UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

TOWARD TACTICAL AUTONOMY IN AERIAL ROBOTICS

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By

Joshua Karinshak

Norman, Oklahoma

2019

TOWARD TACTICAL AUTONOMY IN AERIAL ROBOTICS

A THESIS APPROVED FOR THE

SCHOOL OF AEROSPACE AND MECHANICAL ENGINEERING

BY

Dr. Andrea L'Afflitto, Chair

Dr. Wei Sun

Dr. Choon Yik Tang

Acknowledgments

To begin, I would like to express my sincerest gratitude to my graduate advisor, Dr. Andrea L'Afflitto. Shortly after I joined his Advanced Control Systems Lab, Dr. L'Afflitto invited me to perform graduate research under his direction. Since then, the knowledge and experience gained through his passion for research and investment in his students has opened incredible and unexpected doors for me. Some students remark they wouldn't be where they are without their advisor. I hadn't have even realized I could make it this far.

Next, I would like to thank the other two members of my thesis committee, Dr. Choon Yik Tang and Dr. Wei Sun. Dr. Tang. taught my very first course on control theory, and Dr. Sun introduced me to machine learning. The foundations they built are invaluable to me. Their service on my committee is greatly appreciated.

Thank you to my fellow researchers at the ACSL: Blake, Cole, John-Paul, Julius, Karen, and Tim. Your support and assistance in the lab and in the classroom was by every count needed for me to complete this degree. Through your friendship, I emerged sane.

Finally, thank you to my family for providing me with love and support for $t \in [t_0, \infty)$.

This thesis was partly supported by DARPA under grant no. D18AP00069. I wish to express my gratitude to DARPA for its continuous support

Contents

List of Figures

Abstract

This thesis addresses the issue of the generation of tactical trajectories, as is desirable in the case of robot autonomy in hostile environments. The generation of such trajectory currently lies beyond the purview of contemporary path planning research, which focuses on solutions to the shortest path problem and variations thereof. This novel guidance system is developed as a two-stage process. In the first stage, a graph-based search algorithm is used to generate a global trajectory, as is the norm for state-of-the-art path planners. In the second stage, optimal control techniques are utilized to develop local trajectories that are optimal with respect to user-defined cost matrices over a finite time horizon. The guidance system is implemented for use with quadrotor vehicles. It is tested in simulation and then in flight. In both cases, the trajectory generated by the system makes use of cover present in the environment, rather than fly directly to the goal point. Additionally included is a discussion on modern visual navigation techniques, which, if implemented, would enable autonomous operation in initially unknown environments. In combination with the guidance algorithm developed in this thesis, the resulting system would be fully capable of conducting missions in hostile environments.

Key Words: Model predictive control, path planning, autonomy, quadrotor.

Chapter 1: Introduction

In many cases, the problem of pathfinding is reduced to the shortest path problem. In this regard, pathfinding is a well-studied problem with numerous graph-based solutions descending from the famous A* algorithm [1]. Modern solutions, such as those descending from the D* variant of A*, offer fast solutions to the shortest path problem in dynamic environments, including the case wherein an initially unknown map is procedurally discovered [2]. However, graph-based pathfinding algorithms scale poorly to problems of higher dimension. As such, these algorithms are limited in their ability to optimize beyond spatial optimality. A solution may be found in model predictive control, which can generate a state trajectory optimized with regard to every state and control element, in accordance with user-defined cost matrices [3]. Historically, model predictive control has been limited in application on account of its high computational cost. The fast model predictive control technique proposed by Boyd offers accelerated performance but is limited to convex problems [4]. In this thesis, we propose, develop, and test a two-stage guidance system that implements a graph-search approach at the first stage and refines the proposed trajectory to local optimality via model predictive control at the second stage.

Such a guidance system would be of particular use in situations wherein the shortest path is not necessarily the most desirable—or in some circumstances, even undesirable. The salient example guiding the development of this project is the case of autonomous operation of aerial robots in hostile environments. In such a case, a direct path through an open area would render the system exposed and thus vulnerable to adversarial actions. Further, such a trajectory could be predicted and intercepted by intelligent agents. Under such circumstances, a tactical trajectory that makes use of walls and miscellaneous cover in the

environment would be preferred.

We begin this thesis with a discussion in Chapter 2 on visual navigation techniques, with a focus on simultaneous localization and mapping. The visual odometry and mapping capabilities provided allows a robot to operate reliably in GPS-denied environments without prior map knowledge. The guidance system is developed to be compatible with such techniques, with the intention of adding simultaneous localization and mapping as the project evolves in the future. In Chapter 3, we explore graph-based pathfinding algorithms to develop the first stage of the guidance system. Specifically, we implement a variant of the Lifelong Planning A* search algorithm. In Chapter 4, we discuss and implement fast model predictive control for the second stage of the guidance system, using the optimization techniques described in [4]. In Chapter 5, derive a linear discrete-time model of our target platform. Following that, we discuss the means by which the two stages are integrated, as well as how they interact with the general flight stack of the platform. We conclude Chapter 5 with discussion of experimentation, both in simulation and in real flight tests. In Chapter 6, we recount the entire project and discuss future work.

Chapter 2: Simultaneous Localization and Mapping

2.1. Framing SLAM

The problem of Simultaneous Localization and Mapping (SLAM) is, in essence, a problem of estimates and uncertainties across a sequence of actions and reactions. Per [5], we may consider the problem data of SLAM to be captured by

$$X_{0:k} = \{x_0, \ldots, x_k\}, \tag{2.1}$$

$$U_{0:k} = \{u_0, \ldots, u_k\}, \tag{2.2}$$

$$m_{0:n} = \{m_0, \ldots, m_n\}, \tag{2.3}$$

$$Z_{0:k} = \{z_0, \ldots, z_k\}, \tag{2.4}$$

where m_i denotes the location of the ith static landmark in the environment with $i \in \{0, \ldots, n\}$, and x_j, u_j, and z_j denote the state of the system, the control input applied to the system, and the estimated positions of observable landmarks at the jth timestep respectively, with $j \in \{0, \ldots, k\}$. The goal of SLAM is to estimate both $X_{0:k}$ and $m_{0:n}$ given $X_{0:k-1}$, $U_{0:k}$, and $Z_{0:k}$.

SLAM uses a *motion model* and an *observation model* to form an estimate of the subsequent state. The motion model is captured by $P(x_k|x_{k-1}, u_k)$, i.e. the probability distribution of the current state given the previous state and control input. The observation model is captured by $P(z_k|x_k, m_{0:n})$, i.e. the probability distribution of making the current observation based on the current state and map [5]. The process described thus far applies to *direct* SLAM algorithms, which estimate positions of discrete features in the environment. This is not the case for *indirect* SLAM algorithms, e.g. LSD-SLAM [6] which observes image

intensities. Note that the algorithm will not actually know the true value of $m_{0:n}$; it will only be able to make estimates.

As previously mentioned, the goal of SLAM is to estimate of the state of the world, i.e. $P(x_k, m_{0:n}|z_{0:k}, u_{0:k}, x_0)$. This is calculated using two operations: prediction and correction (referred to by some as time-update and measurement-update). The prediction step [5] is carried out as a function of the motion model, previous state, and previous world model, i.e.

$$P(x_k, m_{0:n}|z_{0:k-1}, u_{0:k}, x_0) = \int P(x_k|x_{k-1}, u_k)P(x_{k-1}, m_{0:n}|z_{0:k-1}, u_{0:k-1}, x_0)\mathrm{d}x_{k-1}. \quad (2.5)$$

The correction step [5] refines the prediction based on the information provided by the new observation z_k:

$$P(x_k, m_{0:n}|z_{0:k}, u_{0:k}, x_0) = \frac{P(z_k|x_k, m_{0:n})P(x_k, m_{0:n}|z_{0:k-1}, u_{0:k}, x_0)}{P(z_k|z_{0:k-1}, u_{0:k})}. \quad (2.6)$$

Thus we now have a framework for tracking estimates and uncertainties over the course of a SLAM algorithm.

2.2. History of SLAM

The SLAM problem was originally formulated in 1986 by Smith and Cheeseman [7]. The authors drew from Kalman filter theory to estimate the position of a system over a series of approximate transforms. A transform $f : \mathbb{R}^n \to \mathbb{R}^n$ between deterministic variables x, $y \in \mathbb{R}^n$ can be written in the form

$$y = f(x). \quad (2.7)$$

If X is a random variable, then it is important to track both the mean $\bar{x} = \mathbb{E}[X]$ and the covariance $C(X)$. The context of SLAM is no exception, as conservative estimates of error can rapidly accumulate. Without loss of generality, we must assume that the output Y of the transform of a random variable X is, itself, a random variable. Per [8], we can relate $\bar{y}$

and $\bar{x}$ approximately by linearizing $Y = f(X)$ about $X = \bar{x}$:

$$Y = \sum_{n=0}^{\infty} \left. \frac{\mathrm{d}^{(n)} f(X)}{\mathrm{d}X^n} \right|_{X=\bar{x}} \frac{(X - \bar{x})^n}{n!} \approx f(\bar{x}) \tag{2.8}$$

The authors of [7] discuss two operations for combining approximate transforms: compounding and merging. Compounding is used to address approximate transforms in series. Consider the transforms $f_1(X)$ and $f_2(X)$. The expected value of the compounded transform $f_{2,1}(X) = f_2(f_1(X))$ is given by the equation

$$\mathbb{E}[f_{2,1}(X)] = f_{2,1}(\bar{x}) = f_2(f_1(\bar{x})). \tag{2.9}$$

To approximate the overall covariance, we first linearize the model by finding the Jacobians of $f_1(X)$ and $f_{2,1}(X)$:

$$J_{f_1} = \left. \frac{\partial f_1(X)}{\partial X} \right|_{X=\bar{x}}, \tag{2.10}$$

$$J_{f_{2,1}} = \left. \frac{\partial f_2(f_1(X))}{\partial f_1(X)} \right|_{X=\bar{x}}. \tag{2.11}$$

The linearized approximation is valid when the variance in the distribution of the state is small. Specifically, [7] suggests that the Jacobian be smooth over an interval the length of one standard deviation about the mean. The covariance $C_{2,1}(X)$ of $f_{2,1}(X)$ is then given by

$$C_{2,1}(X) = J_1 C_1(X) J_1^T + J_{2,1} C_2(X) J_{2,1}^T. \tag{2.12}$$

Together, (2.9) and (2.12) allow one to maintain a running estimate of the position of a system as well as the confidence in that prediction. Intuitively, compounding a sequence of transforms will never produce a result with lower uncertainty than any of its components. If the errors are assumed to be Gaussian, then one also obtains the probability distribution of the state of the system, over time, i.e. the motion model outlined in Section 2.1.

The second operation of interest is merging, which deals with approximate transforms in parallel. A set of approximate transforms is said to be parallel if all elements in the set describe a relation between the same pair of frames. Consider two parallel transforms $g_1(X)$ and $g_2(X)$, as well as $g_{1|2}(X)$, the result of merging g_1 and g_2. Drawing from Kalman filter theory, we define a Kalman gain factor K_1 as:

$$K_1 = C_1(X)[C_1(X) + C_2(X)]^{-1}. \tag{2.13}$$

Following [7], the expected value $\mathbb{E}[g_{1|2}(X)]$ is given by

$$\mathbb{E}[g_{1|2}(X)] = g_{1|2}(\bar{x}) = g_1(\bar{x}) + K_1\big(g_2(\bar{x}) - g_1(\bar{x})\big), \tag{2.14}$$

and the merged covariance $C_{1|2}(X)$ is given by

$$C_{1|2}(X) = C_1(X) - K_1 C_1(X). \tag{2.15}$$

Whereas compounding a set of approximate transforms leads to increased uncertainty, merging can decrease uncertainty.

The merging operation is used to build the observation model described in Section 2.1. Consider a robot that observes and records the apparent position of a landmark before moving to a new position and repeating the observation. There are now two parallel transforms to the landmark: one from the initial position and one from the current position. Merging these transforms will produce a more accurate estimate of where the landmark actually is. A reasonable assumption is that the compounding error in the position estimate of the robot will be much greater than the sensor error of measuring the position of the landmark relative to the observer. Hence, increasing the accuracy of the estimated position of a landmark will also increase the accuracy of the estimated position of the robot.

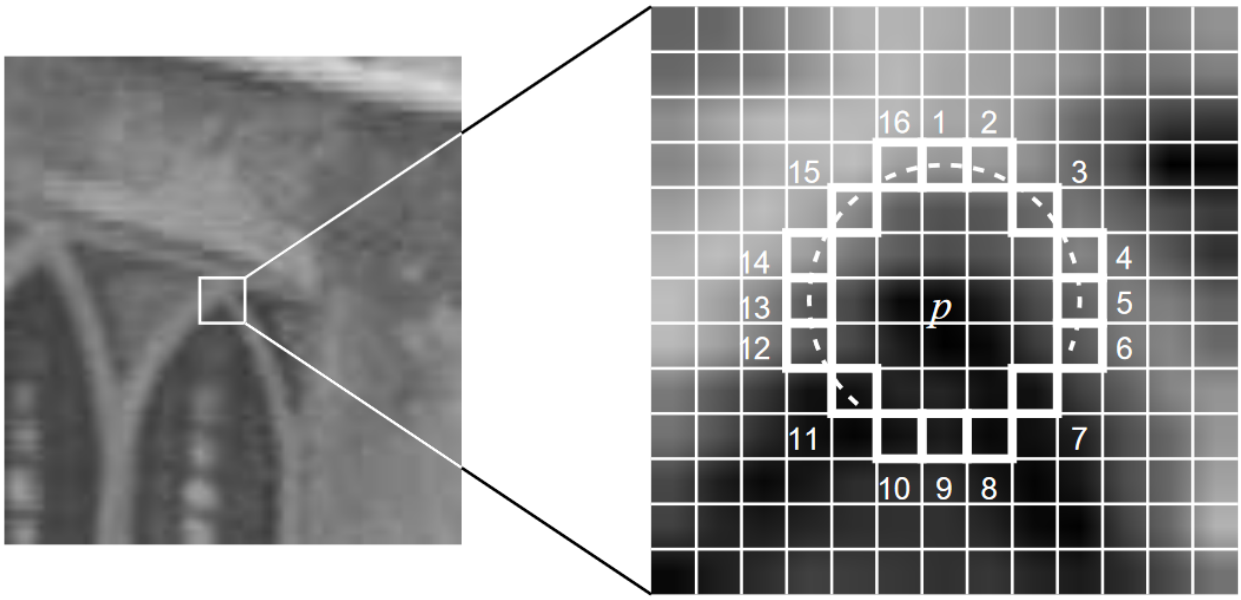


Figure 2.1: A FAST feature and its Bresenham circle [10].

2.3. Visual SLAM

Visual navigation requires some form of image processing to identify landmarks. Many early feature detectors operated by calculating the intensity gradient of an input image. The Harris corner detector [9], for example, compared the eigenvalues of a squared image intensity matrix to identify corners and edges. Originally, these gradient methods were developed for the structure-from-motion problem of computer science. The goal, then, was to reconstruct a three-dimensional map using a two-dimensional video or series of images. There is substantial overlap between this problem and that of SLAM, but one key difference is that the former need not operate in real time as the latter does. Thus, for the sake of structure-from-motion, it did not matter that the gradient methods tended to be slow.

The Features from Accelerated Segment Test (FAST) detector [10] operates using a different principle. Rather than computing the intensity gradient of an image, the FAST detector compares each patch to a Bresenham circle [11] centered about that patch. If a number n of contiguous patches are all darker or all brighter than the center patch, then the center

patch is determined to be a FAST feature. Figure 2.1 shows an example of a FAST feature. The radius r of the Bresenham circle and the number of contiguous pixels n are tunable parameters—common implementations use $r = 3$ and $n = 9$ or $n = 12$. FAST features can be extracted from an image much more rapidly than gradient-based features, since it is often easy to detect that a patch is not a FAST feature. Specifically, any arc of length 12 will necessarily pass through three of four cardinal points, i.e. those directly above, below, left of, and right of the center. Thus, an initial test of only those four points can be used to rule out a FAST feature candidate.

The FAST feature detector outputs binary information: a patch either is or is not a FAST feature. One consequence of this is that FAST features are identical. This is particularly problematic in the context of SLAM, where motion estimates are made by tracking the change in position of a feature from one image to the next. In the case of indistinguishable features, the computational complexity of this problem precludes real-time, onboard execution—foremost because of the need to iterate through every feature in one image to associate it with satisfactorily many features in the other. This process is further stymied by the appearance and disappearance of features between frames. A superior solution is found in providing each feature with a descriptor of some sort. A Harris feature [9], for example, could be associated with the eigenvalues and eigenvectors of its intensity gradient.

Ideally, a feature descriptor would be invariant to both rotation along the roll axis and changes in lighting within a scene. Further, it would be robust to scaling and slight deformations, so that a small change in the viewing angle of a feature would result in a small change in the description of the feature. In opposition to these properties is the desire for descriptions of features to be sufficiently unique that no incorrect pairings of feature points are made, and no pairing is made when a feature is absent in one image. Given the conflicting ideals of a feature descriptor, a number of different solutions have been developed. The

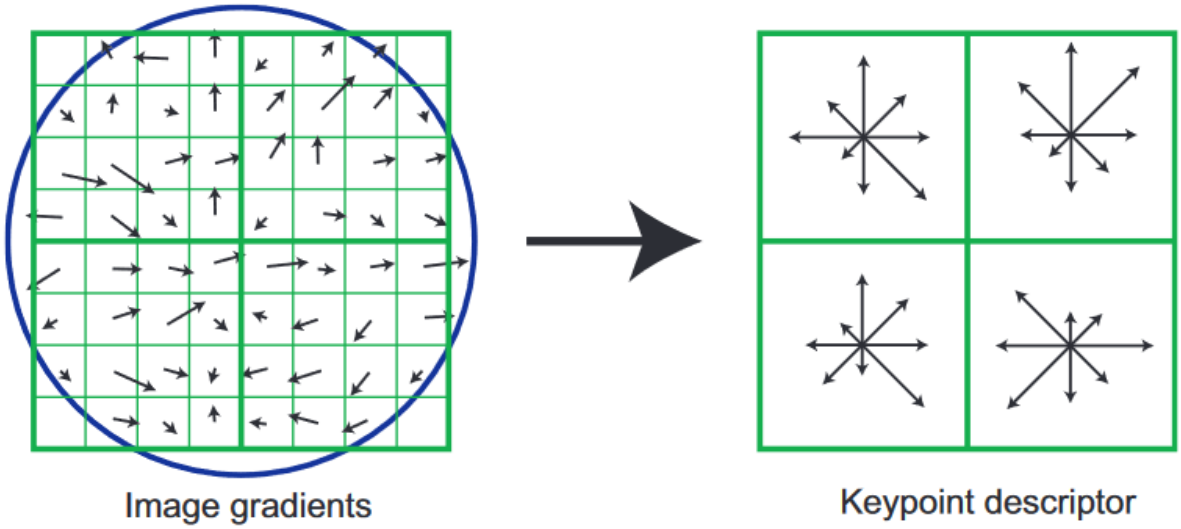


Figure 2.2: Gradient orientation histograms describing a SIFT feature [13].

scale-invariant feature transform (SIFT) feature descriptor [12], for example, uses builds a description of a feature by creating a histogram of intensity gradient orientations within a neighborhood around a feature, as shown in Figure 2.2. SIFT enjoyed a period of popularity due to its synergy with gradient-based feature detectors such as the Harris corner detector described above.

For real-time applications, the ideal is eschewed for the "good enough" in the name of runtime performance. In regard to detection, the FAST feature detector runs roughly twenty times faster than the Harris detector [10], since it avoids computing the gradient of an image at every point. This advantage would be diminished, however, in selecting a gradient-based descriptor like SIFT; thus, we turn our attention to the Binary Robust Independent Elementary Features (BRIEF) feature descriptor [14]. The BRIEF descriptor considers a region—e.g. a square or a rasterized circle—around a feature and then makes a number (typically 128 or 256) of pairwise comparisons between the intensities of pixels within the region. Since the comparison is binary, the result of each can be stored using a single bit. Consequently, BRIEF is very efficient in regard to runtime and memory usage.

On the other hand, since the compared pixels are found via their coordinates relative to the feature point, BRIEF is extremely variant to rotation.

The rotational variance of a BRIEF feature is not an insurmountable problem: if one can formulate a dominant angle describing the feature, it can be rotated appropriately prior to the application of the BRIEF descriptor. For example, the angle could be determined by the direction of the intensity gradient at the feature point. Alternatively, in keeping with the goal of avoiding calculation of gradients, we can orient the feature based on the intensity centroid in a neighborhood around the patch.

Combining the FAST detector and the BRIEF descriptor with the orientation method noted above gives us Oriented FAST and Rotated BRIEF (ORB) [15]. The relatively low computational costs associated with ORB allow it to run with higher speed and on higher resolution images compared to contemporary alternatives. It is not surprising, then, that a high-performance visual SLAM algorithm would incorporate this feature detector-descriptor. ORB-SLAM is exactly that: a modern, high-performance algorithm well-suited for mobile platforms such as robots or cellphones.

Through the process outlined in Section 2.2, ORB-SLAM can compute both an initial motion estimate between frames and the uncertainty corresponding to that estimate. The hypothetical shift in ORB features that would result from such a motion is then computed and reconciled with the actual observed position of ORB features in such a way that the estimate error is minimized. This revised motion estimate is then accepted, and new ORB features are added to the world map accordingly.

During the mapping process, input frames will periodically be designated as keyframes. Because it is computationally expensive to compare each input frame to the one directly preceding it, inputs are compared to the most recent keyframe instead. ORB-SLAM also keeps track of which ORB features are visible within each keyframe via a covisibility graph.

As the map built by ORB-SLAM grows, the cost in time of operating on that map grows. Early SLAM algorithms did not account for this; thus, they tended to slow down as the map expanded. The first algorithm to provide a solution to this was Parallel Tracking And Mapping PTAM [16], which, as its name would suggest, uses separate threads to handle tracking and mapping in parallel. This approach ensures that only the mapping thread slows down as the map expands; the odometry provided by the tracking thread will not. ORB-SLAM uses this approach and expands upon it further by introducing a third thread to perform loop closure.

Though SLAM algorithms are designed to minimize uncertainty in the map, they do not wholly eliminate it—in particular, accumulated error over time can result in drift. This can become particularly problematic when a system returns to a previously visited area: if the system fails to realize that it has already built a map of this area, old features will be added once again to the map in positions offset by the drift error. It is thus desirable to have a method of *closing the loop*, i.e. adjusting the additions made to the map since the original visit according to the terminal condition that the system ends its trajectory where the loop began.

Detecting a loop demands searching through previous keyframes in search of those containing a pattern of features with descriptors consistent with those in the current frame. In the even that a loop *is* detected, the estimates of trajectory and feature locations contained within the loop are reevaluated such that the estimate error is minimized in light of the new terminal condition. Not only is this is a computationally demanding process, but it is also independent of the tracking process, since the latter can be restarted from the beginning of the loop. Hence, a third thread is allocated to the detection and handling of loop closure.

Chapter 3: Local Trajectory Generation

3.1 Trajectory Generation using Model Predictive Control

A problem central to tactical operation is the generation of a trajectory to take the system from its current position to the mission goal. In selecting an algorithm to generate this trajectory, we first consider the features and qualities that an ideal trajectory generator would have. Foremost, it should never fail to generate a trajectory to the mission goal, provided at least one viable trajectory exists. Of equal importance is the requirement that the algorithm never generates a trajectory that would cause the system to crash into a wall or enter some other such inadmissible state. The trajectories generated by the algorithm should be tunable in some way to allow for versatility in diverse indoor and outdoor settings, and they should be optimal with respect to the tuning parameters. Finally, the algorithm must be fast and compact enough to be run on an onboard computer, such as an ODROID-XU4 [27]. To this end, we turn to Model Predictive Control (MPC).

Traditionally, MPC is used to compute a control policy optimal with respect to some cost function over a finite time horizon, while satisfying constraints imposed on both the system and the controller [28]. MPC additionally calculates future states over this time horizon to which the future control inputs are applied. As such, it can be used to generate an optimal state trajectory just as easily as it generates an optimal control policy. It is, however, not without shortcomings. First, solving MPC takes a relatively long time; hence, it is typically not applied to systems that require frequent updates to the reference signal. Alternatively, solutions can be precomputed and retrieved via a lookup table keyed to the initial state. Neither case bodes well for online use in aerial robotics, where time is critical and processing power is limited.

A more recent technique is to reformulate MPC as a quadratic program, which can be solved much more rapidly [4]. While this fast MPC algorithm alleviates the issue of poor runtime performance, it brings with it limitations and concessions. The technique requires a linear model with convex constraints—quadrotors are nonlinear systems and both indoor and outdoor environments tend to be non-convex. Further, optimality is no longer guaranteed—the algorithm is generally terminated prior to convergence, yielding only an approximately optimal solution. However, as noted in [4], the errors induced by early termination result in little—or in some cases, no—increase in the cost of the trajectory.

Finally, the requirement of a convex operating region can be satisfied by generating a locally convex region about the system, which can be updated between MPC iterations as the system moves about its environment. We thus proceed with the use of MPC, having addressed the issues incurred by it.

3.2 Formulation of Model Predictive Control

Following along with [4] we now formulate and solve model predictive control as a quadratic program. We begin with an undisturbed discrete-time linear system over a finite time horizon:

$$x_{\tau+1} = Ax_\tau + Bu_\tau \ , \quad \tau \in \{t \ , \ t+1 \ , \ \ldots \ , \ t+T-1\}, \tag{3.1}$$

in which T denotes the time horizon, $x_\tau \in \mathbb{R}^n$ and $u_\tau \in \mathbb{R}^m$ denote the state and control input at time step τ respectively, $A \in \mathbb{R}^{n\times n}$, and $B \in \mathbb{R}^{n\times m}$.

Though a useful model for analysis, realistic applications teem with disturbances: from applying discrete-time models to real-time systems, from applying linear models to nonlinear systems, from uncertainties in state estimation, and so on. Hence, we introduce the random variable $\omega(\tau)$ to account for all disturbances. By assumption, $\omega(\tau)$ is independent and

identically distributed with known distribution throughout the time horizon such that $\bar{\omega} = \mathbf{E}[\omega(\tau)]$ is known as well. For consistency with [4] as well as our implementation, $\bar{\omega}$ is assumed to be constant. It should be noted, however, that this assumption is not necessary: the techniques for solution presented in this thesis can be applied in the case of varying disturbances, provided the expected value is known at all steps of the time horizon. The model is accordingly refined:

$$x_{\tau+1} = Ax_\tau + Bu_\tau + \bar{\omega} \ , \quad \tau \in \{t \ , \ t+1 \ , \ \ldots \ , \ t+T-1\}. \tag{3.2}$$

We introduce two classes of tuning parameters by which we can influence the behavior of the system: costs and constraints. In the case of the former, we use a quadratic stage cost function, which can be given as

$$l(x_\tau, u_\tau, \tau) = \begin{bmatrix} x_\tau \\ u_\tau \end{bmatrix}^T \begin{bmatrix} Q_\tau & N_\tau \\ N_\tau^T & R_\tau \end{bmatrix} \begin{bmatrix} x_\tau \\ u_\tau \end{bmatrix} + q_\tau^T x_\tau + r_\tau^T u_\tau \ , \ \begin{bmatrix} Q_\tau & N_\tau \\ N_\tau^T & R_\tau \end{bmatrix} \geq 0, \tag{3.3}$$

with $Q_\tau = Q_\tau^T \in \mathbb{R}^{n \times n}$, $N_\tau \in \mathbb{R}^{n \times m}$, and $R_\tau = R_\tau^T \in \mathbb{R}^{m \times m}$. Typically the state cost terms Q_τ and q_τ are held constant until the final timestep, i.e.

$$Q_\tau = \begin{cases} Q, \ t \leq \tau < t+T \\ Q_f, \ \tau = t+T \end{cases}, \tag{3.4}$$

$$q_\tau = \begin{cases} q, \ t \leq \tau < t+T \\ q_f, \ \tau = t+T \end{cases}. \tag{3.5}$$

This allows the designer to distinguish desired transient behavior from steady-state behavior, which is of particular relevance to guidance problems, wherein it is typical to want to arrive at a destination quickly while not overshooting the goal. We proceed thusly for the remainder of this thesis. Additionally, since the final control input $u(t+T)$ has no effect on the system within the time horizon, R_τ and r_τ are taken without loss of generality to be zero at $\tau = t+T$ and is typically held constant otherwise. In some applications, there is a coupling cost N_τ

between the state and control input of a system. A control law for a submarine, for example, might impose a high cost to opening its hatches when the system is in a submerged state. However, for many applications—ours included—there is no need for coupling; as such, N_τ is taken to be zero across the whole time horizon. With these cost matrices, it becomes convenient to separate the stage cost function into two time-independent functions:

$$l(x_\tau, u_\tau) = x_\tau^T Q x_\tau + u_\tau^T R u_\tau + q^T x_\tau + r^T u_\tau \ , \ \tau \in \{t \ , \ t+1 \ , \ \dots \ , \ t+T-1\}, \tag{3.6}$$

$$l_f(x_{t+T}) = x_{t+T}^T Q_f x_{t+T} + q_f^T x_{t+T}. \tag{3.7}$$

The second class of tuning parameters for model predictive control is constraints. One constraint comes from our disturbed linear model: we impose that each state must follow from the previous, given a control input and disturbance. This is an equality constraint—each solution to the MPC problem will satisfy this constraint exactly. Of interest as well are inequality constraints, which can establish solution intervals over which the constraints are satisfied. In particular, we express linear inequality constraints as

$$F_x x_\tau + F_u u_\tau \leq f, \tag{3.8}$$

wherein $F_x \in \mathbb{R}^{l \times n}$, $F_u \in \mathbb{R}^{l \times m}$, and $f \in \mathbb{R}^l$. The state and control aspects of each constraint manifest as individual rows of F_x and F_u respectively, and l denotes the total number of constraints placed on the system. If the state constraints are independent of the control constraints and vice versa, the problem is said to be state-control separable, and we can rewrite the constraints as

$$F_x x_\tau \leq f_x, \quad F_u u_\tau \leq f_u \tag{3.9}$$

with $f_x \in \mathbb{R}^{l_x}$ and $f_u \in \mathbb{R}^{l_u}$. Since each constraint is hyperplanar, our state space $\mathcal{D}_x = \{x_\tau : F_x x_\tau \leq f_x\}$ and control space $\mathcal{D}_u = \{u_\tau : F_u u_\tau \leq f_u\}$ are both convex. Henceforth we assume that each is nonempty, i.e. there exists at least one state and control action that satisfies all of their respective constraints.

Traditionally, the above constraints are said to be hard constraints—they must be satisfied unconditionally; otherwise, the system has entered some inadmissible state $x \notin \mathcal{D}_x$ or been subject to an inadmissible control input $u \notin \mathcal{D}_u$.

In addition to hard constraints, we follow [19] in implementing soft constraints in our formulation of MPC. Unlike hard constraints, soft constraints need not be satisfied—instead, a penalty is applied for violated constraints. In practice, the system is thus reluctant to violate a soft constraint, but will do so if the violation is the best possible action. In quadrotors, for example, one might apply a soft constraint on control inputs above a certain threshold to incentivize efficient flight while still permitting aggressive but inefficient evasive maneuvers. We denote soft constraints with tildes, i.e.

$$\tilde{F}_x x_\tau \leq \tilde{f}_x, \quad \tilde{F}_u u_\tau \leq \tilde{f}_u \tag{3.10}$$

with $\tilde{f}_x \in \mathbb{R}^{\tilde{l}_x}$ and $\tilde{f}_u \in \mathbb{R}^{\tilde{l}_u}$. In keeping with [19], we use the Kreisselmeier-Steinhauser function to quantify the penalty $\theta(x_\tau, u_\tau)$ for violating soft constraints as follows:

$$\theta(x_\tau, u_\tau) = \sum_{i=1}^{\tilde{l}_x} \frac{1}{\rho} \log\left(1 + \mathrm{e}^{\rho(\tilde{F}_{x,i} x_\tau - \tilde{f}_{x,i})}\right) + \sum_{i=1}^{\tilde{l}_u} \frac{1}{\rho} \log\left(1 + \mathrm{e}^{\rho(\tilde{F}_{u,i} u_\tau - \tilde{f}_{u,i})}\right), \tag{3.11}$$

in which ρ is a user-defined weight that increases by a factor of γ after each search step—see (3.31)–(3.40)—$\tilde{F}_{x,i}$ and $\tilde{F}_{u,i}$ denote the ith row of $\tilde{F}_x$ and $\tilde{F}_u$ respectively (and hence, the ith constraint imposed on the system), and $\tilde{f}_{x,i}$ and $\tilde{f}_{u,i}$ denote the ith element of the respective bounds on the aforementioned constraints. For cases wherein only one argument is provided—either a state vector or a control vector—the summation involving the withheld argument is taken to be zero.

Having defined the system and tuning parameters, we now formulate the MPC problem

with soft constraints in full:

$$\begin{aligned}
\text{minimize} \quad & J(z_t) = l_f(x_{t+T}) + \theta(x_{t+T}) + \sum_{\tau=t}^{t+T-1} l(x_\tau, u_\tau) + \theta(x_\tau, u_\tau) \\
\text{subject to} \quad & F_x x_\tau \leq f_x \ , \ F_u u_\tau \leq f_u \ , \ \tau \in \{t \ , \ t+1 \ , \ \ldots \ , \ t+T\} \\
& x_{\tau+1} = Ax_\tau + Bu_\tau + \bar{\omega} \ , \ \tau \in \{t \ , \ t+1 \ , \ \ldots \ , \ t+T-1\}.
\end{aligned} \tag{3.12}$$

This is a quadratic program, a class of optimization problems for which many well-studied numerical methods of solution exist [23, 29]. As an initial solution to this quadratic program, we select the infeasible interior point method [26]. Following that, we refine the scope of our problem and utilize the structures arising from both the quadratic program and the infeasible interior point method to reduce the time complexity of the solution.

3.3 Solution of MPC via Infeasible Interior Point Method

As discussed previously, historic solutions to the MPC problem are precluded from online use owing to steep scaling of computation time with respect to the length of the time horizon. However, as noted in [23] and [24], the structures present in these inefficient solutions can be exploited to reduce the time complexity from $O(T^3(n+m)^3)$ to $O(T(n+m)^3)$. In addition to reducing the order of the time complexity with respect to T from cubic to linear, the solution presented in [4] demonstrates that the solution algorithm can be halted well before convergence without significant loss of quality in performance. The solution presented below follows largely from [4], with modifications for inclusion of soft constraints as described in [19].

We begin by collecting the problem data into block matrices and vectors to represent the problem over the full time horizon. Note that the block matrices that follow will be sparse. In developing source code to implement the solution, special sparse block matrix data structures should be used to prevent waste of computational time by naively evaluating

products of zero submatrices. Alternatively, the problem data could be collected into dense block matrices so that the zero blocks are removed. In this thesis the former formulation is presented, but it should be noted that the difference between it and the latter is purely aesthetic. The state and control vectors over the time horizon—sans the initial state x_t—are first collected into a single vector z_t as such:

$$z_t = \begin{bmatrix} u_t \\ x_{t+1} \\ \vdots \\ u_{t+T-1} \\ x_{t+T} \end{bmatrix} \in \mathbb{R}^{T(m+n)}. \tag{3.13}$$

It was previously stated that our implementation is state-control separable in both cost and constraints. We can thus collect the quadratic costs into an alternating block diagonal matrix H and the linear costs into an alternating vector g:

$$H = \begin{bmatrix} R & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & Q & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & R & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & Q & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & R & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 & Q_f \end{bmatrix} \in \mathbb{R}^{T(m+n)\times T(m+n)}, \tag{3.14}$$

$$g = \begin{bmatrix} r \\ q \\ r \\ \vdots \\ q \\ r \\ q_f \end{bmatrix} \in \mathbb{R}^{T(m+n)}. \tag{3.15}$$

Similarly, the hard and soft constraint matrices are collected into P and $\tilde{P}$ respectively,

with corresponding bounds h and $\tilde{h}$:

$$P = \begin{bmatrix} F_u & 0 & \dots & 0 & 0 \\ 0 & F_x & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \\ 0 & 0 & \dots & F_u & 0 \\ 0 & 0 & \dots & 0 & F_x \end{bmatrix} \in \mathbb{R}^{T(l_u+l_x)\times T(n+m)}, \tag{3.16}$$

$$h = \begin{bmatrix} f_u \\ f_x \\ \vdots \\ f_u \\ f_x \end{bmatrix} \in \mathbb{R}^{T(l_u+l_x)}, \tag{3.17}$$

$$\tilde{P} = \begin{bmatrix} \tilde{F}_u & 0 & \dots & 0 & 0 \\ 0 & \tilde{F}_x & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \\ 0 & 0 & \dots & \tilde{F}_u & 0 \\ 0 & 0 & \dots & 0 & \tilde{F}_x \end{bmatrix} \in \mathbb{R}^{T(\tilde{l}_u+\tilde{l}_x)\times T(n+m)}, \tag{3.18}$$

$$\tilde{h} = \begin{bmatrix} \tilde{f}_u \\ \tilde{f}_x \\ \vdots \\ \tilde{f}_u \\ \tilde{f}_x \end{bmatrix} \in \mathbb{R}^{T(\tilde{l}_u+\tilde{l}_x)}. \tag{3.19}$$

Finally, the matrix C and vector b_t are constructed for representation of the disturbed linear model over the time horizon:

$$C = \begin{bmatrix} -B & I & 0 & 0 & \dots & 0 & 0 & 0 & 0 & 0 \\ 0 & -A & 0 & 0 & \dots & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -B & I & \dots & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -A & \dots & 0 & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \\ 0 & 0 & 0 & 0 & \dots & -B & I & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & -A & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & -A & -B & I \end{bmatrix} \in \mathbb{R}^{Tn \times T(n+m)}, \tag{3.20}$$

$$b_t = \begin{bmatrix} Ax_t + \bar{\omega} \\ \bar{\omega} \\ \vdots \\ \bar{\omega} \\ \bar{\omega} \end{bmatrix} \in \mathbb{R}^{Tn}. \tag{3.21}$$

Using these new representations of the problem data, we can reformulate the MPC problem as:

$$\text{minimize} \quad J(z_t) = z_t^T H z_t + \theta(z_t) \quad \text{subject to} \quad Pz_t \leq h \ , \ Cz_t = b_t, \tag{3.22}$$

in which the use of the Kreisselmeier-Steinhauser function $\theta(\cdot)$ is extended to z_t as such:

$$\theta(z_t) = \sum_{i=1}^{T(\tilde{l}_u + \tilde{l}_x)} \frac{1}{\rho} \log\left(1 + \mathrm{e}^{\rho(\tilde{P}_i z_t - \tilde{h}_i)}\right) \tag{3.23}$$

In accordance with the infeasible start primal barrier method [26] the hard inequality constraints are moved into the cost function via a log barrier function $\phi(\cdot)$, which yields the following revised QP:

$$\text{minimize} \quad J(z_t) = z_t^T H z_t + \kappa\phi(z_t) + \theta(z_t) \quad \text{subject to} \quad Cz_t = b_t, \tag{3.24}$$

$$\phi(z_t) = \begin{cases} \displaystyle\sum_{i=1}^{T(l_u + l_x)} \log \frac{1}{h_i - P_i z_t} \ , & Pz_t < h \\ \infty \qquad\qquad , & Pz_t \not< h \end{cases}, \tag{3.25}$$

where κ is a positive barrier parameter with a user-defined initial value. This equation is to be solved iteratively—at each iteration, κ will be divided by some factor. As such, supposing the cost is finite, the hard constraints will be satisfied at every iteration; further, as κ approaches zero, the cost of (3.24) approaches that of (3.22).

From Fermat's theorem [22], we know that all critical points of a function have a derivative of zero. Thus our general approach to solving the QP will involve equating the derivative of the cost to zero. We note that though (3.22) is not guaranteed to have a unique minimum point, the structure of its cost function does guarantee that any critical point is a global minimum. Thus, since (3.24) converges to (3.22) as κ approaches zero, the same method can be applied to the latter equation to find a global minimum.

Following from (3.24), the gradient of the cost function J is given by

$$\nabla J(z_t) = 2Hz_t + g + \kappa P^T d + \tilde{P}^T \tilde{d}, \tag{3.26}$$

$$d \in \mathbb{R}^{T(m+n)} : d_i = \begin{cases} \dfrac{1}{h_i - P_i z_t} & , \quad Pz_t < h \\ \infty & , \quad Pz_t \not< h \end{cases}, \tag{3.27}$$

$$\tilde{d} \in \mathbb{R}^{T(m+n)} : \tilde{d}_i = \frac{\exp\left(\rho(\tilde{P}_i^T z_t - \tilde{h})\right)}{1 + \exp\left(\rho(\tilde{P}_i^T z_t - \tilde{h})\right)}. \tag{3.28}$$

The primal residuals r_p and dual residuals r_d of the optimization problem, then, are given by

$$r_d = \nabla J(z_t) + C^T \nu, \tag{3.29}$$

$$r_p = Cz - b_t, \tag{3.30}$$

where $\nu \in \mathbb{R}^{Tn}$ is a vector of Lagrange multipliers associated with the equality constraint [17]. A solution is optimal with satisfied constraints when these residuals are zero. Such a solution is found iteratively by computing search steps Δz and $\Delta \nu$ by which to adjust z and

ν, respectively. The search steps are found via the gradient of (3.29) and (3.30):

$$\begin{bmatrix} \nabla^2 J(z_t) & C^T \\ C & 0 \end{bmatrix} \begin{bmatrix} \Delta z \\ \Delta \nu \end{bmatrix} = - \begin{bmatrix} r_d \\ r_p \end{bmatrix}, \tag{3.31}$$

in which the Hessian of the cost function, which is block diagonal, is given by

$$\nabla^2 J(z_t) = 2H + \kappa P^T \mathrm{diag}(d)^2 P + \rho \tilde{P}^T \mathrm{diag}(\hat{d}) \tilde{P}, \tag{3.32}$$

$$\hat{d} \in \mathbb{R}^{Tn} : \hat{d}_i = \frac{\exp\left(\rho(\tilde{P}_i^T z_t - \tilde{h})\right)}{\left(1 + \exp\left(\rho(\tilde{P}_i^T z_t - \tilde{h})\right)\right)^2}. \tag{3.33}$$

This system is solved efficiently using the Schur complement [21]. Consider the general system of equations

$$\begin{bmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}, \tag{3.34}$$

in which M_{11} is invertible. The Schur complement S is given by

$$S = M_{22} - M_{21} M_{11}^{-1} M_{12}. \tag{3.35}$$

Then the solution to the system presented in (3.34), provided the necessary inversions are possible, is given by

$$y = S^{-1}(\beta - M_{21} M_{11}^{-1} \alpha), \tag{3.36}$$

$$x = M_{11}^{-1}(\alpha - M_{12} y). \tag{3.37}$$

Applying (3.34)–(3.37) to the system presented in (3.31) reduces the update step for MPC to the following three equations:

$$Y = -C\left(\nabla^2 J(z_t)\right)^{-1} C^T, \tag{3.38}$$

$$Y \Delta \nu = -r_p + C^T \left(\nabla^2 J(z_t)\right)^{-1} r_d, \tag{3.39}$$

$$\Delta z = \left(\nabla^2 J(z_t)\right)^{-1} (-r_d - C^T \Delta \nu). \tag{3.40}$$

As noted above, $\nabla^2 J(z_t)$ is block diagonal. Its inverse, then, can be calculated by inverting each of the blocks on the diagonal. For convenience, we express these inverted blocks as

$$\left(\nabla^2 J(z_t)\right)^{-1} = \begin{bmatrix} \tilde{R}_0 & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & \tilde{Q}_1 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & \tilde{R}_1 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & \tilde{Q}_{T-1} & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & \tilde{R}_{T-1} & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 & \tilde{Q}_T \end{bmatrix}. \tag{3.41}$$

Due to the structure of the matrices C, H, P, and $\tilde{P}$, Y is symmetric and block tridiagonal. The blocks constituting can be calculated recursively:

$$Y_{11} = B\tilde{R}_0 B^T + \tilde{Q}_1, \tag{3.42}$$

$$Y_{ii} = A\tilde{Q}_{i-1}A^T + B\tilde{R}_{i-1}B^T + \tilde{Q}_i, \tag{3.43}$$

$$Y_{i,i+1} = -\tilde{Q}_i A^T, \tag{3.44}$$

$$Y_{i+1,i} = -A\tilde{Q}_i = Y_{i,i+1}^T. \tag{3.45}$$

Rather than invert Y, it is factored via Cholesky factorization. Since Y is symmetric, it there exists some lower triangular L such that $Y = LL^T$. Like Y, the blocks of L are calculated recursively:

$$L_{11}L_{11}^T = Y_{11}, \tag{3.46}$$

$$L_{ii}L_{i+1,i}^T = Y_{i,i+1}, \tag{3.47}$$

$$L_{ii}L_{ii}^T = Y_{ii} - L_{i,i-1}L_{i,i-1}^T. \tag{3.48}$$

Once L has been calculated, it is used to solve (3.39) for $\Delta\nu$. This solution, in turn, is applied to (3.40) to calculate Δz. These steps are then added to the current values of z and ν, and the process repeats until the residuals are satisfactorily close to zero.

3.4 Techniques and Approximations for Fast MPC Solution

Provided a feasible initial estimate of z_t, the solution will converge to an optimal control policy. The difference between a good initial estimate and a bad one is marked by the number of iterations required for convergence. For the first iteration, then, the initial estimate z_t^0 is constructed such that

$$z_t^0 = \begin{bmatrix} u_t^0 \\ x_{t+1}^0 \\ \vdots \\ u_{t+T-1}^0 \\ x_{t+T}^0 \end{bmatrix}, \tag{3.49}$$

$$x_{t+1}^0 = ... = x_{t+T}^0 = x_t, \tag{3.50}$$

$$u_t^0 = u_{t+1}^0 = ... = u_{t+T-1}^0 = 0, \tag{3.51}$$

since the initial position x_t is assumed to be known and feasible. For subsequent iterations, however, the solution z_τ^* can be used to form the initial guess $z_{\tau+1}^0$ for the next timestep such that

$$z_{\tau+1}^0 = \begin{bmatrix} u_{\tau+1}^* \\ x_{\tau+2}^* \\ \vdots \\ x_{\tau+T}^* \\ u_{\tau+T}^* \\ x_{\tau+T}^* \end{bmatrix}, \tag{3.52}$$

where the final state vector is repeated since the $x_{\tau+T+1}^*$ lies outside the time horizon of the previous iteration. This technique is known as "warm starting" [4] or "hot starting" [24].

3.5 Remarks on Constraint Generation and Operation in Nonconvex Regions

The implementation of model predictive control as a guidance algorithm enables the generation of trajectories to and from any point within the admissible state-space, i.e. the

operating region. However, the requirement that the operating region be convex is a crippling limitation that renders MPC alone insufficient for realistic guidance problems.

There are two approaches one could take to enable operation of MPC in non-convex environments. MPC could be generalized to extend the admissible class of constraints to include linear piecewise-defined constraints. More easily, though, we could vary the state constraints with respect to the state of the system such that the system operates across a sequence of locally convex operating regions. As noted in [4], [24], much of the problem data of MPC that are assumed to be constant can instead be modeled as functions of time without compromising the structure of the problem. Similarly, these data—F_x, $\tilde{F}_x$, f_x, and $\tilde{f}_x$ in particular—can vary with respect to the state.

The latter approach is implicitly implemented and successfully demonstrated in [20], wherein the authors provide different sets of spatial constraints prior to each call to the MPC algorithm based on the position of the system and obstacles in its vicinity. In a similar vein, we compute new bounds for each MPC call using the SemiDefinite Programming Algorithm (SDPA). Given an interior and an exterior set of points, SDPA will compute a polyhedron separating the two, provided a solution exists [25]. A thorough explanation of SDPA can be found in [18]. For a known map, the exterior set can be generated *a priori*; for an initially unknown map, the exterior set must be built as the environmental point cloud is generated using the visual navigation techniques outlined in Chapter 2.

Chapter 4: Global Path Planning

4.1 Motivation for Global Path Planning

The methodology presented in the preceding chapter enables the generation of local convex constraints for any region of a non-convex map, given an admissible initial state. This alone, however, does not guarantee that a trajectory can be generated between any pair of points within a connected non-convex map. Figure 4.1, below, demonstrates a failure case of the algorithm presented thus far.

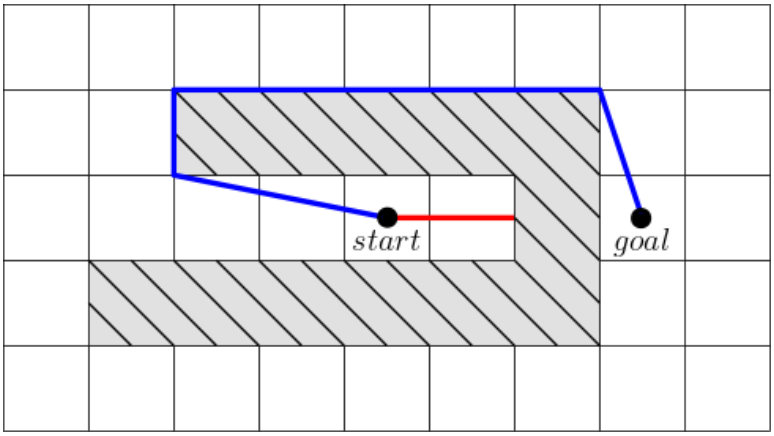


Figure 4.1: FMPC as implemented tries to minimize the distance of the system from the goal over a time horizon. For sufficiently small time horizons, the system will move directly toward the goal (red path) rather than move away from the goal to follow the shortest path to it (blue path).

Sufficiently large bowls are an example of a class of cases for which the system will fail to reach the goal point. Model predictive control guarantees us a cost-minimizing trajectory over a finite time horizon. In the situation presented above, the cost imposed by moving along the desirable trajectory—i.e. the one that brings the system to the goal—is higher than the cost of maintaining the current state of the system. Thus, the system will remain trapped in the bowl indefinitely. The susceptibility of the extended FMPC algorithm to

being trapped as shown marks it as a best-local greedy algorithm [30]—in general, since it is guaranteed to choose the locally optimal solution at each time step, it is not capable of escaping from a local minimum if doing so would require a locally suboptimal action. By intuition, we posit that for any finite time horizon, there exists some maximum bowl depth beyond which the system is unable to escape. It is of little interest to determine the value of this maximum depth, since the problem lies in its existence. By corollary, we note that this behavior would be absent in the case of an infinite time horizon: since the quadratic cost function is formulated such that it is uniquely minimized at the goal position, the cost of any trajectory that does not reach and remain at the goal position would be infinitely higher than one that does. Of course, such an algorithm would take infinitely long to compute; thus, it is nonviable as a solution. Hence, we instead opt to add a second layer to the navigation algorithm—a global path planner, the output of which will be locally optimized by the FMPC layer.

Though relatively young, the field of algorithmic pathfinding developed quickly on account of its relevance in the modern world. Day-to-day applications, for example, include finding the shortest route by car to a commuter's destination (and updating the path in the case of a wrong turn or missed exit), or intelligently pathing an agent toward the player in a video game.

At the heart of algorithmic pathfinding is the shortest path problem, which prompts: given a pair of nodes on a connected, weighted graph, find the path of lowest weight connecting the pair. Thus, every solution to the shortest path problem yields a path from the start to the goal. As such, it provides an excellent starting point for the development of our global path planner. Figure 4.2 provides an example of a graph to which the shortest path problem could be posed. We begin our investigation into a global path planner with the shortest path problem because every solution is guaranteed to generate a trajectory from

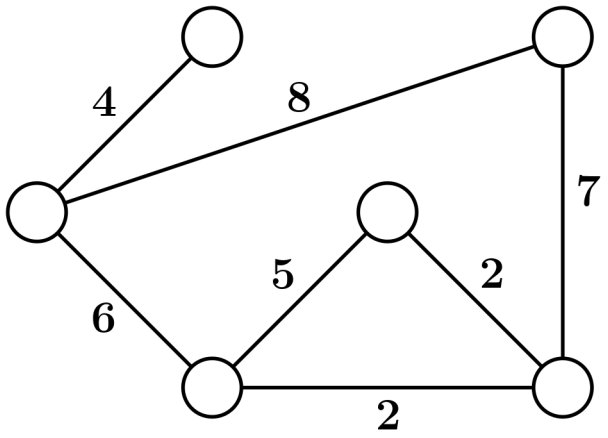


Figure 4.2: A connected, weighted graph.

the start to the goal.

4.2 Dijkstra's Algorithm

Though not the first solution to the shortest path problem [1], Dijkstra's algorithm—first published in 1959 [31]—is notable in that many pathfinding algorithms are built upon it, including the one ultimately implemented in this thesis.

Dijkstra's algorithm begins with the construction of two sets of nodes. The *frontier* is initially empty and will contain every immediately explorable node. The second, initially containing every node, is that of the *unseen* nodes. In this thesis we present a formulation of the algorithm in terms of *node* data structures. Each *node* will have a *cost* to reach from the starting point, a *branchList* of (*node*, *branchWeight*) pairs describing neighboring *nodes* as well as the weight of the branch connecting the two *nodes*, and a *prevNode* that describes the *node* immediately preceding this one on a path from the start to it. We also assume the availability of a *findMinCost*$(\cdot)$ function that can, given a set of *nodes*, find a *node* with a *cost* no greater than that of any other *node* in the set. Having established the relevant structures and functions, we present Dijkstra's algorithm in Algorithm 1.

The path generated from this formulation of Dijkstra's algorithm is obtained by tracing

Algorithm 1 Dijkstra's Algorithm [31]

```
1: selectedNode := start
2: remove start from unseen
3: while goal ≠ selectedNode do
4:     for all (neighbor, weight) ∈ selectedNode.branchList do
5:         costFromSelected := selectedNode.cost + weight
6:         if neighbor ∈ frontier then
7:             if costFromSelected < neighbor.cost then
8:                 neighbor.cost := costFromSelected
9:                 neighbor.prevNode := selectedNode
10:        else
11:            remove neighbor from unseen
12:            insert neighbor into frontier
13:            neighbor.cost := costFromSelected
14:            neighbor.prevNode = selectedNode
15:    selectedNode := findMinCost(frontier)
16:    remove selectedNode from frontier
```

the chain of previous nodes from the goal back to the start—this path is guaranteed to be a shortest path between the two. This assurance follows from the nature of the algorithm: at each iteration, a frontier node with the lowest cost to reach from the start is explored. In other words, a node is only explored via a shortest path to it. By extension, when the goal is explored, it is explored through a shortest path as well. The major weakness of Dijkstra's algorithm is that it is essentially a brute force search for the shortest path. The successor algorithm A* offers greatly improved performance through the use of a heuristic function to guide the search direction.

4.3 The A* Search Algorithm

The A* search algorithm, henceforth referred to simply as A*, was developed in 1968 to decrease the time to find the shortest path between nodes [35]. Whereas Dijkstra's algorithm explores nodes solely in order of increasing travel cost from the starting node, A* additionally considers a heuristic estimate of the remaining cost to the goal. The heuristic value of each

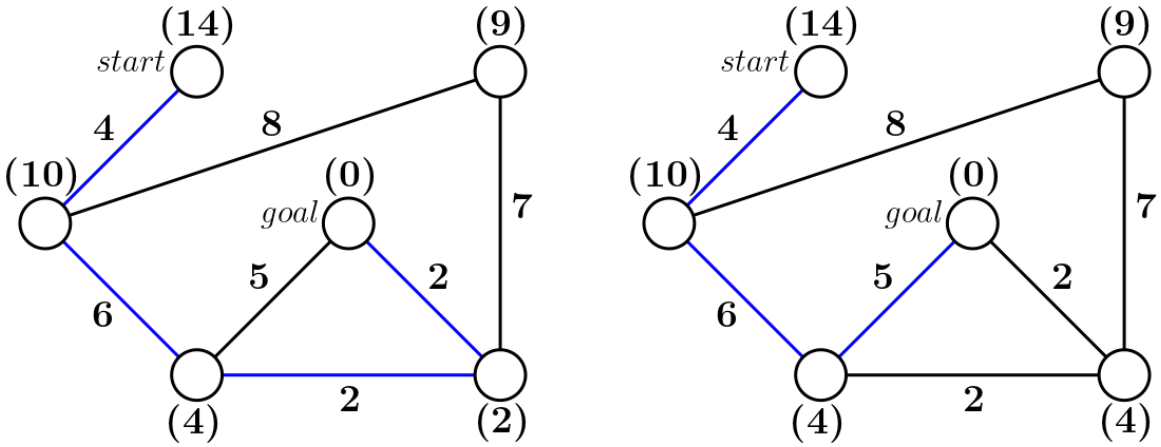
node in the frontier is added to the running cost, and the node with the lowest sum is selected for exploration. Formulating A* from Dijkstra's algorithm, then, is simply a matter of defining and implementing some heuristic function $heuristicEstimate(\cdot)$.

Algorithm 2 A* Algorithm [35]

```
1: selectedNode := start
2: remove start from unseen
3: while goal ≠ selectedNode do
4:     for all (neighbor, weight) ∈ selectedNode.branchList do
5:         costFromSelected := selectedNode.cost + weight + heuristicEstimate(neighbor)
6:         if neighbor ∈ frontier then
7:             if costFromSelected < neighbor.cost then
8:                 neighbor.cost := costFromSelected
9:                 neighbor.prevNode := selectedNode
10:        else
11:            remove neighbor from unseen
12:            insert neighbor into frontier
13:            neighbor.cost := costFromSelected
14:            neighbor.prevNode = selectedNode
15:    selectedNode := findMinCost(frontier)
16:    remove selectedNode from frontier
```

In comparing A* to Dijkstra's algorithm, we note that the former can be considered to be a generalization of the latter. Specifically, Dijkstra's algorithm is the A* algorithm with a heuristic function identically equal to zero.

As previously mentioned, the heuristic function estimates the cost to reach the goal from a particular node. Excepting trivial cases, the nature of this estimate is specific to the type of graph to which A* is being applied. In the case of this thesis, for example, the graph is a spatial grid; thus Manhattan distance (i.e., the l_1 norm) and Euclidean distance (i.e., the l_2 norm) are appropriate metrics to consider. Without loss of generality, though, the identification by A* of the shortest path to every node explored—including the goal node—is guaranteed when the heuristic function is consistent [38].



(a) A graph with a consistent heuristic

(b) A graph with an inconsistent heuristic

Figure 4.3: Two graphs with heuristic values for each node in parentheses. In (a), A* follows the shortest path to the goal, since its heuristic is consistent. In (b), however, A* chooses the wrong node at the third iteration since its heuristic is inconsistent at the lower right node.

A consistent heuristic is one that satisfies the condition

$$h(x) \leq d(x, y) + h(y) \quad \forall x, y \in \mathbb{D}, \tag{4.1}$$

where $h(x)$ captures the heuristic cost of node x, $d(x, y)$ captures the branch cost between nodes x and y, and $\mathbb{D}$ captures the set of all nodes in the graph. An interpretation of equation 4.1 is that when moving from node x to node y, the estimated distance remaining to the goal cannot decrease by more than the distance actually traveled to reach node y from node x. By corollary, a consistent heuristic also satisfies the criterion for admissibility: it never over-estimates the remaining distance to the goal.

In the case that the heuristic is not admissible, A* is still guaranteed to generate a path connecting the start and the goal (provided such a path exists)—it is simply not necessarily guaranteed to be the shortest path. This is exemplified in Figure 4.3.

Of course, if one is willing to forego the shortest-path guarantee, there is nothing wrong with using a non-admissible heuristic. Further, inadmissible heuristics have the potential to generate paths to the goal faster than if an admissible heuristic were used—the Multi-

Heuristic A* variant [37] for example, uses inadmissible heuristics to generate a fast but possibly suboptimal solution, which is then revised in accordance with an admissible heuristic.

It is worth noting that additional non-heuristic costs and values can be injected to alter the behaviour of the algorithm. Pursuant to the application outlined in this thesis, for example, one could overlay potential fields around obstacles in the map to enforce avoidance or attraction while guaranteeing a path to the goal.

4.4 The Motion Primitive Library

The algorithms presented thus far guarantee us the shortest path to a goal within a static environment, provided such a path exists. There are, however, two major shortcomings that render these solutions incomplete for the purpose of this thesis. First, the intent is to create a tactical navigation system—as such, the shortest path is not necessarily the most desirable one. Instead, some degree of wall-following is preferred, for the sake of maximizing cover and minimizing visibility. Second, since the system is intended to operate in initially unmapped areas, it is desirable to include some degree of expedited path revision as the map is incrementally generated. Excepting the case in which a dead end is discovered, the path to the goal point is not expected to change significantly between consecutive iterations. Consequently, computation time can be reduced by revising the algorithm to take as argument some previous path.

The Motion Primitive Library (MPL) [33] provides an implementation that includes these final desired qualities. The full algorithm provided by the authors is much more general than needed for our purposes. We thus present only the aspects used in this thesis—a detailed explanation of the full capabilities provided by the Motion Primitive Library can be found in [32]–[34].

MPL is built atop Lifelong Planning A* (LPA*), which is a variant of A* designed with the intent of dynamic environments [36]. Algorithm 3 presents a formulation of LPA* using notation consistent with Algorithms 1 and 2 and with an undirected graph. The principal idea behind LPA* is that any update to the map that would render the current trajectory infeasible at a particular node would also create a local heuristic inconsistency among the neighboring nodes of that point. To detect local inconsistency, LPA* introduces the *rhs* cost, which is defined for each *node* as

$$node.rhs = \begin{cases} 0\,,\ node = start \\ \min\limits_{n \in node.branchList} (n.startCost + weight)\,,\ \text{otherwise} \end{cases} . \tag{4.2}$$

If the *rhs* cost of a node differs from its *startCost*, then the node is locally inconsistent. Note that unlike A*, LPA* does not have a list of unseen nodes from which the frontier is expanded—this is because nodes may need to be reevaluated to address inconsistencies after a map update. Instead, every node except for the start is initialized such that its *startCost* is infinite. In the formulation provided, it is assumed that during the update of the map, an *updatedNodeSet* will be populated with every node for which its *branchList* changed in some way.

Additionally, MPL provides parametric object avoidance capabilities via path revision in accordance with potential fields within a search corridor. The user is free to specify the the dimensions of these potential fields radially in the xy-plane as R_{xy} and vertically along the z-axis as R_z. The user also provides a weight coefficient to set the maximum strength U_{weight} of the potential fields relative to the travel cost and heuristic cost. The strength of the fields are then calculated for each node within their field using the following formula:

$$U(node) = \max\left(U_{weight}\left(1 - \frac{\sqrt{node.x^2 + node.y^2}}{R_{xy}}\right)\left(1 - \frac{|\,node.z\,|}{R_z}\right)\ ,\ 0\right). \tag{4.3}$$

Algorithm 3 Lifelong Planning A* Algorithm [36]

1: $start.rhs := 0$
2: insert $start$ into $frontier$
3: $minSet :=$ findMinCost($frontier$)
4: $selectedNode := \underset{node \in minSet}{\arg\min}\ (\min(node.startCost, node.rhs))$
5: **while** $selectedNode.startCost + heurisiticEstimate(selectedNode) < goal.startCost$ **or** $goal.startCost \neq goal.rhs$ **do**
6: remove $selectedNode$ from $frontier$
7: **for all** $(neighbor, weight) \in selectedNode.branchList$ **do**
8: **if** $neighbor \neq start$ **then**
9: $neighbor.rhs := \min_{(n,weight) \in neighbor.branchList}(n.startCost + weight)$
10: **if** $neighbor \in frontier$ **then**
11: remove $neighbor$ from $frontier$
12: **if** $neighbor.startCost \neq neighbor.rhs$ **then**
13: insert $neighbor$ into $frontier$
14: **if** $selectedNode.startCost > selectedNode.rhs$ **then**
15: $selectedNode.startCost := selectedNode.rhs$
16: **else**
17: $selectedNode.startCost := \infty$
18: **if** $selectedNode \neq start$ **then**
19: $selectedNode.rhs := \min_{(n,weight) \in selectedNode.branchList}(n.startCost + weight)$
20: **if** $selectedNode.startCost \neq selectedNode.rhs$ **then**
21: insert $selectedNode$ into $frontier$
22: **wait until** map has changed
23: **for all** $node \in updatedNodeList$ **do**
24: **if** $node \neq start$ **then**
25: $node.rhs = \min_{(neighbor,weight) \in node.branchList}(neighbor.cost + weight)$
26: **if** $node \in frontier$ **then**
27: remove $node$ from $frontier$
28: **if** $node.startCost \neq node.rhs$ **then**
29: insert $node$ into $frontier$
30: **goto** 3

Similarly, the user provides radii in each dimension for the search corridor, which is generated around a baseline path provided by LPA*. LPA* is then reiterated, with the following two modifications: the explorable space is limited to the search corridor instead of the entire map, and heuristic values are replaced with values based on the strength of the potential field of the closest obstacle for each node.

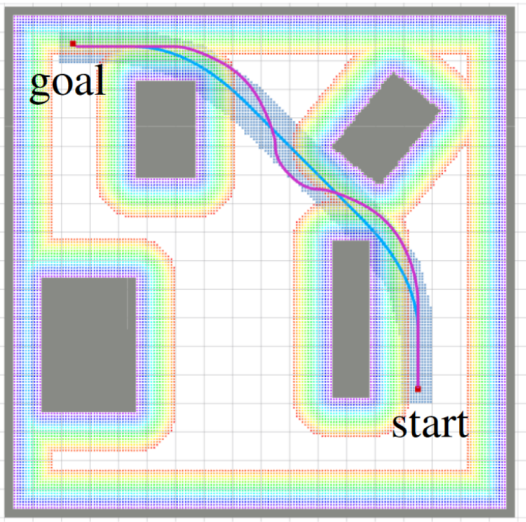


Figure 4.4: After LPA* generates a baseline trajectory (blue), a revised path (red) is generated within a search corridor (blue field) subject to the repulsive potential fields (rainbow fields) around occupied spaces in the environment [33].

Figure 4.4 above provides a visualization of path revision for obstacle avoidance. In our implementation, we use a negative potential field weight to enforce attractive potential fields, rather than repulsive ones. In doing so, we achieve our desired wall-following behavior for tactical navigation.

4.5 Remarks on the Role of FMPC

In light of the capabilities provided by MPL—path generation, wall following, and obstacle avoidance—one might think that the use of FMPC is redundant and unnecessary. Indeed, it would be possible to develop a full solution incorporating only the motion primi-

tive library. The addition of FMPC, however, shores up weaknesses of MPL and provides a better foundation for future developments at a greatly discounted computational cost.

MPL, being built upon a search-based algorithm, necessitates a discrete search space. As such, the environmental map is discretized into uniform cubic spaces called *voxels*. The precision of the map is thus bounded by the size of the voxels used to represent it. With a volumetric map, halving the side length of each voxel will increase the number of voxels eightfold; hence, MPL scales poorly with increased map resolution. In our implementation, adjacent nodes in the search space are placed 0.25m apart. FMPC, on the other hand, operates in a continuous space. Its computational complexity is independent of the map resolution, so it may be run with arbitrary precision. Thus, by using FMPC to refine a coarse path from MPL, one can preserve the beneficial aspects of the latter without compromising precision of the end solution.

Similarly, search-based algorithms scale poorly with dimensionality. Though a typical quadrotor state vector has twelve elements, MPL as implemented in this thesis only considers the three elements corresponding to spatial position. Even the full motion primitive library, which can consider spatial derivatives, does not consider the orientation of the system in generating a trajectory—FMPC, however, generates a full-state trajectory for a linear system across the entire time horizon.

Within this thesis, the trajectory generated by the guidance algorithm is used as a reference signal to a control algorithm. The rationale for this is that FMPC assumes a linear system, but the dynamics of a quadrotor are nonlinear—the system is typically linearized about a hovering state, but this model serves poorly for aggressive maneuvering. As discussed in the preceding chapter, however, many parameters of FMPC need not be constant. One could thus extend FMPC to nonlinear systems—one potential method for doing so would be to linearize the system about the state at every step along the time horizon. Alternatively,

a lookup table of linearizations about multiple states could be implemented, with residual nonlinear effects treated as system disturbances. Further, the ability to vary FMPC parameters promotes compatibility with adaptive control laws, whereas MPL would mandate rigidity. Were such a variant of the guidance algorithm to be developed, one could directly apply the control outputs from FMPC rather than pass the state outputs to an additional controller. Thus, by implementing both FMPC and MPL, it becomes possible to plan optimal full-state trajectories for aggressive maneuvering, which would be highly desirable for tactical navigation.

Finally, the increase in computation time introduced by the addition of FMPC is mitigated via parallel programming. The initial trajectory generated by MPL is constantly revised as the position of the system changes and new areas of the map are discovered. Since MPL is independent of the output of FMPC, this revision process can be restarted immediately after the previous iteration of MPL completes. FMPC and MPL are thus executed simultaneously in separate threads, with the FMPC thread polling the MPL thread at every iteration to get the latest MPL trajectory. The update frequency for the guidance algorithm, then, is determined by the speed of FMPC rather than the speed of both FMPC and MPL.

Chapter 5: Implementation, Application, and Results

5.1 Equations of Motion for a Quadrotor Vehicle

The guidance algorithm developed over the course of this thesis is to be implemented and tested on a quadrotor aerial vehicle. Quadrotors are capable of vertical takeoff and landing, can hover in place, and are more maneuverable in general than fixed-wing aircraft [43], which makes them well-suited for autonomous missions in obstacle-laden environments. Recall that MPC is predicated on having a model of the system for which it is to generate control input—or in this case, a state trajectory. We are thus motivated to develop a model of the quadrotor, beginning with the derivation of equations of motion.

Quadrotors are actuated by four coplanar propellers that each apply a thrust force to the system [42]. Typically, these propellers are driven by electric motors, which are in turn controlled digitally via pulse width modulation. We can thus apply an electromagnetic force to the propeller over a continuous interval up to the point of saturation. Since the propellers of micro aerial vehicles tend to be small and lightweight, they can be accelerated very quickly relative to the dynamics of the vehicle as a whole. As such, we can assume with negligible error that we have direct control over the angular velocity of a propeller. Though each propeller is a separate control input to the system, it is more practical to map control inputs to the propeller combinations described in (5.1) [39].

$$\begin{bmatrix} u_1(t) \\ u_2(t) \\ u_3(t) \\ u_4(t) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & -l_2 & 0 & l_4 \\ l_1 & 0 & -l_3 & 0 \\ c_T & -c_T & c_T & -c_T \end{bmatrix} \begin{bmatrix} T_1(t) \\ T_2(t) \\ T_3(t) \\ T_4(t) \end{bmatrix}, \; t \geq t_0, \tag{5.1}$$

where l_i is the distance from the center of mass C to the source of T_i, and c_T is the drag

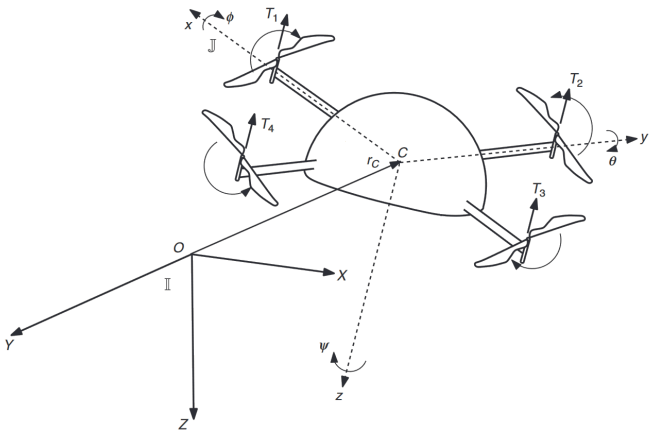


Figure 5.1: A quadrotor has six degrees of freedom: three spatial coordinates x, y, z, and three orientation angles roll ϕ, pitch θ, and yaw ψ. It is actuated by four propellers, which produce thrusts T_1, T_2, T_3, and T_4. r_c denotes the position of the center of mass in the global frame $\mathbb{I}$. Modified from [39].

coefficient corresponding to the propellers, which are assumed to be identical. In our case, each arm of the quadrotor is equally long, i.e. $l_1 = l_2 = l_3 = l_4 = l$. With this mapping, u_1 is the force produced by the propellers along the z-axis. Similarly, u_2, u_3, and u_4 are the moments of force produced by the propellers about the x-, y-, and z-axes.

Since the goal and obstacle positions are constant in the global frame $\mathbb{I} = \{O; X, Y, Z\}$, we choose to track the position of the system in the global frame as well. The state vector $\chi(t)$, then, will consist of a center-of-mass position vector $r_c(t)$, an orientation vector $\gamma(t)$, a center-of-mass velocity vector $v_c(t)$, and an angular velocity vector $\omega(t)$ as shown in (5.2) and (5.3).

$$\chi(t) = \begin{bmatrix} r_c(t) \\ \gamma(t) \\ v_c(t) \\ \omega(t) \end{bmatrix}, \tag{5.2}$$

$$r_c(t) \triangleq \begin{bmatrix} X_c(t) \\ Y_c(t) \\ Z_c(t) \end{bmatrix}, \ \gamma(t) \triangleq \begin{bmatrix} \phi(t) \\ \theta(t) \\ \psi(t) \end{bmatrix}, \ v_c(t) \triangleq \begin{bmatrix} u(t) \\ v(t) \\ w(t) \end{bmatrix}, \ \omega(t) \triangleq \begin{bmatrix} p(t) \\ q(t) \\ r(t) \end{bmatrix}, \ t \geq t_0. \tag{5.3}$$

with $\phi(t) \in [0, 2\pi)$, $\theta(t) \in (-\frac{\pi}{2}, \frac{\pi}{2})$, $\psi(t) \in [0, 2\pi)$.

Each vector in (5.3) has a corresponding equation of motion described in [39]. The

translational kinematic equation is

$$\dot{r}_c = R_{321}^{-1}(\phi(t), \theta(t), \psi(t))\, v_c(t) = R_{321}^{T}(\phi(t), \theta(t), \psi(t))\, v_c(t),\ t \geq t_0, \tag{5.4}$$

where $R_{321}(\cdot, \cdot, \cdot)$ is a 3-2-1 rotation matrix [40] from the global frame to the body frame $\mathbb{J} = \{C; x(t), y(t), z(t)\}$ given by

$$\begin{aligned} &R_{321}(\phi(t), \theta(t), \psi(t)) \\ &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi(t) & \sin\phi(t) \\ 0 & -\sin\phi(t) & \cos\phi(t) \end{bmatrix} \begin{bmatrix} \cos\theta(t) & 0 & -\sin\theta(t) \\ 0 & 1 & 0 \\ \sin\theta(t) & 0 & \cos\theta(t) \end{bmatrix} \begin{bmatrix} \cos\psi(t) & \sin\psi(t) & 0 \\ -\sin\psi(t) & \cos\psi(t) & 0 \\ 0 & 0 & 1 \end{bmatrix}. \end{aligned} \tag{5.5}$$

Note that since $R_{321}(\cdot, \cdot, \cdot)$ is a rotation matrix, it is orthonormal—thus, it is guaranteed that $R_{321}^{-1}(\cdot, \cdot, \cdot) = R_{321}^{T}(\cdot, \cdot, \cdot)$

The rotational kinematic equation follows from the relationship between the relative angular velocity between frames and the Tait-Bryan angles. From [40],

$$\begin{aligned} \omega(t) = \begin{bmatrix} 1 & 0 & -\sin\theta(t) \\ 0 & \cos\phi(t) & \cos\theta(t)\sin\phi(t) \\ 0 & -\sin\phi(t) & \cos\theta(t)\cos\phi(t) \end{bmatrix} \begin{bmatrix} \dot{\phi}(t) \\ \dot{\theta}(t) \\ \dot{\psi}(t) \end{bmatrix} = \begin{bmatrix} 1 & 0 & -\sin\theta(t) \\ 0 & \cos\phi(t) & \cos\theta(t)\sin\phi(t) \\ 0 & -\sin\phi(t) & \cos\theta(t)\cos\phi(t) \end{bmatrix} \dot{\gamma}(t), \\ t \geq t_0. \end{aligned} \tag{5.6}$$

Through inversion, we obtain

$$\dot{\gamma}(t) = \begin{bmatrix} 1 & \sin\phi(t)\tan\theta(t) & \cos\phi(t)\tan\theta(t) \\ 0 & \cos\phi(t) & -\sin\phi(t) \\ 0 & \sin\phi(t)\sec\theta(t) & \cos\phi(t)\sec\theta(t) \end{bmatrix} \omega(t),\ t \geq t_0. \tag{5.7}$$

The translational dynamic equation found in [39] can be simplified with constant mass to yield

$$mg \begin{bmatrix} -\sin\theta(t) \\ \cos\theta(t)\sin\phi(t) \\ \cos\theta(t)\cos\phi(t) \end{bmatrix} - u_1(t) \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} + F(\chi(t), u(t)) = m(\dot{v}_c(t) + \omega(t)^{\times} v_c(t)),\ t \geq t_0, \tag{5.8}$$

where g is the magnitude of gravitational acceleration, m is the mass of the vehicle, assumed to be constant, and $F(\cdot,\cdot) \in \mathbb{R}^3$ captures the aerodynamic forces acting on the system.

$$\dot{v}_c(t) = g \begin{bmatrix} -\sin\theta(t) \\ \cos\theta(t)\sin\phi(t) \\ \cos\theta(t)\cos\phi(t) \end{bmatrix} + \omega(t)^{\times} v_c(t) - \begin{bmatrix} 0 \\ 0 \\ u_1(t)/m \end{bmatrix} + \frac{1}{m} F(\chi(t), u(t)),\ t \geq t_0. \tag{5.9}$$

Similarly, the rotational dynamic equation found in [39] can be simplified to yield

$$\begin{bmatrix} u_2(t) \\ u_3(t) \\ u_4(t) \end{bmatrix} + M(\chi(t), u(t)) = I\dot{\omega}(t) + \omega(t)^{\times} I\omega(t) + I_P \sum_{i=1}^{4} \begin{bmatrix} 0 \\ 0 \\ \dot{\Omega}_{P,i}(t) \end{bmatrix} + \omega(t)^{\times} I_P \sum_{i=1}^{4} \begin{bmatrix} 0 \\ 0 \\ \Omega_{P,i}(t) \end{bmatrix},$$

$$t \geq t_0, \tag{5.10}$$

where $M(\cdot,\cdot) \in \mathbb{R}^3$ captures the aerodynamic moments acting on the system, $I \in \mathbb{R}^{3\times 3}$ denotes the diagonal inertia matrix of the vehicle, I_P denotes the diagonal inertia matrix of a propeller, and $\Omega_{P,i}$ denotes the angular velocity of the ith propeller. Solving for $\dot{\omega}(t)$ yields

$$\begin{aligned} \dot{\omega} = I^{-1} \Bigg(& \begin{bmatrix} u_2(t) \\ u_3(t) \\ u_4(t) \end{bmatrix} + M(\chi(t), u(t)) - \omega(t)^{\times} I\omega(t) \\ & - I_P \sum_{i=1}^{4} \begin{bmatrix} 0 \\ 0 \\ \dot{\Omega}_{P,i}(t) \end{bmatrix} - \omega(t)^{\times} I_P \sum_{i=1}^{4} \begin{bmatrix} 0 \\ 0 \\ \Omega_{P,i}(t) \end{bmatrix} \Bigg),\ t \geq t_0. \end{aligned} \tag{5.11}$$

The equations of motion, then, are captured in (5.4, 5.7, 5.9, 5.11).

5.2 The Quadrotor as a Linear System in Discrete Time

Recall that the algorithm for MPC presented in Chapter 3 requires a linear system in discrete time. The equations of motion produced in the preceding section are nonlinear; however, a usable system model can be obtained through linearization about an equilibrium point.

A quadrotor is at equilibrium when it hovers, which it can do at any spatial point with any yaw bearing. The equilibrium state χ_{eq} is thus given by

$$\chi_{eq} = \begin{bmatrix} r_{eq}^T & \gamma_{eq}^T & v_{eq}^T & \omega_{eq}^T \end{bmatrix}^T = \begin{bmatrix} x_{eq} & y_{eq} & z_{eq} & 0 & 0 & \psi_{eq} & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T. \tag{5.12}$$

The aerodynamic forces and moments acting on a quadrotor at equilibrium are zero. Additionally, the angular velocities of the propellers will be constant and equal in magnitude; thus, the sums of the propellers' angular velocities and angular accelerations are both zero. The translational and rotational kinematic equations are linearly dependent on v and ω respectively—both of which are zero at equilibrium. Thus, the partial derivatives of these equations will be zero for other state elements. The elements that are not necessarily zero, then, are given by

$$\begin{aligned} \frac{\partial \dot{r}_c(t)}{\partial v_c(t)}\Big|_{\chi(t)=\chi_{eq}} &= \frac{\partial}{\partial v_c(t)} \left(R_{321}^T\big(\phi(t), \theta(t), \psi(t)\big) v_c(t) \right) \Big|_{\chi(t)=\chi_{eq}} \\ &= R_{321}^T(0, 0, \psi_{eq}) = \begin{bmatrix} \cos\psi_{eq} & -\sin\psi_{eq} & 0 \\ \sin\psi_{eq} & \cos\psi_{eq} & 0 \\ 0 & 0 & 1 \end{bmatrix}, \end{aligned} \tag{5.13}$$

$$\begin{aligned} \frac{\partial \dot{\gamma}(t)}{\partial \omega(t)}\Big|_{\chi(t)=\chi_{eq}} &= \frac{\partial}{\partial \omega(t)} \left(\begin{bmatrix} 1 & \sin\phi(t)\tan\theta(t) & \cos\phi(t)\tan\theta(t) \\ 0 & \cos\phi(t) & -\sin\phi(t) \\ 0 & \sin\phi(t)\sec\theta(t) & \cos\phi(t)\sec\theta(t) \end{bmatrix} \omega(t) \right) \Bigg|_{\chi(t)=\chi_{eq}} \\ &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \end{aligned} \tag{5.14}$$

for $t \geq t_0$.

The nonzero state-dependent components of the linearized translational dynamic equation are given by

$$\frac{\partial \dot{v}_c(t)}{\partial \gamma(t)}\Big|_{\chi(t)=\chi_{eq}} = g \frac{\partial}{\partial \gamma(t)} \begin{bmatrix} -\sin\theta(t) \\ \cos\theta(t)\sin\phi(t) \\ \cos\theta(t)\cos\phi(t) \end{bmatrix} \Bigg|_{\chi(t)=\chi_{eq}} = \begin{bmatrix} 0 & -g & 0 \\ g & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \; t \geq t_0, \tag{5.15}$$

and the rotational dynamic equation has no nonzero state-dependent elements, i.e.

$$\frac{\partial \dot{\omega}(t)}{\partial \chi(t)}\Big|_{\chi(t)=\chi_{eq}} = \begin{bmatrix} 0 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 0 \end{bmatrix}, \; t \geq t_0. \tag{5.16}$$

The linearized system is thus given by

$$\dot{\chi}(t) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & \cos\psi_{eq} & -\sin\psi_{eq} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \sin\psi_{eq} & \cos\psi_{eq} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & -g & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & g & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \chi(t) + \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -\frac{1}{m} & 0 & 0 & 0 \\ 0 & \frac{1}{I_x} & 0 & 0 \\ 0 & 0 & \frac{1}{I_y} & 0 \\ 0 & 0 & 0 & \frac{1}{I_z} \end{bmatrix} u(t),$$

$t \geq t_0.$ (5.17)

With 12 state elements and 4 control inputs, it is worthwhile to consider further simplifications to reduce the dimensionality of the system. Specifically, if no yaw control in applied spatial coordinate is still reachable. Thus, we impose that $\psi_{eq} = 0$, relying on our underlying controller to enforce this, and remove $\psi(t)$, $\dot{\psi}(t)$, and $u_4(t)$ from our model. Since the time complexity of MPC is $O(T(n+m)^3)$ [4], this simplification can reduce computation time by up to 46%. The zero-yaw linear model in continuous time is given by

$$\begin{bmatrix} \dot{X}_c(t) \\ \dot{Y}_c(t) \\ \dot{Z}_c(t) \\ \dot{\phi}(t) \\ \dot{\theta}(t) \\ \dot{u}(t) \\ \dot{v}(t) \\ \dot{w}(t) \\ \dot{p}(t) \\ \dot{q}(t) \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & -g & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & g & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} X_c(t) \\ Y_x(t) \\ Z_c(t) \\ \phi(t) \\ \theta(t) \\ u(t) \\ v(t) \\ w(t) \\ p(t) \\ q(t) \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ -\frac{1}{m} & 0 & 0 \\ 0 & \frac{1}{I_x} & 0 \\ 0 & 0 & \frac{1}{I_y} \end{bmatrix} \begin{bmatrix} u_1(t) \\ u_2(t) \\ u_3(t) \end{bmatrix}, \ t \geq t_0. \tag{5.18}$$

The system must now be discretized; that is, we must calculate A_d and B_d such that

$$\chi_{k+1} = A_d \chi_k + B_d u_k, \tag{5.19}$$

where k denotes the kth time step, each of which are Δt apart in time. The relation between A, A_d, B, and B_d is thus:

$$A_d = e^{A\Delta t} = \sum_{n=0}^{\infty} \frac{1}{n!}(A\Delta t)^n \quad , \quad B_d = \int_0^{\Delta t} e^{A\tau} B \, d\tau. \tag{5.20}$$

Noting that $A^n = 0$ for $n > 2$, we obtain

$$A_d = e^{A\Delta t} = I + A\Delta t + \frac{1}{2}(A\Delta t)^2 = \begin{bmatrix} 1 & 0 & 0 & 0 & -\frac{g\Delta t^2}{2} & \Delta t & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & \frac{g\Delta t^2}{2} & 0 & 0 & \Delta t & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & \Delta t & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & \Delta t & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & \Delta t \\ 0 & 0 & 0 & 0 & -g\Delta t & 1 & 0 & 0 & 0 & -\frac{g\Delta t^2}{2} \\ 0 & 0 & 0 & g\Delta t & 0 & 0 & 1 & 0 & \frac{g\Delta t^2}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \tag{5.21}$$

from which it follows that

$$B_d = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ -\frac{\Delta t^2}{2m} & 0 & 0 \\ 0 & \frac{\Delta t^2}{2I_x} & 0 \\ 0 & 0 & \frac{\Delta t^2}{2I_y} \\ 0 & 0 & -\frac{g\Delta t^3}{6I_y} \\ 0 & \frac{g\Delta t^3}{6I_x} & 0 \\ -\frac{1}{m} & 0 & 0 \\ 0 & \frac{1}{I_x} & 0 \\ 0 & 0 & \frac{1}{I_y} \end{bmatrix}. \tag{5.22}$$

5.3 Implementation of MPC and MPL

To properly frame the discussion on the structure of our guidance algorithm, it is worthwhile to first discuss the platform on which it is to be implemented. In the development of optimized code, it is critical to consider such factors as processor architecture, instruction set, and operating system. Our intended platform is the ODROID-XU4 running the Ubuntu 16.04 operating system. The XU4 uses the Exynos5422 CPU, which is an eight-core processor implementing the big.LITTLE architecture—that is, it has four Cortex-A15 cores and four Cortex-A7 cores [46]. The former operate at 2.1 GHz, while the latter clock in at 1.4 GHz. The processor cores are grouped into two clusters with independent L2 caches, with the A15 cluster having a 2MB cache and the A7 cluster having a 512 KB one. Both types of core use a 32-bit ARM instruction set architecture. Lastly, for compatibility with the previous flight stack, we choose to implement the algorithm in the C++ programming language.

Operating system, instruction set architecture, and programming language are the primary factors in determining which software libraries are available for use. Notably, we cannot

use the Intel Math Kernel Library—a highly optimized linear algebra library—since it is developed for Intel processors which use the IA-32 or x86-64 architectures [44]. Instead, we use the Eigen3 linear algebra library [45]. Ubuntu, being a *nix operating system, allows us to make use of the POSIX thread (pthread) library, which simplifies algorithm parallelization within a single process. Since the pthread library is not native to Windows operating systems, its inclusion comes at the expense of cross-platform portability. This is an acceptable cost, since most single board computers run *nix operating systems.

Our CPU architecture—having one fast quad core cluster with a large L2 cache and one slow quad core cluster with a small L2 cache—encourages us to use up to four major threads for our algorithm. A fifth thread would either be executed on the slower cluster—in which case the segregated L2 caches would introduce additional latency in cross-cluster data transmission—or on the faster cluster, in which case it competes for CPU time with an existing thread. The specific scenario is determined by the operating system, which schedules CPU use, rather than the programmer. In either case, we are free to implement more threads, provided we accept that the excessive threads will not perform as well as the first four.

Our initial plan for implementation introduced two threads: one for local planning via MPC and one for global planning via MPL. With compiler flags set to maximize the speed of the algorithm, however, our program became too large for our system to successfully compile. We thus come to our realized implementation: the flight stack process producing one thread for local trajectory planning and one thread for socket-based interprocess communication, along with a second process running a single thread for global trajectory planning. Though instigated by necessity, executing MPL as a separate process offers the benefit of modularity: the global path planner can be swapped out for any other, provided the new planner properly implements communication with the main process. In fact, sockets since are language-

agnostic, the server and client need not be written in the same programming language. Thus, if we so desired, the current global planner could be swapped for one written in Python, another popular programming language in the field of robotics.

The base flight stack uses the output of our algorithm as a reference signal for generating control inputs. Conversely, our algorithm needs only needs a state estimate as input from the flight stack. The relatively low degree of cross-thread communication means that the new trajectory generation algorithm can be implemented without significant alterations to the base flight stack. In fact, the base flight stack already includes a data structure—the *Test_Controller* object—to be shared between its own threads. We share this structure with the two additional threads for this process as a channel for passing data between the three. In particular, instance variables are added for the output trajectories of MPC and MPL, the initial position for MPL, the port number with which to communicate with the MPL process, and a number of mutexes for thread safety and process control. Mutexes are objects shared between threads that can be locked and unlocked—when a thread tries to lock an already-locked mutex, it pauses until that mutex is available. Mutexes can thus be used to prevent one thread from reading a variable while another writes to it, or vice versa. Alternatively, mutexes can be used to ensure that one thread does not try to read variables that have not yet been initialized by another thread.

The base flight stack is written such that both new threads can query the vehicle state directly. The local trajectory thread does so to use the vehicle state estimate as input to the constraint generation subroutine and to MPC itself. However, initial experimentation revealed that significant vehicle drift occurred when the current position was used as input to the global trajectory process. As such, we pass as input to MPL the current goal point of MPC rather than the current position—this completely eliminated the previously observed drift by improving consistency in MPL-generated waypoints from one iteration to the next.

As mentioned above, the base flight stack is already multi-threaded. The point at which it launches its child thread serves as a logical point from which to launch both of our own. Immediately beforehand, however, two mutexes are locked for the sake of process control. More specifically, the initialization of the interprocess communication thread requires a socket number to be read in by the local trajectory thread, and MPC itself cannot be run until the global trajectory process has generated its first output. After the two new threads have been launched, the base flight stack algorithm proceeds with its normal operations, using the results of the MPC algorithm as a reference signal. The particulars of the base flight stack are beyond the scope of this thesis; however, the major modifications are captured in Algorithm 4.

Algorithm 4 Modified Flight Stack Thread

1: Initialize Test_Controller *test_controller*
2: Perform miscellaneous base flight stack initializations
3: Lock *LocalControlMutex*
4: Lock *CommsControlMutex*
5: Launch *Local Planner* thread with access to *test_controller*
6: Launch *Interprocess Communication* thread with access to *test_controller*
7: **loop**
8: Execute base flight stack algorithm

Interprocess communication is achieved through the use of a TCP (Transmission Control Protocol) socket. Sockets are formed by a client-server pair, which pass data back and forth by targeting the other's IP (Internet Protocol) address over a specific port number. Coincidentally, sockets form the backbone of the internet by allowing different machines to exchange data over a network. In our case, however, both ends of the socket are on the same machine: we communicate over *localhost*, i.e. the IP address 127.0.0.1. The port number is represented with 16 bits; thus, for any given IP address, there are 65,536 possible ports. It is therefore unlikely that two different applications would attempt to establish a socket over the same port. If this were to happen, however, the later application would fail to form its

connection. This case necessitates that the port number be adjustable, either automatically or manually, to avoid the possibility of conflict with an external application. We resolve this by allowing the user to manually select the port number via input file. This input is read by the Local Planner thread and passed to the Interprocess Communication thread, which then proceeds as described by Algorithm 5.

Algorithm 5 Interprocess Communication Thread

1: *firstPassComplete* := false
2: **wait until** *CommsControlMutex* is unlocked
3: Initialize Socket *CommSock*
4: Connect via *CommSock* to *localhost* over port *test_controller.portNumber*
5: Send *test_controller.missionGoal* to *Global Planner* via *CommSock*
6: **loop**
7: Send *test_controller.localGoal* to *Global Planner* via *CommSock*
8: **wait until** *numWaypoints* is received from *Global Planner* via *CommSock*
9: Lock *GlobalOutputMutex*
10: Resize *test_controller.waypointArray* to *numWaypoints*
11: **for** $i \in \{1, \ldots, numWaypoints\}$ **do**
12: Receive *nextWaypoint* from *Global Planner* via *CommSock*
13: *test_controller.waypointArray*$[i]$:= *nextWaypoint*
14: *test_controller.numWaypoints* := *numWaypoints*
15: Unlock *GlobalOutputMutex*
16: **if not** *firstPassComplete* **then**
17: *firstPassComplete* := true
18: Unlock *CommsControlMutex*

The Interprocess Communication thread is a relatively lightweight subroutine that could be run without difficulty by a minor processor. Most of its time is spent idling at the step represented by line 8 of Algorithm 5 while the Global Planner process computes a new output. Once this new output is received, it is quickly copied over to the local planner, after which the program returns to idling at line 8. Assuming availability of mutexes, the loop of the Interprocess Communication thread runs on the order of microseconds, whereas those of the Local Planner and Global Planner tend to run on the order of milliseconds.

The Global Planner process implements the MPL algorithm described in Chapter 4 and

provided by [34] To adapt the process to our purposes, four major modifications were made to the original code. First, file-reading capabilities were implemented so the user could adjust LPA* parameters without changing hard-coded values and recompiling the code. Second, a socket was added to exchange data with the Interprocess Communication thread. Third, the goal and start positions for LPA* are received over this socket in correspondence with lines 5 and 7 from Algorithm 5. Fourth, the output of LPA*—a set of waypoints guiding the system from the start to the goal—is sent back via the socket to the Interprocess Communication thread in correspondence with lines 8 and 11–13 of Algorithm 5.

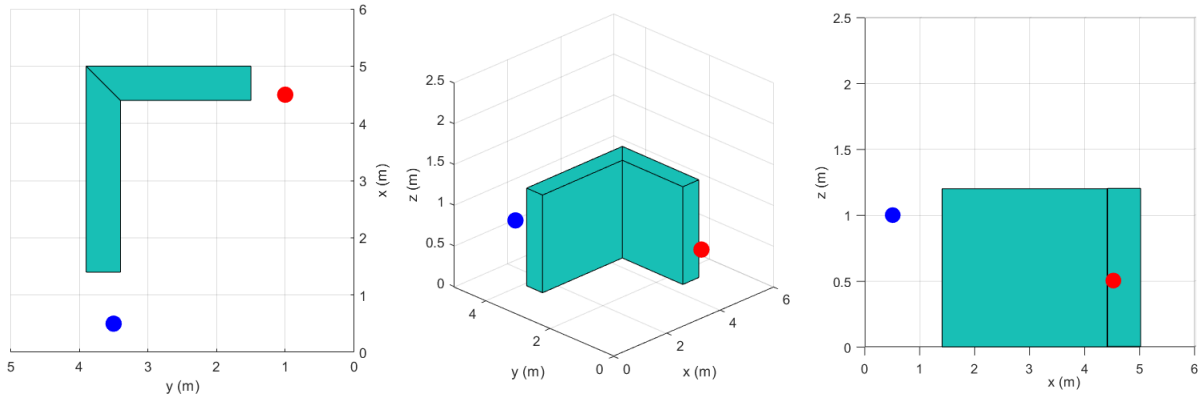
Once the Interprocess Communication thread has received and copied the first output from the Global Planner, the *LocalControlMutex* is released and the Local Planner thread is allowed to proceed beyond its initialization phase. The Local Planner algorithm implements both the SDPA algorithm [25] for constraint generation and the MPC algorithm for local trajectory generation, along with mutexes for read/write protection. When the system reaches a position sufficiently close to the mission goal, it stops executing MPC and instead attempts to maintain that position. If, for some reason, the system subsequently drifts away from the goal, trajectory generation is resumed in order to return. The final step of the loop is to advance the state and control horizons to warm-start the next iteration of MPC, as described by equation 3.52. The *getCurrentPosition*() function allows for convenient execution of simulated missions as well as real flights. In the first case, it will return the state that follows from applying the current control input to the previous state in accordance with the linear model of the system—note that this is not necessarily the same value as the subsequent state in the time horizon, since the runtime-optimized MPC algorithm no longer guarantees primal feasibility. In the second case, the function simply returns the current state estimate provided by the base flight stack algorithm. The Local Planner thread is described in Algorithm 6.

Algorithm 6 Local Planner Thread

Parse input files *Mission_params.txt*, *System_params.txt*, and *MPC_params.txt*
Set *test_controller.portNumber* based on *Mission_params.txt*
Set *test_controller.missionGoal* based on *Mission_params.txt*
test_controller.localGoal := getCurrentPosition()
updateHoverPosition := true
Initialize miscellaneous problem data
Unlock *CommsControlMutex*
wait until *LocalControlMutex* is unlocked
loop
    Lock *PositionMutex*
    currentPosition := getCurrentPosition()
    Unlock *PositionMutex*
    Lock *GlobalOutputMutex*
    if *test_controller.waypointArray* has been updated **then**
        waypointIterator := 0
    else if *test_controller.local_goal* has been reached
        waypointIterator := min(*waypointIterator*+1 , *test_controller.numWaypoints*)
    test_controller.local_goal := *test_controller.waypointArray*[*waypointIterator*]
    Set MPC cost parameters based on *test_controller.local_goal*
    Unlock *GlobalOutputMutex*
    Lock *ConstraintMutex*
    Run SDPA
    Set MPC constraint parameters based on SDPA output
    Unlock *ConstraintMutex*
    if not *test_controller.local_goal* has been reached **then**
        Run MPC
        Set *stateHorizonArray* based on MPC output
        Set *controlHorizonArray* based on MPC output
        updateHoverPosition := true
    else
        if *updateHoverPosition* **then**
            for *state* $\in$ *stateHorizonArray* **do**
                state := *currentPosition*
            updateHoverPosition := false
        stateHorizonArray[*timeHorizon*] = *stateHorizonArray*[*timeHorizon*-1]
    Lock *LocalOutputMutex*
    test_controller.setTrajectory(stateHorizonArray)
    Unlock *LocalOutputMutex*
    for $i \in \{1,\ldots,$*timeHorizon*$-1\}$ **do**
        stateHorizonArray[i] := *stateHorizonArray*[$i+1$]
        controlHorizonArray[i] := *controlHorizonArray*[$i+1$]

5.4 Experimentation and Results

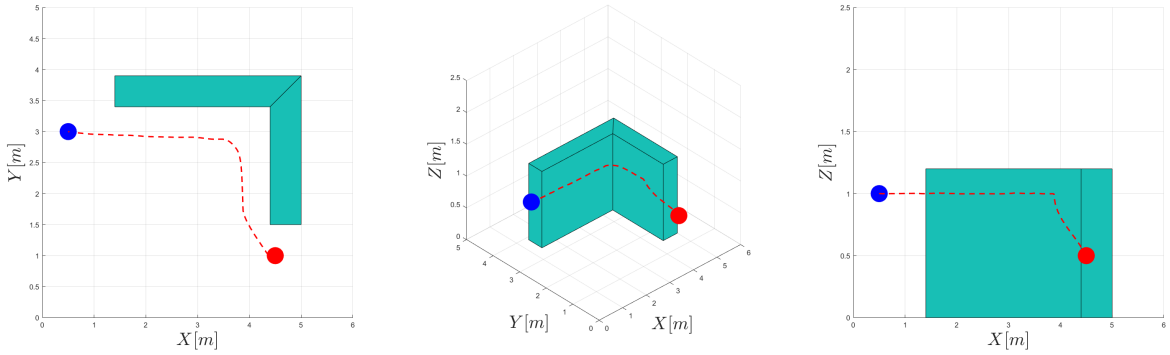
The full guidance algorithm was tested using a map containing an 'L'-shaped obstacle as depicted in Figure 5.2. The mission given to the system was to travel from the point $(0.5, 3, 1)$ to the point $(1, 4.5, 0.5)$. A direct path between the two points could leave the system exposed—the desired behavior in a contested environment would be to use the obstacle as cover by flying along its inside walls en route to the goal.



(a) Top-down view (b) Isometric view (c) Profile view

Figure 5.2: The mission and map on which the algorithm was tested. The starting point is blue, and the goal is red.

The mission was first carried out in simulation—at each time step, the next state of the system was computed by applying the first control input of the MPC time horizon to the current state. The trajectory generated by simulation is plotted in Figure 5.3. The components of the trajectory generated by the guidance algorithm are plotted in Figure 5.4. The simulation successfully demonstrates the desired behavior of the guidance algorithm: the trajectory follows along the interior of the obstacle, rather than move directly toward the goal.



(a) Top-down view (b) Isometric view (c) Profile view

Figure 5.3: In simulation, the algorithm successfully generated a wall-hugging trajectory to the mission goal. The system was specified to be 0.4m wide and 0.45m long.

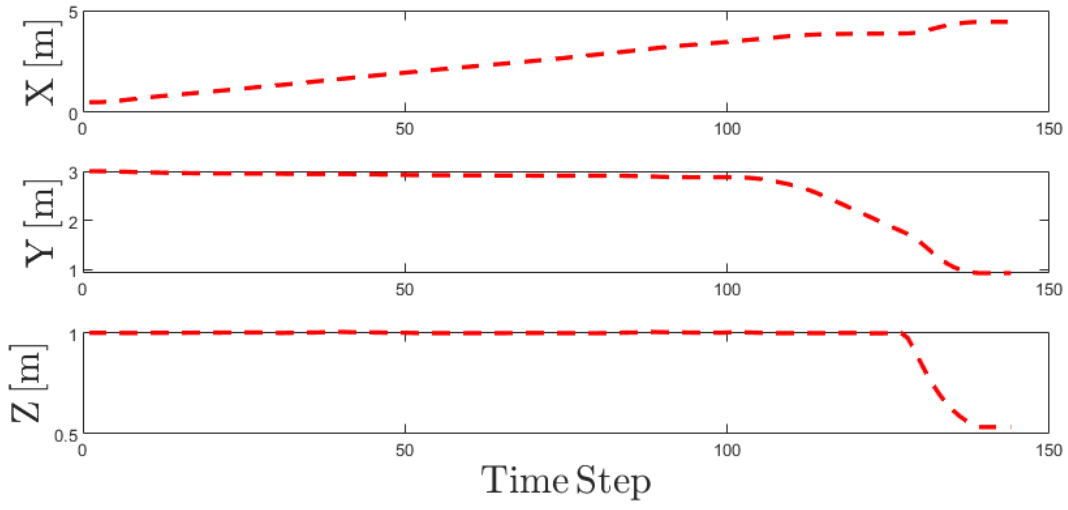
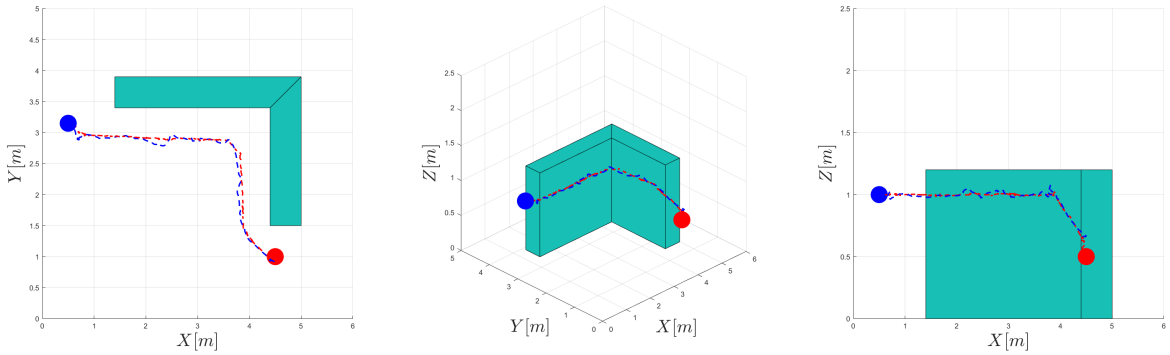


Figure 5.4: The x, y, and z components of the reference signal generated by the guidance algorithm.



(a) Top-down view (b) Isometric view (c) Profile view

Figure 5.5: In flight, the path of the vehicle (blue) follows the reference signal (red) generated in real time by the guidance algorithm.

Following simulation, a flight experiment was conducted to carry out the same mission on a real platform. In this case, the state of the system was determined via the base flight stack, which streamed data in from a Vicon motion capture system. Figure 5.5 plots the vehicle trajectory and the reference signal generated by the guidance algorithm. The results of the flight experiment are very similar to those of the simulation, demonstrating the success of the guidance algorithm in practical applications as well as simulations. The components of the vehicle trajectory and reference signal are plotted in Figure 5.6.

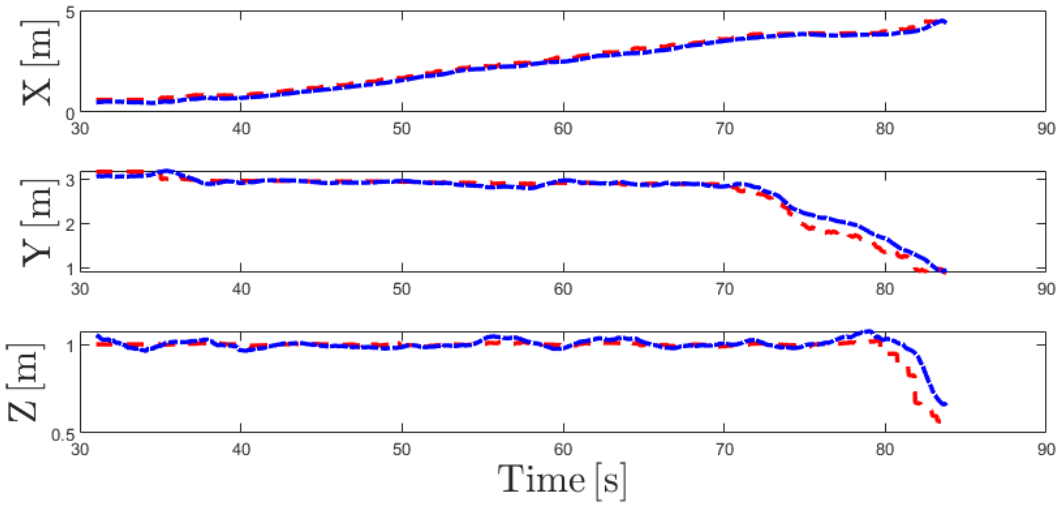


Figure 5.6: The components of the reference signal (red) and system position (blue).

Chapter 6: Conclusion

In this thesis, we set out to address a shortcoming of state-of-the-art pathfinding solutions—specifically in their ability to optimize beyond the shortest path problem. This is particularly problematic in the context of autonomous operation in hostile environments, where shortest-path trajectories could leave the system unnecessarily visible and vulnerable to adversarial action. In developing a solution, we made creative use of optimal control techniques—model predictive control in particular—to generate a locally optimal state trajectory for use as a reference signal rather than use the control input itself. This local stage is preceded by a global planner that implements Lifelong Planning A*, a modern graph-based search algorithm that offers fast replanning capabilities in dynamic environments. The novel guidance system resulting from this work was then implemented and tested on a quadrotor platform with satisfactory performance. Specifically, the vehicle was guided from one point to another via a trajectory that made use of an 'L'-shaped obstacle as cover, rather than an exposed, direct trajectory.

While this is a step forward for autonomous operations in hostile environments, it is far from the last. The next step in the evolution of this project is to implement visual navigation as described in Chapter 2. This addition would lift the requirement of prior map knowledge and allow the system to make state estimates independent of GPS or Vicon motion capture cameras. Additionally, while this thesis demonstrates proof of concept for one set of problem data, it does not explore the behavior of the trajectories when these data are altered. Work is currently ongoing in this regard to develop a taxonomy of flight behaviors using Taguchi methods.

In regard to further algorithmmic optimization of the guidance system, there is room

for improvement at the constraint generation stage. Currently, a new set of constraints is generated for every call to the model predictive control algorithm, However, further parallelization could be implemented to compute constraint sets for regions of the map ahead of time. As the local planner thread spends roughly half its time calculating these constraints, such parallelization could as much as double the frequency of the local planner—or allow for doubly long time horizons at current speed.

Finally, state-dependence of the system model could be implemented as discussed in Chapter 4 to allow for the generation of aggressive trajectories under nonlinear flight conditions.

Bibliography

[1] A. Schriver. "On the History of the Shortest Path Problem", *Documenta Mathematica*, Extra Volume: Optimization Stories, pp. 155–167, Aug. 2012.

[2] A, Stents. "Optimal and Efficient Path Planning for Partially-Known Environments", *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, 4, pp. 3310–3317, May 1994.

[3] K. Ogata. *Modern Control Engineering*, Prentice Hall, 2010.

[4] Y. Wang and S. Boyd. "Fast Model Predictive Control Using Online Optimization", *IEEE Transactions on Control Systems Technology*, 18(2), pp. 267–278, Mar. 2010.

[5] H. Durrant-Whyte and T. Bailey. "Simultaneous Localization and Mapping: Part I The Essential Algorithms", *IEEE Robotics & Automation Magazine*, 13(2), pp. 99–110, Jan. 2006.

[6] J. Engel, T. Schö ps, and D. Cremers. "LSD-SLAM: Large-Scale Direct Monocular SLAM", *Proceedings of the 13th European Conference on Computer Vision*, 2, pp. 834–849, Sep. 2014.

[7] R. Smith and P. Cheeseman. "On the Representation and Estimation of Spatial Uncertainty", *International Journal of Robotics Research*, 5(4), pp. 56–68, Dec. 1986.

[8] R. Smith, M. Self, and P. Cheeseman. "Estimating Uncertain Spatial Relationships in Robotics", *Machine Intelligence and Pattern Recognition*, 5, pp. 435–461, 1988.

[9] C. Harris and M. Stephens. "A Combined Corner and Edge Detector", *Proceedings of the 4th Alvey Vision Conference*, pp. 147–151 Jan. 1988.

[10] E. Rosten and T. Drummond. "Machine Learning for High-Speed Corner Detection", *Proceedings from the 9th European Conference on Computer Vision*, 1, pp. 430–443, July 2006.

[11] J. Bresenham. "A Linear Algorithm for Incremental Digital Display of Circular Arcs", *Commumications of the ACM*, 20(2), pp. 100–106, Feb. 1977.

[12] D. Lowe. "Object Recognition from Local Scale-Invariant Features", *Proceedings from the International Conference on Computer Vision*, 2, pp. 1150–1157, Feb. 1999.

[13] D. Lowe. "Distinctive Image Features from Scale-Invariant Keypoints", *International Journal of Computer Vision*, 60(2), pp. 91–110, Nov. 2004.

[14] M. Calonder, V. Lepetit, C. Strecha, and P. Fua. "BRIEF: Binary Robust Independent Elementary Features", *Proceedings from the 11th European Conference on Computer Vision*, 4, pp. 778–792, Sep. 2010.

[15] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski. "ORB: An Efficient Alternative to SIFT or SURF", *Proceedings from the International Conference on Computer Vision*, pp. 2564–2571, Nov. 2011.

[16] G. Klein. "Visual Tracking for Augmented Reality", *University of Cambridge*, PhD Thesis, 2006.

[17] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge, United Kingdom: Cambridge University Press, 2004.

[18] C. Domann. "Development and Simulation of a Novel Guidance System for Quadrotors Flying in a Contested Environment". Master's thesis. Unpublished.

[19] A. Richards. "Fast Model Predictive Control with Soft Constraints", *European Journal of Control*, 25, pp. 51–59, May 2015.

[20] C. Greatwood and A. Richards. "Implementation of Fast MPC with a Quadrotor for Obstacle Avoidance", *AIAA Guidance, Navigation, and Control Conference*, Aug. 2013.

[21] F. Zhang. *The Schur Complement and Its Applications*. Boston, Massachusetts: Springer, 2005.

[22] J. Stewart. *Calculus*, 7th ed. Cengage, 2012.

[23] D. Bertsekas. "Projected Newton Methods for Optimization Problems with Simple Constraints", *SIAM Journal on Control and Optimization*, 20(2), pp. 221–246, Apr. 1981.

[24] S. Wright. "Applying New Optimization Algorithms to Model Predictive Control", *Chemical Process Control-V.*, 93(316), pp. 147–155, June 1996.

[25] "What is SDP? (For a Beginner of SDP)", *SDPA Project*, http://sdpa.sourceforge.net/whatissdp.pdf, 2005.

[26] Y. Nesterov and A. Nemirovsky. *Interior-Point Polynomial Methods in Convex Programming*. Warrendale, Pennsylvania: SIAM, 1994.

[27] "ODROID-XU4", *Hardkernel*, https://wiki.odroid.com/odroid-xu4/odroid-xu4, 2018.

[28] E. Camacho and C. Bordons. *Model Predictive Control*. New York, Springer, 2004.

[29] P. Tøndel, T. A. Johansen, and A. Bemporad. "An Algorithm for Multi-Parametric Quadratic Programming and Explicit MPC Solutions", *Automatica*, 39(3), pp. 489–497, Mar. 2003.

[30] S. A. Curtis. "The classification of greedy algorithms", *Science of Computer Programming*, 49(1–3), pp. 125–157, Dec. 2003.

[31] E. W. Dijkstra, E. W.. "A note on two problems in connexion with graphs", *Numerische Mathematik*, 1, pp. 269–271, Dec. 1959.

[32] S. Liu, N. Atanasov, K. Mohta, and V. Kumar . "Search-based motion planning for quadrotors using linear quadratic minimum time control", *IEEE Conference on Intelligent Robotics and Systems*, 2017.

[33] S. Liu, K. Mohta, N. Atanasov, and V. Kumar. "Towards Search-based Motion Planning for Micro Aerial Vehicles", *arxiv*, 2018.

[34] S. Liu. "motion_primitive_library", (2018), GitHub repository, https://github.com/sikang/motion_primitive_library

[35] P. E. Hart, N. J. Nilsson, and B. Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths", *IEEE Transactions on Systems Science and Cybernetics SSC4*, 4(2), pp. 100–107, 1968.

[36] S. Koenig, M. Likhachev, and D. Furcy. "Lifelong Planning A*", *Artificial Intelligence*, 155(1–2), pp. 93–146, May 2004.

[37] S. Aine, S. Swarminathan, V. Narayanan, V. Hwang, and M. Likhachev. "Multi-Heuristic A*", *International Journal of Robotics Research*, 35(1–3), pp. 224–243, Aug. 2015.

[38] R. Dechter and J. Pearle. "Generalized best-first search strategies and the optimality of A*", *Journal of the Association for Computing Machinery*, 32(3), pp. 505–536, Jul. 1985.

[39] A. L'Afflitto, R.B. Anderson, and K. Mohammadi. "An Introduction to Nonlniear Robust Control for Unmanned Quadrotor Aircraft", *IEEE Control Systems Magazine*, pp. 102–121, 2018.

[40] A. L'Afflitto. "A Mathematical Perspective on Flight Dynamics and Control", Springer, 2017.

[41] J. S.-H. Tsai, C.-C. Huang, S.-M. Guo, and L.-S. Shieh. "Continuous to discrete model conversion for the system with a singlular system matrix based on matrix sign function", *Applied Mathematical Modelling*, 35, 2018.

[42] S. K. Phang, K. Li, K. H. Yu, B. M. Chen, and T. H. Lee. "Systematic Design and Implementation of a Micro Unmanned Quadrotor System", *Unmanned Systems*, 2(2), pp. 121-141, 2014.

[43] A. Tayebi and S. McGilvray. "Attitude Stabilization of a VTOL Quadrotor Aircraft", *IEEE Transactions on Control Systems Technology*, 14(3), pp. 562-571, 2006.

[44] "Intel Math Kernel Library (Intel MKL) 2019 System Requirements", *Intel*, https://software.intel.com/en-us/articles/intel-math-kernel-library-intel-mkl-2019-system-requirements, 2019.

[45] "Main Page", *Eigen*, http://eigen.tuxfamily.org/index.php?title=Main_Page, 2019.

[46] P. Greenhalgh. "Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7", *ARM*, 2011.