state, its implementation is beyond the scope of this thesis. It would be interesting to see how this method compares in speed and optimality versus the control technique described in the next section.

4.4 Model Predictive Control

The focus of this thesis is on the development of a Model Predictive Control (MPC) algorithm that is able to handle thruster failures. MPC is a control technique that relies heavily on a model of the dynamics of the system to be controlled. Using this model, it predicts the state and control trajectory that the system will follow. An optimization method is then wrapped around this model to minimize some cost function as well as enforce constraints. This optimization is rerun periodically based on the current estimated state to update the control inputs sent to the actuators to account for disturbances.

A general formulation of the problem statement for MPC can be summarized as a minimization over a finite-time prediction horizon, $t_p$, as

$$\min_{x(t), u(t)} h(x(t_i + t_p), t_i + t_p) + \int_{t_i}^{t_i + t_p} g(x(\tau), u(\tau), \tau) d\tau$$

subject to

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

$$x(t_i) = x_i$$

$$x(t) \in X, \forall t \in [t_i, t_i + t_p]$$

$$u(t) \in U, \forall t \in [t_i, t_i + t_p].$$

The cost function to be minimized, given by Equation 4.21a, is split into two terms. The first term specifies the terminal cost, $h$, which is a cost that depends only on the
final state and time. The second term is the incremental cost, \( g \), that is integrated over the entire trajectory. These two cost terms can be used to specify a metric such as minimizing fuel usage and minimizing the time it takes to reach the goal. The constraints, given by Equations 4.21b and 4.21c ensure that the resulting state and control trajectories follow the system dynamics and the system starts at the initial state. Equation 4.21d allows for the possibility of including state constraints such as obstacles. Finally, the constraint given by Equation 4.21e gives the ability to impose control input constraints such as saturation.

This optimization is essentially an open-loop planning technique based on the current estimated state of the spacecraft, \( x_i \). To handle disturbances or unmodeled effects, this optimization is resolved at some multiple of the control period based on the new state of the spacecraft. If the computational burden is not too great, the optimization can even be performed every control period. Because the prediction time horizon is fixed relative to the current time for periodic calls to the optimization algorithm, MPC is also referred to as receding horizon control.

The problem formulation given by Equations 4.21 is generic enough that it may be solved by a range of optimal control techniques. While there are analytical methods to solve this problem\(^2\) most problems of interest can only be solved numerically. Since Equation 4.21a is a minimization over an uncountably infinite number of variables, it must be converted into a minimization over a finite number of variables. While there are many parameterizations that can be used (e.g., Legendre polynomials), a simple piecewise-constant parameterization is often used. Many modern control systems are implemented digitally, which means there is a natural period of time over which the control signals are held constant. Therefore a piecewise-constant parameterization fits nicely into this framework.

Using this piecewise-constant parameterization, the MPC problem statement can

\(^2\)A few examples are solving the Hamilton-Jacobi-Bellman equation, or using calculus of variations and Pontryagin’s minimum principle [53].
be rewritten as a minimization from time 0 to time $p$ as

$$\min_{x,u} \sum_{j=1}^{p} x_j^T Q x_j + \sum_{j=0}^{p-1} u_j^T R u_j$$

subject to

$$x_{j+1} = f(x_j, u_j), \quad \text{for} \quad j = 0 : p - 1$$

$$x_0 = x_c,$$

$$u_{min} \leq u_j \leq u_{max}, \quad \text{for} \quad j = 0 : p - 1,$$

where $x_i$ and $u_i$ denote the state and control at time $i$, $x \in \mathbb{R}^{np}$ is the state trajectory, $u \in \mathbb{R}^{m(p-1)}$ is the control trajectory, $Q \in \mathbb{R}^{n \times n}$ and $R \in \mathbb{R}^{m \times m}$ are positive definite weighting matrices, $f : \mathbb{R}^{n \times m} \rightarrow \mathbb{R}^n$ denotes the discrete dynamics, $x_c \in \mathbb{R}^n$ is the current estimated state, and $u_{min/max} \in \mathbb{R}^{m}$ are the saturation limits of the control inputs. Note that several simplifications have been made in addition to using a piecewise-constant parameterization. The cost function is now a quadratic function to minimize the state and control variables. A quadratic cost function is typically used because it is easier to handle numerically than absolute value functions [53]. The state constraints have been removed because handling obstacles with this controller is beyond the scope of this thesis. The input constraints have taken on a more specific form called “box constraints.” This places an upper and lower limit on the allowable control values.

This inclusion of saturation constraints directly into the control system is one of the main advantages of this method. Since the saturation constraints appear explicitly in this formulation, underactuation or thruster failures can easily be handled. For example, if the spacecraft has a thruster pair that is able to produce a bilateral force on the spacecraft, it would have the following constraint,

$$u_{min} \leq u \leq u_{max}.$$  

(4.23)

If the thruster that produces the positive force were to fail, the constraint can be

\footnote{This formulation is a regulator since it drives the state to zero.}
easily modified to
\[ u_{\text{min}} \leq u \leq 0. \quad (4.24) \]

The other challenges (coupling, multiplicative nonlinearities, and nonholonomicity) are also handled well in this framework. The predictive nature of the controller (i.e., the fact that the optimization problem looks \( p \) control periods into the future) means that the controller is able to design a trajectory where combinations of control actions separated in time can produce motions that are not feasible to produce instantaneously. Also, the coupling and multiplicative nonlinearities are neatly folded into the constraint given by Equation 4.22b.

Many of the challenges are addressed by including constraints in optimization problem. This really just shifts the burden from designing a control algorithm to designing an optimization algorithm. Assuming that the algorithm used to solve this problem “does its job,” the challenges are surmountable. There are many issues, however, that make designing this optimization algorithm difficult. First, the multiplicative nonlinearities in the dynamics means that Equation 4.22b is a nonlinear constraint. If the dynamics were linear instead, the optimization problem would be a quadratic program. Since there exist fast quadratic programming algorithms such as the one presented in Section 3.2, it is feasible to solve these problems in real time every control period. However, since the dynamics are nonlinear, the problem becomes a nonlinear programming problem, placing the control problem under the subset of Nonlinear Model Predictive Control (NMPC). The issues of stability and optimality of NMPC will be discussed in the following sections.

### 4.4.1 Stability

The necessary “ingredients” for stability using NMPC will be described, based off a survey paper\[54\]. Early versions of MPC did not provide any guarantees of stability. Instead, the weighting matrices, \( Q \) and \( R \), were tuned and tested to provide good performance. Stability was achieved because the focus was on stable, linear systems. This approach of iterative tuning and testing has been criticized in \[55\].
The grace of this method is that if the prediction horizon is chosen to be sufficiently large (i.e., longer than the settling time of the system dynamics), the system is stabilized \[54\]. To avoid this heuristic approach to stabilizing the system, several methods to guarantee stability are given below.

One method to assure stability is through the addition of an equality constraint on the terminal state of the system. This constraint enforces that the state be driven to the origin by the end of the finite-time horizon. This terminal equality constraint can be written as

\[ x_p = 0. \] (4.25)

The stabilizing effect of this equality constraint on linear systems with quadratic cost functions was initially studied in \[56, 57, 58\]. It was later shown using Lyapunov stability analysis that employing this constraint in MPC guarantees stability for both discrete \[59\] and continuous \[60\] systems. This only guarantees stability under the assumption that there exists a feasible solution to Problem 4.22.

One problem with the use of a terminal equality constraint is that its exact satisfaction requires, in theory, an infinite number of iterations to solve. While this can be handled in practice by defining termination conditions with a tolerance to allow approximate satisfaction, a technique has been formalized \[61\] to address this issue rigorously. Instead of enforcing an exact terminal equality constraint, the terminal state can be allowed to be within a certain region,

\[ x_p \in \Omega, \] (4.26)

where \( \Omega \) is the allowable terminal set with the origin in its interior. This set, \( \Omega \), can be calculated by determining the null controllable region using a locally stabilizing feedback control law, \( u = Kx \). The controller can now be thought of as a dual-mode controller, where MPC is used outside of the terminal set and the feedback control law is used inside the terminal set.

In addition to a terminal constraint set, a terminal cost can also be used. One approach by \[62\] is to add a terminal cost that approximates the additional cost to
go to the origin. The advantage of this technique is that the (quadratic) cost function approximates the infinite-horizon cost\footnote{This technique has therefore been called “quasi-infinite horizon” NMPC since it is a finite-horizon control problem with a cost function that approximates the infinite-horizon cost.} which has nice stability and robustness properties. Reference \cite{62} provides a procedure for determining the locally stabilizing feedback control law, terminal penalty matrix, and terminal set.

Unfortunately, to use this terminal constraint set and terminal cost, the linearized system must be stabilizable, which is a slightly less restrictive than controllable. Since it was shown in Section \ref{sec:3.3.1} that the (underactuated) linear system is not controllable (and not stabilizable) due to the saturation constraints, these techniques cannot be employed. Instead, the two options are to include the terminal equality constraint to enforce stability (assuming that there is a feasible solution to the optimization problem) or not include the terminal equality constraint and achieve stability through parameter tuning.

\subsection*{4.4.2 Optimality}

Another issue of concern for the NMPC literature is optimality. The stability arguments presented in the previous section assume that the exact, globally optimal solution to the optimization problem can be found in finite time. For convex programming (i.e., linear or quadratic programming), the exact, globally optimal solution can be found within a finite number of iterations, assuming that there is a feasible solution. For linear MPC, therefore, it is reasonable to assume that the exact optimal solution can be determined in real time. However, nonlinear programming algorithms that must be employed for NMPC, in general, require an infinite number of iterations to converge to a local optimum, that is not guaranteed to be the global optimum. While there are techniques to find the global optimum for nonlinear programming problems (e.g., simulated annealing), they generally do not guarantee finding the global optimum and are computationally expensive.

To solve these two issues, reference \cite{63} has a proof showing that the (local or global) optimal solution to the optimization problem does not need to be found to
ensure stability. Instead, as long as the constraints of the optimization problem are met, including a terminal equality constraint or terminal set constraint, the system is stable. In other words, “feasibility implies stability.” This approach has therefore been called Suboptimal MPC.

4.5 Summary

This chapter has first provided a concise list of the challenges faced when developing a control system that directly controls the thrusters of an underactuated spacecraft. These challenges are the coupling of the translational and attitude dynamics, multiplicative nonlinearities, saturation of the control inputs, and nonholonomicity of the spacecraft. With these challenges in mind, a survey of control techniques has been performed to determine which techniques could possibly be used, and which cannot. Since there are many control techniques that have not been mentioned, it cannot be claimed that an exhaustive survey has been performed. Instead, a survey of the most common or promising approaches has been investigated. Three control techniques (reconfigurable control allocation, piecewise trajectory design, and MPC) have been identified as candidate control systems. A special emphasis is placed on MPC, while the other two techniques will serve as baseline techniques for comparison.
Chapter 5

Model Predictive Control

Implementation Issues

The theory behind MPC has been presented, however it is still not straightforward to implement MPC in a real time algorithm from this theory. There are several issues that must be addressed to actually implementing an MPC algorithm. Section 5.1 discusses how the problem needs to be modified to handle the regulator nature of the formulation and how to treat the attitude variables for the 3DOF and 6DOF models. Section 5.2 describes how the nonlinear programming algorithm has been selected and actually implemented in hardware. Since this algorithm takes time to calculate a control trajectory, Section 5.3 details how this large processing delay can be mitigated instead of introducing a large delay into the control system. As the MPC calculations are constrained to take a certain amount of time, analysis of the stability of this algorithm under premature termination conditions are analyzed in Section 5.4.

5.1 Regulation & Attitude Error

The MPC problem statement presented in Section 4.4 represents a open-loop regulation problem since the cost goes to the minimum value of zero only when the state and control are driven to the origin with positive definite $Q$ and $R$ matrices. Since it
is desirable to be able to drive the system to any feasible state, instead of the origin, the problem formulation must be modified. It is commonly known that a simple variable substitution can transform a regulator into a controller that can drive the system to any state. By defining \( e \in \mathbb{R}^n \), the error between the reference state, \( r \in \mathbb{R}^n \), and the actual state, \( x \in \mathbb{R}^n \), as
\[
e = r - x,
\]
the cost function given in Equation \([4.22a]\) and the terminal equality constraint for stability given by Equation \([4.25]\) can be rewritten to drive the error to the origin instead of the state itself. This substitution can easily be made for the position, velocity and angular velocity state variables.

For quaternions, this arithmetic difference makes less sense to use. While it is true that when the error is driven to the origin, the arithmetic difference between the current quaternion, \( q_c \), and the reference quaternion, \( q_r \), will be driven to zero. The issue is that the arithmetic difference between the reference quaternion and actual quaternion does not have physical meaning. Recall that the error quaternion, \( q_e \), the rotation from the current to the reference quaternion has been defined in Equation \([3.10]\). It should also be noted that the error quaternion, for small rotations, is approximately
\[
q_{e13} \approx \begin{bmatrix} \phi / 2 \\ \theta / 2 \\ \psi / 2 \end{bmatrix}^T,
\]
where \( \phi \), \( \theta \), and \( \psi \) are the Euler angles of roll, pitch and yaw \([50]\). By using the error quaternion instead of the arithmetic difference, there is an associated physical meaning, making the weight tuning process more intuitive. Another modification uses the fact that quaternions have a unit length constraint (Equation \([3.7]\)). Because of this, only the three vector components of the quaternion need to be included in the cost function and the terminal equality constraint. The fourth scalar component will be driven to unity automatically when the vector components are driven to zero.

This handles the attitude error problem for the 6DOF case. For the 3DOF case, the arithmetic difference between the reference and current attitude angle can be used. The issue, however, is that this does not allow “angular flips.” For example, if the
spacecraft undergoes a $2\pi$ attitude change during its maneuvering from one position to another, the error is non-zero. To drive the error to zero, the spacecraft will then execute an unnecessary full $-2\pi$ flip, which wastes time and fuel. To eliminate this issue, the following definition of the attitude error, $\theta_e \in \mathbb{R}$, can be made.

$$\theta_e = \sin(\theta_r - \theta_c), \quad (5.3)$$

where $\theta_r \in \mathbb{R}$ and $\theta_c \in \mathbb{R}$ are the reference and current attitude angles. Through the use of a trigonometric function, the difference between the reference and current attitude can be off by multiples of $2\pi$ and still have an error of zero.

### 5.2 Nonlinear Programming Algorithm

Now that the problem statement has been stated (Problem 4.22) with the terminal equality constraint for stability (Equation 4.25) and the attitude error modifications given in the previous section, the algorithm to solve this problem can be selected. Even though the cost function is quadratic, the inclusion of nonlinear dynamics in Equation 4.22b means that a nonlinear programming algorithm must be used instead of a quadratic programming algorithm. The two most common nonlinear programming algorithms are the Sequential Quadratic Programming (SQP) algorithm and the Interior-Point (IP) methods. Section 5.2.1 describes how SQP was selected over IP methods and Section 5.2.2 describes the implementation of the SQP algorithm.

#### 5.2.1 Selection

The SQP algorithm, as the name suggests, solves quadratic programming problems iteratively. Starting at an initial guess, it approximates the cost function as quadratic and the constraints as linear. The problem is therefore approximated by a quadratic programming problem, which is solved exactly by a quadratic programming algorithm. The solution to this quadratic programming problem provides a step direction, which is added to the initial guess. This new point becomes the next guess,
about which the cost function is approximated as quadratic and the constraints as linear. This process is repeated until termination conditions are met. Further details, including some common modifications (Hessian approximation and Merit-function-based step scaling) are given in Section B.1.

IP methods involve selecting an initial guess that is in the feasible region and using barrier functions to enforce inequality constraints. First, inequality constraints are converted to equality constraints, with the introduction of slack variables that are restricted to be non-negative. The values of the slack variables represent how “close” a point is to the inequality constraints or barriers. A typical barrier function is the natural logarithm since it grows to negative infinity as it approaches zero. Therefore, as a point in the feasible region approaches a barrier, the augmented cost function or Lagrangian approaches infinity. Away from the barrier, the effect of the logarithm has little effect on the cost function. Starting with an initial feasible solution, a step direction is calculated by solving a system of equations and a step scaling factor is calculated by minimizing a merit function with a line search. This is repeated until termination conditions are met. When this inner loop terminates, the barrier functions are scaled so that if the optimal solution is on a barrier, the search can continue closer to the barriers. This outer loop continues until termination conditions are met. Further information on IP methods for nonlinear optimization can be found in [64].

Many articles have been published, which compare the advantages and disadvantages of these two algorithms. One reference in particular [65] compares the performance of these two algorithms on optimal control problems. This reference concludes that SQP, in general, takes less time to converge while IP methods perform better on a certain class of problems (where the active set changes frequently). The implementation of both an SQP and IP algorithm are out of the scope of this thesis, therefore SQP has been chosen as the optimization algorithm due to its faster convergence speed in general.
5.2.2 Implementation

Since it is desired that this algorithm run in real time on the actual spacecraft avionics, a SQP algorithm written in C/C++ is needed. While there are many commercially available SQP algorithms, there are no openly available versions with C/C++ source code to the author’s best knowledge. Even if there were openly available versions, they would likely have to be modified to reduce the code size, handle premature termination, and other issues gracefully. Instead of returning an error, the algorithm should always return a “best guess,” perhaps with a warning.

It was chosen to implement the SQP algorithm using Embedded MATLAB®. This allowed easy coding and testing in the MATLAB® environment and automatic embeddable C code generation. Some difficulties were encountered since coding in Embedded MATLAB® is more restrictive than MATLAB®. For example, only a subset of the available MATLAB® functions are supported\(^1\) and variable-size matrices are not directly supported\(^2\)\(^\text{[66]}\). The complete Embedded-MATLAB®-compliant code can be found in Section B.

This SQP algorithm, sqpAlgorithm, was benchmarked against the MATLAB® fmincon function using the following nonlinear programming problem:

\[
\begin{align*}
\min_{x_1, x_2} & \quad 100(x_2 - x_1^2)^2 + (1 - x_1)^2 \\
\text{subject to} & \quad x_1^2 + x_2^2 \leq 1.5.
\end{align*}
\]

The cost function is the Rosenbrock function \([67]\), a commonly used function test optimization algorithms due to its difficult-to-find global optimum in a steep, parabolic valley. A nonlinear inequality constraint is also imposed, so that the solution must be in a circle of radius \(\sqrt{1.5}\). Figure 5-1 shows the iterations of the SQP algorithm with an initial guess of \(x_{init} = \begin{bmatrix} -1.5 \\ 0 \end{bmatrix}^T\).

Several cases were tested with the M code, compiled MEX file, and compiled

---

\(^1\)The function fmincon, which implements a SQP algorithm, is unfortunately not currently supported by Embedded MATLAB®.

\(^2\)If variable-sized matrices are used, there must be a fixed upper bound on its dimensionality.
C code. The M code and MEX files were run on a 2.2 GHz processor while the compiled C code was run directly on the SPHERES hardware. For future reference, the compiled C SQP algorithm, for this example problem, takes up about 17% of the total available flash memory on the satellite.\textsuperscript{3} The results are summarized in Table 5.1. These performance tests show that the M code runs twice as slow as \texttt{fmincon}. However, when it is compiled into a MEX file, it runs about 30 times as fast as \texttt{fmincon}, a significant improvement in speed. However, when this same code is compiled and loaded into the satellite, it runs 500 times slower than the compiled MEX file. This is unfortunate because the MPC optimization problems take on the order of a second to compute with the compiled MEX files. It would therefore take approximately 500 seconds for the exact same problem to run on SPHERES. This processing delay is unacceptable since the dynamics of SPHERES is much faster than this computation time (most tests using SPHERES are less than 10 minutes total).

\textsuperscript{3}There was a difference of 9656 out of 57344 words in total program size when the call to the SQP algorithm was commented out.
Table 5.1: Comparison of the SQP algorithm versus \texttt{fmincon} run time on an example problem.

<table>
<thead>
<tr>
<th>Case</th>
<th>Avg. Time [s]</th>
<th>Std. Dev. [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{fmincon} (M)</td>
<td>0.057</td>
<td>0.004</td>
</tr>
<tr>
<td>sqpAlgorithm (M)</td>
<td>0.107</td>
<td>0.008</td>
</tr>
<tr>
<td>sqpAlgorithm (MEX)</td>
<td>0.0021</td>
<td>0.0003</td>
</tr>
<tr>
<td>sqpAlgorithm (C)</td>
<td>0.9950</td>
<td>0.0006</td>
</tr>
</tbody>
</table>

Since performing this optimization on the SPHERES hardware is too slow, several workarounds have been developed. For ground testing, a GUI has been developed that interfaces with the satellites through the wireless communication channels, and also interfaces with a MATLAB® engine. Therefore, for ground testing, the satellite can communicate its current state and the fact that it wants to update its scheduled controls to the GUI. The GUI then executes the MEX SQP algorithm using the MATLAB engine and transmits the results back to the satellite. Since this GUI is more of an “experimental” GUI, it has not been released for astronaut use on the ISS. Therefore online MPC cannot be performed on ISS, currently. Instead, simulations can be performed to determine a nominal state and control trajectory. The nominal control trajectory is performed in an open-loop fashion. However, since there are disturbances and the spacecraft dynamics are not perfectly modeled, an online feedback controller is also employed to keep the state close to the expected state trajectory.

5.3 Processing Delay

One of the major assumptions of MPC is that a solution to the optimization problem can be calculated in zero time and executed by the actuators immediately. This is clearly not possible since the solution to the optimization problem requires some processing time. This processing time translates into a delay in the control system. As this delay increases the system performance decreases until the system goes unstable from too much delay. This was not an issue for most of the early implementations of MPC since they were used to control large industrial chemical processes (see, e.g.,
For these applications, the dynamics of the system are very slow relative to the available computing power, making the delay due to processing negligible.

For space systems, the avionics that perform the computations are power, volume and mass limited. This, and the fact that the avionics are typically radiation hardened, severely limit the processing power available for the MPC calculations. Because these processing limitations could limit performance and even cause stability issues, a method has been developed to compensate for this large processing delay. This method is shown pictorially in Figure 5-2. This figure shows two parallel “threads,”

![Figure 5-2: Delay compensation method for mitigating the effects of processing delay.](image)

the control loop and the MPC task. The control loop executes the actuator commands at a fixed rate while the MPC task is allotted some multiple of the control period, \( c \), to perform its calculations. As long as \( c \leq p \), there will always be actuator commands for each control period. First, the MPC calculations are initiated knowing the current state and scheduled controls for the next \( c \) control periods. These scheduled controls are the computed controls from a previous call to the MPC task. If there were no previous calls to the MPC task (i.e., for the first control period), the controls are set to zero. Since the control loop knows that the MPC task will not be complete for \( c \) control periods, it executes the previously computed controls. Meanwhile, the MPC task thread uses the state and scheduled control commands to predict the state of
the spacecraft $c$ control periods from the current control period $k$. This is the crucial component for the delay compensation. The effectiveness of this method relies upon the accuracy of this prediction step. The MPC optimization problem is then solved based on this predicted future state. Some amount of buffer time is left to account for variations in processing time for the optimization algorithm. At control period $k + c$, the updated controls are then implemented in the control loop and the process is repeated.

Another small modification can be made to help alleviate the effect of processing delay. Many, if not all, optimization algorithms require some initial guess of the solution, which is then iteratively updated, which slowly converges to the optimal solution. Since the optimization is periodically being resolved every $c$ control periods, information from a previous solution to the optimization can be used to initialize the current optimization problem. For example, at some control period, the initial guess to the optimization problem can be

$$u_{\text{init}} = \begin{bmatrix} u^T_{\text{prev},c+1} & u^T_{\text{prev},c+2} & \cdots & u^T_{\text{prev},p} & \vartheta^T_{m \times c} \end{bmatrix}^T$$

where $u_{\text{prev}} \in \mathbb{R}^{m(p-1)}$ is the previously computed control trajectory. This gives the optimization algorithm a “warm start” by starting it off in a point closer to the optimal solution.

### 5.4 Feasibility & Guaranteed Stability

From Section 4.4.1 it is known that stability can be guaranteed as long as the constraints of the optimization problem are met (feasibility implies stability). While this is a great result, the problem becomes proving that feasibility is possible. In addition, from Section 5.3 it is known that the optimization algorithm is only given a fixed amount of time to complete. This means that the optimization algorithm may not have been allowed to converge to within the set tolerances. This means that to ensure stability, feasibility must be ensured by the last iteration of the optimization
algorithm. A short analysis of the different constraints of the optimization problem will be performed to determine if it is possible to ensure feasibility.

The first constraint, given by Equation 4.22b, enforces the dynamics of the system. The state trajectory must be feasible, given the dynamics of the system. One way of ensuring that this condition is met is through direct integration. The optimization, instead of being a minimization over the state and control trajectories, \( \min_{x,u} \), it can be restated as a minimization over only the control trajectory, \( \min_u \). Given the control trajectory, \( u \), the corresponding state trajectory can be calculated through any means such as a numerical Euler or Runge-Kutta integration. The second constraint, given by Equation 4.22c, can be easily satisfied by using \( x_c \) as the initial state when performing the forward integration. These constraints are then essentially embedded in the cost function and do not appear explicitly as constraints in the optimization problem statement.

In contrast to this direct integration or “single shooting” method, another approach is to treat each control period separately. Dynamic feasibility is ensured in each of these periods through integration, and additional constraints are added to the optimization problem that ensure that the state at the end of one period is the same as the state at the beginning of the next period. Since direct integration is performed over multiple periods, this method is referred to as “direct multiple shooting.” It turns out that the formulation of the problem in this fashion has numerical benefits (e.g., Hessian of the Lagrangian or augmented cost function is block diagonal) that improve the speed of the algorithm. Dynamic feasibility of the overall state trajectory, however, cannot be guaranteed in the event of a premature termination of the algorithm. Despite this, multiple shooting algorithms have been used for real-time optimization for MPC [69, 70]. While it would be interesting to see the tradeoff between a possible performance boost versus the loss of dynamic feasibility for premature termination, implementation of a sparse NLP algorithm that is able to take advantage of the sparse structure of the matrices is beyond the scope of this thesis.

The real issues arise when studying the last two constraints, the control input saturation constraint given by Equation 4.22d and the terminal equality constraint for
stability given by Equation 4.25. Ensuring that the terminal equality constraint can be met, even with enough time to converge within tolerances, is equivalent to knowing that the system is globally controllable. As stated in Section 3.3.2, determining the global controllability of a nonlinear system is still an open problem. Adding on top of this the discrete finite parameterization of the control trajectory, the (finite) prediction horizon, and the control input saturation constraints, makes satisfaction of this constraint much more difficult to know apriori to the convergence or non-convergence of the optimization algorithm. Given the terminal equality constraint, it cannot be guaranteed that the saturation constraints will be exactly met. Satisfaction of the terminal equality constraints may cause the inequality constraints to be violated, at least temporarily as the algorithm converges. This is because the SQP algorithm allows the step direction to be scaled based on a merit function, as described in Section B.1. This merit function weights the satisfaction of the inequality versus the equality constraints. Therefore, the choice of the relative weighting of these terms in the merit function determines a priority on the satisfaction of the inequality versus equality constraints.

Since the terminal equality constraint and the control input saturation inequality constraints cannot be guaranteed to be met, stability of the MPC algorithm is not guaranteed. Further stability analysis is beyond the scope of this thesis, but this is a interesting topic that can be pursued. There exist proofs on the convergence of optimization algorithms and the convergence of dynamic systems. Some kind of combination of these two fields could produce a way to provide and/or prove stability for this MPC algorithm under premature termination conditions.

5.5 Summary

This chapter has elucidated the many issues that arise when actually implementing the MPC theory into an algorithm operating in real time. It has been shown how to modify the problem statement to handle attitude error in the regulator framework. A nonlinear programming algorithm has been selected, implemented and benchmarked
on an example optimization problem. As this algorithm takes a finite time to finish, a novel method to eliminate this delay has been developed to avoid reduced performance or instability resulting from this delay. In light of the actual processing requirements and the fact that the optimization algorithm may be terminated before convergence, feasibility of the constraints have been analyzed and it has been determined that the MPC algorithm cannot guarantee stability.
Chapter 6

Simulation & Hardware Testing

Results

The three thruster failure recovery techniques selected in Chapter 4 have been implemented in simulation and in the actual SPHERES testbed. Most of the 6DOF tests are simulations due to the limited availability of the SPHERES hardware on the ISS. Even if all the tests did run on the ISS, there would still be untested aspects of the MPC algorithm such as the delay due to performing the optimization online. To test the MPC algorithm online with delay compensation, the 3DOF SPHERES testbed at MIT was used.

6.1 Six-Degree-of-Freedom Results

The control algorithms were tested by performing the same “representative” maneuver. For linear systems, the step response of a system is a useful maneuver to perform because it provides many useful metrics to characterize the system (e.g., rise time, settling time, maximum percent overshoot, etc.). Characterization of this step response gives a good idea of how the system would react in many different situations. For this reason, a step command in position was chosen as the representative maneuver. The spacecraft starts at a position of \( r = \begin{bmatrix} -0.3 & -0.3 & -0.3 \end{bmatrix}^T \) m and is commanded to move to a position of \( r = \begin{bmatrix} 0.3 & 0.3 & 0.3 \end{bmatrix}^T \) m, with an initial and final attitude
of \( q = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}^T \). This a good representative maneuver because if any of the twelve thrusters fails, the spacecraft must perform maneuvering other than simply accelerating to start the translation and decelerating to stop the translation.

For nonlinear systems, however, a step response is less useful because the system response depends on many factors such as initial conditions. Therefore, there is no single representative maneuver that would provide all the information necessary to characterize how the system would respond for any arbitrary scenario. With this in mind, additional test maneuvers must be performed to ensure that all aspects of the system response have been observed.

Table 6.1 gives the values of the 6DOF SPHERES parameters. Note that information for only 6 thrusters are given because thrusters 7–12 can be paired with thrusters 1–6, as can be seen in Figure 6-1.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spacecraft mass</td>
<td>( m )</td>
<td>4.3 kg</td>
</tr>
<tr>
<td>Spacecraft inertia</td>
<td>( J )</td>
<td>( \text{diag} \left( \begin{bmatrix} 2.3 &amp; 2.4 &amp; 2.1 \end{bmatrix} \cdot 10^{-2} \right) \text{ kg}\cdot \text{m}^2 )</td>
</tr>
<tr>
<td>Thrust directions</td>
<td>( D )</td>
<td>( \begin{bmatrix} 1 &amp; 1 &amp; 0 &amp; 0 &amp; 0 &amp; 0 \ 0 &amp; 0 &amp; 1 &amp; 1 &amp; 0 &amp; 0 \ 0 &amp; 0 &amp; 0 &amp; 0 &amp; 1 &amp; 1 \ 0 &amp; 0 &amp; 0 &amp; 0 &amp; 1 &amp; -1 \ 0 &amp; 1 &amp; -1 &amp; 0 &amp; 0 &amp; 0 \ 0 &amp; 0 &amp; 1 &amp; -1 &amp; 0 &amp; 0 \end{bmatrix} )</td>
</tr>
<tr>
<td>Thruster lever arms</td>
<td>( L )</td>
<td>0.0965 ( \begin{bmatrix} 1 &amp; -1 &amp; 0 &amp; 0 &amp; 0 &amp; 0 \ 0 &amp; 0 &amp; 1 &amp; -1 &amp; 0 &amp; 0 \end{bmatrix} ) m</td>
</tr>
</tbody>
</table>

Figure 6-1: Thruster locations and directions for 6DOF SPHERES spacecraft.
6.1.1 Baseline Translation

To begin, a simulation of the representative translation with all thrusters operational was performed to provide a baseline for how the system performs under nominal conditions. The control system implemented in this simulation has been used successfully on the ISS for multiple tests [14]. The position controller is run at 0.5 Hz and has a damping of $\zeta = 0.7$ and bandwidth of $\omega_n = 0.2$ rad/s. The attitude controller is run at 1 Hz and has a damping of $\zeta = 0.7$ and bandwidth of $\omega_n = 0.4$ rad/s. Control allocation is performed by multiplying the commanded forces and torques with precomputed mixing matrix [13].

Figure 6-2 shows four plots of the position, velocity, attitude (converted to roll, pitch, and yaw Euler angles), and angular rates versus time. The position plot shows a typical underdamped step response with a slight overshoot, but settling to the commanded position with little velocity. The attitude is held relatively constant throughout the maneuver—residual attitude errors are due to effects such as thrust uncertainty and mismodeled inertias implemented in the simulation to match real effects.

The same simulation was run again, except thruster 9 was disabled, simulating a single thruster failure. Figure 6-3 shows the results of this simulation. It can be seen that the first 30 seconds are the same as the previous simulation. However, when the spacecraft approaches the desired position and attempts to stop, thruster 9, which would normally be used to stop the spacecraft fails to actuate. This causes the spacecraft to drift along a curved trajectory away from the desired position. Around 110 seconds, the spacecraft begins to rotate rapidly so that it can once again thrust towards the desired position. Around 140 seconds, it reaches the desired position, but once again cannot stop itself and drifts away along a similar curved trajectory. In addition, large oscillations in the velocity and angular rate can be observed. This is actually caused by the fact that the position and attitude controllers are operating at different rates and actually begin to fight against each other. The position controller generates forces to maneuver the spacecraft to the
correct position. This, however, creates residual torques due to the failed thruster. The attitude controller then produces commands to correct this residual torque, which cancels the force that the position controller generates. This fighting of the controllers produces unstable behavior (large oscillations in position and a tumbling attitude) and also wastes a lot of fuel in the process. This serves as a motivation for creating a control system that can properly handle thruster failures to avoid this situation.

6.1.2 Reconfigurable Control Allocation

The first technique to be tested to see if it can control a spacecraft with failed thrusters is a reconfigurable control allocator, described in Section 4.2. The reconfigurable control allocator will be solved with the redistributed pseudoinverse and active set methods.
Redistributed Pseudoinverse

The redistributed pseudoinverse method was tested using the same PD controller presented in the previous section. The control allocator can be reconfigured to be aware of the failure on thruster 9 by zeroing the corresponding column of the control effectiveness matrix, $B$. The algorithm described in Section 4.2.1 is run every control period to allocate the commanded forces and torques to the remaining operational thrusters.

Figure 6-4 shows the results of the simulation. The first half of the simulation is very similar to the baseline case. The spacecraft attempts to stop at the desired position, but cannot since thruster 9 is failed. It then follows a curved trajectory away from the desired position. It then is able to rotate and move towards the desired position again. This time, however, as it passes through the desired location at a high velocity, it attempts to stop, which creates a rapid spin in the z-axis. The spacecraft is unable to recover from this tumbling motion and drifts away from the
desired location. This result actually shows much worse performance than the baseline case, even though the control allocator is “aware” of the thruster failure.

Figure 6-4: 6DOF representative maneuver: PD control with redistributed weighted pseudoinverse, thruster 9 failed (simulation).

**Active Set Method**

The poor performance of the redistributed pseudoinverse can be attributed to the fact that it only approximately solves the control allocation problem and may not converge to the exact or optimal solution, if it exists. To test this, the control allocation problem can be solved with the active set method.

Figure 6-5 shows the result of the simulation with the active set control allocation and equal weights (i.e., $W = I$). Around 35 seconds, the spacecraft begins to thrust to decelerate. At this time, the control allocator cannot produce a force to stop while keeping the torque near zero, so the spacecraft begins to rotate. This rotation allows the spacecraft to stop and settle at the desired position. Somewhat luckily, the appropriate torque to rotate back to the original attitude is also feasible and the
spacecraft is able properly complete the maneuver. The next simulation will show why this is a “lucky” situation.

When the initial and desired positions are switched, the results, shown in Figure 6-6, reveal a weakness in the reconfigurable control allocation approach. At the start of the maneuvering, around 20 seconds, the spacecraft thrusts to accelerate in the direction of the desired position. Since thruster 9 has failed, producing this force creates a residual torque on the spacecraft, which causes the spacecraft to begin spinning in the positive z-direction. Despite this spinning, the spacecraft is able to reach its desired position and settle down to zero velocity. However, the spacecraft is left spinning in place because the torque required to stop the spacecraft would send the spacecraft out of position. In this case, the cost required to produce the correct torque outweighs the cost required to produce the correct force.

The same simulation was run with a tuned 0.95 weight on the torques and a 0.05

Figure 6-5: 6DOF representative maneuver: PD control with active set method, thruster 9 failed (simulation).
weight on the forces to see if this would allow the control system to settle at the correct position and attitude. Figure 6-7 shows the result. This clearly shows that position is not held as tightly since achieving the desired force has a much lower weight than before. Also, the control system attempts to achieve the correct attitude by actually slowing down the angular rate of the spacecraft. Even these tuned weights, the spacecraft is unable to settle down to the correct position and attitude. Further increasing the weight on the torques and lowering the weight on the forces, produces results similar to that shown in Figure 6-3.

The main reason why this control approach cannot handle these situations is because of the nonholonomicity of the spacecraft when a thruster fails. To achieve motions in a certain direction, combinations of control inputs, separated in time, must be executed. The attitude and position PD controllers, however, produce a desired force and torque only based off of the current state of the spacecraft, which may not necessarily be feasible commands. Because of this, the spacecraft cannot settle to the
exact desired position and attitude in all situations.

6.1.3 Piecewise Trajectory

The second technique to be tested is the piecewise trajectory following described in Section 4.3.1. This method involves piecing together motions that are known to be feasible to generate a feasible trajectory from the initial state to the final state. To complete the representative maneuver, the spacecraft rotates about the body x-axis, rotates about the body y-axis, translates to the final position, rotates about the y-axis, and rotates about the x-axis. The normal control allocator and PD controllers are used in these tests. This test was run on the ISS during SPHERES Test Session 18 on August 15, 2009. Results of this test compared against the simulation are shown in Figure 6-8. These results show that this technique was able to control the spacecraft from one position to another despite the failed thruster. To show that thruster 9 was not used, the thruster firings are shown in Figure 6-9. The red dotted line indicates
where the thruster was disabled. The thruster was allowed to be used during the initial part of the test just to get the correct initial positioning of the spacecraft.

![Graphs showing thruster performance](image)

Figure 6-8: 6DOF representative maneuver: PD control, piecewise trajectory (hardware).

The close matching of the simulation and the hardware results validates the simulation. The only noticeable difference between the simulation and hardware are offsets in time when the various maneuvers start. Since the maneuvers terminate based on precise state errors, these differences are expected with slight differences in initial conditions and other noise sources. These results also show that there are low enough disturbances that this technique is valid. If the disturbances were too high, the spacecraft would diverge from its nominal trajectory, which the technique would not be able to correct. Running other cases are not really necessary for this technique since it is a series of decoupled maneuvers. Scaling and rearranging these maneuvers would result in achieving any final position and attitude.

With a symmetry argument, it can be shown that this technique can handle any single thruster failure. If this technique was extended to multiple thruster failures,
however, it would only handle 43% of all double thruster failure cases and 4.8% of all triple thruster failure cases. This calculation is based on the SPHERES thruster geometry and the fact that this technique requires two body-fixed torques and one body-fixed force. This technique will serve as a baseline to compare the performance of the MPC algorithm against.

6.1.4 Model Predictive Control

The MPC technique described in Section 4.4 is the final technique to be tested. As mentioned in Section 4.4.1, there are a few different options to choose from in terms of using or not using terminal equality constraints to achieve stability. Each of these options underwent an iterative procedure of weight tuning to achieve the best performance. A brief summary of the findings are provided below:

- No terminal equality constraints: Stability without terminal equality constraints is not guaranteed and achieved only through iterative weight tuning (cut-and-try method). It turned out that the performance without terminal equality constraints was not acceptable. For relatively low position and attitude weights, the spacecraft would simply drift since the cost to use fuel outweighed the costs incurred by drifting. Adjusting the control weights did not change this
qualitative behavior. When the position and attitude weights were increased, the spacecraft would accelerate quickly in position or attitude to reduce the position and attitude errors and cause unstable behavior. The large velocities achieved by these accelerations dramatically reduces the accuracy of the discrete approximation of the continuous dynamics. Because of this, the MPC algorithm is unable to recover and properly regulate the spacecraft’s state.

- Terminal velocity-only equality constraints: As a compromise between none and all of the terminal equality constraints being used, using terminal equality constraints on only the velocity and angular velocity was chosen as an option. The reasoning behind this choice is that the spacecraft could have a large weight on the position and attitude states to steer the system towards the desired position and attitude while the terminal velocity equality constraints would ensure that the spacecraft could always return to a zero-velocity state at the end of the prediction horizon. The results were not encouraging, however. For low position and attitude weights, the spacecraft would mostly drift incurring large position and attitude errors. As these weights were increased, instability would result despite the addition of the velocity constraints.

- Terminal equality constraints: Finally, terminal equality constraints on all of the states were used. An iterative tuning process was still required to achieve stability. As mentioned in Section 5.4, stability is not guaranteed since feasibility of the optimization problem with premature termination is not guaranteed. However, with proper tuning, stability is achieved in simulation. Relatively large velocity weights are required to keep the system from accelerating too fast, which would result in instability. The terminal equality constraints on the position and attitude are mostly what drive the spacecraft towards the desired position and attitude—the weights on position and attitude do little to adjust the “settling time” of the spacecraft.

The final, tuned parameters that were selected for the MPC algorithm are shown in Table 6.2. The control loop runs at 1 Hz while the MPC algorithm updated the
scheduled controls every 4 control periods \((c = 4)\). With 5 predictions, the MPC algorithm was able to predict 20 seconds in advance (each time step is 4 seconds). These parameters were chosen to reflect realistic values that would be able to be performed online in real time. The tuned weights and control bounds are also shown in this table. Note that the control bounds are much smaller than the actual thrust capability of the SPHERES thrusters (approximately 0.1 N). This is because the thruster pulses interfere with the global metrology system of SPHERES. Therefore a portion of the control period is allocated to thrusting and the rest is allocated to global beacon updates. With 200 ms out of every 1 second control period allocated to thrusting, the average/effective maximum thrust is approximately 0.02 N.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control period</td>
<td>-</td>
<td>1 s</td>
</tr>
<tr>
<td>MPC update factor</td>
<td>(c)</td>
<td>4</td>
</tr>
<tr>
<td>Number of predictions</td>
<td>(p)</td>
<td>5</td>
</tr>
<tr>
<td>Position weight</td>
<td>(\text{diag}(Q(1:3,1:3)))</td>
<td>([10, 10, 10])</td>
</tr>
<tr>
<td>Velocity weight</td>
<td>(\text{diag}(Q(4:6,4:6)))</td>
<td>([1000, 1000, 1000])</td>
</tr>
<tr>
<td>Attitude weight</td>
<td>(\text{diag}(Q(7:9,7:9)))</td>
<td>([10, 10, 10])</td>
</tr>
<tr>
<td>Angular velocity weight</td>
<td>(\text{diag}(Q(10:12,10:12)))</td>
<td>([100, 100, 100])</td>
</tr>
<tr>
<td>Control weight</td>
<td>(\text{diag}(R))</td>
<td>([1, 1, 1, 1, 1])</td>
</tr>
<tr>
<td>Control bounds</td>
<td>(u_{\text{max/min}})</td>
<td>(\pm 0.02) N</td>
</tr>
</tbody>
</table>

Figure 6-10 shows the result of the simulation on the representative maneuver. The spacecraft, from the start of its maneuvering, is able to predict that it will not be able to stop properly without thruster 9. Because of this, it begins rotating immediately to point other thrusters in the direction of motion so that it can stop its motion towards the desired position. After it reaches the desired position, it is able to rotate back to the desired attitude. Additional simulations with different initial conditions were performed and it was observed that the spacecraft was still able to execute the commanded maneuver.

Table 6.3 shows a comparison of MPC versus the piecewise trajectory technique in terms of several metrics. The first two metrics are the time and fuel used to
execute the representative maneuver. When using MPC, the maneuver is completed in a fraction of the time it takes to complete the same maneuver with the piecewise trajectory technique. This increase in speed comes at the cost of a little more fuel usage.

Table 6.3: Comparison of the piecewise trajectory and MPC techniques.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Piecewise trajectory</th>
<th>MPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Representative maneuver completion time</td>
<td>90 s</td>
<td>35 s</td>
</tr>
<tr>
<td>Representative maneuver fuel used</td>
<td>2.5% of tank</td>
<td>2.6% of tank</td>
</tr>
<tr>
<td>Percent of single thruster failures handled</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Percent of double thruster failures handled</td>
<td>42.8%</td>
<td>85.7%</td>
</tr>
<tr>
<td>Percent of triple thruster failures handled</td>
<td>4.8%</td>
<td>71.4%</td>
</tr>
</tbody>
</table>

The next few metrics are the percentage of thruster failure cases the technique handles for different numbers of failures. As previously mentioned, for the piecewise trajectory approach these percentages were calculated by determining the thruster failures that could occur while still being able to produce the requisite body-fixed force and torques. For MPC, numerous simulations were run with various thruster
failure cases run on the representative maneuver with two different initial conditions. The number of simulations that needed to be run was greatly reduced by using the symmetries of the SPHERES thruster placement. One important note from these simulations is that the weights did not need to be retuned for the various thruster failure cases. The same weights shown in Table 6.2 were equally valid for the various thruster failure cases. From Table 6.3 it can be seen that MPC handles a much larger percentage of double and triple thruster failures over the piecewise trajectory technique.

All of the results presented on MPC up to this point have been simulations. The hardware results in Section 6.1.3 showed very good agreement with the simulation results, validating the simulation itself as representative of the real testing environment. It can be argued, therefore, that the simulation results for MPC in this section are representative of what would occur in the actual hardware testbed on the ISS. That is only partially accurate, however, since there are parts of the MPC algorithm that are not being tested in the simulation environment. Further tests with the 3DOF testbed have been performed to address this issue.

6.2 Three-Degree-of-Freedom Results

To test unexercised parts of the MPC algorithm, tests were also performed on the 3DOF SPHERES testbed at the SSL. The 6DOF simulation is not a real-time simulation so the effects of finite computation time is not present, disallowing the testing of the delay compensation method outlined in Section 5.3. Also, since there is no capability to run the MPC algorithm online on the 6DOF ISS testbed, as explained in Section 5.2.2 the MPC algorithm must be tested on the 3DOF MIT testbed.

The representative maneuver used for the 3DOF tests is similar to that used for the 6DOF tests. The spacecraft starts at a position of $\mathbf{r} = [0 \ 0]^T$ m and is commanded to move to a position of $\mathbf{r} = [0.3 \ 0.3]^T$ m, with an initial and final attitude of $q = 0^\circ$. Table 6.4 gives the values of the 3DOF SPHERES parameters.
Table 6.4: 3DOF SPHERES parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spacecraft mass</td>
<td>$m$</td>
<td>12.4 kg</td>
</tr>
<tr>
<td>Spacecraft inertia</td>
<td>$J$</td>
<td>0.0856 kg·m²</td>
</tr>
<tr>
<td>Thrust directions</td>
<td>$D$</td>
<td>$\begin{bmatrix} 1 &amp; 0 &amp; 0 \ 0 &amp; 1 &amp; 1 \end{bmatrix}$</td>
</tr>
<tr>
<td>Thruster lever arms</td>
<td>$L$</td>
<td>0.0965 $[0 \ 1 \ -1]$ m</td>
</tr>
</tbody>
</table>

6.2.1 Baseline Translation

The first test performed on the 3DOF SPHERES hardware was a baseline translation with all thrusters operational. The gains of the PD controller were adjusted to match the new mass and inertia of the spacecraft.

Figure 6-11 shows the resulting trajectory. The satellite clearly is able to complete the maneuver, but there is some steady state error in position. This is actually due to the fact that the table cannot be perfectly leveled. The table legs’ length can be adjusted to remove large errors, but small tip/tilt errors might still be present. In addition, the glass on the table is not perfectly flat. This non-zero gradient imparts a force due to gravity on the spacecraft, creating a steady state error in position since there is no integrator in the controller. It is important to recognize this source of error because it appears in all of the following results.

Figure 6-12 shows the resulting trajectory when thruster 9 is failed. The behavior is qualitatively very similar to the behavior seen in Section 6.1.1. The spacecraft is unable to stop itself when it reaches the desired position and cannot reach the desired position or attitude after this point. This shows very undesirable behavior that will be fixed by using MPC.

6.2.2 Model Predictive Control

The MPC algorithm was tested on the 3DOF SPHERES testbed with the parameters given in Table 6.5. The main control loop was set to run at 1 Hz and the MPC algorithm was initiated every other control period ($c = 2$), for an effective rate of 0.5
Hz. This algorithm planned 10 time steps in advance (equivalent to 20 seconds since each time step is 2 seconds) and was run online but off-board on a desktop computer. The MPC algorithm took approximately 1.5 seconds to run (per iteration) on a 2.83 GHz processor leaving approximately 0.5 seconds as a buffer for processing time variations and communication delays to and from the spacecraft. Therefore, there is an effective 2 second delay between when the updated controls are requested and when the updated controls are actuated. Table 6.5 also shows the tuned state and control weights that were used and the control bounds. The control bounds here are set to 0.04 instead of 0.02 to give the algorithm the ability to produce larger effective thrust to counteract the slant in the table. This is done by simply allocating 400 ms out of 1 second for thrusting instead of the typical 200 ms. This comes at the cost of less beacon updates for global metrology.

To test the effectiveness of the delay compensation method outlined in Section 5.3, the test was run with and without the prediction step. Note that this “prediction step” is different from the predictive nature of MPC, despite its similar terminology.
Without the prediction step, this two-second delay is too large for the controller to handle, causing the control system to not function properly. The resulting trajectory for this test was qualitatively very similar to that shown in Figure 6-12.

With the prediction step, the two-second delay can be mitigated. Figure 6-13 shows the resulting trajectory with the predicted states overlayed on the same plot. Ideally, if the model matched the actual dynamics perfectly, the predicted states

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control period</td>
<td></td>
<td>1 s</td>
</tr>
<tr>
<td>MPC update factor</td>
<td>$c$</td>
<td>2</td>
</tr>
<tr>
<td>Number of predictions</td>
<td>$p$</td>
<td>10</td>
</tr>
<tr>
<td>Position weight</td>
<td>diag($Q(1:2,1:2)$)</td>
<td>[100 100]</td>
</tr>
<tr>
<td>Velocity weight</td>
<td>diag($Q(3:4,3:4)$)</td>
<td>[100 100]</td>
</tr>
<tr>
<td>Attitude weight</td>
<td>diag($Q(5,5)$)</td>
<td>10</td>
</tr>
<tr>
<td>Angular velocity weight</td>
<td>diag($Q(6,6)$)</td>
<td>1000</td>
</tr>
<tr>
<td>Control weight</td>
<td>diag($R$)</td>
<td>[1000 1000 500]</td>
</tr>
<tr>
<td>Control bounds</td>
<td>$u_{max/min}$</td>
<td>±0.04 N</td>
</tr>
</tbody>
</table>
would match perfectly with the actual states. However, due to process noise, sensor noise and other unmodeled effects, the predicted and actual states do not match exactly. Both the position and angle predictions match well, however there is an obvious delay in the velocity and angular velocity predictions. Despite this error, the prediction allows the delay to be compensated enough to achieve stability in the system. The spacecraft is able to reach the desired position by executing an angular flip. The steady state error in position is due to local gradient of the glass table, as was also seen in Figure 6-11.

Figure 6-13: 3DOF representative maneuver: MPC, thruster 9 failed (hardware).
Chapter 7

Conclusion

7.1 Thesis Summary

The overall objective of this thesis was to design and analyze a reconfigurable control system that is able to recover from thruster failures that cause the spacecraft to become underactuated. The following provides a summary of the chapters in this thesis to achieve this objective.

Chapter 1 motivated the need for thruster FDIR. Thruster failures account for 24% of all attitude and orbit control system failures and can have a serious impact on the spacecraft’s ability to control itself. The move towards less expensive spacecraft through deployment systems such as ESPA and P-POD as well as the emerging idea of having clusters of spacecraft such as System F6 have motivated the need for performing thruster FDIR without the use of additional hardware. The SPHERES testbed was also introduced as a means to design and test the algorithms presented in this thesis.

Chapter 2 surveyed the literature to determine what techniques existed to solve the thruster FDIR problem. Many general techniques for actuator FDI were discussed, which mostly involve using banks of Kalman filters and using the innovations to detect and isolate the failure. Techniques for detecting specifically thruster failures were also presented. The first technique uses the thruster geometry and an estimate of the spacecraft’s acceleration and angular acceleration to determine the likelihood that
each thruster failed. The second technique extends the first since it is more likely to correctly isolate infrequently-commanded failed-off thrusters, can distinguish a failure on closely positioned thrusters, and can detect multiple failures. This last thruster FDI technique has already been implemented, tested, and shown to work on the SPHERES testbed. It was determined from this that more work on thruster FDI is not necessary. Thruster failure recovery techniques were also surveyed. General actuator failure recovery techniques for linear systems were discussed and deemed inappropriate for the problem. There also exists a large body of literature on failure recovery techniques using redundancy and reconfigurable control allocation. There is less literature on control techniques for underactuated spacecraft. Some have designed control systems for underactuated attitude control and others have analyzed safe trajectory design for translational control in the event of thruster failures. However, no one has tackled the problem of designing a reconfigurable control system for thruster failures that cause the spacecraft to become underactuated.

Chapter 3 provided a brief overview of rigid-body dynamics and used this to develop a 6DOF and 3DOF model of a thruster-controlled spacecraft. Controllability analyses were then performed on these models to gain further insight into the system. First, LTI controllability analyses were performed. This controllability analysis needed to be extended to handle the unilateral nature of the thruster control inputs. It was determined, however, that the Jacobian linearization limited the system dynamics in a way to render the system LTI uncontrollable in the event of any thruster failure. A more advanced nonlinear controllability test was performed on the system to see if it was small-time locally controllable. This analysis also showed that the system could lose the STLC property with a single thruster failure. These controllability analyses were based off of a specific thruster geometry, but the ideas can be extended to a more general system. These controllability results provide guidance for determining the types of control techniques that are applicable to the system model.

Chapter 4 began by outlining the challenges involved with designing a control system that could handle thruster failures and the underactuated spacecraft dynamics. These challenges were the coupling of the attitude and translational dynamics, the
multiplicative nonlinearities in the differential equations, the saturation of the control inputs, and the nonholonomic nature of the underactuated spacecraft dynamics. Three control techniques were chosen as candidates for a thruster failure recovery system. The first was using a reconfigurable control allocation. Even though this technique was designed for overactuated spacecraft, it was unclear whether it could work for an underactuated spacecraft. The second was by designing a feasible trajectory that could be tracked by a simple controller and control allocator. These two candidates served as a baseline for comparison with the third technique. The third technique uses MPC, which solves an optimization problem to determine the optimal open-loop control trajectory to minimize some cost function. This optimization problem is periodically resolved, based on the actual state of the spacecraft to “close the loop.”

Chapter 5 analyzed the theory of MPC as it is applied to the thruster-controlled spacecraft. First, a way to handle the attitude representations for the 6DOF and 3DOF models when performing the substitution of regulating the error to the origin instead of the state to the origin was given. Next, the process of selecting and implementing the nonlinear programming algorithm used to solve the MPC problem was given. The implemented SQP algorithm was benchmarked against the fmincon function in MATLAB®, showing a significant improvement in speed. The processing delay of actually performing this optimization problem is quite significant since it is a nonlinear programming problem. This delay can cause severe performance degradation as well as instability. A method to mitigate this delay was developed, which uses a prediction step that predicts the state that spacecraft will be in when the controls are actually applied. Finally, the feasibility and guaranteed stability through the use of terminal equality constraints is reanalyzed in the case where the optimization is terminated prematurely to meet the stringent timing requirements. It turns out that guaranteed stability is lost in this case.

Chapter 6 tests the three candidate thruster failure recovery techniques in simulation as well as in hardware tests on the ISS and at MIT. First, simulations of the spacecraft without and with a thruster failure provides a motivation for developing
this thruster failure recovery technique. The spacecraft is unable to properly control its position and attitude and wastes a lot of fuel in the process. Reconfigurable control allocation was tested with two implementations, the redistributed pseudoinverse and the active set method. The simulation of the redistributed pseudoinverse method actually performed worse than the baseline case and sent the spacecraft into a rapid tumble. The active set method performed much better than the redistributed pseudoinverse method and was able to achieve the desired position and attitude in some cases. Another case, however, showed that the system could not achieve both the position and attitude at the same time, highlighting the importance of handling the nonholonomic behavior of the system. The piecewise trajectory path planning technique was tested in simulation as well as on the ISS testbed. The spacecraft was able to follow the planned trajectory closely and achieved the desired position and attitude. Close agreement of the simulation and actual test results validated the simulation as well as this control technique. As this technique uses decoupled attitude and translation maneuvers, rearranging these maneuvers can easily result in achieving different attitudes and positions. The MPC algorithm was tested in simulation to see if it provides an improvement in performance over the piecewise trajectory technique. The simulation shows that it is able to complete the same maneuver in a fraction of the time with a small increase in fuel usage. Analysis of many thruster failure cases up to triple failures, shows that the MPC algorithm is able to handle a much higher percentage of possible thruster failure cases over the piecewise trajectory method. Additional 3DOF testing was performed in the SPHERES testbed at MIT to demonstrate the MPC algorithm operating in real time. This is actually a significant difference from simulation since the MPC algorithm has a large computational delay. Without delay compensation, the spacecraft cannot control its position and attitude with MPC. However, with the delay compensation implemented, the spacecraft was able to properly control itself.
7.2 Contributions

The following list provides a summary of the contributions made in this thesis:

- The LTI controllability analysis [37] was slightly extended to handle the unilateral nature of thrusters (a thruster can only produce positive force, not negative force). A simple rank check on the controllability matrix does not capture this control saturation and would classify uncontrollable systems as controllable. By using a positive span, where the coefficients of a linear combination of the columns of the controllability matrix can only be non-negative, controllability can be properly determined for any system with unilateral control inputs.

- Nonlinear controllability analyses were applied to a 6DOF spacecraft to determine if the SPHERES spacecraft was STLC in the event of any thruster failures, which, to the author’s best knowledge, has not been published before. Previous published results only analyzed the controllability of a 3DOF spacecraft model [43]. This controllability analysis showed that any single thruster failure can cause the spacecraft to lose the STLC property. While these results were derived using the specific thruster geometry of SPHERES, it is possible to extend these results to a more general spacecraft. The controllability analysis showed that a fully actuated spacecraft was STLC. The loss of any thruster on a fully actuated spacecraft means that the ability to generate a force and torque in a specific direction is lost, which is true independent of the thruster geometry. Therefore it may be argued that any fully actuated spacecraft with a thruster failure no longer passes the STLC test. Note that the possibility still exists that there is a transformation that will prove that the system is still STLC, since the test only provides sufficient and not necessary conditions for STLC.

- A survey and analysis of control techniques were performed, which clearly identified control techniques that would not work and techniques that could work. It was determined that linear state feedback would not work since the Jacobian-linearized system is LTI uncontrollable. Saturation control does not work as it
typically uses linear state feedback with the knowledge of the control saturations. Feedback linearization would not work due to the severe saturation of the control inputs. Dynamic programming was shown to be impractical given the memory requirements for implementation. A couple of different implementations of reconfigurable allocation was shown to not work through simulation. Nonholonomic control techniques that use the Philip Hall basis as additional controls does not work since the presented representation of the system does not pass the STLC test. Control techniques that involve planning, on the other hand, work well. The piecewise trajectory design was shown to work in simulation as well as in actual hardware. Other planning techniques such as RDTs would also likely work well. MPC, which basically plans a trajectory on a fixed time horizon, was shown to work well in simulation and, to a limited extent, hardware testing.

- An implementation of SQP in Embedded MATLAB® is provided in Appendix B. This provides other people with code that can be converted to C for ease of implementation in an embedded processor. This code has been proven to be able to be converted to C, compiled, and properly executed on the SPHERES testbed. A compiled version of this code has been shown to run much faster than the fmincon function in MATLAB®.

- A method to mitigate the effects of computational delay when performing the MPC calculations online has been developed and shown to work on an actual hardware system. This technique uses a prediction step to predict the approximate future state that the system will be in when the updated controls are actuated. This has been shown to work on the 3DOF SPHERES testbed with a very large delay of two seconds compared to the one second control period.
7.3 Recommendations & Future Work

The following list provides possible areas of extension to the work presented in this thesis or new areas of research to solve the thruster failure recovery problem:

- RDTs can be used to develop feasible trajectories that the spacecraft can follow, as an alternative method thruster failure recovery technique. This method was outlined in Section 4.3.2. It would be useful to see a comparison of this method in terms of computation speed and optimality versus the MPC algorithm. This technique will obviously provide a suboptimal solution compared to the MPC algorithm, but the degree of suboptimality is important. To mitigate this suboptimality, the trajectory output from the RDT algorithm can be further optimized through a second stage where the path is run through an optimization algorithm [71].

- Stability of the MPC algorithm implemented in this thesis is not guaranteed. This is because finding a feasible solution (not necessarily optimal) solution to the optimization with premature termination conditions is a very tough problem. Even proving that a feasible solution can be found (without actually finding it) is nontrivial as it is similar to proving the global controllability of a nonlinear system. One possible extension of this work is to find a way to guarantee stability of prematurely terminated MPC through a combination of convergence properties of SQP and the convergence of a Lyapunov function to the origin.

- An intermediate step to guaranteeing stability through an alternative approach, is to guarantee stability by finding an initial feasible solution quickly before the optimal solution is found. The RDT approach mentioned above can be used or a different optimization technique can be used. One example is presented in [72], which parameterizes the search space with Legendre polynomials and iterates to satisfy the constraints over the optimality.

- Stability of the MPC algorithm had to be shown through simulation. Since
it can be very time consuming to test every possible failure case for multiple initial conditions to determine which cases are stable, it would be ideal to have a method of analytically determining the robustness of the system or how close it is to instability. This would preclude the need for extensive simulations to be performed. An alternative to this would be to have an online health monitoring system to determine if the system has “gone unstable.” One potential candidate for a health monitoring variable is the value of the cost function. Many stability analyses use this cost function as a Lyapunov function candidate. Since this function must be monotonically decreasing, for the system to be guaranteed to be stable, it can be monitored for any large jumps in cost over time.

- The MPC algorithm presented uses a relatively large amount of fuel for regulation. Inspection of Figure 6-10 shows many changes in velocity indicating that the control system is spending a lot of fuel to keep the spacecraft still to counteract drift. Additions to the algorithm such as a “dead zone” can be implemented to see if the fuel usage for regulation can be decreased.

- The real-time iteration scheme using multiple shooting mentioned in Section 5.4 could be implemented as an alternative way to solve the optimal control problem. This method provides a fast solution to the optimization problem by precomputing many of the matrices and running a single iteration of a sparse SQP algorithm every control period. This reduces the delay due to computation and it would be interesting to see how this performs in comparison with the implementation presented in this thesis.
Appendix A

Small-Time Local Controllability of SPHERES

Knowing whether a dynamical system is small-time locally controllable (STLC) is useful to gain insight into the system. If it is STLC, it means that the system is able to follow any trajectory arbitrarily closely. Practically, this is very useful property for a system that needs to avoid obstacles. If a system is not STLC, the system may still be controllable, but it must go through global state changes to achieve certain states. For these systems, path planning to avoid obstacles becomes a necessity.

STLC can be analyzed for any system written in control-affine form,

\[ \dot{x} = f(x) + \sum_{i=1}^{m} g_i(x)u_i \quad (A.1) \]

where \( x \in \mathbb{R}^n \) is the state, \( f : \mathbb{R}^n \rightarrow \mathbb{R}^n \) is the drift vector field, \( u_i \) is the \( i^{th} \) control input, and \( g_i : \mathbb{R}^n \rightarrow \mathbb{R}^n \) is the \( i^{th} \) control vector field. The sufficient conditions for STLC \([42]\) are the following:

1. The drift velocity field is zero at \( x_0 \): \( f(x_0) = 0 \).

2. The Lie Algebra Rank Condition is satisfied by good Lie bracket terms up to degree \( i \):

\[ \text{For practical systems, this is limited by the update rate and bandwidth of the control system} \]
dim(\mathcal{L}(\{f, g_1, \ldots, g_m\}) = \text{span}(\{\phi_1, \ldots, \phi_n\} | \phi_1, \ldots, \phi_n \text{ are good}) = n.

3. All bad Lie brackets of degree \( j \leq i \) are neutralized.

4. Inputs are symmetric: \( U \in U_{\pm} \).

Refer to Section 3.3.2 for more details on the meaning of these conditions. Complete STLC results for the 3DOF and 6DOF models of SPHERES, presented in Sections 3.2 and 3.3.2 for various thruster failures will be derived in the following sections. For all STLC calculations, conditions 1 and 4 are met since STLC will only be analyzed for zero-velocity states and the inputs are symmetric since they are paired. Therefore only conditions 2 and 3 need to be analyzed.

### A.1 Three-Degree-of-Freedom SPHERES Model

![Thruster locations and directions for 3DOF SPHERES spacecraft.](image)

The 3DOF SPHERES spacecraft is shown in Figure A-1 with mass \( m \), inertia \( J \), and equal thruster lever arms \( l \). The dynamics in Section 3.2.2 can be rewritten in control-affine form as

\[
\begin{bmatrix}
\dot{r} \\
\dot{v} \\
\dot{\phi} \\
\dot{\omega}
\end{bmatrix} = \begin{bmatrix} v \\ 0 \\ \omega \\ 0 \end{bmatrix} + \sum_{i=1}^{m} \begin{bmatrix} 0 \\ \frac{1}{m} \Theta^T(q)d_i \\ 0 \\ \frac{1}{J} l_i \end{bmatrix} u_i.
\]  

(A.2)
STLC for the case of all thrusters operational is given in Section 3.3.2 and will not be repeated here.

For the case of thrusters 1 and/or 4 failed, the non-zero, good Lie brackets up to depth 3 are

\[
\{ \{g_2, g_3, [f, g_2], [f, g_3] \} \} = \begin{cases}
\begin{bmatrix}
0 & 0 & \frac{\sin(q)}{m} & \frac{-\cos(q)}{m} \\
0 & 0 & \frac{-\cos(q)}{m} & \frac{-\cos(q)}{m} \\
-\frac{-\sin(q)}{m} & -\frac{-\sin(q)}{m} & -\frac{\omega \cos(q)}{m} & -\frac{\omega \cos(q)}{m} \\
\frac{\cos(q)}{m} & \frac{\cos(q)}{m} & \frac{-\omega \sin(q)}{m} & \frac{-\omega \sin(q)}{m} \\
0 & 0 & \frac{l}{J} & \frac{l}{J} \\
\frac{l}{J} & \frac{l}{J} & 0 & 0
\end{bmatrix}
\end{cases}
\]  

(A.3)

Note that this set of vector fields is not full rank. Given that the good Lie brackets have reached a degree of 3, the bad Lie brackets that must be neutralized are

\[
\mathcal{B} = \{ [g_2, [f, g_2]], [g_3, [f, g_3]] \} = \begin{bmatrix}
0 & 0 \\
0 & 0 \\
\frac{-2l \cos(q)}{Jm} & \frac{2l \cos(q)}{Jm} \\
\frac{-2l \sin(q)}{Jm} & \frac{2l \sin(q)}{Jm} \\
0 & 0 \\
0 & 0
\end{bmatrix}
\]  

(A.4)

These, however, cannot be expressed as a linear combination of good Lie brackets of lower degree. Therefore the bad Lie brackets cannot be neutralized, condition 4 is not met and the system with thrusters 1 and/or 4 does not pass the test for STLC.

For the case of thrusters 2 and/or 5 failed, the Lie Algebra is spanned by the following set of good Lie brackets:

\[
\mathcal{L} = \text{span} \left( \{ g_1, g_3, [f, g_1], [f, g_3], [g_1, [f, g_3]], [[f, g_1], [f, g_3]] \} \right).
\]  

(A.5)

This can be shown to be full rank by taking the determinant of the matrix formed
by placing the vector fields into the columns of a matrix:

\[
\begin{vmatrix}
0 & 0 & -\frac{\cos(q)}{m} & \frac{\sin(q)}{m} & 0 & -\frac{l \sin(q)}{J_m} \\
0 & 0 & -\frac{\sin(q)}{m} & -\frac{\cos(q)}{m} & 0 & \frac{l \cos(q)}{J_m} \\
\frac{\cos(q)}{m} & -\frac{\sin(q)}{m} & -\frac{\omega \sin(q)}{m} & -\frac{\omega \cos(q)}{m} & \frac{l \sin(q)}{J_m} & \frac{l \omega \cos(q)}{J_m} \\
\frac{\sin(q)}{m} & \frac{\cos(q)}{m} & -\frac{\omega \cos(q)}{m} & -\frac{\omega \sin(q)}{m} & -\frac{l \cos(q)}{J_m} & \frac{l \omega \sin(q)}{J_m} \\
0 & 0 & 0 & \frac{l}{J} & 0 & 0 \\
0 & -\frac{l}{J} & 0 & 0 & 0 & 0
\end{vmatrix} = \frac{l^4}{J^4 m^4}. \quad (A.6)
\]

Since the determinant is non-zero for \( l \neq 0 \), condition 2 is met. The bad Lie brackets that must be neutralized are

\[
B = \{ [g_1, [f, g_1]], [g_3, [f, g_3]] \} = \begin{pmatrix}
0 & 0 \\
0 & 0 \\
0 & \frac{2l \cos(q)}{J_m} \\
0 & \frac{2l \sin(q)}{J_m} \\
0 & 0 \\
0 & 0
\end{pmatrix}.
\quad (A.7)
\]

The first bad Lie bracket is neutralized trivially while the second bad Lie bracket can be neutralized, by inspection, with the vector field \( g_1 \) of degree 1, which is less than the 3, the degree of the bad Lie bracket. Therefore condition 3 is met. Since all conditions are met, the system with thrusters 2 and/or 5 failed is STLC from zero-velocity states.

The case of thrusters 3 and/or 6 failed is STLC by a symmetry argument with thrusters 2 and/or 5. The case of thrusters (2 and/or 5) and (3 and/or 6) failed is not STLC and not controllable by inspection.
A.2 Six-Degree-of-Freedom SPHERES Model

The 6DOF SPHERES spacecraft is shown in Figure A-2 with mass \( m \), principal moment of inertias \( J_{xx}, J_{yy}, \) and \( J_{zz} \), and equal thruster lever arms \( l \). The dynamics in Section 3.2.1 can be rewritten in control-affine form as

\[
\begin{bmatrix}
\dot{r} \\
\dot{v} \\
\dot{\theta} \\
\dot{\omega}
\end{bmatrix} =
\begin{bmatrix}
v \\
0 \\
S^{-1}(\theta)\omega \\
-J^{-1}\omega \times J\omega
\end{bmatrix} + \sum_{i=1}^{m} \frac{1}{m} \Theta^T(\theta) d_i u_i
\]  

(A.8)

where the matrix

\[
S^{-1}(\theta_1, \theta_2) =
\begin{bmatrix}
1 & \sin(\theta_1) \tan(\theta_2) & \cos(\theta_1) \tan(\theta_2)
\end{bmatrix}
\]  

(A.9)

and \( \theta_1, \theta_2, \) and \( \theta_3 \) are the roll, pitch, and yaw Euler angles. This alternate attitude representation is used because quaternions introduce an extra degree of freedom that is constrained to satisfy the (integrable) unit length constraint. This redundant degree of freedom is problematic since the Lie Algebra Rank Condition would not be able to be met. The use of Euler angles eliminates this problem while only introducing the minor problem that the \( S^{-1} \) matrix is undefined at the singularity of \( \theta_2 = \pm \pi/2 \).

For the case of all thrusters operational, the Lie Algebra is spanned by the following set of good Lie brackets:

\[
\mathcal{L} = \text{span} \left( \{g_1, g_2, g_3, g_4, g_5, g_6, [f : g_1], [f : g_2], [f : g_3], [f : g_4], [f : g_5], [f : g_6]\} \right).
\]  

(A.10)

This can be shown to be full rank by taking the determinant of the matrix formed by placing the vector fields into the columns of the matrix, \( L \),

\[
\det(L) = \frac{-64l^6}{J_{xx}^2 J_{yy}^2 J_{zz}^2 m^6 \cos(\theta_2)}.
\]  

(A.11)
Here it becomes immediately obvious that at the singularity of $\theta_2 = \pm \pi/2$, the determinant is undefined, and therefore the analysis is invalid. To be completely rigorous, an alternate set of rotation angles could be used to analyze STLC at this singularity. This will not be done here since it can be assumed that if the spacecraft is STLC at all other attitudes, it will be STLC at this singularity. Since the determinant is non-zero for $l \neq 0$, condition 2 is met. The only bad Lie bracket that needs to be neutralized is the drift vector field since there are no other bad Lie brackets of degree $\leq 2$. Therefore condition 3 is met. Since all conditions are met, the system with all thrusters operational is STLC from zero-velocity states.

For the case of thrusters 1 and/or 7 failed, the non-zero, non-redundant, good Lie brackets up to depth 3 are

$$\{g_2, g_3, g_4, g_5, g_6, [f, g_2], [f, g_3], [f, g_4], [f, g_5], [f, g_6], [g_2, [f, g_3]]\}. \quad (A.12)$$

Note that this set of vector fields is not full rank. Given that the good Lie brackets have reached a degree of 3, the bad Lie brackets that must be neutralized are

$$\mathcal{B} = \{[g_2, [f, g_2]], [g_3, [f, g_3]], [g_4, [f, g_4]], [g_5, [f, g_5]], [g_6, [f, g_6]]\}. \quad (A.13)$$

While the first bad Lie bracket can be neutralized, the other 4 cannot. Therefore, condition 4 is not met and the system with thrusters 1 and/or 7 does not pass the test for STLC.
Figure A-2: Thruster locations and directions for 6DOF SPHERES spacecraft.
Appendix B

Optimization

Two algorithms for solving nonlinear and quadratic linear programming problems, respectively, are given in this section. In optimization, “programming” does not refer to computer programming, but rather the minimization of a given cost function with given constraints. Since any maximization problem can be converted into minimization by multiplying the cost function by -1, optimization can refer to minimization without loss of generality.

B.1 Nonlinear Programming: Sequential Quadratic Programming

Nonlinear programming refers to the problem of minimizing a nonlinear cost function \( f : \mathbb{R}^n \rightarrow \mathbb{R} \) by searching for an optimal \( x \in \mathbb{R}^n \) that satisfies the given nonlinear equality and inequality constraints. This can be expressed as

\[
\min_{x} f(x) \quad \text{(B.1a)}
\]

subject to \( a(x) = 0 \) \hspace{1cm} \text{(B.1b)}

\( b(x) \geq 0 \) \hspace{1cm} \text{(B.1c)}
where \( a : \mathbb{R}^n \rightarrow \mathbb{R}^{n_{eq}} \) defines \( n_{eq} \) equality constraints and \( b : \mathbb{R}^n \rightarrow \mathbb{R}^{n_{ineq}} \) defines \( n_{ineq} \) inequality constraints.

One algorithm to solve a nonlinear programming problem is called Sequential Quadratic Programming (SQP). As the name suggests, it solves quadratic programming problems to iteratively converge on a locally optimal solution. Starting at an initial guess, the nonlinear cost function is approximated as quadratic and the constraints are approximated as linear. This quadratic programming problem is solved to determine a step direction that is added to the current guess. This process then repeats by approximating the cost function as quadratic and the constraints as linear at the new guess.

A few modifications to this basic algorithm are commonly incorporated into SQP algorithms. The first is a method to estimate the Hessian of the Lagrangian or the cost function augmented with Lagrange multipliers and the constraints. Since it is often difficult or impractical to determine an analytical expression for this Hessian, it is useful to estimate it iteratively. Many references (e.g., [73]) are responsible for introducing the Broyden-Fletcher-Goldfarb-Shanno (BFGS) approximation to the Hessian. This method was later modified by [74] to maintain a positive definite Hessian in more cases. This BFGS estimate is initialized as the identity matrix, which causes the step direction to be equal to the direction of steepest descent. As the iterations progress, the estimate converges to the actual Hessian. The step direction when using the actual Hessian is called a Newton step, since it is a form of root finding using the Newton(-Raphson) method. Since the BFGS method approximates the actual Hessian, it is sometimes referred to as a quasi-Newton method.

It is also useful to determine derivatives/Jacobians numerically instead of evaluating an analytical expression for the derivatives. There are many options such as forward, backward and centered differencing. Centered differencing is the most accurate but requires the most computation. A forward differencing numerical derivative approximation was used for its lower computational burden. The step size used was a value suggested in [75], \( \sqrt{\epsilon} \), where \( \epsilon \) is the smallest increment representable by the machine’s precision.
Another problem with the original SQP algorithm is that it is only guaranteed to converge to an optimum when it is close to the optimum. To make the algorithm converge globally instead of locally, a modification was proposed \cite{76,74} to scale the step direction. This uses a merit function that weighs the reduction in the cost function, equality constraint satisfaction, and inequality constraint satisfaction. A line search can be used to find the optimum scaling factor that minimizes this merit function. The merit function actually used in the algorithm is given by \cite{64}.

This SQP algorithm, based off of \cite{64}, was implemented in Embedded MATLAB®. The code is provided below.

```matlab
function [x,iter,costFunc,eqFunc] = sqpAlgorithm(...
    sqpParams,mpcParams,mpcInputs,x)
%SQPALGORITHM calculates the local minimum of a nonlinear cost
%function with nonlinear equality and inequality constraints.
%
% minimize f(x)
% subject to g(x) = 0, h(x) \geq 0
%
% Note:
% - f(x) is defined in the function sqpCostFunc
% - g(x) is defined in the function sqpEqFunc
% - h(x) is defined in the function sqpIneqFunc
%
%#eml

%% Unpack parameters

% Variable and constraint dimensionality
numVars = sqpParams.numVars;
numEqCons = sqpParams.numEqCons;
umIneqCons = sqpParams.numIneqCons;

% Termination conditions
maxSQPIter = sqpParams.maxSQPIter;

%% Initialization of optimization problem

% Initialize the Lagrangian Hessian to the identity matrix
lagrHess = eye(numVars);

% Calculate the initial function values
costFunc = sqpCostFunc(mpcParams,mpcInputs,x);
eqFunc = sqpEqFunc(mpcParams,mpcInputs,x);
ineqFunc = sqpIneqFunc(mpcParams,mpcInputs,x);

% Calculate the gradients and Jacobians of the functions
```

133
[costGrad, eqJaco, ineqJaco] = sqpDerivatives(sqpParams, ...
    mpcParams, mpcInputs, x, costFunc, eqFunc, ineqFunc);

%% Main SQP loop
iter = uint8(1);
while iter ≤ maxSQPIter

    % Determine step vector
    [stepVec, QPIter] = sqpQuadProg(sqpParams, lagrHess, costGrad, ...
        [eqJaco; ineqJaco], [-eqFunc; -ineqFunc]);

    % Determine Lagrange multipliers for the next iteration
    [eqLagr ineqLagr numIneqActive boolIneqActive] = ...
    sqpCalcLagrMults(sqpParams, stepVec, costGrad, lagrHess, ...
        eqJaco, ineqJaco);  

    % Determine best scaling of the step direction
    stepScale = sqpCalcStepScale(sqpParams, mpcParams, mpcInputs, ...
        x, stepVec, ineqFunc, ineqLagr, numIneqActive, boolIneqActive);

    % Calculate the next iteration of x
    stepVec = stepScale*stepVec;
    x = x + stepVec;

    % Calculate gradients and Jacobians for the next iteration and
    % BFGS Hessian update
    costFunc = sqpCostFunc(mpcParams, mpcInputs, x);
    eqFunc = sqpEqFunc(mpcParams, mpcInputs, x);
    ineqFunc = sqpIneqFunc(mpcParams, mpcInputs, x);
    [costGradNext, eqJacoNext, ineqJacoNext] = sqpDerivatives(...,
        mpcParams, mpcInputs, x, costFunc, eqFunc, ineqFunc);

    % Calculate gamma, theta, and eta for BFGS update
    gamma = (costGradNext - costGrad);
    if numEqCons > 0
        gamma = gamma - (eqJacoNext - eqJaco).'*eqLagr;
    end
    if numIneqCons > 0
        gamma = gamma - (ineqJacoNext - ineqJaco).'*ineqLagr;
    end
    if stepVec.'*gamma ≥ 0.2*stepVec.'*lagrHess*stepVec
        theta = 1;
    else
        theta = (0.8*stepVec.'*lagrHess*stepVec)/...
            (stepVec.'*lagrHess*stepVec-stepVec.'*gamma);
    end
    eta = theta*gamma + (1-theta)*lagrHess*stepVec;

    % Update Lagrangian Hessian via the BFGS update
    lagrHess = lagrHess + (eta*eta.')/(eta.'*stepVec) - ...
        (lagrHess*(stepVec*stepVec.')*lagrHess)/...
        (stepVec.'*lagrHess*stepVec);
% Advance to next step
costGrad = costGradNext;
eqJaco = eqJacoNext;
ineqJaco = ineqJacoNext;
iter = iter+1;
end

function [eqLagrNext, ineqLagrNext, numIneqActive, boolIneqActive] = ...
    sqpCalcLagrMults(sqpParams, stepVec, costGrad, lagrHess,...
    eqJaco, ineqFunc, ineqJaco)
%SQPCALCLAGRMULTS calculates an estimate of the Lagrange multipliers
%for the next iteration.
%
%#eml

% Unpack constraint dimensionality
numVars = sqpParams.numVars;
numEqCons = sqpParams.numEqCons;
numIneqCons = sqpParams.numIneqCons;

% Initialize small-value tolerance for determining if a constraint is
% active or inactive
tol = 1e-6;

% Estimate the value of the next inequality constraint function
if numIneqCons == 0
    ineqFuncNext = [];
else
    ineqFuncNext = ineqJaco*stepVec + ineqFunc;
end

% Calculate the number and indices of the active inequality
% constraints as well as the active inequality constraint Jacobian
numIneqActive = uint8(0);
boolIneqActive = eml.nullcopy(zeros(1,numIneqCons));
ineqJacoActive_ = eml.nullcopy(zeros(numIneqCons,numVars));
for a = 1:numIneqCons
    if ineqFuncNext(a) < tol
        numIneqActive = numIneqActive+1;
        boolIneqActive(a) = true;
        ineqJacoActive_(numIneqActive,:) = ineqJaco(a,:);
    else
        boolIneqActive(a) = false;
    end
end

% Construct the matrix with the equality constraint Jacobian and
% active inequality constraint Jacobian
consJaco_ = [eqJaco; ineqJacoActive_];

% Calculate the vector of equality constraint Lagrange multipliers and
% active inequality constraint Lagrange multipliers
consLagrNext_ = pinv(consJaco*_consJaco_')*consJaco_*...
   (lagrHess*stepVec+costGrad);

% Separate out the equality constraint Lagrange multipliers
if numEqCons == 0
    eqLagrNext = [];
else
    eqLagrNext = consLagrNext_(1:numEqCons);
end

% Separate out the active inequality constraint Lagrange multipliers
ineqLagrActive = consLagrNext_(numEqCons+1:end);

% Construct the full inequality constraint Lagrange multipliers
ineqLagrNext = eml.nullcopy(zeros(numIneqCons,1));
idxIter = uint8(1);
for a = 1:numIneqCons
    if boolIneqActive(a)
        ineqLagrNext(a) = ineqLagrActive(idxIter);
        idxIter = idxIter+1;
    else
        ineqLagrNext(a) = 0;
    end
end

function stepScale = sqpCalcStepScale(sqpParams,mpcParams,...
   mpcInputs,x,stepVec,ineqFunc,ineqLagr,numIneqActive,...
   boolIneqActive)
%SQPCALCSTEPSCALE calculates the step scaling factor to balance
%reducing the cost function against constraint violations.
%
% Unpack constraint dimensionality
numIneqCons = sqpParams.numIneqCons;

% Declare the number of iterations for the two searches below
numMeritFuncIter = sqpParams.numMeritFuncIter;
numIneqConsIter = sqpParams.numIneqConsIter;

% Initialize small-value tolerance for determining if an inactive
% constraint is satisfied
tol = 1e-6;

% Perform line search for best decrease in merit function
% Initialize the lower and upper step scale
stepScaleLower = 0;
stepScaleUpper = 1;
for a = 1:numMeritFuncIter
% Calculate the middle point of the current interval
stepScaleMiddle = (stepScaleUpper-stepScaleLower)/2...   
   + stepScaleLower;

% Calculate a relatively small step size
epsilon = (stepScaleUpper-stepScaleLower)/10;

% Calculate the merit function just below and above the middle of
% the lower and upper bounds
meritA = sqpMeritFunc(sqpParams,mpcParams,mpcInputs,...
    x,stepVec,stepScaleMiddle-epsilon,ineqLagr);
meritB = sqpMeritFunc(sqpParams,mpcParams,mpcInputs,...
    x,stepVec,stepScaleMiddle+epsilon,ineqLagr);

% Update the lower/upper bounds depending on the relative values
% (assumes that there is only one minimum in the range)
if meritA < meritB
    stepScaleUpper = stepScaleMiddle+epsilon;
else
    stepScaleLower = stepScaleMiddle-epsilon;
end

% Calculate the converged step scale
stepScale1 = (stepScaleUpper-stepScaleLower)/2 + stepScaleLower;

% Calculate the step looking only at the inequality constraints
% If there are inactive constraints
if numIneqCons - numIneqActive > 0
    % Initialize the lower and upper step scale
    stepScaleLower = 0;
    stepScaleUpper = 1;

    % If the current iteration satisfies the inequality
    % constraints
    if min(ineqFunc)>-tol
        % Initialize the guessed step scale
        stepScale2 = 1;

        for a = 1:numIneqConsIter
            % Calculate the inequality constraint function
            ineqFunc = sqpIneqFunc(mpcParams,x + stepScale2*stepVec);

            % Calculate the minimum of the inactive inequality
            % constraints
            minIneqInactiveFunc = realmax('double');
            for b = 1:numIneqCons
                if ~boolIneqActive(b) && ...
ineqFunc(b) < minIneqInactiveFunc
    
minIneqInactiveFunc = ineqFunc(b);
end
end

% If the constraints are satisfied,
if minIneqInactiveFunc ≥ 0

  % If the current guess is the upper limit, return
  % that value
  if stepScale2 == stepScaleUpper
    break
  end

  % Redefine the lower bound
  stepScaleLower = stepScale2;

  % Calculate new guess between upper/lower bounds
  stepScale2 = stepScale2 + ...
              (stepScaleUpper-stepScale2)/2;

else % The constraints are not satisfied

  % Redefine the upper bound
  stepScaleUpper = stepScale2;

  % Calculate new guess between upper/lower bounds
  stepScale2 = stepScale2 - ...
              (stepScale2-stepScaleLower)/2;

end

else % Current iteration does not satisfy inequality constraints

  for a = 1:numIneqConsIter

    % Calculate the middle point of the current interval
    stepScaleMiddle = (stepScaleUpper-stepScaleLower)/2 ...
    + stepScaleLower;

    % Calculate a relatively small step size
    epsilon = (stepScaleUpper-stepScaleLower)/10;

    % Calculate the merit function just below/above the middle
    % of the lower and upper bounds
    ineqFuncA = min(sqpIneqFunc(mpcParams,...
                    x+(stepScaleMiddle-epsilon)*stepVec));
    ineqFuncB = min(sqpIneqFunc(mpcParams,...
                    x+(stepScaleMiddle+epsilon)*stepVec));

    % Update lower/upper bounds depending on relative values
    % (assumes that there is only one maximum in the range)
    if ineqFuncA < ineqFuncB
stepScaleLower = stepScaleMiddle - epsilon;
else
stepScaleUpper = stepScaleMiddle + epsilon;
end

% Calculate the converged step scale
stepScale2 = (stepScaleUpper - stepScaleLower) / 2 + ...
stepScaleLower;
end

else % There are no inactive inequality constraints
stepScale2 = 1;
end

%% Output the minimum step scale from both searches
if stepScale1 < stepScale2
stepScale = stepScale1;
else
stepScale = stepScale2;
end

function [costGrad, eqJaco, ineqJaco] = sqpDerivatives(sqpParams, ...
mpcParams, mpcInputs, x, costFunc, eqFunc, ineqFunc)
% SQPDERIVATIVES calculates the cost function gradient and constraint
% function Jacobians with user-supplied functions or via finite
differencing.
%#eml
% Unpack parameters
% Variable and constraint dimensionality
numVars = sqpParams.numVars;
umEqCons = sqpParams.numEqCons;
umIneqCons = sqpParams.numIneqCons;

% Derivative availability flags
costGradAvail = sqpParams.costGradAvail;
eqJacoAvail = sqpParams.eqJacoAvail;
ineqJacoAvail = sqpParams.ineqJacoAvail;

% Small step size suggested in Numerical Optimization by Jorge Nocedal
% and Stephen J. Wright, equation 7.6 p. 168
epsilon = sqrt(eps);

% Calculate cost function gradient
if costGradAvail
    costGrad = sqpCostGrad(mpcParams, mpcInputs, x);
else % Calculate the gradient via forward differencing

% Initialize the cost gradient vector
costGrad = zeros(numVars,1);

% Calculate the cost gradient by calculating the cost function at
% small steps away from the current location in each of the
% directions of the variable space
for a = 1:numVars

    % Create a new point slightly perturbed along one of the axes
    % in the variable space
    xplusdx = x;
    xplusdx(a) = xplusdx(a)+epsilon;

    % Calculate an element in the cost gradient
    costGrad(a) = ...
      (sqpCostFunc(mpcParams,mpcInputs,xplusdx)-costFunc)...
       /epsilon;

end

end

%% Calculate the equality constraint Jacobian

% If there are equality constraints, calculate the equality constraint
% Jacobian
if numEqCons > 0

    % If the equality Jacobian is available, use the user-supplied
    % function in sqpEqJaco.m
    if eqJacoAvail
      eqJaco = sqpEqJaco(mpcParams,mpcInputs,x);
    else % Calculate the Jacobian via forward differencing

        % Initialize the equality constraint Jacobian
        eqJaco = zeros(numEqCons,numVars);

        for a = 1:numVars

            % Create new point slightly perturbed along one of the
            % axes in the variable space
            xplusdx = x;
            xplusdx(a) = xplusdx(a)+epsilon;

            % Calculate a column in the equality constraint Jacobian
            eqJaco(:,a) = ...
              (sqpEqFunc(mpcParams,mpcInputs,xplusdx)-eqFunc)...
               /epsilon;

        end

    end

end
else % No equality constraints, no need to calculate Jacobian
  eqJaco = [];
end

% Calculate the inequality constraint Jacobian
% If there are inequality constraints, calculate the inequality
% constraint Jacobian
if numIneqCons > 0
  % If the inequality Jacobian is available, use the user-supplied
  % function in sqpIneqJaco.m
  if ineqJacoAvail
    ineqJaco = sqpIneqJaco(mpcParams, mpcInputs, x);
  else % Calculate the Jacobian via forward differencing
    % Initialize the inequality constraint Jacobian
    ineqJaco = zeros(numIneqCons, numVars);
    for a = 1:numVars
      % Create new point slightly perturbed along one of the
      % axes in the variable space
      xplusdx = x;
      xplusdx(a) = xplusdx(a) + epsilon;

      % Calculate a column in the inequality constraint Jacobian
      ineqJaco(:, a) = ...
        (sqpIneqFunc(mpcParams, mpcInputs, xplusdx) - ineqFunc) / epsilon;
    end
  end
else % No inequality constraints, no need to calculate Jacobian
  ineqJaco = [];
end

function output = sqpMeritFunc(sqpParams, mpcParams, mpcInputs, x,...
  stepVec, stepScale, ineqLagr)
%SQP MERIT FUNCTION calculates the value of the merit function, which is
% a combination of the cost function and constraint functions.
%
% Unpack parameters
numEqCons = sqpParams.numEqCons;
numIneqCons = sqpParams.numIneqCons;
% Set relative weighting on equality constraint satisfaction
beta = 0.5;
% Calculate the value of the merit function
output = sqpCostFunc(mpcParams,mpcInputs,x + stepScale*stepVec);
if numEqCons > 0
    output = output+beta*...
    sum(sqpEqFunc(mpcParams,mpcInputs,x+stepScale*stepVec).^2);
end
if numIneqCons > 0
    output = output-ineqLagr.'*...
    sqpIneqFunc(mpcParams,mpcInputs,x+stepScale*stepVec);
end

B.2 Quadratic Programming: Active Set Method

The general form for a quadratic programming problem is

$$\min_x \frac{1}{2} x^T H x + g^T x$$ \hspace{1cm} (B.2a)

subject to \hspace{1cm} Ax \geq b \hspace{1cm} (B.2b)$$

where $H \in \mathbb{R}^{n \times n}$ is a positive definite weighting matrix of the quadratic cost, $g \in \mathbb{R}^n$ is a weighting of the linear cost, and $A \in \mathbb{R}^{(n_{eq}+n_{ineq}) \times n}$ and $b \in \mathbb{R}^{n_{eq}+n_{ineq}}$ form the linear inequality constraints.

Note that the equality constraints are expressed as inequality constraints in this problem statement. This is done because the quadratic programming algorithm is an active set method. This means that the algorithm keeps track of the inequality constraints that are “active” or exactly met. Since equality constraints are the same as inequality constraints that are always active, they can be treated as inequality constraints that are always kept in the active set. This active set determines a feasible subspace to search for the optimum. A step direction in this subspace is determined. This step can be “blocked” by inactive inequality constraints. If this is the case, the step direction is scaled down and the blocking constraint is added to the active set. If there are no blocking constraints inequality constraints can be removed from the active set. A similar process continues until adding or removing constraints from the active set will no longer help to minimize the problem.
Since this algorithm needs to start with an initial feasible solution, a linear programming algorithm is used to find this initial feasible solution. This algorithm uses a very similar active set method to determine the initial feasible solution, with the introduction of a slack variable.

This algorithm was implemented in Embedded MATLAB®, based off of the active set method in [77] and the quadratic programming algorithm in [78].

```matlab
function [x,iter] = sqpQuadProg(sqpParams,H,g,A,b)
%SQPQUADPROG calculates the global minimum of a quadratic cost function with linear equality and inequality constraints.
%
%#eml

%% Unpack optimization parameters

% Variable and constraint dimensionality
numVars = sqpParams.numVars;
numEqCons = sqpParams.numEqCons;
umIneqCons = sqpParams.numIneqCons;
numCons = numEqCons + numIneqCons;

% Termination conditions
maxQPIter = sqpParams.maxQPIter;

% Added to make Embedded MATLAB happy
idxActivate = uint8(0);

%% Find an initial feasible solution

% Calculate initial x. If there are equality constraints, find a solution to those constraints using the backslash operator. If there are no equality constraints, set it to a vector of zeros.
if numEqCons > 0
    x = A(1:numEqCons,:)
end
else
    x = zeros(numVars,1);
end

% Calculate the value of the constraints
funcCons = A*x - b;

% Calculate the largest constraint violation
if numIneqCons > 0
    conViol = min(funcCons(numEqCons+1:numCons));
else
    conViol = 0;
end
```

143
% If initial x is infeasible, calculate feasible x
if conViol < 0

% Calculate new feasible x through a LP with a slack variable
xs = sqpLinProg(sqpParams,[[A;zeros(1,numVars)],[zeros(numEqCons,1);ones(numIneqCons,1);1]],[b;-1e-5],[x;-conViol+1]);

% Extract feasible x from returned solution
x = xs(1:numVars);
end

%% Form initial active set and projection subspace basis

% All equality constraints are in the active set
numActive = numEqCons;

% Form vector that indicates which constraints are active
boolActive = [true(numEqCons,1); false(numIneqCons,1)];

% Initialize the active set and calculate Q and R to determine the
% projection subspace basis
activeSet_ = zeros(size(A));
idxActive_ = uint8([1:numEqCons zeros(1,numIneqCons)]);
if numEqCons > 0
    for a = uint8(1:numEqCons)
        activeSet_(a,:) = A(a,:);
    end
    [Q_,R_] = qr(activeSet_.');
else
    Q_ = eye(numVars);
    R_ = zeros(numVars,numCons);
end
projBasis_ = zeros(numVars);
projBasis_(:,1:(numVars-numActive)) = Q_(:,numActive+1:numVars);

%% Main QP loop

iter = uint8(1);
while iter < maxQPIter

% Reset flag that indicates that a constraint has been added. This
% is useful since we do not want to remove a constraint in the
% same iteration where a constraint has been added.
addCon = false;

% Calculate the unscaled step vector
gradCost=H*x+g;

% Calculate the cost function gradient
gradCost=H*x+g;

% Calculate the projected Hessian and step vector
projHess_ = projBasis_.'*H*projBasis_;
projStepVec_ = pinv(projHess_)*-projBasis_.'*gradCost;

% Calculate the step vector
stepVec = projBasis_*projStepVec_;

%%% Calculate the next iterate

% Update constraint values (should be >= 0 when satisfied)
funcCons = A*x - b;

% Calculate the constraint gradient in the direction of the step
gradCons = A*stepVec;

% Determine which inactive inequality constraints block the % direction of the step
minGradCons = min(gradCons.*¬boolActive);

% If there are no blocking constraints
if minGradCons > -1e-10*norm(stepVec)
    % Calculate the value of next iterate with an unscaled step
    x = x + stepVec;
    % If the only active constraints are equality constraints, the % minimum has been reached and the QP is solved.
    if numActive == numEqCons
        break
    end

else % There are blocking constraints

    % Find distance to the blocking constraints
distance = abs(funcCons./gradCons);

    % Calculate the step scale as the distance to the closest % blocking constraint
    stepScale = realmax('double');
    for a = uint8(1:numCons)
        if ((gradCons(a) < -1e-10*norm(stepVec)) &&...
            ¬boolActive(a) &&...
            distance(a) < stepScale)
            stepScale = distance(a);
            idxActivate = a;
        end
    end

    % If the distance to the nearest constraint is greater than % one, an unscaled step to the subspace minimum can be made % without violating any constraints
    if stepScale > 1
        % Calculate the value of the next iterate with an % unscaled step
        x = x + stepVec;
    end

end
% If the only active constraints are equality constraints, % the minimum has been reached and the QP is solved. If % there are active inequality constraints, it is possible % that a constraint can be removed, yielding a better % solution.
if numActive == numEqCons
    break
end

else % Scaled step

    % Calculate the value of next iterate with a scaled step
    x = x + stepScale*stepVec;

    % Set flag indicating that a constraint has been added to % the active set
    addCon = true;

    % Add the activated constraint to the active set
    numActive = numActive+1;
    idxActive_(numActive)=idxActivate;
    boolActive(idxActivate)=true;
    activeSet_(numActive,:) = A(idxActivate,:);
    [Q_,R_] = qr(activeSet_.');

    % Update projection subspace
    for a = uint8(1:numVars)
        if a > numVars - numActive
            projBasis_(:,a) = zeros(numVars,1);
        else
            projBasis_(:,a) = Q_(:,a+numActive);
        end
    end

    % Remove constraints from the active set
    if ~addCon && numActive > numEqCons

        % Update the cost function gradient
        gradCost = H*x+g;

        % Estimate the current Lagrange multipliers
        lagrMult = activeSet_.'\gradCost; % p.165 gill

        % Ensure positive Lagrange mults for equality constraints
        if numEqCons > 0
            lagrMult(1:numEqCons) = abs(lagrMult(1:numEqCons));
        end
    end
end
% Determine the minimum Lagrange multiplier
[minLagrMult, idxDeactivate] = min(lagrMult);

% If there are no negative inequality Lagrange multipliers,
% the optimal solution has been found, exit the QP
if minLagrMult >= 0
    break
end

% Remove the deactivated constraint from the active set
numActive = numActive - 1;
boolActive(idxActive(idxDeactivate)) = false;
idxIter = idxDeactivate;
while idxIter <= numCons
    if idxIter >= numCons - numActive
        idxActive(idxIter) = 0;
    else
        idxActive(idxIter) = idxActive(idxIter + 1);
    end
    idxIter = idxIter + 1;
end

idxIter = idxDeactivate;
while idxIter <= numActive
    activeSet(idxIter,:) = activeSet(idxIter + 1,:);
    idxIter = idxIter + 1;
end
activeSet(numActive + 1,:) = zeros(1, numVars);

[Q, R] = qr(activeSet.');

% Update projection subspace
idxIter1 = numActive + 1;
idxIter2 = 1;
while idxIter1 <= numVars
    projBasis(:, idxIter2) = Q(:, idxIter1);
    idxIter1 = idxIter1 + 1;
    idxIter2 = idxIter2 + 1;
end

end

% Prepare for the next iteration
% Increment the iteration variable
iter = iter + 1;

function x = sqpLinProg(sqpParams, A, b, x)
% SQPLINPROG calculates an initial feasible solution to a quadratic
% programming problem.
了

% Unpack optimization parameters

% Variable and constraint dimensionality
numVars = sqpParams.numVars + 1;
numEqCons = sqpParams.numEqCons;
numIneqCons = sqpParams.numIneqCons + 1;
numCons = numEqCons + numIneqCons;

% Termination conditions
maxQPIter = sqpParams.maxQPIter;

% General initialization

% Initialize the simplex iteration flag
simplexIter = false;

% Calculate cost gradient for all iterations since it is constant
gradCost = [zeros(numVars-1,1);1];

% Added to make Embedded MATLAB happy
idxActivate = uint8(0);

% Form initial active set and projection subspace basis

% All equality constraints are in the active set
numActive = numEqCons;

% Form vector that indicates which constraints are active
boolActive = [true(numEqCons,1); false(numIneqCons,1)];

% Initialize the active set and calculate Q and R to determine the
% projection subspace basis
activeSet_ = zeros(size(A));
idxActive_ = uint8([1:numEqCons zeros(1,numIneqCons)]);
if numEqCons > 0
    for a = uint8(1:numEqCons)
        activeSet_(a,:) = A(a,:);
    end
    [Q_,R_] = qr(activeSet_);
else
    Q_ = eye(numVars);
    R_ = zeros(numVars,numCons);
end
projBasis_ = zeros(numVars);
projBasis_(:,1:(numVars-numActive)) = Q_(:,numActive+1:numVars);

% Main LP loop
iter = uint8(1);
while iter < maxQPIter
%%% Calculate the unscaled step vector

%% If performing a simplex iteration (there is only one degree of
%% freedom when performing a simplex iteration since the number of
%% active constraints is equal to the number of variables minus 1)
if simplexIter

%% Calculate the projected gradient
projGradCost = projBasis.*gradCost;

%% Flip step vector if gradient is increasing
if projGradCost(1) > 0
    stepVec = -projBasis(:,1);
else
    stepVec = projBasis(:,1);
end

else % Not a simplex iteration

%% Calculate the step vector
stepVec = -projBasis*projBasis.*gradCost;
end

%%% Calculate the next iterate

%% Update constraint values (should be \textgreater= 0 when satisfied)
funcCons = A*x - b;

%% Calculate the constraint gradient in the direction of the step
gradCons = A*stepVec;

%% Determine which inactive inequality constraints block the
%% direction of the step
minGradCons = min(gradCons.*~boolActive);

%% If there are no blocking constraints, there is no bounded
%% solution to the LP problem
if minGradCons > -1e-10*norm(stepVec)

%% Make a large step
if norm(stepVec) > 1e-10
    stepScale = 1e10;
    x = x + stepScale*stepVec;
end

break
else % There are blocking constraints

%% Find distance to the blocking constraints
distance = abs(funcCons./gradCons);
% Calculate the step scale as the distance to the closest
% blocking constraint
stepScale = realmax('double');
for a = uint8(1:numCons)
    if ((gradCons(a) < -1e-10*norm(stepVec)) &&
        ~boolActive(a) &&
        distance(a) < stepScale)
        stepScale = distance(a);
        idxActivate = a;
    end
end

% Calculate the value of the next iterate with a scaled step
x = x + stepScale*stepVec;

% For a LP, if the slack variable is less than or equal to
% zero, the initial feasible point has been found!
if x(numVars) <= 0
    break
end

% Add the activated constraint to the active set
numActive = numActive+1;
idxActive_(numActive)=idxActivate;
boolActive(idxActivate)=true;
activeSet_(numActive,:) = A(idx Activate,:);
[Q_,R_] = qr(activeSet_.');

% Update projection subspace
for a = uint8(1:numVars)
    if a > numVars - numActive
        projBasis_(::,a) = zeros(numVars,1);
    else
        projBasis_(::,a) = Q_(::,a+numActive);
    end
end

% Remove constraints from the active set
% A constraint must be removed if the current iteration is a
% simplex iteration.
if simplexIter
    % The next iteration is not a simplex iteration
    simplexIter = false;
    % Estimate the current Lagrange multipliers
    lagrMult = activeSet_.'\gradCost;
    % Ensure positive Lagrange mults for equality constraints
    if numEqCons > 0
        ...
lagrMult(1:numEqCons) = abs(lagrMult(1:numEqCons));
end

% Determine the minimum Lagrange multiplier
[minLagrMult,idxDeactivate] = min(lagrMult);

% If there are no negative inequality Lagrange multipliers, 
% the optimal solution has been found, exit the LP
if minLagrMult > 0
    break
end

% Remove the deactivated constraint from the active set
numActive = numActive-1;
boolActive(idxActive_(idxDeactivate)) = false;
idxIter = idxDeactivate;
while idxIter <= numCons
    if idxIter >= numCons - numActive
        idxActive_(idxIter) = 0;
    else
        idxActive_(idxIter) = idxActive_(idxIter+1);
    end
    idxIter = idxIter+1;
end
idxIter = idxDeactivate;
while idxIter <= numActive
    activeSet_(idxIter,:) = activeSet_(idxIter+1,:);
    idxIter = idxIter+1;
end
activeSet_(numActive+1,:) = zeros(1,numVars);

[Q_,R_] = qr(activeSet_.');

% Update projection subspace
idxIter1 = numActive+1;
idxIter2 = 1;
while idxIter1 <= numVars
    projBasis_(:,idxIter2) = Q_(:,idxIter1);
    idxIter1 = idxIter1+1;
    idxIter2 = idxIter2+1;
end

% Prepare for the next iteration

% Increment the iteration variable
iter = iter+1;
% If the number of active constraints is equal to the number of 
% variables minus one, a simplex iteration must be performed
if numActive == numVars-1
    simplexIter = true;
end
Bibliography


154


