Planning and Control for Quadrotor Flight through Cluttered Environments

by

Benoit Landry

B.S., Massachusetts Institute of Technology (2014)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2015

Author . . . .
Department of Electrical Engineering and Computer Science
May 22, 2015

Certified by . . . .
Russ Tedrake
Associate Professor of Computer Science
Thesis Supervisor

Accepted by . . . .
Albert R. Meyer
Chairman, Masters of Engineering Thesis Committee

Planning and Control for Quadrotor Flight through Cluttered Environments

by

Benoit Landry

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2015, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Previous demonstrations of autonomous quadrotor flight have typically been limited to sparse environments due to the computational burden associated with planning for a large number of obstacles. We hypothesized that it would be possible to do efficient planning and robust execution in obstacle-dense environments using the novel Iterative Regional Inflation by Semidefinite programming algorithm (IRIS), mixed-integer semidefinite programs (MISDP), and model-based control approaches. Here, we present experimental validation of this hypothesis using a small quadrotor in a series of indoor environments including a cubic meter volume containing 20 interwoven strings. We chose one of the smallest hardware platforms available on the market (34g, 92mm rotor to rotor), allowing for these dense environments and explain how to overcome the many system identification, state estimation, and control problems that result from the small size of the platform and the complexity of the environments.

Thesis Supervisor: Russ Tedrake
Title: Associate Professor of Computer Science

Acknowledgments

I would like to thank my research advisor, Professor Russ Tedrake, for his generous advice and for never doubting my ability to push results further. I would also like to acknowledge Anirudha Majumdar for his constant guidance throughout the duration of this project. There is no doubt that his support was critical in my success. I would also like to thank Pete Florence, for his help modeling the quadrotor and for his invaluable assistance in running the experiments. Pete's insights on producing appealing results and demonstrations can be seen in every element of this work. I would finally like to thank Andy Barry, for sharing his infinite wisdom on hardware and graduate school in general. Andy never hesitated to volunteer his own time to help. Three o'clock coffee breaks with Ani, Pete and Andy kept me away from common research pitfalls more often than I would probably like to acknowledge. I would also like to thank the rest of our research group, who never ran out of ideas when my inspiration was failing me.

A special thank you to Lyndsey, whose loving support was essential at every step of the way. Lyndsey never failed to pick up the phone no matter how late my nights in the lab would drag until. I never saw her put anything else than my best interest at the heart of any advice she offered, often showing better understanding of myself than I ever had.

Finally, I must thank my family Michel, Luce, Marianne, Simon-Pierre, Vanessa and Liam. They will always have been the first ones to have shared my vision and they have never stopped pushing me to realize it.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The quadrotor is quickly becoming one of the canonical systems of modern robotics. It supports a wide range of payloads and offers the type of underactuated system that is still manageable for a lot of well-established linear control approaches. Moreover, advances in research have brought the quadrotor to a level of usability that makes it increasingly attractive for commercial applications like surveying and delivery. Despite these advances, the quadrotor still suffers from a variety of shortcomings that roboticists will need to overcome in order to get the platform to truly be a viable option for real-world applications. For example, flight times, usually in the order of minutes, are still too short to allow quadrotors to be effective delivery vehicles. Quadrotors, highly dynamic systems, also have fairly low reliability, something that can lead to spectacular failures. Quadrotors and small helicopters can be dangerous machines when hovering over crowded areas. These risks may be mitigated in part by increasing the robustness and efficiency with which they may detect and avoid obstacles.

A few algorithms have been put forward in order to enable quadrotors to fly through crowded environments. The most successful ones of those often require an exponential increase in planning time with respect to the number of obstacles. For example, by introducing an integer variable for each face of obstacles in the environment. Most of these algorithms therefore start to perform poorly as the number of obstacles are increased beyond a modest handful of obstacles.

Recently, it has been suggested that the challenges of increasing obstacle density may be overcome by the novel Iterative Regional Inflation by semidifinite programming algorithm (IRIS). Rather than enumerate the obstacles, the algorithm performs convex segmentation of free space by seeding it with polytopes enclosing an obstacle free ellipsoid that is iteratively grown. This approach is only limited by the mixed-integer program that is then used to plan a trajectory through the polytopes, and not by the number of obstacles. Hence, it can produce trajectories in environments containing many more obstacles than was previously demonstrated.

We hypothesized that it would be possible to fly a small quadrotor through environments containing greater number of obstacles than ever demonstrated before by leveraging IRIS, MISDPs, and model-based control approaches. Here we present experimental validation of this hypothesis. We choose one of the smallest platforms on the market (34g, 92mm rotor to rotor) allowing for even more dense environments and show how to overcome the other system identification, state estimation and control problems that result from the small platform and the crowded environments.

1.1 Experiment Overview

The goal of the proposed experiment was to test our hypothesis that IRIS and a model-based controller could fly a quadrotor in environments containing more obstacles than previously demonstrated. The quadrotor was controlled by the off-board software Drake and followed trajectories computed by IRIS and mixed-integer semidefinite programs. The sensing was accomplished with Vicon sensors as well as an IMU onboard the quadrotor. The exact location of the obstacles was given to the planner ahead of time, as a set of convex hulls. The level of performance of the planning algorithm was evaluated by looking at its ability to find collision-free trajectories in real-world environments that contained more obstacles than previously demonstrated, and the ability of the proposed control system to execute those trajectories.

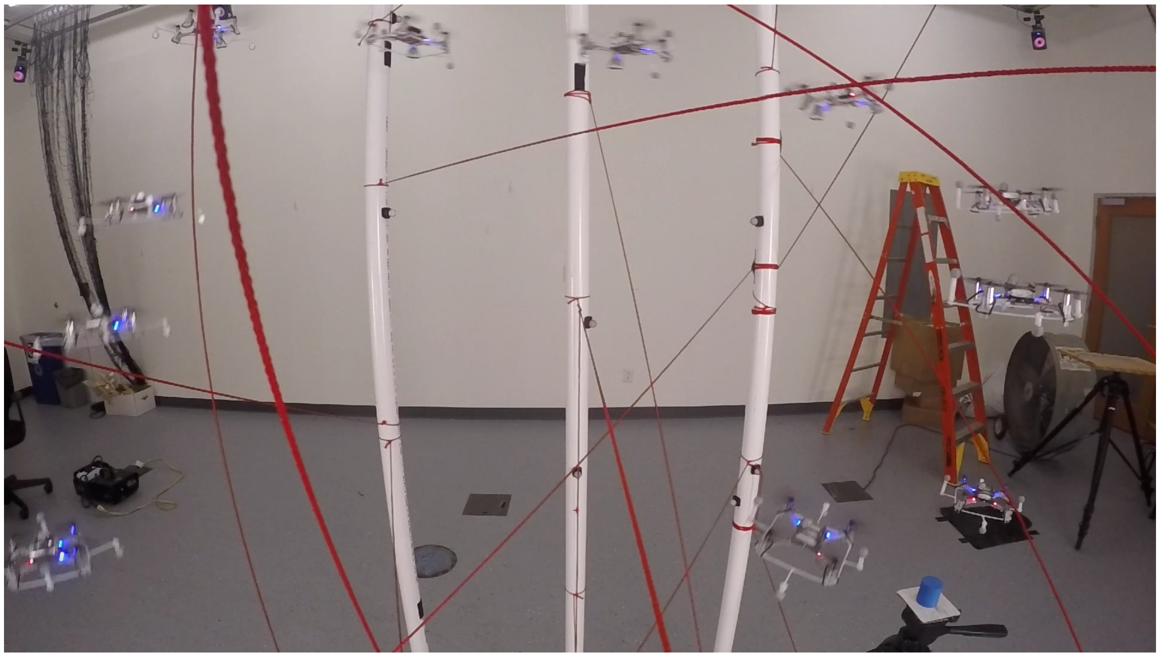


Figure 1-1: View from inside one of the obstacle-dense environments used in the experiment. This one shows a cubic meter volume containing 20 interwoven strings.

1.2 Related Work

1.2.1 UAVs in Obstacle-Dense Environments

There has been a few demonstrations of unmanned aerial vehicles (UAVs) maneuvering in environments with obstacles. [20] showed quadrotors capable of flying through small openings. [2] demonstrated an aircraft with rotating wings flying between two poles closer to each other than the plane's wingspan. [24] also demonstrated an algorithm for planning trajectories indoors with a quadrotor.

Even though [24]'s approach has been shown capable of navigating indoors, none of the model-based approaches have been demonstrated in overwhelmingly crowded environments.

1.2.2 Planning Collision-Free Trajectories

Rapidly-Exploring Random Trees (RRT*)

The approach in [24] consists of running RRT* in the entire space where the quadrotor might fly. The algorithm only expands the randomly-exploring tree with straight trajectories in order to make its expansion more efficient. It therefore results in a piece-wise linear collision-free trajectory. Finally, the algorithm computes a smooth trajectory using a quadratic program between each node along the path returned by RRT*.

Mixed-Integer Programs

[19] also demonstrated the use of mixed-integer quadratic programs in order to plan collision-free trajectories. In this particular work, the integer variable enforces non-penetration by making sure that the sampled location is on the collision-free side of at least one of the faces of each obstacle. The approach worked well in practice, but it suffered from having the number of integer variables grow rapidly with the number of obstacles. Moreover the technique cannot guarantee to generate collision-free trajectories between the sample points.

Differential Flatness

[18] is one of the most cited and first uses of differential flatness for planning with quadrotors. In their experiments, they set a series of desired poses through hoops and similar obstacles, and use a quadratic program to compute trajectories in the flat output space between those poses. The method was used again in later experiments [14]. An introduction to differential flatness is provided in section 4.1.

Iterative Regional Inflation by Semidefinite Programming (IRIS)

IRIS, our choice of algorithm for convex segmentation of free-space, has previously been used in the context of footstep planning [6]. In this application, the algorithm generates collision-free regions in the robot's configuration space. A mixed-integer

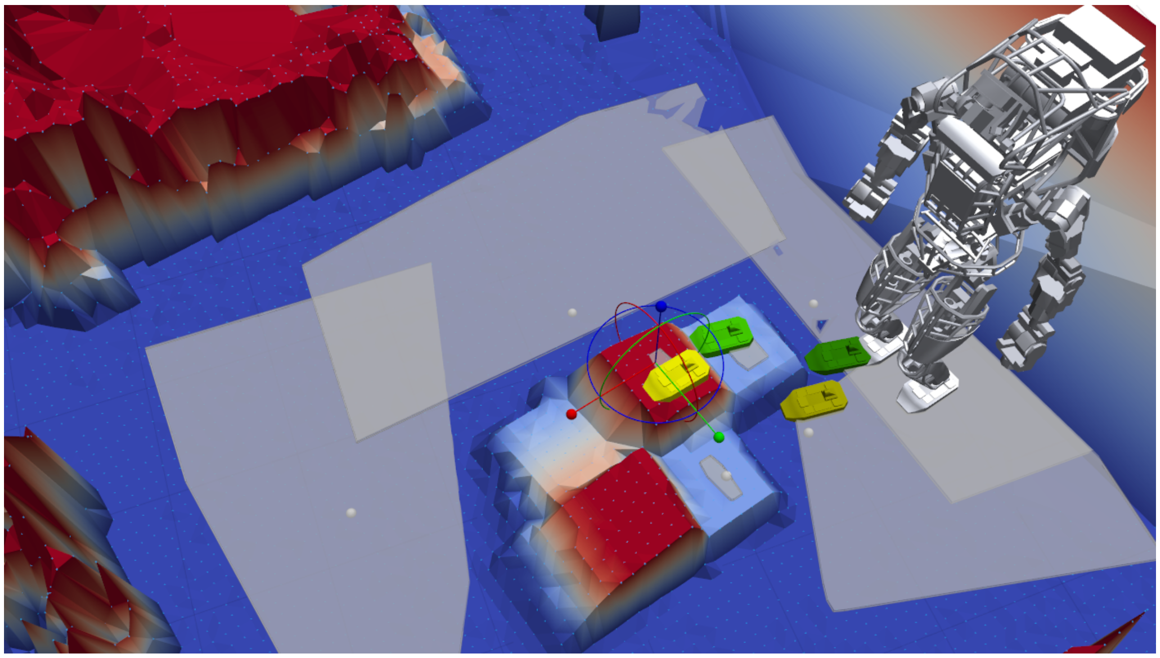


Figure 1-2: IRIS used for footstep planning on the Atlas robot. The gray areas are the safe regions computed by the algorithm. Image courtesy of Robin Deits [6]

program is then used to assign steps to each regions while simultaneously optimizing the pose of the robot.

[4] then went on to demonstrate that we can formulate the problem of planning minimum-snap collision-free trajectories as a mixed-integer semidefinite program by using the obstacles-free regions returned by IRIS and by planning in differentially flat space. This document in part aims to provide experimental demonstrations for these results.

1.2.3 Feedback Control for Path Following

Flat Outputs PD Controller

[20] demonstrated a feedback controller inspired by the work of Hoffmann et al. [12]. In this approach, the controller computes the position error e_p and the velocity error of the quadrotor e_v and then specifies the desired center of mass acceleration $\ddot{r}_{des}$

using those errors and feed-forward terms.

$$\ddot{r}_{des} = k_p e_p + k_d e_v + \ddot{r}_T(\hat{x}) \quad , \tag{1.1}$$

where $\ddot{r}_T(\hat{x})$ is the feed-forward term (the acceleration in the direction tangential to the trajectory) for the current state $\hat{x}$. The resulting acceleration of the quadrotor's center of mass is then mapped to a desired roll and pitch that are stabilized by an attitude controller.

Time-varying LQR

Finally, our own control strategy for the experiment, time-varying linear quadratic regulator (TVLQR), has already been shown capable of tracking extremely dynamic trajectories. For example, [2] demonstrated an aircraft with rotating wings executing a knife-edge maneuver in order to fly between two poles without colliding with either of them.

Chapter 2

Hardware

The proposed experiment uses the Crazyflie Nano Quadcopter. The development of the Crazyflie started in 2009 as a side project by three employees of the Swedish consulting company Epsilon AB. In 2010, a video showing the first prototype of the Crazyflie was sent to the website Hackaday.com and received a lot of attention. After the success of the video, the three creators of the Crazyflie decided to launch Bitcraze AB, that eventually became Bitcraze.se, in order to finance the development and manufacturing of the quadrotor. The version 1.0 was available during the first months of the experiment. A second version of the Crazyflie became available in December 2015. We started developing on the Crazyflie 1.0, but eventually switched to the Crazyflie 2.0, since this last one was equiped with a higher energy capacity battery, slightly larger motors and a better on-board micro-controller [1].

2.1 Crazyflie 2.0

The Crazyflie 2.0 is one of the smallest quadrotors available on the market. Its small size and general safety makes it an ideal candidate for applications that require it to fly near people. It measures 92mm from propeller to propeller and weighs 34g with optical markers mounted on it. It has a flight time of around 5 minutes. The small size of the Crazyflie presents many challenges in using it as a research platform. Its small inertia requires controllers that can react with very little latency. Its payload

(a) Version 1.0 (b) Version 2.0

Figure 2-1: The Crazyflie Nano Quadcopters by Bitcraze.se

capacity is also limited, making it challenging to add additional sensors.

On the other hand, a great advantage of the Crazyflie is that all the software produced by Bitcraze is fully open-source. This gives us complete control over the firmware running on the quadrotor, the firmware running on the radio as well as the client library running on the base-station. Bitcraze hosts all of its codebase on Github. In the same spirit we distribute the entirety of our software on Github as well[1].

2.1.1 Microcontrollers

The Crazyflie 2.0 is equipped with two microcontrollers. The first one, an ARM Cortex-M4 embedded processor (STM32F405), is used to run the main application. The processor is a 32 bits architecture and can run at 168MHz. The ARM Cortex-M4 also has a floating point unit that support all ARM single-precision data-processing instructions and data types [22]. The Crazyflie 2.0 also has a second microcontroller to handle power management and the radio. That microprocessor is an ARM Cortex-M0 (nRF51822) that runs at 32MHz [23]. This second microcontroller has Bluetooth capability but it was not leveraged in this experiment due to lack of a library for it on the client side as well as good reasons to believe it would provide poorer performance

[1] https://github.com/blandry/crazyflie-tools

than the Crazyradio USB dongle.

2.1.2 Inertial Measurement Unit (IMU)

The Crazyflie Nano Quadcopter is equipped with an IMU, the MPU-9250. The IMU contains a 3-axis gyroscope and a 3-axis accelerometer [13]. The experimental setup described here relies entirely on the IMU to get orientation measurements, both for the on-board and off-board controllers. Note that these measurements are given in the sensor frame, which is oriented in what is known as the "X" configuration with its x-axis pointing between motor number 4 and 1. Since our models assume an x-axis pointing toward motor 1, we rotate the measurements by 45° using a constant rotation matrix before using them.

2.1.3 Other On-Board Sensors

The Crazyflie is also equipped with an on-board magnetometer as well as a barometer. However the radio protocol limits the size of the packets being sent from the Crazyflie to the Crazyradio to 29 bytes of payload. This corresponds to a maximum of 7 floating point numbers on the ARM micro-controller, and therefore prevents us from sending the entirety of the sensor measurements with each packet without cutting in half the frequency at which the state estimator gets sensor updates.

2.1.4 Battery and Power

The battery used by the Crazyflie 2.0 is a single cell Lithium-Polymer (LiPo) battery. This is the most popular type of battery in the R/C industry and provides the best power to weight ratio. It supplies 3.7V and has a capacity of 240mAh. The battery also comes with a Protection Circuit Module (PCM) attached to it that prevents the user from under or over charging the battery or from shorting it[2]. The battery is easily removable from the quadrotor so connectors were built for it so that they could

[2]http://wiki.bitcraze.se/projects:crazyflie:hardware:explained

be charged in parallel and reduce delay between experiments. The battery's discharge rate is 15C, which in theory should provide 4 minutes of continuous flight.

2.1.5 Motors

The Crazyflie 2.0 has four brushed DC motors. The motors are coreless which in theory provides faster acceleration. They can produce 12000rpm per volt, with a nominal voltage of 4.2V. Those numbers allow us to compute the theoretical maximal thrust that is predicted by the k_F parameter we will describe later. The most interesting point about the motors is that most quadrotors use brushless motors, which use an electronic circuit to accurately regulate their speed with respect to the input signal. The Crazyflie 2.0 motors are however not brushless, and are powered by an unregulated power supply. Therefore higher motor speeds, requiring more torque in order to fight the increasing air resistance and therefore more current, tend to make the battery voltage drop. As we shall see later, this can be remedied with software feedback.

2.1.6 Propellers

The propellers are conventional 45mm plastic propellers. They have a tendency to bend during collisions and so experimenters should make sure they regularly change the propellers for new ones in order to give the controller optimal conditions.

2.2 Radio

The Crazyflie is equipped with a Nordic Semiconductor nRF51822 which handles radio communication. The easiest way to communicate with the chip is to use the Crazyradio. The Crazyradio is a USB dongle that integrates a Nordic Semiconductor nRF24LU1+. The chips can communicate with each other over the 2.4GHz ISM band. Both chips can be reprogrammed, although the nRF51822 on the Crazyflie requires a JTAG connector, while the nRF24LU1+ can be reprogrammed over USB.

Figure 2-2: The Crazyradio by Bitcraze.se

The radios can be run at up to 2Mbs.

2.3 Optical Tracking

Our experimental setup contains both on-board and off-board sensors. First, the on-board gyroscope and accelerometer provide measurements that can be used to estimate the state of the quadrotor. Second, an optical tracking system provides additional information on the position of the quadrotor. Optical tracking has been extensively used to perform similar experiments in the past. They are often considered “ground truth” for these types of demonstrations.

The Vicon tracking system uses infrared cameras to localize small reflecting markers. The markers come in various sizes. The system used in the experiment runs at 120Hz. Even though Vicon provides both position and orientation measurements, we found that its orientation measurements for the Crazyflie were very noisy, most likely due to the small size of our platform, and we therefore opted for only using its position measurement.

2.4 Marker Frame

Because of the size of the Crazyflie Nano Quadcopter, it was not possible to mount the optical markers directly on the quadrotor and still have the optical tracking system

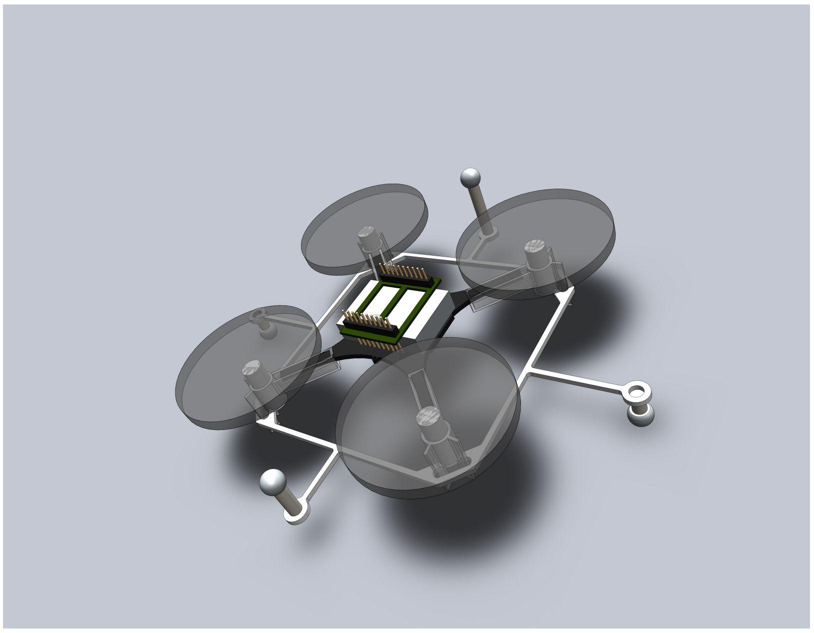


Figure 2-3: CAD rendering of the polyoxymethylene frame used to hold the markers mounted on the Crazyflie.

distinguish between them. We therefore designed a simple frame that we laser-cut out of polyoxymethylene. The frame weighs a total of 2.7g and greatly improves our ability to track the quadrotor.

Chapter 3

Software

3.1 Firmware

The Crazyflie Nano Quadcopter firmware is based on FreeRTOS, an open source real-time operating system. The firmware currently distributed with the Crazyflie runs a variety of services called "tasks". Commands are usually sent to the quadrotor using the Crazy RealTime Protocol (CRTP) described below and an internal proportional integral derivative (PID) controller stabilizes the desired Euler angles contained in the commands. This firmware has been modified for the purpose of this experiment. We do not actually use any of the controller software distributed with the Crazyflie.

The software on both the nRF51822 and the STM32F405 is modifiable. However only the firmware of the main micro controller (STM32F405) was modified for the experiment. The firmware was modified to optimize two tasks: the forwarding of the latest control inputs to the actuators and of the measurements to the base station. Like in the original firmware, the nRF51822 controls the antenna and does not process any of the incoming or outgoing data. Instead it forwards it to the STM32F405 using the serial communication between the two. The Syslink task in the STM32F405 then builds packets from the incoming data stream. In order to reduce the latency in the control loop, the Syslink task then immediately checks if the packet contains control input (desired euler angles and angular velocities) or sensor requests. If it contains control inputs, it simply updates the memory location where the Offboardctrl

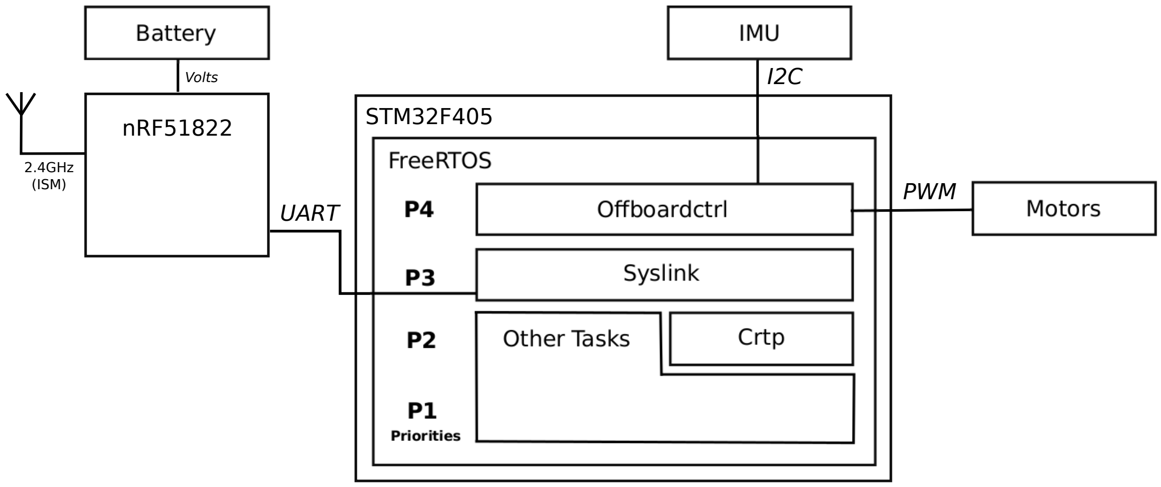


Figure 3-1: Overview of the firmware architecture.

task looks for desired euler angles and angular velocities. On its next iteration, the Offboardctrl task will use these desired values to compute motor commands using a simple PD controller. If instead the packet is an empty packet received on the sensor channel (a sensor request), then it reads the memory location where the Offboardctrl stores the last IMU measurements, makes a packet and sends it back on the UART. Finally if the packet is neither of those, it proceeds with the original communication scheme by putting it in the delivery queue of the Crtp task for later dispatching to the appropriate task. Prioritizing packets related to sensors and actuator inputs prevents build-up in the firmware queues. It also always favors the most recent control signals, disregarding ones that have not made it through the controller or the actuators yet if needed.

3.2 Python Controller

Bitcraze distributes an open-source Python library that allows users to develop their own client applications. The library provides simple calls to the USB radio that can be sent to the Crazyflie. It also allows the user to define callbacks for received data from the quadrotor. The library was modified to improve its performance in the context of our experiment. We used the modified library to write a multi-threaded Python controller that runs on the base station.

3.2.1 Main Thread

The main thread takes care of synchronizing the different components of the control system. It launches the state estimator and the controller. It then only manages the communication loop. The communication loop consists of requesting sensor measurements, receiving those measurements, forwarding those measurements to the state estimator and getting the latest state estimate, then requesting control inputs for that estimate and finally forwarding them to the quadrotor.

3.2.2 Controller Thread

The controller thread either computes control inputs from a state estimate and a time-invariant linear controller, or keeps track of the latest control inputs sent by Drake over an LCM channel. When using the control inputs sent by Drake, the main thread attempts to minimize the delay between the time the latest state estimate was published on the LCM channel Drake is listening to and the time a control input is recovered from the controller thread. This is accomplished by pausing the main thread with a lock immediately after it broadcasts the state estimate, and having the controller thread unpause it by removing the lock as soon as it receives a control input from Drake over LCM.

3.2.3 Estimation Module

The state estimator was implemented as its own separate module that contains both python and C code. The state estimator receives Vicon measurements over an LCM channel and IMU measurements from the main thread running the communication loop. It combines them using a low pass filter, a complementary filter and an extended Kalman filter.

3.2.4 Manual Commands

Several experiments required us to send arbitrary inputs to the quadrotors. Therefore we also implemented a simple client that allows us to send manual commands, like

```
  7   6   5    4   3   2   1   0
+----+----+----+----+----+----+----+----+
|       Port       |  Link  | Chan. |
+----+----+----+----+----+----+----+----+
```

Figure 3-2: The CRTP header byte.

switching from LQR stabilization to time-varying LQR at the push of a button, from a USB board called the nanoKONTROL.

3.3 Drake

Drake is a robotics toolbox developed by our research group implemented in MATLAB and C++ [26]. The package allows us to implement a model of the quadrotor using the Unified Robot Description Format (URDF). It also allows us to easily compute and run (over LCM) various controllers like the time-varying LQR demonstrated in this experiment.

3.4 Crazy RealTime Protocol (CRTP)

One can communicate with the Crazyflie using the Crazy RealTime Protocol (CRTP). A CRTP packet contains one header byte and 29 bytes of payload. The header contains three fields. The first one, the port, specifies what application the packet is directed to. The second field, the link, is currently unused. Finally the channel field is used to specify sub-task or functionality. Currently, both the up and down link use this protocol. However the down link is created by adding a payload to each acknowledgment expected by the radio. This, along with the limited size of the CRTP packets, are the main challenges in the use of the protocol [1].

[1]https://wiki.bitcraze.io/projects:crazyflie:crtp

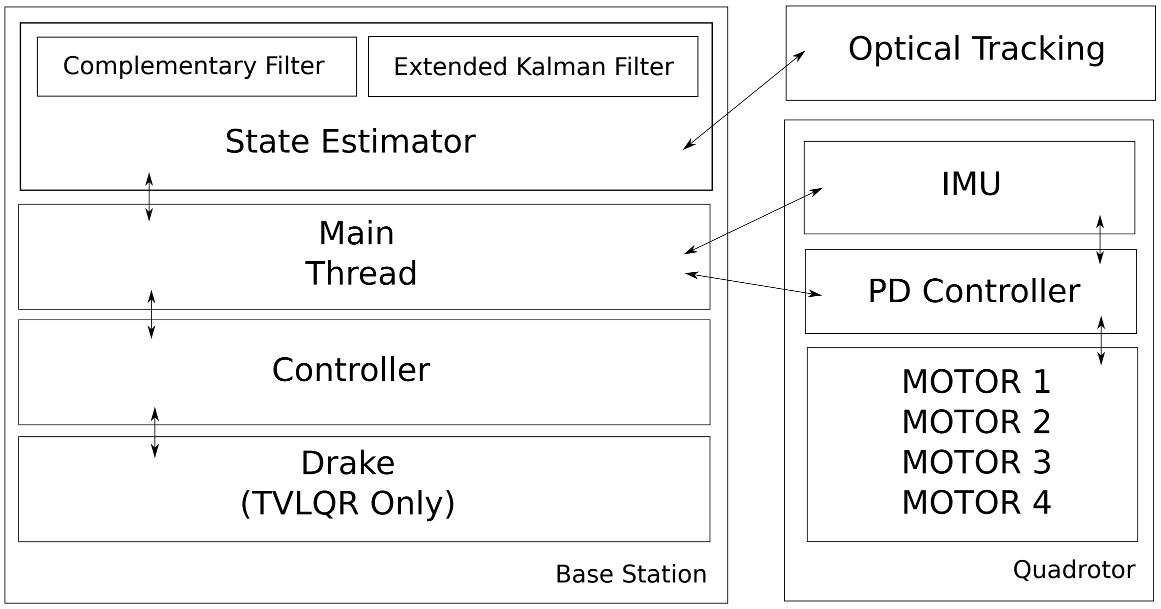


Figure 3-3: Overview of the software architecture.

3.5 Lightweight Communication and Marshalling (LCM)

We opted for the Lightweight Communication and Marshalling (LCM) package in order to link the different components of our control system. LCM allows the user to define different message types and pass them seamlessly over a network by either publishing or subscribing to specific channels.

Chapter 4

Modeling

4.1 Differentially Flat Quadrotor Model

Differentially flat systems were first introduced in 1992 by Fliess and al. [8]. Generally, a system is said to be flat if there exists a set of output, in equal number to the number of inputs, such that all the states of the system can be computed from these outputs (without integration). Once flat outputs are identified, it is possible to generate plans in the flat output space only, and extract the corresponding full-state trajectory at a later stage.

Planning in the flat output space also has limitations. For example, differential flatness alone does not guarantee that one can follow arbitrary trajectories in output space. For example, the system 4.1 proposed by Nieuwstadt [27] has the flat output x_2, but it does not allow trajectories with a negative $\dot{x}_2$.

$$\begin{aligned} \dot{x}_1 &= u, \\ \dot{x}_2 &= x_1^2. \end{aligned} \tag{4.1}$$

Even though our IRIS segmentation is agnostic to the plant model, the mixed-integer semidefinite program does require it to be differentially flat. We therefore use the differentially flat quadrotor model introduced in [18] also used by others [24]. The model consists of a single floating rigid body and four inputs. We define the inputs as the square of the angular velocities ω^2 of the motors on the quadrotor. A spinning propeller produces two forces on the quadrotor, namely lift and drag. Those forces are directly proportional to ω^2. Each input ω_i^2 can therefore be said to produce a certain linear force F_i on the center of mass as well as to create a moment M_i according to

$$F_i = k_f \omega_i^2, \tag{4.2}$$

$$M_i = k_m \omega_i^2. \tag{4.3}$$

We can then write the dynamics of the quadrotor:

$$m\ddot{r} = mg\mathbf{z}_W + \left(\sum_{i=1}^{4} F_i \right) \mathbf{z}_B, \tag{4.4}$$

$$\dot{\boldsymbol{\omega}} = I^{-1} \left(-\boldsymbol{\omega} \times I\boldsymbol{\omega} + \begin{bmatrix} 0 & k_f L & 0 & -k_f L \\ -k_f L & 0 & k_f L & 0 \\ k_m & -k_m & k_m & -k_m \end{bmatrix} \begin{bmatrix} \omega_1^2 \\ \omega_2^2 \\ \omega_3^2 \\ \omega_4^2 \end{bmatrix} \right), \tag{4.5}$$

where m is the mass of the quadrotor, $\ddot{r}$ is the acceleration of its center of mass, $\mathbf{z}_W$ is a unit vector in the direction of gravity, $\mathbf{z}_B$ is a unit vector pointing in the same direction as the propellers (in the world frame), I is the inertia matrix of the quadrotor, L is the distance between each propeller and the center of mass of the quadrotor, and $\boldsymbol{\omega}$ is the angular velocity of the quadrotor in the body frame [24].

The flat outputs in this model are the positions x, y, z and ψ where ψ is the yaw of the quadrotor.

4.2 System Identification

The model described above requires the identification of certain parameters of the Crazyflie. We solve the problem of parameter identification with a two step approach. First, we directly or semi-directly measure each parameter of the model with a series of simple experiments. Second, we log flight data for a series of maneuvers, and use

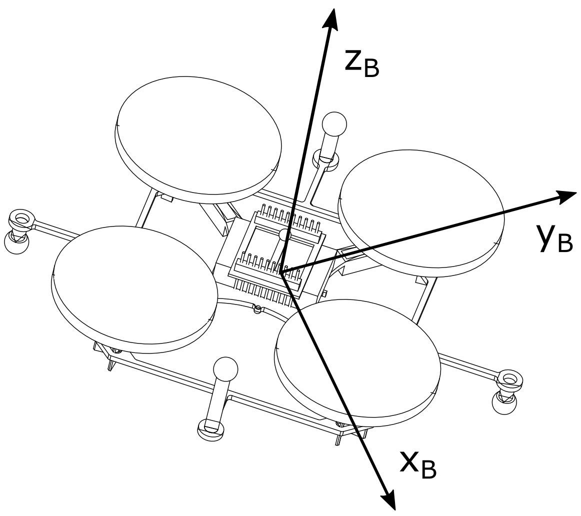


Figure 4-1: The body frame used in the differentially flat model developed by [18]. $\mathbf{z}_B$ points in the same direction as the propellers in the world frame.

an optimization-based algorithm to adjust those parameters.

First, we made a detailed model of the quadrotor using *SolidWorks* and extracted the inertia estimate from that model.

The k_f parameter was then identified by measuring the thrust produced by the quadrotor placed upside-down on a scale and fitting the corresponding parameter.

$$F = k_f(\omega_1^2 + \omega_2^2 + \omega_3^2 + \omega_4^2). \tag{4.6}$$

Note that we had to add the controller described in section 7.2 to the quadrotor's firmware in order for this input model to work.

The k_m parameter can then be measured by slowly increasing a pair of opposite motors (motors spinning in the same direction) and measuring the resulting angular velocities using the on-board gyroscope. This test is possible because the resulting spinning motion of the quadrotor acts like a mechanical gyroscope stabilizing it without the use of any feedback control. This enables us to perform long enough flights (a few seconds) to acquire the needed data. We then differentiate the gyroscope rate, filter it, and use the relationship below to fit the k_m parameter.

$$w^2 := \omega_2^2 + \omega_4^2 - \omega_1^2 - \omega_3^2 \quad , \tag{4.7}$$

$$I_{xz}\dot{\omega}_x + I_{xy}\dot{\omega}_y + I_{zz}\dot{\omega}_z = k_m w^2 - (I_{xy}\omega_x + I_{yy}\omega_y + I_{yz}\omega_z) + \omega_y(I_{xx}\omega_x + I_{xy}\omega_y + I_{xz}\omega_z), \tag{4.8}$$

where ω_x, ω_y and ω_z are the angular velocities of the quadrotor around x_B, y_B and z_B (the axes in the body frame).

The second step in our parameter identification approach is to log a series of flights with the quadrotor and fit the resulting data by tuning the parameters of our model. This can be done using MATLAB's Grey-box system identification toolbox. The initial guesses given to the algorithm are the ones measured as explained above.

Because the quadrotor is a highly unstable system, one of the challenges of this second step is that it is hard to get long enough flights when using purely open-loop trajectories. It is therefore tempting to use a simple controller like a PD controller in order to extend the time we can acquire data to fit. However, adding a controller can render some of the parameters unidentifieable [10].

Imagine that we were trying to perform system identification on the following simple linear system:

$$\dot{\mathbf{x}} = A\mathbf{x} + B\mathbf{u}. \tag{4.9}$$

The grey-box approach we are using could then be understood as trying to compute the entries of A and B from data on the evolution of $\mathbf{x}$ and $\mathbf{u}$ over time. Now if we use a linear feedback controller in order to stabilize the system, we end up with the following system:

$$\mathbf{u} = -K\mathbf{x} \quad , \tag{4.10}$$

$$\dot{\mathbf{x}} = A\mathbf{x} + B(-K\mathbf{x}), \tag{4.11}$$

$$\dot{\mathbf{x}} = (A - BK)\mathbf{x}, \tag{4.12}$$

$$\dot{\mathbf{x}} = \tilde{A}\mathbf{x}. \tag{4.13}$$

Now even though we know K, we do not know any of the entries of B, and therefore

Parameter	Value
I_{xx}	$2.3951 \cdot 10^{-5} kg \cdot m^2$
I_{yy}	$2.3951 \cdot 10^{-5} kg \cdot m^2$
I_{zz}	$3.2347 \cdot 10^{-5} kg \cdot m^2$
K_m	$1.8580 \cdot 10^{-5} N \cdot m \cdot s^2$
K_f	$0.005022 N \cdot s^2$

Table 4.1: The parameters identified with our approach for the Crazyflie.

cannot distinguish between A and B when identifying the entries of $\tilde{A}$. This is referred to as loosing the system's identifiabiliy. There has been substantial work in identification of closed-loop systems [10, 9] that in some cases have even performed better than some open-loop identification methods [11].

Our approach to this problem is therefore to use an open-loop control input that also contains a stabilizing input. First we put a stabilizing controller around the plant. Then we also send an additional open-loop input that pushes the system slightly out of equilibrium:

$$u = -K\tilde{x} + u_0. \tag{4.14}$$

This results in slightly unstable flights that generate the right amount of data for the system identification algorithm. When fitting the data, we model the quadrotor dynamics with the stabilizing controller inside the plant and only take the open-loop component of that experiment as the actual input to the system. In our linear system example from above, this would be equivalent to identifying the following system:

$$\dot{\mathbf{x}} = (A - BK)\mathbf{x} + Bu_0. \tag{4.15}$$

Note that we also take the delay of the control loop into account when running the optimization by shifting the input tape by the appropriate number of time steps. This value can be measured by placing motion capture markers on the propellers of the quadrotor and computing the delay between a step input and the resulting velocity of the propeller. In our case this resulted in a 28ms delay.

Chapter 5

Planning

We compute collision-free trajectories for the quadrotor prior to it flying through the obstacle field using the work of Deits on convex segmentation of free space and mixed-integer semidefinite programs [4]. The algorithm consists of first segmenting space into convex polytopes that are obstacle-free. This step requires our obstacles to be described as a set of convex hulls. The segmentation is done using the IRIS algorithm. We then use a mixed-integer semidefinite program to assign a series of polynomial trajectories to the polytopes in a way that both guarantees no collision and that minimizes snap.

5.1 Generating Convex Safe Regions

First, we inflate the convex hulls representing the obstacles by moving each one of their planes in the direction of their normal by an amount equal to the radius of the quadrotor. This then allows us to treat the quadrotor as a dimensionless object for the rest of the planning.

We compute convex regions of safe space using the software IRIS (Iterative Regional Inflation by Semi-definite programming). IRIS solves the problem of segmenting 3D space into a set of convex regions through a series of convex optimizations. More specifically, IRIS alternates between two optimizations. The first optimization is a quadratic program that finds hyperplanes that separate a convex region of space

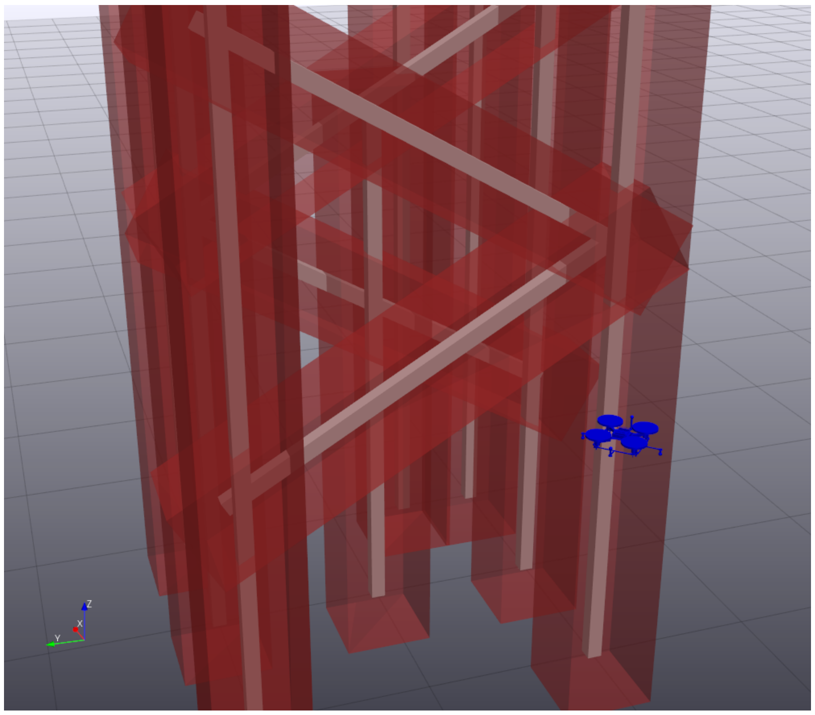


Figure 5-1: Example of the result of inflating obstacles of one of the environments. The red faces are the inflated obstacles and the white ones the original ones. The inflation then allows us to treat the quadrotor as being dimensionless.

from the obstacles. The second optimization is a semidefinite program that takes the intersection of those hyperplanes, and finds an ellipsoid of maximum volume inscribed inside those hyperplanes.

IRIS starts with an initial point q_0 and a list of obstacles. In some applications, IRIS has used a human operator in order to find those initial points. This had the advantage of letting the operator define which regions of space should be used by the robot and which ones should not [5]. Here we however leverage other work from Deits that uses a simple heuristic in order to seed the ellipsoid. The heuristic separates the space into a discrete grid, and computes the cell that is the farthest from any obstacles or previously computed polytope. The output of IRIS is a set of “safe regions” than can each be written as a set of hyperplanes:

$$P = \{x \quad | \quad Ax \leq b\}. \tag{5.1}$$

5.2 Generating trajectories with an MISDP

We then to design a piecewise polynominal trajectory that goes from our start point to our goal while staying inside the safe-regions returned by IRIS by following the procedure described in [4]. As explained in section 4.1, we can plan these trajectories by fixing the quadrotor yaw's to zero and only consider x, y and z. These trajectories are given an arbitrary timespan between 0 and 1 that can later easily be scaled to tune the velocity at which the quadrotor executes the trajectory. We parametrize the trajectories using the coefficients of the picewise polynominal. We also constraint the resulting trajectory to have continuous derivatives up to the $(d-1)^{\text{th}}$ derivative, where d is the degree of each polynomial piece.

We can write the assignment of polynomial pieces to convex regions of free space with a matrix of binary variables, $H \in \{0,1\}^{R\times N}$, where R is the number of convex regions and N is the number of polynomial trajectory pieces. If $H_{r,j} = 1$, then polynomial j must be contained within region r. We then constraint that each polynomial piece be fully contained in at least one region. This constraint can be represented as a combination of linear 5.2 and semidefinite constraints for a polynomial of arbitrary degree. The binary constraint can be written as

$$\sum_{r=1}^{R} H_{r,j} = 1 \quad \forall j \in 1, \dots, N, \tag{5.2}$$

and we can formulate the semidefinite constraint as sum of squares constraint. First we define the polynomial j as a linear combination of polynomial basis functions $\phi_1(t), ..., \phi_{d+1}(t)$.

$$P_j(t) = \sum_{k=1}^{d+1} C_{j,k}\phi_k(t) \quad t \in [0,1], \tag{5.3}$$

Therefore if $H_{r,j}$ is set to 1, then the polynomial must be contained inside the safe region r

$$A_r P_j(t) \leq b_r \quad \forall t \in [0,1], \tag{5.4}$$

or

$$A_r \sum_{k=1}^{d+1} C_{j,k}\phi_k(t) \leq b_r \quad \forall t \in [0,1]. \tag{5.5}$$

The constraint 5.5 can be written as m constraints of the form

$$a_{r,l}^T \sum_{k=1}^{d+1} C_{j,k}\phi_k(t) \leq b_{r,l} \quad \forall t \in [0,1], \tag{5.6}$$

where

$$A_r = \begin{bmatrix} a_{r,1}^T \\ a_{r,2}^T \\ \vdots \\ a_{r,m}^T \end{bmatrix} \text{ and } b = \begin{bmatrix} b_{r,1} \\ b_{r,2} \\ \vdots \\ b_{r,m} \end{bmatrix}. \tag{5.7}$$

By redistributing the terms in 5.6 in order to get

$$\sum_{k=1}^{d+1} (a_{r,l}^T C_{j,k})\phi_k(t) \leq b_{r,l} \quad \forall t \in [0,1], \tag{5.8}$$

with which we can now define $q(t)$

$$q(t) := b_{r,l} - \sum_{k=1}^{d+1} (a_{r,l}^T C_{j,k})\phi_k(t) \geq 0 \quad \forall t \in [0,1], \tag{5.9}$$

from which we can define our sum of square constraint since $q(t) \geq 0 \quad \forall t \in [0,1]$ if and only if $q(t)$ can be written as

$$q(t) = \begin{cases} t\sigma_1(t) + (1-t)\sigma_2(t) & \text{if } d \text{ is odd} \\ \sigma_1(t) + t(1-t)\sigma_2(t) & \text{if } d \text{ is even} \end{cases} \tag{5.10}$$

$$\sigma_1(t), \sigma_2(t) \text{ are sums of squares} \tag{5.11}$$

We then use the same cost defined by [18], a quadratic cost function on the square of the fourth derivative with respect to time (often called snap). If we are designing a trajectory of degree $d \leq 3$, we use the squared norm of the d^{th} derivative of the

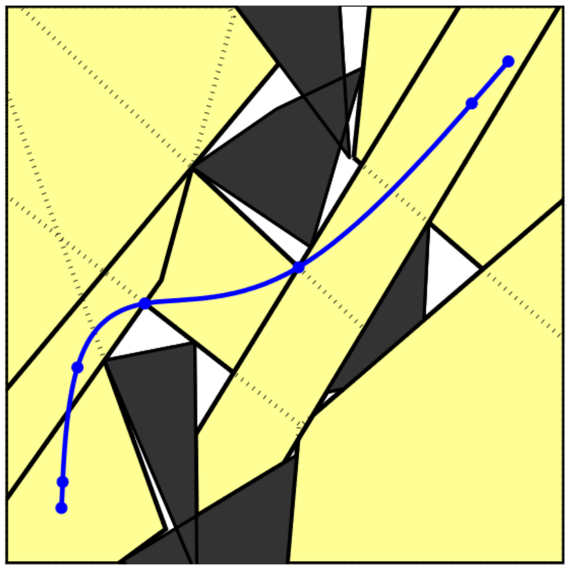


Figure 5-2: Example of a trajectory computed by IRIS and the MISDP formulated in [4]. The yellow regions are the safe regions, the gray ones are obstacles and the piecewise polynomial trajectory is shown in blue. Image courtesy of Robin Deits [4]

polynomial.

The full trajectory optimization therefore becomes a mixed-integer semidefinite program (MISDP), where the integer variables are the assignments to safe regions $H_{r,j}$ and the semidefinite program finds minimum-snap polynomial trajectories in those safe regions. Like in [4], we do not directly solve the MISDP because it tends to be numerically unstable with our solver Mosek [21]. Instead we solve the problem of degree $d = 3$ which can be formulated as a mixed-integer second-order cone program (MISOCP) and use the resulting integer solution as the integer solution of the degree-5 MISDP.

Chapter 6

State Estimation

We perform state estimation by fusing two different sources of sensor information: position from the optical tracking system (x, y and z) and IMU readings (gyroscope and accelerometer). Even though Vicon (our optical tracking system) could also provide a measurement of the orientation of the quadrotor, we found that the platform's small size tended to generate noisy measurements of its orientation. The IMU data is passed through a nonlinear complementary filter to estimate the orientation of the quadrotor. Then the roll, pitch, yaw, accelerometer readings and Vicon position measurement are passed to an extended Kalman filter to produce final estimates of the quadrotor's position and velocity. It has not escaped our attention that the Kalman filter would be capable of performing orientation estimation as well, but the overall good performance and stability of the nonlinear complementary filter convinced us to keep it as part of the control system. It also has the advantage of keeping the Kalman filter very straightforward.

6.1 Complementary Filter

We use the filter already implemented in the Crazyflie Nano Quadcopter for the onboard atittude estimation. We also modified the same implementation of the filter by [15] in order to provide attitude estimates to the offboard controller. The filter is based on Mahony's work on complementary filters [16]. The idea is to compute an error e_k between the predicted acceleration due to gravity α_k and a_k the measured one from the accelerometer in order to filter measurements and then integrate the filtered angular velocities to get an attitude estimate θ_k. The filter also applies an integral feedback I_k.

$$\theta_k^- = \theta_{k-1} + (\omega_k + I_{k-1})\Delta t, \tag{6.1}$$

$$e_k = a_k - \alpha_{k-1}, \tag{6.2}$$

$$\theta_k = \theta_k^- + K_p e_k \Delta t, \tag{6.3}$$

$$I_k = I_{k-1} + K_i e_k \Delta t. \tag{6.4}$$

6.2 Extended Kalman Filter

We modified the extended Kalman filter proposed by Bloesh et al. [3] in order to estimate the quadrotor's position and velocity. We first define the following states

$$x := \begin{bmatrix} r & v & b_f \end{bmatrix}, \tag{6.5}$$

where r is the position of the center of mass of the quadrotor, v is its velocity and b_f is the bias of the accelerometer. We define the inputs to the filter to be the current roll, pitch and yaw estimate and the last accelerometer measurement $\tilde{f}$. Our only measurement is the optical tracking $\tilde{r}$. After defining C as the rotation matrix corresponding to the orientation input (rotating vectors from the inertial into the body coordinate frame), we can then write the predict dynamics as such:

$$\hat{r}_{k+1} = \hat{r}_k + \Delta t \hat{v}_k + \frac{\Delta t^2}{2}(\hat{C}_k^T \hat{f}_k + g), \tag{6.6}$$

$$\hat{v}_{k+1} = \hat{v}_k + \Delta t(\hat{C}_k^T \hat{f}_k + g), \tag{6.7}$$

$$\hat{b}_{f,k+1} = \hat{b}_{f,k}, \tag{6.8}$$

with its Jacobian

$$\begin{bmatrix} I & \Delta t I & -\frac{\Delta t^2}{2}\hat{C}_k^T \\ 0 & I & -\Delta t \hat{C}_k^T \\ 0 & 0 & I \end{bmatrix}. \tag{6.9}$$

We also define the process covariance as such

$$\begin{bmatrix} \frac{\Delta t^3}{3}Q_f + \frac{\Delta t^5}{20}Q_{bf} & \frac{\Delta t^2}{2}Q_f + \frac{\Delta t^4}{8}Q_{bf} & -\frac{\Delta t^3}{6}C_k^T Q_{bf} \\ \frac{\Delta t^2}{2}Q_f + \frac{\Delta t^4}{8}Q_{bf} & \Delta t Q_f + \frac{\Delta t^3}{3}Q_{bf} & -\frac{\Delta t^2}{2}\hat{C}_k^T Q_{bf} \\ -\frac{\Delta t^3}{6}Q_{bf}\hat{C}_k & -\frac{\Delta t^2}{2}Q_{bf}\hat{C}_k & \Delta t Q_{bf} \end{bmatrix}, \tag{6.10}$$

where Q_f is the accelerometer covariance and Q_{bf} is the covariance of the accelerometer bias. We can measure Q_f by logging the measurement of the accelerometer for with the quadrotor stationary for some time and measuring the covariance of that signal. The covariance of the bias can be measured using the Allan Variance, but we instead estimated it using published values for a similar accelerometer [7] and then tuned it further manually.

The measurement covariance is in our case the covariance of the Vicon position measurement, that once again can be simply computed from a log of measurements with the quadrotor stationary.

It was important to us that the state estimator be capable of handling lack of measurements from the optical tracking. We indeed predicted that dense enough obstacles courses would sometimes hide the quadrotor from the cameras. Figure 6-1 shows our state estimator attempt at estimating the quadrotor's position based solely on orientation (the output of the complementary filter) and accelerometer measurements. The gray areas show time-intervals where the state estimator was not sent any optical tracking measurements. The measurements it would have otherwise received are overlaid to show the quadrotor's "true" position.

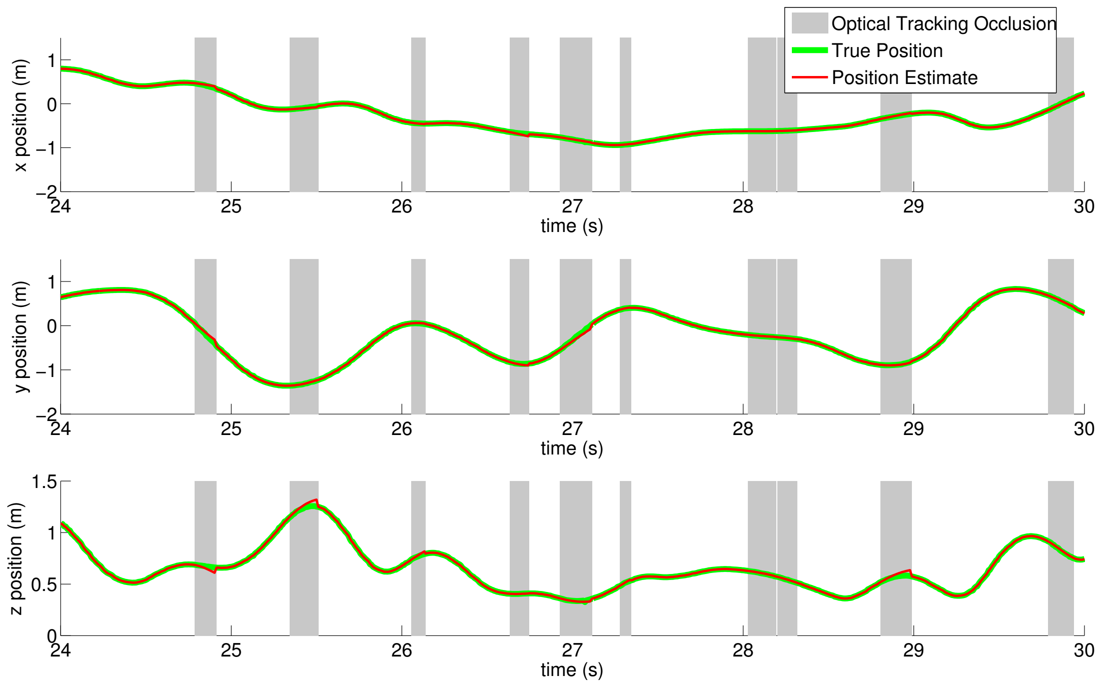


Figure 6-1: Estimated and real position of the quadrotor with simulated occlusions of the position measurements. From the plot we observe that our robustness to occlusions is much better in the x and y position than in the vertical z position, something most likely related to the complementary filter we use.

Chapter 7

Controller

7.1 On-board Attitude Controller

We first hypothesized that it would possible for the entire controller to run off-board. Even though we had a certain amount of success with that approach, we eventually opted to add an on-board attitude controller. This controller has the advantage of being able to run significantly faster because it does not suffer from the delay introduced by the radio link.

The inputs to the on-board attitude controller are the desired Euler angles ϕ_{des}, θ_{des}, ψ_{des} and a nominal squared angular velocity for the propellers ω_0^2. The attitude controller first maps the nominal squared angular velocity to duty cycles for the motors $u_{D,0}$ using the software-based speed controller described in 7.2. The on-board attitude controller then computes the error between the current and desired Euler angles e_{rpy} and angular velocities e_w and uses a linear PD controller to stabilize them.

$$u_D = -K \begin{bmatrix} e_{rpy} \\ e_w \end{bmatrix} + u_{D,0}, \tag{7.1}$$

where u_D is the input (duty cycles) actually sent to the physical hardware (the motors).

7.2 Software-Based Speed Controller

The model used for the planning defines inputs to the plant as being the square of the velocity of the propellers ω^2. This is a reasonable input model for most quadrotors, because most of them use brushless motors that are actuated through speed controllers. Speed controllers have an internal control system that takes as input the desired velocity of the motor they control. However in our case, the Crazyflie Nano Quadcopter uses DC motors and unregulated power. This means that the commands sent to the physical motors, in the form of duty cycles u_D, do not always accurately reflect the resulting velocities they produce. We get around this complication by adding feedback around the desired ω^2 for a motor using the measured battery voltage. This effectively creates an approximation for a software-based speed controller.

$$u_D = \frac{V_{max}}{V_{actual}}(\sqrt{\omega^2 - \beta}) + \alpha. \tag{7.2}$$

In this mapping, u_D is the duty cycle sent to the hardware, V_{max} is the nominal voltage of the battery, V_{actual} is the current battery voltage, α can be interpreted as the minimum duty cycle that must be sent to the hardware in order to get any angular velocity ω with the motors. Parameter β then accounts for the fact that the propellers start out at a certain non-zero velocity.

7.3 TVLQR

The mixed-integer semidefinite program provides us with a feasible trajectory in the flat output space that avoids collisions with obstacles. We can then follow the well documented steps to invert the flat output [18] and get a nominal trajectory $\mathbf{x}_0(t)$ in the full 12-DOF state-space of the quadrotor

$$\mathbf{x} := \begin{bmatrix} x \\ y \\ z \\ \phi \\ \theta \\ \psi \\ \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}, \tag{7.3}$$

where x, y, and z are the quadrotor's position and ϕ, θ, and ψ the Euler angles corresponding to its orientation. Inverting the flat outputs also gives us a desired trajectory for the square of the speed of each propellers $\omega_{0,i}^2(t)$.

We then turn our attention to the task of stabilizing these trajectories using the time-varying linear quadratic regulator [25]. First we augment the differentially flat model used by the planning algorithm to take into account the on-board attitude controller. This is done by simply adding a model of the on-board attitude controller (a system that computes propeller square angular velocities from the error between the current and desired euler angles and angular velocities of the quadrotor) and connecting the output of that system to the input of the differentially flat model. This does not create a state $\mathbf{x}$ that is different from the differentially flat model in any way but it does generate a new input and therefore a new feed-forwad term $u_0(t)$:

$$u := \begin{bmatrix} \phi_{des} & \theta_{des} & \psi_{des} & \omega_{\mathbf{x}des} & \omega_{\mathbf{y}des} & \omega_{\mathbf{z}des} & \omega_{0,des} \end{bmatrix}, \tag{7.4}$$

where $\omega_{0,des}$ is a nominal squared velocity for all the propellers. Note that this last input should be intuitive to anyone who has flown R/C aircraft because it often corresponds to the throttle stick.

We then define the error dynamics $\bar{\mathbf{x}}(t)$ and $\bar{u}(t)$.

$$\bar{\mathbf{x}}(t) = \mathbf{x}(t) - \mathbf{x}_0(t), \tag{7.5}$$

$$\bar{u}(t) = u(t) - u_0(t). \tag{7.6}$$

We can then linearize the plant dynamics $\dot{\mathbf{x}}(t) = f(\mathbf{x}, u)$ around the nominal trajectories and get the following error dynamics

$$\dot{\bar{\mathbf{x}}}(t) = A(t)\bar{\mathbf{x}}(t) + B(t)\bar{u}(t). \tag{7.7}$$

Now if we define the following quadratic cost on the tracking error

$$\begin{aligned} J(\mathbf{x}', t') = \int_0^\infty \bar{\mathbf{x}}^T(t)Q\bar{\mathbf{x}}(t) + \bar{u}^T(t)R\bar{u}(t)dt, \\ Q = Q^T \geq 0, R = R^T > 0, \mathbf{x}(t) = \mathbf{x}'(t). \end{aligned} \tag{7.8}$$

It can then be shown that the optimal cost-to-go J^* has the form

$$J^*(\bar{\mathbf{x}}, t) = \bar{\mathbf{x}}^T S(t)\bar{\mathbf{x}}, S(t) = S^T(t) \geq 0, \tag{7.9}$$

where $S(t)$ is the solution to the differential algebraic Riccati equation:

$$-\dot{S} = Q - SBR^{-1}B^TS + SA + A^TS. \tag{7.10}$$

Although the boundary condition on $S(t)$, i.e. the value of $S(t_f)$ where t_f is the time where the plant reaches the goal state, can be chosen along with the costs Q and R to give the best tracking, or it can be made the same as the infinite-horizon cost if the trajectory ends on a fixed point we want to stabilize.

The result is a time-varying controller $K(t)$ that takes the full state of the quadrotor as input and returns a control input to be sent to the on-board attitude controller.

Chapter 8

Results

In order to test our planning and control approach, we designed a series of obstacle courses of increasing difficulty. First, a simple "forest" simulation, mimicking a series of vertical obstacles. Then we experiment with a few vertical obstacles contained between two "walls" that the quadrotor has to either go over or under. This is particularly challenging for our controller because it makes it easy for the quadrotor to enter a vortex ring state as it travels downward and comes to a zero airspeed in order to accelerate back up. The third course starts to increase the number of obstacles with a total of 11 vertical and angled poles. Finally, an environment made of 6 poles and 20 strings placed in various orientations is used to demonstrate the scalability of our approach.

8.1 Modeling Obstacle Fields

Each obstacle field was built with either PVC pipes, plastic fencing or cotton strings. We then measured the dimensions of each one of them and entered them in a file using the Unified Robot Description Format (URDF). The URDF, along with Drake, allows us to seamlessly extract convex hull representations of the resulting obstacle fields.

8.2 Computing Trajectories and Feedback Controllers

We then inflate each obstacle by moving the faces of the convex hulls by an amount equal to the radius of the quadrotor in the direction of their normal. This then allows us to treat the quadrotor as being essentially dimensionless. In practice, we found that this approach performed slightly better than the alternate one that requires the polynomial trajectories to be a certain distance away from the planes of the safe regions returned by IRIS. We then run IRIS over the obstacle field, also specifying a bounding box limiting the space segmented by IRIS. This helps the mixed-integer program run faster, but most importantly it makes sure that the quadrotor does not simply fly around the obstacle fields. We can then run the mixed-integer semidefinite program described in section 5.2, invert the flat outputs and compute a time-varying LQR controller using Drake.

Computing trajectories for the four environments we experimented with usually converged within a few minutes (less than 10) in agreement with the performance shown in [4]. However the trajectory in the environment with the strings was built by concatenating shorter trajectories that took more time to converge, and we simply stopped the optimization after 20 minutes. By then the planner had found feasible trajectories that satisfied all the constraints but the cost hadn't converged within the same tolerance as with the other environments.

8.3 Controller Tuning

8.3.1 TVLQR Cost Function

Even though we used a model-based approach, TVLQR still requires us to tune gains through the quadratic cost functions Q and R. We tune those with a greedy approach. First we set the R matrix to a trivial cost (the identity) and then make the cost associated with the other states very small. We then run a trajectory and evaluate how well the quadrotor tracks it. Then one by one we increase the gains for position, orientation, and their derivatives. Once we have increased each entry of the diagonal of Q, we fine tune it further by trial and error and general intuition about the system.

8.3.2 Trajectory Speed

As described in chapter 5, our planning algorithm does not tie the resulting trajectory to a specific velocity. We therefore tune the velocity of the trajectory by starting with a very comfortable (making its execution time very long). We then slowly increase the speed of the trajectory until the system stops tracking well enough or is simply unable to follow it. We believe that a more analytical approach to this tuning could benefit us greatly and suggest it as further work in section 9.1.

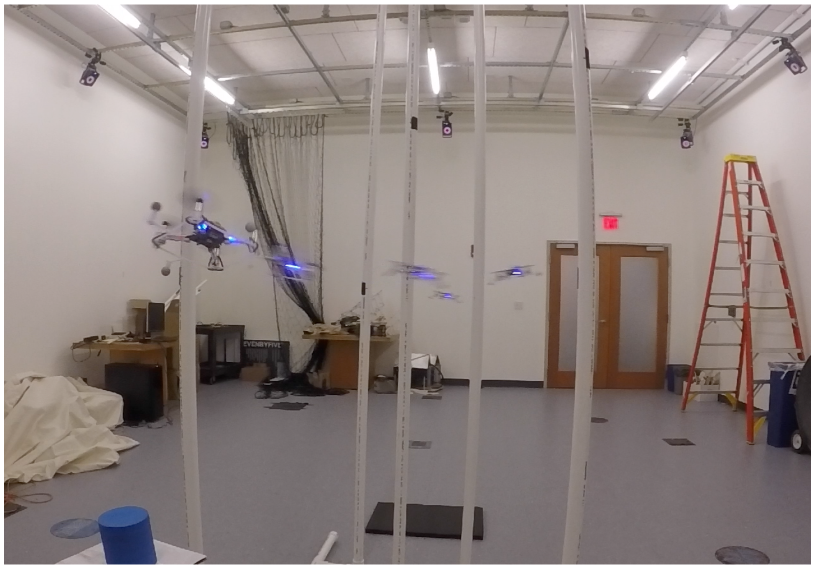
8.4 Experimental Results

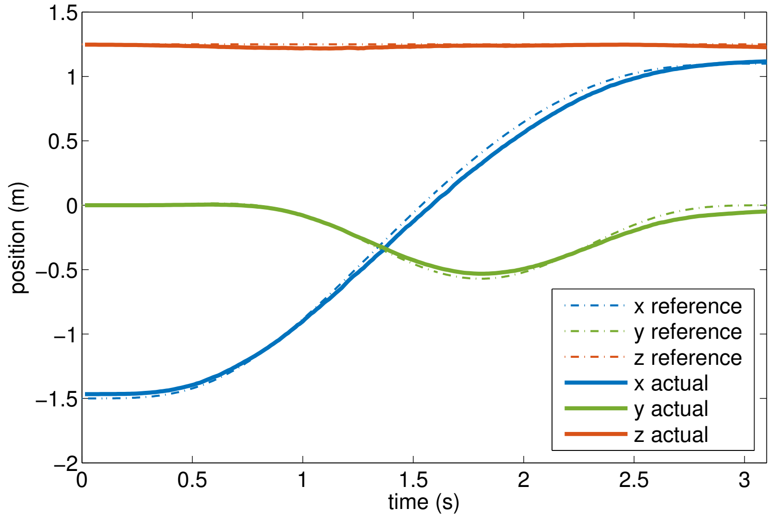
Figures 8-1 to 8-8 show the results of running the controllers in the four environments. A picture of each environment with the quadrotor flying through them is shown, along with the resulting tracking of position over time. Then a 3D visualizer is used to show the overlaid plan and resulting trajectories. Finally we plot the error of the tracking for each environment.

8.5 Discussion

As it can be seen from the plots, IRIS, MISDPs and model-based control approaches are indeed capable of flying a small quadrotor like the Crazyflie through environments that are densely populated with obstacles, confirming our hypothesis. The number of obstacles we used ranged from 5 to 26, all contained within a cubic meter volume or so. The plots also show how our controller was capable of tracking the trajectories with errors that did not deviate from the planned trajectories by more than 10cm or so, and that were lower than that on average.

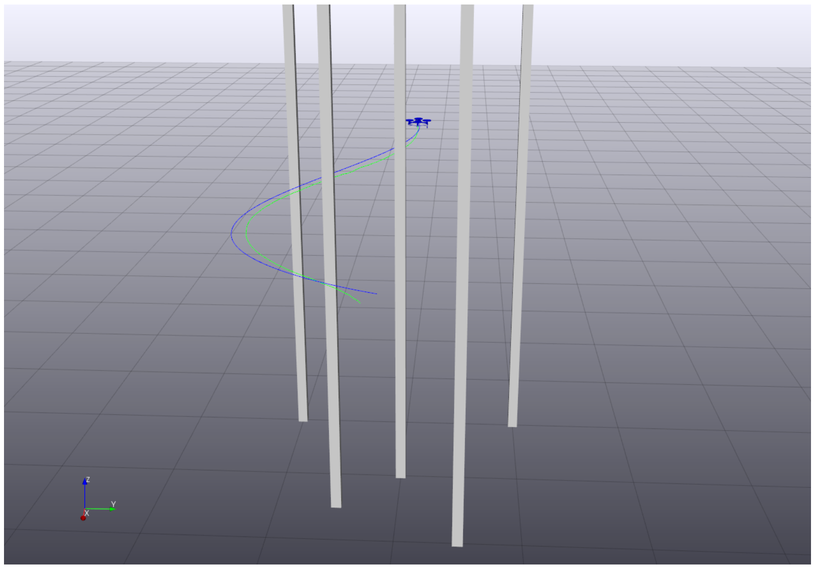
We found that the limitation of the planning algorithm lies in the number of obstacle-free regions that can be handled efficiently by the mixed-integer program,


(a) Picture of the "forest" obstacle course.

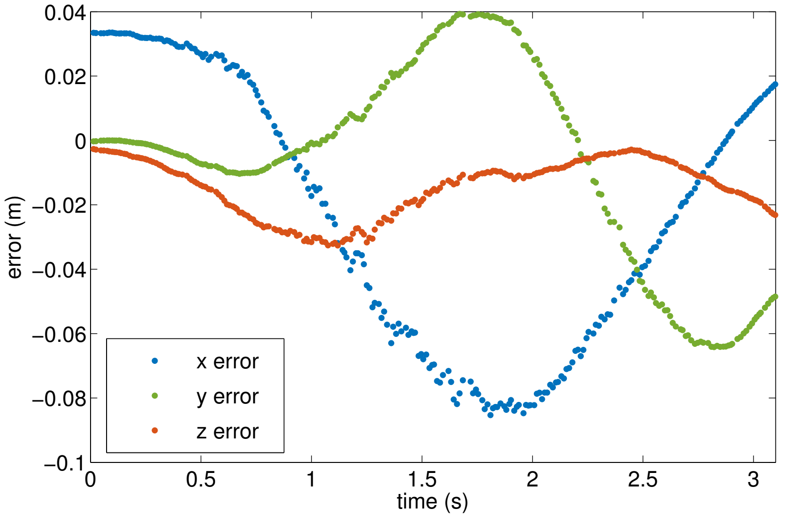


(b) Position of the quadrotor over time.

Figure 8-1: The "forest" obstacle course: the trajectory planned by our planning algorithm and the resulting trajectory described by the quadrotor executing it using our controller. We see that the tracking gets worse when doing the more dynamic maneuvers.

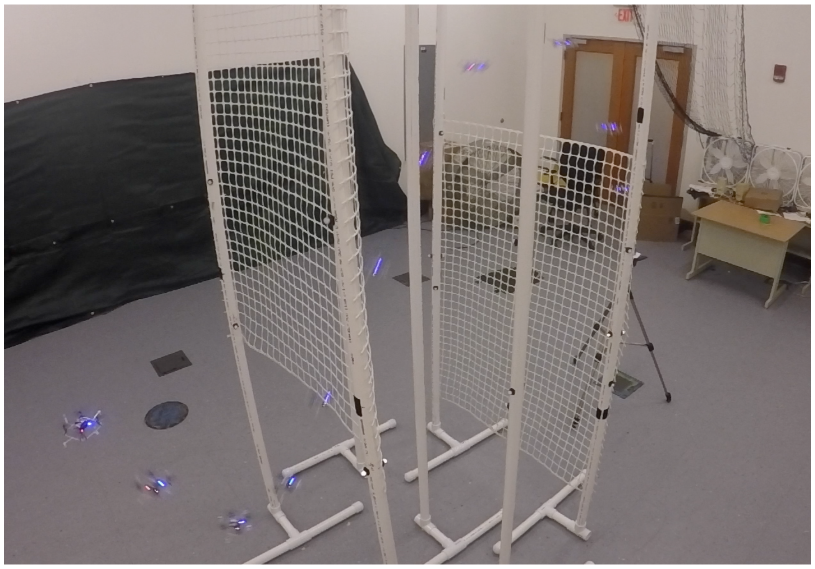


(a) Planned trajectory in blue and trajectory followed by the quadrotor in green.

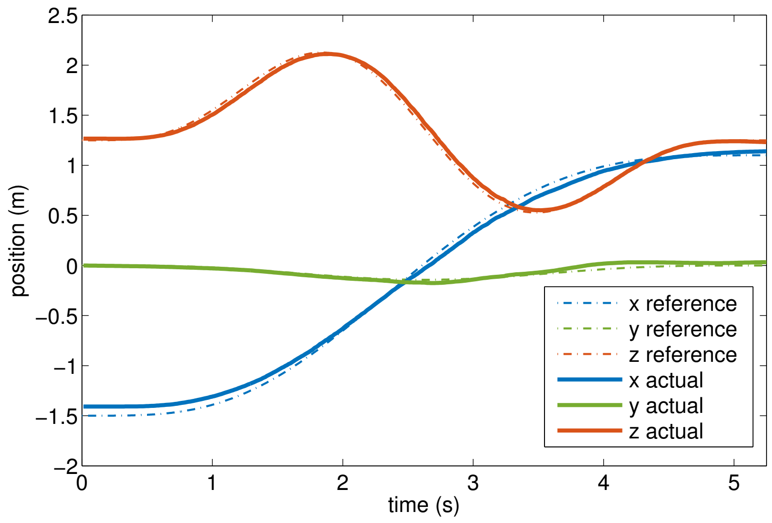


(b) Error of the tracking over time.

Figure 8-2: The "forest" obstacle course: Showing here the 3D visualization of the trajectory as well as the error of the position over time.

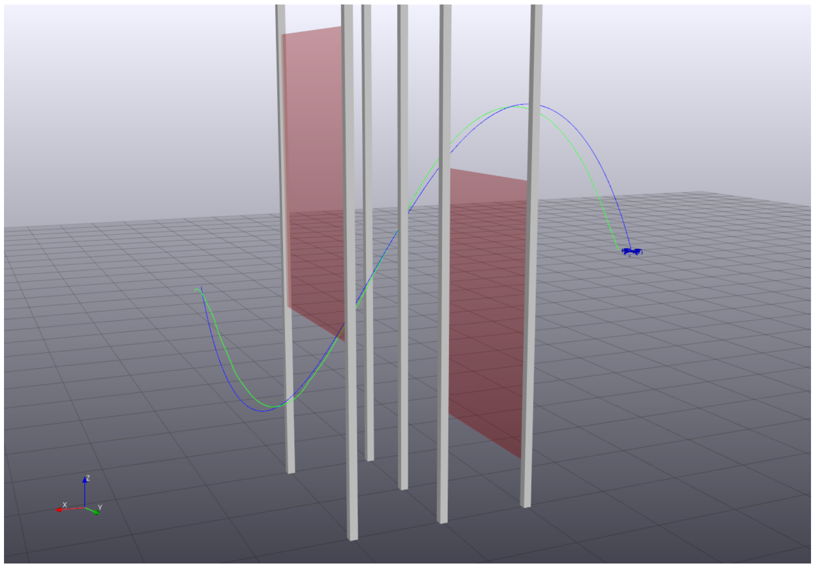


(a) The obstacle course with "walls".

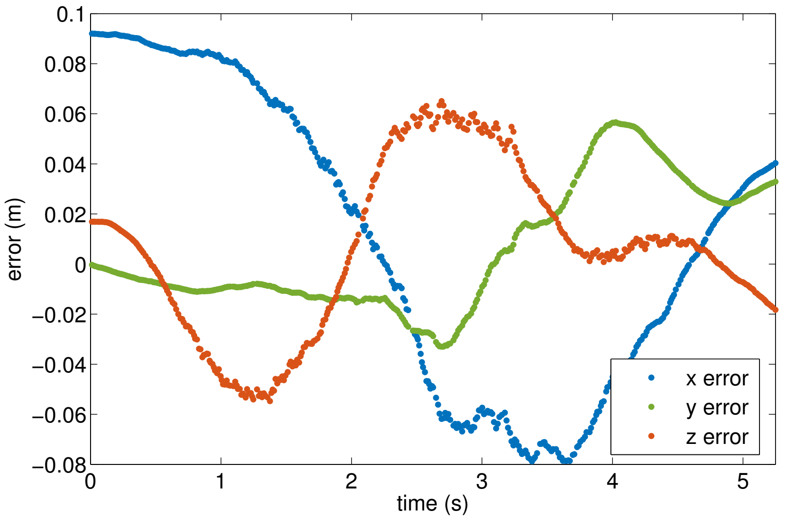


(b) Position of the quadrotor over time.

Figure 8-3: The obstacle course with "walls": the trajectory planned by our planning algorithm and the resulting trajectory described by the quadrotor executing it using our controller. Once again the rapid turns are much harder to track than the more linear motions.

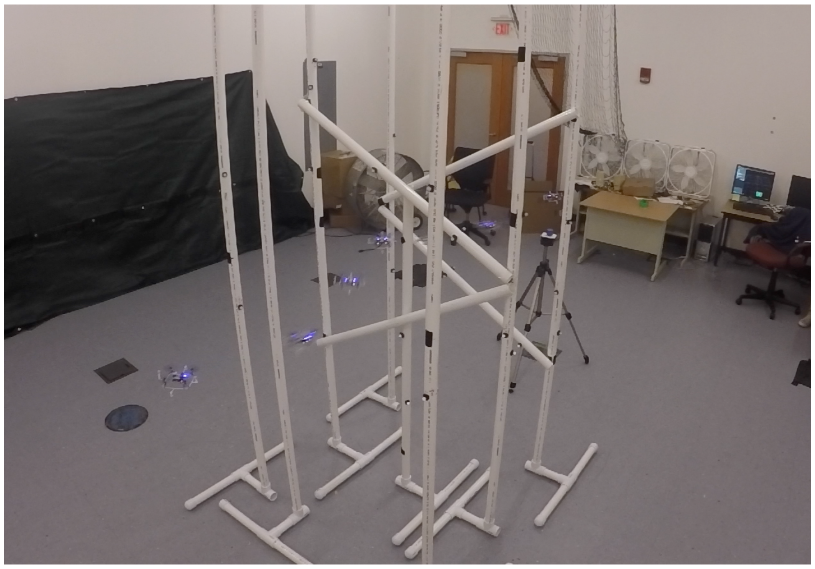


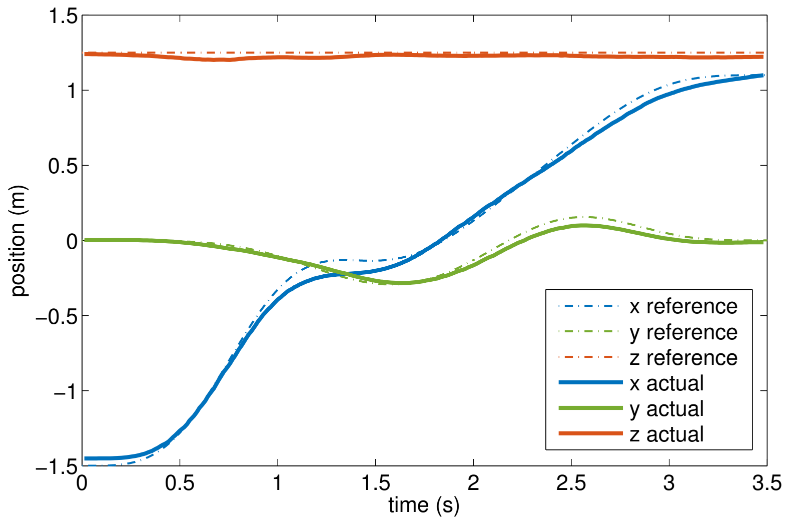
(a) Planned trajectory in blue and trajectory followed by the quadrotor in green.



(b) Error of the tracking over time.

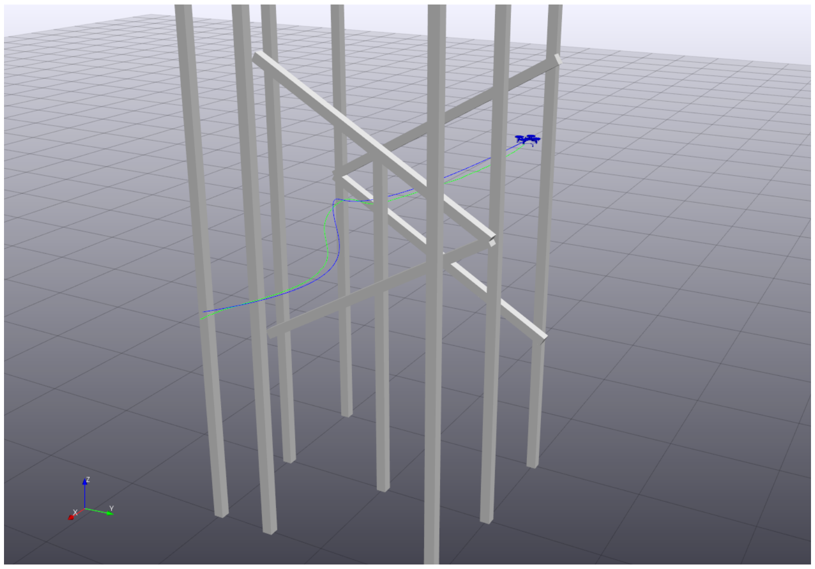
Figure 8-4: The obstacle course with “walls”: the trajectory planned by our planning algorithm and the resulting trajectory described by the quadrotor executing it using our controller. Notice the convergence to the planned trajectory even though there was a large error in the initial conditions.


(a) The obstacle course with 11 pipes.

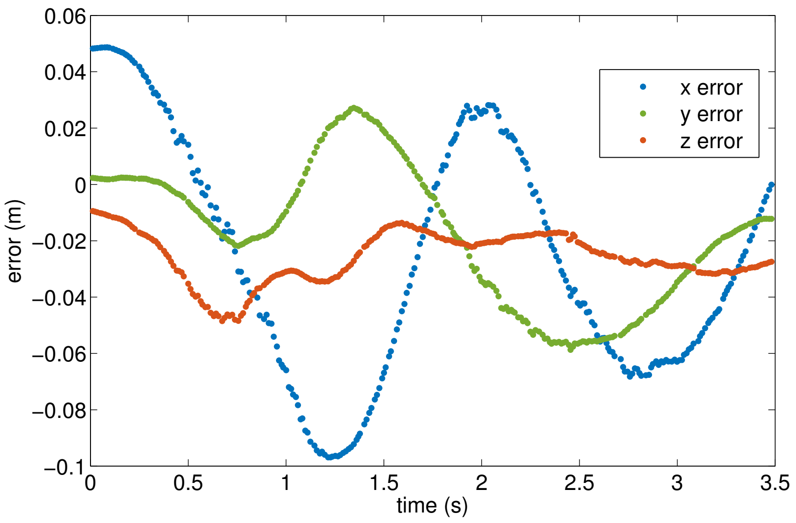


(b) Position of the quadrotor over time.

Figure 8-5: The obstacle course with 11 pipes: the trajectory planned by our planning algorithm and the resulting trajectory described by the quadrotor executing it using our controller. The first turn was extremely dynamic and difficult to track, but the tracking converged later.

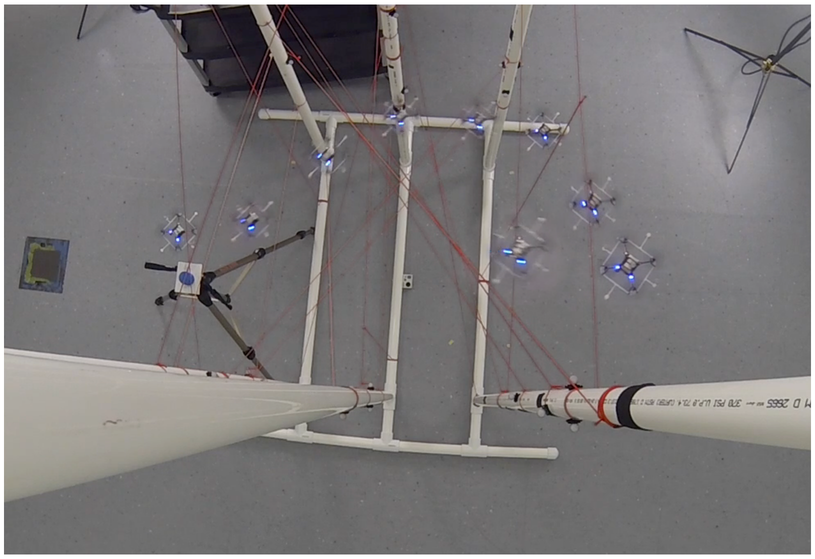


(a) Planned trajectory in blue and trajectory followed by the quadrotor in green.

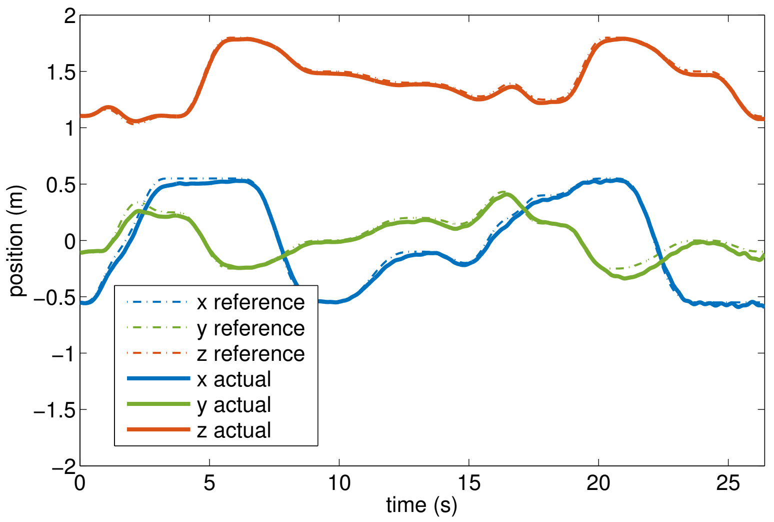


(b) Error of the tracking over time.

Figure 8-6: The obstacle course with 11 pipes: the trajectory planned by our planning algorithm and the resulting trajectory described by the quadrotor executing it using our controller.

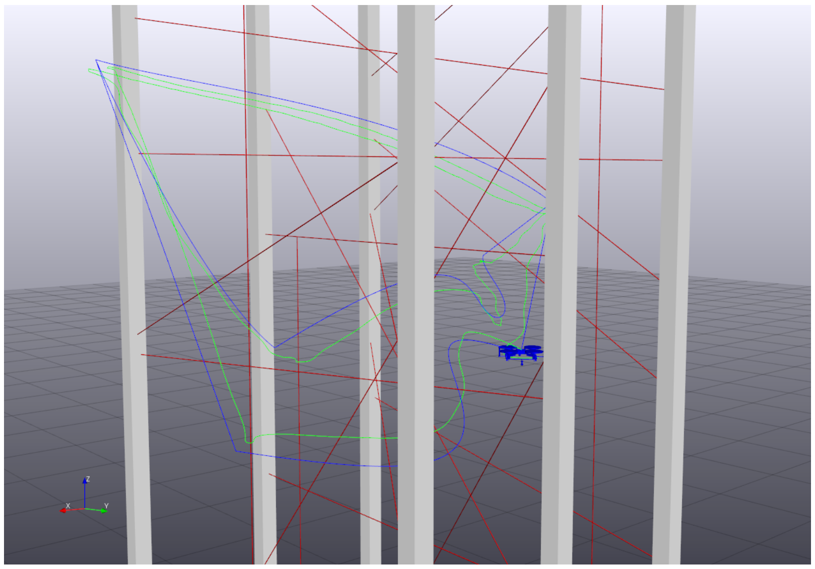


(a) The beginning of the obstacle course with strings

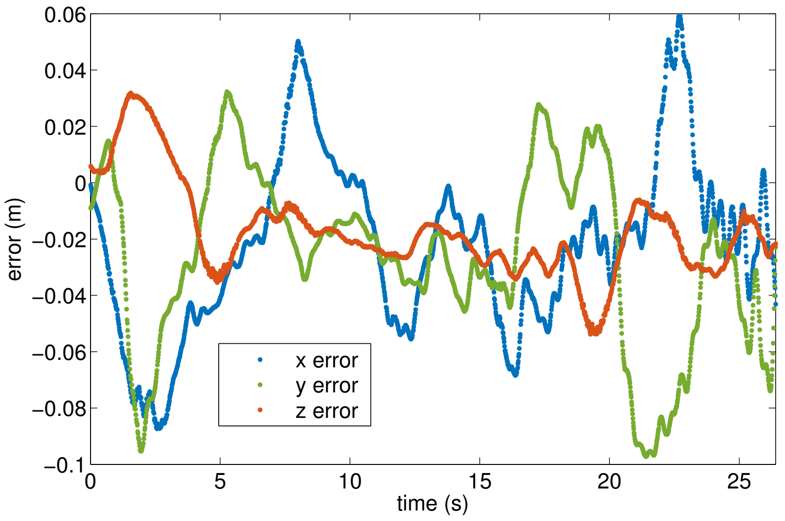


(b) Position of the quadrotor over time.

Figure 8-7: The obstacle course with strings: the trajectory planned by our planning algorithm and the resulting trajectory described by the quadrotor executing it using our controller. The first figure only shows the beginning of the trajectory.



(a) Planned trajectory in blue and trajectory followed by the quadrotor in green.



(b) Error of the tracking over time.

Figure 8-8: The obstacle course with strings: the full trajectory was made by concatenating shorter trajectories.

although this limitation only appeared once we reached a significant number of obstacles like with the environment containing the strings. In this case, we also found that using the feasible but not necessarily optimal trajectory returned by the solver was highly sufficient to complete the task.

We also unsurprisingly found that the quality of the tracking depended on the speed at which we were planning it. This turned out to be more of a challenge than expected, because there was no analytical way to compute a maximal velocity for a given trajectory and controller.

We also found that TVLQR performed reasonably well for this task. However the results give us good reasons to believe that time might not be the best dimension to parameterize over. We believe that better performance could be produced by transverse LQR, where the parameter would be the tangential velocity of the tracked trajectory instead of time. This controller would be closer in spirit to what was demonstrated in [20].

We also initially attempted to run the trajectories without the on-board attitude controller (producing a completely off-board controller). Even though we were able to track trajectories with a certain level of success with this approach, we found that the on-board controller added a significant amount of stability to the system. Indeed errors in roll and pitch quickly take the quadrotor into unstable states if the control loop is not running quickly enough (at least around 100Hz in the case of the Crazyflie).

Chapter 9

Conclusions and Future Work

9.1 Future Work

We found that the use of differential flatness, although numerically convenient, failed to capture some of the limits of the system. For example, there was no way to include the actuator limits in the planning stage, leaving us having to set the speed of the trajectory mostly by trial and error. We have identified this as a potential area for future work, and believe that parametric funnels [17] might provide the proper framework for that improvement.

Finally, as the speed of the trajectories are increased, the quality of the tracking given by the feedback controller inevitably went down. We believe that transverse LQR could be a suitable alternative to time-varying LQR and allow us to push the limits of our platform further. Finally, IRIS could also be used in various reactive planning schemes that could be added to the demonstrated control system.

9.2 Conclusion

We demonstrated experimental validation of the algorithm originally proposed by Deits to generate collision-free trajectories in obstacle-dense environments. This confirmed our hypothesis that it would be possible to fly a small quadrotor like the Crazyflie through environments containing significantly more obstacles than demonstrated in the past using IRIS, MISDPs and model-based control. The most cluttered environment we flew the quadrotor through contained 26 obstacles in a cubic meter volume and the quadrotor was able to follow a collision-free trajectory through it within 10cm of precision.

Bibliography

[1] About bitcraze. Accessed: 2014-11-05.

[2] Andrew James Barry. *Flying between obstacles with an autonomous knife-edge maneuver*. PhD thesis, Massachusetts Institute of Technology, 2012.

[3] Michael Bloesch, Marco Hutter, Mark A Hoepflinger, Stefan Leutenegger, Christian Gehring, C David Remy, and Roland Siegwart. State estimation for legged robots-consistent fusion of leg kinematics and imu. *Robotics*, page 17, 2013.

[4] Robin Deits and Russ Tedrake. Efficient mixed-integer planning for uavs in cluttered environments. 2014.

[5] Robin Deits and Russ Tedrake. Footstep planning on uneven terrain with mixed-integer convex optimization. Technical report, DTIC Document, 2014.

[6] Robin LH Deits. *Convex Segmentation and Mixed-Integer Footstep Planning for a Walking Robot*. PhD thesis, Massachusetts Institute of Technology, 2014.

[7] Naser El-Sheimy, Haiying Hou, and Xiaoji Niu. Analysis and modeling of inertial sensors using allan variance. *Instrumentation and Measurement, IEEE Transactions on*, 57(1):140–149, 2008.

[8] M Fliess, J Levine, P Martin, and P Rouchon. On differentially flat nonlinearsystems. In *Symposium on Nonlinear Control System Design, Bordeaux, France*, pages 159–163, 1992.

[9] Urban Forssell. Closed-loop identification: Methods, theory, and applications. 1999.

[10] Ivar Gustavsson, Lennart Ljung, and Torsten Söderström. Identification of processes in closed loopâĂŤidentifiability and accuracy aspects. *Automatica*, 13(1):59–75, 1977.

[11] Håkan Hjalmarsson, Michel Gevers, and Franky De Bruyne. For model-based control design, closed-loop identification gives better performance. *Automatica*, 32(12):1659–1673, 1996.

[12] Gabriel M Hoffmann, Steven L Waslander, and Claire J Tomlin. Quadrotor helicopter trajectory tracking control. In *AIAA Guidance, Navigation and Control Conference and Exhibit*, pages 1–14, 2008.

[13] InvenSense. *MPU 9250 Product Specification Revision 1.0.* Rev. 1.

[14] Alex Kushleyev, Daniel Mellinger, Caitlin Powers, and Vijay Kumar. Towards a swarm of agile micro quadrotors. *Autonomous Robots*, 35(4):287–300, 2013.

[15] Sebastian OH Madgwick. An efficient orientation filter for inertial and inertial/magnetic sensor arrays. *Report x-io and University of Bristol (UK)*, 2010.

[16] Robert Mahony, Tarek Hamel, and Jean-Michel Pflimlin. Nonlinear complementary filters on the special orthogonal group. *Automatic Control, IEEE Transactions on*, 53(5):1203–1218, 2008.

[17] Anirudha Majumdar, Mark Tobenkin, and Russ Tedrake. Algebraic verification for parameterized motion planning libraries. In *American Control Conference (ACC), 2012*, pages 250–257. IEEE, 2012.

[18] Daniel Mellinger and Vijay Kumar. Minimum snap trajectory generation and control for quadrotors. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 2520–2525. IEEE, 2011.

[19] Daniel Mellinger, Aleksandr Kushleyev, and Vijay Kumar. Mixed-integer quadratic program trajectory generation for heterogeneous quadrotor teams. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 477–483. IEEE, 2012.

[20] Daniel Mellinger, Nathan Michael, and Vijay Kumar. Trajectory generation and control for precise aggressive maneuvers with quadrotors. *The International Journal of Robotics Research*, page 0278364911434236, 2012.

[21] Mosek ApS. The MOSEK optimization software, 2014.

[22] Nordic Semiconductor. *ARM Cortex-M4 32b MCU+FPU, 210DMIPS, up to 1MB Flash/192+4KB RAM, USB OTG HS/FS, Ethernet, 17 TIMs, 3 ADCs, 15 comm. interfaces and camera*, June 2013. Rev. 4.

[23] Nordic Semiconductor. *ARM-based 32-bit MCU, 256 KB Flash, CAN, 12 timers, ADC, DAC and comm. interfaces, 1.8 V*, November 2014. Rev. 2.

[24] Charles Richter, Adam Bry, and Nicholas Roy. Polynomial trajectory planning for quadrotor flight. In *International Conference on Robotics and Automation*, 2013.

[25] Russ Tedrake. Lqr-trees: Feedback motion planning on sparse randomized trees. 2009.

[26] Russ Tedrake. Drake: A planning, control, and analysis toolbox for nonlinear dynamical systems, 2014.

[27] M Van Nieuwstadt, M Rathinam, and RM Murray. Differential flatness and absolute equivalence of nonlinear control systems. *SIAM Journal on Control and Optimization*, 36(4):1225–1239, 1998.